Conditions in if statements use expressions that are conceptually either true or false. These expressions are called **relational expressions** or **Boolean expressions**.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equals</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equals</td>
</tr>
<tr>
<td>==</td>
<td>equality</td>
</tr>
<tr>
<td>∼=</td>
<td>inequality</td>
</tr>
</tbody>
</table>
Logical Operators

- The result of a logical operation is 1 if it is true and 0 if it is false.
- Logical operators can be used to create compound statements that evaluate to true or false

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>complements elements of A</td>
</tr>
<tr>
<td>xor</td>
<td>exclusive OR</td>
</tr>
<tr>
<td>all</td>
<td>TRUE if all elements of an array are TRUE</td>
</tr>
<tr>
<td>any</td>
<td>TRUE if any elements of an array are TRUE</td>
</tr>
</tbody>
</table>
Logical Operators

- The result of a logical operation is 1 if it is true and 0 if it is false.
- Logical operators can be used to create compound statements that evaluate to true or false

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>∼</td>
<td>complements elements of A</td>
</tr>
<tr>
<td>xor</td>
<td>exclusive OR</td>
</tr>
<tr>
<td>all</td>
<td>TRUE if all elements of an array are TRUE</td>
</tr>
<tr>
<td>any</td>
<td>TRUE if any elements of an array are TRUE</td>
</tr>
</tbody>
</table>

The following short circuit operators only work with scalars

- `&&`: `(exprA && exprB)` - `exprB` is only evaluated if `exprA` is true.
- `||`: `(exprA || exprB)` - `exprB` is not evaluated if `exprA` is true
## Truth tables

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A &amp; B</th>
<th>A</th>
<th>B</th>
<th>¬A</th>
<th>xor(A,B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
## Precidence rules

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>NOT</td>
<td>Highest</td>
</tr>
<tr>
<td>&amp;</td>
<td>AND</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>OR</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>short circuit AND</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- It is a good idea to use paranthesis on long expressions
Precidence rules

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>NOT</td>
<td>Highest</td>
</tr>
<tr>
<td>&amp;</td>
<td>AND</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>OR</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>short circuit AND</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- It is a good idea to use paranthesis on long expressions
- a|b&c is evaluated as a|(b&c)
Precidence rules (always use parenthesis)

1    >> b=10;
2    >> 1|b>0 & 0
3    ans =
4    logical
5    1
Precidence rules (always use parenthesis)

1  >> b=10;
2  >> 1|b>0 & 0
3   ans =
4   logical
5   1

1  >> (1|b>0) & 0
2   ans =
3   logical
4     0
Precidence rules (always use parenthesis)

```matlab
1 >> b=10;
2 >> 1|b>0 & 0
3 ans =
4 logical
5 1

1 >> (1|b>0) & 0
2 ans =
3 logical
4 0

1 >> 1|(b>0 & 0)
2 ans =
3 logical
4 1
```
Logical data types

- Logical data types can be used as indices to extract specific elements of vectors.

```
1  >> x=1:10
2  x =
3    1  2  3  4  5  6  7  8  9  10
4  >> y=[3  5  6  1  8  2  9  4  0  7]
5  y =
6    3  5  6  1  8  2  9  4  0  7
7  >> y(x<4)
8  ans =
9    3  5  6
```
However, not 0–1 vectors are **logical data types**

```plaintext
1 >> rem(y,2)
2 ans =
3  1 1 0 1 0 0 1 0 0 1
```
However, not 0–1 vectors are **logical data types**

```matlab
1 >> rem(y,2)
2 ans =
3 1 1 0 1 0 0 1 0 0 1
```

If we try to extract the odd entries as:

```matlab
1 >> y(rem(y,2))
2 Array indices must be positive integers or logical ... values.
```

This is due to the fact that `rem` does not return a logical data type
Logical data types

- We can use the logical function to fix this problem

```matlab
>> y=[3 5 6 1 8 2 9 4 0 7]
y =
3 5 6 1 8 2 9 4 0 7
>> y(logical(rem(y,2)))
ans =
3 5 1 9 7
```
find command

- We can extract elements from a vector satisfying a certain condition.

```matlab
x = [1 1 1 4 5 2 1]
>> ans = find(x==1)
ans = 1 2 3 7
>> x(find(x==1))
ans = 1 1 1 1
```

find also works for matrices, check the documentation for usage.
find command

• We can extract elements from a vector satisfying a certain condition.

```
1  >> x=[1 1 1 4 5 2 1]
2  x =
3       1  1  1  4  5  2  1
4  >> find(x==1)
5  ans =
6       1  2  3  7
7  >> x(find(x==1))
8  ans =
9       1  1  1  1
```

• find also works for matrices, check the documentation for usage
**find command**

- We can extract elements from a vector satisfying a certain condition.

```
1 >> x=[1 1 1 4 5 2 1]
2 x =
3     1 1 1 4 5 2 1
4 >> find(x==1)
5 ans =
6     1 2 3 7
7 >> x(find(x==1))
8 ans =
9    1 1 1 1
```

- *find* also works for matrices, check the documentation for usage
- Other useful commands: *any* and *all*
Selection control – if statements

If-blocks are used to decide which instruction to execute next depending on whether an expression is true or not.

- if ... end
  if logical_expression
  statement1
  statement2
  end
Selection control – *if* statements

*if*-blocks are used to decide which instruction to execute next depending on whether an *expression* is true or not.

- **if ...end**
  
  ```
  if logical_expression
    statement1
    statement2
  end
  ```

- **if ...else ...end**
  
  ```
  if logical_expression
    statements evaluated if TRUE
  else
    statements evaluated if FALSE
  end
  ```
Selection control – if statements

if-blocks are used to decide which instruction to execute next depending on whether an expression is true or not.

- if ...elseif ...else ...end
  if logical_expression1
    block of statements evaluated
    if logical_expression1 is TRUE
  elseif logical_expression2
    block of statements evaluated
    if logical_expression2 is TRUE
  else
    block of statements evaluated
    if no other expression is TRUE
  end
Selection control – *if* statements - Examples

% selection statements
% if ... end
a=input('Enter an integer:');
if(mod(a,2)==0)
   fprintf('Your integer %d is even\n',a);
end

% if...else... end
% we can add more feedback
if(mod(a,2)==0)
   fprintf('Your integer %d is even\n',a);
else
   fprintf('Your integer %d is odd \n',a);
end
Selection control – if statements - Examples

```matlab
1 %a code segment that categories height
2 height = input('Enter your feet:');
3 if (height > 7)
4    disp ('very tall');
5 elseif (height > 6)
6    disp ('tall');
7 elseif (height < 5)
8    disp ('short');
9 else
10    disp ('average');
11 end
```
Switches between several cases depending on an expression, which is either a scalar or a string.

```matlab
a=input('Enter an integer:');
switch(mod(a,2))
    case 0
        fprintf('Your integer %d is even\n',a);
    case 1
        fprintf('Your integer %d is odd \n',a);
    otherwise
        fprintf('The number %f is not an integer\n',a);
end
```
Selection control – Switch/Case statements

*Switches* between several cases depending on an expression, which is either a scalar or a string.

```matlab
1   a=input('Enter an integer:');
2   switch(mod(a,2))
3       case 0
4           fprintf('Your integer %d is even\n',a);
5       case 1
6           fprintf('Your integer %d is odd \n',a);
7       otherwise
8           fprintf('The number %f is not an integer\n',a);
9       end
```

Handy for avoiding tedious *if*..*elseif*.. statements
Iteration control – for Loops

- *loops* are used to repeat a block of statements until some condition is satisfied.

- Usage:

  ```
  for index = first:step:last
    block of statements
  end
  ```
Iteration control – *for* Loops (Example)

Use a *for* loop to plot $\cos(nx)$ using subplots for $n = 1 - 9$ on $[0, 2\pi]$
Use a `for` loop to plot $\cos(nx)$ using subplots for $n = 1 - 9$ on $[0, 2\pi]$.