Lecture 2
Control Structures

Dr. Mahesha Narayana

Intended Learning Outcomes

At the end of this lecture, students will be able to:

- Illustrate various looping structures and their purpose
- Write simple codes involving specific looping structures

Topics

- Relational operators
- Logical operators
- Conditional statements
- Flow chart

Relational operators

Table 3.1

Operator	**Meaning**
$<$	Less than.
$<=$	Less than or equal to.
$>$	Greater than.
$>=$	Greater than or equal to.
$==$	Equal to.
$\sim=$	Not equal to.

For example, suppose that `x = [6,3,9]` and `y = [14,2,9]`. The following MATLAB session shows some examples.

```
>>z = (x < y)
z =
    1    0    0
>>z = (x ~= y)
z =
    1    1    0
>>z = (x > 8)
z =
    0    0    1
```

The relational operators can be used for array addressing.

For example, with x = [6,3,9] and y = [14,2,9], typing

z = x(x<y)

finds all the elements in x that are less than the corresponding elements in y. The result is z = 6.

Logical operators

Table 3.2

Operator	**Name**	**Definition**
~	NOT	~A returns an array the same dimension as A; the new array has ones where A is zero and zeros where A is nonzero.
&	AND	A & B returns an array the same dimension as A and B; the new array has ones where both A and B have nonzero elements and zeros where either A or B is zero.
|	OR	A | B returns an array the same dimension as A and B; the new array has ones where at least one element in A or B is nonzero and zeros where A and B are both zero.

(continued …)

Table 3.2 (continued)

Operator	Name	Definition
&&	Short-Circuit AND	Operator for scalar logical expressions. A && B returns true if both A and B evaluate to true, and false if they do not.
||	Short-Circuit OR	Operator for scalar logical expressions. A || B returns true if either A or B or both evaluate to true, and false if they do not.

Order of Precedence for Operator Types

Table 3.3

Precedence	**Operator type**
First	Parentheses; evaluated starting with the innermost pair.
Second	Arithmetic operators and logical NOT (~); evaluated from left to right.
Third	Relational operators; evaluated from left to right.
Fourth	Logical AND.
Fifth	Logical OR.

The `if` Statement

The `if` statement's basic form is

`if` *logical expression*
    statements
`end`

Every `if` statement must have an accompanying end statement. The end statement marks the end of the *statements* that are to be executed if the *logical expression* is true.

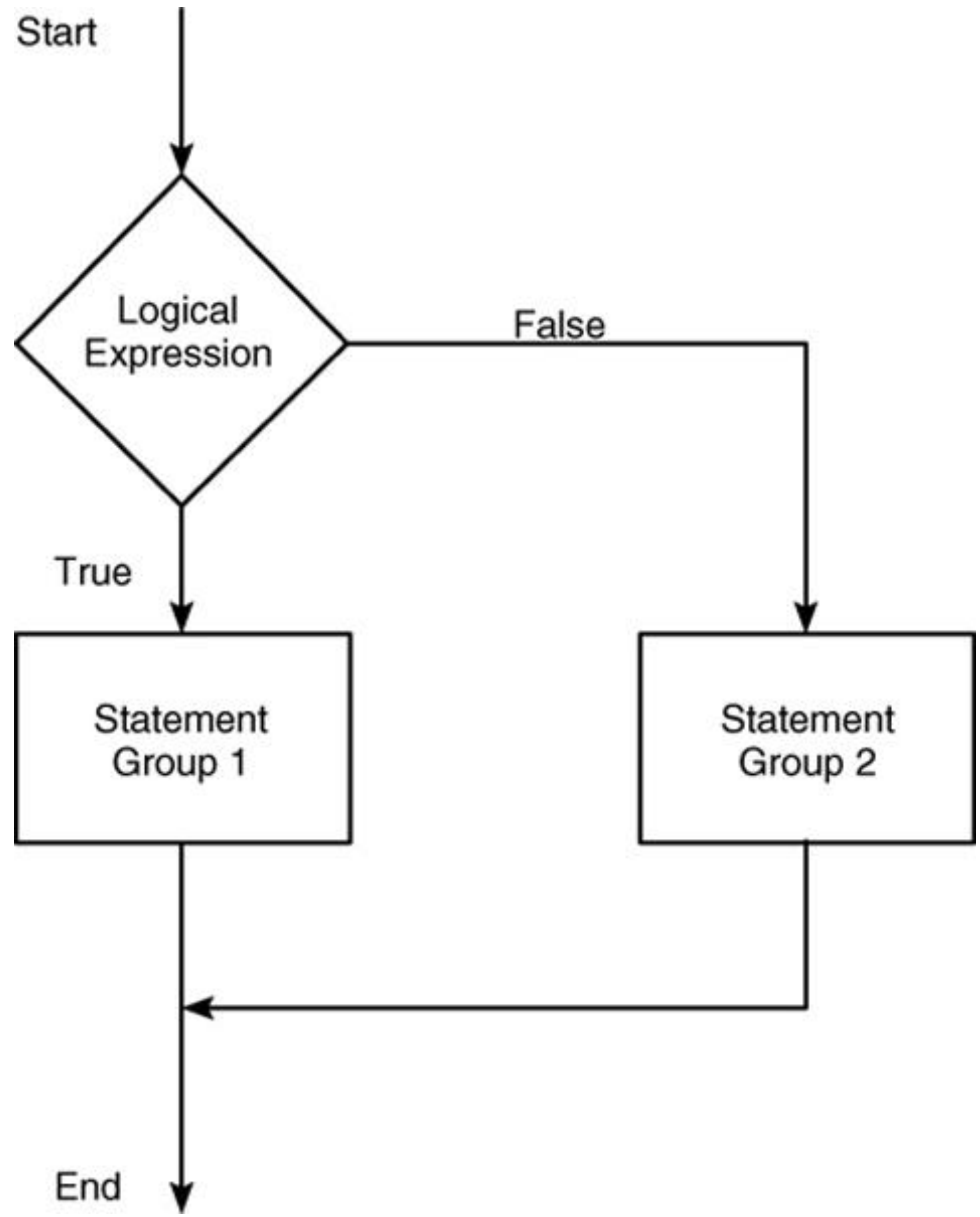
The `else` Statement

The basic structure for the use of the `else` statement is

`if` *logical expression*
  statement group 1
`else`
  statement group 2
`end`

Flowchart of the `else` structure

Figure 3.1



- Can have several elseif conditions…
- Else is optional and executes if all other tests fail

When the test, if *logical expression,* is performed, where the logical expression may be an *array,* the test returns a value of true only if *all* the elements of the logical expression are true!

For example, if we fail to recognize how the test works, the following statements do not perform the way we might expect.

```
x = [4,-9,25];
if x < 0
   disp('Some elements of x are negative.')
else
   y = sqrt(x)
end
```

Because the test `if x < 0` is false, when this program is run it gives the result

```
y =
  2    0 + 3.000i    5
```

Instead, consider what happens if we test for `x` positive.

```
x = [4,-9,25];
if x >= 0
  y = sqrt(x)
else
  disp('Some elements of x are negative.')
end
```

When executed, it produces the following message:

```
Some elements of x are negative.
```

The test `if x < 0` is false, and the test if `x >= 0` also returns a false value because `x >= 0` returns the vector `[1,0,1]`.

The following statements

```
if logical expression 1
   if logical expression 2
     statements
  end
end
```

can be replaced with the more concise program

```
if logical expression 1 & logical expression 2
  statements
end
```

The `elseif` Statement

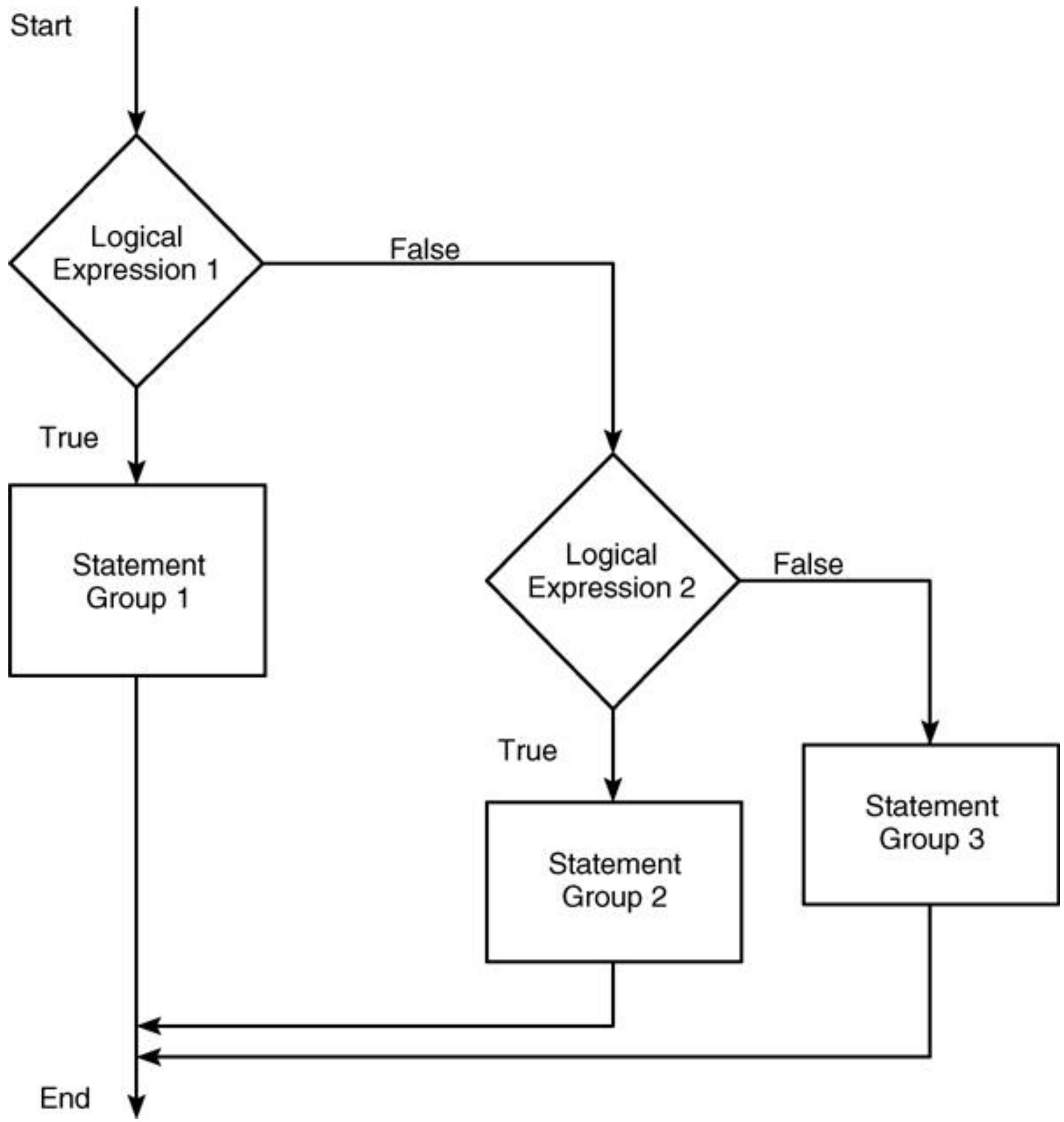
The general form of the `if` statement is

```
if logical expression 1
     statement group 1
elseif logical expression 2
     statement group 2
else
     statement group 3
end
```

The `else` and `elseif` statements may be omitted if not required. However, if both are used, the `else` statement must come after the `elseif` statement to take care of all conditions that might be unaccounted for.

Flowchart for the general `if-elseif-else` structure.

Figure 3.2

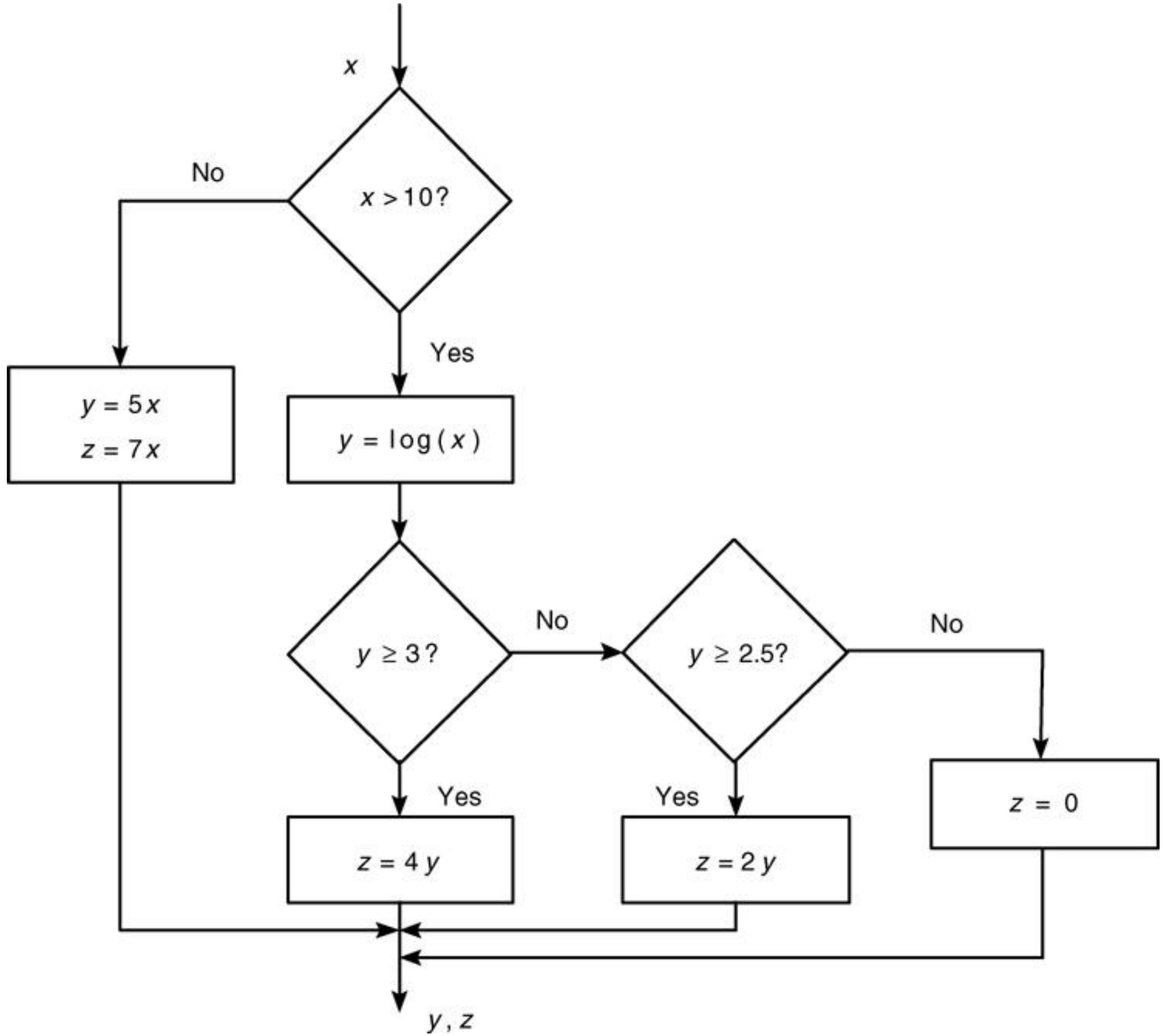


For example, suppose that `y = log(x)` for *x* `> 10`, *y* `= sqrt(`*x*`)` for `0 <= x <= 10`, and `y = exp(x) - 1` for `x < 0`. The following statements will compute `y` if `x` already has a scalar value.

```
if x > 10
  y = log(x)
elseif x >= 0
  y = sqrt(x)
else
  y = exp(x) - 1
end
```

Flowchart illustrating nested `if` statements

Figure 3.3



If statement example

```
%DEMO
function output = DEMO(input)

%put help info here!
if input > 0
    fprintf('Greater than 0')
elseif input < 0
    fprintf('Less than 0')
else
    fprintf('Equals Zero')
end

%Set return value if needed
outvar = 1;
```

Strings

A *string* is a variable that contains characters. Strings are useful for creating input prompts and messages and for storing and operating on data such as names and addresses.

To create a string variable, enclose the characters in single quotes. For example, the string variable `name` is created as follows:

```
>>name = 'Leslie Student'
name =
    Leslie Student
```

(continued …)

Strings (continued)

The following string, `number`, is *not* the same as the variable number created by typing `number = 123`.

```
>>number = ’123’
number =
    123
```

Strings and the input Statement

The prompt program on the next slide uses the `isempty(x)` function, which returns a 1 if the array `x` is empty and 0 otherwise.

It also uses the input function, whose syntax is

```
x = input('prompt', 'string')
```

This function displays the string `prompt` on the screen, waits for input from the keyboard, and returns the entered value in the string variable `x`.

The function returns an empty matrix if you press the **Enter** key without typing anything.

Strings and Conditional Statements

The following prompt program is a script file that allows the user to answer *Yes* by typing either `Y` or `y` or by pressing the **Enter** key. Any other response is treated as the answer *No*.

```
response = input('Want to continue? Y/N [Y]: ','s');
if (isempty(response))|(response=='Y')|(response=='y')
  response = 'Y'
else
  response = 'N'
end
```

for Loops

A simple example of a `for` loop is

```
for k = 5:10:35
  x = k^2
end
```

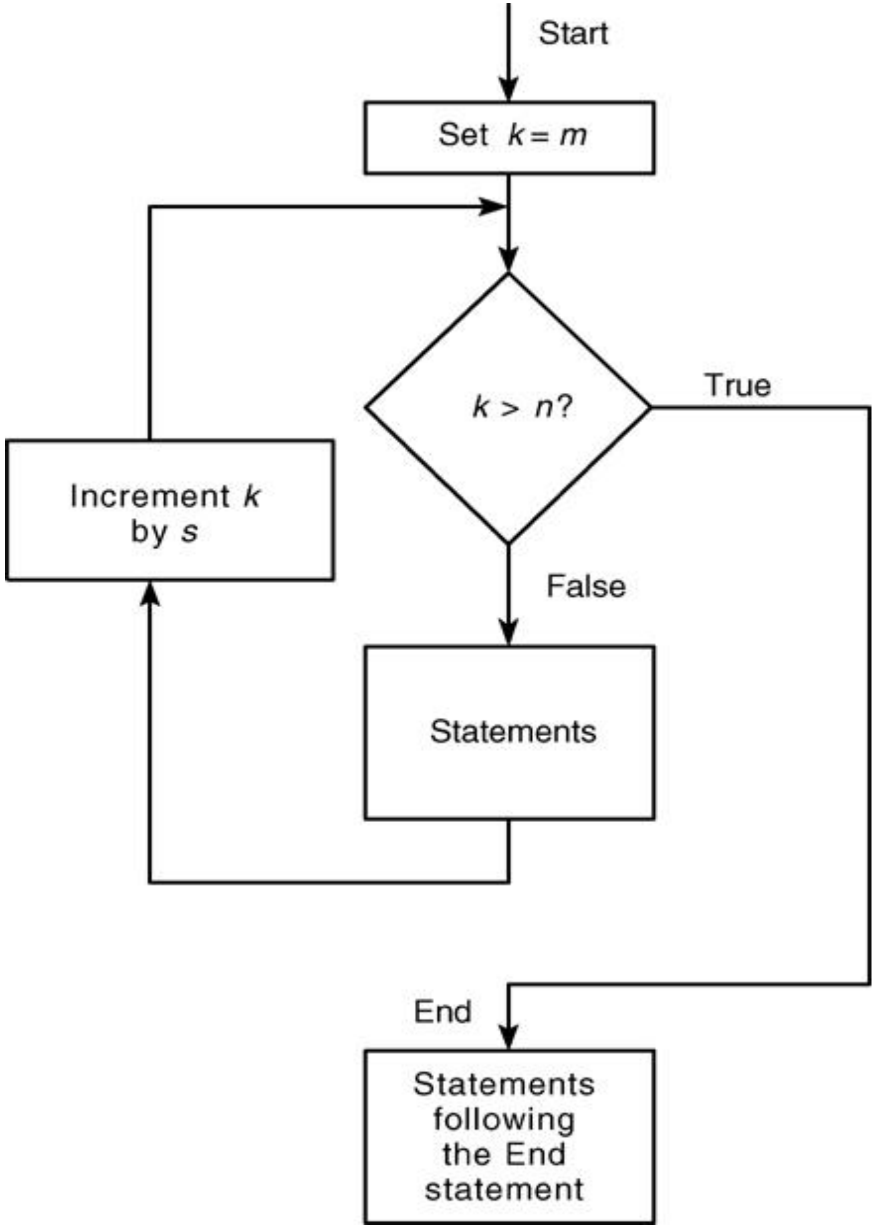
The *loop variable* `k` is initially assigned the value 5, and `x` is calculated from `x = k^2`. Each successive pass through the loop increments `k` by 10 and calculates x until k exceeds 35. Thus `k` takes on the values 5, 15, 25, and 35, and `x` takes on the values 25, 225, 625, and 1225. The program then continues to execute any statements following the end statement.

Flowchart of a for Loop

Figure 3.4

```
for j=m:s:n
   % computations;
end
```

- Repeats for specified number of times
- ALWAYS executes computation loop at least once!!!
- Can use + or – increments
- Can escape (BREAK) out of computational loop



Note the following rules when using for loops with the loop variable expression `k = m:s:n`:

- The step value `s` may be negative.
  Example: `k = 10:-2:4` produces k = 10, 8, 6, 4.
- If `s` is omitted, the step value defaults to 1.
- If `s` is positive, the loop will not be executed if `m` is greater than `n`.
- If `s` is negative, the loop will not be executed if `m` is less than `n`.
- If `m` equals `n`, the loop will be executed only once.
- If the step value `s` is not an integer, round-off errors can cause the loop to execute a different number of passes than intended.

The continue Statement

The following code uses a `continue` statement to avoid computing the logarithm of a negative number.

```
x = [10,1000,-10,100];
y = NaN*x;
for k = 1:length(x)
  if x(k) < 0
    continue
  end
  y(k) = log10(x(k));
end
y
```

The result is `y = 1, 3, NaN, 2`.

Example

Evaluate the following summation: $\text{Sum} = \sum_{i=1}^{10} i^3$

```
% filename: for1.m
% Example:  Use a for loop to find sum(i^3) for i = 1 to 10.
Sum = 0;            %Initialize variable to zero
for i = 1:1:10
    Sum = Sum + i^3;
end
fprintf('Sum = %0.0f\n',Sum);
```

```
% filename: for2.m
% Example:  Use a for loop to find sum(i^3) for i = 1 to 10.
Sum = 0;            %Initialize variable to zero
for i = [1,2,3,4,5,6,7,8,9,10]
    Sum = Sum + i^3;
end
fprintf('Sum = %0.0f\n',Sum);
```

Results:

```
>> for1
Sum = 3025
>> for2
Sum = 3025
>>
```

`while` Loop

The `while` loop is used when the looping process terminates because a specified condition is satisfied, and thus the number of passes is not known in advance. A simple example of a while loop is

```
x = 5;
while x < 25
   disp(x)
   x = 2*x - 1;
end
```

The results displayed by the `disp` statement are 5, 9, and 17.

The typical structure of a while loop follows.

```
while logical expression
    statements
end
```

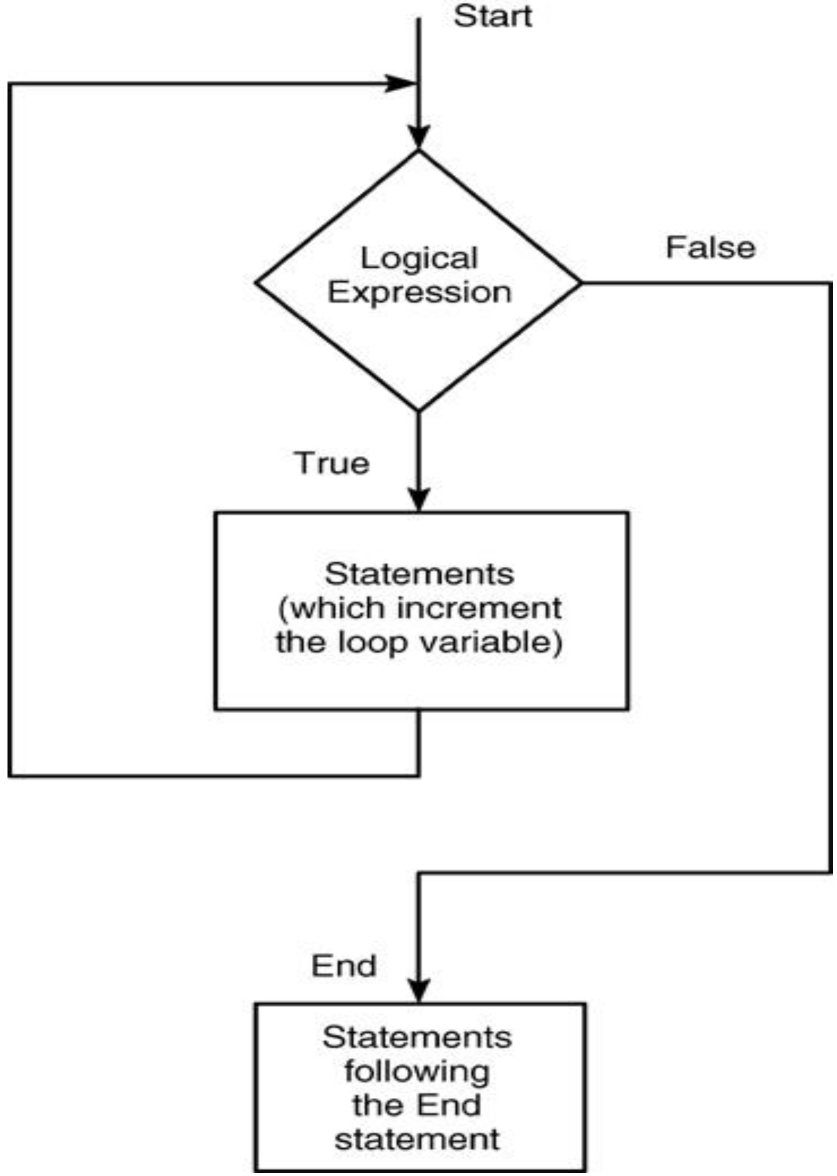
For the `while` loop to function properly, the following two conditions must occur:

1. The loop variable must have a value before the while statement is executed.

2. The loop variable must be changed somehow by the *statements.*

Flowchart of the while loop

Figure 3.5



- Will do computational loop ONLY if while condition is met
- Be careful to initialize while variable
- Can loop forever if while variable is not updated within loop!!!

A simple example of a `while` loop is

```
x = 5;k = 0;
while  x < 25
   k = k + 1;
   y(k) = 3*x;
   x = 2*x-1;
end
```

The loop variable `x` is initially assigned the value 5, and it keeps this value until the statement `x = 2*x - 1` is encountered the first time. Its value then changes to 9. Before each pass through the loop, `x` is checked to see if its value is less than 25. If so, the pass is made. If not, the loop is skipped.

Another Example of a `while` Loop

Write a script file to determine how many terms are required for the sum of the series $5k^2 - 2k$, $k = 1, 2, 3, \ldots$ to exceed 10,000. What is the sum for this many terms?

```
total = 0;k = 0;
while total < 1e+4
   k = k + 1;
   total = 5*k^2 - 2*k + total;
end
disp('The number of terms is:')
disp(k)
disp('The sum is:')
disp(total)
```

The sum is 10,203 after 18 terms.

TRY IT

What is the greatest value of n that can be used in the sum $1^2 + 2^2 + \quad + n^2$ and get a value of less than 100?

```
>> S = 1; n = 1;
>> while S+ (n+1)^2 < 100; n = n+1; S = S + n^2;
   end
>> [n, S]
ans = 6  91
```

The lines of code between while and end will only be executed if the condition S+ (n+1)^2 < 100 is true.

The `switch` Structure

The `switch` structure provides an alternative to using the `if`, `elseif`, and `else` commands. Anything programmed using `switch` can also be programmed using `if` structures.

However, for some applications the `switch` structure is more readable than code using the `if` structure.

Syntax of the switch structure

`switch` *`input expression`* (which can be a scalar or string).

```
switch input expression
   case value1
         statement group 1
   case value2
         statement group 2
   .
   .
   .
   otherwise
         statement group n
end
```

The following switch block displays the point on the compass that corresponds to that angle.

```
switch angle
 case 45
   disp('Northeast')
 case 135
   disp('Southeast')
 case 225
   disp('Southwest')
 case 315
   disp('Northwest')
 otherwise
   disp('Direction Unknown')
end
```

Disp() and fprintf()

- disp(X) – prints elements of an array X
- disp('hello world') – prints the string

- fprintf(fid, format, A) – does the following:
  - **Write A to file fid using format (omitting fid prints to screen)**
  - **format is a string containing output string and format instructions for each variable in A**
  - **Variables of all printable data types: Conversion specifications involve the character %, optional flags, optional width and precision fields, optional subtype specifier, and conversion characters: d, i, o, u, x, X, f, e, E, g, G, c, and s.**
  - **The special formats \n,\r,\t,\b,\f can be used to produce linefeed, carriage return, tab, backspace, and formfeed characters respectively.**

- Let's use DEMO to explore these differences.
- We will discuss I/O in further depth in a later lecture

Demonstration Problem 1

```
% This program will calculate the
% area and circumference of ten circles,
% allowing the radius as an input,
% but will only provide output for circles
% with an area that exceeds 20.
N = 0; R = 0.0; AREA = 0.0; CIRC = 0.0;
for J = 1:1:10
  R = input('Please enter the radius: ');
  AREA = pi * R^2;
  CIRC = 2.0 * pi * R;
  if AREA > 20.0
    fprintf('\n Radius = %f units',R)
    fprintf('\n Area = %f units squared', AREA)
    fprintf('\n Circumference = %f units\n', CIRC)
  else
    N = N + 1;
  end
end
  fprintf('\n Number of circles that do not have area > 20: %.0f \n', N)
```

Demonstration Problem 2

```
% Sample program to determine the number of years required for an initial
% balance to reach a final value
% Deposit = initial deposit
% Desired_Balance = final value to be reached
% I = interest rate (percent)
% N = number of years
% Filename:  Interest2.m
Deposit = input('Enter the amount of the initial deposit: $');
Desired_Balance = input('Enter the desired final balance: $');
I = input('Enter the percent interest rate: ');
N = 0;     %Initialize the number of years
Balance = Deposit;  %Initial value in the account
while Balance < Desired_Balance
    N = N+1;
    Balance = Balance*(1+I/100);
end
fprintf('\nResults:\nFinal balance = $%0.2f\n',Balance);
fprintf('Number of years to reach final balance = %0.0f\n',N);
```

Session Summary

- Control structures pass control in the program based on some conditions to be satisfied
- The syntax for simple if…. else…. statement is

```
if logical expression
        statement group 1
else
        statement group 2
End
```

- The syntax for a 'for loop' is

```
for j=1:m
  % computations;
End
```

- The syntax for a 'while loop' is

```
while logical expression
    statements
end
```