

Chapter 8 LIVECODE experiments

This chapter gives a brief tutorial on using the free and open-source programming language LIVECODE for programming behavioral experiments for cognitive research. The tutorial will cover some basic LIVECODE operations, programming a Stroop experiment, and scripts for preliminary data-analysis.

8.1 Download LIVECODE

The community edition of LIVECODE can be downloaded for free from the LIVECODE website:

<http://www.livecode.com>

Beginners to LIVECODE should peruse the website for helpful tutorials, and should read through the manual.

8.2 What is LIVECODE

LIVECODE is a powerful and easy to learn language for creating applications. The language is cross-platform and can be used on macs, pcs, linux machines, and even mobile devices (ios and android). LIVECODE makes it easy to

create the visual layout for your program, and to add elements (text fields, buttons, etc.) by dragging and dropping. All of the elements in a LIVECODE program can be scripted.

8.3 *Basic LIVECODE concepts*

8.3.1 **Stacks, Cards and Elements**

The starting point for creating a program is the stack. When you create a new stack, you will see a new window appear. The window can be resized to any size that you wish. When you a user loads your program, they will see the stack load up as the this window.

Stacks are composed of cards. Each stack can have many cards. Cards are like individual slides in a powerpoint or keynote presentation. New cards can be added by selecting New Card from the Object menu.

Many different kinds of elements can be added to cards by dragging and dropping. These elements are accessed through the tools palette (Tools>Tools Palette). Typical elements are text fields (for displaying and entering text), buttons (for receiving user input), media (for showing images, playing movies and sound clips), scrollbars, graphic objects, and so on. The size and position of these

objects on each card can be changed using the mouse, much like elements inside a powerpoint.

The stack, cards, and elements can all be given unique names using the property inspector. The property inspector is accessed by double-clicking on the element of choice (but make sure you are using the pointer tool, Tools>Pointer Tool).

8.3.2 Scripting

Scripts are used to implement algorithms and to control how part of your stack work. Almost every thing in LIVECODE is scriptable. The stack itself is an entity that can be scripted. Each card is its own entity that be scripted. And all of the elements on each card are their own entities that can be scripted. It is important to realize that LIVECODE scripts work in a hierarchical fasion. The stack script is at the top of the hierarchy, the card scripts are next, and the element scripts are at the bottom. Generally, the scripts that are written at each level are specific to that level. However, writing a function at a higher level (in the stack script), allows that function to be available for use at the lower levels. The reverse is not true. Writing a function inside a button script that is on a card does not mean that function will be available to other elements on the same card or other cards. Similarly, variables

that are created inside scripts for the stack, cards, or elements, are local to those entities, but they can be made global (accessible to all) if this is specified in the script.

You can access the scripting editor for the stack by Object>Stack Script.

Similarly, you can access the scripting editor for the card by Object>Card Script.

If you drag a text field and a button onto an empty stack, then you access the scripts for these elements by right-clicking on them and choosing edit script.

SHORTCUTS TO THE EDITOR: When you have selected the edit tool (the arrow with the crosshairs in it) you can double-click on objects and the property menu for that object will appear. Click on the play button and choose edit script. Also, if you are on a mac you can hold down option-command and click on an object to automatically open the script editor.

8.3.2.0.1 Getting started with Scripting

To get a hang of the basic syntax of LIVECODE drag a text field and a button onto the card. Open the script associated with the button. You will see an editor appear with the following text:

```
on mouseUp
```

```
end mouseUp
```

These statements are the bookends of the scripts that can be written for the button. They are the opening and closing clauses. The statement “on mouseUp” is waiting for the button to receive a mouse-click. When you click a mouse, you first press the button down, then you release the button. When the button is released it will receive a “mouseUp” message. When this message is received the button will execute the script that is contained between the on and end mouseUp messages.

Type the following between the on and end statements so that it looks like this:

```
on mouseUp
  put "hello world" into field 1
end mouseUp
```



Once you have changed the contents of the script, you must press the ****apply**** button located in the left hand corner. Now the button is ready to run. Change the cursor from the pointer arrow to the browse arrow (just the big arrow), and click the button. If you have made a text field, then when you press the button you should see the message hello world appear inside the text field. You have made your first program in LIVECODE.

8.3.3 LIVECODE syntax

The LIVECODE language is written mostly in English and is relatively easy to follow. The following introduces you to important central concepts in the vocabulary. LIVECODE comes with its own dictionary which is very helpful. Also, if you can't find the answer in the dictionary, you might try googling.

- put: the word put is used to move information between variables.

```
put 1 into x
put 2 into x
put "hello world" into field 1
```

Take a look at the first example. In this example there are two livecode intrinsics, these are words that are reserved for use by LIVECODE. The intrinsics are put and into. You will also see the double dashes “-”. These allow inline commenting for your code. Anything that appears after the double dashes (# also works) will not be interpreted as code. Commenting is useful to keep track and make notes as you code. It is also helpful for others when they are trying to make sense of your code. It is good to practice commenting as you code.

- into: replaces the contents of a variables. **Replaces** is very important, into first wipes the variable, then inserts the new information into the variable

- after: appends new information to a variable. After is different from into, after does not erase information from a variable, it puts new information after the old information in the variable.

```
put 1 into x  
put 1 after x  
put 2 into x
```

The words put, into, and after are central to the language. They take a general form:

```
put something into somevariabl  
put something after somevariab
```



The something, can be any kind of information, numbers, letters, words, whole paragraphs. The somevariable is simply a variable that you create to hold the information, think of it like a container with a name. You choose the name by writing a string of letters following the command into or after. Here's a few more examples

```
put field 1 into field 2
put word 1 of field 1 into fie
put item 1 of x into xx
put the last word of field 1 i
put char 1 of word 5 of field
put char 1 of word 5 of field
```

These examples show some more
LIVECODE intrinsics like field, char, word,
and item. These will be discussed shortly.

8.3.4 Manipulating words and letters (strings)

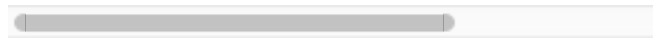
We know by now that variables are
containers for information. The
programmer can give a variable any
name, and this variable name is used to
retrieve the stored information. The
information that is stored in a variable
can be numbers or letters. Depending on
how the data is stored, and the kind of
data that is stored, the data can be
addressed or found in different ways.
LIVECODE is particularly useful for
dealing with information from written
language. To illustrate, let's say field 1
has the following sentence:

The quick brown fox jumped over the
lazy dog

Now let's look at some example scripts
that we could write in the button to
manipulate the words in the sentence.

The examples assume that you have dragged a second text field (field 2) onto the stack.

```
put field 1 into field 2
put word 1 of field 1 into fie
put word 3 of field 1 into fie
put char 1 of word 3 of field
put the last word of field 1 i
```



We see a hierarchical structure. Fields are the highest level, they contain all of the words and letters. Words are the next highest level and are defined by series of characters separated by a space. Chars are the lowest level, they refer to individual characters.

8.3.5 Manipulating numbers (items)

LIVECODE uses the above syntax to parse any series of characters that are separated by a space. It doesn't matter if the characters are letter or numbers, LIVECODE would treat them the same way. Importantly, there are other ways to store data in variables. Data is often stored in small parcels that are kept separate by a delimiter. Think of storing some data in Microsoft Excel, each piece of data is put in separate row or column. Excel is using a delimiter to keep the individual pieces of data separate. In LIVECODE the comma “,” is the default item-delimiter. The comma is used to

keep pieces of data separate. For example, if we wanted to keep the numbers 1 through 10 separate, we could put them into a variable in the following way:

```
put "1,2,3,4,5,6,7,8,9,10" into
```



Now x contains the numbers 1 through 10, with each separated by a comma. This means that variable x is item-delimited. Whenever a variable is item-delimited, the individual items can be addressed just like individual words can be addressed in the above example.

put item 3 of x into field 1 – the third item happens to be a 3 so a three is placed in field 1

IMPORTANT for later: When you store numbers using an item-delimiter then you will be able to perform math functions on those numbers.

```
put sum(x) into field 1
```

8.3.6 Advanced data storage (arrays)

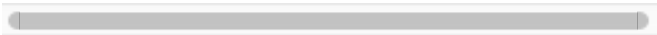
LIVECODE supports more powerful ways to store data. One way is to use arrays. You can think of an array as a variable that can contain variables, each of which can contain more variables and so on. Please read the livecode user guide to

learn more about using arrays. Arrays will also be covered in more detail in the data-analysis section.

8.3.7 Logic statements (IF THEN)

Logic statements are a fundamental component of programming. They are used to specify the conditions under which operations occur. The most common logic statement is the IF THEN statement. Let's take an example

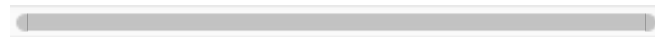
```
put random(2) into x
if x is 1 then
  put "heads" into field 1
else
  put "tails" into field 1
end if
```



The first statement uses a function called random. Random generates a random number between 1 and the number that is put in between parentheses. In this case, random(2) will either generate a 1 or a 2, and then put this value into x. The next lines show the IF THEN statement. The statement looks at the value of x and then prints different results to field 1 depending on the value of x. If x is 1 then the word heads is printed to field 1. Notice the word else, which refers to any condition that does not meet the first

condition. If x is anything else besides 1 then the word tails will be put into x. Here are some more examples

```
If x == 1 then
    put true into testme
end if
if x > 5 then
    put true into testme
end if
if x < 21 then
    put false into testme
end if
if x >= 5 then
    if x <= 10 then
        put true into testme
    end if
end if
```



This last example shows us that IF THEN statements can be nested within each other, so that multiple conditions can be evaluated. Look at the full user guide, or the dictionary for more information on syntax when using IF THEN statements.

8.3.7.0.1 Other logic statements (SWITCH)

There are times when you may wish to evaluate several different conditions. You may find yourself nesting many IF THEN statements together, and the final result could be clumsy and difficult to follow. One option is to use a SWITCH

statement. Learn more about SWITCH and CASE by using the dictionary or the user guide.

8.3.8 Loops

Another central concept in programming is the repeat loop. There are many tasks that involve doing operations over and over again. These tasks are often automatized with repeat loops. Consider the problem of adding the numbers from 1 to 10. You could try this using the add function without a repeat loop:

```
add 1 to x
add 2 to x
add 3 to x
add 4 to x
and so on
```

If you kept typing you would eventually add up all of the numbers from 1 to 10. It would be easier to use a repeat loop. There are several forms of the repeat in LIVECODE, the examples illustrates two of them.

```
Repeat 10 times
  add 1 to x
  add x to totalsum
end repeat
```

This repeat loop runs 10 times. Each time a 1 is added to x, then the contents of x is added to the variable totalsum. On the first loop, x will become 1, and a 1 will be added to total sum. On the second loop, 1 will be added to x, so x will become 2, and then x will be added to total sum, which will now equal 3. At the end of the loop, the contents of the variable totalsum will be the sum of all the numbers from 1 to 10.

```
repeat with n = 1 to 10
  add n to totalsum
end repeat
```

This repeat loop also runs 10 times. The syntax is different. Here, the repeat loop runs in integer steps from the first to last integer, in this case from 1 to 10. For each step the variable n will be replaced with a 1, next a 2, next a 3, and so on. Inside this loop we simply add the value of n, which is changing across iterations of the loop to the variable totalsum.

Notice, just like the IF THEN statements, the repeat statements are closed with “end repeat”. This tells the computer which lines should be looped , and which should not. Repeat loops can also be nested within other.

Repeat loops can be used with words too. Let’s take another look at this sentence “The quick brown fox jumped over the lazy dog”. We could put this sentence into a variable:

```
put "The quick brown fox jumpe
```

Now x contains the entire sentence. Remember item-delimiters? Currently each word in the sentence is separated by a space, let's use a repeat loop to separate each word by a ",", instead. This will allow us to illustrate a different form of the repeat loop.

```
Repeat for each word k in x
  put k & "," after x2
end repeat
```

This repeat loop does what it says. It goes through each word (remember words are separated by spaces). In this case, the loop goes through each word in the variable x (the one that contains the sentence). As the loop moves forward it puts each word into the variable k (this variable name is arbitrary, we could use any string of letters to name the variable). To unroll this, each step of the way the variable k will contain the next word in the sentence:

- Step 1, k = The
- Step 2, k = quick
- Step 3, k = brown
- Etc.

Inside the repeat loop we see only one line of code: put k & “,” after x2

The & character is a special character used by LIVECODE, it means AND in the sense of appending (put this and that somewhere). We are telling LIVECODE to put the contents of k (which on step 1 is the word The), AND “,” after the variable x2. When you use quotation marks “” in LIVECODE, this tells LIVECODE to treat whatever is inside the quotes as text. So, the statement is telling LIVECODE to do the following to x2.

- Step 1, x2 = The,
- Step 2, x2 = The, quick,
- Step 3, x2 = The, quick, brown,
- Etc.

8.3.9 Functions

When you are writing code you have the option of writing functions. A benefit of writing functions is that they are portable, and they can help you solve the same problem in the future. In this section we will cover the syntax for writing a function, and describe why they are useful.

Let's imagine you want to find the sum of the numbers in an itemized variable x.


```
put "1,2,3,4,5,6,7,8,9,10" int
```



Now, x contains the numbers 1 through 10. We could write some code to add these numbers together for us.


```
repeat for each item j in x
  add j to thesum
end repeat
put thesum into field 1
```

This code will add all of the numbers and put the result into field 1. We didn't need to write this script. LIVECODE already has a function called sum, that will do the same operation.

```
put sum(x) into field 1
```

This line of code would accomplish the same as above. The sum function is general, it will add together any item delimited set of numbers. Although LIVECODE already has a built in (intrinsic) sum function, it is instructive to write our own.

```
function mysum x
  repeat for each item j in
    add j to temp
  end repeat
  return temp
end mysum
```




Once you have written the function, then you call it using the function name and brackets ().

```
put mysum(x) into field 1
```

The code for the function can be placed inside the stack script, and this would allow the function to be used in all of the card and other element scripts. You could also place the function inside a card or a button. Here is an example of placing the function inside a button script

```
on mouseUp
  put "1,2,3,4,5,6,7,8,9,10"
  put mysum(someNumbers) into
end mouseUp
```

```
function mysum x
  repeat for each item j in
    add j to temp
  end repeat
  return temp
end mysum
```



Note that the code for the function is placed underneath the end of the `mouseUp` message. When you click apply to the changes made in the button script, LIVECODE will compile the function and add it to the button so that it can be used.

8.4 *Practice coding*

If you are new to programming entirely, then acquiring basic competency usually means a personal investment in time and effort to learning and applying the syntax to programming problems. It is helpful to have examples of working code, and it is helpful to have a series of simple to complex problems that you can attempt to solve on your own. The process of learning how to creatively construct algorithms for solving different problems is very important, and will come in handy when designing completely new experiments or running completely new kinds of analyses. Solving problems on your will also develop your ability to debug your own code.

There are lots of different websites that list ranges of problems that can be solved by most any programming language. Check out <http://projecteuler.net>) as an example. Here you will be presented with problems, usually involving math, that you can try to solve with LIVECODE scripts. An example problem might be

something like, what is the sum of all the numbers from 1 to 100? How would you use LIVECODE to solve this problem?

Below I have created three lists intended to help establish basic familiarity with LIVECODE. The first is a list of common LIVECODE features, this can act as a checklist of things you should know how to do in LIVECODE. The second is a list of common commands. You should know how to use each of these commands, and be able to produce examples of working code that utilizes each command. Finally, there is a list of simple to intermediate programming problems that you should be able to solve in LIVECODE.

8.4.1 LIVECODE features checklist

This is a checklist of basic to intermediate LIVECODE operations. You should be confident that you can perform each of these steps, and solve each of these problems.

8.4.2 Using the interface

8.4.2.0.1 Stacks and Cards

1. Create a new stack.
2. Resize the stack window

3. Save the stack to a folder on your computer
4. Give the stack a name using the property inspector
5. Add new cards to the stack
6. Give new cards unique names using the property inspector
7. Understand the difference between using the two arrow buttons

8.4.2.0.2 Text fields

1. Add a text field to a card
2. Give the text field a unique name
3. Resize the text field
4. Type text directly into the text field
5. Add text to the text field using the property inspector
6. Change the font of the text using the property inspector
7. Change the color of the text using the property inspector

8. Change the background color of the text field
9. Change the size of the text
10. Change the formatting of the text to left, right, or center justification
11. Hide or show the border of the text field
12. Lock the text so a user is prevented from making changes to the text field
13. Hide or show a scroll bar to the text field

8.4.2.0.3 Buttons

1. Add a button to a card
2. Give the button a unique name
3. Give the button a label that is different from the name of the button
4. Resize the button
5. Change the font of the button
6. Change the background color of the button

8.4.2.0.4 Other Elements

1. Add an image element to a card
2. Use the property inspector to load an image from your hard drive into the image element so that it displays on the card
3. Name the image element
4. Change the size of the image element
5. Add a graphic element (rectangle, circle, etc.)
6. Name the graphic element
7. Change the fill color of the graphic element
8. Change the line width of the graphic element
9. Change the line color of the graphic element

8.4.2.0.5 Other basic livecode operations

1. Open the message box
2. Open the livecode dictionary
3. Open the property inspector for any livecode element

4. Open the application browser
5. Copy and paste livecode elements (text fields, buttons etc.) inside a card
6. Copy and paste a card inside a stack
7. Open the script editor for any livecode element
8. Open the script editor for a card
9. Open the script editor for a stack

8.4.3 LIVECODE commands

For each of these basic livecode commands you should be able to give two examples of how the command can be used in a script, and explain the meaning of the command. Here is an example using the command put.

Put: The command put is used to place content into a variable. Examples of content could be a number, a string of characters, of the contents of another variable

```
put 1 into x
put x into y
put "hello world" into field 1
```



1. put
2. into
3. after
4. &
5. space
6. tab
7. line
8. word
9. char
10. item
11. set
12. the
13. it
14. on mouseUp
15. end mouseUp
16. on mouseDown – end mouseDown
17. on openCard – end openCard

18. on openStack – end openStack
19. on preOpenStack – end preOpenStack
20. on keyDown – end keyDown
21. send
22. the milliseconds
23. if
24. then
25. else
26. is
27. <
28. >
29. =
30. <=
31. >=
32. switch
33. repeat x times – end repeat
34. repeat with n =x to x – end repeat

- 35. repeat for each line n in field 1 – end
repeat
- 36. repeat for each word n in field 1 – end
repeat
- 37. repeat for each char n in field 1 – end
repeat
- 38. repeat for each item n in x – end
repeat
- 39. repeat forever – end repeat
- 40. exit repeat
- 41. repeat while – end repeat
- 42. show
- 43. hide
- 44. local
- 45. global
- 46. sort

8.4.4 Simple programming problems

For each of these programming problems, give example code that solves the problem. Create a livecode stack and

solve each problem with a new button labeled with the number for each problem.

1. create a variable, give it a name of your choice, then put any number into the variable.
2. put some text into a variable
3. Add two numbers together and put the result into a variable
4. Subtract two numbers and put the result into a variable
5. Multiple two numbers and put the result into a variable
6. Put a number into a variable called x, and another number into a variable called y. Add the x and y variables together and put the result into a new variable
7. Put a number into a variable x. Then write another line of code that replaces the contents of x with a new number
8. Put a number into a variable x. Then write another line of code that appends a new number after the existing contents of x.

9. Put the following sentence into a variable "This is a short sentence.". Then, put the 4th word of the variable into a new variable.
10. Using the above sentence, put the number of words in the variable into a new variable
11. Using the above sentence, put the first letter of the 4th word into a new variable
12. From 11, create a logical test to determine whether the first letter is the letter "s".
13. Create a variable that contains the numbers 1 to 3 separated by commas
14. Create a variable that contains the numbers 1 to 3 separated by spaces
15. Create a variable that contains the numbers 1 to 3 separated by tabs
16. Create a variable that contains the numbers 1 to 3 separated by new lines
17. Use a repeat loop to create a variable that contains the numbers 1 to 100 separated by commas

18. Using the above variable, use the sum function to put the sum of the above variable into a new variable
19. From 13, use the average function to put the average of the above variable into a new variable
20. From 13, use the stdDev function to put the standard deviation of the above variable into a new variable
21. From 13, use the max function to find the largest number and put it into a new variable
22. From 13, use the min function to find the smallest number and put it into a new variable
23. Use a repeat loop to add up all of the numbers from 1 to 100
24. Use a repeat loop, if then statements, and the mod function to list all of the odd numbers from 1 to 100.
25. Copy a paragraph (e.g., something from Wikipedia) into a text field. Write a repeat loop that finds all of the words beginning with the letter "a", and lists (separated by new lines) them in a new variable.

26. Using the same paragraph, write a repeat loop that finds all of the 3 letter words, and lists them in a new variable
27. From 24, write a repeat loop that counts all of the 3 letter words and puts the final count into a new variable
28. From 24, write a repeat loop that finds all of the unique 3 letter words. The final list should not contain any repeated words.
29. From 24, write a repeat loop that lists the number of letters in each word in the paragraph. E.g., if the first sentence is “Luke I am your father”, the answer should be 4,1,2,4,6. (for this problem include punctuation as part of the letter count. Extra challenge, can you write code that does not count the punctuation).
30. From 27, Your code from above should produce a list of numbers representing the length of each word in the paragraph. Using this list, count and list the number of occurrences of each word length.

8.5 Logic for programming experiments

8.5.1 Experiments are designed to create data-files

Experiments are designed with the aim of producing data. At this level, experiments are machines designed to produce data-files. It is important to keep this in mind when programming experiments for several reasons. First, the format of the data file changes how the data will ultimately be analysed. Good formatting decisions will facilitate the process of data-analysis after the data has been collected. Second, like any product, data files need to accomplish the job for which they were designed. **The most important aspect of creating data files is to ensure that all of the important aspects of the experiment are preserved in the data. Any researcher should be able to look at your data file and reconstruct the series of events that occurred in the experiment. This includes the order of trials, and the events that happened on each trial, including the stimulus that was presented, the experimental condition(s) for the stimulus, and the responses (times and accuracy) made on each trial.**

When programming experiments from scratch, the programmer is responsible for deciding how the data file will be formatted, and is responsible for ensuring that the data file stores all of the necessary information. There is room for

flexibility here. The data could potentially be stored in many different formats to achieve the same goals of storing all of the necessary information. This tutorial outlines one tried and true method that can be used to create data files for most any multi-trial design, using long-format data files.

8.5.2 Long-format data files

The basic strategy is to produce a table in a text file where each line corresponds to the data for each trial in an experiment. Each line will contain words and numbers separated by a tab (or other delimiter of your choice) that represent the order, events, conditions, and responses on each trial. It is sensible to have the first row correspond to the first trial, the second for the 2nd trial, and so one for the remaining trials.

The following is an example of a data-file for 12 trials in a simple Stroop experiment.

A data file like the above stores all of the necessary information to reconstruct the events of the experiment. We can see the order of trials. For each trial we can see what word was presented, and what color it was presented in. We can see when the stimulus appeared on the screen (OnsetTime), and when the response was made. We can also see the response key that was pressed on each trial, and whether or not the response

was correct or incorrect. Many experiment builder programs (like PsychoPy) format their data-files in a similar manner. As well, data in this format will facilitate data-analysis later on.

One thing to note about the data is the column for congruency, which represents the names for each of the levels of the congruency variable. Usually it is a very good idea to include columns that describe the conditions of the experiment, as this makes data-analysis easier later on. Sometimes it is not strictly necessary to include the condition names because they can be inferred from other information in the table. For example, congruent conditions can be inferred by determining whether the word and color involve the same or different words. However, as a general rule, it is always a good idea to include all of the condition information in the data file, even if the conditions can be inferred at a later stage.

8.5.3 The Data-file is the Design

When you are new to experimental design, you might think that something like a data file comes along only near the end of a project. First, the experiment needs to be designed, then programmed, then subjects need to be run, and only then do data file appears. However, the creation of data files is central to the very

beginning of programming an experiment. Of course, the data files will be empty because you will not have run any subjects. Yet, the data file will represent and define the experimental design at its most granular level, and it can be used to tell the computer how to run each trial of the experiment. Consider the following empty data file. It is empty because it contains the information that would be presented on each trial, but it is missing all of the response information:

This data file implies that certain kinds of events would happen on each trial. On trial 1, the word red would be presented in red ink. On trial 2, the word blue would be presented in red ink, and so on. A complete data file is a record of what did happen in an experiment, and an empty data file is a record of what should happen in an experiment. In order to use this list to control the experiment, all that is needed is a script to read each line, interpret the events that are supposed to occur, and then present the events and collect response information the subject. In this way there are two major components to programming an experiment:

1. Creating the list of trials (making the empty data-file)
2. Interpreting the list to run each trial

8.5.4 Creating the list of trials

Most behavioral experiments follow a similar logic. On each trial some stimulus is presented, and a response is collected. Usually, there are many kinds of stimuli, and each belong to various conditions. The list of trials for an experiment specifies which stimulus (and its conditions) occur on which trial. Depending on the purposes of the experiment, it is usually desirable to control the order with which stimuli are presented. Sometimes the order might be deliberately randomized, or deliberately sequenced. There are other decisions such as determining the frequency with which particular stimuli, or conditions occur across the trials.

Consider the problem of running a simple Stroop experiment with 4 colors, Red, Blue, Green, and Yellow. These four colors can be combined to produce 4 congruent items (blue in blue, red in red, green in green, and yellow in yellow), and 12 incongruent items (red in blue, red in green, etc.). So far we have 2 conditions, and 16 items. Typically, Stroop experiments are conducted with an equal proportion of congruent and incongruent trials. How can this be accomplished? There are 12 incongruent items and only 4 congruent items. If we present all 12 incongruent items, then the congruent items need to be presented 3 times each ($4 \times 3 = 12$). So, at a minimum, we need to have 24 trials (12 congruent and 12 incongruent). And, if we want to run more trials, and keep the number of congruent

and incongruent trials balanced, then we would run trials in multiples of 24. Let's run 96 trials (24×4).

So, how to make a trial list with 96 trials that conforms to the above demands? One simple strategy would be to write the list as a table, just like the one above by hand. Once the first 24 items are all written down, then they could be copy and pasted below to make the list 96 trials long. Sometimes this simple approaches are the fastest and easiest. Let's explore how to create the list using scripts in LIVECODE.

The following scripts are assumed to be placed in a button, and the results are placed into a text field

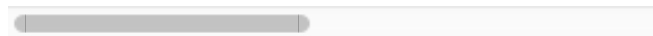
```

on mouseUp
  put empty into field 1

  put "red blue green yellow" in
  put "red blue green yellow" in

  repeat for each word n in Strc
    repeat for each word j in
      if n is j then
        put "congruent" in
      else
        put "incongruent"
      end if
      put n & tab & j & tab
    end repeat
  end repeat
end mouseUp

```



The text field should now contain lines with the following 16 lines, each representing 4 of the possible congruent items, and 12 of the possible incongruent items.

red	red	congruent
red	blue	incongruent
red	green	incongruent
red	yellow	incongruent
blue	red	incongruent
blue	blue	congruent
blue	green	incongruent
blue	yellow	incongruent
green	red	incongruent
green	blue	incongruent
green	green	congruent
green	yellow	incongruent
yellow	red	incongruent
yellow	blue	incongruent
yellow	green	incongruent
yellow	yellow	congruent

Although this is the full list of possible stimuli, it is not the full list that balances the number of congruent and incongruent items. Remember, we need to make sure that each congruent item appears three times each. Consider how the modified code accomplishes this task:

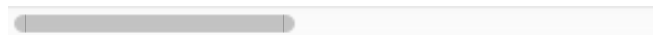
```

on mouseUp
  put empty into field 1

  put "red blue green yellow" in
  put "red blue green yellow" in

  repeat for each word n in Strc
    repeat for each word j in
      if n is j then
        put "congruent" in
        put n & tab & j &
        put n & tab & j &
        put n & tab & j &
      else
        put "incongruent"
        put n & tab & j &
      end if
    end repeat
  end repeat
end mouseUp

```



Here is the result, as you can see all 24 trials are present.

red	red	congruent
red	red	congruent
red	red	congruent
red	blue	incongruent
red	green	incongruent
red	yellow	incongruent
blue	red	incongruent
blue	blue	congruent
blue	blue	congruent
blue	blue	congruent
blue	green	incongruent
blue	yellow	incongruent
green	red	incongruent
green	blue	incongruent
green	green	congruent
green	green	congruent
green	green	congruent
green	yellow	incongruent
yellow	red	incongruent
yellow	blue	incongruent
yellow	green	incongruent
yellow	yellow	congruent
yellow	yellow	congruent
yellow	yellow	congruent

It would be easy to change the script to make 96 trials. Just add another repeat loop to repeat the process 4 total times.

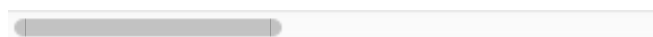
```

on mouseUp
  put empty into field 1

  put "red blue green yellow" in
  put "red blue green yellow" in

  repeat 4 times
    repeat for each word n in
      repeat for each word j
        if n is j then
          put "congruent
          put n & tab &
          put n & tab &
          put n & tab &
        else
          put "incongrue
          put n & tab &
        end if
      end repeat
    end repeat
  end repeat
end mouseUp

```



At this point, you should have a list of 96 trials that satisfy the constraints of the design from above. A next step is to randomize the order of these trials so that participants are unable to predict the order of items and responses. Lines in a text file can easily be randomized using the sort function. For example:

```
sort lines of field 1 by randc
```



The above line will randomize the order of the lines in field 1. Assuming that field 1 contains the 96 Stroop trials, this list will now be randomized. Each line refers to each trial. If you want to be more specific, and add trial numbers, consider the following code:

```

on mouseUp
  put empty into field 1

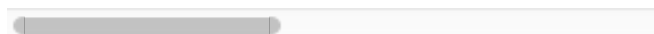
  put "red blue green yellow" in
  put "red blue green yellow" in

  repeat 4 times
    repeat for each word n in
      repeat for each word j
        if n is j then
          put "congruent
          put n & tab &
          put n & tab &
          put n & tab &
        else
          put "incongrue
          put n & tab &
        end if
      end repeat
    end repeat
  end repeat

  sort lines of field 1 by randc

  put field 1 into tempVar
  repeat for each line k in temp
    add 1 to lineCounter
    put lineCounter & tab & k
  end repeat
  put outputWithTrialNumbers into
end mouseUp

```



Here is the result from the first 24 lines of the output from this script:

1	blue	green	incongruent
2	blue	blue	congruent
3	yellow	red	incongruent
4	blue	red	incongruent
5	green	green	congruent
6	red	red	congruent
7	red	red	congruent
8	red	green	incongruent
9	yellow	green	incongruent
10	yellow	yellow	congruent
11	green	green	congruent
12	blue	blue	congruent
13	red	red	congruent
14	blue	yellow	incongruent
15	yellow	yellow	congruent
16	blue	blue	congruent
17	yellow	blue	incongruent
18	green	blue	incongruent
19	blue	blue	congruent
20	green	yellow	incongruent
21	red	yellow	incongruent
22	yellow	yellow	congruent
23	blue	blue	congruent
24	yellow	yellow	congruent

The full 96 trials have now been randomized, and the first part of creating the Stroop experiment is “finished”. There can often be many more considerations at this present stage. Note that this list of 24 trials only contains 10

incongruent trials, and 14 congruent trials. This is due to the fact that all 96 trials were first compiled, and then randomized. This means that the 12 congruent and 12 incongruent trials will not necessarily be distributed evenly within each bin of 24 trials. Issues like this arise when programming longer experiments. For example, imagine an experiment with 8 blocks of 96 trials. It wouldn't make sense to create the list of trials in the same way that we did here. Instead, each block of 96 trials should be separately randomized to ensure that each block is balanced with the same number of items and conditions as each other block. In the present example, an experimenter may want to make other changes to the trial list, for example to vary the relative proportion of congruent vs. incongruent items, or to change the names of the words, or colors and so on.

8.5.5 Interpreting the list to run each trial

The next bits of code will show how to interpret the contents of each line of the trial list to run each trial of the experiment. The code assumes that you have a card with the following elements.

1. A text field named trials that contains the list of 96 trials

2. An empty text field named StroopStimDisplay placed in the center of the card (this will display the Stroop item)
3. An empty text field named dataVar (to store the data on each trial)
4. A button named begin placed in the bottom center of card (to initiate the experiment)

8.5.6 Presenting a stimulus

The following code will be placed in the button Begin. The logic of the code is as follows. Each press of the button should initiate the next trial. The button will increment a counter (that counts 1 to 96 for each trial). The value of the counter will serve as an index into the appropriate line from the trials list (1 for line 1, 2 for line 2 etc.). The 2nd word in each line will be presented as the word on each trial, and the 3rd word will be used to set the font color of the word.

```

on mouseUp
global TrialCount
add 1 to TrialCount
put line TrialCount of field
if TrialCount <97 then
    put word 2 of trialTemp in
    put word 3 of trialTemp in
    put wordVar into field Str
    set the foregroundcolor of
    show field StroopStimDispl
    put the milliseconds into
    put trialTemp & tab & onse
else
    put "Finished" into field
    show field StroopStimDispl
end if
end mouseUp

```



Now, everytime the button is clicked, a new Stroop item should appear in the field StroopStimDisplay. As well, the trial information for each trial should be stored into the field dataVar.

8.5.7 Collecting a response

After each trial is presented on screen, the participant should be able to enter a response to identify the ink color of the presented stimulus. Response collection will be handled by the card script. The goal of response collection could be to

1. Save the identity of the response key that was pressed
2. Save the time when the response occurred
3. Trigger the next trial automatically so the button doesn't need to be pressed manually to start the next trial

The following code goes in the card script:

```
on keydown whichkey
  put the milliseconds into
  put RT & tab & whichkey &
  hide field StroopStimDispl
  send mouseUp to button beg
end keydown
```



With the code in the button and the card in place, the basic elements of interpreting the trial list to present an item and collect and record a response are in place. The code will run for 96 trials, and then a message saying “finished” will appear in the field StroopStimDisplay.

8.6 *Look and feel*

The above code shows the basic logic that goes into coding a multi-trial Stroop experiment. There are many additions that could be made to make the

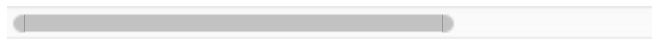
experiment look better and run more smoothly. The size of the font in the field StroopStimDisplay can be made larger (using the property inspector). The look of the text field can be changed. A basic text field usually starts with a visible border. This can be turned off so that the word appears on its own without a border. The data field can be hidden. The trials field can be hidden.

For most experiments, including the demo Stroop experiment in the repository, it is usually desirable to have several cards in your stack. The first card can act as an opening screen, perhaps with a title for the experiment, and a text field for entering subject numbers or counterbalancing codes. There might be a button that is pressed to begin the experiment and take the subject to a second card that displays instructions. This button can also be used to insert the code that creates the trial list. At this point, it would also be important to initialize aspects of the experiment. For example, the button could have code that empties the contents of data field, sets the global counter to zero, and hides or shows important elements on different cards. The card containing the instructions could also have a button that sends the subject to a third card containing the code from above that implements the Stroop experiment. Finally, when the trials are over, the participant could automatically be sent to

a final card that displays a message like “Thank you for your participation, please find the experimenter”.

Many experiment builder programs like e-prime or PsychoPy present trials on a completely blank computer screen. This can be accomplished in LIVECODE by hiding things like the menubar. Check out what the following commands do:

```
hide menubar
show menubar
set the decorations of this st
set the decorations of this st
set the backdrop to black
set the backdrop to none
```



In combination with setting the background colors of the card and fields, it is possible to create a completely blank screen for presenting stimuli, if so desired.

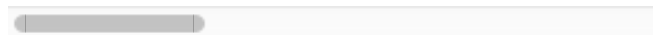
8.6.1 Preventing unwanted actions

The example code shown so far does implement a basic Stroop experiment, however the code allows for some unwanted behavior. We expect that subjects would not press buttons until a stimulus is presented on the screen, and then only press one button (say r to identify a red stimulus) and not many buttons. Unfortunately, even the best

subjects will press buttons in error and at inopportune moments. The current version of the code is vulnerable to this kind of button pressing. For example, try pressing the “r” button three times in rapid succession. You should see three trials automatically be presented in rapid succession on the screen. To prevent this kind of behavior, we need to build in logical switches or toggles into the code that allow responses to be collected only during certain times. Fortunately, this is very easy. Consider a variable called `toggleResponse`. If the contents of the variable is 0 then we do not allow response collection, but if the contents is 1, then we do allow it. Consider, the following code implementing the toggle.

button begin script

```
on mouseUp
global TrialCount
global toggleResponse -- new
put 0 into toggleResponse --
add 1 to TrialCount
put line TrialCount of field
if TrialCount <97 then
    put word 2 of trialTemp in
    put word 3 of trialTemp in
    put wordVar into field Str
    set the foregroundcolor of
    show field StroopStimDispl
    put 1 into toggleResponse
    put the milliseconds into
    put trialTemp & tab & onse
else
    put "Finished" into field
    show field StroopStimDispl
end if
end mouseUp
```

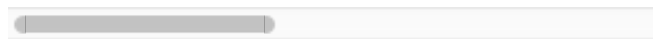


Card script

```

on keydown whichkey
    global toggleResponse -- c
    put the milliseconds into
    if toggleResponse is 1 the
        put 0 into toggleRespo
        put RT & tab & whichke
        hide field StroopStimD
        send mouseUp to buttor
    end if
end keydown

```



The toggling logic implemented above will only allow response collection to occur after a Stroop stimulus has been presented. It also only collects one response before turning off, so it prevents multiple key presses from triggering multiple trials in rapid succession. It is worth noting that this kind of toggle is not always desired. Sometimes it is desirable to collect premature responses that occur before the presentation of a stimulus. In this case a different toggle logic would need to be implemented. This shows a great example of the challenges of programming experiments from scratch (having to deal with these issues), as well as the power of programming from scratch (it is possible to do what you want, but you have to figure out how to do it).

8.7 *Saving the data*

At the end of the experiment the field containing the data should have 96 lines.

One option for saving the data-file would be to

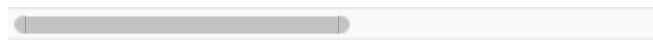
1. Make the data field visible
2. Copy and paste the data into a text file
3. Save the text-file to a folder on your computer

Another option is to use LIVECODE to create a file and save the contents to that file. Here is a general strategy for saving text to files in LIVECODE.

1. Create a folder for your experiment, then save your experiment stack into this folder.
2. Inside your experiment folder you should now see the LIVECODE stack. Create a new folder inside the current folder called data.
3. Your experiment folder should now contain the LIVECODE stack, and a folder called data
4. Create a new button on your card, give it a name like savedata (or incorporate this code into your script in another way)

The code for the button script should read as follows (note: this code assumes that your data is in a field called dataVar):

```
on mouseUp
    put the filename of this s
    set the itemDelimiter to t
    delete the last item of fr
    put "\data\" after fname
    put field dataVar into url
end mouseUp
```



If your data field has text inside it, then clicking this button should create a new field called data.txt inside the data folder that you created on your hard-drive. This kind of code can be changed to add more functionality. For example, instead of using “data” as the filename, a subject number could be used. Note that in the demo version there is a field on the first card for entering a subject number, the contents of this field could be used to define the filename for the data file.

8.8 *Data-analysis*

If you have created the Stroop experiment and run yourself in it, then you should have created a datafile with 96 lines that looks something like the shortened version below:

1	blue	blue	congruent	
2	blue	blue	congruent	
3	yellow	yellow	congruent	
4	red	blue	incongruent	13
5	red	blue	incongruent	13
6	yellow	yellow	congruent	
7	yellow	yellow	congruent	
8	blue	blue	congruent	
9	green	red	incongruent	13
10	green	blue	incongruent	
11	blue	blue	congruent	
12	green	green	congruent	
13	blue	yellow	incongruent	
14	red	red	congruent	138999
15	yellow	yellow	congruent	

The next step might be to get the mean reaction times for congruent vs incongruent trials. This involves finding all of the reaction times for each trial, then separating them into congruent and incongruent types, and finally computing the average for each type.

Here is a straightforward approach using a repeat loop. This assumes you have opened a new stack for accomplishing the analysis. Field 1 could contain the data, and field 2 could be reserved for the output of the analysis.

```
repeat for each line n in fiel
  put word 6 of n – word 5 c
  if word 4 of is congruent
    put RT & "," after Cor
  else
    put RT & "," after Inc
  end if
end repeat
put "Congruent" & tab & "Incor
put average(CongruentVar) & ta
```

