Developing and Assessing New List Operators in App Inventor

Soojin Kim

May 2015

Acknowledgments

This work was supported by Wellesley College Faculty Grants and by the National Science Foundation under grant DUE-1226216.

I would like to thank my adviser Franklyn Turbak for his incredible support. I feel very lucky to have had such an enthusiastic adviser. This project would not have been possible without his guidance.

Thank you to my family and friends for encouraging me in everything I do and for always making me laugh. Their unconditional support means a lot to me.

Thank you to Brian Tjaden, Benjamin Wood and Ann Velenchik for being a part of my thesis committee and providing valuable feedback.

Thank you to all the students who participated in my user study and suggested improvements to my project. Your help is greatly appreciated.

Finally, thank you to all my professors at Wellesley who have helped me grow both academically and personally during my time here. I am grateful to have had such dedicated professors, and I know that what I have learned from them will help me beyond Wellesley.

Abstract

MIT App Inventor is a blocks-based programming environment in which users connect puzzle-shaped blocks to build apps for Android devices. App Inventor supports a Python-like list data structure typically manipulated with loops, but it lacks important list operations that can greatly simplify list manipulation.

I have extended the implementation of App Inventor to include new blocks that map, filter, reduce, and sort lists. Since App Inventor does not have first class functions, many of these blocks incorporate the parameter and body declarations of functional arguments associated with these operators. For example, the mapping block has a socket for the input list, a parameter declaration to name each item in the list, and a socket for the expression that denotes the result of transforming the item. There are three sorting blocks: one using a default comparator that arranges the list items in increasing order; one with a key function that returns a proxy value used for sorting the list with the default comparator; and one that uses a comparator provided by the user.

In addition, I also addressed the problem that some App Inventor list operators are destructive (they change their input list) while others are nondestructive (they return a new list). This inconsistency can be limiting and confusing to users. To solve this problem, I have implemented a mechanism for many list operators that allows users to choose between making the operator destructive or nondestructive. This flexibility eliminates complicated programming workarounds necessary in the current system.

To test the usability of the map, filter, reduce and sort operators, I conducted a user study with 18 students at Wellesley College who had previous experience working with App Inventor. Most users, even those with no previous exposure to map, filter or reduce, were able to successfully complete a majority of the tasks. Two common problems that users faced, however, was manipulating a list of lists using map, filter and/or reduce and sorting a list with two keys. In future work, the results and feedback from this study will be used to iterate through the design of these new blocks.

Contents

Chapter 1

Introduction

1.1 Overview of App Inventor

MIT App Inventor [Ai2] is a blocks-based programming environment where users connect puzzle-shaped blocks together to build Android apps. This program aims to lower barriers to learning coding and making apps in several ways. First, the blocks' shapes, including their plugs and sockets, help users determine how the blocks can be fit together to write a syntactically valid program. Second, the number of sockets indicates the number of arguments, and the label on each socket suggests the type of values appropriate for that socket. Third, users can drag and drop blocks from a menu rather than having to remember the names of each block. Finally, App Inventor is a visual program that lacks textual markers such as parenthesis, braces and colons, reducing the number of ways to make errors.

App Inventor consists of two main parts: the Designer and the Blocks Editor. In the Designer, the user chooses the components that will appear in the interface of the app and drags them onto the Screen. Each component has specific properties that the user can adjust. For instance, as seen in Figure 1-1, a Canvas has a background image, background color, font, width, height and more.

In the Blocks Editor, blocks are shaped like puzzle pieces and categorized into drawers based on their functionality. In order to specify the behavior of her app, the user drags and drops blocks from these drawers into the workspace and connects them together. Figure 1-2 shows a simple program that includes variables, event handlers, expressions and statements. This program initializes two global variables: a number called `randomInteger` and a list called `listofRandomIntegers`. When the user presses the button on her app, she triggers an event handler, prompting a random integer

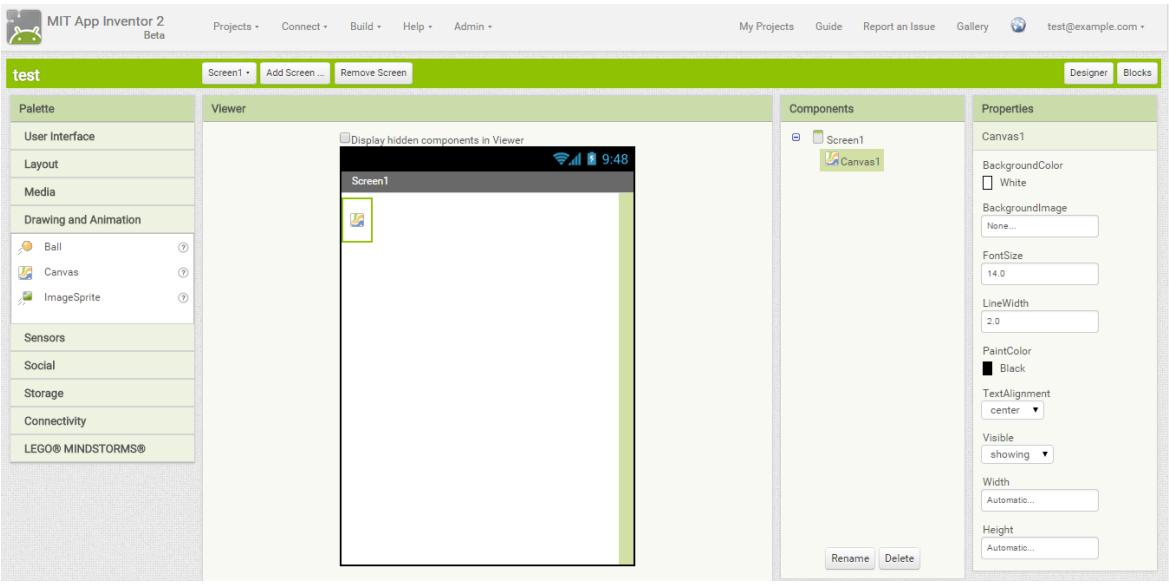

Figure 1-1: A Designer window with a canvas component

in between 1 and 100 to be displayed and added to the end of `listofRandomIntegers`. The `set randomInteger to` block is a statement that performs an action and does not return anything. It thus has a notch on the upper left corner. On the other hand, the `random integer from 1 to 100` block is an expression that evaluates to and returns a value, as indicated by the plug on the left hand side of the block.

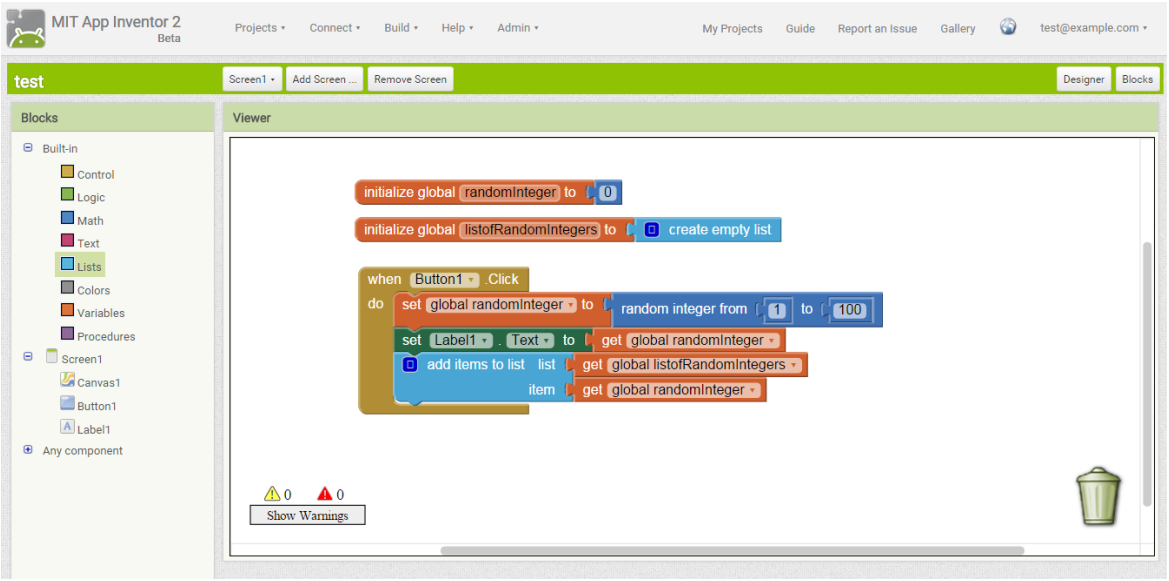

Figure 1-2: A program in the Blockly Editor containing variables, event handlers, expressions and statements

While building her app on the computer, the user can live test her app on an Android device. The device is connected to the computer through either Wifi connection or a USB cable. As the user continues to build her app on the computer, the changes she makes are reflected on the device. As seen in Figure 1-3, the user can also right click on an expression block and run **Do It**, which executes the block and displays the value returned by that block in a bubble. **Do It** results displayed in bubbles are used frequently in the remainder of this paper and in the user study.

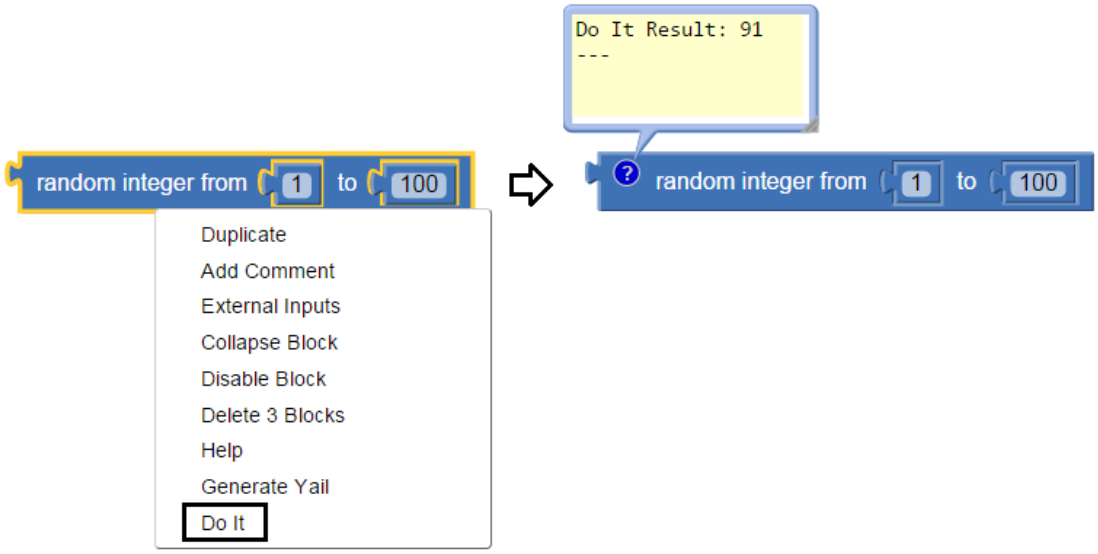

Figure 1-3: Executing **Do It** to return the value of the block

1.2 Lists

Lists are one way to organize data in App Inventor and resemble Python lists and JavaScript arrays. App Inventor lists are characterized by three key properties:

1. They are heterogeneous, which means they can contain items of different types.
2. They have indexed mutable slots, so list slots can be accessed and changed through their indexes. App Inventor indexes, however, begin at 1, not 0 like Python lists and JavaScript arrays.
3. They are resizable; the number of slots in the list can change over time.

Figure 1-4 displays the list operators that App Inventor supports. In the current App Inventor, loops are used for list processing involving mapping, filtering or reducing. I have developed an

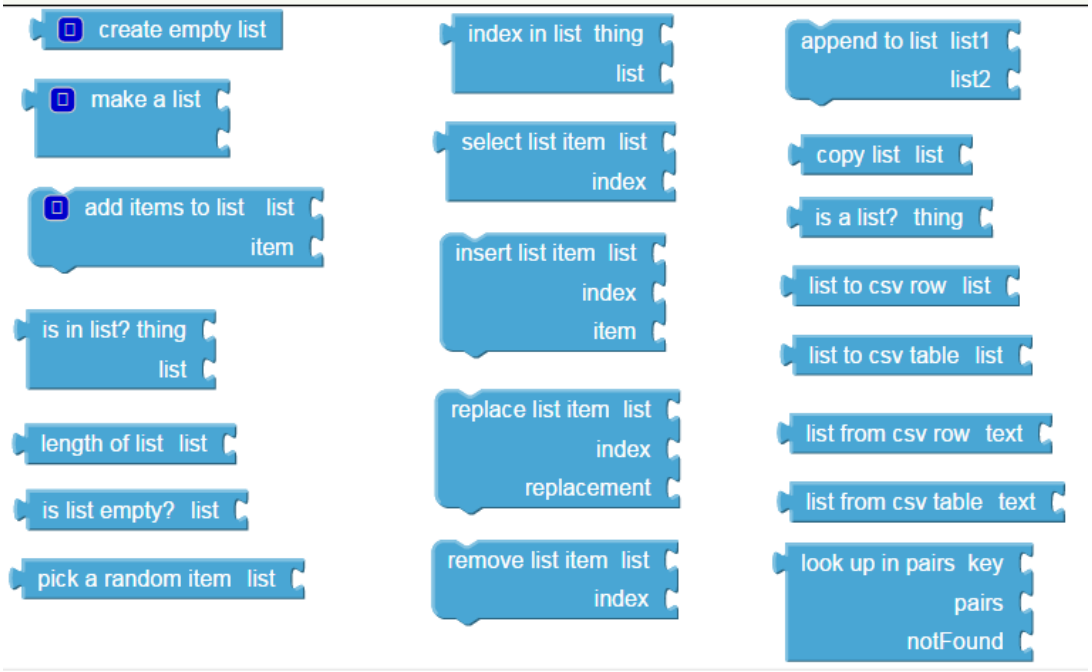

Figure 1-4: Current list operators

example to show how a user could perform these operations with the existing App Inventor list operators. Consider Berry, who is setting up a lemonade stand where she charges $2 per lemonade. As shown below, she keeps a simple record of the number of lemonades she sold on each date. She planned on doing this for seven consecutive days but was not able to set up her stand on some days. For those days, she wrote down N/A for the number of lemonades sold.

Date	Number of Lemonades Sold	Daily Profit
6/1/14	13	
6/2/14	20	
6/3/14	N/A	
6/4/14	18	
6/5/14	N/A	
6/6/14	10	
6/7/14	16	
	Total Profit	?

Table 1.1: Berry's record of number of lemonades sold on each date

Berry is interested in finding her total profit. Thus, she needs to multiply all the number entries in the number of lemonades sold column by 2 to get a list of daily profits and then sum up all the daily profit entries to calculate her total profit.

Figure 1-5 shows one way to find Berry's total profit in the current App Inventor using loops. This procedure has two local variable declarations: a list called `listofDailyProfits` and a number called

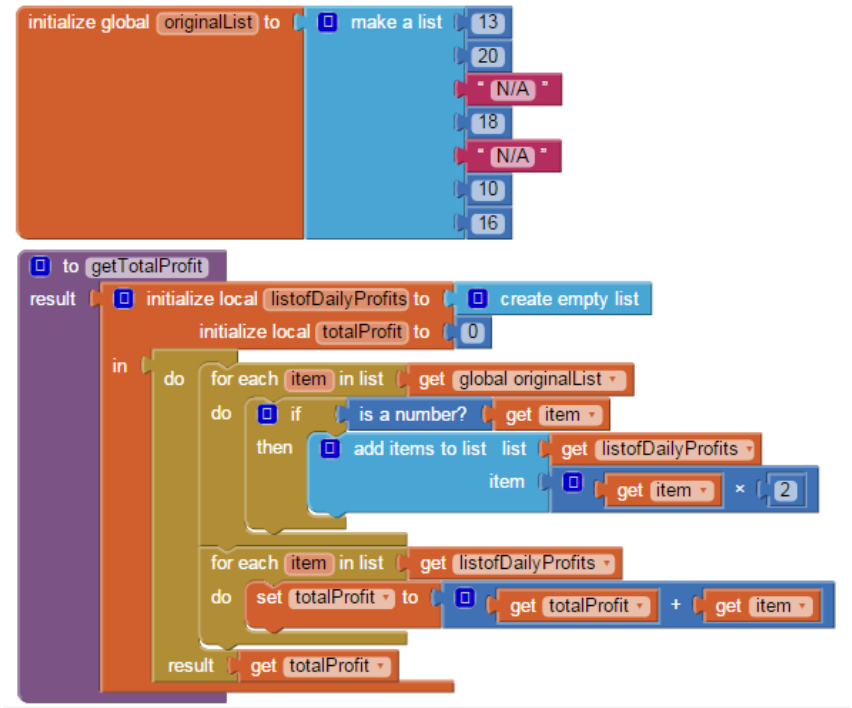

Figure 1-5: Calculating Berry's total profit using loops that map, filter and reduce over the input list

`totalProfit`. The first loop uses a conditional statement to check if each item in the `originalList` is a number. If so, the item is multiplied by 2, the price of each lemonade, and added to the `listofDailyProfits`. Otherwise, the item is discarded because it equals "N/A." The first loop performs both a filter and a map. The second loop reduces the `listofDailyProfits` by using the variable `totalProfit` to keep a running sum of Berry's profits while iterating through the list. Using loops to map, filter and reduce a list involves variable declarations, a conditional statement, and an `add items to list` operator. As more operations are added, the procedure becomes increasingly complicated.

1.2.1 Map, Filter, Reduce and Sort

App Inventor currently lacks important list operators such as map, filter, reduce and sort that exist in many functional programming languages. Therefore, users typically use loops in order to iterate through a list. Defining one's own loops can be a tedious and error-prone process. Mapping, filtering or reducing a list involves correctly initializing and accumulating a new list as an answer. App Inventor also does not support any sorting operation, which means that users must use loops to implement their own sorting algorithm or find programming workarounds, which will be described

in Chapter 3.

To address this problem, I have added map, filter, reduce and three sort blocks that can greatly simplify list manipulation. In text-based programming languages, map, filter, reduce and sort are higher-order list operators that take in a function as one of their inputs. For example, map takes in an input list and a function, and returns a new list that is the result of applying the function to each element in the input list. Below is one use of map in Python to create a new list in which each element of the original list is doubled.

```
>>>map(lambda item: item*2, [5,3,8,11,2])
[10,6,16,22,4]
```

However, App Inventor does not have first class functions (the ability to pass in functions as parameters). Therefore, many of my new blocks have built into them the parameter declarations of the functional arguments. For example, as seen in Figure 1-6, the mapping block has a socket for the input list, a parameter declaration to name each item in the list, and a socket for the expression that denotes the result of transforming the item. Rather than taking in the function (lambda item: item * 2), the map block has built into it the parameter item and only takes in (item * 2), the body expression of the function.

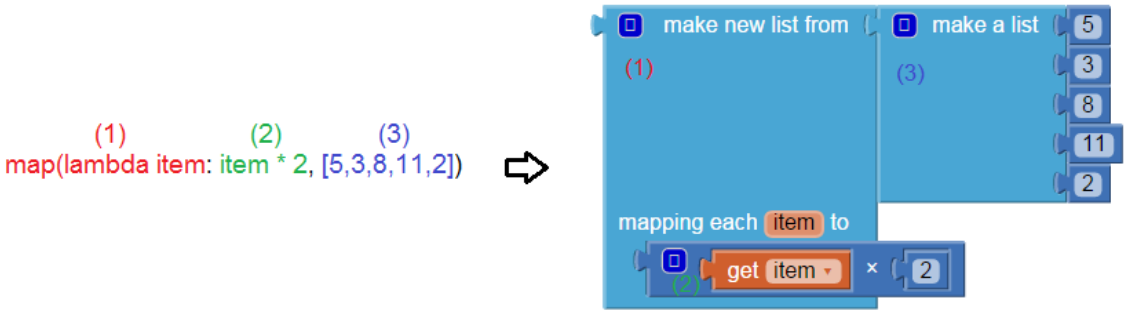

Figure 1-6: The map block takes in an input list and the body expression of the functional argument.

Revisiting Berry's example from above, Berry can now find her total profit using the filter, map and reduce blocks as shown in Figure 1-7. She no longer needs to use a conditional statement to check that the list items are indeed numbers, or declare a variable to keep a running sum of her profits. The replacement of loops with these higher-order operators greatly simplifies list processing for users.

In addition to the map, filter and reduce blocks, I added three types of sort blocks. The basic

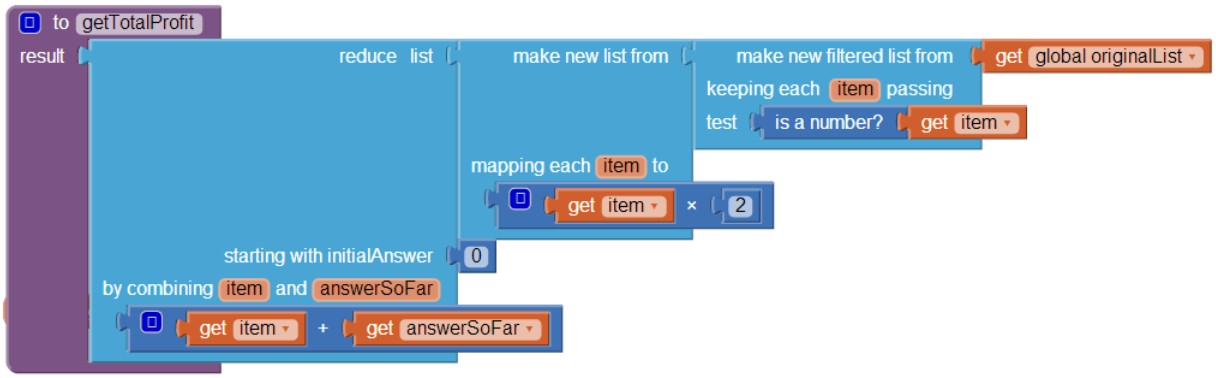

Figure 1-7: Calculating Berry's total profit using filter, map and reduce operators

sort block uses a default comparator and returns a new list whose elements are those of the input list sorted in increasing order. In Figure 1-8, the basic sort is provided a list of names as its input. Since it uses a default comparator that is alphabetical on strings, it returns a new list of names sorted in alphabetical order. The **Do It** notation for lists is similar to Scheme in that items in a list are separated by spaces and enclosed in parenthesis.

Figure 1-8: Basic sort block that sorts the input list alphabetically

Another sort block has a key function that returns proxy values used by the default comparator to sort the input list in increasing order. In Figure 1-9, using the same list of names as the input, I defined a key function that evaluates to the length of each item in the list. This sort with key therefore returns a new list sorted in increasing order by the length of student names. This algorithm

is stable; that is, items with equal key values maintain their relative positions in the original list. Zoe and Sam, for example, both have a length of 3, but since Zoe precedes Sam in the input list, she also precedes Sam in the output list.

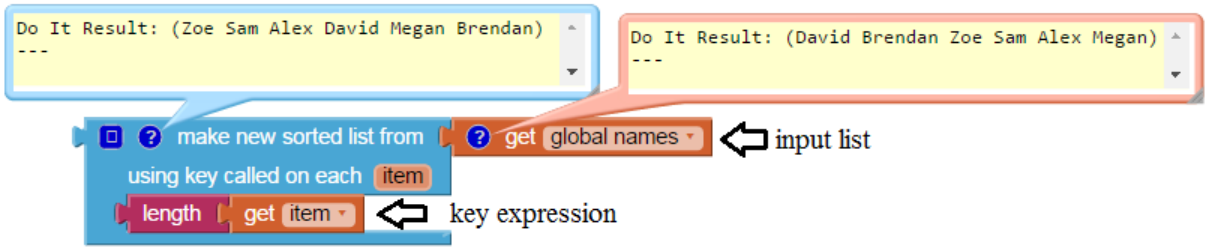

Figure 1-9: Sort with key block that sorts the input list in increasing order by the length of student names

The sort with comparator sorts the input list according to a comparator provided by the user. In Figure 1-10, I defined a comparator in which `item1` is greater than `item2`, so this sort with comparator returns a new list of names sorted in reverse alphabetical order.

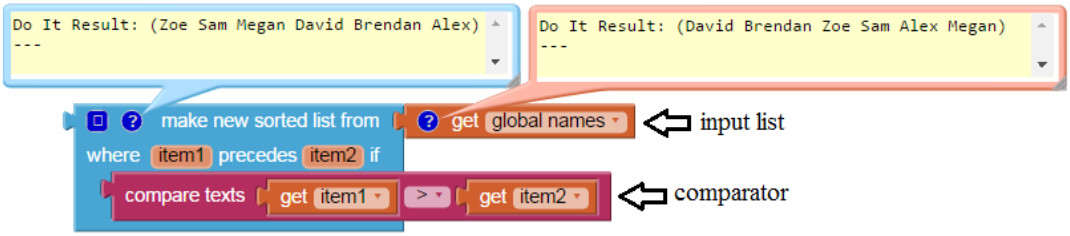

Figure 1-10: Sort with comparator block that sorts the input list in reverse alphabetical order

1.2.2 Destructive vs. Nondestructive Operators

List operators in the current App Inventor change the input list instead of returning a new list and are thus destructive. For example, the `insert` block inserts the given item at the specified index of the input list and does not return anything.

This inflexibility can be limiting to users because some procedures are easier to define with a nondestructive version of an operator. An example of such a procedure is `insertAll`, which takes in two inputs, an input list and an item, and returns a list of lists that is the result of inserting the item at every possible index of the input list. `insertAll(4,[9,5,6])` will therefore return the list of lists [[4,9,5,6],[9,4,5,6],[9,5,4,6],[9,5,6,4]]. Using a destructive version of the insert list item

operator will be problematic because during the first iteration, 4 will be inserted in the first index of [9,5,6], thereby changing the input list to be [4,9,5,6]. During the second iteration, 4 will be inserted into the second index of [4,9,5,6], changing the input list to be [4,4,9,5,6] and so forth. As seen in Figure 1-11, this problem can be solved by inserting 4 into the copy of the input list during each iteration. However, it would be useful in this case to have a nondestructive version of the insert list item operator.

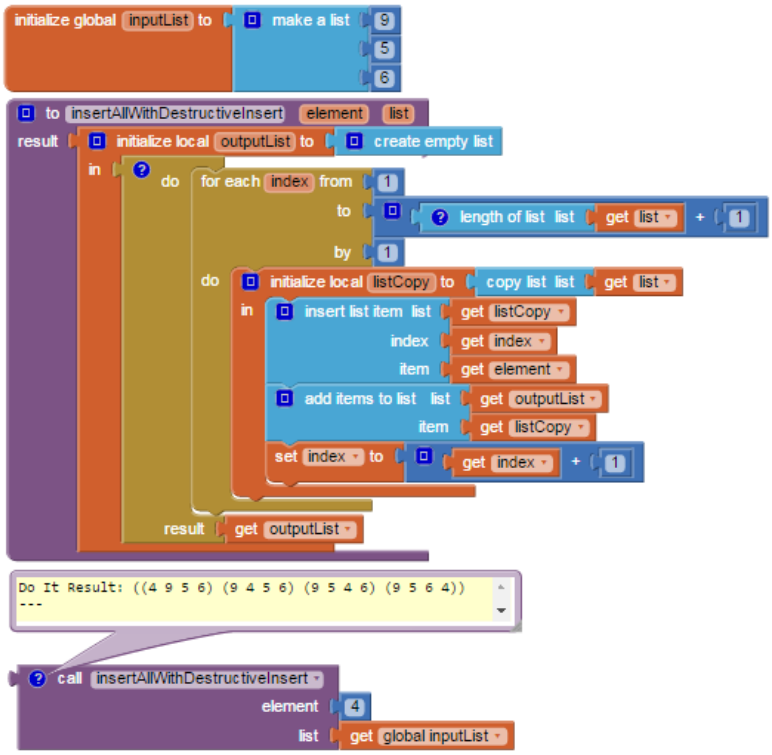

Figure 1-11: Using a loop and a destructive insert list operator to implement `insertAll`

In order to eliminate complicated programming workarounds and give users greater flexibility, I have implemented a mutator for my new operators as well as some of the existing operators–add, insert, remove, replace and append–that allows users to choose between making the operator destructive or nondestructive. In App Inventor, a mutator is a block that can change shape and functionality.

Figure 1-12 shows an insert operation whose default state, like the other blocks, is destructive. If the user clicks on the blue icon, however, he will see a pair of radio buttons and can select the `makes new list` button. This will automatically deselect the `change existing list` button. The insert operation will then become nondestructive and will return the newly created list resulting from inserting 4 at index 1 of the input list. The block thus changes shape from having a notch

(returns nothing) to a plug (returns a list).

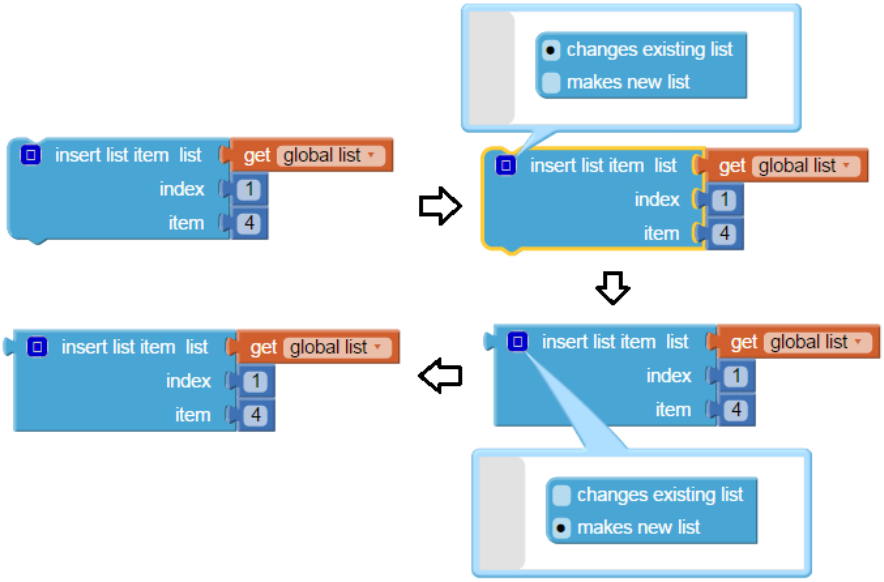

Figure 1-12: Switching between the destructive and nondestructive versions of the insert operator

With the addition of this new mechanism, implementing the `insertAll` procedure from above becomes a simpler task as illustrated in Figure 1-13. The user can now use a nondestructive version of the insert list item operator to insert 4 into each index of the input list and add the resulting lists as items to the output list. He no longer has to copy the input list during each iteration.

Moreover, once the insert operation is nondestructive, the loop can be replaced by a map over the insertion indices. To obtain a list of the insertion indices, I have defined a range procedure as shown in Figure 1-14. This procedure takes in a low and high value, and returns a new list of the numbers in between the low and high value in sequential order. Because `insertAll` inserts the given item at every possible index of the input list and also at the end of the input list, it returns an output list whose length is greater than the input list by 1. Therefore, the insertion indices range from 1 to one greater than the length of the input list of `insertAll`.

In Figure 1-15, I map over the list resulting from the range procedure by inserting 4 into each index of the input list. The existence of a nondestructive insert operator allows users to use a map operator instead of a loop and further simplifies the `insertAll` procedure.

1.2.3 Results from User Study

I conducted a user study with 18 Wellesley College students to test the usability of the map, filter, reduce and sort operators. I did not, however, compare these higher-order operators to loops or test

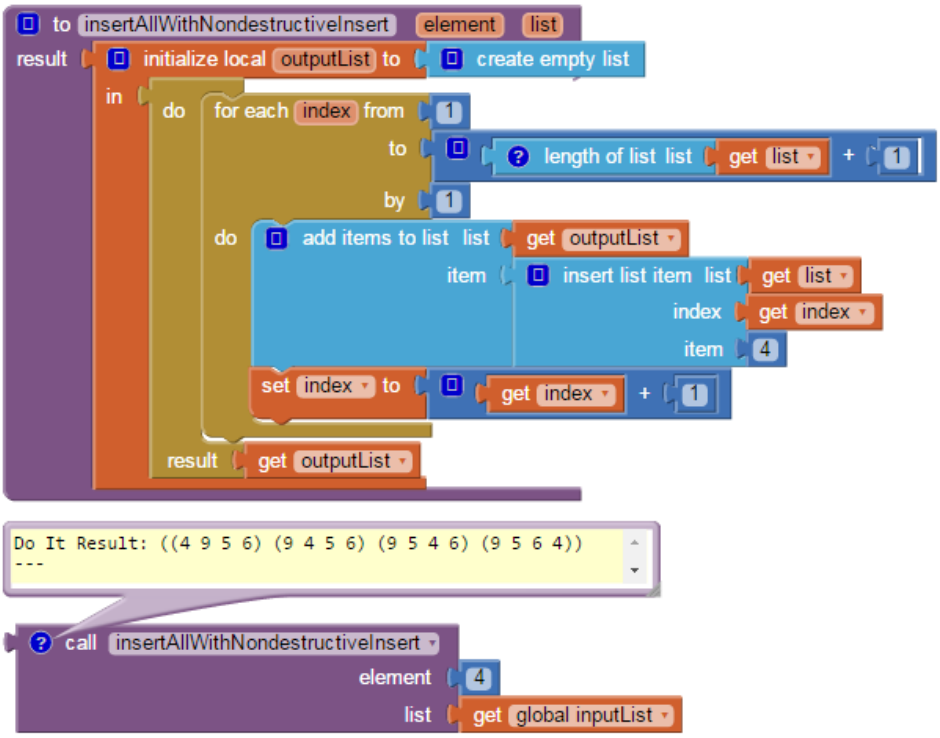

Figure 1-13: Using a loop and a nondestructive insert list operator to implement `insertAll`

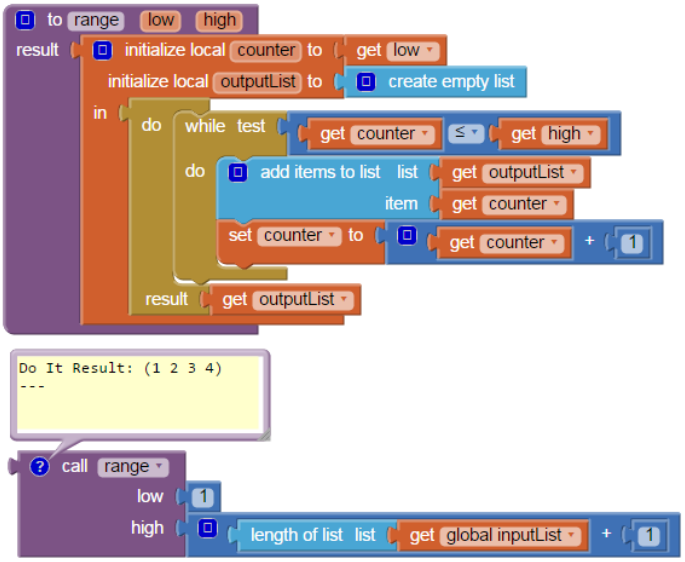

Figure 1-14: Range procedure

the destructive vs. nondestructive mechanism, which will be done in a future study.

The results indicate that most users were able to successfully map, filter, and reduce over a simple list and sort a list in increasing or decreasing order by the element itself or by one key. However,

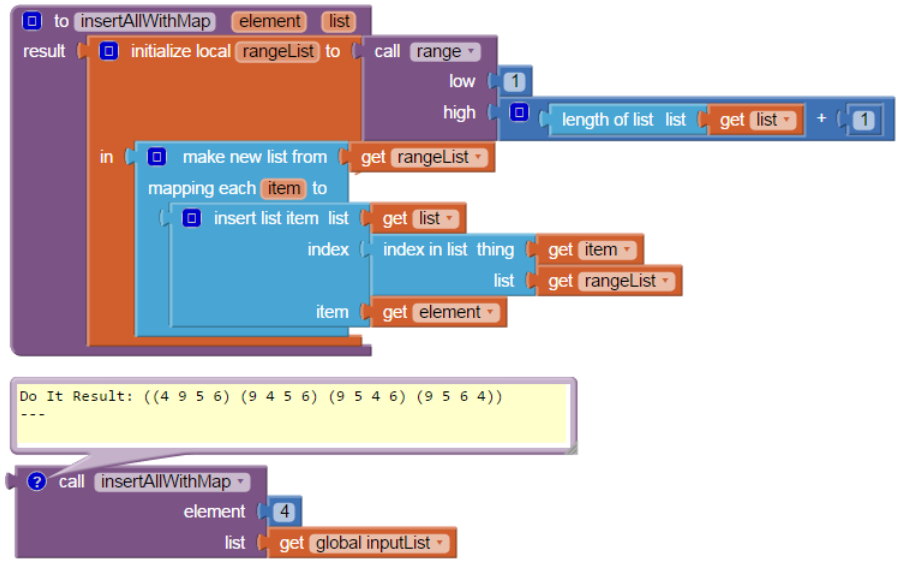

Figure 1-15: Using a map operator and a nondestructive insert list operator to implement `insertAll`

there were two problems that most users faced. First, during the mapping, filtering and reducing portion of the study, users found it challenging to use any one of these operators or a combination of the three on a list of lists. The concept of mapping and reducing were particularly difficult to understand compared to filtering. Second, during the sorting portion, users struggled to a sort a list given a primary and secondary key, although this may partly be due to a design flaw in the study, as discussed in Chapter 5. Instructive tutorials on manipulating lists of lists and sorting with two keys will be helpful in addressing these problems. Users also suggested ways to improve the usability of these operators, which will be incorporated in future work.

1.3 Road Map

The rest of this paper is organized as follows:

- Chapter 2 discusses list operators in other programming languages, both text-based and blocks-based.
- Chapter 3 examines the higher-order list operators I have added, which include map, filter, reduce and three different sorts.
- Chapter 4 explains the new mechanism in which the user can choose to make an operator destructive or nondestructive.

- Chapter 5 presents the design and results from a user study I conducted to test the usability of the higher-order operators.
- Chapter 6 discusses the implementation of the higher-order operations and destructive vs. nondestructive mechanism.
- Chapter 7 concludes the paper, summarizing both the design and the usability of the new list operators, as well as discussing future work that could be done as an extension of this project.
- Appendix A is my user study protocol that includes an explanation of each block followed by specific tasks.
- Appendix B is a code listing containing the changes I made to list.js to describe the new operators in the Blockly Editor.
- Appendix C is another code listing of the changes I made to runtime.scm to give the new operators their functionality.

Chapter 2

Related Work

2.1 Sequential Data Structures in Text-based Languages

2.1.1 List and Array Properties

Languages such as Python, JavaScript, Scheme and ML support sequential data structures that are used for organizing data. I will compare these structures using three dimensions:

1. heterogeneity: a list is heterogeneous if it can contain items of different types.
2. mutability of list slots: the value of list slots can change.
3. resizability: the number of list slots can change.

In Python, lists are mutable and heterogeneous. They are also resizable, so items can be added or removed from the original list after its creation [Poi14].

JavaScript arrays are similar to Python lists as both data structures are heterogeneous, mutable and resizable. The contents of an array can be modified in several ways: 1) by popping the last item off the list 2) by pushing an item onto the end of the list 3) by shifting all the items one place down after removing the first item 4) by using splice to insert or remove items at a particular index and 5) by getting and setting a particular index of an array [W3S15]. The first four operations can also be performed in Python.

Functional languages such as Scheme and ML also support lists. Scheme lists nodes are linked and represented as pairs, which consist of a car and a cdr. In a nonempty list, the car is the first item in the list and the cdr is a list containing the remaining items of the list. The car of each consecutive pair refers to the items of the list. Scheme lists are heterogeneous and mutable through

methods such as set-car! and set-cdr! that update the values of the list. Moreover, lists are resizable as items can be inserted and removed from the original list at different indexes [Shi99].

Like Scheme, ML lists are singly linked. However, items in an ML list must be of the same type and are immutable. ML lists are not resizable because the original list cannot be modified once it is created; adding and removing items are nondestructive operations that return a new list rather than changing the existing list.

The design choices of App Inventor lists closely resemble those of Python. Both lists can contain items of different types and can be modified through the addition, removal or updating of items. One difference between App Inventor and Python lists, however, is that App Inventor indexes start with 1, not 0 like in Python. These properties allow App Inventor's target audience—users with limited programming background—to easily understand and manipulate lists.

	Heterogeneous	Mutable list items	Resizable	Linked list nodes
App Inventor lists	✓	✓	✓	
Python lists	✓	✓	✓	
JavaScript arrays	✓	✓	✓	
Scheme lists	✓	✓	✓	✓
ML lists				✓

Table 2.1: Summary of list or array properties in different languages

2.1.2 Destructive vs. Nondestructive Data Structures and Functions

CLU is a programming language that supports both mutable and immutable data structures. While mutable datatypes can be modified after creation, immutable ones cannot. For homogeneous collections, CLU provides mutable arrays as well as an immutable counterpart in sequences. For heterogeneous collections, mutable records and immutable structs are used [LG86]. Similar to CLU, Python has mutable and immutable sequences in the form of lists and tuples, respectively. Within lists, however, Python does not give users a choice of deciding between a destructive or a nondestructive operation. There are a few exceptions to this, which include methods like sort(), which is destructive, and sorted(), which is nondestructive. Examples of sort() and sorted() are shown in Figure 2-1.

This limitation means that if the user wants to convert from a destructive operation to a nondestructive one, he must copy the original list and perform the operation on the copy. If the user wants to convert from a nondestructive operation to a destructive one, he has to ensure that the number and contents of slots in the input list are modified to match those in the output list. It would be useful, however, if the user can choose whether he wants an operation to be destructive or

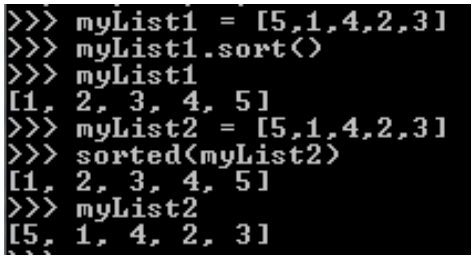

Figure 2-1: sort() vs. sorted() in Python

nondestructive. This gives users greater flexibility when manipulating lists and eliminates the complicated workarounds neccessary in the current App Inventor. While similar to the concept of CLU's mutable and immutable dataypes and Python's lists and tuples, my mutator-based mechanism in App Inventor is different in that it allows the user to choose a destructive or nondestructive operation within the same datatype. That is, instead of switching between mutable data structures and their immutable counterparts, users work with lists only and can set list operators to be destructive or nondestructive.

2.2 Lists in Blocks-based Languages

2.2.1 Scratch

Scratch [Scr] is a blocks-based programming environment in which users create stories and animations by using blocks to specify the behavior of sprites. In Scratch, lists are a type of data structure that behave as re-sizable arrays and store data. However, they are limited in that users cannot set a variable equal to a list or declare a list of lists. Figure 2-2 shows the list operators that Scratch supports.

Scratch does not have higher-order list operators, so users typically use a for loop to iterate through the elements of a list. For example, as seen in Figure 2-3, in order to implement a destructive map that doubles all the elements of the input list, the user can initialize a counter to keep track of the current list item and use a repeat block to iterate through all the items, replacing each item by its double.

2.2.2 Snap!

Snap! [Sna] is an extended reimplementation of Scratch. Unlike Scratch lists, Snap! lists can be anonymous (exist without a name) and can be inserted as an item into another list. Figure 2-4

Figure 2-2: List operators in Scratch

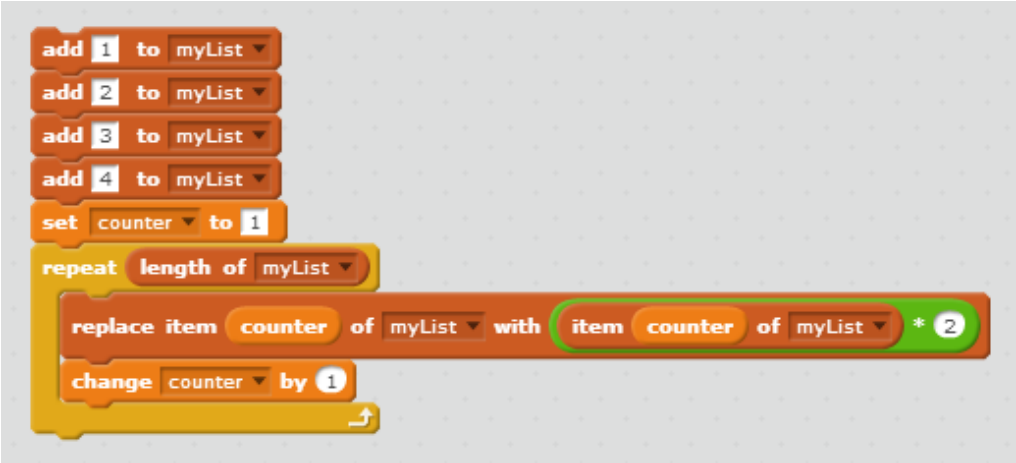

Figure 2-3: Mapping a list in Scratch by doubling its elements

displays the list operators supported by Snap!.

As Snap! is inspired by Scheme, it supports Scheme-like concepts like higher-order list operators. The higher-order operators include map, filter in the form of the `keep items such that` block and reduce in the form of the `combine with` block. Using the same example as in the Scratch section, a user can use the map block to double the elements of a list as seen in Figure 2-5. The map block takes in two inputs: a reporter block and a list. The reporter block in this case is the multiplication block and has one empty input. After each iteration, the next list item is placed into the empty input and is multiplied by two. Furthermore, the multiplication block has a gray ring around it, which is part of Snap!'s representation of Scheme's `lambda` and indicates that the input is a block

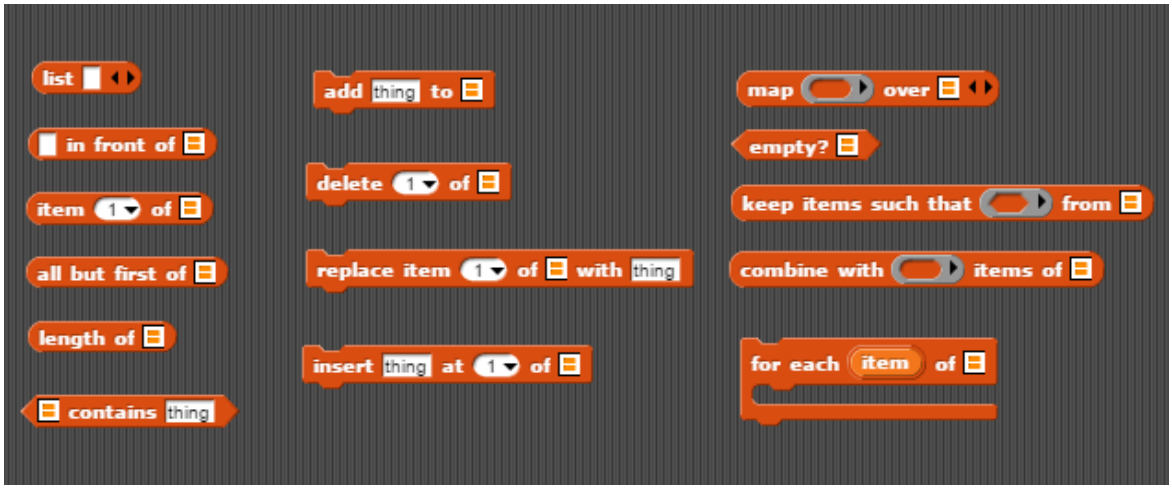

Figure 2-4: List operators in Snap!

rather than the value of the block. This ensures that the reporter block iterates through the list and continues to multiply each list item by two.

Figure 2-5: Mapping a list in Snap! by doubling its elements

In addition to supporting higher-order operators, Snap! allows for both imperative and functional programming. In imperative programming, users use command blocks such as `add`, `delete`, `replace` and `insert` that change the input list. In functional programming, however, users use reporter blocks such as `in front of` and `all but first` that return a new list instead of modifying the existing list. For example, the `in front of` block returns a new list that is result of adding an item to the beginning of the input list. Snap! provides users with the option of switching between imperative and functional programming, which is a feature that I aimed to add to App Inventor by implementing a mechanism that allows users to choose between a destructive and nondestructive version of each list operator.

Chapter 3

Higher-Order Operators

3.1 Map

Mapping is a common list operation that takes in a function and an input list, and applies the function to each item in the input list. In Python, for example, the map operator can be used to return a new list in which each item of the original list is incremented by 1.

```
>>>map(lambda item: item+1, [32,20,23,18,27])
[33,21,24,19,28]
```

In the current App Inventor, users typically use loops to map over a list. For example, suppose a user would like to implement the Python example from above and return a new list in which every element of the input list is incremented by 1. One solution to this problem is shown in Figure 3-1. In the `mapIncrement` procedure, I initialize a local variable `outputList` to an empty list. A loop is then used to iterate through `inputList`, incrementing each item by 1 and adding it to the `outputList`. The loop block has a built-in parameter called `item` which refers to the current list item that it is iterating over. As seen in this example, performing a nondestructive map operation with loops involves declaring an extra output list and adding the transformed items of the input list to this output list.

In order to simplify the process of mapping, I have added a map block to App Inventor, as shown in Figure 3-2. Unlike the map operator in text-based languages, however, this map block does not take in a function as one of its parameters. In fact, App Inventor does not support first class functions, or the ability to pass in functions as parameters. To bypass this problem, I have built into the block the parameter declaration of the functional argument. Therefore, the map block

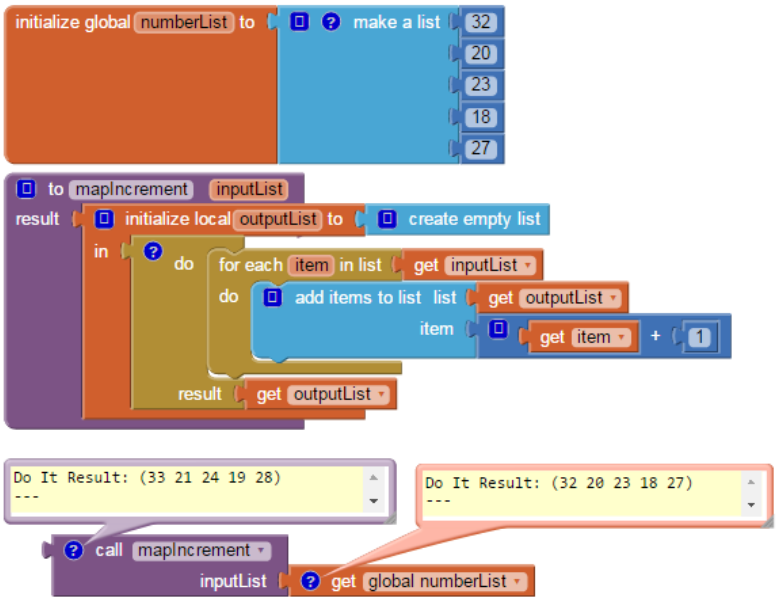

Figure 3-1: Using a loop to return a new list in which every element of the input list is incremented by 1.

simply takes in an input list and the body expression of the functional argument. The block returns a new list that is the same length as the input list in which each element is the result of evaluating the body expression for each corresponding item in the input list.

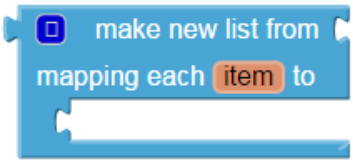

Figure 3-2: Map

Revisiting the example from above, Figure 3-3 illustrates this bypassing of functional arguments. Instead of taking in the function (lambda item: item + 1) like Python's map, this map block has a built-in parameter item and only takes in (item + 1), the body expression of the function. This block provides a simpler way to map over a list compared to loops, as it only requires two inputs and does not involve initializing other variables.

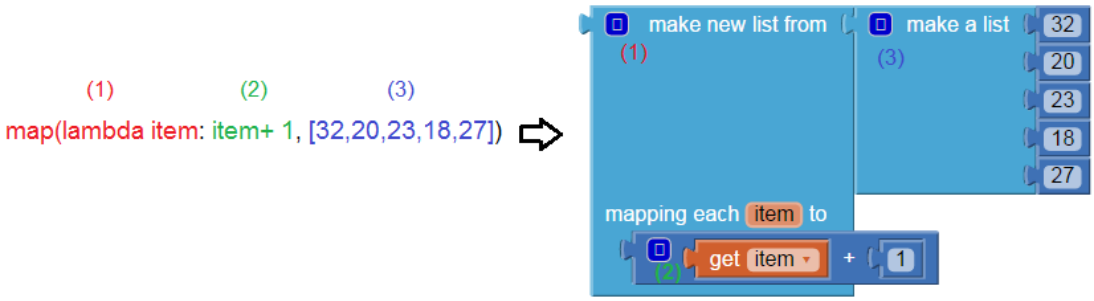

Figure 3-3: The map block has a built-in parameter named item and takes in the body expression of the lambda function.

3.2 Filter

Loops can also be used to filter a list by a certain criteria. For instance, a user may want to return a new list by keeping only the items of the input list that are even. A number is even if the modulo of the number divided by 2 equals 0; in other words, there is no remainder after dividing it by 2. As seen in Figure 3-4, a conditional statement is used inside a loop to check if each list item is even. If it is, the item is added to the output list. Otherwise, it is discarded. This loop uses a similar pattern to the loop that mapped over a list, as both involve the declaration of an output list and addition of items to this output list.

As an alternative to defining tedious loops to filter lists, I have developed a filter block as shown in Figure 3-5. It has two expression sockets: the first specifies an input list and the second specifies a boolean expression. The block returns a new list that only includes the items of the input list for which the boolean expression returns true.

If the user wants to filter for only the even items in `numberList` as in the above example, she simply needs to provide two inputs to the filter block: the `numberList`, and a boolean expression that checks to see if dividing each item by 2 results in a remainder of 0. If the remainder equals 0, the number is even and will be a part of the output list. Otherwise, it is discarded.

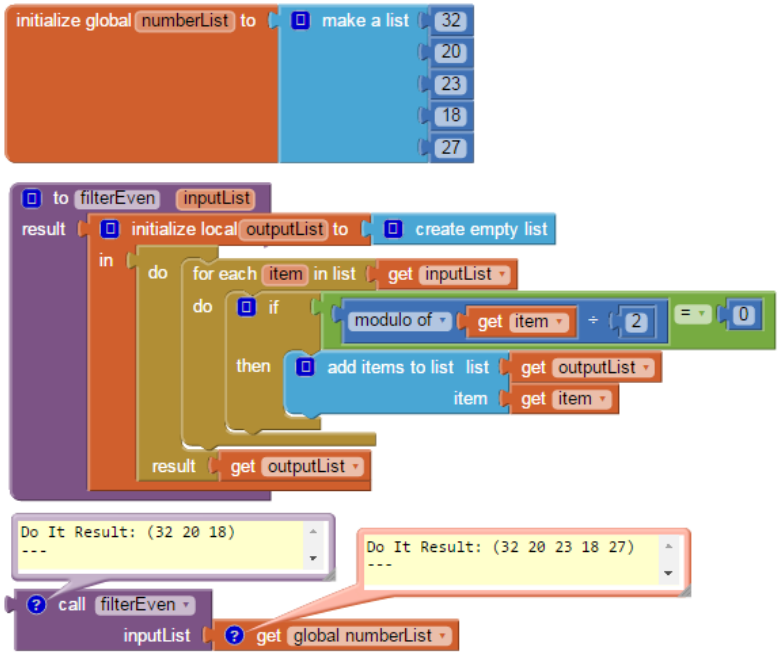

Figure 3-4: Using a loop to return a new list by keeping only the items of the input list that are even

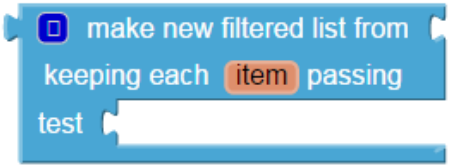

Figure 3-5: Filter

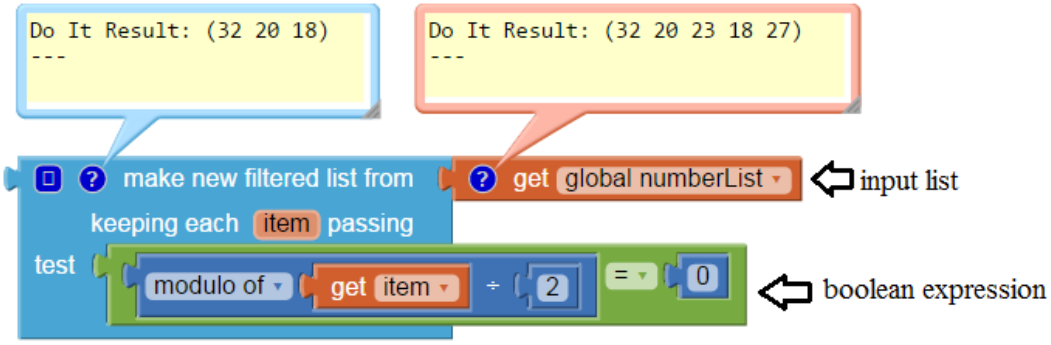

Figure 3-6: Using a filter operator to return a new list by keeping only the items of the input list that are even

3.3 Reduce

To reduce a list using loops, a local or global variable must be initialized to keep track of the accumulated answer after each iteration. In Figure 3-7, I have shown one way to find the sum of

a list's item using loops and a local variable answer. Answer is initialized to 0 and continuously updated inside a loop as it is set equal to the sum of the answer and the current list item.

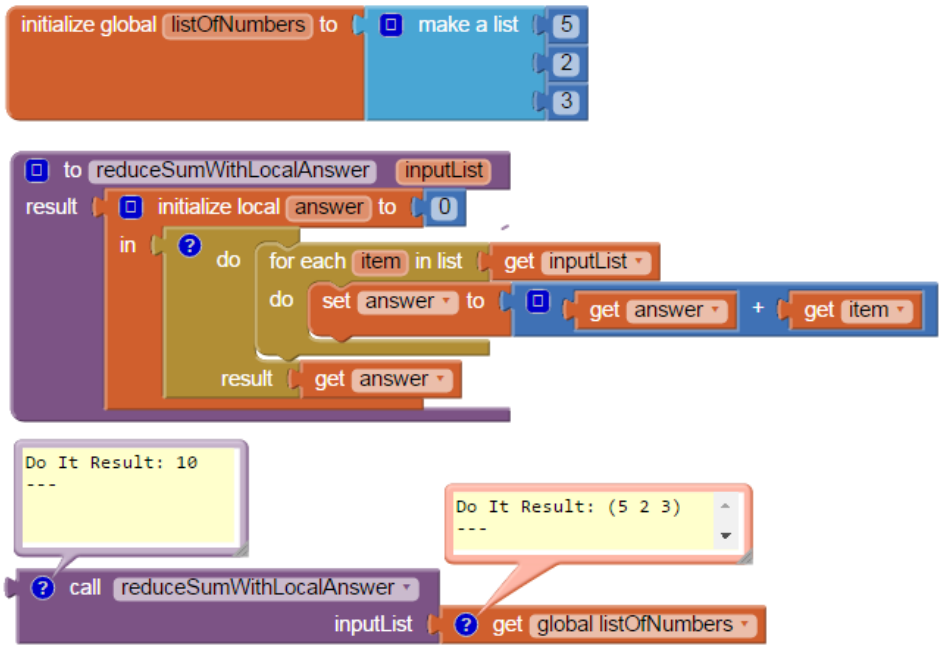

Figure 3-7: Using a local variable answer to find the sum of a list's items

An alternative to the solution in Figure 3-7 is to declare a global rather than local variable for answer. Because App Inventor is an event-based program, variables may sometimes have to be global in order to be referenced by an event handler [FTP14]. In Figure 3-8, the user initializes the global variable answer to 0. Before entering the loop, he re-initializes answer to 0 because answer may still be equal to the sum of the list items from a previous execution. The rest of the procedure involving the loop is the same as Figure 3-7.

A common mistake that users make, however, is not re-initializing the global variable answer before each iteration of the loop. As shown in Figure 3-9, such a program returns the correct sum of list items after the first execution. However, during the second execution, answer is not re-initialized to 0 and still equals 10, the sum of the list items from the previous execution. Therefore, 20, not 10, is returned as the answer after the second execution. All subsequent executions after the first will return an incorrect sum of the list items.

My solution to this problem is to provide a `reduce` block shown in Figure 3-10 that has built into it the initialization of the necessary variables. This block combines the elements of a list into a single value. It has three expression sockets: an input list, an initial value and a body expression. If the list is empty, then the initial value is returned. Otherwise, `answerSoFar`, or the accumulating

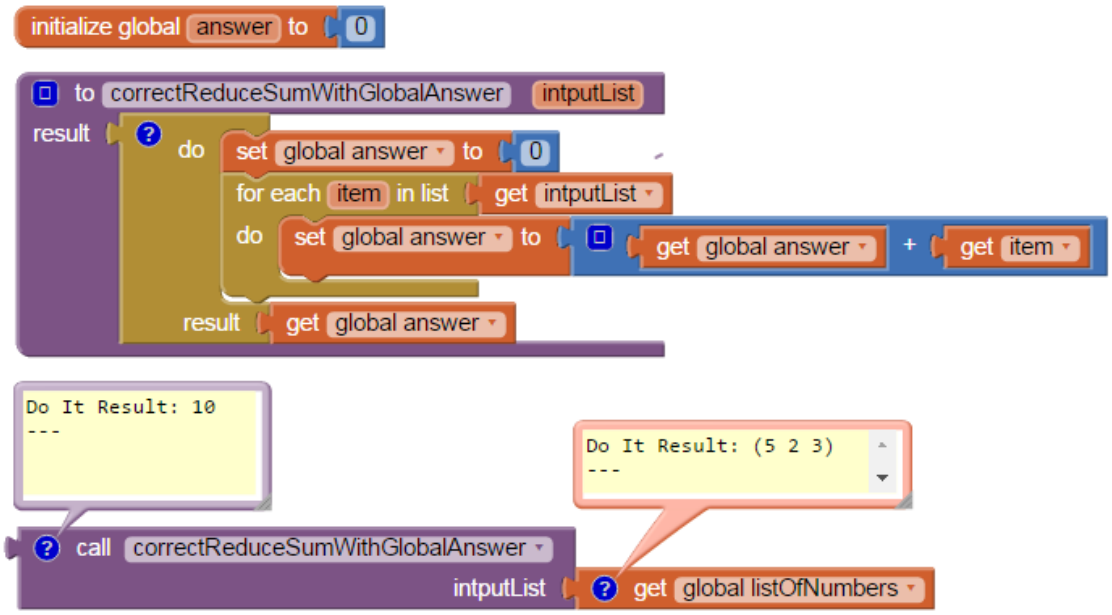

Figure 3-8: Correctly using a global variable answer to find the sum of a list's items

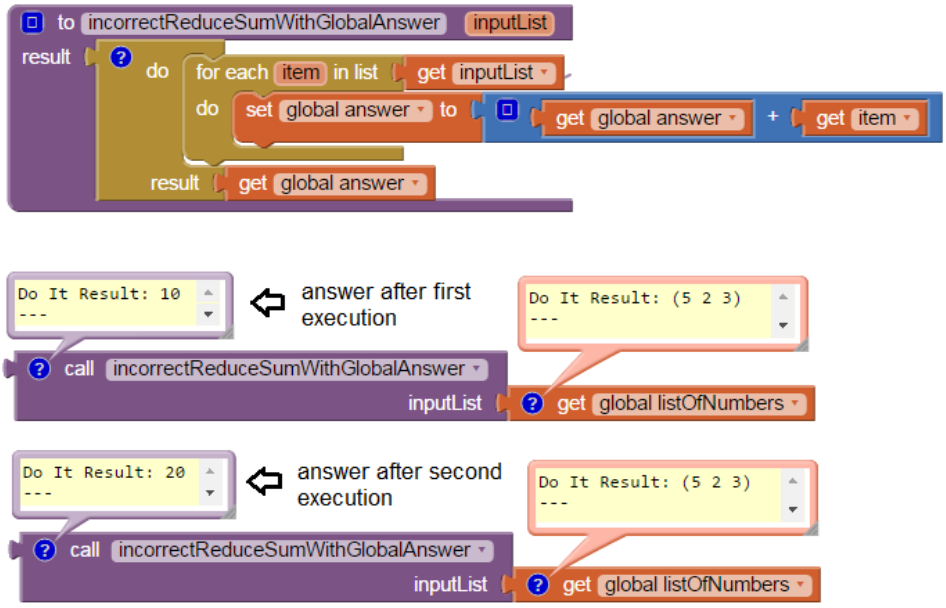

Figure 3-9: Incorrectly uses a global variable answer to find the sum of a list's items. It returns the correct answer after the first execution, but returns an incorrect answer for the second and all subsequent executions.

answer, is initialized to the initial value. The reduce block combines each item of the input list with the `answerSoFar` using the body expression in order to calculate the final answer, which it returns.

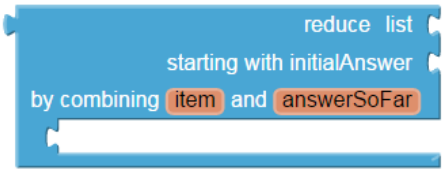

Figure 3-10: Reduce

With this `reduce` block, summing up the items of the input list becomes a simpler task. As seen in Figure 3-11, the reduce block takes in three inputs to find the sum of the list items:

1. the input list
2. an initialAnswer of 0 since the sum of an empty list is 0
3. a body expression that adds each item to answerSoFar to calculate the total sum

By using this `reduce` block instead of a loop, the user no longer has to initialize a global or local variable to keep track of the accumulating answer.

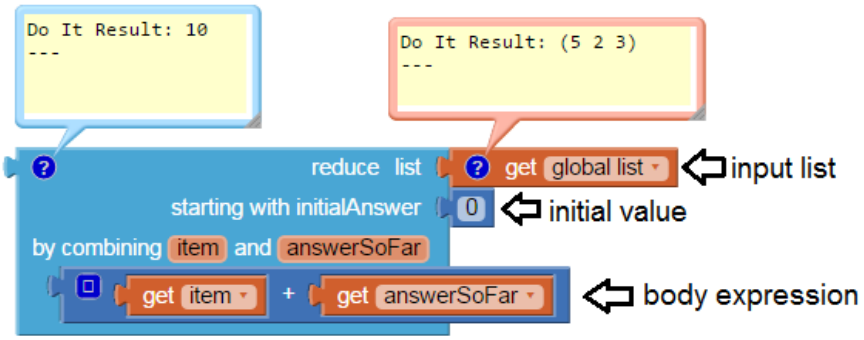

Figure 3-11: Using reduce to find the sum of the items of a list

3.4 Sort

App Inventor currently lacks an easy way to sort lists; users either copy complicated sorting procedures from sites or use programming workarounds to leverage JavaScript's sorting method within App Inventor.

For example, as shown in Figure 3-12, the Imaginity site [Dut13] provides a bubble sort algorithm for App Inventor lists. This procedure takes in three inputs: an input list, a string that specifies whether to sort in ascending or descending order, and a string that specifies whether the list contains

numbers or strings. Two loops are used to keep track of the indexes of the pair of items that are being compared. Through various conditional statements, this procedure checks to see if the current pair of items are in the correct order. If not, the first item in the pair switches positions with the second item in the pair.

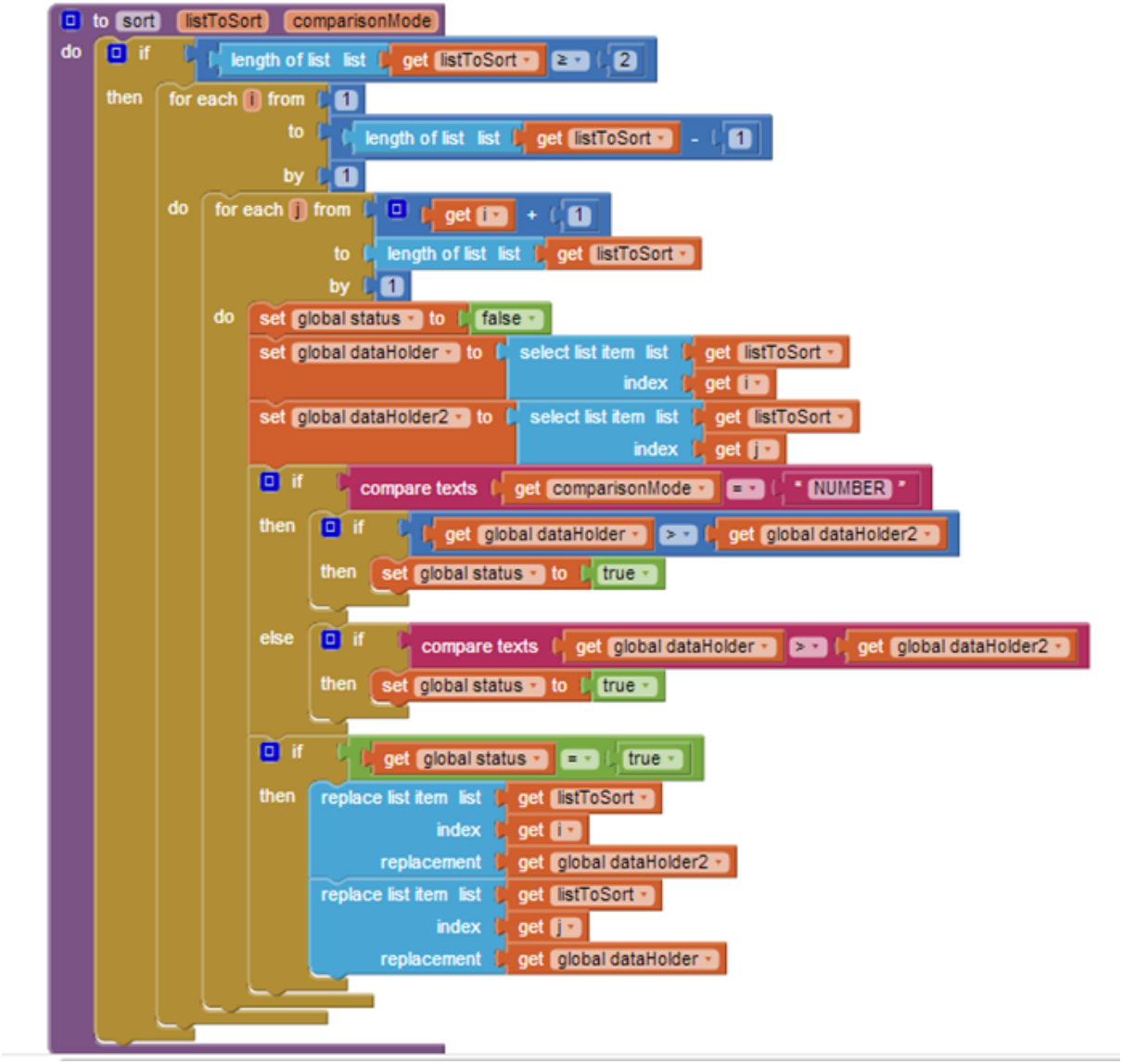

Figure 3-12: Bubble sort [Dut13]

The App Inventor Code Snippets page [Tai15] suggests one trick to sorting lists, which is to call JavaScript's sort method using the WebViewer component as seen in Figure 3-13. This procedure works by first converting the list to a csv row, executing the sort() method from JavaScript in the URL of a web page and then returning the sorted list in the title of the page.

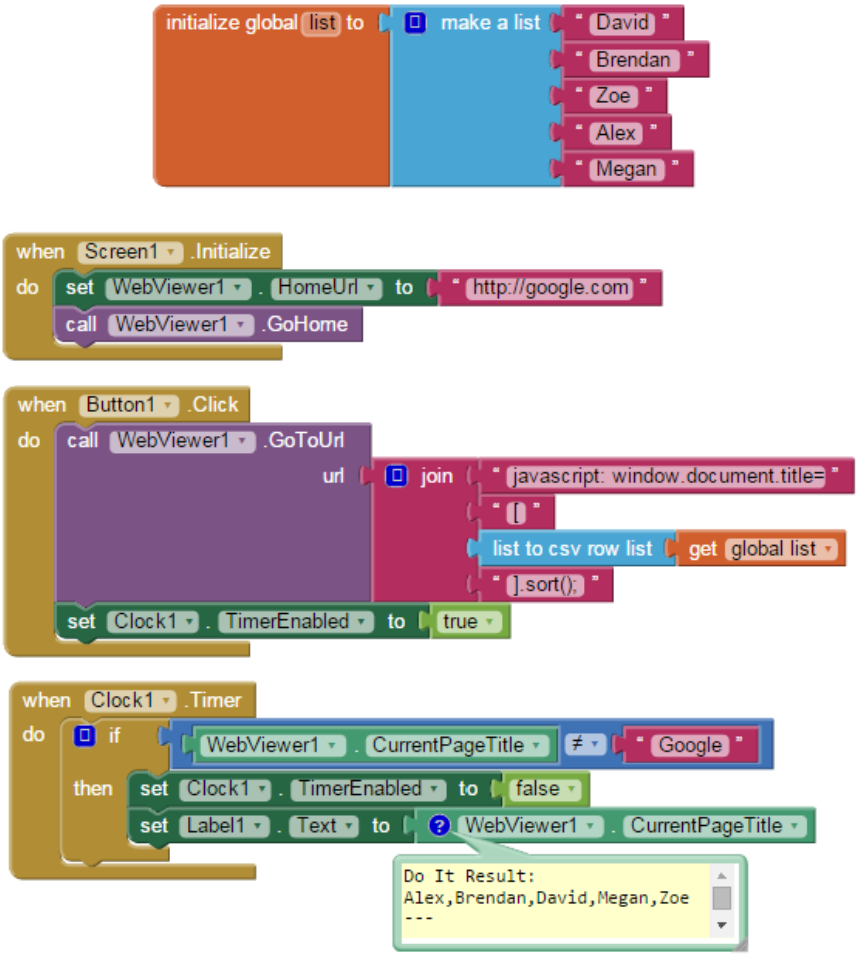

Figure 3-13: Sorting a list using the WebViewer component [Tai15]

To simplify sorting within App Inventor, I have added three different sort blocks: one using a default comparator that arranges the list items in increasing order; one with a key function that returns a proxy value used for sorting the list with the default comparator; and one that uses a comparator provided by the user.

As shown in Figure 3-14, the basic sort block returns a new list whose elements are those of the input list sorted in increasing order. It uses a default comparator defined on any two App Inventor values. It groups items of the same type together, and then sorts accordingly within the same type group. The current order of the types is booleans, numbers, strings, lists and then components. False is defined to be less than true, and components are compared using their hashcodes.

Figure 3-15 illustrates how the default comparator operates on a list containing items of different types. The list `differentKindOfValues` has four different types of values: numbers, strings, booleans and lists. Since the order of the types is booleans, numbers, strings and then lists, booleans

Figure 3-14: Basic sort

`false` and `true` are followed by numbers `19` and `23` and strings "bunny" and "dog." Finally, since the first item of `(17 fish)` is a number and the first item of `(cat 3)` is a string, `(17 fish)` is smaller and placed prior to `(cat 3)` in the output list.

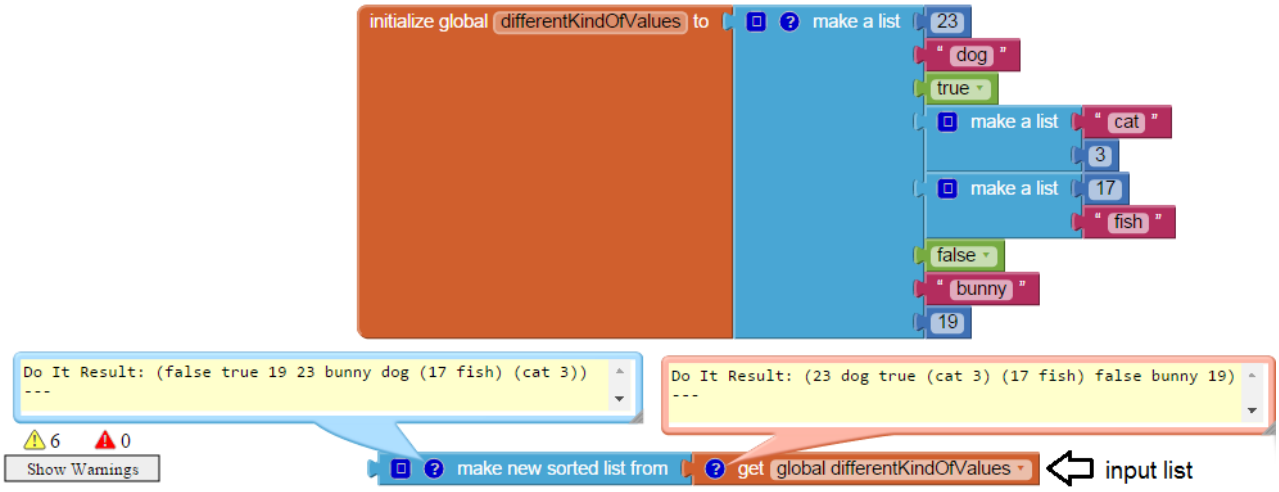

Figure 3-15: Basic sort on a hetergenous list

As shown in Figure 3-16, the sort with key block has two expression sockets: an input list and a key expression. It returns a new output list whose elements are those of the input list sorted in increasing order determined by using a default comparator on the result of the key expression applied to each item. If two items have the same key, they maintain their relative position in the output list.

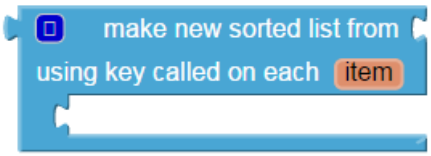

Figure 3-16: Sort with key

Figure 3-17 shows one use of sort with key, which is to sort the input list in increasing order by the absolute value of each item.

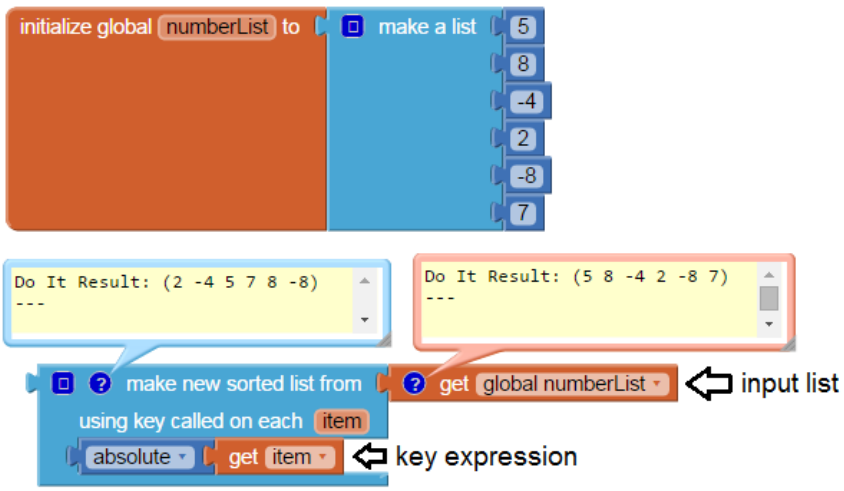

Figure 3-17: Sort with key that sorts the input list in increasing order by absolute value

Both the basic sort and the sort with key use a default less-than-or-equal comparator and are stable algorithms. This means that they preserve the original order of equal elements in the output list. This is particularly important when sorting a list with two keys. If the sort is stable, sorting first by the secondary key and then by the primary key will produce an output list that is correctly sorted by both keys.

The final sort is the sort with comparator block and is illustrated in Figure 3-18. It has two expression sockets: an input list and a boolean expression. It returns a new output list whose elements are those of the input list sorted according to a comparator defined by a boolean expression that determines if `item1` is less than or equal to `item2`.

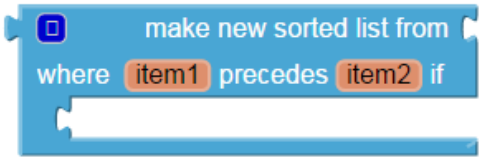

Figure 3-18: Sort with comparator

The sort with comparator is used for all cases that require sorting in decreasing order. For example, Figure 3-19 shows a list of lists called `studentList` in which each item is a list containing a student's age and name. The sort with comparator block sorts the list in decreasing order by student age, the first index of each item. Because this sort uses a greater-than-or-equal operator, it is stable. Ben and Anna, who are both 18 years old, maintain their relative position in the output list.

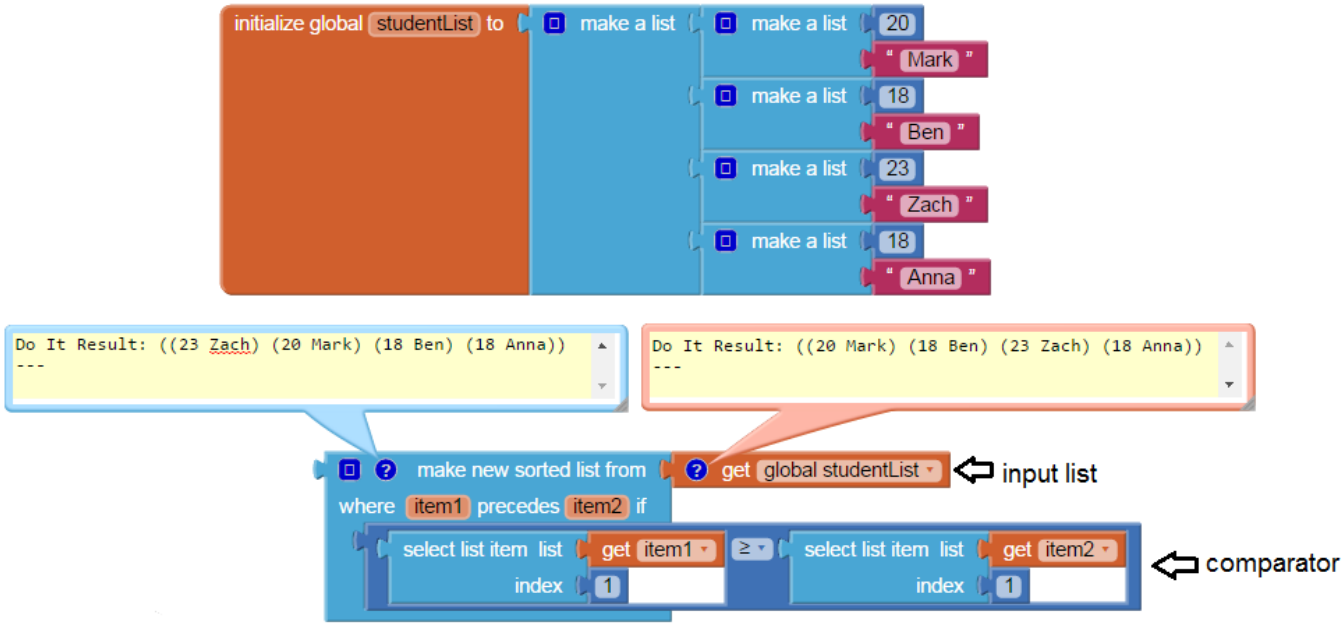

Figure 3-19: Stable sorting with the comparator

If this sort used a strictly greater-than operator, however, it becomes unstable, as shown in Figure 3-20. Although Ben precedes Anna in the input list, Anna precedes Ben in the output list.

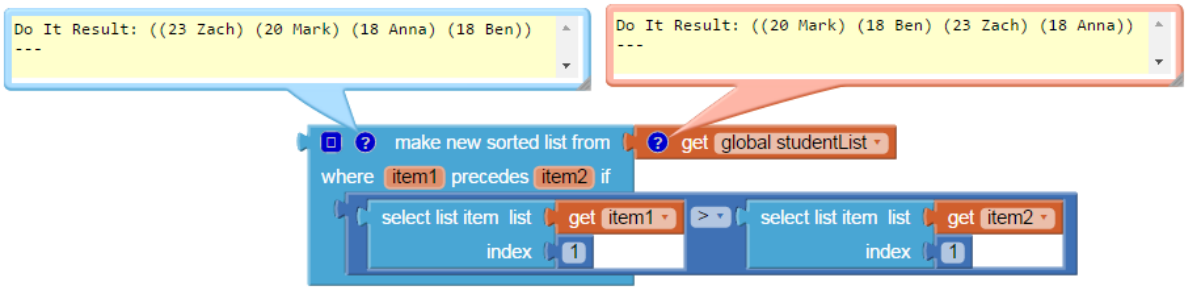

Figure 3-20: Unstable sorting with the comparator

The table below summarizes when one would use each of the sorting blocks. The basic sort block takes in a list as its sole input and uses a default comparator that sorts the list elements in ascending order. Therefore, its use is limited to one case. Like basic sort, sort with key also uses the default comparator and is restricted to cases that sorts the list items in increasing order. However, it can use any number of keys as a means of arranging the list items. Finally, because the sort with comparator block uses a comparator provided by the user, it is adaptable to any case and can arrange the items in any order, including descending order which was not possible with the basic

sort and sort with key.

	Basic sort	Sort with key	Sort with comparator
Element itself, ascending	✓	✓	✓
Element itself, descending			✓
One key, ascending		✓	✓
One key, descending			✓
Two keys, both ascending		✓	✓
Two keys, at least one descending			✓

Table 3.1: Summary of the different uses of the three sort blocks

Chapter 4

Destructive vs. Nondestructive Operators

In the current App Inventor, most list operators are destructive and modify the existing list while a few are nondestructive and return a new list. In order to simulate a nondestructive version of a destructive operation, the user must make a procedure that makes a copy of the original list, perform the destructive operator on the copy and then return the copy. In Figure 4-1, I have followed these steps and defined a nondestructive filter from a destructive filter operation. My example keeps only the items of `listOfAnimals` whose lengths are greater than 3. This process is tedious and requires that programmers understand this pattern.

The opposite direction—making a destructive version of a nondestructive operator–is more difficult. In Figure 4-2, I have defined a destructive filter from a nondestructive filter operation. This procedure deletes the elements of the existing `inputList` and replaces them by the elements of the new `outputList` returned by the nondestructive operator.

Converting from one version of an operation to another requires complicated and repetitive programming on the part of the user. To eliminate this problem, I have implemented a mechanism for list operators that gives the user a choice of choosing between a destructive or nondestructive version of each operator. This involved creating a mutator, which is a block that can change shape and functionality based on whether the user selected the destructive or nondestructive option. With this mutator, list processing is greatly simplified as users no longer have to resort to programming workarounds illustrated in Figure 4-1 and 4-2.

This mechanism is illustrated in Figure 4-3 using the filter operator. When the user pulls out

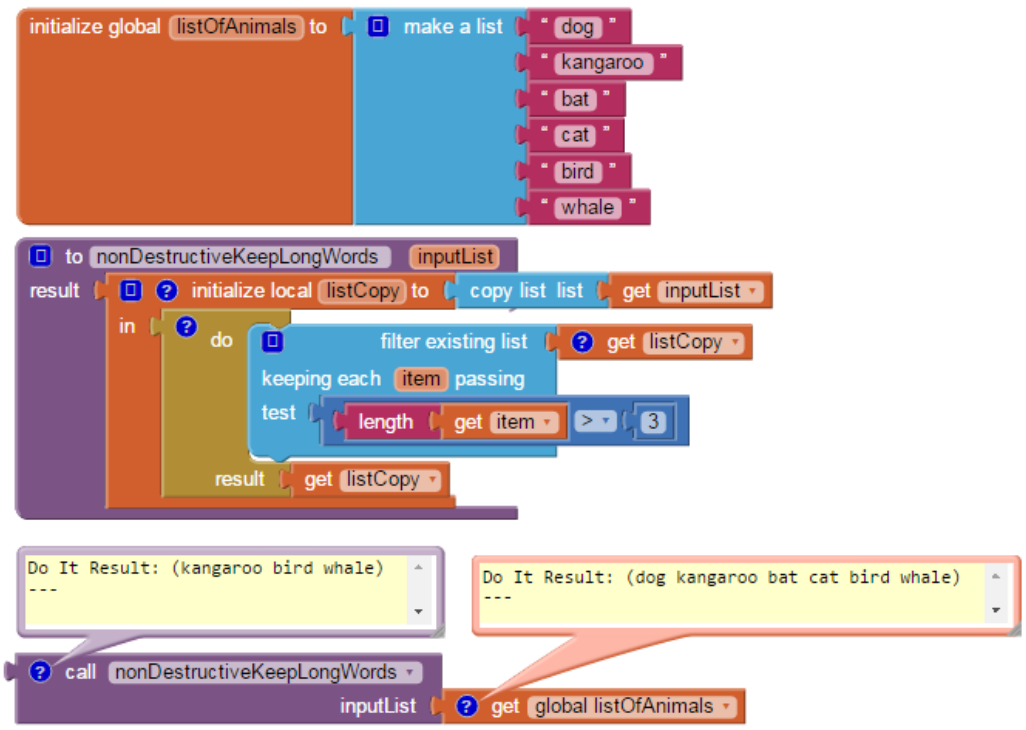

Figure 4-1: Simulating a nondestructive version of a destructive filter operator

a filter block from the list drawer, its state is destructive. However, he may change the state to nondestructive by clicking on the blue button on the upper left corner of the block; this button indicates that the block is a mutator. A bubble will then appear, showing two radio buttons: one for the option `changes existing list` and another for the option `makes new list`. If the user selects the radio button for `makes new list`, `changes existing list` is automatically deselected, and the block changes from a destructive to nondestructive filter operator. This change in functionality can be visually observed through the change in the block's shape. The destructive operator has a notch on the left side indicating that it performs an action on the existing list and returns nothing. On the other hand, the nondestructive operator has a plug on the left side, which means that it returns a new list instead of modifying the existing list.

Having a choice between a destructive and nondestructive version of an operator is important, particularly for sorting. Python, for example, supports both sort(), which changes the existing list and returns None, and sorted(), which returns a newly sorted list. sort() may be faster than sorted() in certain cases because it is an in-place operation and does not create a new list. However, the nondestructive sorted() allows users to engage in functional programming. One benefit of functional programming is that it avoids side effects and mutating data—therefore, it may be easier for beginner

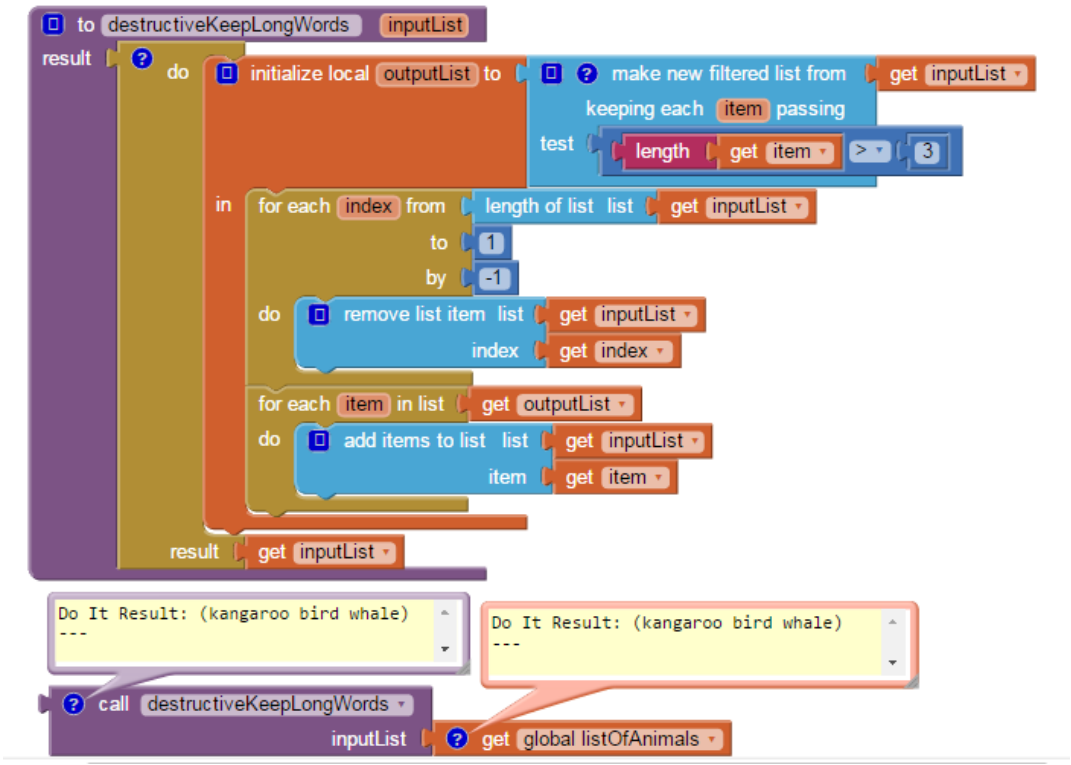

Figure 4-2: Simulating a destructive version of a nondestructive filter operator

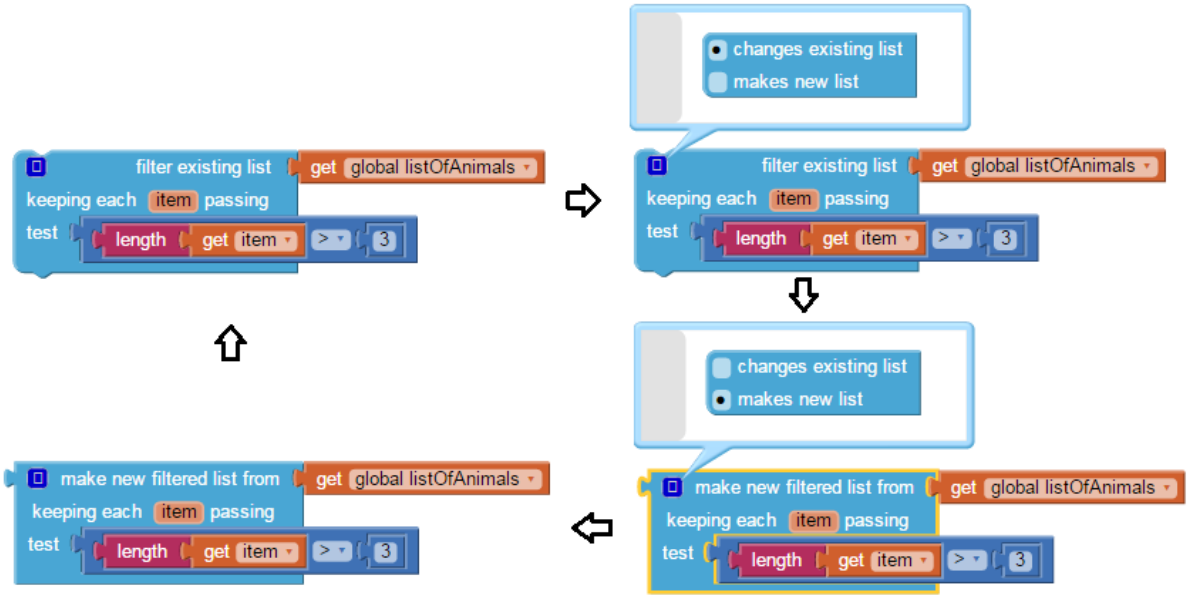

Figure 4-3: Switching between the destructive and nondestructive versions of the filter operator

programmers to understand and trace the steps of their program.

Chapter 5

User Study

5.1 Purpose and Structure of the Study

I conducted a user study with 18 students at Wellesley College who have previously worked with App Inventor through a course or a project. The purpose of this study was to determine if the new list operators I have added are usable by App Inventor's target audience, which consists of users with limited programming background. The results are mixed; they indicate that those with previous exposure to these operators performed better, but a few number without previous knowledge were still able to use these operators.

In this chapter, I will summarize the important points of the study. The detailed protocol is included in Appendix A. The study lasted up to 90 minutes and began with users filling out a pre-task survey answering questions about their age, major(s), previous knowledge of App Inventor and map, filter and reduce (if any), and computer science courses they have taken at Wellesley. After giving a brief tutorial on each of the new list operator blocks, I asked users to answer questions from a hard copy of a written protocol. The tasks involved writing new programs as well as explaining the meaning of programs that use these blocks. I took notes during each study and recorded a screencast for each user that captured her voice and her actions on the screen.

The first part of the study consisted of eight tasks involving mapping, filtering and/or reducing over a simple list or a list of lists. The different types of tasks are listed below:

Task 1 Mapping over a simple list

Task 2 Mapping over a list of lists

Task 3 Filtering over a simple list

Task 4 Filtering over a list of lists

Task 5 Reducing over a simple list

Task 6 Reducing over a list of lists

Task 7 Explaining the meaning of a composition of map, filter and reduce blocks

Task 8 Writing a program that required using a combination of map, filter and/or reduce blocks

The second part of the study involved six sorting tasks using a list of lists. Users could use any of the three sorting blocks for each task. The different types of tasks are listed below:

Task 9 Sorting a list in ascending order by the item itself

Task 10 Sorting a list in descending order by the item itself

Task 11 Sorting a list with one ascending key

Task 12 Sorting a list with one descending key

Task 13 Sorting a list with two ascending keys

Task 14 Sorting a list with one ascending key and one descending key

Since the default state of each list operator block is destructive, I asked users to use the mutator to change the state to be nondestructive. This decision to make the default state destructive will be revisited in future work. Users tested their programs by connecting the computer to an Android device and running `Do It` on the blocks.

At the end of the study, users filled out a post task survey where they indicated why it was easy or difficult to use these blocks and shared any suggestions for improvements.

5.2 Demographic of Users

I conducted the study with 18 female users from 7 different majors, with 11 users (61%) majoring in computer science or double majoring in computer science and another subject. Users' ages ranged from 18 to 23 years old, with the majority of students of age 20 or 21. 10 participants, or 56%, had previous exposure to map, filter or reduce in languages like Python, Racket, ML and Scheme while the remaining 8 users, or 44%, did not.

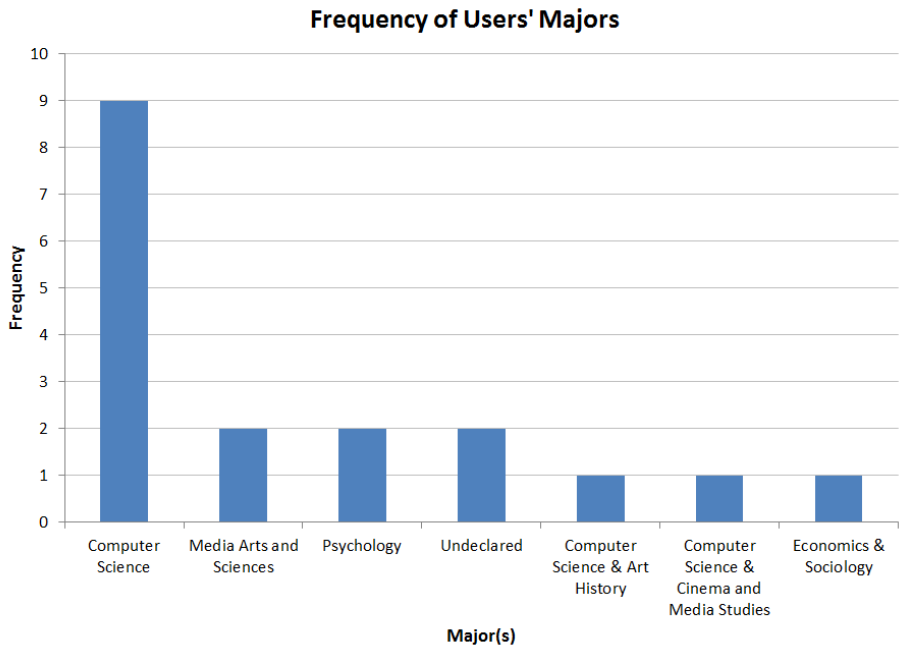

Figure 5-1: Bar graph of users' majors

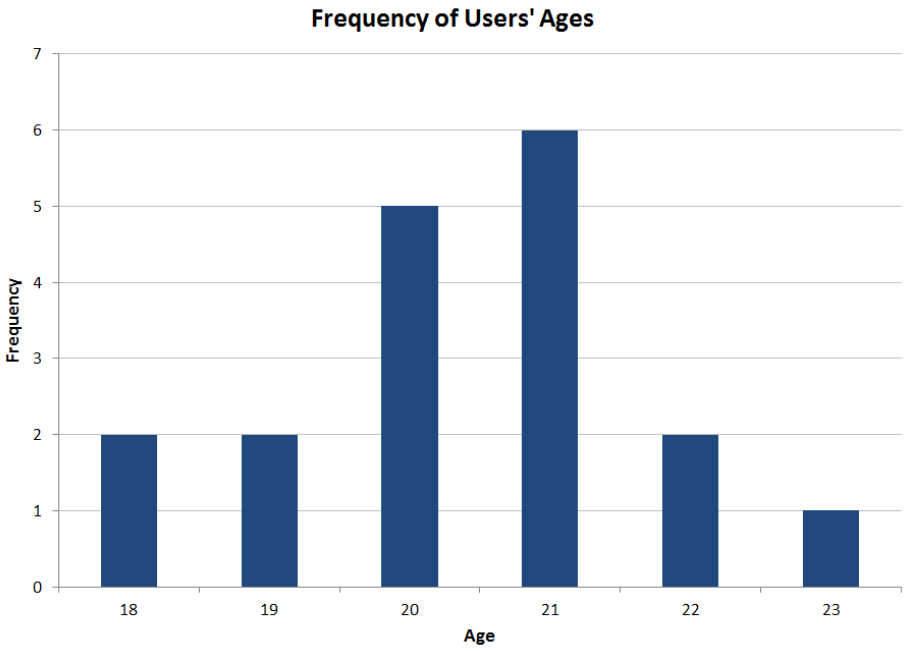

Figure 5-2: Bar graph of users' ages

5.3 Results

5.3.1 Map, Filter and Reduce

In order to control for level of skill and programming background, I divided the users into two groups: Group 1, consisting of students who had previous exposure to map, filter or reduce, and Group 2,

consisting of students who had no previous exposure to any of these three operators. There were 10 and 8 students in each group, respectively.

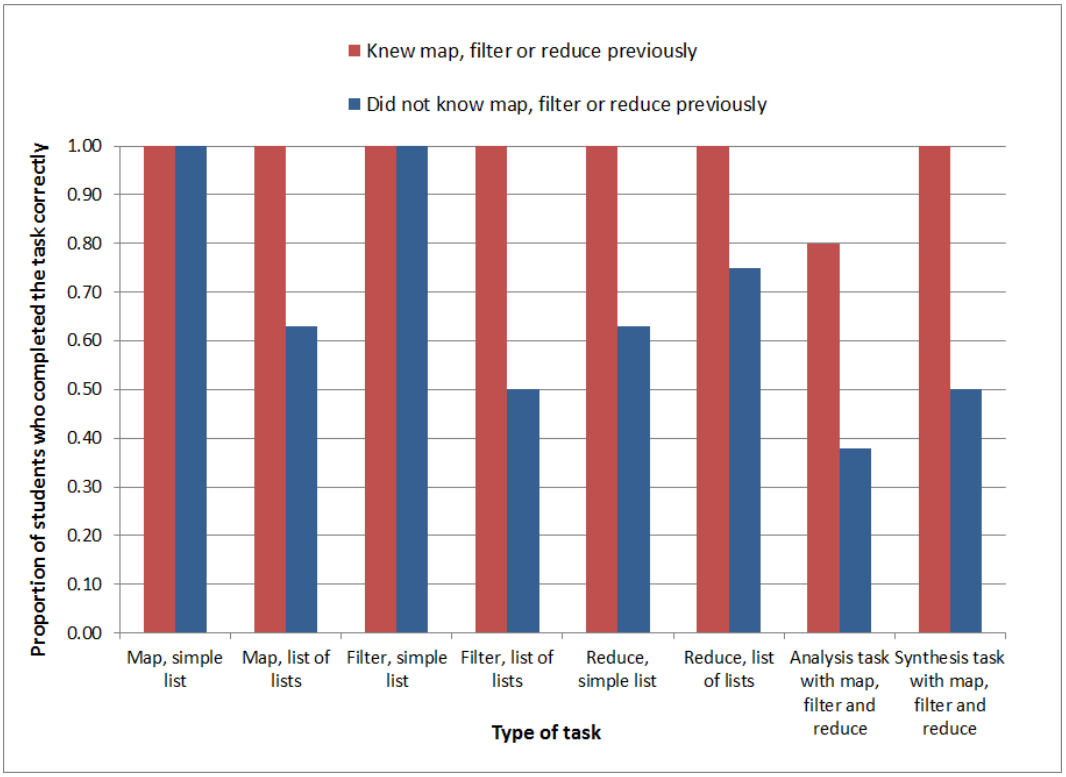

Figure 5-3: Comparison of Group 1's and Group 2's performance on map, filter and reduce portion of the study

For all eight tasks, Group 1 had a higher proportion of students who completed the task correctly than Group 2. As a group, they had an average success rate of 98% on each task, compared to 67% success rate for Group 2. The main purpose of dividing the users into two groups, however, is not to compare their performances; such a comparison only shows the effect of previous knowledge on a user's performance. Rather, I wanted to examine each group independently to see how well they can use these new blocks in App Inventor.

All users in Group 1 successfully completed at least 7 tasks out of the 8 total. The only task which participants found challenging was the analysis task in which they had to explain the meaning of a program that filtered, mapped and then reduced over a list of lists. This task had a success rate of 80%, as some users missed or misinterpreted one step along the way. However, all users correctly completed the synthesis task where they had to write a program that used a combination of these blocks. This suggests that most users in this group knew how to use the map, filter and reduce blocks individually, as well as together with the other blocks. Several users also commented in the

post task survey that working with these operators previously in programming languages helped them in their understanding of the corresponding blocks in App Inventor.

In Group 2, all users successfully mapped and filtered over a simple list. However, they particularly struggled with the concept of reducing, using any of these operators on a list of lists and finally using these operators in combination with one another. Reduce introduces the idea of continuously updating the accumulating answer while iterating through each list item, which may be a challenging concept for beginner programmers. Moreover, some users said that it was difficult to keep track of the functionality of each block and the differences between the blocks after a brief tutorial. This confusion may have contributed to the 38% and 50% success rate on the analysis and synthesis tasks, respectively, in which users either explained the meaning or wrote programs that used a combination of these blocks.

There were two common problems that users in both groups faced. First, 6 users (60%) in Group 1 and 8 users (100%) in Group 2 had trouble selecting the correct item of each sublist when mapping, filtering or reducing over a list of lists. Overall, 14 users (78%) faced this problem. For these tasks, I provided a list of lists called `studentList` in which every item is also a list containing the age and name of a student. The user's first task with the `studentList` was to use the map block to return a new list with only the student names. The correct solution is shown in Figure 5-4. The map block takes in the `studentList` as the input list and the second index of each item as the body expression; this is because item is a list containing a student's age in the first index and his or her name in the second index. This block returns a list that results from mapping each item in `studentList` to the the student's name.

Some mistakes that users made during this task are shown in Figures 5-5 through 5-8. In both Figures 5-5 and 5-6, users manipulated the global `studentList` instead of item in the body expression of the map block. Figure 5-5 maps each item to the second index of the `studentList`. Figure 5-6 uses two select list item blocks to map each item to the name of the first student in `studentList`.

In Figures 5-7 and 5-8, users incorrectly manipulate item in the body expression. Figure 5-7 attempts to use two select list item blocks on item when item is a list, not a list of lists. Users who came up with the solution in Figure 5-8 understood that `studentList` is a list of lists and used two select list item blocks to try to access the student names. However, the select list item block takes in a list and an index, and users incorrectly used item as an index to this block. These mistakes identify two common points of confusion: what item refers to in a list of lists, and how to use the map block to iterate through the items of a list of lists. The fact that a high percentage of users struggled with mapping, filtering or reducing over a list of lists suggests that a tutorial on this topic

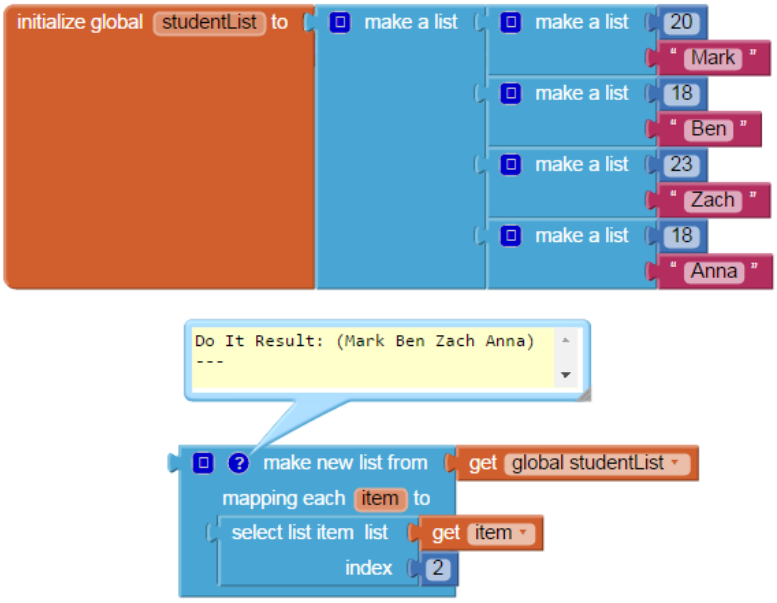

Figure 5-4: `studentList` is a list of lists containing the age and name of each student. The map block returns a new list with only the student names.

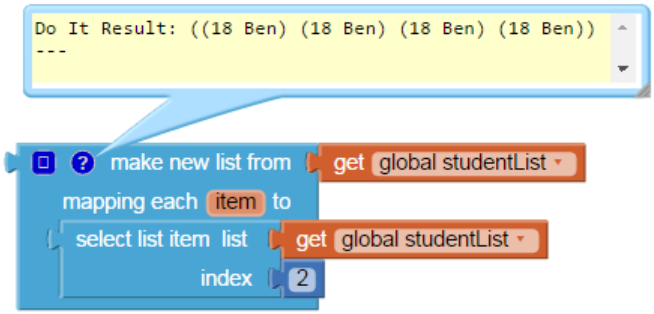

Figure 5-5: Maps each item to [18, "Ben"], the second item in `studentList`

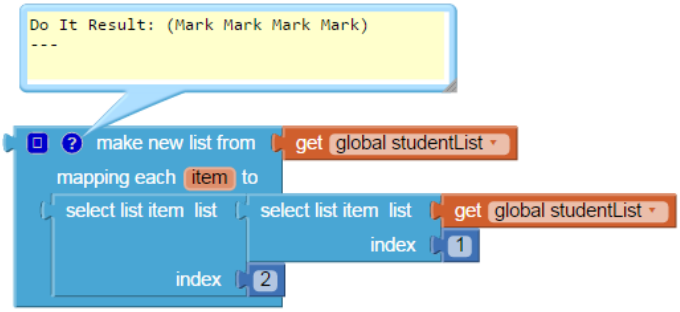

Figure 5-6: Maps each item to "Mark", the name of the first student in `studentList`

would be helpful.

Figure 5-7: Returns an error because it is not possible to use two select list item blocks on item, which is a list, not a list of lists

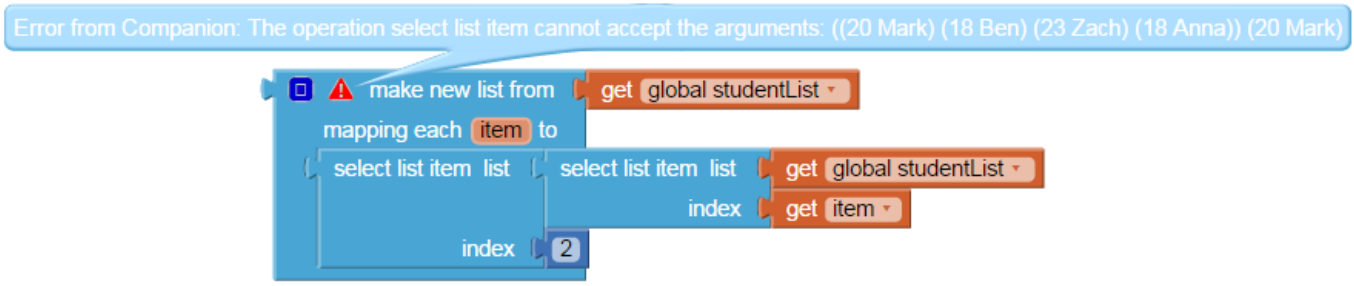

Figure 5-8: Returns an error because item, a list, is used as the index of one of the select list item blocks

While the most common problem was manipulating a list of lists with these new operators, the second most common mistake that users made occurred when concatenating strings using reduce. 6 users (60%) in Group 1 and 6 users (75%) in Group 2 concatenated the accumulating answer–`answerSoFar`–to each list `item` instead of concatenating each list `item` to `answerSoFar`. This resulted in a string that was in reverse order. Overall, 12 users (67%) made this mistake, but most realized that the resulting string was backwards and switched the ordering of `item` and `answerSoFar` in the join block.

The reduce block documentation therefore needs to be clear that this operator iterates through the list from left to right; that is, it begins accumulating the answer from the beginning of the list. Some students may be familiar with foldl and foldr in OCaml. While foldl walks through the list from left to right, foldr walks from right to left. foldr is more common than foldl in OCaml, so students who know this language will most likely expect the list items to be processed in the opposite order of reduce, which is equivalent to foldl.

The mapping, filtering and reducing tasks revealed other less common but still important prob-

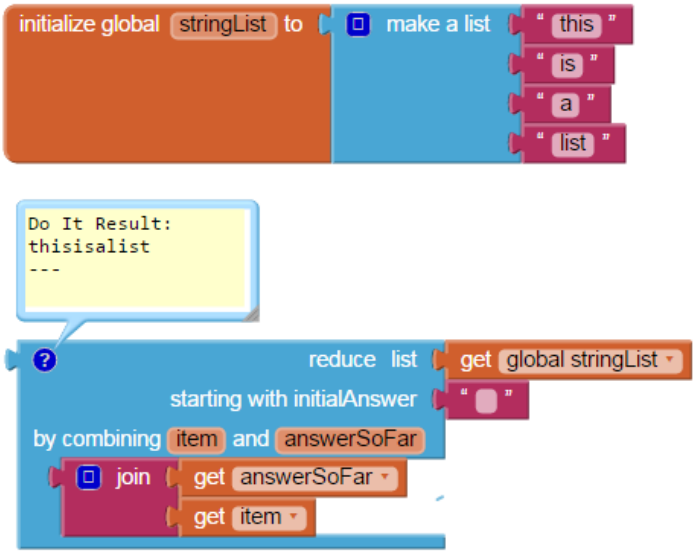

Figure 5-9: The correct way to concatenate strings in a list using reduce

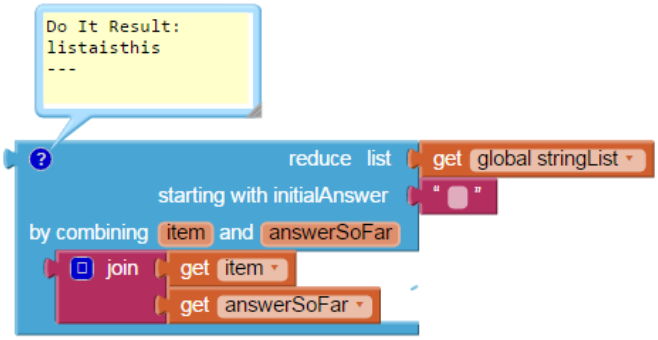

Figure 5-10: An incorrect way to concatenate strings in a list using reduce

lems. Three users, all from Group 2, placed the map block inside a for loop because they did not understand that the map block iterates through all the items of the input list. Others faced type problems when using the reduce block. For example, five users, all from Group 2, used 0 as the initial answer when the result has to be a string. Three users, again from Group 2, used the mathematical + operator instead of the join block to concatenate strings.

5.3.2 Sort

When analyzing the results from the sorting portion of the study, I again divided the users into students who knew map, filter or reduce previously (Group 1) and student who did not (Group 2). Once again, Group 1 performed just as well or better than Group 2 on all the tasks. Overall, the

results show that a large majority of users in both groups were able to sort the given list in ascending or descending order by the element itself or by one key.

However, sorting with two keys proved particularly challenging. 8 users (80%) in Group 1 and 4 users (50%) in Group 2 correctly completed this type of sorting task, but these results are in part due to flaws in the design of the study. Both the basic sort and the sort with key use a default less-than-or-equal comparator and are stable algorithms; they preserve the order of two equal items in the input list. During the tutorial of these blocks, I failed to mention that in order to make the sort with comparator stable, two items in the list must be compared using $\leq$ or $\geq$. Users thus used $<$ or $>$ to compare two items, leading to an unstable sorting algorithm. This became problematic during two user studies when users used two sort with comparator blocks to first sort by the secondary key and then by the primary key. This method correctly sorts the list with two keys but only if the sorting algorithm is stable. Another flaw in the design of the study is that the list of lists I asked users to sort only contained five items. Therefore, users were sometimes able to use an incorrect method to a return a correctly sorted list.

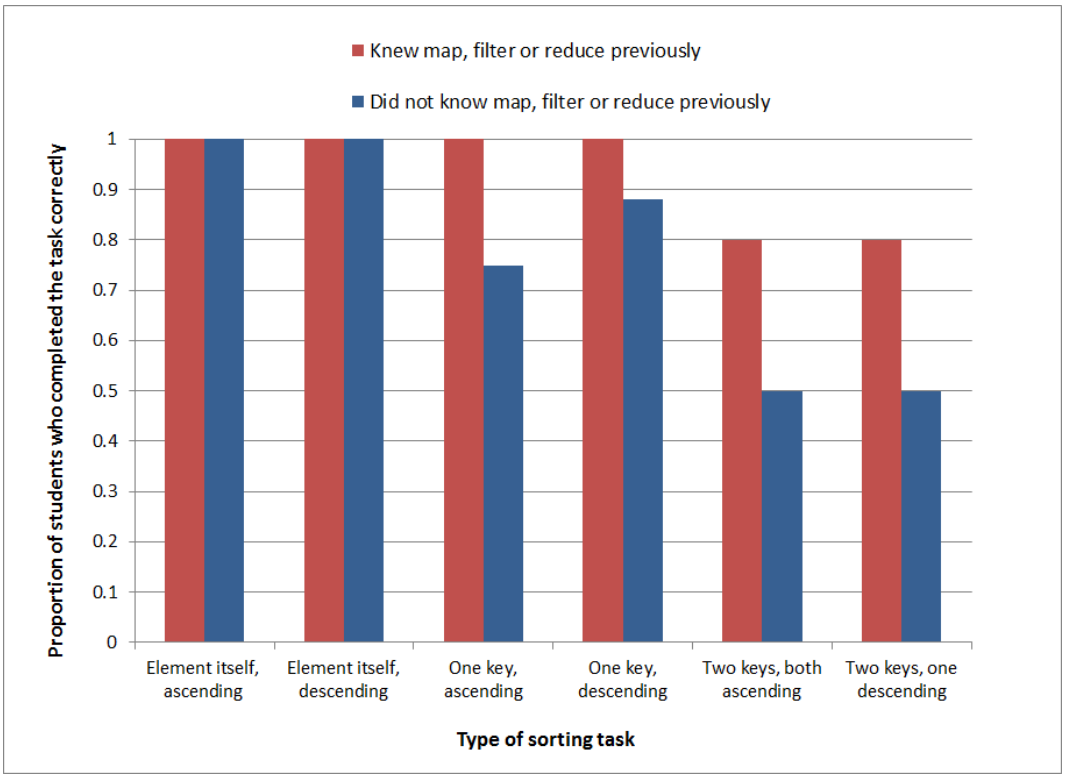

Figure 5-11: Comparison of Group 1's and Group 2's performance on the sorting portion of the study

In addition to analyzing the success rate of each task for each group, I also examined users' sort

operator of choice for the three tasks that have more than one solution. Any of the three blocks can sort the items of a list of lists in ascending order, but all users used the basic sort for this task. They said that they chose the basic sort because I had explained in a previous example that it sorts by the first index of each sublist in a list of lists. Others reasoned that it was simple and easy to use. Among those who correctly sorted the given list of lists with one ascending key, 7 users (44%) used sort with key while 9 users (56%) used sort with comparator. This suggests that users may be slightly more inclined to use the sort with comparator block. Some commented that they found the comparator more intuitive than the key because they think of sorting as comparing a pair of items in a list. The comparator allows them to visualize how the list is being sorted, unlike key which is more abstract. One user mistakenly thought that the key block changes the input list by mapping each item to its key value. Finally, for tasks that involve sorting with two ascending keys, users had several options. Among those who successfully completed this task, 3 users (25%) used a combination of the basic sort and sort with key, which are both stable. 9 users (75%) used two stable sort with comparators or a single sort with comparator with conditional statements to determine the sorting order in the different cases. No user used a key that is a list containing the primary and secondary keys, as this is a difficult solution to develop without seeing an example beforehand.

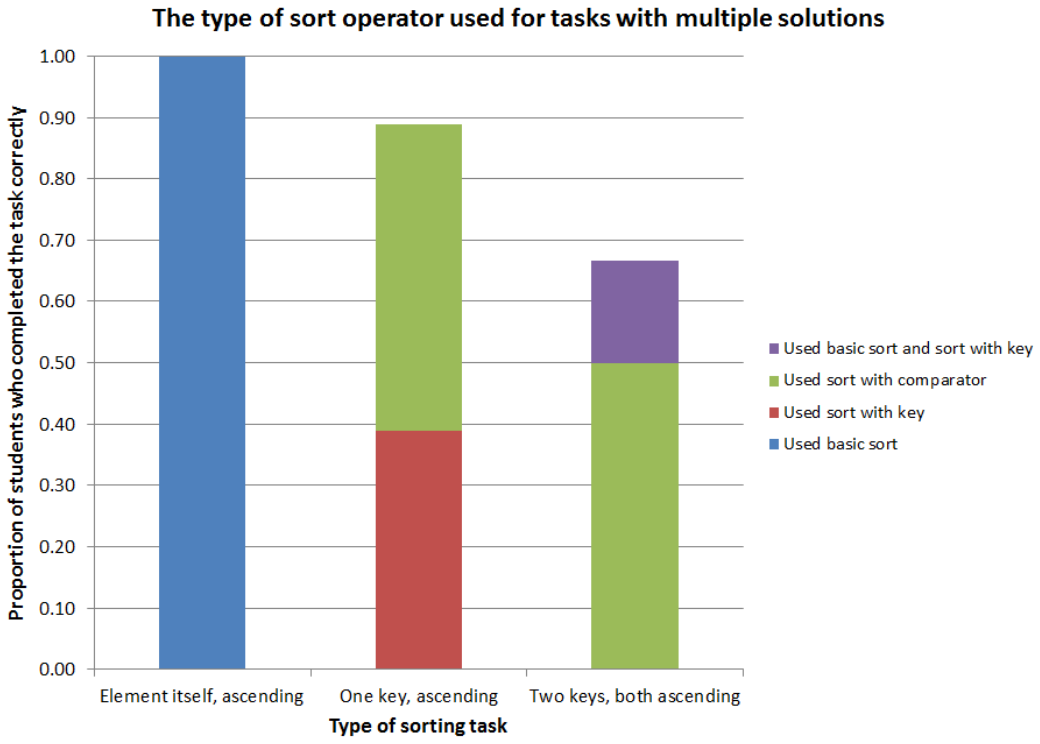

Figure 5-12: A bar graph of users' preferred sort operators for tasks that have multiple solutions

Similar to the mapping, filtering and reducing portion of the study, there were two common problems that users encountered during the sorting tasks. The first is that when using the sort with comparator, a total of 9 users (50%) tried to use the boolean operator `item1 > item2` to sort the list in increasing order or the boolean operator `item1 < item2` to sort the list in decreasing order. 6 users (60%) in Group 1 and 3 users (38%) in Group 2 made this mistake. This confusion may be clarified by explaining that `item1` will be placed before `item2` in the output list only if the boolean operator returns true. Otherwise, `item2` is placed before `item1`. The documentation should also include examples of sorting in increasing and decreasing order using the comparator, as many users will likely sort in one of these two ways.

In addition to confusing which boolean operators sort in increasing and decreasing order, 3 users (30%) in Group 1 and 3 users (38%) in Group 2 tried to use the sort with key (which sorts in increasing order only) for tasks that required sorting in decreasing order. In total, 6 users, or 33%, made this mistake, which suggests that the uses of each sort block should be clearly defined for the user.

Another less common problem was that two users, both from Group 2, the basic sort to sort by a particular index of each sublist in a list of lists. For example, as shown in Figure 5-13, I had provided a list of lists called `AllTextbooks` in which each item is a list containing the name, price and quantity of the textbook(s) the user had to buy for each subject. I asked users to sort in increasing order by quantity, the third index of each sublist. One solution that users wrote was using the basic sort to sort the third index of the global `AllTextbooks` list. Since numbers are considered smaller than strings by the basic sort, the resulting answer is the list [2, 35, "Chemistry"].

Finally, six users (four users from Group 1 and two users from Group 2) encountered type problems when using the sort with comparator, confusing the inequality block in the math drawer with the compare text block in the text drawer, or vice versa.

5.3.3 Summary of Key Points

Based on the data from the user study, I created a summary that addresses some of the common points of confusion. The key points are organized into categories below:

1. All operators
    - All the new list operators that I have added iterate through the items of the input list. Therefore, there is no need to use these blocks inside a for loop.

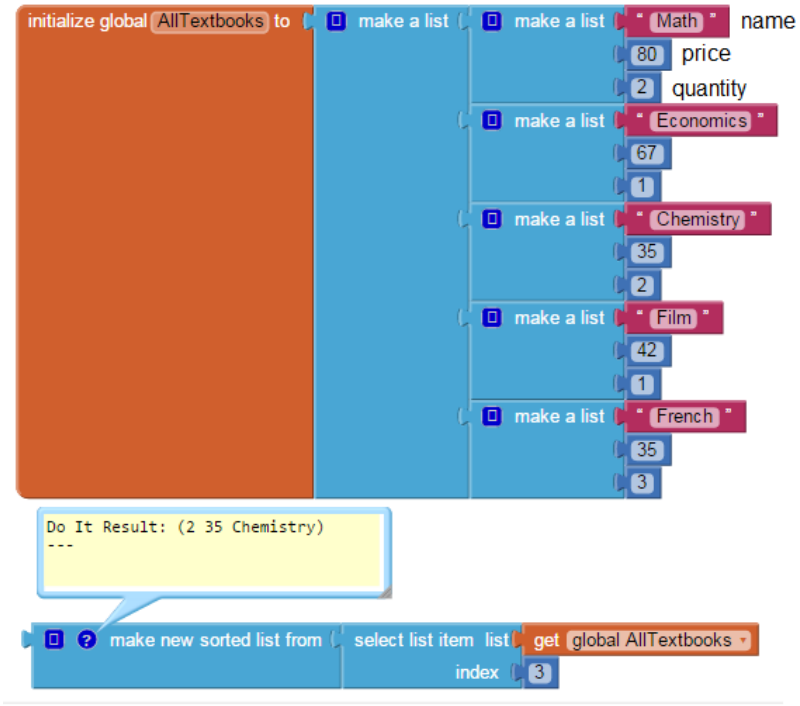

Figure 5-13: Users attempted to use the basic sort to sort the list in ascending order by quantity. The basic sort, however, cannot iterate through a particular index of each sublist in a list of lists.

- In order to iterate through the items of the given list, the body expressions of map, filter, reduce, sort with key and sort with comparator must manipulate the parameters built into these blocks.
- If the input list is a list of lists, each item in the list is a list. Therefore, a select list item block is needed to index into a particular item of each sublist.

2. Reduce

   - The reduce block traverses the input list from left to right.
   - The type of the `initialAnswer` has to always match the type of the value that needs to be returned. For example, if the result has to be a string, the `initialAnswer` needs to be a string as well.

3. General notes about implementation and uses of three sorting blocks

   - Basic sort and sort with key use a default less-than-or-equal operator and are stable algorithms. In order to make sort with comparator stable, users must use $\leq$ or $\geq$ to compare two list items. In future work, however, more research should be done to determine if

a less-than or a less-than-or-equal operator should be used to make a sorting algorithm stable.

- Basic sort and sort with key can only sort in increasing order. In order to sort in decreasing order, users must use the sort with comparator. The uses of the three sorting blocks are summarized in Table 3.1.

4. Basic sort

   - Basic sort takes in a list as its sole input; it cannot index into a particular index of a sublist in a list of lists.

5. Sort with key

   - Sort with key does not change the input list by mapping each item to its key value. It simply uses a list of proxy values returned by the key to sort the input list.
   - In order to sort by more than one key, users must define a list containing the primary key followed by all secondary keys and input this list as the key expression.

6. Sort with comparator

   - In order to sort the list items in increasing order with the comparator, users must use the boolean operator `item1` $<$ `item2`. In order to sort in decreasing order, users must use the boolean operator `item1` $>$ `item2`.
   - The compare text block is used to compare strings, while the mathematical inequality signs are used for numerical comparisons.

5.4 Feedback

App Inventor aims to lower the barrier to programming, and by providing these blocks, I had hoped to abstract over the complexities associated with these higher-order operators. The results and feedback from the user study indicate that several improvements could be made upon these blocks to strengthen users' understanding of the blocks.

5.4.1 Map, Filter and Reduce

In the post task survey, many users commented that the brief tutorial and simple examples of each block were helpful. Students who had worked with these three operators previously said they felt

more comfortable using the blocks because of their previous knowledge. For example, one user said: "I have worked with map, filter, and reduce a lot in different languages, so the concepts were familiar and I was able to interpret pretty quickly what parts the blocks should have."

For students who had no previous exposure to map, filter or reduce, the main source of difficulty was keeping track of the functionality of each block and the differences between the blocks. For example, one user said: "These blocks were fairly simple to use, but I sometimes became frustrated because I would forget which block was useful in what kind of scenario. Reading the English on the blocks also helped with this though when I would get stuck."

One user liked the naming of the variable `answerSoFar` in reduce and the `test` label in filter, while another user was confused about what `item` and `answerSoFar` refer to. This feedback emphasizes the importance of clear, precise labeling on the blocks, especially for beginning programmers who do not know these three operators from other programming languages. One user said that while filtering was intuitive, mapping and reducing were more difficult to understand. Therefore, it is especially crucial to use approachable syntax on the map and reduce blocks. Further studies can be conducted to test which labels are most helpful to users in understanding the functionality of each block.

5.4.2 Sort

Overall, users found the basic sort and sort with comparator easy and intuitive to use. Three users, however, said they did not completely understand how and when to use sort with key and asked for more concrete examples involving key.

Users were divided about the idea of having three different sort blocks, as shown below through three user quotes:

- "I didn't like that there were three different blocks for three different kinds of sorting...I almost think it would be easier if you had to explicitly decide how you want a list to be sorted every time you want to sort a list."
- "I liked that there were three options, so I could use the one I felt most comfortable with."
- "There are multiple ways you can perform a single task, especially with these three specific sort blocks. That made it both easier (can use any) and more difficult (many options for how to execute) to use."

I believe that having three sorting blocks is beneficial if the uses of each block is specified in

the documentation, as in Table 3.1. Python supports a basic sort and sort with key, but it is also important to provide a sort with comparator because many users in this study found it clear and instructive. They liked that they were able see how a pair of items in the list are compared through the comparator.

Chapter 6

Implementation

I implemented the new blocks from scratch as baked-in language features rather than creating a function in App Inventor and wrapping it in a procedure block. The latter option is currently not possible, as App Inventor does not support first-class functions.

In this chapter, I will discuss the implementation of the higher-order operators using reduce and the basic sort as examples. I will also present the modifications required to implement the mutator for the map block.

6.1 Higher-order Operators

6.1.1 Reduce

I first described the reduce block in the JavaScript-based Blockly framework for App Inventor. I made the following specifications:

1. The block is an expression that evaluates to and returns a value.
2. The block has two built-in parameters: `item`, which refers to the current list item that is being iterated over and `answerSoFar`, which refers to the accumulating answer.
3. The block has three inputs: a list, an initial value which can be of any type, and a body expression that combines the current list item with the accumulating answer.

Listing 6.1: Describing the reduce block in JavaScript

```
Blockly.Blocks['lists_reduce'] = {
  // For each loop.
```

```
  category : 'Lists',
  helpUrl : Blockly.Msg.LANG_LISTS_REDUCE_HELPURL,
  init : function() {
    this.setColour(Blockly.LIST_CATEGORY_HUE);
    this.appendValueInput('LIST')
        .setCheck(Blockly.Blocks.Utilities.YailTypeToBlocklyType("list",Blockly.Blocks.Utilities.INPUT))
        .appendField(Blockly.Msg.LANG_LISTS_REDUCE_TITLE_REDUCE)
        .appendField(Blockly.Msg.LANG_LISTS_REDUCE_INPUT_INLIST)
        .setAlign(Blockly.ALIGN_RIGHT);
    this.appendValueInput('INITANSWER')
        .appendField(Blockly.Msg.LANG_LISTS_REDUCE_INPUT_INITIAL_ANSWER)
        .setAlign(Blockly.ALIGN_RIGHT);
    this.appendDummyInput('DESCRIPTION')
        .appendField(Blockly.Msg.LANG_LISTS_REDUCE_INPUT_COMBINE)
        .appendField(new
            Blockly.FieldParameterFlydown(Blockly.Msg.LANG_LISTS_REDUCE_INPUT_VAR,
                                          true, // name is editable
                                          Blockly.FieldFlydown.DISPLAY_BELOW),
                    'VAR1')
        .appendField(Blockly.Msg.LANG_LISTS_REDUCE_INPUT_AND)
        .appendField(new
            Blockly.FieldParameterFlydown(Blockly.Msg.LANG_LISTS_REDUCE_INPUT_ANSWER,
                                          true, // name is editable
                                          Blockly.FieldFlydown.DISPLAY_BELOW),
                    'VAR2')
        .setAlign(Blockly.ALIGN_RIGHT);
    this.appendIndentedValueInput('COMBINE');
    this.setOutput(true, null);
    this.setTooltip(Blockly.Msg.LANG_LISTS_REDUCE_TOOLTIP);
  },
  getVars: function() {
    var names = []
    names.push(this.getFieldValue('VAR1'));
    names.push(this.getFieldValue('VAR2'));
    return names;
  },
  blocksInScope: function() {
    var combineBlock = this.getInputTargetBlock('COMBINE');
    if (combineBlock) {
      return [combineBlock];
    } else {
      return [];
    }
  },
  declaredNames: function() {
    return this.getVars();
  },
  renameVar: function(oldName, newName) {
    if (Blockly.Names.equals(oldName, this.getFieldValue('VAR1'))) {
      this.setFieldValue(newName, 'VAR1');
    }
    if (Blockly.Names.equals(oldName, this.getFieldValue('VAR2'))) {
      this.setFieldValue(newName, 'VAR2');
    }
  },
  typeblock: [{ translatedName: Blockly.Msg.LANG_LISTS_REDUCE_TITLE_REDUCE }]
};
```

In addition to giving the reduce block its shape, I specified the labels that are indicative of the block's functionality and inputs. The label on the reduce block reads: `reduce list by starting with initialAnswer by combining item and answerSoFar`.

Listing 6.2: Labeling the reduce block

```
Blockly.Msg.LANG_LISTS_REDUCE_HELPURL =
    'http://appinventor.mit.edu/explore/ai2/support/blocks/lists#reduce';
  Blockly.Msg.LANG_LISTS_REDUCE_TITLE_REDUCE = 'reduce';
  Blockly.Msg.LANG_LISTS_REDUCE_INPUT_INLIST = 'list';
  Blockly.Msg.LANG_LISTS_REDUCE_INPUT_INITIAL_ANSWER = 'starting with initialAnswer';
  Blockly.Msg.LANG_LISTS_REDUCE_INPUT_COMBINE = 'by combining';
  Blockly.Msg.LANG_LISTS_REDUCE_INPUT_VAR = 'item ';
  Blockly.Msg.LANG_LISTS_REDUCE_INPUT_AND = 'and';
  Blockly.Msg.LANG_LISTS_REDUCE_INPUT_ANSWER = 'answerSoFar';
  Blockly.Msg.LANG_LISTS_REDUCE_INPUT_COLLAPSED_TEXT = 'reduce list';
  Blockly.Msg.LANG_LISTS_REDUCE_TOOLTIP = 'Reduces the list to a single value by
      combining the list elements'
     +  'using the block in the body section. If the list is empty, initialAnswer is
         returned. Otherwise, '
     + 'the function is first applied to initialAnswer and the first list item, and then
         on the accumulating answer'
     + 'and the next list item. This continues until the end of the list.';
```

The reduce block also has a YAIL (Young Android Intermediate Language) associated with it that gives the block its functionality. YAIL is generated by taking the inputs the user provides to the block and passing them as parameters to a Kawa procedure that reduces the list.

Listing 6.3: Generating YAIL for the reduce block

```
Blockly.Yail['lists_reduce'] = function() {
  // For each loop.
  var emptyListCode = Blockly.Yail.YAIL_CALL_YAIL_PRIMITIVE + "make-yail-list" +
      Blockly.Yail.YAIL_SPACER;
  emptyListCode += Blockly.Yail.YAIL_OPEN_COMBINATION + Blockly.Yail.YAIL_LIST_CONSTRUCTOR
      + Blockly.Yail.YAIL_SPACER;
  emptyListCode += Blockly.Yail.YAIL_CLOSE_COMBINATION + Blockly.Yail.YAIL_SPACER +
      Blockly.Yail.YAIL_QUOTE + Blockly.Yail.YAIL_OPEN_COMBINATION;
  emptyListCode += Blockly.Yail.YAIL_CLOSE_COMBINATION;
  emptyListCode += Blockly.Yail.YAIL_SPACER + Blockly.Yail.YAIL_DOUBLE_QUOTE + "make a
      list" + Blockly.Yail.YAIL_DOUBLE_QUOTE + Blockly.Yail.YAIL_CLOSE_COMBINATION;
  var loopIndexName1 = Blockly.Yail.YAIL_LOCAL_VAR_TAG + this.getFieldValue('VAR1');
  var loopIndexName2 = Blockly.Yail.YAIL_LOCAL_VAR_TAG + this.getFieldValue('VAR2');
  var listCode = Blockly.Yail.valueToCode(this, 'LIST', Blockly.Yail.ORDER_NONE) ||
      emptyListCode;
  var initAnswerCode = Blockly.Yail.valueToCode(this, 'INITANSWER',
      Blockly.Yail.ORDER_NONE);
  var bodyCode = Blockly.Yail.valueToCode(this, 'COMBINE', Blockly.Yail.ORDER_NONE) ||
      Blockly.Yail.YAIL_FALSE;
  var code = Blockly.Yail.YAIL_REDUCE + initAnswerCode + Blockly.Yail.YAIL_SPACER +
      loopIndexName2 + Blockly.Yail.YAIL_SPACER
          + loopIndexName1 + Blockly.Yail.YAIL_SPACER + bodyCode +
              Blockly.Yail.YAIL_SPACER
          + listCode + Blockly.Yail.YAIL_CLOSE_COMBINATION;
   return [ code, Blockly.Yail.ORDER_ATOMIC ];
```

```
};
```

Kawa is a version of Scheme that runs on the Java-based Android system and is the environment where I specified the behavior of the reduce block. Since Scheme does not have a built-in reduce function, I defined my own recursive procedure `yail-list-reduce` that accumulates the values in the input list by traversing the list from left to right. App Inventor uses YAIL lists that consists of a header followed by the contents of the list. After verifying that the user provided a list as the argument to the reduce block, I took the tail of the YAIL list to obtain its contents in the form a Scheme list. I performed my Scheme procedure `yail-list-reduce` on the contents and returned the resulting value.

Listing 6.4: Kawa code that reduces a list

```
(define (yail-list-reduce ans binop yail-list)
  (define (reduce accum func lst)
      (cond ((null? lst) accum)
            (else (reduce (func accum (car lst)) func (cdr lst)))))
  (let ((verified-list (coerce-to-yail-list yail-list)))
    (if (eq? verified-list *non-coercible-value*)
        (signal-runtime-error
         (format #f
                 "The second argument to reduce is not a list. The second argument is: ~A"
                 (get-display-representation yail-list))
         "Bad list argument to reduce")
        (reduce ans binop (yail-list-contents verified-list)))))
```

6.1.2 Basic sort

The basic sort uses a default less-than-or-equal comparator that is defined on any two App Inventor values. As the first step, I defined a list `typeordering` which specifies the order of the different App Inventor types in increasing order. The current order of the types is booleans, numbers, texts, lists and then components. To compare items of the input list, I created a less-than-or-equal operator that takes in two values, `val1` and `val2`. It returns true in two cases: 1) the type of `val1` is less than the type of `val2` according to the list `typeordering` or 2) `val1` and `val2` both have the same type and `val1` is less than `val2` in that type. For booleans, I chose false to be less than true. Strings and numbers are compared using the standard lexicographical and numerical order, respectively. If the two items are lists, they are recursively compared starting from the first item in each list. Finally, components are first compared using their class names. If they are instances of the same class, they are compared using their hashcodes.

I then implemented a mergesort algorithm which takes in two inputs: a list and the less-than-

or-equal comparator that I defined. Mergesort divides the input list into two lists of equal or almost equal length, recursively sorts each sublist, and merges the two sorted sublists together in the order defined by the comparator. My algorithm is stable; if two items have the same value, they maintain their relative position in the output list.

Listing 6.5: Kawa code that implements the default comparator used by basic sort

```
;;Implements a generic sorting procedure that works on lists of any type.

(define typeordering '(boolean number text list component))

(define (typeof val)
   (cond ((boolean? val) 'boolean)
         ((number? val) 'number)
         ((string? val) 'text)
         ((yail-list? val) 'list)
         ((instance? val com.google.appinventor.components.runtime.Component) 'component)
         (else (signal-runtime-error
               (format #f
                         "typeof called with unexpected value: ~A"
                         (get-display-representation val))
                 "Bad arguement to typeof"))))

(define (indexof element list)
   (list-index (lambda (x) (eq? x element)) list))

(define (type-lt? type1 type2)
   (< (indexof type1 typeordering)
      (indexof type2 typeordering)))
(define (is-leq? val1 val2)
    (let ((type1 (typeof val1))
          (type2 (typeof val2)))
      (or (type-lt? type1 type2)
          (and (eq? type1 type2)
               (cond ((eq? type1 'boolean) (boolean-leq? val1 val2))
                     ((eq? type1 'number) (<= val1 val2))
                     ((eq? type1 'text) (string<=? val1 val2))
                     ((eq? type1 'list) (list-leq? val1 val2))
                     ((eq? type1 'component) (component-leq? val1 val2))
                     (else (signal-runtime-error
                       (format #f
                              "(isleq? ~A ~A)"
                                  (get-display-representation val1)
                                  (get-display-representation val2))
                         "Shouldn't happen")))))))

;;false is less than true
(define (boolean-leq? val1 val2)
   (not (and val1 (not val2))))

;;throw exception is not yail-list
(define (yail-list-necessary y1)
  (cond ((yail-list? y1) (yail-list-contents y1))
        (else y1)))

(define (list-leq? y1 y2)
```

```
  (define (helper-list-leq? lst1 lst2)
    (cond ((and (null? lst1) (null? lst2)) #t)
          ((null? lst1) #t)
          ((null? lst2) #f)
          ((is-lt? (car lst1) (car lst2)) #t)
          ((is-eq? (car lst1) (car lst2)) (helper-list-leq? (cdr lst1) (cdr lst2)))
          (else #f)))
  (helper-list-leq? (yail-list-necessary y1) (yail-list-necessary y2)))

;;Component are first compared using their class names. If they are instances of the same
    class,
;;then they are compared using their hashcodes.
(define (component-leq? comp1 comp2)
   (or (string<? (*:getSimpleName (*:getClass comp1))
                 (*:getSimpleName (*:getClass comp2)))
       (and (string=? (*:getSimpleName (*:getClass comp1))
                      (*:getSimpleName (*:getClass comp2)))
            (<= (*:hashCode comp1)
                (*:hashCode comp2)))))

(define (merge lessthan? lst1 lst2)
   (cond ((null? lst1) lst2)
         ((null? lst2) lst1)
         ((lessthan? (car lst1) (car lst2)) (cons (car lst1) (merge lessthan? (cdr lst1)
             lst2)))
         (else (cons (car lst2) (merge lessthan? lst1 (cdr lst2))))))

(define (mergesort lessthan? lst)
   (cond ((null? lst) lst)
         ((null? (cdr lst)) lst)
         (else (merge lessthan? (mergesort lessthan? (take lst (quotient (length lst) 2)))
                        (mergesort lessthan? (drop lst (quotient (length lst) 2)))))))

(define (yail-list-sort y1)
   (cond ((yail-list-empty? y1) (make YailList))
         ((not (pair? y1)) y1)
         (else (kawa-list->yail-list (mergesort is-leq? (yail-list-contents y1))))))
```

6.2 Destructive vs. Nondestructive Mechanism

In order to keep track of the state of the map mutator, I created a new field called `FieldRadioButton` that is inherited from `FieldCheckbox`. `lists_mutatorcontainer`, the block inside the bubble of the mutator, has two radio buttons: one for changing the existing list and another for making a new list. These two radio buttons belong to one `RadioButtonGroup`. Only button in the group can be selected at one time; if a new button is selected, the previously selected button is automatically deselected.

Listing 6.6: Adding radio buttons to the block inside the bubble of the mutator

```
Blockly.Blocks['lists_mutatorcontainer'] = {
  init: function() {
```

```
    this.setColour(Blockly.LIST_CATEGORY_HUE);
    var group = new Blockly.RadioButtonGroup();
    this.appendDummyInput()
        .appendField(new Blockly.FieldRadioButton(group), 'CHANGE_LIST')
        .appendField("changes existing list");
    this.appendDummyInput()
        .appendField(new Blockly.FieldRadioButton(group), 'MAKE_NEW_LIST')
        .appendField("makes new list");
    this.contextMenu = false;
  }
};
```

I then modified four different functions in the map operator to ensure that the block structure is updated based on which radio button is selected:

1. decompose: uses the given block structure to construct a mutator
2. compose: uses the given state of the mutator to construct the block structure
3. mutatationToDom: uses the given state of a block structure to create the XML representation for the mutator
4. domToMutation: uses the given XML representation of the mutator to create the block structure

When a different radio button is selected, these four functions are called to change the block and its XML representation from an expression to a statement or a statement to an expression.

Listing 6.7: Modifying the description of map block to include the mutator

```
Blockly.Blocks['lists_map'] = {
  // For each loop.
  category : 'Lists',
  helpUrl : Blockly.Msg.LANG_LISTS_MAP_HELPURL,
  init : function() {
    this.setColour(Blockly.LIST_CATEGORY_HUE);
    this.appendValueInput('LIST')
       .setCheck(Blockly.Blocks.Utilities.YailTypeToBlocklyType("list",Blockly.Blocks.Utilities.INPUT))
       .appendField(Blockly.Msg.LANG_LISTS_MAP_DEST_TITLE_MAP, 'TITLE')
       .setAlign(Blockly.ALIGN_RIGHT);
    this.appendDummyInput('DESCRIPTION')
        .appendField(Blockly.Msg.LANG_LISTS_MAP_INPUT_ITEM)
        .appendField(new Blockly.FieldParameterFlydown(Blockly.Msg.LANG_LISTS_MAP_INPUT_VAR,
                                                       true, // name is editable
                                                       Blockly.FieldFlydown.DISPLAY_BELOW),
                     'VAR')
        .appendField(Blockly.Msg.LANG_LISTS_MAP_INPUT_TO)
        .setAlign(Blockly.ALIGN_RIGHT);
    this.appendIndentedValueInput('TO');
    this.setPreviousStatement(true);
    this.setNextStatement(true);
    this.setTooltip(Blockly.Msg.LANG_LISTS_MAP_TOOLTIP);
```

```
  this.setMutator(new Blockly.Mutator([]));
  this.changeList = true;
},
// onchange: Blockly.WarningHandler.checkErrors,
updateBlock_: function() {
  if (this.changeList) {
    if (this.outputConnection && this.outputConnection.targetConnection) {
      this.outputConnection.disconnect();

      var children = this.parentBlock_.childBlocks_;
      for (var child, x = 0; child = children[x]; x++) {
        if (child == this) {
          children.splice(x, 1);
          break;
        }
      }
      this.setParent(null);
      this.parentBlock_ = null;
    }

    if (this.outputConnection) {
      if (this.outputConnection.inDB_) {
        this.outputConnection.dbList_[this.outputConnection.type].removeConnection_(this.outputConnection);
      }
      this.outputConnection.inDB_ = false;
      if (Blockly.highlightedConnection_ == this.outputConnection) {
        Blockly.highlightedConnection_ = null;
      }
      if (Blockly.localConnection_ == this.outputConnection) {
        Blockly.localConnection_ = null;
      }
    }

    this.outputConnection = null;
    this.setFieldValue(Blockly.Msg.LANG_LISTS_MAP_DEST_TITLE_MAP, 'TITLE');
    this.setPreviousStatement(true);
    this.setNextStatement(true);
    this.previousConnection.dbList_[this.previousConnection.type].addConnection_(this.previousConnection);
    this.nextConnection.dbList_[this.nextConnection.type].addConnection_(this.nextConnection);
    this.render();

  } else {
    if (this.previousConnection && this.previousConnection.targetConnection) {
      this.previousConnection.disconnect();

      var children = this.parentBlock_.childBlocks_;
      for (var child, x = 0; child = children[x]; x++) {
        if (child == this) {
          children.splice(x, 1);
          break;
        }
      }
      this.setParent(null);
      this.parentBlock_ = null;
   }

    if (this.previousConnection) {
      if (this.previousConnection.inDB_) {
        this.previousConnection.dbList_[this.previousConnection.type]
```

```
        removeConnection_(this.previousConnection);
      }
      this.previousConnection.inDB_ = false;
      if (Blockly.highlightedConnection_ == this.previousConnection) {
        Blockly.highlightedConnection_ = null;
      }
      if (Blockly.localConnection_ == this.previousConnection) {
        Blockly.localConnection_ = null;
      }
    }

    if (this.nextConnection) {
      if (this.nextConnection.inDB_) {
        this.nextConnection.dbList_[this.nextConnection.type].removeConnection_(this.nextConnection);
      }
      this.nextConnection.inDB_ = false;
      if (Blockly.highlightedConnection_ == this.nextConnection) {
        Blockly.highlightedConnection_ = null;
      }
      if (Blockly.localConnection_ == this.nextConnection) {
        Blockly.localConnection_ = null;
      }
      if (Blockly.localConnection_ == this.nextConnection) {
        Blockly.localConnection_ = null;
      }
    }

    this.previousConnection = null;
    this.nextConnection = null;
    this.setFieldValue(Blockly.Msg.LANG_LISTS_MAP_NONDEST_TITLE_MAP, 'TITLE');
    this.setOutput(true, null);
    this.outputConnection.dbList_[this.outputConnection.type].addConnection_(this.outputConnection);
    this.render();
   }
},
mutationToDom: function() {
  var container = document.createElement('mutation');
  if (! this.changeList) {
    container.setAttribute('destructive', this.changeList);
  }
  return container;
},
domToMutation: function(xmlElement) {
  if(!xmlElement.getAttribute('destructive')){
    this.changeList = true;
  } else {
    this.changeList = (xmlElement.getAttribute('destructive') == "true");
  }
  this.updateBlock_();
},
decompose: function(workspace) {
  var containerBlock = Blockly.Block.obtain(workspace,'lists_mutatorcontainer');
  containerBlock.initSvg();
  var changeListButton = containerBlock.getField_('CHANGE_LIST');
  var makeNewListButton = containerBlock.getField_('MAKE_NEW_LIST');
  var group = changeListButton.group;
  if (this.changeList) {
    group.setSelected(changeListButton);
  } else {
```

```
      group.setSelected(makeNewListButton);
    }
    return containerBlock;
  },
  compose: function(containerBlock) {
   this.oldChangeList = this.changeList;
    this.changeList = containerBlock.getFieldValue('CHANGE_LIST') == 'TRUE' ? true : false;
    if (this.oldChangeList != this.changeList) {
      this.updateBlock_();
    }
  },
  saveConnections: Blockly.saveConnections,
  getVars: function() {
    return [this.getFieldValue('VAR')];
  },
  blocksInScope: function() {
    var toBlock = this.getInputTargetBlock('TO');
    if (toBlock) {
      return [toBlock];
    } else {
      return [];
    }
  },
  declaredNames: function() {
    return [this.getFieldValue('VAR')];
  },
  renameVar: function(oldName, newName) {
    if (Blockly.Names.equals(oldName, this.getFieldValue('VAR'))) {
      this.setFieldValue(newName, 'VAR');
    }
  },
  typeblock: [{ translatedName: Blockly.Msg.LANG_LISTS_MAP_INPUT_COLLAPSED_TEXT }]
};
```

Chapter 7

Conclusion

7.1 Summary of Work

In App Inventor, loops are used for list processing involving mapping, filtering, reducing or sorting. To help users manipulate lists more easily, I added map, filter, reduce and sort operators. Many of these new operations have one or more parameters to refer to items in the list. There are three different types of sort. The first sorts the input list in increasing order using a default comparator defined on any two App Inventor values. The second sorts the input list in increasing order using a default comparator and allows the user to specify a key, or the criteria by which she would like to sort the list. Finally, the third sorts the input list using a comparator that the user provides. With the addition of these new operators, users do not have to define complicated loops and initialize relevant variables each time they would like to iterate through a list.

Another limitation relating to App Inventor lists is that some list operators change the input list and are destructive while others return a new list and are nondestructive. In order to give users greater flexibility when processing lists, I have developed a mechanism where users can switch the state of an operator from destructive to nondestructive, or vice versa. The default state is currently destructive, as App Inventor lists resemble Python lists that support destructive operations. This new mechanism means users no longer have to write additional code to convert from one version of an operator to another.

Following the implementation of the map, filter, reduce and sort operators, I conducted a user study to identify improvements that could be made upon these operators. 18 Wellesley students with previous knowledge of App Inventor participated in the study. While most users were able to

map, filter or reduce over a simple list, they struggled to use these operators, both individually and together, when given a list of lists. Sorting with two keys was also a common problem that users faced. The gap in understanding how to process lists of lists and sort a list given two keys suggests that users could benefit from tutorials on these two topics.

7.2 Future Work

7.2.1 Design of the Blocks

As the next step in the design of these new list operators, I would like to finalize the labels to be displayed on each block. The labels must clearly and precisely capture the functionality of each operator. This will be an iterative process that involves gathering feedback from members of the App Inventor developer team as well as the general community of users.

7.2.2 Additional User Studies

In a future study, it may be interesting to randomize the order of the mapping, filtering and reducing portion and the sorting portion. The study I conducted started with the mapping, filtering and reducing tasks, so users had already worked with higher-order operators and list of lists by the time they reached the sorting tasks. Randomizing the order will ensure that the difference in a user's performance between the two portions of the study can be attributed to their understanding of the operators themselves rather than external factors.

Another extension of the study I conducted is to compare the usability of loops to the usability of these new higher-order operators. I added map, filter and reduce in order to provide users with another, perhaps easier, means to iterate through a list other than through loops. By asking users to complete the same task first using loops and then using map, filter and reduce, I can compare how well they can use these two different groups of iterative operators. Such a comparison can made through several measures: the amount of time it takes to complete the task, the number of mistakes users make, and the number of times users test their code while using loops and while using the higher-order operators.

Furthermore, it is important to also test the mutators that allows users to switch between a destructive and nondestructive version of each operator. Destructive operators are statements that perform an action while nondestructive operators are expressions which return a value. App Inventor makes the distinction between statements and expressions through the shapes of the blocks. While

statement blocks have a notch and do not return anything, expression blocks have a plug on the left side to return a value. A future study can examine how a block's shape and labels affect a user's understanding of the block's type and the destructive vs. nondestructive mechanism.

Finally, testing a wider range of users will be helpful in analyzing the usability of these new list operator blocks as well as the destructive vs. nondestructive mechanism. In my study, I targeted students who have previous experience working with App Inventor, but another group of students to test are those who know Python but have not worked with App Inventor. Since App Inventor lists resemble Python lists, I am interested to see how easy or difficult it is to use one's knowledge of Python to understand list operators in App Inventor. Furthermore, I only studied Wellesley students, but having a study with a broader demographic base in terms of sex and age will be useful in gaining insights into the usability of these list operators.

Bibliography

[Ai2] MIT Center for Mobile Learning, App Inventor 2 home page, http://appinventor.mit.edu, accessed April 22, 2015.

[Dut13] Sajal Dutta. *List Sorting on App Inventor*. Imaginity. 2013. URL: http://www.imagnity.com/tutorials/app-inventor/list-sorting-on-app-inventor/.

[FTP14] F. Martin D. Wolber F. Turbak M. Sherman and S. C. Pokress. "Events-first programming in App Inventor". In: *Journal of Computing Sciences in Colleges* 29.6 (2014), pp. 81–89.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. Cambridge, Massachusetts: The MIT Press, 1986.

[Poi14] Tutorials Point. *Python Lists*. http://www.tutorialspoint.com/python/python_lists.html. 2014.

[Scr] Scratch project, MIT Lifelong Kindergarten Group, http://scratch.mit.edu/, accessed April 22, 2015.

[Shi99] Olin Shivers. *SRFI 1: List Library*. http://srfi.schemers.org/srfi-1/srfi-1.html. 1999.

[Sna] Snap! project, University of California, Berkeley, https://snap.berkeley.edu/, accessed April 22, 2015.

[Tai15] Taifun. *App Inventor Code Snippets*. Pure Vida Apps. 2015. URL: http://puravidaapps.com/snippets.php.

[W3S15] W3Schools. *JavaScript Arrays*. http://www.w3schools.com/js/js_arrays.asp. 2015.

Appendix A

User Study Protocol

A.1 Protocol

My project centers around extending the implementation of App Inventor to include new blocks that map, filter, reduce, and sort lists.

The study will consist of tasks to test the usability of these new list operator blocks and will be screencast. You will begin by filling out a consent form and a short questionnaire. Next, I will introduce my new list operator blocks to you in the context of some simple App Inventor programs. I will then ask you to explain the meaning of programs that use these blocks, as well as write new programs that use these blocks. There will also be a post-task survey.

The study will take about 90 minutes. At any point, you may terminate the study. You will receive a $15 gift certificate for your participation regardless of whether or not you finish.

A.2 Part 1: Map, Filter and Reduce

A.2.1 Map

The map block has two expression sockets: the first specifies an input list and the second specifies a body expression for each item in the input list. The block returns a new list that is the same length as the input list in which each element is the result of evaluating the body expression for each corresponding item in the input list.

Task 1 Return a new list in which every element of the input list is doubled.

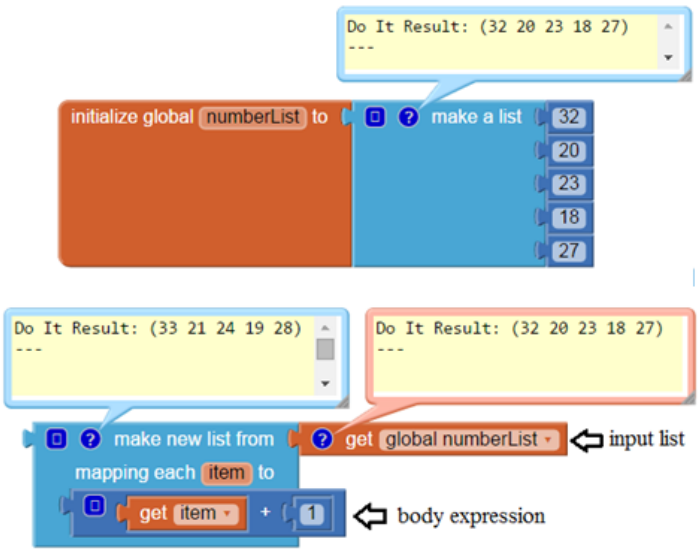

Figure A-1: Map example that returns a new list in which every element of the input list is incremented by 1.

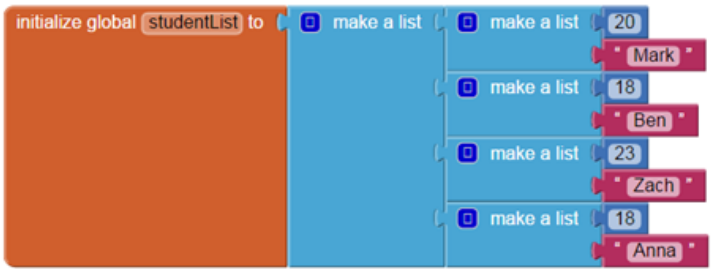

Figure A-2: Each sublist in the list contains the student's age and name.

Task 2 studentList is a list of lists—each item in the list is also a list containing information about the student's age and name. Using studentList as the input list, return a new list with only the student names.

A.2.2 Filter

The filter block has two expression sockets: the first specifies an input list and the second specifies a boolean expression. The block returns a new list that only includes the items of the input list for which the boolean expression returns true.

Task 3 Return a new list by keeping only the items of the input list that are less than 25.

Task 4 Using studentList, return a new list by keeping only the students whose age is less than 19.

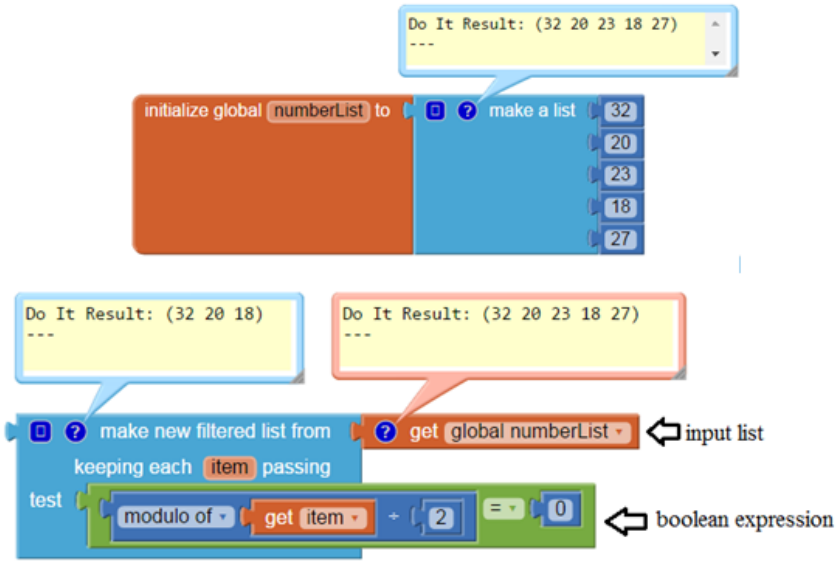

Figure A-3: Filter example that returns a new list by keeping only the items of the input list that are even.

A.2.3 Reduce

The reduce block combines a list to a single value. It has three expression sockets: an input list, an initial value and a body expression. The reduce block combines each item of the input list with the answerSoFar using the body expression in order to calculate the final answer, which it returns.

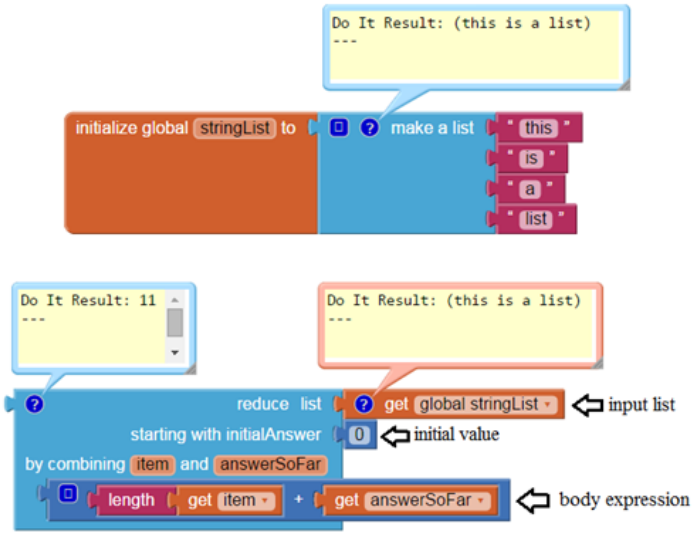

Figure A-4: Reduce example that counts the total number of characters in the list.

Task 5 From the list, create the string "thisisalist."

Task 6 Using studentList, return the sum of all the ages of the students in the list.

A.2.4 Analysis

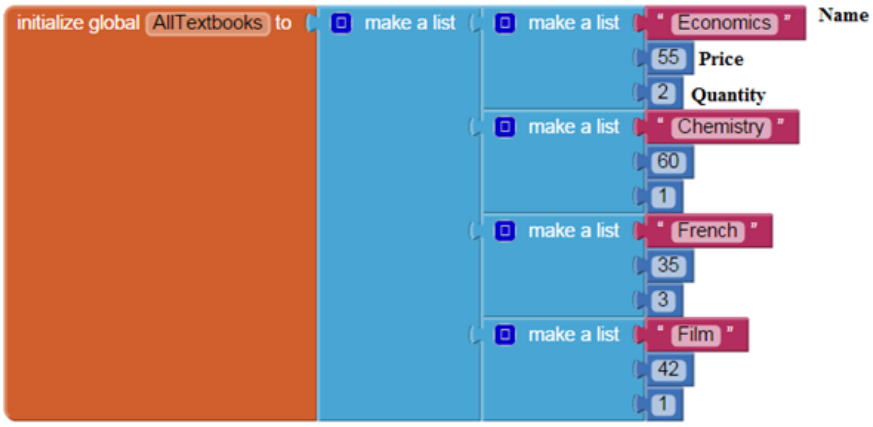

Figure A-5: Each sublist in the list contains the name, price and quantity of the textbooks required for each subject.

Task 7 AllTextbooks is a list of lists in which each sublist contains the name, price and quantity of textbooks that the student has to buy for each subject. As a side note, concatenating "$\backslash n$" to a string creates a newline, as shown in Figure A-6. What do the blocks in Figure A-7 do?

```
>>> s = "Economics" + "\n" + "Chemistry"
>>> print s
Economics
Chemistry
```

Figure A-6: Concatenating "$\backslash n$" to a string creates a newline.

A.2.5 Synthesis

Task 8: You have a budget of $100 for each subject. That is, you're not willing to pay more than $100 for textbooks for a single subject. Using map, filter, and reduce, sum up the cost of textbooks for subjects that don't go over your budget.

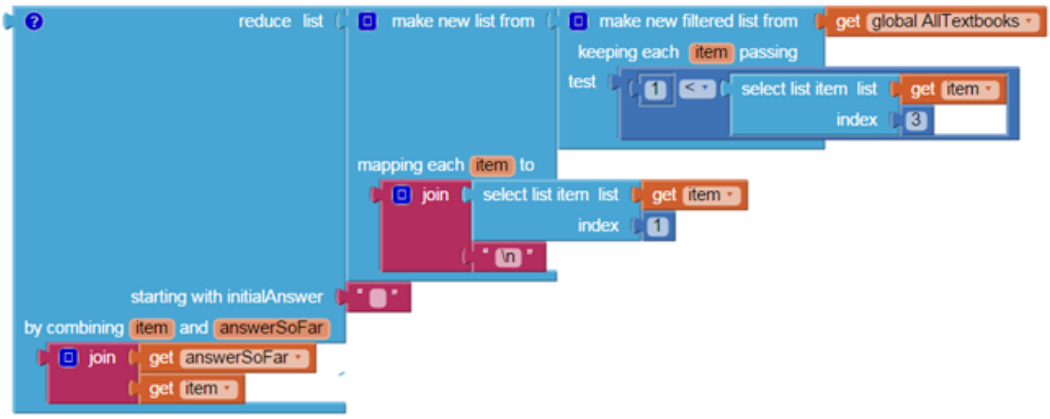

Figure A-7: Task 7

A.3 Part 2: Sort

A.3.1 Basic sort

This sorting block returns a new list whose elements are those of the input list sorted in increasing order. It uses a default comparator defined on any two App Inventor values. It groups items of the same type together, and then sorts accordingly within the same type group. The current order of the types is booleans, numbers, strings, lists and then components.

Figure A-8: Basic sort block that sorts the input list alphabetically

A.3.2 Sort with key

This sorting block has two expression sockets: an input list and a key expression. It returns a new output list whose elements are those of the input list sorted in increasing order determined by using a default comparator on the result of the key expression applied to each item. If two items have the same key, they maintain their relative position in the output list.

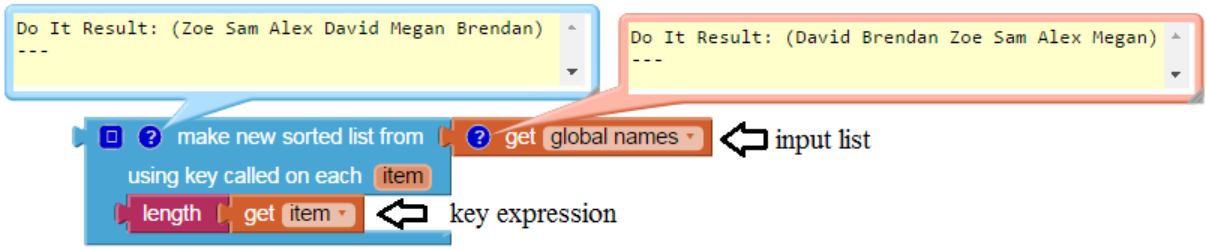

Figure A-9: Sort with key block that sorts the input list in increasing order by the length of student names

A.3.3 Sort with comparator

This sorting block has two expression sockets: an input list and a boolean expression. It returns a new output list whose elements are those of the input list sorted according to a comparator defined by a boolean expression that determines if item1 is less than or equal to item2.

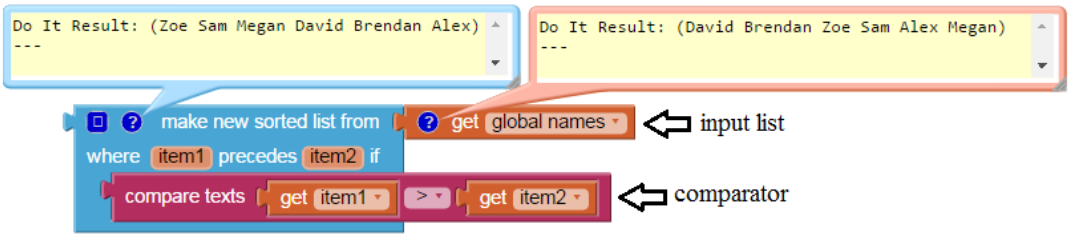

Figure A-10: Sort with comparator block that sorts the input list in reverse alphabetical order

A.3.4 Basic sort with a list of lists

This is a list of lists in which each list item is a list containing the person's age and name. The default comparator used by the basic sort block compares the first item in each list item. In this case, the basic sort block arranges the list items in ascending order by age.

Figure A-11: Given a list of lists, the basic sort operator sorts by the first index of each sublist.

A.3.5 Tasks

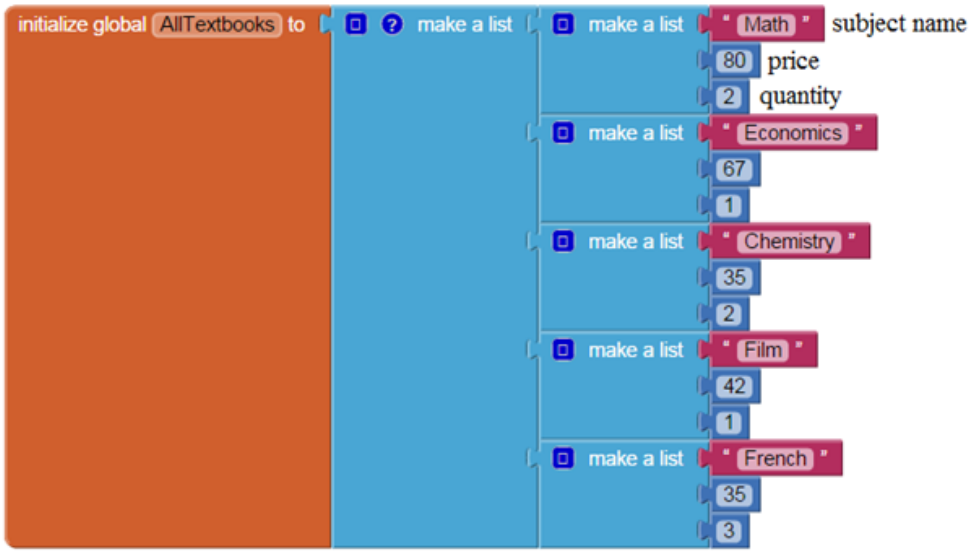

Figure A-12: AllTextbooks list

Task 9 Sort list in ascending order by subject name.

Task 10 Sort list in descending order by subject name.

Task 11 Sort list in ascending order by quantity; if two elements have the same quantity, keep their same relative order as in the input list.

Task 12 For each subject, you will be spending a certain amount of money to buy textbooks - let's

call this subject cost. Sort list in descending order by subject cost.

Task 13 Sort list in ascending order by quantity; if two textbooks have the same quantity, sort in ascending order by name.

Task 14 Sort list in ascending order by price; if two textbooks have the same price, sort in descending order by quantity.

Appendix B

lists.js

```
Blockly.Blocks['lists_map'] = {
  // For each loop.
  category : 'Lists',
  helpUrl : Blockly.Msg.LANG_LISTS_MAP_HELPURL,
  init : function() {
    this.setColour(Blockly.LIST_CATEGORY_HUE);
    this.appendValueInput('LIST')
       .setCheck(Blockly.Blocks.Utilities.YailTypeToBlocklyType("list",Blockly.Blocks.Utilities.INPUT))
       .appendField(Blockly.Msg.LANG_LISTS_MAP_DEST_TITLE_MAP, 'TITLE')
       .setAlign(Blockly.ALIGN_RIGHT);
    this.appendDummyInput('DESCRIPTION')
        .appendField(Blockly.Msg.LANG_LISTS_MAP_INPUT_ITEM)
        .appendField(new Blockly.FieldParameterFlydown(Blockly.Msg.LANG_LISTS_MAP_INPUT_VAR,
                                                       true, // name is editable
                                                       Blockly.FieldFlydown.DISPLAY_BELOW),
                     'VAR')
        .appendField(Blockly.Msg.LANG_LISTS_MAP_INPUT_TO)
        .setAlign(Blockly.ALIGN_RIGHT);
    this.appendIndentedValueInput('TO');
    this.setPreviousStatement(true);
    this.setNextStatement(true);
    this.setTooltip(Blockly.Msg.LANG_LISTS_MAP_TOOLTIP);
    this.setMutator(new Blockly.Mutator([]));
    this.changeList = true;
  },
  // onchange: Blockly.WarningHandler.checkErrors,
  updateBlock_: function() {
    if (this.changeList) {
      if (this.outputConnection && this.outputConnection.targetConnection) {
        this.outputConnection.disconnect();

        var children = this.parentBlock_.childBlocks_;
        for (var child, x = 0; child = children[x]; x++) {
          if (child == this) {
            children.splice(x, 1);
            break;
          }
        }
```

```
    this.setParent(null);
    this.parentBlock_ = null;
  }

  if (this.outputConnection) {
    if (this.outputConnection.inDB_) {
      this.outputConnection.dbList_[this.outputConnection.type].removeConnection_(this.outputConnection);
    }
    this.outputConnection.inDB_ = false;
    if (Blockly.highlightedConnection_ == this.outputConnection) {
      Blockly.highlightedConnection_ = null;
    }
    if (Blockly.localConnection_ == this.outputConnection) {
      Blockly.localConnection_ = null;
    }
  }

  this.outputConnection = null;
  this.setFieldValue(Blockly.Msg.LANG_LISTS_MAP_DEST_TITLE_MAP, 'TITLE');
  this.setPreviousStatement(true);
  this.setNextStatement(true);
  this.previousConnection.dbList_[this.previousConnection.type].addConnection_(this.previousConnection);
  this.nextConnection.dbList_[this.nextConnection.type].addConnection_(this.nextConnection);
  this.render();

} else {
  if (this.previousConnection && this.previousConnection.targetConnection) {
    this.previousConnection.disconnect();

    var children = this.parentBlock_.childBlocks_;
    for (var child, x = 0; child = children[x]; x++) {
      if (child == this) {
        children.splice(x, 1);
        break;
      }
    }
    this.setParent(null);
    this.parentBlock_ = null;
 }

  if (this.previousConnection) {
    if (this.previousConnection.inDB_) {
      this.previousConnection.dbList_[this.previousConnection.type]
      .removeConnection_(this.previousConnection);
    }
    this.previousConnection.inDB_ = false;
    if (Blockly.highlightedConnection_ == this.previousConnection) {
      Blockly.highlightedConnection_ = null;
    }
    if (Blockly.localConnection_ == this.previousConnection) {
      Blockly.localConnection_ = null;
    }
  }

  if (this.nextConnection) {
    if (this.nextConnection.inDB_) {
      this.nextConnection.dbList_[this.nextConnection.type].
      removeConnection_(this.nextConnection);
    }
```

```
      this.nextConnection.inDB_ = false;
      if (Blockly.highlightedConnection_ == this.nextConnection) {
        Blockly.highlightedConnection_ = null;
      }
      if (Blockly.localConnection_ == this.nextConnection) {
        Blockly.localConnection_ = null;
      }
      if (Blockly.localConnection_ == this.nextConnection) {
        Blockly.localConnection_ = null;
      }
    }

    this.previousConnection = null;
    this.nextConnection = null;
    this.setFieldValue(Blockly.Msg.LANG_LISTS_MAP_NONDEST_TITLE_MAP, 'TITLE');
    this.setOutput(true, null);
    this.outputConnection.dbList_[this.outputConnection.type].
    addConnection_(this.outputConnection);
    this.render();
   }
},
mutationToDom: function() {
  var container = document.createElement('mutation');
  if (! this.changeList) {
    container.setAttribute('destructive', this.changeList);
  }
  return container;
},
domToMutation: function(xmlElement) {
  if(!xmlElement.getAttribute('destructive')){
    this.changeList = true;
  } else {
    this.changeList = (xmlElement.getAttribute('destructive') == "true");
  }
  this.updateBlock_();
},
decompose: function(workspace) {
  var containerBlock = Blockly.Block.obtain(workspace,'lists_mutatorcontainer');
  containerBlock.initSvg();
  var changeListButton = containerBlock.getField_('CHANGE_LIST');
  var makeNewListButton = containerBlock.getField_('MAKE_NEW_LIST');
  var group = changeListButton.group;
  if (this.changeList) {
    group.setSelected(changeListButton);
  } else {
    group.setSelected(makeNewListButton);
  }
  return containerBlock;
},
compose: function(containerBlock) {
 this.oldChangeList = this.changeList;
  this.changeList = containerBlock.getFieldValue('CHANGE_LIST') == 'TRUE' ? true : false;
  if (this.oldChangeList != this.changeList) {
    this.updateBlock_();
  }
},
saveConnections: Blockly.saveConnections,
getVars: function() {
  return [this.getFieldValue('VAR')];
```

```
  },
  blocksInScope: function() {
    var toBlock = this.getInputTargetBlock('TO');
    if (toBlock) {
      return [toBlock];
    } else {
      return [];
    }
  },
  declaredNames: function() {
    return [this.getFieldValue('VAR')];
  },
  renameVar: function(oldName, newName) {
    if (Blockly.Names.equals(oldName, this.getFieldValue('VAR'))) {
      this.setFieldValue(newName, 'VAR');
    }
  },
  typeblock: [{ translatedName: Blockly.Msg.LANG_LISTS_MAP_INPUT_COLLAPSED_TEXT }]
};


Blockly.Blocks['lists_filter'] = {
  // For each loop.
  category : 'Lists',
  helpUrl : Blockly.Msg.LANG_LISTS_FILTER_HELPURL,
  init : function() {
    this.setColour(Blockly.LIST_CATEGORY_HUE);
    this.appendValueInput('LIST')
        .setCheck(Blockly.Blocks.Utilities.YailTypeToBlocklyType("list",Blockly.Blocks.Utilities.INPUT))
        .appendField(Blockly.Msg.LANG_LISTS_FILTER_DEST_TITLE_FILTER, 'TITLE')
        .setAlign(Blockly.ALIGN_RIGHT);
    this.appendDummyInput('DESCRIPTION')
        .appendField(Blockly.Msg.LANG_LISTS_FILTER_INPUT_ITEM)
        .appendField(new
            Blockly.FieldParameterFlydown(Blockly.Msg.LANG_LISTS_FILTER_INPUT_VAR,
                                          true, // name is editable
                                          Blockly.FieldFlydown.DISPLAY_BELOW),
                    'VAR')
      .appendField(Blockly.Msg.LANG_LISTS_FILTER_INPUT_PASSING)
      .setAlign(Blockly.ALIGN_RIGHT);
    this.appendIndentedValueInput('TEST')
        .appendField(Blockly.Msg.LANG_LISTS_FILTER_INPUT_TEST);
    this.setPreviousStatement(true);
    this.setNextStatement(true);
    this.setMutator(new Blockly.Mutator([]));
    this.setTooltip(Blockly.Msg.LANG_LISTS_FILTER_TOOLTIP);
    this.changeList = true;
  },
  //  onchange: Blockly.WarningHandler.checkErrors,
//onchange: Blockly.WarningHandler.checkErrors,
  updateBlock_: function() {
    if (this.changeList) {
      if (this.outputConnection && this.outputConnection.targetConnection) {
        this.outputConnection.disconnect();

        var children = this.parentBlock_.childBlocks_;
        for (var child, x = 0; child = children[x]; x++) {
          if (child == this) {
            children.splice(x, 1);
```

```
        break;
      }
    }
    this.setParent(null);
    this.parentBlock_ = null;
  }

  if (this.outputConnection) {
    if (this.outputConnection.inDB_) {
      this.outputConnection.dbList_[this.outputConnection.type].
      removeConnection_(this.outputConnection);
    }
    this.outputConnection.inDB_ = false;
    if (Blockly.highlightedConnection_ == this.outputConnection) {
      Blockly.highlightedConnection_ = null;
    }
    if (Blockly.localConnection_ == this.outputConnection) {
      Blockly.localConnection_ = null;
    }
  }

  this.outputConnection = null;
  this.setFieldValue(Blockly.Msg.LANG_LISTS_FILTER_DEST_TITLE_FILTER, 'TITLE');
  this.setPreviousStatement(true);
  this.setNextStatement(true);
  this.previousConnection.dbList_[this.previousConnection.type].addConnection_(this.previousConnection);
  this.nextConnection.dbList_[this.nextConnection.type].addConnection_(this.nextConnection);
  this.render();

} else {
  if (this.previousConnection && this.previousConnection.targetConnection) {
    this.previousConnection.disconnect();

    var children = this.parentBlock_.childBlocks_;
    for (var child, x = 0; child = children[x]; x++) {
      if (child == this) {
        children.splice(x, 1);
        break;
      }
    }
    this.setParent(null);
    this.parentBlock_ = null;
 }

  if (this.previousConnection) {
    if (this.previousConnection.inDB_) {
      this.previousConnection.dbList_[this.previousConnection.type].
      removeConnection_(this.previousConnection);
    }
    this.previousConnection.inDB_ = false;
    if (Blockly.highlightedConnection_ == this.previousConnection) {
      Blockly.highlightedConnection_ = null;
    }
    if (Blockly.localConnection_ == this.previousConnection) {
      Blockly.localConnection_ = null;
    }
  }

  if (this.nextConnection) {
```

```
      if (this.nextConnection.inDB_) {
        this.nextConnection.dbList_[this.nextConnection.type].removeConnection_(this.nextConnection);
      }
      this.nextConnection.inDB_ = false;
      if (Blockly.highlightedConnection_ == this.nextConnection) {
        Blockly.highlightedConnection_ = null;
      }
      if (Blockly.localConnection_ == this.nextConnection) {
        Blockly.localConnection_ = null;
      }
      if (Blockly.localConnection_ == this.nextConnection) {
        Blockly.localConnection_ = null;
      }
    }

    this.previousConnection = null;
    this.nextConnection = null;
    this.setFieldValue(Blockly.Msg.LANG_LISTS_FILTER_NONDEST_TITLE_FILTER, 'TITLE');
    this.setOutput(true, null);
    this.outputConnection.dbList_[this.outputConnection.type].addConnection_(this.outputConnection);
    this.render();
   }
},
mutationToDom: function() {
  var container = document.createElement('mutation');
  if (! this.changeList) {
    container.setAttribute('destructive', this.changeList);
  }
  return container;
},
domToMutation: function(xmlElement) {
  if(!xmlElement.getAttribute('destructive')){
    this.changeList = true;
  } else {
    this.changeList = (xmlElement.getAttribute('destructive') == "true");
  }
  this.updateBlock_();
},
decompose: function(workspace) {
  var containerBlock = Blockly.Block.obtain(workspace,'lists_mutatorcontainer');
  containerBlock.initSvg();
  var changeListButton = containerBlock.getField_('CHANGE_LIST');
  var makeNewListButton = containerBlock.getField_('MAKE_NEW_LIST');
  var group = changeListButton.group;
  if (this.changeList) {
    group.setSelected(changeListButton);
  } else {
    group.setSelected(makeNewListButton);
  }
  return containerBlock;
},
compose: function(containerBlock) {
 this.oldChangeList = this.changeList;
  this.changeList = containerBlock.getFieldValue('CHANGE_LIST') == 'TRUE' ? true : false;
  if (this.oldChangeList != this.changeList) {
    this.updateBlock_();
  }
},
saveConnections: Blockly.saveConnections,
```

```
  getVars: function() {
    return [this.getFieldValue('VAR')];
  },
  blocksInScope: function() {
    var testBlock = this.getInputTargetBlock('TEST');
    if (testBlock) {
      return [testBlock];
    } else {
      return [];
    }
  },
  declaredNames: function() {
    return [this.getFieldValue('VAR')];
  },
  renameVar: function(oldName, newName) {
    if (Blockly.Names.equals(oldName, this.getFieldValue('VAR'))) {
      this.setFieldValue(newName, 'VAR');
    }
  },
  typeblock: [{ translatedName: Blockly.Msg.LANG_LISTS_FILTER_INPUT_COLLAPSED_TEXT }]
};


Blockly.Blocks['lists_reduce'] = {
  // For each loop.
  category : 'Lists',
  helpUrl : Blockly.Msg.LANG_LISTS_REDUCE_HELPURL,
  init : function() {
    this.setColour(Blockly.LIST_CATEGORY_HUE);
    this.appendValueInput('LIST')
        .setCheck(Blockly.Blocks.Utilities.YailTypeToBlocklyType("list",Blockly.Blocks.Utilities.INPUT))
        .appendField(Blockly.Msg.LANG_LISTS_REDUCE_TITLE_REDUCE)
        .appendField(Blockly.Msg.LANG_LISTS_REDUCE_INPUT_INLIST)
        .setAlign(Blockly.ALIGN_RIGHT);
    this.appendValueInput('INITANSWER')
        .appendField(Blockly.Msg.LANG_LISTS_REDUCE_INPUT_INITIAL_ANSWER)
        .setAlign(Blockly.ALIGN_RIGHT);
    this.appendDummyInput('DESCRIPTION')
        .appendField(Blockly.Msg.LANG_LISTS_REDUCE_INPUT_COMBINE)
        .appendField(new
            Blockly.FieldParameterFlydown(Blockly.Msg.LANG_LISTS_REDUCE_INPUT_VAR,
                                          true, // name is editable
                                          Blockly.FieldFlydown.DISPLAY_BELOW),
                    'VAR1')
        .appendField(Blockly.Msg.LANG_LISTS_REDUCE_INPUT_AND)
        .appendField(new
            Blockly.FieldParameterFlydown(Blockly.Msg.LANG_LISTS_REDUCE_INPUT_ANSWER,
                                          true, // name is editable
                                          Blockly.FieldFlydown.DISPLAY_BELOW),
                    'VAR2')
        .setAlign(Blockly.ALIGN_RIGHT);
    this.appendIndentedValueInput('COMBINE');
    this.setOutput(true, null);
    this.setTooltip(Blockly.Msg.LANG_LISTS_REDUCE_TOOLTIP);
  },
  // onchange: Blockly.WarningHandler.checkErrors,
  getVars: function() {
    var names = []
    names.push(this.getFieldValue('VAR1'));
```

```
    names.push(this.getFieldValue('VAR2'));
    return names;
  },
  blocksInScope: function() {
    var combineBlock = this.getInputTargetBlock('COMBINE');
    if (combineBlock) {
      return [combineBlock];
    } else {
      return [];
    }
  },
  declaredNames: function() {
    return this.getVars();
  },
  renameVar: function(oldName, newName) {
    if (Blockly.Names.equals(oldName, this.getFieldValue('VAR1'))) {
      this.setFieldValue(newName, 'VAR1');
    }
    if (Blockly.Names.equals(oldName, this.getFieldValue('VAR2'))) {
      this.setFieldValue(newName, 'VAR2');
    }
  },
  typeblock: [{ translatedName: Blockly.Msg.LANG_LISTS_REDUCE_TITLE_REDUCE }]
};

Blockly.Blocks['lists_sort'] = {
  // For each loop.
  category : 'Lists',
  helpUrl : Blockly.Msg.LANG_LISTS_SORT_HELPURL,
  init : function() {
    this.setColour(Blockly.LIST_CATEGORY_HUE);
    this.appendValueInput('LIST')
        .setCheck(Blockly.Blocks.Utilities.YailTypeToBlocklyType("list",Blockly.Blocks.Utilities.INPUT))
        .appendField(Blockly.Msg.LANG_LISTS_SORT_DEST_TITLE_SORT, 'TITLE')
    this.setPreviousStatement(true);
    this.setNextStatement(true);
    this.setMutator(new Blockly.Mutator([]));
    this.setTooltip(Blockly.Msg.LANG_LISTS_SORT_TOOLTIP);
    this.changeList = true;
  },
  // onchange: Blockly.WarningHandler.checkErrors,
  updateBlock_: function() {
    if (this.changeList) {
      if (this.outputConnection && this.outputConnection.targetConnection) {
        this.outputConnection.disconnect();

        var children = this.parentBlock_.childBlocks_;
        for (var child, x = 0; child = children[x]; x++) {
          if (child == this) {
            children.splice(x, 1);
            break;
          }
        }
        this.setParent(null);
        this.parentBlock_ = null;
      }

      if (this.outputConnection) {
        if (this.outputConnection.inDB_) {
```

```
      this.outputConnection.dbList_[this.outputConnection.type].removeConnection_(this.outputConnection);
    }
    this.outputConnection.inDB_ = false;
    if (Blockly.highlightedConnection_ == this.outputConnection) {
      Blockly.highlightedConnection_ = null;
    }
    if (Blockly.localConnection_ == this.outputConnection) {
      Blockly.localConnection_ = null;
    }
  }

  this.outputConnection = null;
  this.setFieldValue(Blockly.Msg.LANG_LISTS_SORT_DEST_TITLE_SORT, 'TITLE');
  this.setPreviousStatement(true);
  this.setNextStatement(true);
  this.previousConnection.dbList_[this.previousConnection.type].addConnection_(this.previousConnection);
  this.nextConnection.dbList_[this.nextConnection.type].addConnection_(this.nextConnection);
  this.render();

} else {
  if (this.previousConnection && this.previousConnection.targetConnection) {
    this.previousConnection.disconnect();

    var children = this.parentBlock_.childBlocks_;
    for (var child, x = 0; child = children[x]; x++) {
      if (child == this) {
        children.splice(x, 1);
        break;
      }
    }
    this.setParent(null);
    this.parentBlock_ = null;
 }

  if (this.previousConnection) {
    if (this.previousConnection.inDB_) {
      this.previousConnection.dbList_[this.previousConnection.type].
      removeConnection_(this.previousConnection);
    }
    this.previousConnection.inDB_ = false;
    if (Blockly.highlightedConnection_ == this.previousConnection) {
      Blockly.highlightedConnection_ = null;
    }
    if (Blockly.localConnection_ == this.previousConnection) {
      Blockly.localConnection_ = null;
    }
  }

  if (this.nextConnection) {
    if (this.nextConnection.inDB_) {
      this.nextConnection.dbList_[this.nextConnection.type].removeConnection_(this.nextConnection);
    }
    this.nextConnection.inDB_ = false;
    if (Blockly.highlightedConnection_ == this.nextConnection) {
      Blockly.highlightedConnection_ = null;
    }
    if (Blockly.localConnection_ == this.nextConnection) {
      Blockly.localConnection_ = null;
    }
```

```
        if (Blockly.localConnection_ == this.nextConnection) {
          Blockly.localConnection_ = null;
        }
      }

      this.previousConnection = null;
      this.nextConnection = null;
      this.setFieldValue(Blockly.Msg.LANG_LISTS_SORT_NONDEST_TITLE_SORT, 'TITLE');
      this.setOutput(true, null);
      this.outputConnection.dbList_[this.outputConnection.type].addConnection_(this.outputConnection);
      this.render();
     }
  },
  mutationToDom: function() {
    var container = document.createElement('mutation');
    if (! this.changeList) {
      container.setAttribute('destructive', this.changeList);
    }
    return container;
  },
  domToMutation: function(xmlElement) {
    if(!xmlElement.getAttribute('destructive')){
      this.changeList = true;
    } else {
      this.changeList = (xmlElement.getAttribute('destructive') == "true");
    }
    this.updateBlock_();
  },
  decompose: function(workspace) {
    var containerBlock = Blockly.Block.obtain(workspace,'lists_mutatorcontainer');
    containerBlock.initSvg();
    var changeListButton = containerBlock.getField_('CHANGE_LIST');
    var makeNewListButton = containerBlock.getField_('MAKE_NEW_LIST');
    var group = changeListButton.group;
    if (this.changeList) {
      group.setSelected(changeListButton);
    } else {
      group.setSelected(makeNewListButton);
    }
    return containerBlock;
  },
  compose: function(containerBlock) {
   this.oldChangeList = this.changeList;
    this.changeList = containerBlock.getFieldValue('CHANGE_LIST') == 'TRUE' ? true : false;
    if (this.oldChangeList != this.changeList) {
      this.updateBlock_();
    }
  },
  saveConnections: Blockly.saveConnections,
  typeblock: [{ translatedName: Blockly.Msg.LANG_LISTS_SORT_INPUT_COLLAPSED_TEXT }]
};

Blockly.Blocks['lists_sort_comparator'] = {
  // For each loop.
  category : 'Lists',
  helpUrl : Blockly.Msg.LANG_LISTS_SORT_COMPARATOR_HELPURL,
  init : function() {
    this.setColour(Blockly.LIST_CATEGORY_HUE);
    this.appendValueInput('LIST')
```

```
        .setCheck(Blockly.Blocks.Utilities.YailTypeToBlocklyType("list",Blockly.Blocks.Utilities.INPUT))
        .appendField(Blockly.Msg.LANG_LISTS_SORT_COMPARATOR_DEST_TITLE_SORT, 'TITLE')
        .setAlign(Blockly.ALIGN_RIGHT);
    this.appendDummyInput('DESCRIPTION')
        .appendField(Blockly.Msg.LANG_LISTS_SORT_COMPARATOR_INPUT_COMPARATOR)
        .appendField(new
            Blockly.FieldParameterFlydown(Blockly.Msg.LANG_LISTS_SORT_COMPARATOR_INPUT_VAR1,
                                          true, // name is editable
                                          Blockly.FieldFlydown.DISPLAY_BELOW),
                'VAR1')
        .appendField(Blockly.Msg.LANG_LISTS_SORT_COMPARATOR_INPUT_AND)
        .appendField(new
            Blockly.FieldParameterFlydown(Blockly.Msg.LANG_LISTS_SORT_COMPARATOR_INPUT_VAR2,
                                          true, // name is editable
                                          Blockly.FieldFlydown.DISPLAY_BELOW),
                'VAR2')
        .appendField(Blockly.Msg.LANG_LISTS_SORT_COMPARATOR_INPUT_COMPARATOR2)
        .setAlign(Blockly.ALIGN_RIGHT);
    this.appendIndentedValueInput('COMPARE');
    this.setMutator(new Blockly.Mutator([]));
    this.setPreviousStatement(true);
    this.setNextStatement(true);
    this.setTooltip(Blockly.Msg.LANG_LISTS_SORT_COMPARATOR_TOOLTIP);
    this.changeList = true;
  },
//onchange: Blockly.WarningHandler.checkErrors,
  updateBlock_: function() {
    if (this.changeList) {
      if (this.outputConnection && this.outputConnection.targetConnection) {
        this.outputConnection.disconnect();

        var children = this.parentBlock_.childBlocks_;
        for (var child, x = 0; child = children[x]; x++) {
          if (child == this) {
            children.splice(x, 1);
            break;
          }
        }
        this.setParent(null);
        this.parentBlock_ = null;
      }

      if (this.outputConnection) {
        if (this.outputConnection.inDB_) {
          this.outputConnection.dbList_[this.outputConnection.type].removeConnection_(this.outputConnection);
        }
        this.outputConnection.inDB_ = false;
        if (Blockly.highlightedConnection_ == this.outputConnection) {
          Blockly.highlightedConnection_ = null;
        }
        if (Blockly.localConnection_ == this.outputConnection) {
          Blockly.localConnection_ = null;
        }
      }

      this.outputConnection = null;
      this.setFieldValue(Blockly.Msg.LANG_LISTS_SORT_COMPARATOR_DEST_TITLE_SORT, 'TITLE');
      this.setPreviousStatement(true);
      this.setNextStatement(true);
```

```
    this.previousConnection.dbList_[this.previousConnection.type].addConnection_(this.previousConnection);
    this.nextConnection.dbList_[this.nextConnection.type].addConnection_(this.nextConnection);
    this.render();

  } else {
    if (this.previousConnection && this.previousConnection.targetConnection) {
      this.previousConnection.disconnect();

      var children = this.parentBlock_.childBlocks_;
      for (var child, x = 0; child = children[x]; x++) {
        if (child == this) {
          children.splice(x, 1);
          break;
        }
      }
      this.setParent(null);
      this.parentBlock_ = null;
   }

    if (this.previousConnection) {
      if (this.previousConnection.inDB_) {
        this.previousConnection.dbList_[this.previousConnection.type].
        removeConnection_(this.previousConnection);
      }
      this.previousConnection.inDB_ = false;
      if (Blockly.highlightedConnection_ == this.previousConnection) {
        Blockly.highlightedConnection_ = null;
      }
      if (Blockly.localConnection_ == this.previousConnection) {
        Blockly.localConnection_ = null;
      }
    }

    if (this.nextConnection) {
      if (this.nextConnection.inDB_) {
        this.nextConnection.dbList_[this.nextConnection.type].removeConnection_(this.nextConnection);
      }
      this.nextConnection.inDB_ = false;
      if (Blockly.highlightedConnection_ == this.nextConnection) {
        Blockly.highlightedConnection_ = null;
      }
      if (Blockly.localConnection_ == this.nextConnection) {
        Blockly.localConnection_ = null;
      }
      if (Blockly.localConnection_ == this.nextConnection) {
        Blockly.localConnection_ = null;
      }
    }

    this.previousConnection = null;
    this.nextConnection = null;
    this.setFieldValue(Blockly.Msg.LANG_LISTS_SORT_COMPARATOR_NONDEST_TITLE_SORT,
        'TITLE');
    this.setOutput(true, null);
    this.outputConnection.dbList_[this.outputConnection.type].addConnection_(this.outputConnection);
    this.render();
   }
},
mutationToDom: function() {
```

```
    var container = document.createElement('mutation');
    if (! this.changeList) {
      container.setAttribute('destructive', this.changeList);
    }
    return container;
  },
  domToMutation: function(xmlElement) {
    if(!xmlElement.getAttribute('destructive')){
      this.changeList = true;
    } else {
      this.changeList = (xmlElement.getAttribute('destructive') == "true");
    }
    this.updateBlock_();
  },
  decompose: function(workspace) {
    var containerBlock = Blockly.Block.obtain(workspace,'lists_mutatorcontainer');
    containerBlock.initSvg();
    var changeListButton = containerBlock.getField_('CHANGE_LIST');
    var makeNewListButton = containerBlock.getField_('MAKE_NEW_LIST');
    var group = changeListButton.group;
    if (this.changeList) {
      group.setSelected(changeListButton);
    } else {
      group.setSelected(makeNewListButton);
    }
    return containerBlock;
  },
  compose: function(containerBlock) {
   this.oldChangeList = this.changeList;
    this.changeList = containerBlock.getFieldValue('CHANGE_LIST') == 'TRUE' ? true : false;
    if (this.oldChangeList != this.changeList) {
      this.updateBlock_();
    }
  },
  saveConnections: Blockly.saveConnections,
  getVars: function() {
    return [this.getFieldValue('VAR')];
  },
  blocksInScope: function() {
    var compareBlock = this.getInputTargetBlock('COMPARE');
    if (compareBlock) {
      return [compareBlock];
    } else {
      return [];
    }
  },
  declaredNames: function() {
    return [this.getFieldValue('VAR')];
  },
  renameVar: function(oldName, newName) {
    if (Blockly.Names.equals(oldName, this.getFieldValue('VAR'))) {
      this.setFieldValue(newName, 'VAR');
    }
  },
  typeblock: [{ translatedName:
      Blockly.Msg.LANG_LISTS_SORT_COMPARATOR_INPUT_COLLAPSED_TEXT }]
};
```

```
Blockly.Blocks['lists_sort_key'] = {
  // For each loop.
  category : 'Lists',
  helpUrl : Blockly.Msg.LANG_LISTS_SORT_KEY_HELPURL,
  init : function() {
    this.setColour(Blockly.LIST_CATEGORY_HUE);
    this.appendValueInput('LIST')
        .setCheck(Blockly.Blocks.Utilities.YailTypeToBlocklyType("list",Blockly.Blocks.Utilities.INPUT))
        .appendField(Blockly.Msg.LANG_LISTS_SORT_KEY_DEST_TITLE_SORT, 'TITLE')
        .setAlign(Blockly.ALIGN_RIGHT);
    this.appendDummyInput('DESCRIPTION')
      .appendField(Blockly.Msg.LANG_LISTS_SORT_KEY_INPUT_KEY)
      .appendField(new
          Blockly.FieldParameterFlydown(Blockly.Msg.LANG_LISTS_SORT_KEY_INPUT_VAR,
                                        true, // name is editable
                                        Blockly.FieldFlydown.DISPLAY_BELOW),
                'VAR')
      .setAlign(Blockly.ALIGN_RIGHT);
    this.appendIndentedValueInput('KEY');
    this.setPreviousStatement(true);
    this.setNextStatement(true);
    this.setMutator(new Blockly.Mutator([]));
    this.setTooltip( Blockly.Msg.LANG_LISTS_SORT_KEY_TOOLTIP);
    this.changeList = true;
  },
  onchange: Blockly.WarningHandler.checkErrors,
  updateBlock_: function() {
      if (this.changeList) {
        if (this.outputConnection && this.outputConnection.targetConnection) {
          this.outputConnection.disconnect();

          var children = this.parentBlock_.childBlocks_;
          for (var child, x = 0; child = children[x]; x++) {
            if (child == this) {
              children.splice(x, 1);
              break;
            }
          }
          this.setParent(null);
          this.parentBlock_ = null;
        }

        if (this.outputConnection) {
          if (this.outputConnection.inDB_) {
            this.outputConnection.dbList_[this.outputConnection.type].removeConnection_(this.outputConnection);
          }
          this.outputConnection.inDB_ = false;
          if (Blockly.highlightedConnection_ == this.outputConnection) {
            Blockly.highlightedConnection_ = null;
          }
          if (Blockly.localConnection_ == this.outputConnection) {
            Blockly.localConnection_ = null;
          }
        }

        this.outputConnection = null;
        this.setFieldValue(Blockly.Msg.LANG_LISTS_SORT_KEY_DEST_TITLE_SORT, 'TITLE');
```

```
    this.setPreviousStatement(true);
    this.setNextStatement(true);
    this.previousConnection.dbList_[this.previousConnection.type].addConnection_(this.previousConnection);
    this.nextConnection.dbList_[this.nextConnection.type].addConnection_(this.nextConnection);
    this.render();

  } else {
    if (this.previousConnection && this.previousConnection.targetConnection) {
      this.previousConnection.disconnect();

      var children = this.parentBlock_.childBlocks_;
      for (var child, x = 0; child = children[x]; x++) {
        if (child == this) {
          children.splice(x, 1);
          break;
        }
      }
      this.setParent(null);
      this.parentBlock_ = null;
   }

    if (this.previousConnection) {
      if (this.previousConnection.inDB_) {
        this.previousConnection.dbList_[this.previousConnection.type].
        removeConnection_(this.previousConnection);
      }
      this.previousConnection.inDB_ = false;
      if (Blockly.highlightedConnection_ == this.previousConnection) {
        Blockly.highlightedConnection_ = null;
      }
      if (Blockly.localConnection_ == this.previousConnection) {
        Blockly.localConnection_ = null;
      }
    }

    if (this.nextConnection) {
      if (this.nextConnection.inDB_) {
        this.nextConnection.dbList_[this.nextConnection.type].removeConnection_(this.nextConnection);
      }
      this.nextConnection.inDB_ = false;
      if (Blockly.highlightedConnection_ == this.nextConnection) {
        Blockly.highlightedConnection_ = null;
      }
      if (Blockly.localConnection_ == this.nextConnection) {
        Blockly.localConnection_ = null;
      }
      if (Blockly.localConnection_ == this.nextConnection) {
        Blockly.localConnection_ = null;
      }
    }

    this.previousConnection = null;
    this.nextConnection = null;
    this.setFieldValue(Blockly.Msg.LANG_LISTS_SORT_KEY_NONDEST_TITLE_SORT, 'TITLE');
    this.setOutput(true, null);
    this.outputConnection.dbList_[this.outputConnection.type].addConnection_(this.outputConnection);
    this.render();
   }
},
```

```
  mutationToDom: function() {
    var container = document.createElement('mutation');
    if (! this.changeList) {
      container.setAttribute('destructive', this.changeList);
    }
    return container;
  },
  domToMutation: function(xmlElement) {
    if(!xmlElement.getAttribute('destructive')){
      this.changeList = true;
    } else {
      this.changeList = (xmlElement.getAttribute('destructive') == "true");
    }
    this.updateBlock_();
  },
  decompose: function(workspace) {
    var containerBlock = Blockly.Block.obtain(workspace,'lists_mutatorcontainer');
    containerBlock.initSvg();
    var changeListButton = containerBlock.getField_('CHANGE_LIST');
    var makeNewListButton = containerBlock.getField_('MAKE_NEW_LIST');
    var group = changeListButton.group;
    if (this.changeList) {
      group.setSelected(changeListButton);
    } else {
      group.setSelected(makeNewListButton);
    }
    return containerBlock;
  },
  compose: function(containerBlock) {
   this.oldChangeList = this.changeList;
    this.changeList = containerBlock.getFieldValue('CHANGE_LIST') == 'TRUE' ? true :
        false;
    if (this.oldChangeList != this.changeList) {
      this.updateBlock_();
    }
  },
  saveConnections: Blockly.saveConnections,
  getVars: function() {
    return [this.getFieldValue('VAR')];
  },
  blocksInScope: function() {
    var keyBlock = this.getInputTargetBlock('KEY');
    if (keyBlock) {
      return [keyBlock];
    } else {
      return [];
    }
  },
  declaredNames: function() {
    return [this.getFieldValue('VAR')];
  },
  renameVar: function(oldName, newName) {
    if (Blockly.Names.equals(oldName, this.getFieldValue('VAR'))) {
      this.setFieldValue(newName, 'VAR');
    }
  },
  typeblock: [{ translatedName: Blockly.Msg.LANG_LISTS_SORT_KEY_INPUT_COLLAPSED_TEXT }]
};
```

Appendix C

runtime.scm

```
(define (yail-list-map proc yail-list)
  (let ((verified-list (coerce-to-yail-list yail-list)))
    (if (eq? verified-list *non-coercible-value*)
        (signal-runtime-error
         (format #f
                 "The second argument to map is not a list. The second argument is: ~A"
                 (get-display-representation yail-list))
         "Bad list argument to map")
         (kawa-list->yail-list (map proc (yail-list-contents verified-list))))))

(define (map-destructive proc lst)
    (cond ((null? lst) *the-null-value*)
        (begin
          (set-car! lst (proc (car lst)))
          (map-destructive proc (cdr lst))
          *the-null-value*)))

(define (yail-list-map! proc yail-list)
  (let ((verified-list (coerce-to-yail-list yail-list)))
    (if (eq? verified-list *non-coercible-value*)
        (signal-runtime-error
         (format #f
                 "The second argument to map is not a list. The second argument is: ~A"
                 (get-display-representation yail-list))
         "Bad list argument to map")
         (map-destructive proc (yail-list-contents verified-list)))))

(define (yail-list-filter pred yail-list)
  (let ((verified-list (coerce-to-yail-list yail-list)))
    (if (eq? verified-list *non-coercible-value*)
        (signal-runtime-error
         (format #f
                 "The second argument to filter is not a list. The second argument is: ~A"
                 (get-display-representation yail-list))
         "Bad list argument to filter")
        (kawa-list->yail-list (filter pred (yail-list-contents verified-list))))))

(define (yail-list-filter! pred yail-list)
```

```
  (let ((verified-list (coerce-to-yail-list yail-list)))
    (if (eq? verified-list *non-coercible-value*)
        (signal-runtime-error
         (format #f
                 "The second argument to filter is not a list. The second argument is: ~A"
                 (get-display-representation yail-list))
         "Bad list argument to map")
        (set-cdr! verified-list (filter pred (yail-list-contents verified-list))))))

(define (yail-list-reduce ans binop yail-list)
  (define (reduce accum func lst)
     (cond ((null? lst) accum)
           (else (reduce (func accum (car lst)) func (cdr lst)))))
  (let ((verified-list (coerce-to-yail-list yail-list)))
    (if (eq? verified-list *non-coercible-value*)
        (signal-runtime-error
         (format #f
                 "The second argument to reduce is not a list. The second argument is: ~A"
                 (get-display-representation yail-list))
         "Bad list argument to reduce")
        (kawa-list->yail-list (reduce ans binop (yail-list-contents verified-list))))))

(define (reverse-list lst)
    (cond ((null? lst) lst)
          ((null? (cdr lst)) lst)
          (else (append (reverse-list (cdr lst)) (list (car lst))))))

(define (yail-list-reverse yl)
  (kawa-list->yail-list (reverse-list (yail-list-contents yl))))

(define (yail-list-reverse! y1)
  (set-cdr! y1 (reverse-list (yail-list-contents y1))))

;;Implements a generic sorting procedure that works on lists of any type.

(define typeordering '(boolean number text list component))

(define (typeof val)
   (cond ((boolean? val) 'boolean)
         ((number? val) 'number)
         ((string? val) 'text)
         ((yail-list? val) 'list)
         ((instance? val com.google.appinventor.components.runtime.Component) 'component)
         (else (signal-runtime-error
                (format #f
                          "typeof called with unexpected value: ~A"
                          (get-display-representation val))
                 "Bad arguement to typeof"))))

(define (indexof element list)
   (list-index (lambda (x) (eq? x element)) list))

(define (type-lt? type1 type2)
   (< (indexof type1 typeordering)
      (indexof type2 typeordering)))

(define (is-lt? val1 val2)
    (let ((type1 (typeof val1))
        (type2 (typeof val2)))
```

```
     (or (type-lt? type1 type2)
         (and (eq? type1 type2)
              (cond ((eq? type1 'boolean) (boolean-lt? val1 val2))
                    ((eq? type1 'number) (< val1 val2))
                    ((eq? type1 'text) (string<? val1 val2))
                    ((eq? type1 'list) (list-lt? val1 val2))
                    ((eq? type1 'component) (component-lt? val1 val2))
                    (else (signal-runtime-error
                      (format #f
                             "(islt? ~A ~A)"
                                 (get-display-representation val1)
                                 (get-display-representation val2))
                        "Shouldn't happen")))))))

(define (is-eq? val1 val2)
    (let ((type1 (typeof val1))
          (type2 (typeof val2)))
      (and (eq? type1 type2)
           (cond ((eq? type1 'boolean) (boolean-eq? val1 val2))
                 ((eq? type1 'number) (= val1 val2))
                 ((eq? type1 'text) (string=? val1 val2))
                 ((eq? type1 'list) (list-eq? val1 val2))
                 ((eq? type1 'component) (component-eq? val1 val2))
                 (else (signal-runtime-error
                       (format #f
                             "(islt? ~A ~A)"
                                 (get-display-representation val1)
                                 (get-display-representation val2))
                        "Shouldn't happen")))))))

(define (is-leq? val1 val2)
    (let ((type1 (typeof val1))
          (type2 (typeof val2)))
      (or (type-lt? type1 type2)
          (and (eq? type1 type2)
               (cond ((eq? type1 'boolean) (boolean-leq? val1 val2))
                     ((eq? type1 'number) (<= val1 val2))
                     ((eq? type1 'text) (string<=? val1 val2))
                     ((eq? type1 'list) (list-leq? val1 val2))
                     ((eq? type1 'component) (component-leq? val1 val2))
                     (else (signal-runtime-error
                       (format #f
                             "(isleq? ~A ~A)"
                                 (get-display-representation val1)
                                 (get-display-representation val2))
                        "Shouldn't happen")))))))

;;false is less than true
(define (boolean-lt? val1 val2)
   (and (not val1) val2))

(define (boolean-eq? val1 val2)
   (or (and val1 val2)
      (and (not val1) (not val2))))

(define (boolean-leq? val1 val2)
   (not (and val1 (not val2))))

(define (list-lt? y1 y2)
```

```
  (define (helper-list-lt? lst1 lst2)
    (cond ((null? lst1) (not (null? lst2)))
          ((null? lst2) #f)
          ((is-lt? (car lst1) (car lst2)) #t)
          ((is-eq? (car lst1) (car lst2)) (helper-list-lt? (cdr lst1) (cdr lst2)))
          (else #f)))
  (helper-list-lt? (yail-list-contents y1) (yail-list-contents y2)))

(define (list-eq? y1 y2)
   (define (helper-list-eq? lst1 lst2)
      (cond ((and (null? lst1) (null? lst2)) #t)
            ((is-eq? (car lst1) (car lst2)) (helper-list-eq? (cdr lst1) (cdr lst2)))
            (else #f)))
   (helper-list-eq? (yail-list-contents y1) (yail-list-contents y2)))

;;throw exception is not yail-list
(define (yail-list-necessary y1)
  (cond ((yail-list? y1) (yail-list-contents y1))
        (else y1)))

(define (list-leq? y1 y2)
  (define (helper-list-leq? lst1 lst2)
    (cond ((and (null? lst1) (null? lst2)) #t)
          ((null? lst1) #t)
          ((null? lst2) #f)
          ((is-lt? (car lst1) (car lst2)) #t)
          ((is-eq? (car lst1) (car lst2)) (helper-list-leq? (cdr lst1) (cdr lst2)))
          (else #f)))
  (helper-list-leq? (yail-list-necessary y1) (yail-list-necessary y2)))

;;Component are first compared using their class names. If they are instances of the same
    class,
;;then they are compared using their hashcodes.
(define (component-lt? comp1 comp2)
   (or (string<? (*:getSimpleName (*:getClass comp1))
                 (*:getSimpleName (*:getClass comp2)))
       (and (string=? (*:getSimpleName (*:getClass comp1))
                      (*:getSimpleName (*:getClass comp2)))
            (< (*:hashCode comp1)
               (*:hashCode comp2)))))

(define (component-eq? comp1 comp2)
   (and (string=? (*:getSimpleName (*:getClass comp1))
                  (*:getSimpleName (*:getClass comp2)))
        (= (*:hashCode comp1)
           (*:hashCode comp2))))

(define (component-leq? comp1 comp2)
   (or (string<? (*:getSimpleName (*:getClass comp1))
                 (*:getSimpleName (*:getClass comp2)))
       (and (string=? (*:getSimpleName (*:getClass comp1))
                      (*:getSimpleName (*:getClass comp2)))
            (<= (*:hashCode comp1)
                (*:hashCode comp2)))))

(define (merge lessthan? lst1 lst2)
   (cond ((null? lst1) lst2)
         ((null? lst2) lst1)
```

```
       ((lessthan? (car lst1) (car lst2)) (cons (car lst1) (merge lessthan? (cdr lst1)
           lst2)))
       (else (cons (car lst2) (merge lessthan? lst1 (cdr lst2))))))

(define (mergesort lessthan? lst)
   (cond ((null? lst) lst)
         ((null? (cdr lst)) lst)
         (else (merge lessthan? (mergesort lessthan? (take lst (quotient (length lst) 2)))
                        (mergesort lessthan? (drop lst (quotient (length lst) 2)))))))

(define (yail-list-sort y1)
   (cond ((yail-list-empty? y1) (make YailList))
         ((not (pair? y1)) y1)
         (else (kawa-list->yail-list (mergesort is-leq? (yail-list-contents y1))))))

(define (yail-list-sort! y1)
  (cond ((yail-list-empty? y1) (make YailList))
         ((not (pair? y1)) y1)
         (else
            (set-cdr! y1 (mergesort is-leq? (yail-list-contents y1))))))

(define (yail-list-sort-comparator lessthan? y1)
   (cond ((yail-list-empty? y1) (make YailList))
         ((not (pair? y1)) y1)
         (else (kawa-list->yail-list (mergesort lessthan? (yail-list-contents y1))))))

(define (yail-list-sort-comparator! lessthan? y1)
   (cond ((yail-list-empty? y1) (make YailList))
         ((not (pair? y1)) y1)
         (else
            (set-cdr! y1 (mergesort lessthan? (yail-list-contents y1))))))

(define (merge-key lessthan? key lst1 lst2)
     (cond ((null? lst1) lst2)
           ((null? lst2) lst1)
           ((lessthan? (key (car lst1)) (key (car lst2))) (cons (car lst1) (merge-key
               lessthan? key (cdr lst1) lst2)))
           (else (cons (car lst2) (merge-key lessthan? key lst1 (cdr lst2))))))

(define (mergesort-key lessthan? key lst)
     (cond ((null? lst) lst)
           ((null? (cdr lst)) lst)
           (else (merge-key lessthan? key (mergesort-key lessthan? key (take lst (quotient
               (length lst) 2)))
                            (mergesort-key lessthan? key (drop lst (quotient (length
                                lst) 2)))))))

(define (yail-list-sort-key key y1)
   (cond ((yail-list-empty? y1) (make YailList))
         ((not (pair? y1)) y1)
         (else (kawa-list->yail-list (mergesort-key is-leq? key (yail-list-contents
             y1))))))

(define (yail-list-sort-key! key y1)
  (cond ((yail-list-empty? y1) (make YailList))
         ((not (pair? y1)) y1)
         (else
            (set-cdr! y1 (mergesort-key is-leq? key (yail-list-contents y1))))))
```