PROCEDURAL PROGRAMMING: SELECTION/BRANCHING

ASSOC. PROF. TUNÇ DURMAZ
tdurmaz@yildiz.edu.tr
MAY 31, 2021

SELECTION/BRANCHING

- So far, every piece of MATLAB code that we have seen has consisted of a sequence of commands.
- This flow of control is handled behind the scenes by the MATLAB interpreter.
- Matlab interpreter is a program running in the background that
  - reads the statements that you write and carries them out one by one,
    - allocating space for variables,
    - writing values into those variables, and
    - reading values from them,
    - accessing elements of arrays,
    - calling functions, and
    - displaying results on the screen.

SELECTION/BRANCHING

- Executing the statements in the order that the programmer wrote them is called sequential control.
- Sequential control is the most natural and the most common sequence in any program written in any programming language.
- It is the primary example of a control construct.
- A control construct is simply a method by which the interpreter selects the following statement to be executed after the execution of the current statement has concluded.

SELECTION/BRANCHING

- Utilizing (or not) particular keywords, the programmer tells the interpreter which construct to use
- So far, such keywords have been absent from our code, and as a result, we have been utilizing only one type of control construct—sequential control.
- In this part of the lecture, we will introduce keywords that signal the interpreter to base its decision as to which statement is to be executed
  - not only on the order in which the statements are written
  - but also on the values of expressions.
- This new control construct is called selection or branching.

SELECTION - if statements

- An if-statement is used when the programmer wishes to have the interpreter choose whether or not to execute a statement or set of statements on the basis of the values of variables

```
function guess_my_number(x)
if x == 2
fprintf('Congrats! You guessed my number.\n');
end
```

SELECTION - if statements

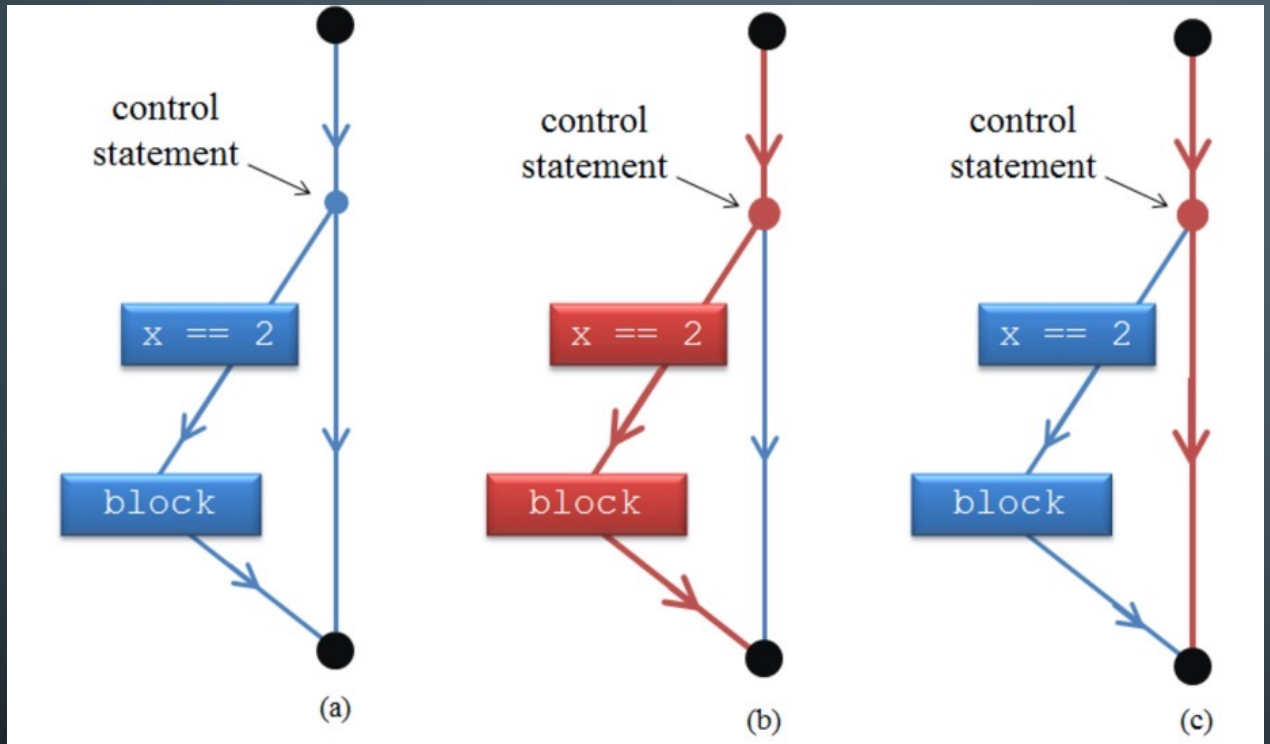


Figure 2.23 *Schematic of if-statement. The flow of control of an if-statement is shown. The black dots represent code before and after the if-statement. Control flows along the lines in the direction of the arrows. The control statement (labeled) is* `if x == 2`*. There are two possible paths as shown in (a) : one labeled* `x == 2`*, in which a block of statements, labeled "block", is executed, and one for every other possibility (the unlabeled vertical line) in which no statements are executed. If* `x` *equals 2, then the left branch is followed and the block of statements is executed, as shown by the red path in (b). Otherwise, the vertical path is followed, as shown in red in (c), and no statements are executed.*

A block is a set of adjacent statements, and each block in the figure is controlled by a control statement. In the example code that we showed above, there is just one statement (the **fprintf** function call statement), but in general there can be any number of statements in a block.

SELECTION - if statements

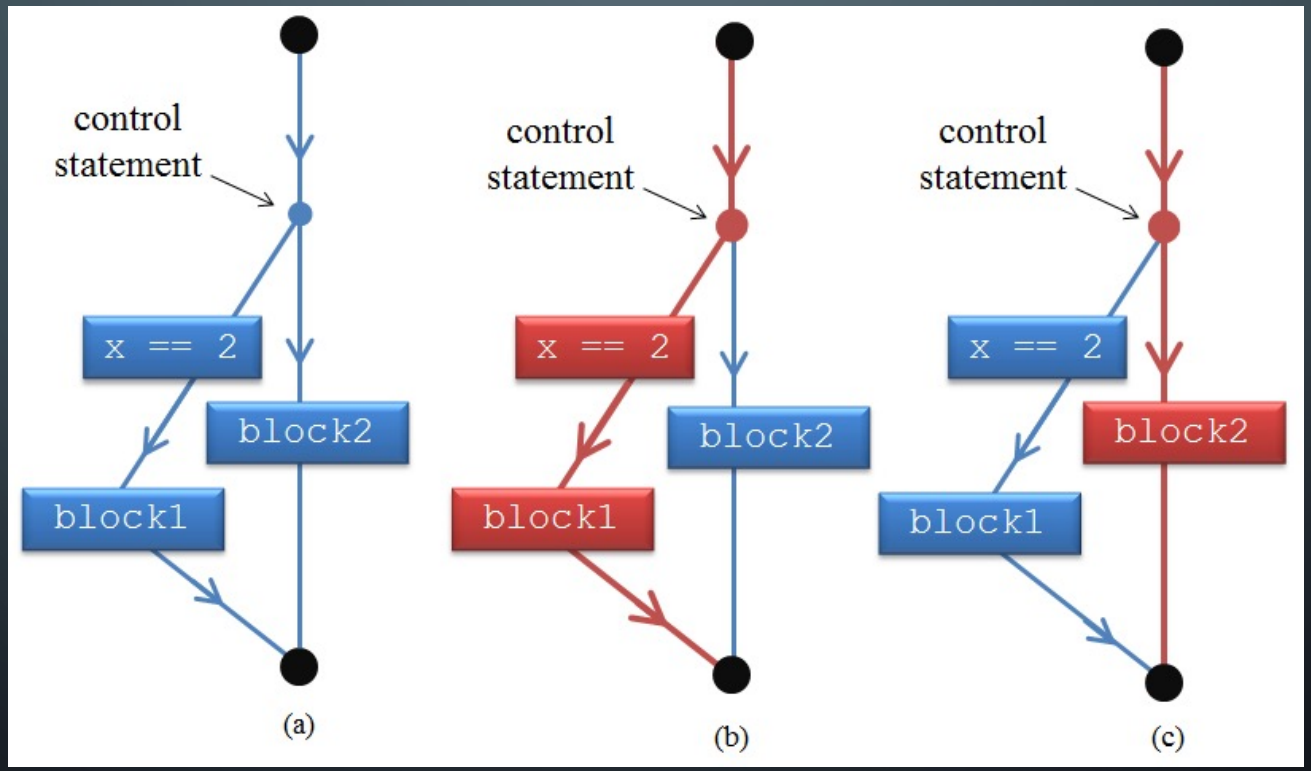
- Here is what the improved function looks like:

```
function guess_my_number(x)
if x == 2
   fprintf('Congrats! You guessed my number!\n');
else
   fprintf('Not right, but a good guess.\n');
end
```

- We have used another selection construct—the if-else-statement—and to do that we've introduced another keyword—else. This statement selects one of two statements to be executed but, as in the case of the if-statement above, each of these statements could be replaced by a block of two or more statements.

SELECTION - if statements

- Here is the schematic of if-else statement:



SELECTION - if statements

- Let's improve our function just a bit by trying to cheer up the poor user who fails to pick the correct number:

```
function ultimate_question(x)
if x == 42
   fprintf('Wow! You answered the ultimate question.\n');
elseif x < 42
   fprintf('Too small. Try again.\n');
else
   fprintf('Too big. Try again.\n');
end
```

- The new keyword is elseif . It is used in a new construct—the if-elseif-else- statement, and it allows us to check a second condition.

SELECTION - if statements

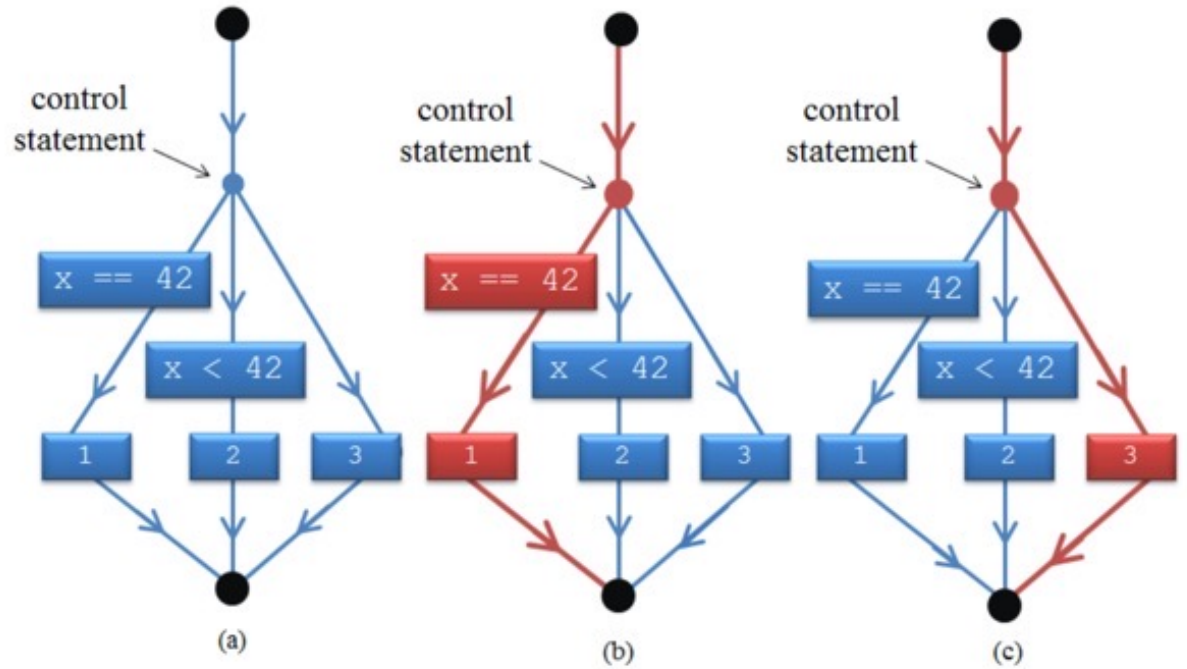


Figure 2.25 *Schematic of if-elseif-else The flow of control of an if-elseif-else-statement is shown. This figure is similar to the previous figure except that there are three blocks of code (labeled 1, 2, and 3), instead of two. There are three possible paths as shown in (a): one for* `x == 42`*, one for* `x < 42`*, and one for every other possibility. If* `x` *equals 42, then the left branch is followed and block 1 is executed, as shown by the red path in (b). If* `x` *neither equals 42, nor is less than 42 (i.e.,* `x` *is greater than 42), then block 3 is executed, as shown in (c). The red path for* `x < 42` *is not shown.*

SELECTION - if statements

- If we wanted to require a bit more thinking on the part of the person trying to answer the “ultimate question”, we could omit the else clause from our last example to get a harder ultimate_question:

```
function ultimate_question(x)
if x == 42
   fprintf('Wow! You answered the ultimate question.\n');
elseif x < 42
   fprintf('Too small. Try again.\n');
end
```

SELECTION - if statements

- So far, our examples have selected from among one, two, or three blocks of code, but any number of blocks can be included by including additional elseif keywords.

```
function day_of_week(n)
if n == 1
   fprintf('Sunday,');
   day_type = 2;
elseif n == 2
   fprintf('Monday,');
   day_type = 1;
elseif n == 3
   fprintf('Tuesday,');
   day_type = 1;
elseif n == 4
   fprintf('Wednesday,');
   day_type = 1;
elseif n == 5
   fprintf('Thursday,');
   day_type = 1;
elseif n == 6
   fprintf('Friday,');
   day_type = 1;
elseif n == 7
   fprintf('Saturday,');
   day_type = 2;
else
   fprintf('Number must be from 1 to 7.\n');
   return
end
if day_type == 1
   fprintf(' which is a week day\n');
else
   fprintf(' which is a weekend day\n');
end
```

SELECTION - The return statement

- There is a new keyword in the code above: **return**.
- When a return-statement is executed (in any programming language), it halts the function in which it appears, in this case, day_of_week, and returns control to the caller of the function. (When return is executed in the Command Window, it does nothing; in a script, it causes the script to halt, and control is returned to the Command Window)
- If that return-statement had been omitted, execution would have continued to the final if-else-statement:

```
>> day_of_week(-2)
Number must be from 1 to 7.
Undefined function or variable "day_type".
Error in day_of_week (line 27)
if day_type == 1
```

SELECTION - The conditional

- The name for the expression that follows the keywords if and elseif is **conditional**.
- A conditional is the expression that determines whether or not a block in an if-statement is executed.
- It can have one of two values—true or false.
- If it is true, the block is executed; if it is false, the block is not executed.
- We have seen several simple examples already: `x == 2`, `x == 42`, `x < 42`, `n == 1`, `n == 2`, `...`, and `day_type == 1`.

SELECTION - If Statement Summary

if-statement:

```
if conditional
   block
end
```

if-else-statement:

```
if conditional
   block
else
   block
end
```

if-elseif-statement:

```
if conditional
   block
elseif conditional
   block
end
```

if-elseif-else statement:

```
if conditional
   block
elseif conditional
   block
else
   block
end
```

SELECTION - Relational Operators

- A relational operator produces a value that depends on the relation between the values of its two operands.
- The operators, ==, and <, which we have seen in if-statements above, are examples of relational operators.
  - The operator == is symbolized by two equal signs (a notation borrowed from C/C++). It is the "is-equal-to" operator, or "equals" operator, and it means exactly what both names imply. When we use it as a conditional in an if-statement, it causes the block it governs to be executed if and only if its first operand is equal to its second operand. Note that this operator is very different from the assignment operator, which is symbolized by just one equal sign and which causes the value of the variable at its left to be set to the value at its right.
  - The operator symbolized by < is the "is-less-than" or "less then" operator, and it also has the meaning we would expect. When we use it as a conditional in an if-statement, it causes the block it governs to be executed if and only if its first operand is less than its second operand.

SELECTION - Relational Operators

- There are six relational operators:

OPERATOR	MEANING
==	is equal to
~=	is not equal to
>	is greater than
<	is less than
>=	is greater than or equal to
<=	is less than or equal to

SELECTION - Relational Operators

- Relational operators can appear outside control statements and they can produce a value.

```
>> 10 == 20
ans =
     0

>> 3 == 35-32
ans =
     1
```

- In MATLAB, when the operator == finds that its first operand is not equal to its second operand, it returns the value zero, which means “false”.
- When the == operator finds that its first operand is equal to its second operand, it returns the value 1, which mean “true”.

SELECTION - Relational Operators

- Every the relational operator returns 0 when its expression is false and 1 when its expression is true.

```
>> x = (45*47 > 2105) + 9
x =
    10
```

- In fact all the arithmetic operators have higher precedence than all the relational operators.

```
>> x = 45*47 > 2105 + 9
x =
     1
```

SELECTION - Relational Operators

- When an expression involves division, there is a danger that we may divide by zero, as in this example:

```
>> x = 16;
>> y = 0;
>> z = x/y
z =
   Inf
```

- We could handle that with an if-statement in the following way:

```
if y ~= 0
   z = x/y;
else
   z = x;
end
```

- It is also possible to accomplish the same thing with a single arithmetic expression involving a relational. operator as follows:

```
z = x/(y + (y==0))
```

- The following one sets z to 0, instead of x, when y is zero:

```
z = (y~=0)*x / (y + (y==0))
```

SELECTION - Relational Operators

- A convenient feature of the relational operators is that they are array operators. Thus, by giving them two operands that are arrays of the same size and shape, we can compare many pairs of values with just one expression.

```
>> [4 -1 7 5 3] > [5 -9 6 5 -3]
ans =
     0     1     1     0     1
```

- Also, like the array operators, if one operand is a scalar, then the other operand can have any size and shape, allowing us to compare many values to one value.

```
>> [4 -1 7 5 3] <= 4
ans =
     1     1     0     0     1
```

SELECTION - Logical Operators

- A logical operator produces a value that depends on the truth of its two operands.
- There are three logical operators:

OPERATOR	MEANING
&&	and
||	or
~	not

SELECTION - Logical Operators

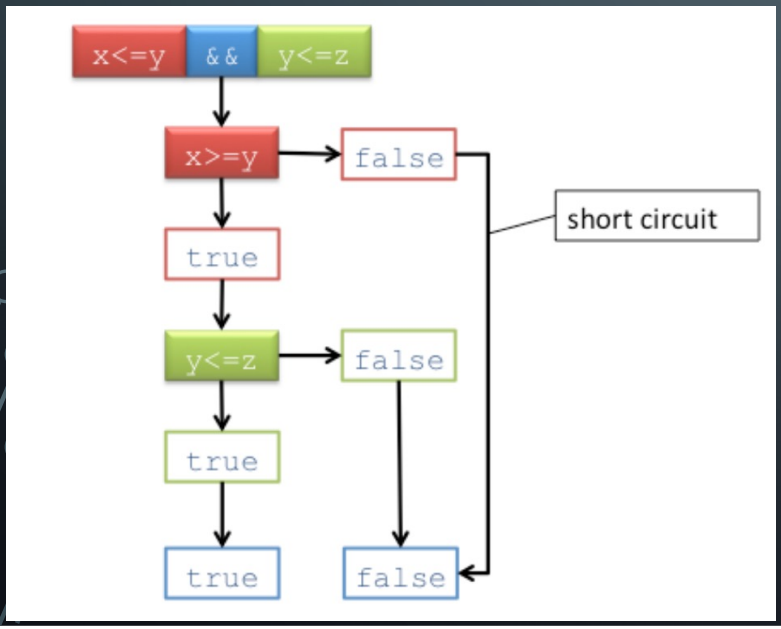
- Consider the following example:

```
function a = order3(x,y,z)
if x <= y && y <= z
   a = 1;
elseif x >= y && y >= z
   a = -1;
else
   a = 0;
end
```

- The && operator takes two operands. If both are true (values are nonzero), then it returns true (value of 1). Otherwise, it returns false (value of 0). To be clear, if either one of its operands is false (value of 0), it returns false (value of 0). This is the normal, everyday meaning of the word “and”.

SELECTION - Logical Operators

- In the expression, x <= y && y <= z , the first operand of the logical "and" operator && (i.e., the operand to its left), x <= y, is evaluated first, and if that operand is false, then the && operator returns false (value of 0)—without evaluating its second operand at all!
- It ignores its second operand when its first operand is false because evaluating the second operand would be a waste of time. Regardless of whether that second operand is true or false, the answer will be false whenever the first operand is false.
- **Skipping the evaluation of the second operand because its value will have no effect on an operator's result is called short circuiting.**



*Schematic of short-circuiting for **&&**. The flow of control is shown within a short-circuiting logical "and" operation: **&&**. The first operand of **&&** is red, and its possible outputs—true or false—are outlined in red. The second operand of **&&** is green, and its possible outputs are outlined in green. The two possible outputs of **&&** are outlined in blue. If the output of the first operand is false, then the path labeled "short-circuit" is followed, bypassing the evaluation of the second operand.*

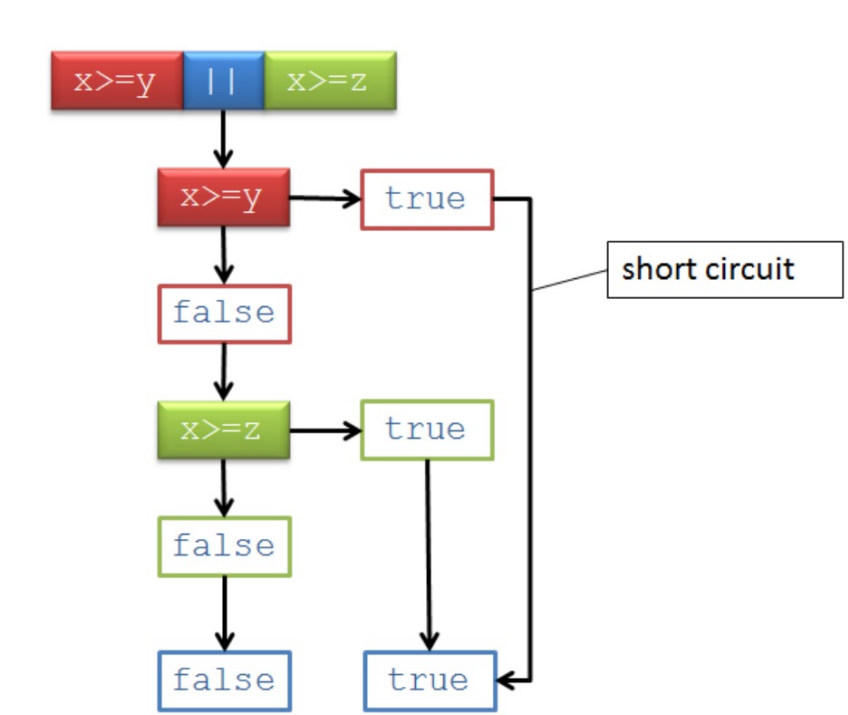
SELECTION - Logical Operators

- There is a second short-circuiting logical operator—the logical "or" operator, symbolized by ||.
- It returns true (value of 1) if at least one of its operands is true—the first one, the second one, or both—and (value of 0) if both operands are false.
- Consider the example below:

```
function a = not_smallest_version_2(x,y,z)
if x >= y || x >= z
   a = 1;
else
   a = 0;
end
```

SELECTION - Logical Operators

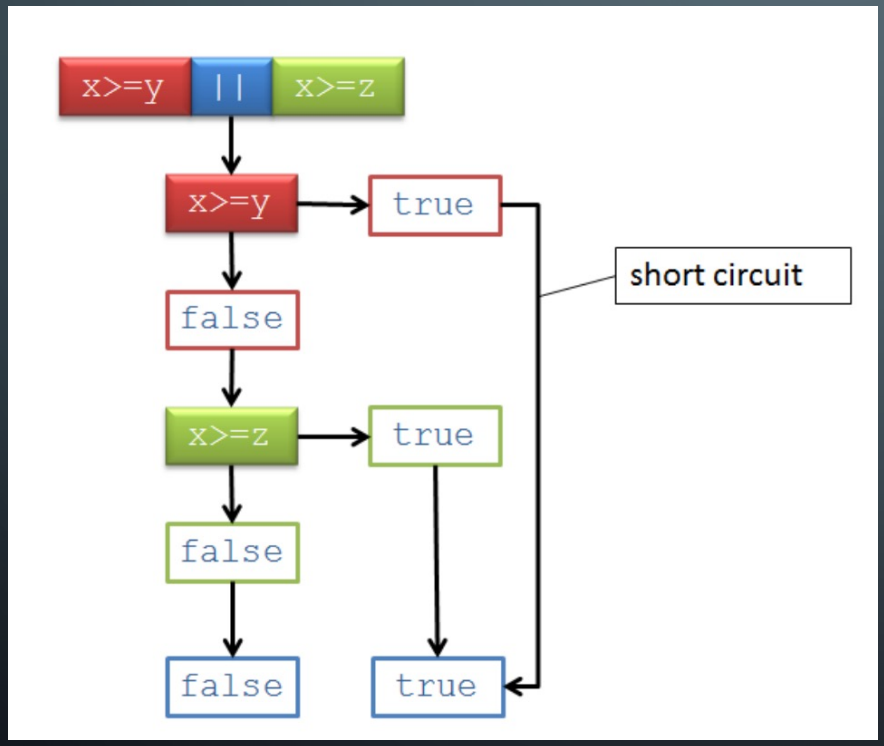
- Short-circuiting takes place if the first operand is true, because in that case, the ||operator will return 1, regardless of the value of the second operand.



Schematic of short-circuiting for ||. The flow of control is shown within a short-circuiting logical "or" operation: ||. The first operand of || is red, and its possible outputs –true or false—are outlined in red. The second operand of || is green, and its possible outputs are outlined in green. The possible outputs of || are outlined in blue. If the output of the first operand is true, then the path labeled "short-circuit" is followed, bypassing the evaluation of the second operand

SELECTION - Logical Operators

- Short-circuiting takes place if the first operand is true, because in that case, the ||operator will return 1, regardless of the value of the second operand.



Short circuiting may seem of little importance, because it may seem to be a trivial matter whether or not one operand is evaluated. However, it can save a lot of time if the second operand requires a long time to evaluate.

SELECTION – Logical Operators

- In the table below, the outputs of the logical "and" operator && and the logical "or" operator || are given for their four possible inputs and the outputs for the related function xor ("exclusive or") is given as well.
- There is no operator for xor. It is a function that takes two inputs as for example: xor(x,y).

INPUTS		&&	||	XOR
0	0	0	0	0
0	nonzero	0	1	1
nonzero	0	0	1	1
nonzero	nonzero	1	1	0

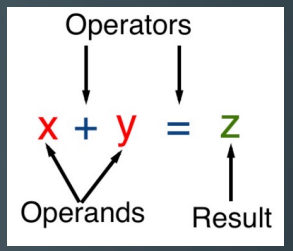
- It should be noticed that the inputs are **0** and "nonzero", instead of **0** and **1**. These are MATLAB's denotations of the values that are treated as meaning false and true when used as inputs to logical operations.
- Note that up to now we have seen only the number **1** used to mean true because that is the value that all the relational and logical operators return when their expressions are true.

SELECTION - Logical Operators

- These operators allow any value to be used as input, and among all possible input values, only zero means false. Here are examples of non-zero values being used as input to && and ||:

```
>> -1 && 1
ans =
     1
>> -1 || 0
ans =
     1
```

SELECTION - Logical Operators



- The third operator is the logical "not" operator, symbolized by ~
- The "not" operator is a unary prefix operator, meaning that it takes only one operand and it comes before its operand.
- Consider the following example:

```
function a = not_smallest_version_4(x,y,z)
if ~(x < y && x < z)
   a = 1;
else
   a = 0;
end
```

- The evaluation of the conditional proceeds as follows:
  - (1) the relational operation x < y is evaluated,
  - (2) the relational operation x < z is evaluated,
  - (3) the logical "and" operation x < y && x < z inside the parentheses is evaluated, and
  - (4) the not-operator is applied to the result. If the value produced by the "and" operation is 1 (meaning true), then the not-operation returns 0 (meaning false). If the value of the "and" operation is 0 (false), then the not-operation returns 1 (true).
  - In other words, ~ merely changes a zero to a 1 and a nonzero value to a zero.

SELECTION - Logical Operators

- Like the relational operators, but unlike && and ||, the operator ~ is an array operator: Thus, it can be applied to an array producing a “not” operation on each element:

```
>> ~[1    -1     0     0    pi     0     4]
ans =
     0     0     1     1     0     1     0
```

```
>> ~[-1<1   4==4   2>3   2~=2   9~=4   6>=7   6<=7]
ans =
     0     0     1     1     0     1     0
```

SELECTION – Logical Operators

- **Two more logical operators:**
  - "ELEMENT-WISE" VERSIONS OF THE LOGICAL "AND" OPERATOR AND THE LOGICAL "OR" OPERATOR.
- These operators are symbolized by a single & and a single |
- Like their double-symbol counterparts, they are both binary operators, and they perform the same logical operations as they do, namely, "and" and "or".
- However, these operators, like the relational operators and ~, are array operators: They are the array versions of && and ||, each of which can be applied only to scalars. Like the other binary array operators, they can also take one scalar operand and one non-scalar operand.

```
>> [4 0 pi -1 0 1/3] & [1 1 -2 0 0 8]
ans =
     1     0     1     0     0     1

>> [4 0 pi -1 0 1/3] | [1 1 -2 0 0 8]
ans =
     1     1     1     1     0     1

>> [1 0 2;0 4 -1] | [0 0 .3;0 4 0]
ans =
     1     0     1
     0     1     1
```

SELECTION – Operator Precedence

- The table below gives the complete precedence table for all MATLAB operators:

PRECEDENCE	OPERATOR
0	Parentheses: (...)
1	Exponentiation ^ and Transpose '
2	Unary +, Unary –, and logical negation: ~
3	Multiplication and Division (array and matrix)
4	Addition and Subtraction
5	Colon operator :
6	Relational operators: <, <=, >, >=, ==, ~=
7	Element-wise logical "and": &
8	Element-wise logical "or": |
9	Short-circuit logical "and": &&
10	Short-circuit logical "or": ||

- When there are two or more operators of the same precedence, left-to-right associativity is used, meaning that the order of operation is from left to right. Parentheses can override this associativity rule also.
- If you are in doubt, then anyone who reads your code will probably be in doubt too. In those cases, even if you know the rules, you would do well to add redundant parentheses to make the order of operations perfectly clear.
- MATLAB makes it easy to review the table. All you have to do is type **help precedence** in the Command Window

SELECTION - Nested Selection Statements

- We refer to the inclusion of one control construct inside another construct as nesting.
- For our first example, we will use nesting to rewrite a function that we wrote earlier without nesting—**ultimate_question**. We repeat that function below (cf. Slide 9):

```
function ultimate_question(x)
if x == 42
   fprintf('Wow! You answered the ultimate question.\n');
elseif x < 42
   fprintf('Too small. Try again.\n');
else
   fprintf('Too big. Try again.\n');
end
```

- We used this function to introduce the **elseif** clause, and that is the best way to write it, but in order to show how nesting works, here is a version with nesting that does not use **elseif**:

SELECTION - Nested Selection Statements

```
function ultimate_question_nested(x)
if x == 42
   fprintf('Wow! You answered the ultimate question.\n');
else
   if x < 42
      fprintf('Too small. Try again.\n');
   else
      fprintf('Too big. Try again.\n');
   end
end
```