Laboratory 11

Objective

In this week’s laboratory we will

The aim of the laboratory is to help you:

1. Practice translating handwritten pseudocode to working Scribble scripts,
2. Gain some experience in typing program code and in correcting syntax errors,
3. Learn to be able to create applications that communicate via Scribble Mesh across a network.

Required resources

To perform this laboratory at Monash:

• You will need a Windows-based PC in a Monash student laboratory
• Your authcate credentials to login to the computer and to the unit-web site on Moodle, access and watch Web-hosted videos, search the Web and download an image file.

To perform this laboratory at home on your own computer:

• A working copy of Scribble
• Your authcate credentials to login to the computer and to the unit-web site on Moodle
• Web access that allows you to access and watch videos, search the web and download an image file.

Tasks (on and off-campus students)

1. What would Scribble be like if you typed the code? As you know, the reason we have used Scribble (a variant of the MIT developed of Scratch), rather than some other programming language in this unit, was to avoid syntax errors. You have created applications by dropping and dragging code-blocks in an editor. Most programming languages have to be typed using some
sort of a text editor. The text you type, if the program is to execute as you intended, has to match the rules (the syntax rules) that define the language. As the text files that make up an application are processed (either by an interpreter - for example Java - or a compiler - for example C++ - the language parser has to break up the text in the program into symbols and match them against the allowed symbols in the language. Sometimes this process requires several passes through the text file before the language processor understands what the program has to do and can prepare either to perform the application (an interpreted language) or create a set of machine code instructions (a compiled language) that can execute the application later.

Typing code - once you know the language rules - can be a much faster way to create programs than dropping and dragging code blocks. However, it's very easy to make mistakes. Sometimes these mistakes aren't easy to spot. Many code editors (though often you can just use a plain text editor) provide help, spotting typing mistakes (in a way similar to a spell checker) or completing code sections as you type them — for example automatically ending an IF block by inserting the ENDIF without you having to type that.

When you finish this unit and next semester (or some other later semester), do another programming unit, you will find you are able to design the code reasonably well - that's been our aim in teaching Scribble. However, you might be a little thrown by having to type code and deal with syntax errors.

In this set of laboratory exercises you will get an idea of what it's like to type code. A simple text editor has been created for you type in Scratch code (Scribble isn't available yet, but Scratch is so close it won't matter) and convert it to a graphic image. You can find it on Moodle at the following URL:

http://moodle.vle.monash.edu/mod/page/view.php?id=1666164

When you visit that page you should see something like the following screen. Notice the items marks (1) and (2).
Item (1) is a place where you can type in Scratch code, item (2) is a place where the code will appear as a graphic image. The code won’t run and can’t be imported into a Scratch or Scribble project. Open that page and text box on that page and type the following (simple) code (into area (1)):

```
when green flag clicked
say [Hello]
```

When you type that code, you should see a graphic rendering of the code in area (2). If it doesn’t happen as you type, just click the [Render] button underneath area 1.

Nice! Works well. The program that is running (all language interpreters and compliers are - of course - computer programs) reads the text you type in, parses it to attempt to recognise the symbols that are used, matches them to the rules of the program language (Scratch), and then in this case converts that to a graphic representation of the code typed. Typing can be very efficient. However, the problems come with making mistakes - which are impossible to avoid - and having to remember the specifics of often very exact and demanding syntax. Just with the simple example we have, make a tiny change to the code. Remove the “d” from the statement “When Green Flag Clicked”. The program is no longer able to match the code typed with the set of rules for the language and an error is generated.

In this case it’s not a serious error and the graphic rendering simply replaces the symbol it doesn’t recognise with a red code block. In a programming environment where more than a graphic symbol is being generated, the error can cause bigger problems than a graphic being drawn incorrectly. Code might still “compile” - the language rules might be obeyed - but in some incorrect way. The programmer might think they have typed their code correctly, and their program complier might not report any errors but the code won’t run correctly. More typically, the program won’t compile and they will be directed to a line number and asked to fix the error (and then the next and the next as each error is worked through in turn). This can be especially tricky with control code blocks (like loops and IF blocks). If the start and ends of those blocks aren’t typed correctly, it can be very difficult to work out what’s wrong.
Let's see how our tool copes with loops and IF code blocks. Type (you can copy and paste it, but don’t - actually type it - so you can watch the rendering happen as you type, you can literally see that parsing program recognise the symbols as you type them) the following code segment into the rendering tool.

```
when gf clicked
forever
  turn cw (15) degrees
  say [Hello!] for (2) secs
  if <mouse down?> then
    change [mouse clicks v] by (1)
  end
```

If you type that code correctly, you should see the following rendering of the code. (Note the shorthand `gf` for `green flag` - either are recognised. Shortcuts like that are common in many languages - which is great if you know what you are doing, but can make the code harder to read if you are just learning.).

Examine the code … note how when it was typed for you the code was indented. Often, in text based languages that kind of indentation is used by programmers to help them work out where code structures begin and end. It’s the same as the “C” shapes and colour coding that Scribble provides. The shapes and the colours make it easy to see what’s in a loop or IF statement. Without using indentation, it can be very hard - esp. in a highly nested block of code - for a programmer to be sure where a structure begins and ends. A misplaced “end” can cause all sorts of problems for a programmer.
Looking again at our code … the code you just typed … there is a mistake. Something has been left out, but the parser is able to work it out and renders the code correctly. Look at the code. Does the “end” belong to the IF block, or the FOREVER block? When typing Scratch code like this, they both should have an “end”. The parser knows that the FOREVER block must have an end so it assumes it’s there. Type “end” as the last line of code. So the code now reads.

```scratch
when gf clicked
forever
  turn cw (15) degrees
  say [Hello!] for (2) secs
  if <mouse down?> then
    change [mouse clicks v] by (1)
  end
end
```

There should be no change to graphic representation of the code that you see.

1.1. Spotting errors in typed code. Copy and paste each of the following code segments into the code editor. (There is a short cut to help you in the explanatory text on the edit page). They each contain a syntax errors. Try to find the error(s) and fix it so that there are no red code blocks in the graphic representation of the code. (Note some of these code segments have featured in past exams).

<table>
<thead>
<tr>
<th>Code to copy &amp; paste</th>
<th>Corrected code representation</th>
</tr>
</thead>
</table>
| `when gf is clicked` | ![Corrected code](image1)
| set x to 6            | ![Corrected code](image2) |
| set y to [7]          |                             |
| say < x + y >         |                             |

| `if ((x) > (5)) and ((x) < (10))` | ![Corrected code](image3) |
| `say (Hello!)`                  |                             |
| `end`                            |                             |
2. **Diffusion limited aggregation.** We are going to build a *Scribble* application that simulates a process found in nature called diffusion limited aggregation (DLA). You can read about the process here: [http://en.wikipedia.org/wiki/Diffusion-limited_aggregation](http://en.wikipedia.org/wiki/Diffusion-limited_aggregation)

In our application a sprite is going to randomly walk over the stage (simulating a particle undergoing Brownian motion). When the sprite touches the color black (a black blob will be positioned in the middle of the stage at the start of the process). It will simulate aggregating there (by stopping motion and stamping its image there). It will then move to a random position on the edge of the stage and begin the process again. To start working on this application download this *Scribble* application (the link URL follows) and use it as a starting point. It has a light blue canvas background and two sprites (black in color). One is the sprite representing the particle that will perform the random walk.


Spend a moment checking that application. Soon we will build a script in the sprite named *particle*. (The other sprite, named *centre*, has no script, but will sit in the middle of the stage, waiting for the particle to bump into it.)
The script you need to create to simulate DLA is shown below in pseudocode form. This really is pseudocode - it’s not exactly Scribble or Scratch syntax. It is Scribble and Scratch like, but not 100% right in terms of syntax.

```plaintext
when green flag is clicked
clear
forever
  if < (pick random (0) to (1) = 1 >
    if < (pick random (0) to (1) = 1 >
      go to x: (-240) y: (pick random (-180) to (180))
    else
      go to x: (240) y: (pick random (-180) to (180))
  else
    if < (pick random (0) to (1) = 1 >
      go to x: (pick random (-240) to (240) y: (-180))
    else
      go to x: (pick random (-240) to (240) y: (180))
  repeat until < touching color (black) ? >
  if on edge, bounce
  move (2) steps
  turn (pick random (-15) to (15)) degrees
  stamp
```

There are two tasks we are going to perform using this pseudocode. One of them will be to create the script in Scribble, the other will be to type the code into our code editor and generate and error free graphic representation of the code. For each task, we will time how long it task you. Use your smartphone, watch or a web-based timer (for example: [http://www.online-stopwatch.com/full-screen-stopwatch/](http://www.online-stopwatch.com/full-screen-stopwatch/)) to keep the time.

2.1. **Timing how long it takes to type the code.** Start your timer, then type the pseudocode above into the code editor. Stop the timer when you have no errors in the graphic representation of the code. Sadly, we can’t run the code in the editor, but its a reasonable simulation of how long it would have taken to type. Stop your timer when you have finished and take a note of the time.

2.2. **Building the application in Scribble.** Use the application starting point you downloaded (2) above and make the application work. Build the script represented by the pseudocode using the drop and drag Scribble code editor in the sprite named `particle`. Start your timer before you begging and stop it when you have finished. Make a note of the time taken to build the script.

2.3. Which was faster to build - the actual script or the typed graphic representation of the code. Why do you think that was? What would make (in terms of editor support, or your own practice/habits) typing code faster?

2.4. Once you have built the application, run it and watch it work. You might like to turn on turbo mode to make it run faster. Leave it for a while and come back to it to see the pattern created. The process it is simulating is common in nature. What you are watching on screen is very similar, for example, to the way snowflakes are formed.
3. **Coffee with a friend.** (On-campus students: Find a partner. Off-campus students, if you are using Windows you can be your own partner and run two Scribble session on the same computer. If you are using a Mac, you’ll have to make the Barista part of the application and connect to a copy of the CoffeeOrders application over the network.) We are going to use Mesh - a way of linking applications over a network in Scribble - to make a system that might be used at a coffee shop. One of you will make the application that takes orders (called CoffeeOrders). This application is the one staff taking orders would use to record the coffees that have been ordered and the people who ordered them. The other will show a list of current orders so that the Barista knows what they have to make. When the coffees are ready they can be deleted from the list. This application will be called Barista.

3.1. **CoffeeOrders.** Download the application starting point from the following URL:


This application starting point contains the background needed for the application and a sprite named ProcessOrder. Start making the application by creating global variables named: OrderTaken, Name, Size, Sugars, and Type. Some of these variables will be filled with values and made available via mesh to the Barista application. The Sprite ProcessOrder will need just two scripts. These scripts will ask the user to enter information about each coffee ordered. When the details about a coffee have been entered a broadcast message will be sent. That message will be received by the Barista application. Once the Barista application has processed that order - it will send a message, which will be received by this application. A change to a flag variable will allow order processing, which would have halted to proceed. Pseudocode for the two scripts required follows.

```plaintext
when ProcessOrder clicked
ask (Enter the name of the customer) and wait
set (Name) to (answer)
script Variables (Number)
ask (How many coffees are they ordering?)
set (Number) to (answer)
repeat (Number)
  ask (What type of coffee do they want (Long Black, Latte, Cappucino etc.)?) and wait
  set (Type) to (answer)
  ask (What size do they want?) and wait
  set (Size) to (answer)
  ask (How many sugars?) and wait
  set (Sugars) to (answer)
  set (OrderTaken) to <false>
  broadcast (Order)
wait until <(OrderTaken)>
```
when I receive (OrderTaken)
set (OrderTaken) to <true>

Once you have made those scripts - and done your best to make sure they work - it’s time to make your sprite and the application available to your partner (or your other application) via the network. To do that is simple. Look at the top of the screen. You notice an option labelled “Share”. Click that and you’ll see in the menu that appears an option there called “Host Mesh”. Select that.

Once you have selected that, you will see a dialog box appear. This will show your computer’s IP address. Note that number down - you will need to give it to your partner - it’s your computers internet address. It will be used to connect to the application you just created.

When you have done that you’ll also need to share your sprite. Right click on it - in the sprite selection area - after you have started hosting a Mesh. A popup menu appears, choose the option share this sprite from the options available. When your partner is ready. Start adding coffee orders. (I’ll have a large cap, no sugar, thx).

3.2. Barista. Download the application starting point from the following URL:


This application starting point contains the background need for the application and a sprite named Sprite1. Start making the application by creating two lists (with global scope) named: CoffeeOrders and Order. A list of lists will be used to store the list of current orders. the main list will be the one named CoffeeOrders. When the application runs it will be filled with copies of the list named Order. The first item of the Order list will be the customer name, the second the type of coffee, the third the size ordered and the forth item the number of sugars they want.

The sprite will also need some variables. They can be defined with sprite level scope. name them x, y and count. As orders are filled they will be removed from the list. New orders
will be created using values entered in the other application (CoffeeOrders) and transferred to this application over the network using Mesh. The sprite will need Pseudocode for the three scripts required follows. Create the first two of them now (the third is a little more complex).

```
when greenflag clicked
    hide
    clear
    set font to (Arial)
    set font size to (10)
    broadcast (Refresh)

when I receive (Refresh)
    show
    clear
    set (y) to (110)
    set (count) to (0)
    repeat (length of (CoffeeOrders))
        change (count) by (1)
        set (x) to (-225)
        go to x: (x) y: (y)
        write (join (count) ( ))
        go to x: ((x) + (15)) y: (y)
        write ( item (1) of (item (count) of (CoffeeOrders)))
        go to x: ((x) + (100)) y: (y)
        write ( item (2) of (item (count) of (CoffeeOrders)))
        go to x: ((x) + (250)) y: (y)
        write ( item (3) of (item (count) of (CoffeeOrders)))
        go to x: ((x) + (300)) y: (y)
        write (item (4) of (item (count) of (CoffeeOrders))) ( sugars))
        change (y) by (-15)
    hide
```

Those two scripts write the values in the list CoffeeOrders on the canvas using write code blocks. The next script is the one that gets the orders that are being taken using the other application. Before we can write the script we will need to connect to that application. Get an IP number for the other application and then use the Share menu at the top of the Scribble application and choose the Join Mesh option. A dialog will appear, type into that the IP address of the other application. (If you are an off campus student using a Mac - connect to 130.194.70.191.)
Once you have done that the data in the other application will be available to this application via the sensor value code block. You will probably see a copy of the order taking sprite as well. We don’t need to see that. Select it and go to the looks code block group and just double click on the hide code block to hide the Sprite. Once you are connected, all the variables in the other application are available via the code block sensor value - they will appear in the drop down list. This is a view of that code block with the variable Name selected:

That will allow you to make the final script required for your application’s sprite. The pseudocode of that script follows.

```plaintext
when I receive (order)
delete (all) of order
add ((Name) sensor value) to (Order)
add ((Type) sensor value) to (Order)
add ((Size) sensor value) to (Order)
add ((Sugars) sensor value) to (Order)
add (copy of (order)) to (Coffeeorders)
broadcast (OrderTaken)
wait (0.5) secs
broadcast (Refresh)
```

There is one last task to complete this application. A script is needed to delete order once they have been filled. Add the script described by the following pseudocode to the canvas.

```plaintext
when canvas clicked
ask (Which order # has been completed) and wait
delete (answer) from (CoffeeOrders)
broadcast (Refresh)
```

Your application is complete. Click the green flag and wait for orders taken by your partner to appear on your screen. Once you have made the coffee, click the canvas to remove the order from the “to-do” list. Hurry your customers are waiting!
4. **On-campus students only.** If there is any time left, just work on your assignment 2. Ask your tutor for any help you need.

5. **On-campus students.** That’s it - with about 30 minutes to go in the class your tutor will distribute the in-class test for this laboratory. This is worth 1% of your mark for the unit. It should only take 15 to 20 minutes to complete. You must work on it on your own. Give your answer sheet back to your tutor at the end of the class. You will get the result next week (don’t worry, it’s not hard).

4. **Off-campus students.** On the “study guide” tab of the unit-web site for this week you will find a link to this week’s on-line test for the semester. This is worth 1% of your mark for the unit. It should only take 15 to 20 minutes to complete. You must work on it on your own. The test will be available for you to do for 1 week (you won’t be able to access it after midnight on Sunday). If you have any problem accessing the test please send Peter O’Donnell (peter.odonnell@monash.edu) an email.