The Beauty and Joy of Computing[1]

Lab Exercise 5: Using Lists for Data

Objectives

By completing this lab exercise, you should learn to

- Describe properties and operations on a list data structure;
- Use list operation blocks in BYOB to write programs that use lists;
- Write code using the iterator pattern in a variety of settings; and
- Describe and implement two different ways of rearranging items in a list.

Background (Pre-Lab Reading)

You should read this section before coming to the lab. It describes how various things work in BYOB and provides pictures to show what they look like in BYOB. You should have a good understanding of this material before the lab – if you can get that good understanding from just reading through this, that's great, but you would probably benefit from experimenting with the new list operations in BYOB on your own.

In the lab activities up until this point, we have typically dealt with one or two numbers at a time. You can solve some interesting mathematical problems this way, but one of the strengths of computers is their ability to process large amounts of data. Imagine writing a program that kept track of the grades of 30 students in a class – would you create a different variable for each individual student? That would be pretty impractical even for 30 students, but imagine what it's like for a university to keep track of 17,000 students or the IRS to keep track of 300,000,000 social security numbers. What we need is a way to keep track of a collection of data using a single variable, organizing it so that we can use it in the way we need. Methods for organizing collections of data are called **data structures**, and we explore one of the simplest possible data structures in this lab.

Lists

A simple way to organize information is in a **list**. We use lists all the time in daily life. We make shopping lists, to-do lists, and mailing lists. Each of our lab exercises starts with a list of objectives, and ends with a list of new terms used in that lab. At the start of a semester, each instructor gets a list of students that are in their class. What I just gave you in this very paragraph was a list of examples of lists (got that?). Like a lot of commonly-used concepts, we don't often step back and ask what are the important properties of this well-understood idea, so we'll consider this carefully now.

A list is a collection of **items**, which are ordered in a sequence. Since they are in a sequence, there is a first item and a last item in the list, and for every item we can identify the next item in the list. If we count from the first item in the list, every item has a position, which we usually call

[1]*Lab Exercises for "The Beauty and Joy of Computing"*
Copyright © 2012-2014 by Stephen R. Tate – Creative Commons License
See http://www.uncg.edu/cmp/faculty/srtate/csc100labs for more information

an **index**, in the list. For example, consider a list of errands we have to do: get gas, pick up dry cleaning, do grocery shopping, and pay bills. The first item in the list is “get gas”, the last item is “pay bills”, and the item at index 3 is “do grocery shopping”.

Given these properties of a list, we can describe some actions we might want to perform on a list. A common action is to *add a new item* to the end of a list. For example, with our list of errands we might decide to add “meet Jim for lunch” to the list, and if we were keeping a written list it would naturally go at the end of the list. More generally, we might add items at other positions in the list, so if our list is prioritized in the order we need to do things, we might want to put “meet Jim for lunch” as a higher priority item, right after “get gas”, so we would *insert the new item* at index 2. Think about what happens to the position of the other items in this list when this happens: “pick up dry cleaning” is no longer in position 2 (since that’s where the new item is), but is now at index 3. In fact, everything after our insertion has its index increased by one, since there is one additional item before it now. This is important to remember: an item’s index is not fixed, since additions and deletions before the item will cause its index to change.

Other operations include *deleting an item* from a list, so if we were near the dry cleaning store doing something else, we might go ahead and pick up our dry cleaning and delete it from the list. Deletions are just like arbitrary insertions: while deleting the first or the last item from a list is probably most common, in general we can delete any item at any position in the list. One other way we might change a list is to *replace an item* of the list with something else – for example, if we later decide that we’d rather meet Jim for coffee, we can replace the item that says “meet Jim for lunch” with the new entry “meet Jim for coffee”.

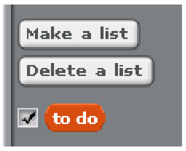
The actions we just described are all the ways we might want to change a list, but we also would like to look at items in the list to learn some information, even if we’re not changing the list. We might want to *determine how many items are in a list* – for example, our original list (before we thought about Jim) had four items in it, so we say it had **length** 4. We can also *ask for a particular item* from the list. Just like deletions and insertions, we often are interested in either the first or last item in a list, but we could also ask for the 3rd item (or more generally for any item using its index). Finally, we might *ask whether a particular item is in a list*. For example, you might wonder if you need to go grocery shopping, which is the same as asking whether “go grocery shopping” is on your to-do list.

Hopefully all of the previous discussion was clear and somewhat obvious. There are many things that we work with on a daily basis, but it often surprises people how difficult it can be to clearly define these things and the operations we might want to perform on them. This is a very important skill for a programmer! You must be able to very precisely and clearly describe what you are working with, and how you can work with it.

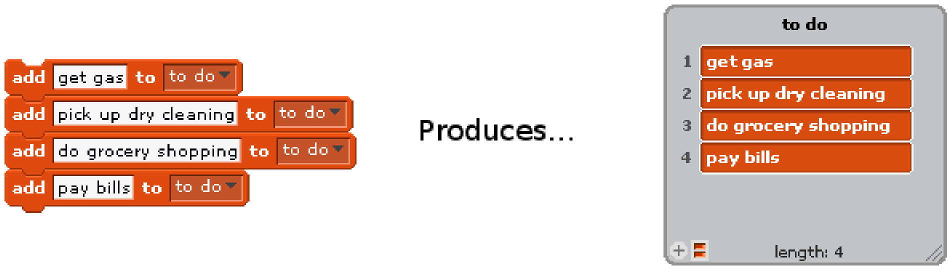
Lists in BYOB

Programs often manage lists as part of their operation. The examples we described in the previous section might appear very directly in a program to manage a to-do list. We might also have a program that manages student records, and in that case the program would manage several lists: lists of students, lists of classes they took, lists of grades, lists of awards they received, etc. Almost every modern programming language, including BYOB, provides a way to work with lists. There is even a programming language named **LISP** (for LISt Processing) in which everything is a list – all data is stored in lists, and programs are lists of instructions that are stored exactly like data lists. BYOB can store a list in a variable, can pass lists as arguments to blocks (functions), and provides blocks to perform each of the list operations we described above. List operations are all located in the "Variables" category of the blocks palette.

Lists can be stored in regular variables, but there are also special variables that are just for storing lists. You create a list variables using the "Make a list" button in the blocks palette. The following picture shows this "Make a list" button, along with a list that we made named "to do":



While the "to do" list variable looks practically identical to a regular variable, there's a subtle color difference: list variables (and list blocks) are a darker shade of orange (almost tan), while regular variables and associated blocks are light orange. The "Make a list" button creates an empty list. This might be a bit of a strange concept when you first see it – an empty list? How can it be a "list" if it doesn't have any items? This doesn't match up very well with our intuition of a list, but it's not that different from starting a to-do list on paper: We pull out a blank piece of paper, write "To Do" at the top, and this is the start of our to-do list even if it doesn't have any items on it yet. You can add items to the end of a list with the "add" list command block, so we'll use that to set up our to-do list. Notice the check beside the "to do" block in our picture above, which makes it a watch variable, and just like the watch variables we've seen before this displays the value of the variable on the stage in what we call a "watch box." List variable watch boxes have a special form, showing the item at each index. The following picture shows the four "add" blocks that we execute to set up our list, and the resulting list displayed in its watch box:



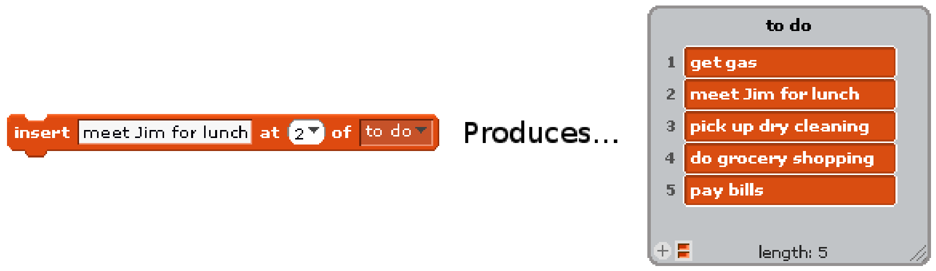
```
add [get gas] to [to do]
add [pick up dry cleaning] to [to do]
add [do grocery shopping] to [to do]
add [pay bills] to [to do]
```

Produces…

```
to do
1 get gas
2 pick up dry cleaning
3 do grocery shopping
4 pay bills
length: 4
```

Pay attention to several things in the displayed list: the name of the list (to do) is at the top of the display, each item is shown with its index, and the length of the list is shown at the bottom. The shaded area in the lower right corner allows you to resize the box if you'd like to, and there's a "+" button on the lower left that allows you to manually add items to the list through this watch box. You can click on any item to select it and edit it, and when you do that an "x" appears that allows you to delete the item. You can also click on the top of the box to drag it to a different position on the stage. But of course, changing a list by typing in the watch box is not what we want to do – we want to change the list under the control of our program, using BYOB blocks!

Returning to the example of the previous section, when talking about operations we might want to perform on a list, we said we might want to insert a new item (like “meet Jim for lunch”) in the list at index 2. There’s a block for that! The following picture shows the block that performs this action, and the result of executing that block on the list contents:

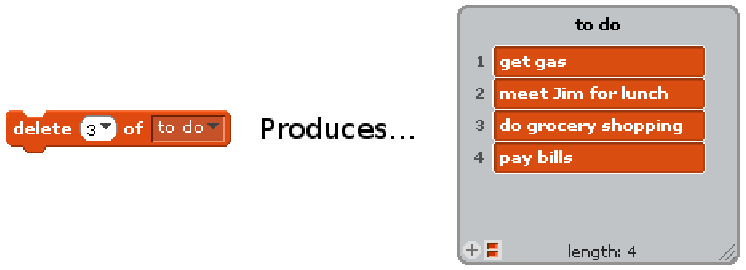


insert meet Jim for lunch at 2 of to do

Produces...

to do
1 get gas
2 meet Jim for lunch
3 pick up dry cleaning
4 do grocery shopping
5 pay bills
length: 5

Notice that the index slot, which we have set to 2, has an arrow that will open a drop-down menu that contains options “1” (to insert at the first position), “last” to insert at the end, and “any” which will insert at a random location. These are the most common positions to insert, but you can type anything you want to in the box (like we did when we put in “2”), or you can drag a variable or reporter block into the index parameter. The “any” possibility sounds interesting, but is almost never what we want to do – control where you put things in the list so that you have predictable results! In this class at least, it is best to avoid the “any” argument and never use it. In our discussion above, we mentioned that indices of the items later in the list will increase by one, which is obvious from the picture. For example, “pick up dry cleaning” is now shown at index 3. We talked about deleting items from the list as well, and the BYOB block that will delete the index 3 item (which is “pick up dry cleaning”) from the list is shown below, along with the result when applied to the last list:



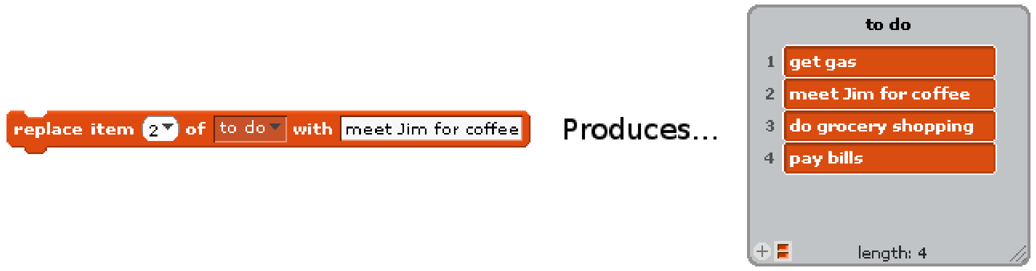
delete 3 of to do

Produces...

to do
1 get gas
2 meet Jim for lunch
3 do grocery shopping
4 pay bills
length: 4

There’s also an “all” option in the drop down menu for the first delete parameter, which will delete all items from a list so that it is empty again.

The last operation we described that can change the list is to replace an item with a new value. The following BYOB block will replace the index 2 item (now “meet Jim for lunch”) with the new value “meet Jim for coffee”, with the resulting list shown on the right:

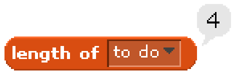


replace item 2 of to do with meet Jim for coffee

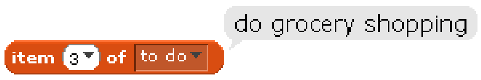
Produces...

to do
1 get gas
2 meet Jim for coffee
3 do grocery shopping
4 pay bills
length: 4

BYOB also provides reporter blocks to determine the length of a list and to get a specific item from a list, shown below:

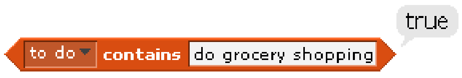


```
length of (to do)
```



```
item (3) of (to do)
```

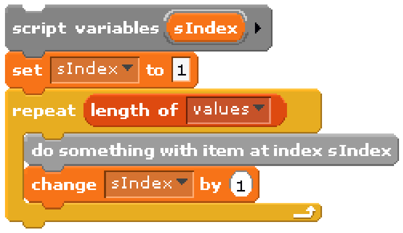
There is also a predicate to test if a particular item is in a list:



```
<(to do) contains [do grocery shopping]>
```

Processing a List

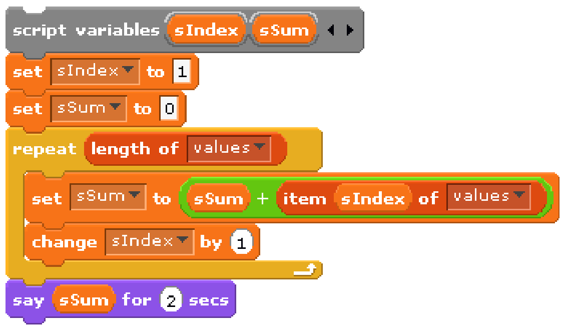
There is a loop pattern that is very common when processing a list, which we use in the fairly common situation in which we want to step through a list and do something with each item. We might be looking for a specific item (in which case the “do something with each item” might be a comparison), or looking for the minimum value in a list of numbers, or adding up the sum of the numbers in a list. This is really just our counter pattern, where the counter is used as an index into the list, and the number of iterations is determined by the length of the list. In general, a variable which allows us to select one item at a time out of a data structure is called an **iterator**, and so our counter (index) variable is an iterator, and we call this the **iterator pattern**. This is the third pattern we’ve seen, after the counter pattern and the accumulator pattern, and if you continue to program you will see these a lot. In software engineering, a **pattern** is a reusable solution to a commonly occurring problem – a general software structure that can be used in many situations by making well-defined modifications or additions. Patterns can get fairly complex – there are several books written on common patterns in programming, and this is often a topic of discussion in a software engineering class. Returning to our list iterator pattern, here’s the idea: we want to go through the list, with a variable stepping through each index in the list. Here’s the basics of this pattern, using a list of numbers that we have named “values”:



```
script variables (sIndex)
set sIndex to 1
repeat (length of (values))
    do something with item at index sIndex
    change sIndex by 1
```

This code will look at a different item in the list each time we go through the body of the “repeat” loop, and as noted above, it is almost identical to the counter pattern. The first visible difference is the name we use for the counter, which reflects the fact this variable represents an index in our list. Remember that variable names should always reflect what they represent! The second visible difference is that the repeat block uses “length of …” to control the number of iterations, since we want to go through each item in the list. There is also a difference that is not visible: By using this pattern we know we’re processing a list one item at a time, and that puts us as a programmer in a certain mind-set that helps you focus on the job you need to do. While patterns are useful as specific pieces of code that can be reused, they are also useful guiding the programmer into thinking about a problem in a productive way.

As an example of this pattern, let's say we want to add up all the values in a list of numbers. In this case, we want to add each list item (i.e., each value) to a running total. This is in fact just a variation of the accumulator pattern that we saw in the last lab, but now we are accumulating the answer as we step through the list. The key components of the accumulator pattern, combined with the iterator pattern, are the following: a variable declaration of the "accumulator variable" which will hold our answer, initialization of this variable, and updating the variable for each element inside the loop. As a useful example, consider the following script to add up all the values in the "values" list:

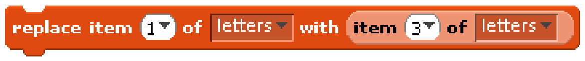


```
script variables sIndex sSum
set sIndex to 1
set sSum to 0
repeat (length of values)
    set sSum to (sSum + (item sIndex of values))
    change sIndex by 1
say sSum for 2 secs
```

Keep in mind that accumulating an answer does not necessarily mean *adding* up the values, even though that's a natural interpretation of the term "accumulate." We could multiply to get the product of all values in a list. We could update the maximum value to determine the largest value in a list. We could "join" strings together to create a long string containing all items in the list. These are all examples of the accumulator pattern, which we see over and over again in almost all programs.

Swapping Items in a List

A common action that is performed on lists is to swap two items. If we have a list containing four characters – (a,b,c,d) – and we swapped the 1st and 3rd items, it will change the list to (c,b,a,d). Swapping is done efficiently by using the "replace item" built-in block, but it is a little tricky. In the example of swapping the 1st and 3rd items, we could think of doing something like this:



```
replace item 1 of letters with (item 3 of letters)
```

If we do this on the original list, the result is (c,b,c,d). But we have completely lost the "a" that had been in the first element, and don't have a way to recover it! The solution to this problem is simple once you understand it, but not obvious at first.

To describe the solution, let me tell you a story[2] that might help: When my kids were younger they were very picky about what they wanted to drink, and what they wanted to drink it out of. My son had his favorite glass with a tyrannosaurus on it, and my daughter had her favorite glass with a cat. My son really liked apple juice, while my daughter would only drink cranapple juice. And if you put the wrong drink in the wrong glass, beware! You would think the world was coming to an end! Here's a picture of the apocalypse-inducing situation:

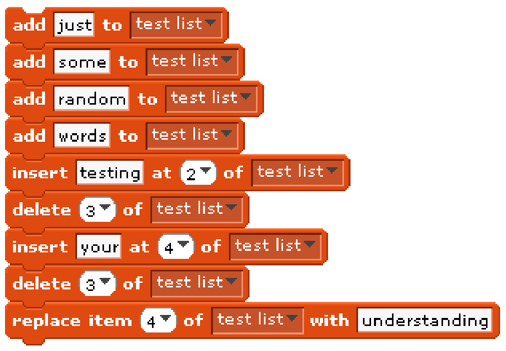
[2] Not really a true story. But it *could* have happened!

How can we swap the drinks in the glasses without throwing any juice away? The answer is to use a spare glass: we pour the cranapple juice into the spare glass, then pour the apple juice into the tyrannosaurus glass, and finally pour the cranapple juice from the spare glass into the cat glass. Sure, we have to wash an extra glass, but we didn't waste any juice, and we narrowly avoided the end of times. Hopefully you see how this analogy relates to lists in BYOB and swapping items: the spare glass is an extra variable (a script variable would be great for this), and pouring juice from one glass to another is the same as using the "set" and "replace" blocks.

Self-Assessment Questions

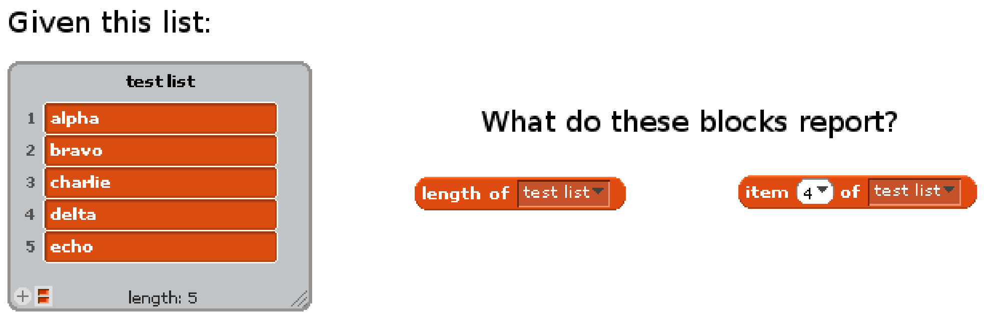
Use these questions to test your understanding of the Background reading. Answers can be found at the end of the lab exercise.

1. If the following script is executed when "test list" starts off empty, what is stored in "test list" after the script executes?



```
add just to test list
add some to test list
add random to test list
add words to test list
insert testing at 2 of test list
delete 3 of test list
insert your at 4 of test list
delete 3 of test list
replace item 4 of test list with understanding
```

2. What do the following two blocks report for the given list?



Given this list:

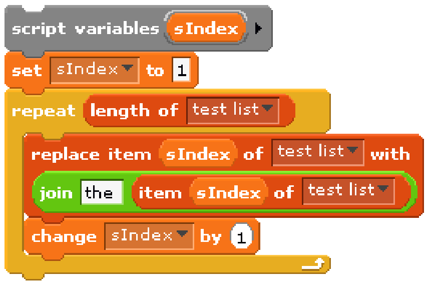
```
test list
1 alpha
2 bravo
3 charlie
4 delta
5 echo
length: 5
```

What do these blocks report?

```
length of test list
```

```
item 4 of test list
```

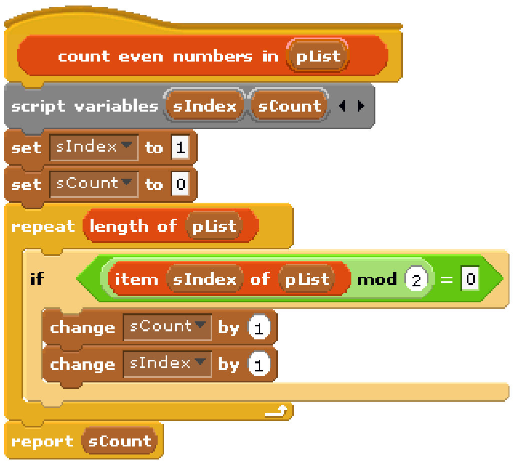
3. Consider the following use of the iterator pattern:



```
script variables sIndex
set sIndex to 1
repeat length of test list
    replace item sIndex of test list with join the item sIndex of test list
    change sIndex by 1
```

Describe, at a high level, what this script does.

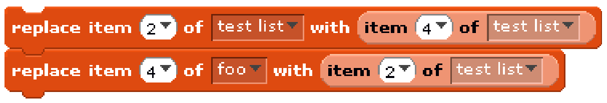
4. In trying to make a block that counts how many even numbers are in a list, I define the following block using the iterator pattern:



```
count even numbers in pList
script variables sIndex sCount
set sIndex to 1
set sCount to 0
repeat length of pList
    if item sIndex of pList mod 2 = 0
        change sCount by 1
        change sIndex by 1
report sCount
```

Unfortunately, this doesn't work. Why not? What happens if I execute this block with the list [2, 1, 6, 9] (the correct answer is 2, since there are two even numbers in the list)?

5. A programmer wanted to swap the items at position 2 and 4 of "test list" – thinking about this, he thinks "I want to replace item 2 with item 4 and vice-versa," so creates the following script:

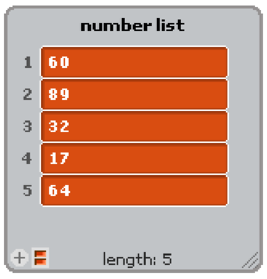


```
replace item 2 of test list with item 4 of test list
replace item 4 of foo with item 2 of test list
```

Unfortunately, this doesn't work. What is the result if this script is run with the list "test list" shown in question 2?

6. Answer the following question about the provided list and script (warning: this is a little tricky – actually perform the operations on paper so you can see what's going on):

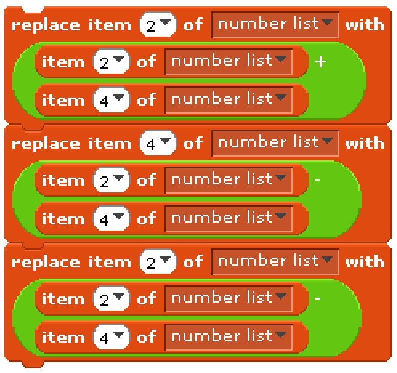
Given this list:



number list	
1	60
2	89
3	32
4	17
5	64

length: 5

What does this do?



```
replace item 2 of number list with (item 2 of number list + item 4 of number list)
replace item 4 of number list with (item 2 of number list - item 4 of number list)
replace item 2 of number list with (item 2 of number list - item 4 of number list)
```

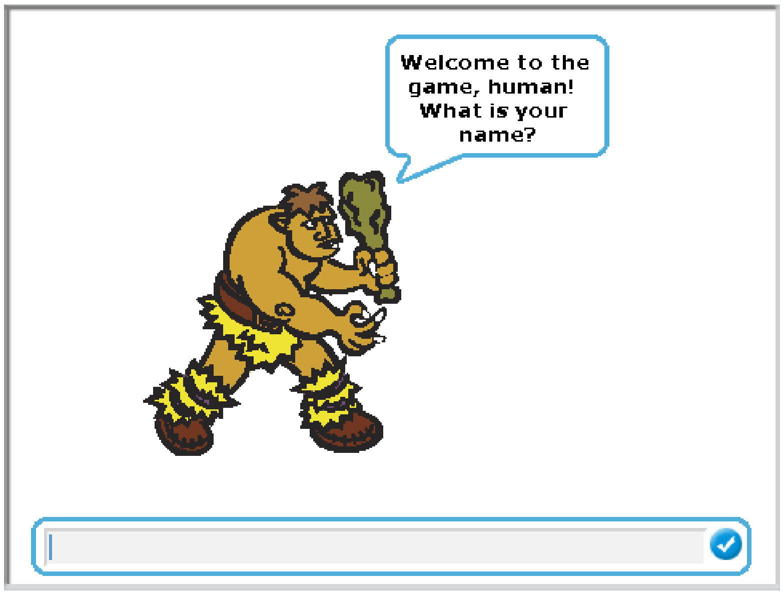
Activities (In-Lab Work)

The following activities describe what you are supposed to do during lab time.

Activity 1: This activity is a little different from the ones you've had in previous labs. Rather than producing code to do something useful, the goal is to explore how certain things work in BYOB, and answer questions based on what you learn. You can determine all of the answers by experimenting with the blocks in BYOB, but you'll need to decide *how* to do this yourself. This is a great thing about working with computers – if you don't know how something works, you can usually try it out and discover how it works pretty easily. The questions are listed here, but you will provide your answers on the post-lab quiz – during lab time you should experiment, explore, take notes, and come up with answers to these questions, but save the answers for the quiz. Note that your answer on the quiz should not just say what the answer is, but you will need to describe what you did to determined the answer. Here are the questions:

- An operation is "case sensitive" if strings that differ only in the case of characters (upper or lower case) are treated differently – in other words, in a case sensitive operation, "Alonzo" is treated differently from "alonzo". Is the "=" operator in BYOB case sensitive?
- Is the "contains" list predicate case sensitive?
- The valid index values for a list are integers from 1 to the length of the list, inclusive. What happens if the "item … of …" block is given an invalid index (either too large, or possibly a negative index)?
- What happens if the "insert … at … of …" block is given an invalid index?

Activity 2: In this activity, you will write BYOB scripts to manage a player list for a multi-player game. In particular, your code should maintain a list of player names so that it can tell when a player is a new or returning player. This list should start out empty, and then your script should make a “welcome” screen that asks the user for their name, something like this:



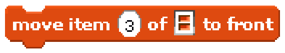
You should make your own choice of sprite and welcome message (using the “ask” block) – don’t just copy what I did. There is only one requirement on the welcome message: don’t be boring! (Just asking “What is your name?” is boring.) Once you get the user’s name, you should use the list “contains” predicate to determine if they are in the current player list. If they are, you should welcome them back – for example, if they entered “Kotter” as their name and they had been here before, you should say “Welcome back, Kotter!” You can modify the message a little bit, but two things are required and important: (1) It should be grammatically correct, and with proper spacing (space between words and after punctuation) – don’t be sloppy! (2) You must have something in the “say” bubble both before and after the player’s name (in my case, “Welcome back, ” was before, and the exclamation point was after). You’ll use “join” blocks to construct this response. If the user is new, you should print a different messages with the same requirements – something like “Nice to meet you, Kotter!” New users should also be added to the player list so that they will be recognized when they return. Finally, you should put all of this into a “forever” loop so that you keep asking for player names.

Once this is working, save your program in a file named Lab5-Activity2.

Activity 3: In Activity 1, you should have found that the list “contains” block and the “=” comparison block treat strings differently – one is case sensitive, and the other is not. Create your own “contains” predicate that uses a loop and the “=” block so that your block treats case sensitivity the same way as the “=” block (and so differently from the built-in contains block). This is a straightforward use of the iterator pattern, so start with that. (*Hint: When you find the value in the list, there is no need to finish iterating through the list – just use the report block to immediately report the correct Boolean value.*) Name your predicate sensibly so that it’s clear from the block name what it does. Once you get this working, replace your use of the standard “contains” predicate in the player list script from the previous activity, and test to make sure it does what you think it should.

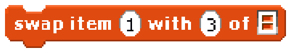
Save the resulting program as Lab5-Activity3.

Activity 4: Sometimes we want to move items around in a list, and for this activity you will create two different blocks that each rearrange items in a list. Consider the problem of having our program output a list of "recent players" of the game. An easy way to do this is to modify the player welcome dialog that you wrote for Activity 2 so that when a player is already in the list, they are moved to the front of the list – if you do this, and you add new players at the front of the list, then the most recently seen players will always be the first ones in the list. To accomplish this, make a "move item … of … to front" block that moves an item to the front of a list. This block should look like this:



This is fairly easy to accomplish using the "delete" and "insert" built-in BYOB blocks, but be careful about the index. Your block should not mess up the list if it is provided with an invalid index!

The second way to rearrange a list it to swap two items, as described in the pre-lab reading. You should create a block that does this, and it should look like this:



To accomplish this, use the "replace item" block, using "spare storage" as described in the Pre-Lab reading (do not use "insert" or "delete" for this part of the solution).

When you have created and tested these two blocks, save your program as Lab5-Activity4.

Activity 5: For this activity, you are to make a reporter block that takes a list of player scores (each one being a positive integer) as a parameter, and reports the high score. This is just an example of the accumulator pattern described in the Pre-Lab reading. In fact, you can define a particular (and very simple) reporter block so that this activity can be solved by taking the code in the accumulator pattern example from the Pre-Lab reading and replacing exactly one block with your custom block (you will also report the answer rather than "say"ing it). There's even a hint about this in the Pre-Lab reading. Test your solution, and save it as Lab5-Activity5.

Submission

In this lab, you should have saved the following files: Lab5-Activity2, Lab5-Activity3, Lab5-Activity4, and Lab5-Activity5. *Special note: Lab5-Activity2 should be submitted, since that code is modified in Activity 3 and you need to turn in the original code. However, if your Lab5-Activity5 file contains all of the code produced in Activities 3 and 4 in addition to Activity 5, you can just turn in the Lab5-Activity5 file.* Turn these in using whatever submission mechanism your school has set up for you.

Discussion (Post-Lab Follow-up)

Move To Front: In Activity 4, you created a "move … to front" block. It turns out that this is a very simple and surprisingly useful concept. We'll describe ways to talk about the efficiency of algorithms in the next lab, but for now some intuition should be good enough. Think about your "contains …" block from Activity 3: If you have a list of 1000 items and you scan the list looking for a value, then if the item is at the end of the list it takes a long time to find since we must go through everything before it in the list. However, if the item is at or near the front of the list, we will find it much faster. An algorithm that works with a list could move each item to the front of the list every time it is accessed, and then the items that are accessed most often naturally stay near the front of the list. Consider a program that counts how many times each word occurs in a document. The word "the" occurs a lot, so will stay near the front of the list and be found quickly. The word "mytacism" would not appear much[3], if at all, so would not be near the front of the list.

Moving accessed items to the front of a list is not a difficult technique to implement, and in fact it seems a little simple-minded: a very simple rule that doesn't make complex decisions about which items should be placed at what locations in the list. What's great about this is that, despite the fact that it is very simple, it works remarkably well. While the math is beyond what we do in this class, a famous computer science research paper[4] showed that this simple technique is no more than twice as slow as any other rule – even if we try to be super-intelligent about how to order the list, and if we can make a plan for ordering the list knowing all of the list accesses that will ever happen in the future. One of life's great pleasures is when simple and easy things work so well!

Terminology

- *accumulator pattern*: A software structure in which an answer is "accumulated" by updating a running answer as we go through a list or other set of values. The running answer can be a sum if we're adding up items in a list, the largest value seen so far if we are computing the max at each position, or many other possibilities.
- *data structure*: A way of organizing a collection of data in a program, such as putting data in a list.
- *index*: The position of an item in a list (e.g., the second item in a list has index 2).
- *item*: A generic term for something in a list. Items in a list can be numbers, strings, or even other lists.
- *iterator pattern*: A pattern in which a loop examines a list item-by-item, updating an index variable at each step of the iteration.
- *LISP*: A programming language in which data as well as programs are organized and presented as lists. LISP stands for "LISt Processing".
- *list*: A data structure in which data is stored in a sequence of items, where each item has a position (or an index) in the list.
- *list length*: The number of items in a list.
- *pattern*: A reusable software solution to a commonly occurring problem.

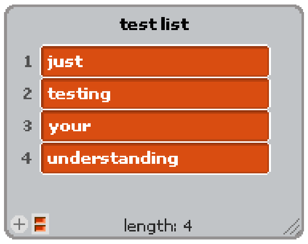
Answers to Pre-Lab Self-Assessment Questions

[3] Yes, it's a real word – look it up!

[4] D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, Vol. 28, No. 2, Feb 1985, pp. 202-208.

Answers to the Self-Assessment Questions are given below – see the Pre-Lab Reading section for the questions.

1. After running the script, “test list” has the following contents:

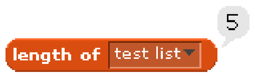

test list

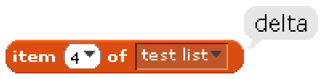
1 just
2 testing
3 your
4 understanding

length: 4

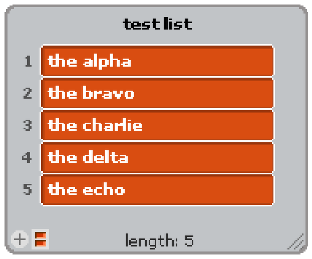
Note that if you missed this question, you can try out the script in BYOB and execute one block at a time while watching the list contents so that you see how each operation works.

2. The answers are shown below:





3. This script puts the word “the ” (with a space at the end) in front of every item in the list. For example, if it were run when “test list” had the contents shown in question 2, then “test list” would be changed to the following:

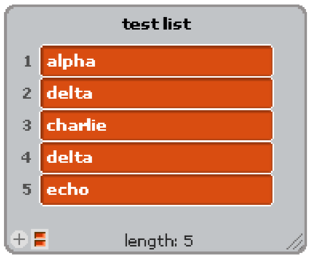

test list

1 the alpha
2 the bravo
3 the charlie
4 the delta
5 the echo

length: 5

4. The problem with this definition is that the index is only incremented when looking at an even list item. In other words, the sIndex variable will no longer increment once it sees an odd number, and it will get "stuck" at that number. For the list [2, 1, 6, 9], the block will properly count the first element as an even number (setting sCount to 1 and sIndex to 2), but the next time through it will not change either sCount or sIndex. Subsequent iterations of the loop will keep looking at item 2, and since it is odd it will never get past this. In the end, it will report 1, rather than the correct answer of 2.

5. The problem with this is exactly what is explained in the pre-lab reading, and is the reason we generally need a “spare” variable when swapping values. The resulting list is shown below:

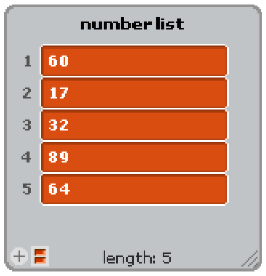

test list

1	alpha
2	delta
3	charlie
4	delta
5	echo

length: 5

Note that “delta” was duplicated, and “bravo” disappeared.

6. The script changes “number list” to the following list – note that items in positions 2 and 4 are swapped:


number list

1	60
2	17
3	32
4	89
5	64

length: 5

What’s important about this is that we swapped two values without using a spare variable! This is a cute but confusing trick – it will always work with numbers, but should be avoided because it’s not very understandable. And, of course, since we need to do arithmetic with the values it won’t work at all unless the list items are numbers.