Computer Programming

Zilogic Systems

1. Introduction

- Turtle - responds to user commands.
- Can move forward, backward, turn left, turn right.
- Has a pen fitted to its belly.
- Draws as it moves.

2. Basic Turtle Functions

- To move the turtle forward, in the direction it is facing use the `fd()` function.
- The no. of steps is to be provided as argument.

```
fd(<steps>)
forward(<steps>)
```

- To move the turtle backward, use the *bk()* function.
- As with `fd` the no. of steps is to be provided as argument.

```
bk(<steps>)
back(<steps>)
backward(<steps>)
```

- To change direction the turtle is facing use the `rt()` and `lt()` functions. `rt()` turns the turtle to the right, clockwise. `lt()` turns the turtle left, anti-clockwise.
- Both accept the angle in degrees as argument.

```
rt(<angle>)
right(<angle>)
lt(<angle>)
left(<angle>)
```

3. Drawing Basic Shapes

- The following sequence causes the turtle to draw a square.

```
fd(100)
rt(90)
fd(100)
rt(90)
fd(100)
rt(90)
fd(100)
```

4. Repetition

- To repeat a block of commands for certain no. of times, `repeat` block can be used.

```
repeat (<count>) {
       ... statements ...
```

```
}
```

- The previous program can be re-written as

```
repeat (4) {
    fd(100)
    rt(90)
}
```

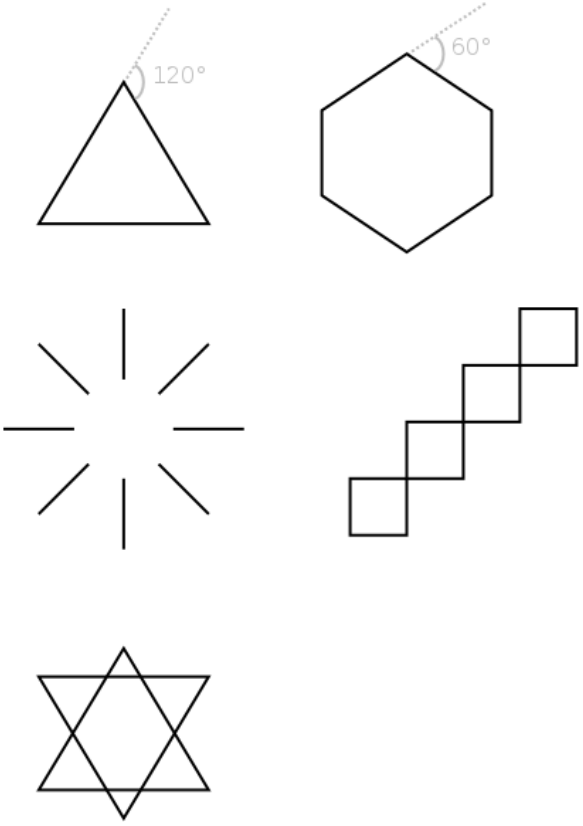
5. More Functions

- It is possible to control the pen position using `pu()` and `pd()` functions.
- The `pu()` raises the pen up, which prevents the turtle from drawing when it moves.
- The `pd()` does the exact opposite and lowers the pen down.

```
pu()
penup()
pd()
pendown()
```

6. Exercise - I

- Draw the following shapes:



7. Variables

- Variables are memory locations used to store values, that can be retrieved later on in the program.
- Variables are identified by their names. Variables start with an alphabet and can contain alphabets, numbers and underscore.

- Valid variables: `angle`, `step`, `bar1`, `pen_color`.
- Invalid variables: `9eleven`, `25`, `incorrect#`
- To store values in variables the following command can be used.

```
<variable> = <value>
```

- Example:

```
angle = 45
```

- Whenever the variable is referred to in other commands' arguments, the value stored in the variable will be used.

```
angle = 45
rt(angle)
```

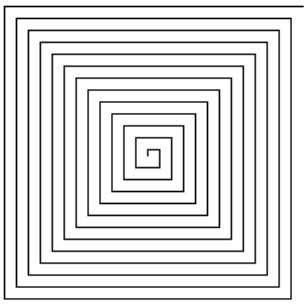
- A variable can appear wherever a value can appear.

8. Arithmetic Expressions

- The basic arithmetic operators are: `+` for addition, `*` for multiplication, `-` for subraction, `/` for division.
- Variables and values can be combined using operators to form expressions. Examples:
  - `5 + 4`
  - `a / 5`
  - `a + b * 2`
- Order of evaluation follows the BODMAS rule. Parenthesis / brackets can be use to change the order of precedence. Examples:
  - `5 - 4 * 2`, Result: `-3`
  - `(5 - 4) * 2`, Result: `-2`
- Arithmetic expressions can appear wherever a value can appear.

9. More Shapes

- The following uses variables to draw a spiral.



```
step = 4 ❶
repeat(50) {
           fd(step) ❷
           rt(90)
```

```
            step = step + 4 ❸
}
```

❶ The `step` variable keeps track of the step size.
❷ One side of the spiral is draw in every iteration, with the current step size.
❸ The `step` is incremented at every iteration.

- So far we have seen function that only accept arguments. Functions can also return values and as such functions that return values can be used in expressions just like variables.
- The `inputnum()` function can be used get a number as input from the user. The function returns the number.
- The following code fragment gets input from the user a stores the return value in `var`.

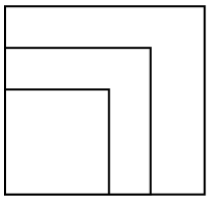
```
var = inputnum()
```

- The following code fragment draws a square with the length of the side input from the user.

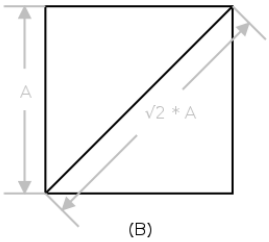
```
side = inputnum()
repeat (4) {
    fd(side)
    rt(90)
}
```

10. Exercise - II

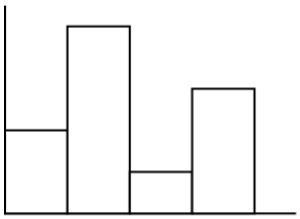
- Write a program that draws 3 squares, with size input from the user, as shown in the figure A below.
- Write a program that draws a crossed square, as shown in the figure B below.
- Write a program that inputs 4 values from the user and plots the values as a bar graph, as shown in the figure C below.



(A)



(B)



(C)

11. Defining Functions

- It is possible to break the program to smaller sub-programs (AKA functions) and invoke them.
- This way the program is much more readable, and maintainable.
- Functions can accept arguments and return values.

- The general syntax for defining functions is given below.

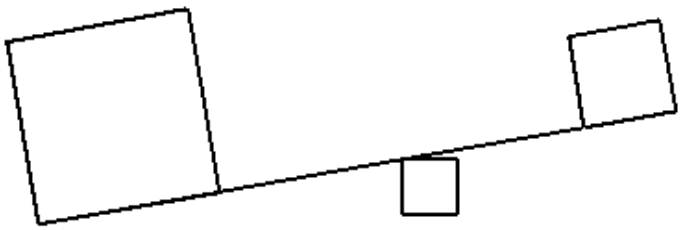
```
def <funcname>(<args>)
{
        ...
        ...
}
```

- The `funcname` is the name of the function. Functions also have the same naming convention as that of variable names.
- `args` is a comma separated list of argument variables that the function accepts. The function may return a value using the `return` statement.
- For example, we could write a function to draw a square. The function accepts the length of the side of square. The function does not return any value.

```
def square(side)
{
        repeat (4) {
                fd(side)
                rt(90)
        }
}

square(10)
```

- When `square` is invoked, the argument passed is stored in the argument variable `side`.



```
// Large Weight
lt(10)
square(100)
rt(90)

fd(200)

// Fulcrum
rt(10)
square(30)
lt(10)

fd(100)

// Small Weight
lt(90)
square(50)
```

- To return values the function should use the `return` statement as shown below.

```
return <expression>.
```

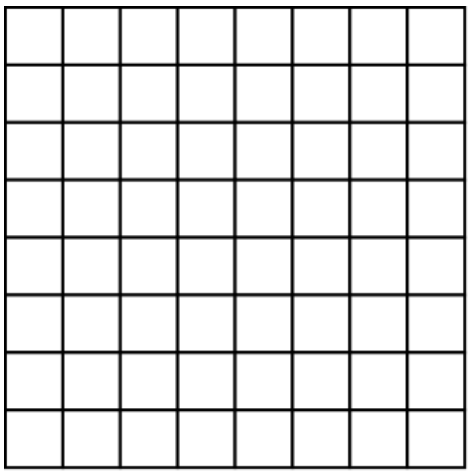
- The result of evaluation of `expression` is returned.
- For example, we could write a function that calculates the area of a triangle. The function accepts the height and base of the triangle. The function returns the area of the triangle.

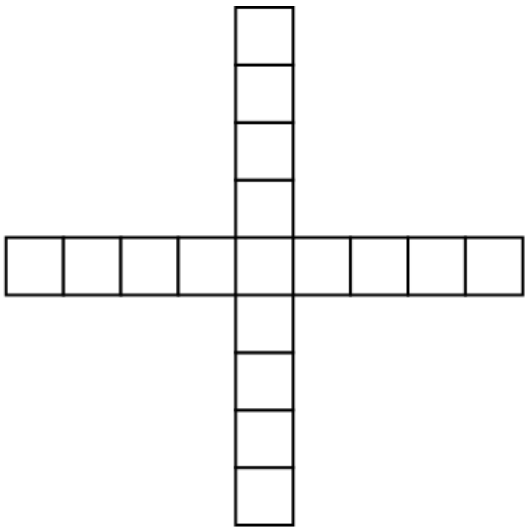
```
def area(height, base)
{
        return (1 / 2) * base * height
}

myarea = area(100, 20)
```

12. Exercise - III

- Write a function `square()` to draw a square. The function should accept the length of the sides as argument.
- Write a function `row()` that uses the `square()` function to draw a row of squares. The function should accept the no. of squares in the row as argument. Make sure that the turtle returns back to same position and angle it started from.
- Write a function `grid()` that uses the `row()` function to draw 8 rows, to form a 8x8 grid. Do you see how functions can be **combined as building blocks** to do more complex things?
- Write a function `plus_grid()`, that uses the `row()` function to draw a plus shaped grid. Do you see how functions can be **re-used** to do completely different things?





13. Local Variables

- Variables assigned to within functions are accessible only within the function.
- These memory locations are allocated when the function is entered and are de-allocated when the function returns.

14. Conditional Execution

- It sometimes required to execute instructions only if a certain condition is satisfied.
- This can be acheived using the `if` statement.

- The general syntax of the `if` statement is given below.

```
if (<conditional-expression>) {
        ... statements ...
}
```

- The conditional expression evaluates to true or false.
- If the conditional expression evaluates to true the statements are executed.
- The conditional expression is formed by combining two arithmetic expressions using relational operators.
- Relational operators

`>`	Greater than
`<`	Less than
`==`	Equal to
`!=`	Not Equal to
`>=`	Greater than or equal to
`<=`	Less than or equal to

- Examples of a conditional expressions, assume a = 10.

`5 > 10`	false
`a > 5`	true
`a != 15`	true

- The `if` statement can also have an else part, which is executed when the condition is false. The general syntax is shown below.

```
if (<condition>) {
   ... statements executed if true ...
} else {
   ... statements executed if false ...
}
```

- The `if` statement can also be used to construct an `if ... else if` ladder. The general syntax is shown below.

```
if (<condition-1>) {
   ... statements-1 ...
} else if (<condition-2>) {
   ... statements-2 ...
} else if (<condition-3>) {
   ... statements-3 ...
} else {
   ... statements-4 ...
}
```

15. `for` Statement

- In many cases, while repeating a set of statements, a variable is updated on each iteration, and the value of the variable determines when the iteration stops.

```
step = 4
```

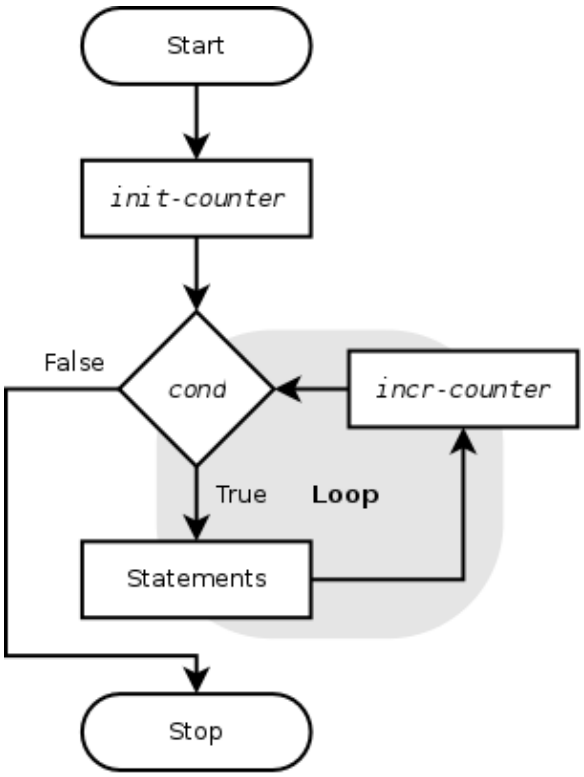
```
repeat(50) {
           fd(step)
           rt(90)
           step = step + 4
}
```

- In the spiral example, the `step` variable is updated every time in the loop. The variable that is updated on each iteration is called the counter variable.
- The loop can be considered to terminate when the value of `step` reaches 200.
- Such loops can be simplifed by using the `for` statement. The general syntax is given below.

```
for (<init-counter>;<cond>;<incr-counter>) {
       ... statements ...
}
```

- The flowchart showing the execution of the `for` statement is shown below.

Figure 1. `for` statement flowchart



- The spiral example, rewritten with `for` statement.

```
for (step = 4; step < 200; step = step + 4) {
    fd(step)
    rt(90)
}
```

16. Clock Dial

- Example of a clock dial.

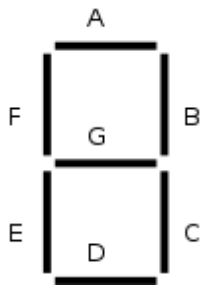
```
def draw_dash(i)
```

```
{
    if (i % 15 == 0)
        d = 20
    else
        d = 10
    fd(d)
    bk(d)
}

def dial()
{
    for (i = 0; i < 360; i += 6) {
        pu(); fd(110);
        pd(); draw_dash(i);
        pu(); bk(110);
        rt(6);
    }
}
```

17. Exercise IV

1. Write a function `draw_sseg()` to draw a seven segment digit. The function should accept seven arguments - one for each segment. Each argument specifies 1 if the segment is on, and 0 if the segment is off.



2. Write a function `draw_digit()` to draw a seven segment digit. The function should accept one argument - the digit. The function should interally invoke `draw_sseg()` with the right parameters.
3. Write a function `draw_num()` to draw multiple seven segment digits. The function should accept two arguments - the number to display, and the number of digits. The function should extract each digit and draw it using `draw_digit()`.
4. Rewrite the spiral using `for` statement.