

UCL/CAS Training for Teachers

Algorithms and Programming Module 1

*“When I use a word,” Humpty Dumpty said in rather a scornful tone,
 “It means just what I choose it to mean --- neither more nor less.”*
 - Lewis Carroll, *Alice’s Adventures in Wonderland & Through the Looking-Glass*

<i>Table 2: Defining or Naming the function square or sq with arguments 1, 2, 3 ...</i>				
<pre>square1 or sql -> fd1 lt fd1 lt fd1 lt fd1 lt</pre>				

WORKBOOK 3
UNPLUGGED PROGRAMMING
CONTROL STRUCTURE: FUNCTIONS
LINES, SQUARES, BUILDING BLOCKS
CRACKING THE CODE
COMPUTATIONAL THINKING

- ❖ Addressed to Teachers
 Activities are graded: easy to hard – 0 to 5*. You should attempt every activity marked without a star or marked with one *.

CONTENTS

Computational Thinking and Programming. How it Comes Together	3
Functions with Arguments	3
User Functions.....	3
A Simple user-function without Arguments about turn	4
User-functions with Arguments.....	4
Defining or Naming a function With Arguments square1, 2, 3 ... or sq1, 2, 3...	5
Definition of the function fetch1,2,3, or fe1,2,3.....	6
Mission 1 Programming with Our User-functions	7
Mission 2 Check your drawings.....	12
Mission 3 Programming with Our User-functions – Crack the Code.....	13
Mission 4 Check your drawings.....	16
Mission 5: Drawing with Penup (up) and Pendown (Down)	17
Two More Instructions: up and down --- to help us Move without Drawing.....	17
Our toolbox: Instructions and Programming Control Structures	17
Mission 6: More Functions without Arguments – Letters and Words	20
Mission 7: Different Structures for Capital Letters	23
Our toolbox: Instructions and Programming Control Structures	23
Mission 8: Other Algorithms for Drawing letters	25
A User-function rectangle or re	25

COMPUTATIONAL THINKING AND PROGRAMMING. HOW IT COMES TOGETHER

- ❖ In the simple rectilinear world (straight line and right-angle turn) that our toolbox is designed to traverse for now, two elementary building blocks have stood out in addition to the three basic moves of our pet/robot: the RETURN line and the RETURN square. Both have symmetric representations illustrated by RETURN programs (1) and (2):

```
repeat 4 [fd1 lt] .....program (1)
```

```
repeat 2 [fd1 lt lt] .....program (2)
```

They are represented in Table 1 in their symmetric forms and written in shorthand form in UPL. We now take these two programs a step further by defining them as *user-functions* and using their *user-function* forms as instructions in our toolbox.

FUNCTIONS WITH ARGUMENTS

We employ the *function* control structure in programming in order to have a toolbox which we can use to implement fully computational thinking in problem solving by programming. It's the control structure which enables us to make the building blocks of programming and, in our case, the building blocks of our drawings, because our programs are designed to have graphical results. In our toolbox the instructions

```
forward1 or fd1 for short,  
left turn, or lt  
right turn or rt
```

are also *system-functions*: names that stand for quite complicated programs, (which we don't see (abstraction) -- written in machine code). The *user-functions* that we are going to create are like *forward1* in that they use arguments 1,2,3,... We write the code in UPL to define the *user-function* in UPL, and then, as Humpty Dumpty says, the function name means what we say it means (in the definition). When the time comes to switch to the screen, we say what we mean – that is, define the function and give it a name -- in Scratch, Python, Coffeescript or Logo.

USER FUNCTIONS

The instructions in our toolbox, with the programming control structures, *sequence* and *repetition* give us the basis for working out how to build programs which draw diagrams and structures. We can create our own instructions (tools), which we think will improve or extend our drawing capability, and importantly, make it easier for us to build and read our programs. We do this by choosing a name for a *user-function* to represent such a program of instructions and making use of that name as an extra

instruction – a **user-function** -- in our toolbox. Simply, we give a name to a program, and use that name as an (shorthand) instruction instead of writing out the code every time we want to use it.

A SIMPLE USER-FUNCTION WITHOUT ARGUMENTS ABOUT TURN

But first, we define a very simple **user-function** to add to our toolbox. Often when drawing we want to turn and face in the opposite direction. To do this we could use the instruction (**system-function**) **lt** twice. So, for example, in program(2) in the return unit line program we have:

```
repeat 2 [fd1 lt lt].....program(2)
```

If we define a user-function **about turn** or **at** for short as

```
about turn or at -> lt lt
```

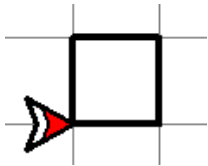

using the symbol -> to mean 'is defined as':

We can then write **at** instead of **lt lt** in program(2) as

```
repeat 2 [fd1 at].....program(2a)
```

or wherever these two instructions occur. This doesn't save us much, but it is a simple example of a **user-function** without arguments, which we can add to our toolbox. We will define more useful functions without arguments later in this Workbook.

USER-FUNCTIONS WITH ARGUMENTS

Table 1 Building Blocks: RETURN Unit Square and RETURN Unit line			
instruction: square1 or sq1	fd1 lt fd1 lt fd1 lt fd1 lt	repeat 4[fd1 lt]	
Instruction: fetch1 or fe1	fd1 at fd1 at	repeat 2[fd1 at]	

We use these two programs to illustrate how we define a function with a single argument like the *system-function forward* or **fd** with argument values 1,2,3,... and the *user-function fetch* or **fe** with arguments 1,2,3,... which we define below.

DEFINING OR NAMING A FUNCTION WITH ARGUMENTS SQUARE1, 2, 3 ... OR SQ1, 2, 3...

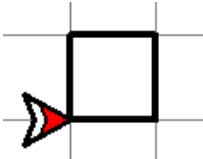
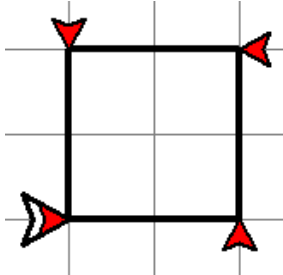
“When I use a word,” Humpty Dumpty said in rather a scornful tone,
 “It means just what I choose it to mean --- neither more nor less.”
 --- Lewis Carroll, Alice’s Adventures in Wonderland & Through the Looking-Glass

square or **sq** is the function name for the program. Together with a parameter p, where p (stands for paces) and can be replaced by argument values 1, 2, 3 ... which determine the size of the square, we can define the function as follows. We write:

```
sqp -> fdp lt fdp lt fdp lt fdp lt
```

```
or equivalently repeat 4[fdp lt]
```

We include this definition as follows in Table 2, using the symbol -> to mean ‘is defined as’:

<p>squarep -> or</p> <pre>sqp -> fdp lt fdp lt fdp lt fdp lt</pre>		
<p>Definition of function sqp</p> <p>Where p can take values 1,2,3,...</p>	<p>Instruction: sq1</p> <p>Building block: unit square (Return program)</p>	<p>Instruction: sq2</p> <p>Building block: square with side 2 paces (Return program)</p> <p>Representation of the four repeats of [fd2 lt]</p>


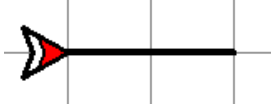
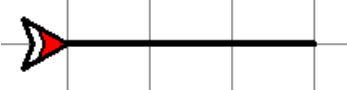
Once we have defined the *function sq*, we are no longer concerned with the details of the program it represents --- we just use **sq1** or **sq2** or **sq3**, ... as an instruction from our tool

box for our pet robot to draw a square with side length 1, 2, 3,... paces as shown in Table 2, starting and finishing at the position and direction that the pet/robot is currently holding. What is important is that the drawing of the square whenever the instruction **sq1**, or **sq2** or **sq3**,... is used executes exactly the code we have used to define it.

- ❖ What is important at this stage is that learners know what the instructions **sq1**, **sq2**, **sq3** ... mean and can use them as instructions in their code. Function definitions can be left to be formally included in code when we make the transition to programming at the screen.
- ❖ The special feature of a **function** with an argument is that we use just one (formal) definition to include all possible values of the arguments 1, 2, 3... We have included the formal definition for **user-functions** in UPL for completeness. When we move to the screen we must define the user-functions formally in Scratch 2.0/ Python 3/Coffeescript/Logo in whichever language we are using.

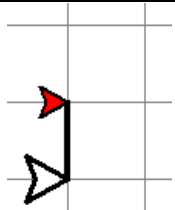
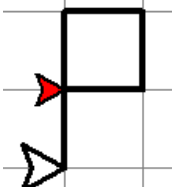
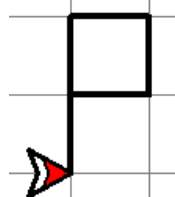
DEFINITION OF THE FUNCTION **FETCH1,2,3, OR FE1,2,3**

We also introduce another useful **user-function**, the RETURN program of **fetch** or **fe**, which like **forward** has different arguments 1,2,3... We call this **user-function** **fetch** or **fe** for short. We chose **fetch** as the name for this **function**, because it's like the command you might give to your pet dog to retrieve a stick or ball that you have thrown, and return it to you (and your dog will probably turn to face the same way ready to go again. (A RETURN **function**). It is good practice to name functions to describe, and so remind us of, what they do.

<i>Table 3 Definition of function fetch, or fe with parameter p and arguments 1, 2, 3 ...</i>	
fep -> fdp at fdp at where p can take values 1, 2, 3,...	
fetch1 fe1	
fetch2 fe2	
fetch3 fe3	

MISSION 1 PROGRAMMING WITH OUR USER-FUNCTIONS

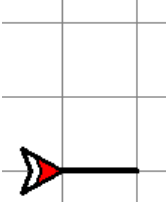
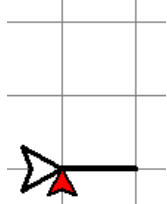
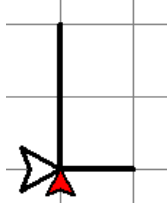
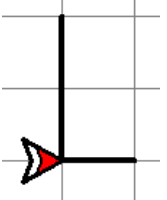
Example 1. Draw a capital P. (RETURN program) with our toolbox: `fd1`, `lt`, `rt`, and *user-functions/instructions*: **about turn** or `at`, **square** or `sq`, **fetch** or `fe`:

<i>Table 4</i>	
<i>Code</i>	<i>Drawing</i>
<code>lt fd1 rt</code>	
<code>sq1</code>	
<code>rt fd1 lt</code>	

`lt fd1 rt sq1 rt fd1 lt.....(RETURN) program(3)`

Example 2. using our toolbox: `fd1`, `lt`, `rt`, `sq`, `fe`

Draw a capital L (RETURN program).

<i>Table 5</i>	
<i>Code</i>	<i>Drawing</i>
<code>fe1</code>	
<code>lt</code>	
<code>fe2</code>	
<code>rt</code>	

`fe1 lt fe2 rt (RETURN) program(4)`

Read, Track and Crack the Code for the programs in Table 6:

<i>Table 6: Crack the Code: UPL programs – functions and repeats</i>		<i>Return Program ?</i>
1	<code>lt fe2 rt</code>	Yes
2	<code>lt lt fe2 at</code>	
3	<code>rt fe1 lt</code>	
4	<code>* fe1 lt fe1 lt fe1 lt fe1 lt</code>	
5	<code>* lt fd1 rt sq1 rt fd1 lt</code>	
6	<code>sq2</code>	
7	<code>lt fd2 lt sq1</code>	
8	<code>* lt fd2 lt sq1 lt fd2 lt</code>	
9	<code>** rt sq1 at sq1 rt</code>	
10	<code>repeat 2[at fd2]</code>	
11	<code>* fe1 rt fe2 rt felat</code>	
12	<code>** sq1 fd1 sq1 fd1</code>	

Table 7: Read, Track and Crack the Code – functions and repeats

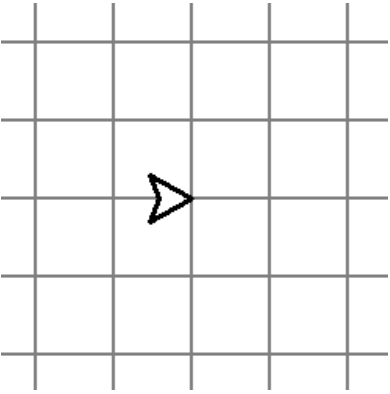
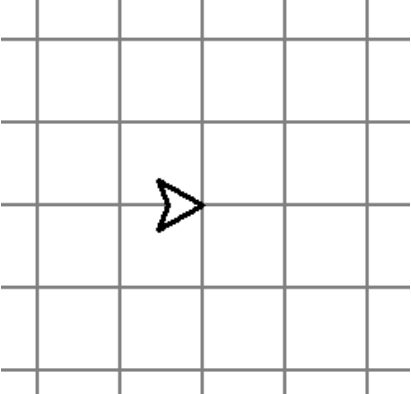
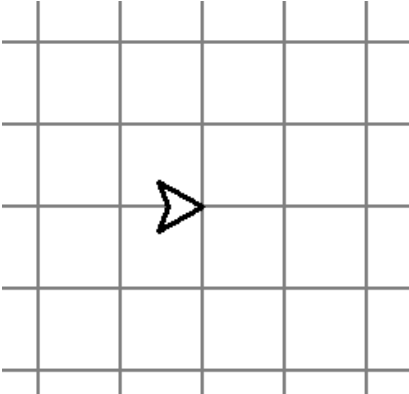
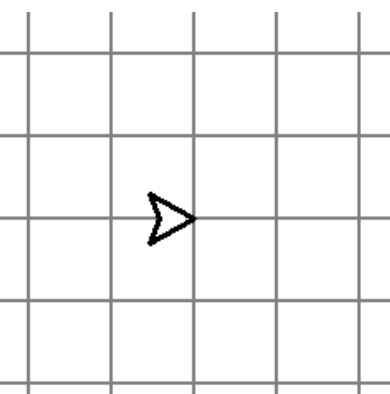
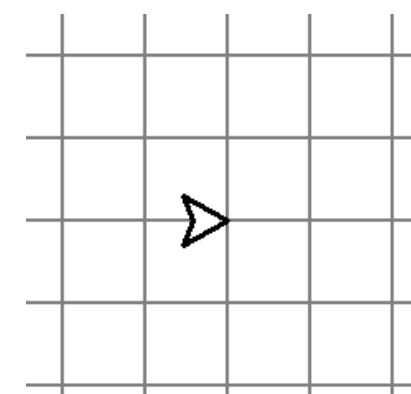
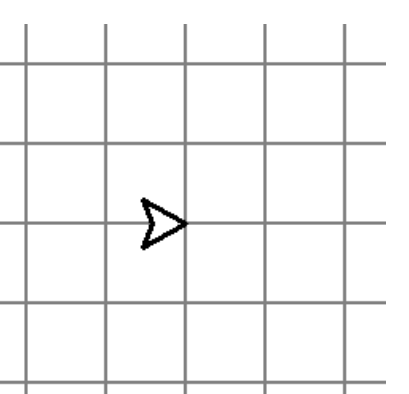
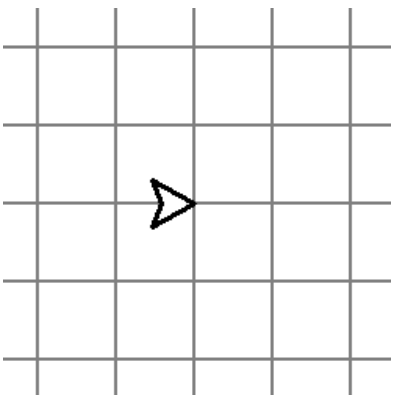
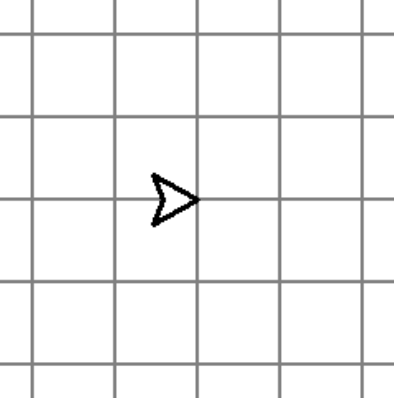
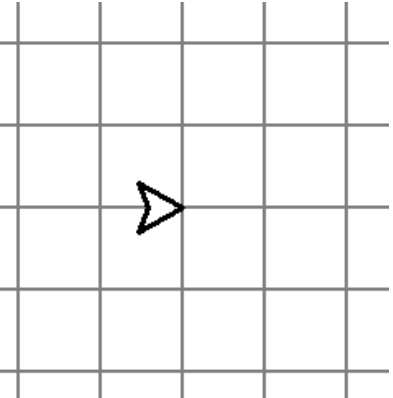
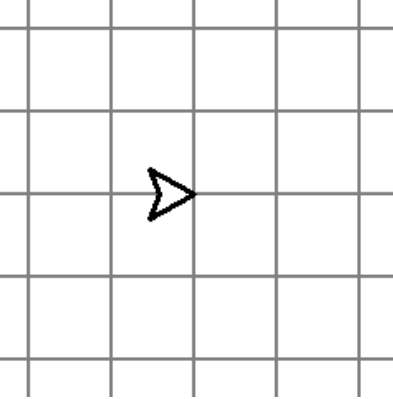
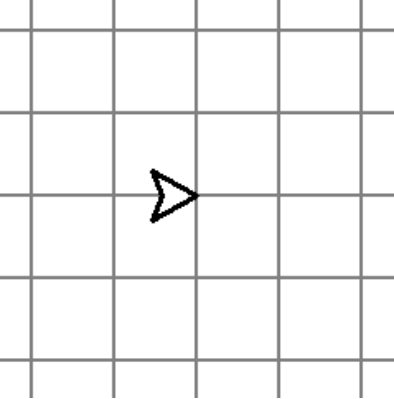
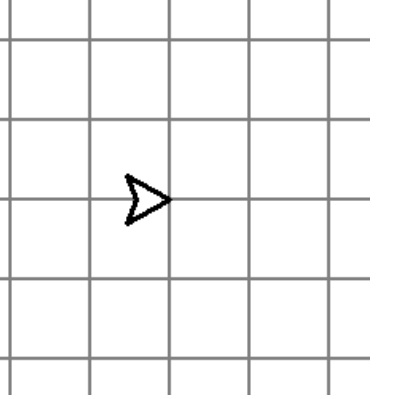
<p>lt fe2 rt</p>	<p>lt lt fe2 at</p>	<p>rt fe1 lt</p>
		
<p>1</p>	<p>2</p>	<p>3</p>
<p>* fe1 lt fe1 lt fe1 lt fe1 lt</p>	<p>* lt fd1 rt sq1 rt fd1 lt</p>	<p>sq2</p>
		
<p>4</p>	<p>5</p>	<p>6</p>

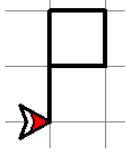
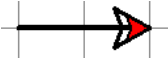

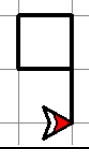

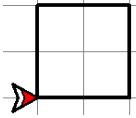
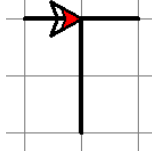
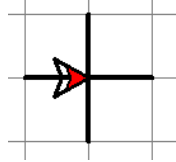
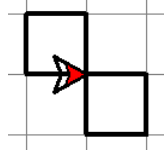
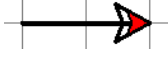
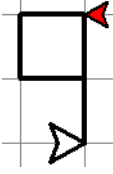
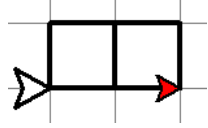
Table 7 cont'd: Read, Track and Crack the Code

<p>lt fd2 lt sql</p>	<p>* lt fd2 lt sql lt fd2 lt</p>	<p>** rt sql at sql rt</p>
		
<p>7</p>	<p>8</p>	<p>9</p>
<p>repeat 2[at fd2]</p>	<p>* fe1 rt fe2 rt fe1 at</p>	<p>** sql fd1 sql fd1</p>
		
<p>10</p>	<p>11</p>	<p>12</p>

MISSION 2 CHECK YOUR DRAWINGS

Match your drawings in Table 7 with those in the Table 9, and enter the results in Table 8 below. For example, Table 7.1 matches Table 9.E.

<i>Table 8</i>												
Table 7	1	2	3	4	5	6	7	8	9	10	11	12
Table 9	E											

<i>Table 9</i>		
		
A	B	C
		
D	E	F
		
G	H	I
		
J	K	L

MISSION 3 PROGRAMMING WITH OUR USER-FUNCTIONS – CRACK THE CODE

Read, Track and Crack the Code for the programs in Table 10:

<i>Table 10: Crack the Code: UPL programs – functions and repeats</i>		<i>Return Program ?</i>
1	<code>* sq1 at sq1 at</code>	
2	<code>* repeat 2[lt fd1 rt fe1]</code>	
3	<code>* repeat 2[fe2 lt] at</code>	
4	<code>** repeat 4[sq1 lt]</code>	
5	<code>** repeat 4[fd1 lt fd1 rt fd1 lt]</code>	
6	<code>** repeat 4[fe1 lt]</code>	
7	<code>** fd1 lt repeat 2[sq1 fd1]</code>	
8	<code>*** sq2 fd1 lt fd1 rt repeat 4[fe1 lt] fd1 rt fd1 at</code>	
9	<code>** repeat 2[sq1 fd1] at fd2 at</code>	
10	<code>*** fd1 lt repeat 2[sq1 fd1] lt fd1 lt fd2 lt</code>	
11	<code>repeat 2[fd2 lt fd1 lt]</code>	
12	<code>** fd1 lt repeat 2[fd2 lt fd1 lt] lt fd1 at</code>	

Table 11 : Read, Track and Crack the Code – functions and repetition

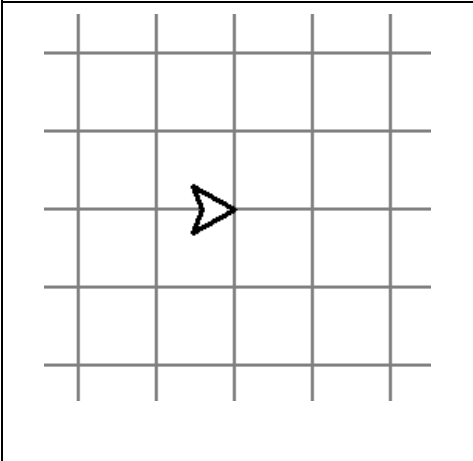
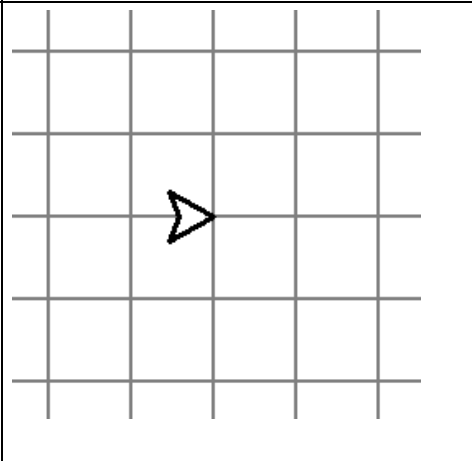
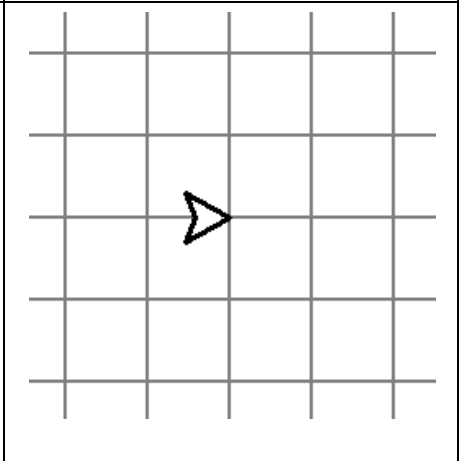
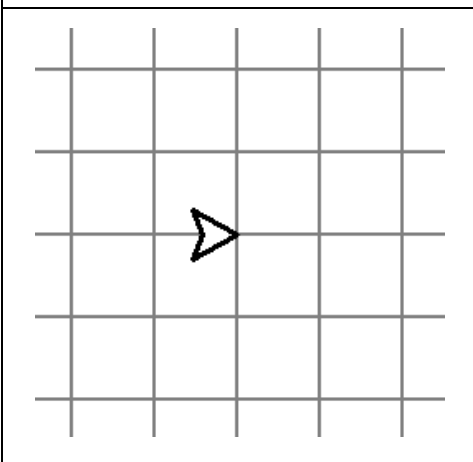
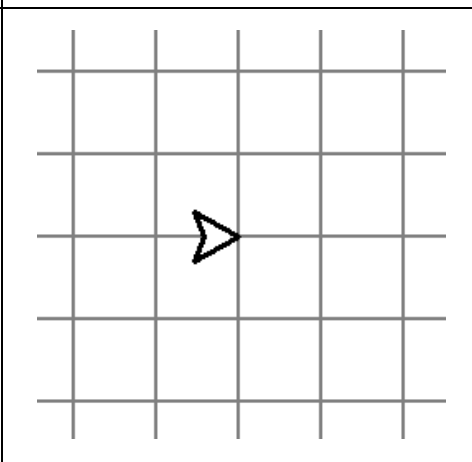
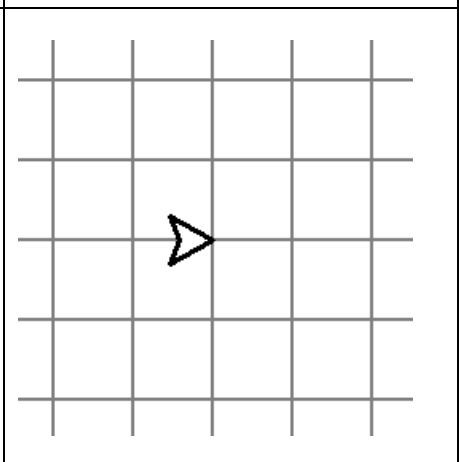
<p>* sql at sql at</p>	<p>* repeat 2[lt fd1 rt fe1]</p>	<p>* repeat 2[fe2 lt] at</p>
		
<p>1</p>	<p>2</p>	<p>3</p>
<p>** repeat 4[sql lt]</p>	<p>** repeat 4[fd1 lt fd1 rt fd1 lt]</p>	<p>** repeat 4[fe1 lt]</p>
		
<p>4</p>	<p>5</p>	<p>6</p>



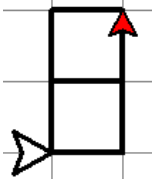
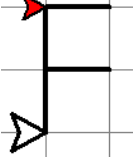
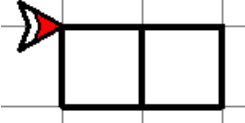
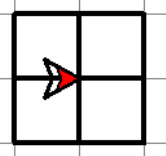
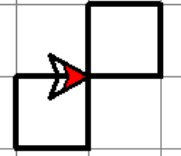
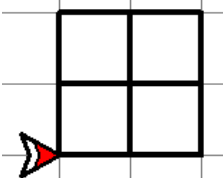
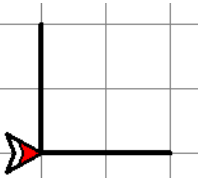
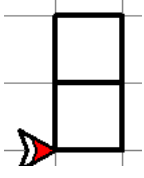
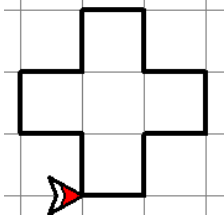
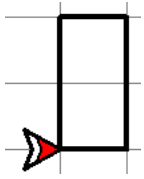
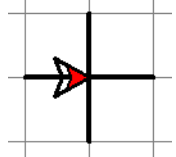
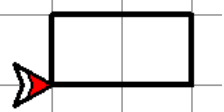
Table 11 cont'd: Read, Track and Crack the Code – functions and repetition

<pre>** fd1 lt repeat 2[sql fd1]</pre>	<pre>*** sq2 fd1 lt fd1 rt repeat 4[fe1 lt] fd1 rt fd1 at</pre>	<pre>** repeat 2[sql fd1] at fd2 at</pre>
7	8	9
<pre>***fd1 lt repeat 2[sql fd1] lt fd1 lt fd2 lt</pre>	<pre>repeat 2[fd2 lt fd1 lt]</pre>	<pre>**fd1 lt repeat 2[fd2 lt fd1 lt] lt fd1 at</pre>
10	11	12

MISSION 4 CHECK YOUR DRAWINGS

Match your drawings in Table 11 with those in the Table 13, and enter the results in Table 12 below.

<i>Table 12</i>												
Table 11	1	2	3	4	5	6	7	8	9	10	11	12
Table 13												

<i>Table 13</i>		
		
A	B	C
		
D	E	F
		
G	H	I
		
J	K	L

MISSION 5: DRAWING WITH PENUP (UP) AND PENDOWN (DOWN)

TWO MORE INSTRUCTIONS: UP AND DOWN --- TO HELP US MOVE WITHOUT DRAWING

We add two more instructions for our pet/robot from the pool of operations, common to Scratch, Python, Coffeescript and Logo, which are essential tools for drawing when we want to move the pet/robot from one position to another without drawing. This is particularly useful when we are trying to implement a human algorithm in our problem solving approach, where a human can move to any position from another on the paper without drawing.

These instructions are:

penup or up for short. Raises the pen. Instructions to move the pet/robot after this instruction do not draw until the **pendown** instruction is met.

pendown or down or pd for short. Lowers the pen. Instructions to move the pet/robot after this instruction draw until the **penup** instruction is met.

OUR TOOLBOX: INSTRUCTIONS AND PROGRAMMING CONTROL STRUCTURES

PROGRAM INSTRUCTIONS

- 1) `forward1,2,3,...` or `fd1,2,3,...`
- 2) `left turn` or `lt`
- 3) `right turn` or `rt,`
- 4) `about turn` or `at,`
- 5) `penup` or `up` for short.
- 6) `pendown` or `down` for short.
- 7) `square1,2,3...` or `sq1,2,3...`
- 8) `fetch1,2,3...` `fe1,2,3...`

EXAMPLE 1 *Crack the Code*

```
up lt fd2 at down fd2 up lt fd2 at down fd1 up at fd1 lt fd1
lt down fd1 up at fd1 lt fd1 lt
```

In this example, we have programmed the pet/robot to copy the actions of a human drawing a capital 'F'. Human and pet/robot are assumed to start from the same place (empty arrow head). Unfortunately, the pet/robot is constrained to turn and move along the grid (for the time being) in order to take up positions on the grid, so adopting a human algorithm for drawing may not be a 'good' program. See example 2, where the pet/robot takes advantage of a symmetrical drawing of 'F', and can make use of the *user-function fe*.

<i>Table 14 Copying a Human Drawing an 'F'</i>			
<code>up lt fd2 at</code>	<code>down fd2</code>	<code>up lt fd2 at</code>	<code>down fd1</code>
Move to the top of the 'F'	Draw the vertical stroke	Move to the top of the 'F' again	Draw the horizontal stroke
<code>up at fd1 lt fd1 lt</code>	<code>down fd1</code>	<code>up at fd1 lt fd1 lt</code>	
Move to the mid-point of the vertical line	Draw the horizontal stroke	Return to the starting point and complete the RETURN program	

A much simpler program to do the same thing using a drawing symmetry in the letter F.

<i>Table 15 Program to suit a pet/robot drawing an 'F' using symmetry in the drawing</i>	
<code>repeat 2[lt fd1 rt fe1]</code>	<code>rt fd2 lt</code>

Read, Track and Crack the Code for the programs in Table 16 :

Table 16 : Crack the Code: UPL programs – functions and repeats		Return Program ?
1	<code>* up lt fd2 at down fd2 (human)</code>	
2	<code>lt fe2 rt</code>	
3	<code>* up lt fd2 at down fd2 lt fd1 (human)</code>	
4	<code>** up lt fd2 at down fd2 up at fd2 rt down fd2 rt fd2 rt fd2 (human)</code>	
5	<code>repeat 4[fd2 lt]</code>	
6	<code>** up lt fd2 at down fd2 up at fd2 lt fd1 at down fd2 (human)</code>	

- 7) * Add code to programs in 1, 3 in Table 16 to make them RETURN programs.
- 8) * Write code suited to a pet/robot to produce a RETURN program for the drawing produced by the code in 3 in Table 16. Use the **user-function fe** in your code if you can.
- 9) ** Add code to programs in 4, 6 in Table 16 to make them RETURN programs.
- 10) ** Write code suited to a pet/robot to produce a RETURN program for the drawing produced by the code in 6 in Table 16. Use the **user-function fe** in your code if you can.
- 11)
 - (a) *** Write the code to copy how you draw a capital 'E'.
 - (b) *** Write code suited to a pet/robot to draw a capital 'E'.

MISSION 6: MORE FUNCTIONS WITHOUT ARGUMENTS – LETTERS AND WORDS

We have written the RETURN code to draw capital L, I, T, F, E, H, P.

Now we can define a user-function without arguments for each letter:

capitalL or cl (for short) -> fe1 lt fe2 rt

Fill in the code to define the functions for the capital letters in Table 17:

<i>Table 17 : functions without arguments for capital letters</i>		<i>Return Program ?</i>
1	(Example for capital L) cl -> fe1 lt fe2 rt	
2	cf ->	
3	ci ->	
4	ce ->	
5	ct ->	
6	ch ->	
7	cp ->	

We have a number of capital letters that we can use to write words with. But we decide to space them out with equal spaces between letters. This touches on the subject of font design.




Example 3. **Crack the Code:**

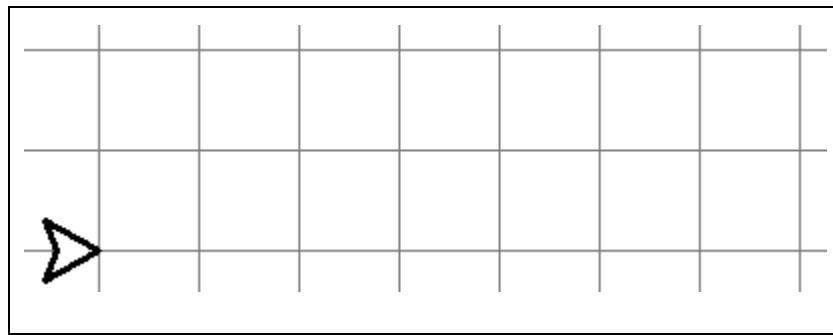
cl up fd2 down ce up fd3 down ct

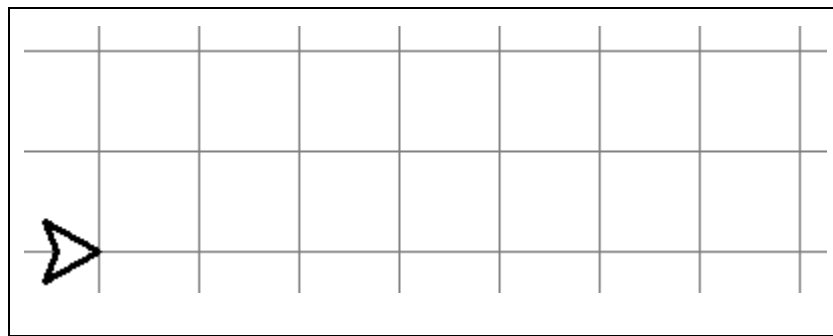
<i>Table 18: Capital Letters</i>
cl up fd2 down ce up fd3 down ct

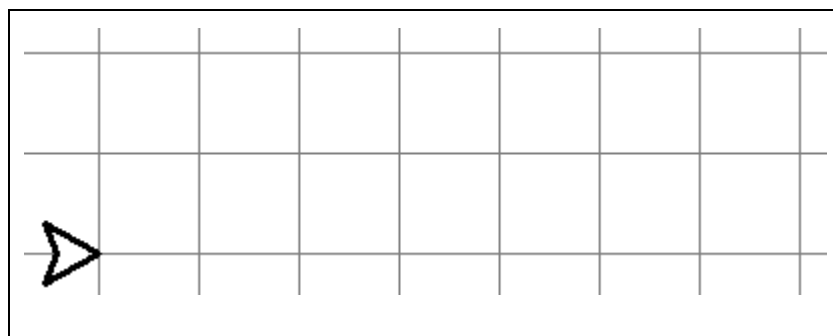
Crack the Code

With our Toolbox expanded to include functions: **ce**, **cf**, **ch**, **ci**, **cl**, **cp**, and **ct** which draw the corresponding capital letters as RETURN programs starting on a common base, **crack the code** for the word programs in Table 19 (except for 6)).

<i>Table 19</i>								
								
1) ct up fd2 down ch up fd2 down ce								
								
2) cp up fd2 down ce up fd2 down ct								
								
3) ct up fd2 down ci up fd1 down cp								


4) *** ch up fd2 down c1 cf


5) *** ct up fd2 down ce repeat 2[up fd2 down c1]


6) ****fe2 repeat 2[lt fd1 rt fe2] rt repeat 2[lt fd1 rt fe2]

7) *** Write programs for the words a) HILL b) FEE c) FEET

- 8) **** What extra repetition do you notice in Table19, 6). What extra instruction do we need to add to be able to use the **repeat** statement to reflect the final symmetry in the drawing process?

MISSION 7: DIFFERENT STRUCTURES FOR CAPITAL LETTERS

OUR TOOLBOX: INSTRUCTIONS AND PROGRAMMING CONTROL STRUCTURES

We summarise our toolbox here:

PROGRAM INSTRUCTIONS

- 9) **forward**1,2,3,... or **fd**1,2,3,...
- 10) **left turn** or **lt**
- 11) **right turn** or **rt**,

USER-FUNCTION INSTRUCTIONS

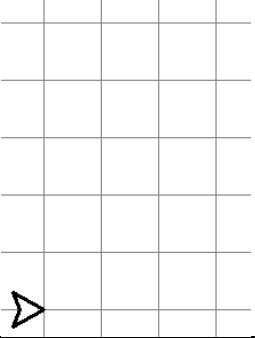

- 12) **about turn** or **at**
- 13) **square**1,2,3... or **sq**1,2,3...
- 14) **fetch**1,2,3... or **fe**1,2,3...

PROGRAMMING CONTROL STRUCTURES

- **sequence** (in the code)
- **repetition** (looking for symmetry in the drawing to use repetition in the code AT)
- **functions** (choosing and defining useful building blocks and defining **user-functions** AT and A)

Crack the code in Table 20:

1)

<i>Table 20</i>	
	
<p>A</p> <pre>repeat 4[fd1 lt fd1 rt fd1 lt]</pre>	<p>B</p> <pre>fd3 lt fd1 lt fd2 rt fd4 lt fd1 lt fd5 lt</pre>

Programs in Table 20 A, B illustrate a different approach, and therefore we use different algorithms for the programs which produce the shape/letter drawings. It's more of an 'outline the block shape' approach. Notice we have made them RETURN programs. And we have fitted the capital letter L to fill a Fibonacci box, of dimensions 5 paces high and 3 paces wide.

2) Use our Toolbox, with a block outline approach to fill the same Fibonacci box, to write RETURN programs for capital letters T, F, E and H.

3) ** Use a Fibonacci box 5 paces wide and 3 paces high to design an outline block version of capital letter M or W. Use our Toolbox to write a program to draw your design

MISSION 8: OTHER ALGORITHMS FOR DRAWING LETTERS

Rather than taking an outline approach as our algorithm to draw the capital letters and other shapes as we have done in Mission 6, we might adopt an algorithm which involved 'sticking' together different sized rectangles, or one in which we just 'stick' squares together to form the letters.

A USER-FUNCTION RECTANGLE OR RE

In this task we define another *user-function*: **rectangle** or **re** -- with arguments 1, 2, 3 ... (paces) -- which, will be useful later to complete the Aqado board.

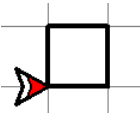
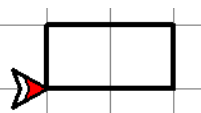
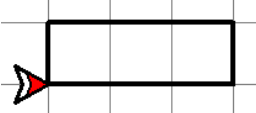

Formally we can define **rectangle** or **re** for short as a *user-function* with a parameter **p**, which stands for arguments 1, 2, 3...:

```
rectanglep or rep -> repeat 2[fdp lt fd1 lt].....definition
```

But in this task, all we need to know is that in the *user-function* we have named **rep**, we can replace the **p** in **rep** by an argument 1, 2, 3 ... to give **re1**, **re2**, **re3** ... and these in turn will draw the rectangles in Table 21. Note: the rectangle will be drawn from the point and in the direction the pet/robot is facing when the *user-function* is called as an instruction.

For example, when we use the *user-function* instruction **re3**, we substitute a 3 for **p** on both sides of the definition ->

so that **re3** is equivalent to **repeat 2[fd3 lt fd1 lt]**

<i>Table 21</i>			
			
A re1	B re2	C re3	D fd1 lt re3

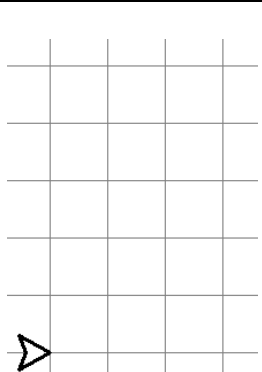
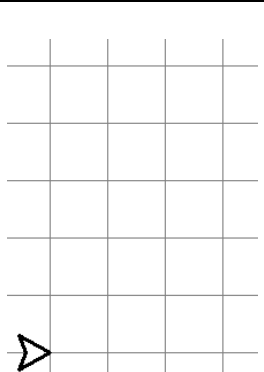
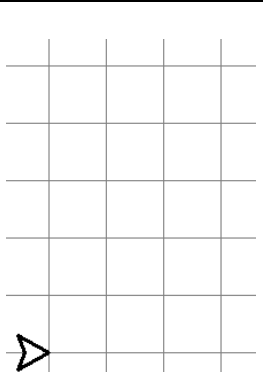
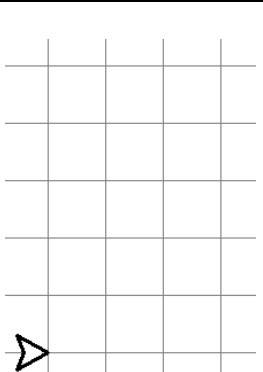
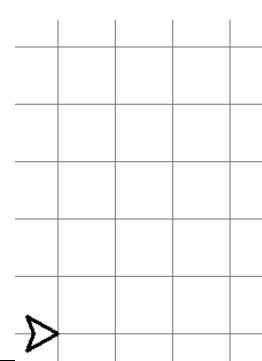
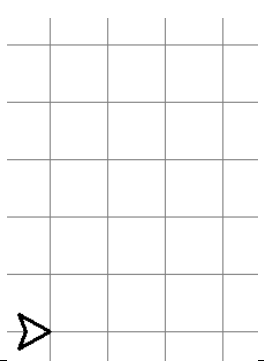
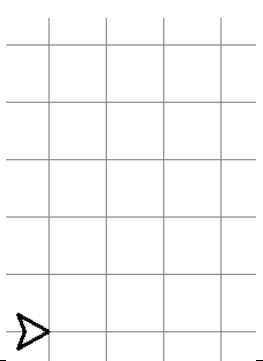
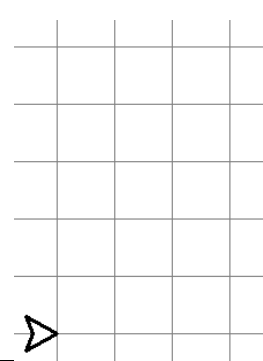
Note: **rectanglep** or **rep**

1. is defined so that every rectangle has the same width, 1 pace
2. is a RETURN program. (This means that when we use the function say, re4 that the rectangle is drawn from the point the pet/robot is in, and in the direction it is facing, and the pet/robot

returns to that position and direction. Examine carefully the program and drawing in Table 21D).

Crack the code in Table 22:

3. `fd1 lt re5 rt re2 at fd1 at`
4. `re3 lt fd1 rt fd1 lt re4 lt fd1 lt fd1 lt`
5. In Table 22, use our toolbox with **user-function rectanglep** or **rep** to draw the capital letters I, T, F, and E in Fibonacci boxes of height 5 paces and width 3 paces.
6. In Table 22, use our toolbox with the with **user-function** unit square **sq1** .
 Write a program to 'stick' unit squares together to draw the capital letters T, H in Fibonacci boxes of height 5 paces and width 3 paces.

Table 22			
			
3. <code>fd1 lt re5 rt re2 at fd1 at</code>	4. <code>re3 lt fd1 rt fd1 lt re4 lt fd1 lt fd1 lt</code>	5. Capital I	5. Capital T
			
5. Capital F	5. Capital E	6. Capital T	6. Capital H