Programming in MATLAB

User-Defined Functions

The first line in a function file must begin with a *function definition line* that has a list of inputs and outputs. This line distinguishes a function M-file from a script M-file. Its syntax is as follows:

```
function [output variables] = name(inputvariables)
```

Note that the output variables are enclosed in *square brackets,* while the input variables must be enclosed with *parentheses.* The function name (here, `name`) should be the same as the file name in which it is saved (with the .m extension).

User-Defined Functions: Example

```
function z=fun(x,y)
u=3*x;
z=u+6*y.^2;
```

Note the use of a semicolon at the end of the lines. This prevents the values of `u` and `z` from being displayed. Note also the use of the array exponentiation operator (`.^`). This enables the function to accept `y` as an array.

Call this function with its output argument:

```
>>z=fun(3,7)
  z=
      303
```

The function uses x = 3 and y = 7 to compute z.

User-Defined Functions: Example (continued)

Call this function without its output argument and try to access its value. You will see an error message.

```
>>fun(3,7)
ans=
      303
>>z
???Undefined function or variable 'z'.
```

Assign the output argument to another variable:

```
>>q=fun(3,7)
q=
     303
```

You can suppress the output by putting a semicolon after the function call.

For example, if you type `q=fun(3,7);` the value of `q` will be computed but not displayed (because of the semicolon).

A function may have more than one output. These are enclosed in square brackets.

For example, the function `circle` computes the area *A* and circumference *C* of a circle, given its radius as an input argument.

```
function [A,C]=circle(r)
A=pi*r.^2;
C=2*pi*r;
```

The function is called as follows, if the radius is 4.

```
>>[A,C]=circle(4)
A=
    50.2655
C=
    25.1327
```

A function may have no input arguments and no output list.
For example, the function `show_date` computes and stores the date in the variable `today`, and displays the value of `today`.

```
function show_datetoday = date
```

Examples of Function Definition Lines

1. One input, one output:

```
function [area_square] = square(side)
```

2. Brackets are optional for one input, one output:

```
function area_square = square(side)
```

3. Three inputs, one output:

```
function [volume_box] = box(height,width,length)
```

4. One input, two outputs:

```
function [area_circle,circumf] = circle(radius)
```

5. No named output:

```
function sqplot(side)
```

Programming in MATLAB

❖A computer program is a sequence of commands.

❖ In a simple program the commands are executed one after the other in the order they are typed.

❖MATLAB provides several tools that can be used to control the flow of a program.

❖ Conditional statements , the switch structure make it possible to skip commands or to execute specific groups of commands in different situations.

❖For loops and while loops make it possible to repeat a sequence of commands several times.

❖changing the flow of a program requires some kind of decision-making process within the program.

❖The computer must decide whether to execute the next command or to skip one or more commands and continue at a different line in the program.

❖The program makes these decisions by comparing values of variables.

RELATIONAL AND LOGICAL OPERATORS

❖A relational operator compares two numbers by determining whether a comparison statement is true or false.

❖ If the statement is true, it is assigned a value of 1. If the statement is false, it is assigned a value of 0.

❖A logical operator examines true/false statements and produces a result that is true (1) or false (0)

❖ Relational and logical operators can be used in mathematical expressions a to make decisions that control the flow of a computer program.

Relational operators:

Relational operators in MATLAB are:

Relational operator	**Description**
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
= =	Equal to
~=	Not Equal to

• **Relational operators are used as arithmetic operators within a mathematical expression**.

The result can be used in other mathematical operations, in addressing arrays, and together with other MATLAB commands (e.g., if) to control the flow of a program.

• **When two numbers are compared, the result is**

1 (logical true) if the comparison, according to the relational operator, is true.

0 (logical false) if the comparison is false.

• **If two scalars are compared, the result is a scalar 1 or 0.**

• **If two arrays are compared**

(only arrays of the same size can be compared), the comparison is done *element-by-element, and the result is a logical array of the same size with 1s* and 0s according to the outcome of the comparison at each address.

•**If a scalar is compared with an array**,

• the scalar is compared with every element of the array, the result is a logical array with 1s and 0s according to the outcome of the comparison of each element.

```
>> 5>8
ans =
     0
>> a=5<10
a =
     1

>> y=(6<10)+(7>8)+(5*3==60/4)

y =
     2
```

Checks if 5 is larger than 8.

Since the comparison is false (5 is not larger than 8) the answer is 0.

Checks if 5 is smaller than 10, and assigns the answer to a.

Since the comparison is true (5 is smaller than 10) the number 1 is assigned to a.

Using relational operators in math expression.

Equal to 1 since 6 is smaller than 10.

Equal to 0 since 7 is not larger than 8.

Equal to 1 since 5*3 is equal to 60/4.

```
>> b=[15 6 9 4 11 7 14]; c=[8 20 9 2 19 7 10];
```
Define vectors b and c.

```
>> d=c>=b
```
Checks which c elements are larger than or equal to b elements.

```
d =
     0     1     1     0     1     1     0
```
Assigns 1 where an element of c is larger than or equal to an element of b.

```
>> b == c
```
Checks which b elements are equal to c elements.

```
ans =
     0     0     1     0     0     1     0
```

```
>> b~=c
```
Checks which b elements are not equal to c elements.

```
ans =
     1     1     0     1     1     0     1
```

```
>> f=b-c>0

f =
     1     0     0     1     0     0     1

>> A=[2 9 4; -3 5 2; 6 7 -1]

A =
     2     9     4
    -3     5     2
     6     7    -1

>> B=A<=2

B =
     1     0     0
     1     0     1
     0     0     1
```

Subtracts c from b and then checks which elements are larger than zero.

Define a 3×3 matrix A.

Checks which elements in A are smaller than or equal to 2. Assigns the results to matrix B.

❖The results of a relational operation with vectors, are vectors with 0s &1s, are called logical vectors and can be used for addressing vectors.

❖ When a logical vector is used for addressing another vector, it extracts from that vector the elements in the positions where the logical vector has 1s.

```
>> r = [8 12 9 4 23 19 10]

r =
     8    12     9     4    23    19    10

>> s=r<=10

s =
     1     0     1     1     0     0     1

>> t=r(s)

t =
     8     9     4    10

>> w=r(r<=10)

w =
     8     9     4    10
```

- `>> r = [8 12 9 4 23 19 10]`: Define a vector r.
- `>> s=r<=10`: Checks which r elements are smaller than or equal to 10.
- `s = 1 0 1 1 0 0 1`: A logical vector s with 1s at positions where elements of r are smaller than or equal to 10.
- `>> t=r(s)`: Use s for addresses in vector r to create vector t.
- `t = 8 9 4 10`: Vector t consists of elements of r in positions where s has 1s.
- `>> w=r(r<=10)`: The same procedure can be done in one step.

❖**Order of precedence:** In a mathematical expression that includes relational and arithmetic operations, the arithmetic operations (+, –, *, /, \) have precedence over relational operations.

❖The relational operators themselves have equal precedence and are evaluated from left to right.

```
>> 3+4<16/2

ans =
     1

>> 3+(4<16)/2

ans =
    3.5000
```

+ and / are executed first.

The answer is 1 since 7 < 8 is true.

4 < 16 is executed first, and is equal to 1, since it is true.

3.5 is obtained from 3 + 1/2.

Logical operators

Logical operators in MATLAB are:

Logical operator	Name	Description
& Example: A&B	AND	Operates on two operands (A and B). If both are true, the result is true (1); otherwise the result is false (0).
| Example: A|B	OR	Operates on two operands (A and B). If either one, or both, are true, the result is true (1); otherwise (both are false) the result is false (0).
~ Example: ~A	NOT	Operates on one operand (A). Gives the opposite of the operand; true (1) if the operand is false, and false (0) if the operand is true.

- **Logical operators have numbers as operands.**
  A nonzero number is true, and a zero number is false.

- **Logical operators are used as arithmetic operators**
within a mathematical expression. The result can be used in other mathematical operations, in addressing arrays, and together with other MATLAB commands (e.g., if) to control the flow of a program.

- **Logical operators can be used with scalars and arrays.**

- **The logical operations AND and OR can have both operands as scalars, arrays, or one array and one scalar.**

- **If both are scalars, the result is a scalar 0 or 1.**

- **If both are arrays,**
  - they must be of the same size and
  - the logical operation is done *element-by-element. The result is an array of the same size with 1s and 0s* according to the outcome of the operation at each position.

- **If one operand is a scalar and the other is an array,**
  - the logical operation is done between the scalar and each of the elements in the array and the outcome is an array of the same size with 1s and 0s.

- **The logical operation NOT** has one operand.
  - ❖When it is used with a scalar the outcome is a scalar 0 or 1.
  - ❖ When it is used with an array, the outcome is an array of the same size with 1s in positions where the array has nonzero numbers and 0s in positions where the array has 0s.

```
>> 3&7
ans =
     1
>> a=5|0
a =
     1
```

- 3 AND 7.
- 3 and 7 are both true (nonzero), so the outcome is 1.
- 5 OR 0 (assign to variable a).
- 1 is assigned to a since at least one number is true (nonzero).

```
>> ~25
ans =
     0
>> t=25*((12&0)+(~0)+(0|5))
t =
    50
>> x=[9 3 0 11 0 15]; y=[2 0 13 -11 0 4];
>> x&y
ans =
     1     0     0     1     0     1
>> z=x|y
z =
     1     1     1     1     0     1
```

- NOT 25.
- The outcome is 0 since 25 is true (nonzero) and the opposite is false.
- Using logical operators in a math expression.
- Define two vectors x and y.
- The outcome is a vector with 1 in every position where both x and y are true (nonzero elements), and 0s otherwise.
- The outcome is a vector with 1 in every position where either or both x and y are true (nonzero elements), and 0s otherwise.

```
>> ~(x+y)
ans =
     0     0     0     1     1     0
```

The outcome is a vector with 0 in every position where the vector x + y is true (nonzero elements), and 1 in every position where x + y is false (zero elements).

Order of precedence:

Arithmetic, relational, and logical operators can be combined in math expressions. When an expression has such a combination, the result depends on the order in which the operations are carried out.

The following is the order used by MATLAB:

Precedence	**Operation**
1 (highest)	Parentheses (if nested parentheses exist, inner ones have precedence)
2	Exponentiation
3	Logical NOT (~)
4	Multiplication, division
5	Addition, subtraction
6	Relational operators (>, <, >=, <=, = =, ~=)
7	Logical AND (&)
8 (lowest)	Logical OR (|)

CONDITIONAL STATEMENTS

A conditional statement is a command that allows MATLAB to make a decision of whether **to execute a group of commands that follow the conditional statement**, or ***to skip these commands***.

In a conditional statement expression .

If the expression is true, a group of commands that follow the statement are executed.

If the expression is false, the computer skips the group.

The basic form of a conditional statement is:

Examples:

```
if a < b
if c >= 5
if a == b
if a ~= 0
if (d<h)&(x>7)
if (x~=13)|(y<0)
```

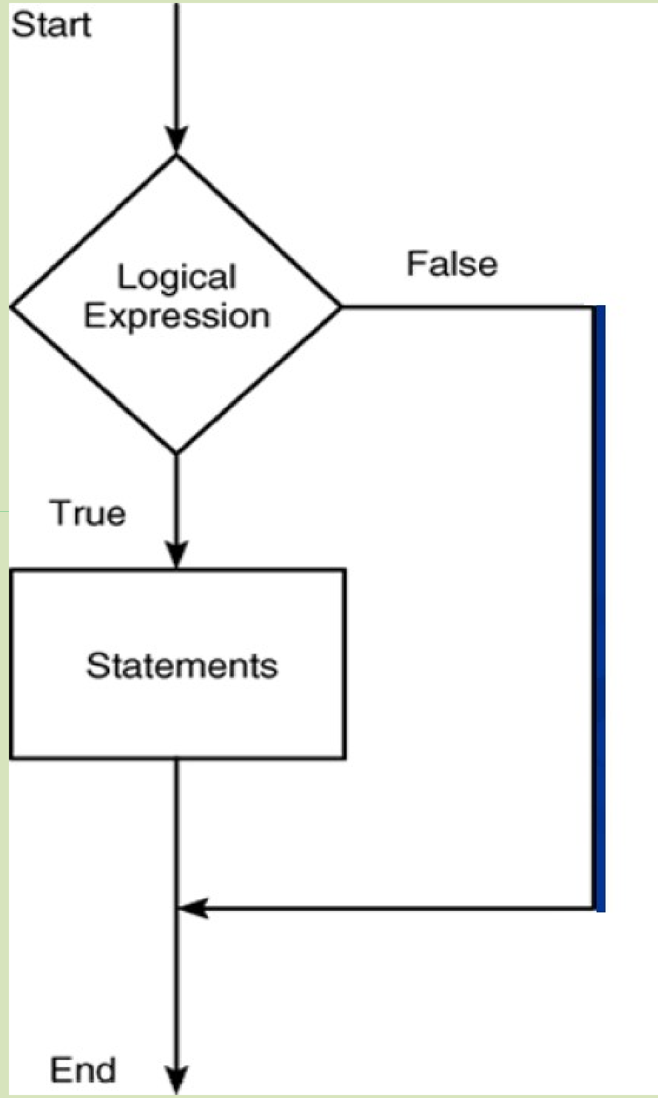
The `if` Statement

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.

Flowchart representation of the `if` statement.

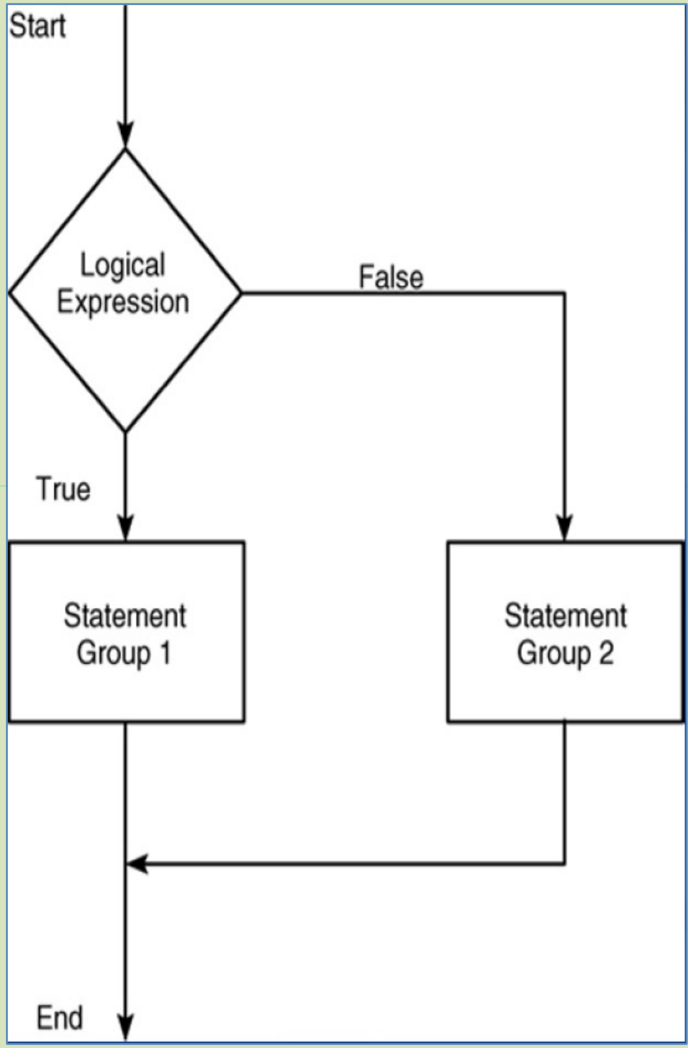


The `else` Statement

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.



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 expression2
   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.

Understanding **if/elseif /else** statement:

```
If        you study very well
       you get grade A
else if   you study well
       you get grade B
else if   you study little
        you get grade C
else if   you study a little bit
        you get grade D
else      you get grade F
end
```

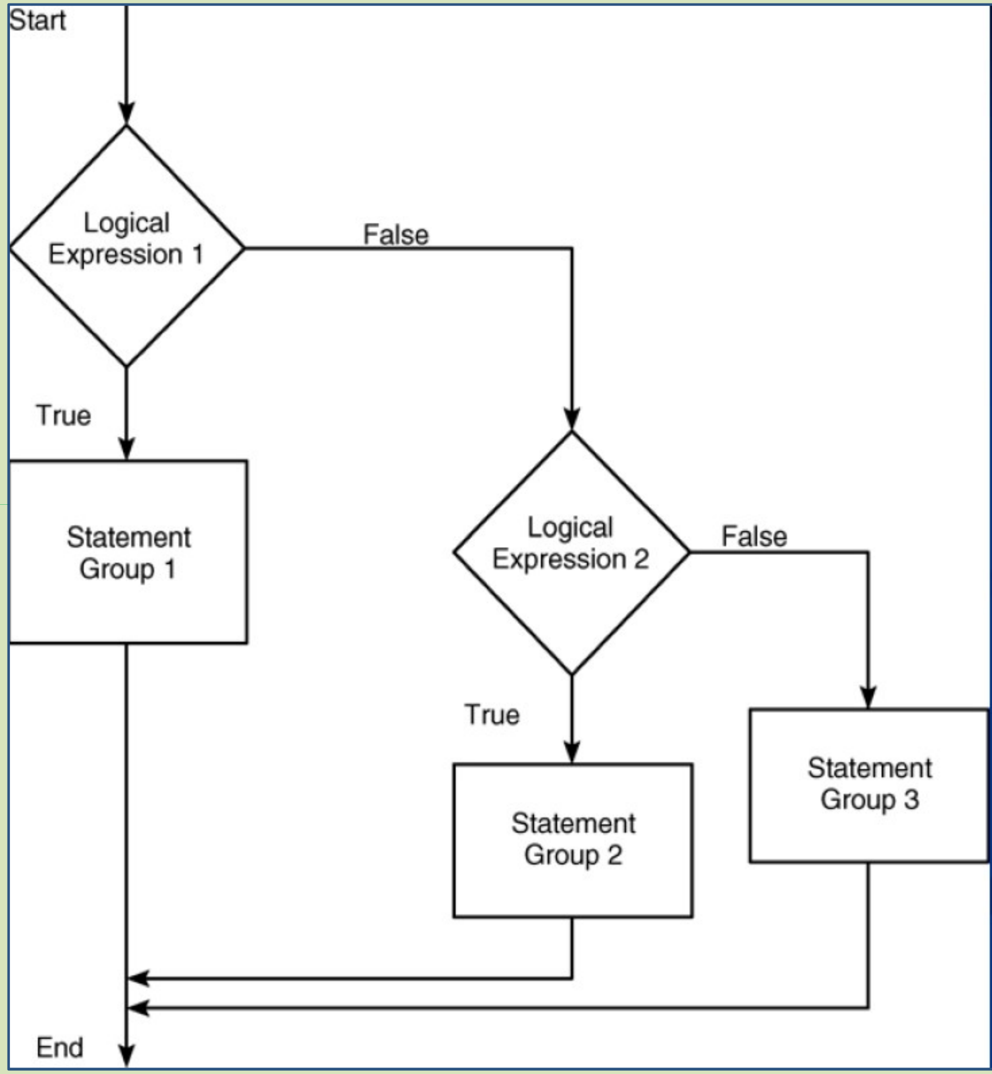
Syntax

```
if       expression
   statements
elseif expression
   statements
else
   statements
end
```

Example

```
if I == J
  A(I,J) = 2;
elseif abs(I-J) == 1
  A(I,J) = -1;
else
  A(I,J) = 0;
end
```

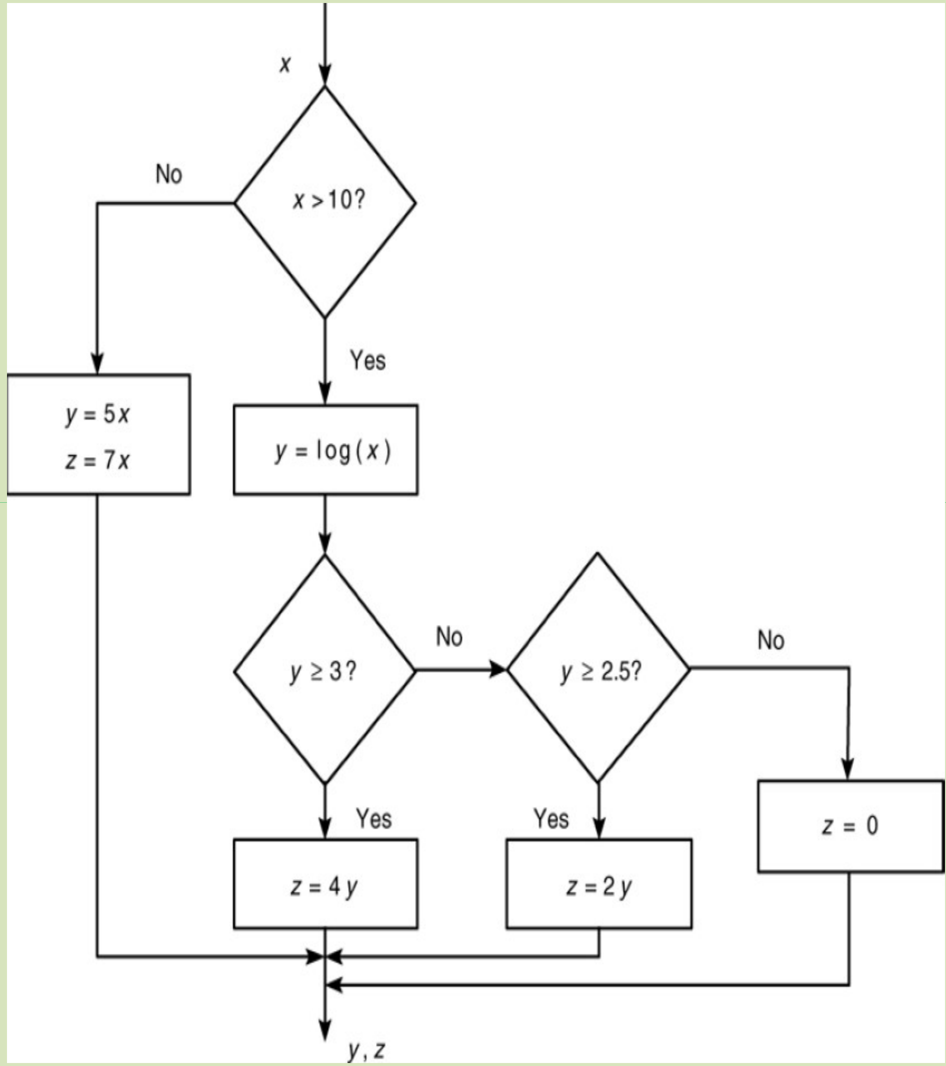
Flowchart for the general `if-elseif-else` structure.



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.



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
```

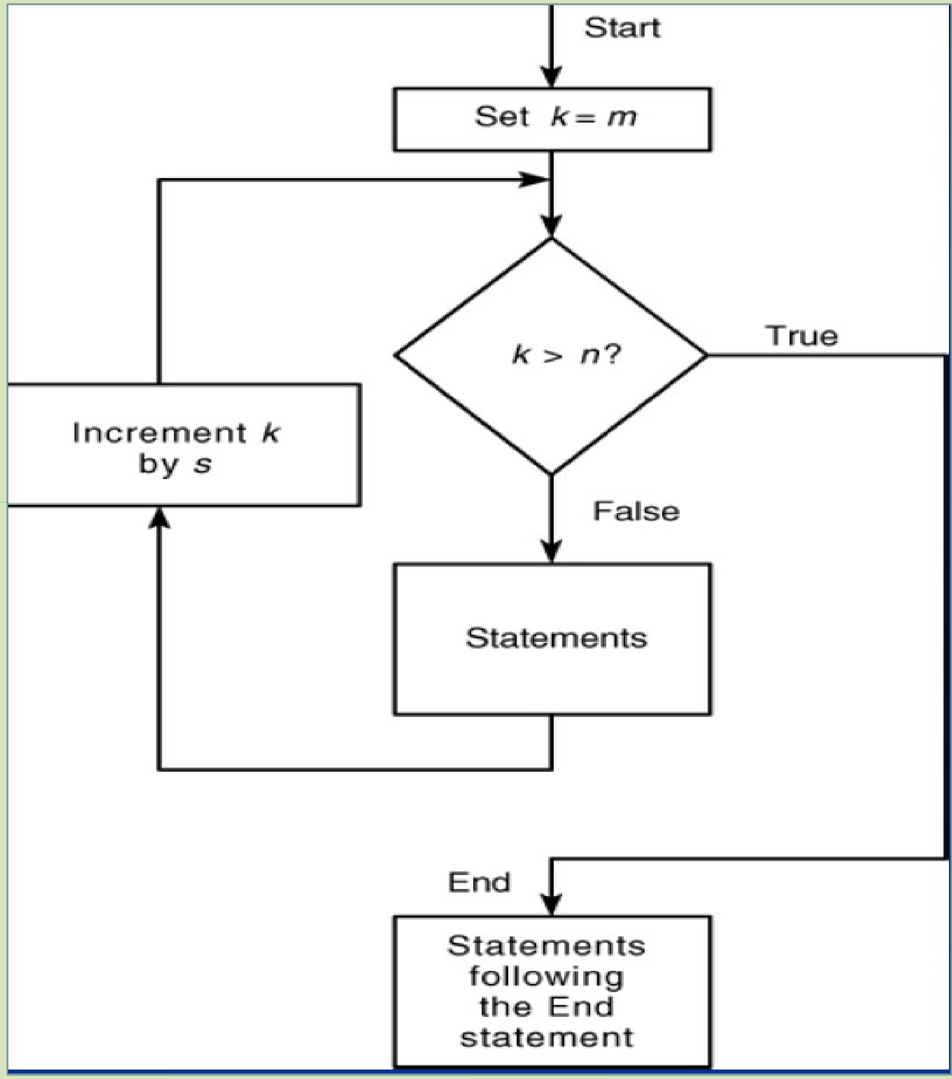
for Loops

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.



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.`

Use of a *Mask*

We can often avoid the use of loops and branching and thus create simpler and faster programs by using a logical array as a *mask* that selects elements of another array. Any elements not selected will remain unchanged.

The following session creates the logical array `C` from the numeric array `A` given previously.

```
>>A=[0,-1,4;9,-14,25;-34,49,64];
>>C=(A>=0);
```

The result is

$$C = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

We can use this mask technique to compute the square root of only those elements of A given in the previous program that are no less than 0 and add 50 to those elements that are negative. The program is

```
A=[0,-1,4;9,-14,25;-34,49,64];
C=(A>=0);
A(C)=sqrt(A(C))
A(~C)=A(~C)+50
```

Use of Logical Arrays as Masks

```
A = [0,-1,4;9,-14,25;-34,49,64];
For m= 1:size(A,1)
        for n=1:size(A,2)
                if  A ( m , n ) >=0
                  B ( m , n ) = sqrt ( a ( m , n ) ) ;
                else
                  B (m , n) = A ( m , n)  + 50 ;
                end
        end
end
-----------------------------------------------------
>>  B
        0   49   2
        3   36   5
        16   7   8
```

While Loops

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.

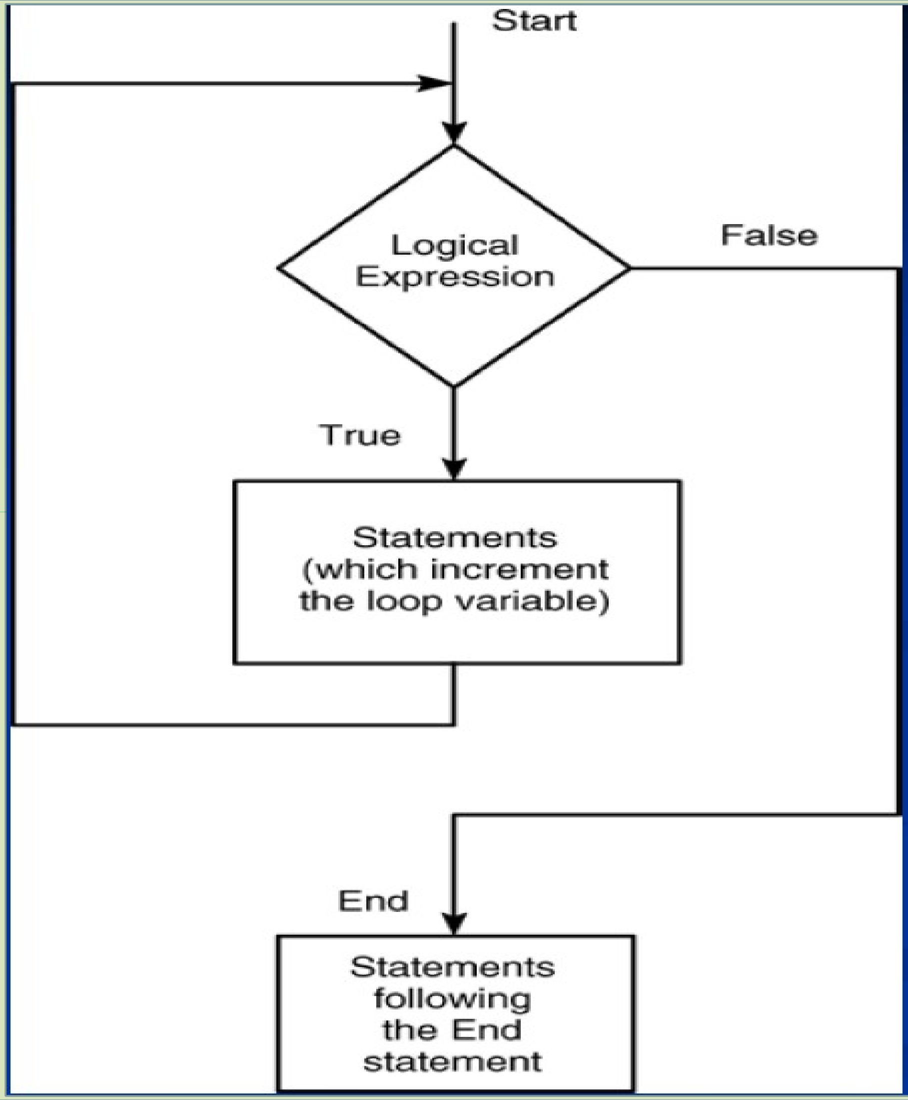
```
while logical expression
    Statements
end
```

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.



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.

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).

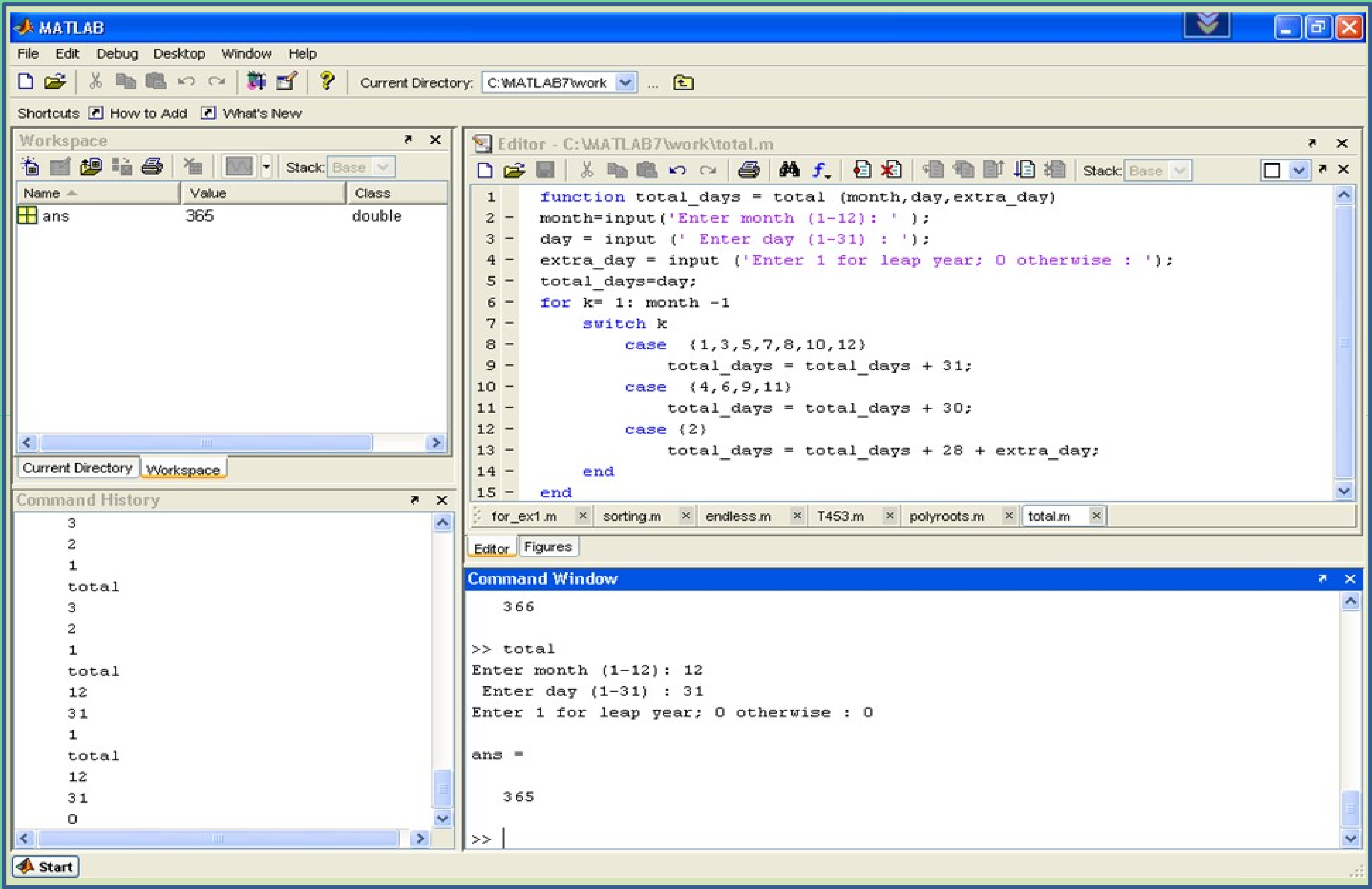
```
case value1
    statement group 1
case value2
    statement group 2
.
.
.
otherwise
    statement group n
end
```

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
```

Example

```
function total_days = total (month,day,extra_day)
month=input( 'Enter month (1-12): ' );
day = input (' Enter day (1-31) : ');
extra_day = input ('Enter 1 for leap year; 0 otherwise : ');
total_days = day;
for k= 1: month -1
   switch k
      case  {1,3,5,7,8,10,12}
         total_days = total_days + 31;
      case  {4,6,9,11}
         total_days = total_days + 30;
      case {2}
         total_days = total_days + 28 + extra_day;
   end
End
```


MATLAB
File Edit Debug Desktop Window Help
Current Directory: C:\MATLAB7\work
Shortcuts How to Add What's New
Workspace
Name Value Class
ans 365 double
Current Directory Workspace
Command History
3
2
1
total
3
2
1
total
12
31
1
total
12
31
0
Editor - C:\MATLAB7\work\total.m
function total_days = total (month,day,extra_day)
month=input('Enter month (1-12): ');
day = input (' Enter day (1-31) : ');
extra_day = input ('Enter 1 for leap year; 0 otherwise : ');
total_days=day;
for k= 1: month -1
switch k
case {1,3,5,7,8,10,12}
total_days = total_days + 31;
case {4,6,9,11}
total_days = total_days + 30;
case {2}
total_days = total_days + 28 + extra_day;
end
end
for_ex1.m sorting.m endless.m T453.m polyroots.m total.m
Editor Figures
Command Window
366
>> total
Enter month (1-12): 12
Enter day (1-31) : 31
Enter 1 for leap year; 0 otherwise : 0
ans =
365
>>
Start