4b. Conditions

Martin Alfaro
PhD in Economics

INTRODUCTION

This section lays the basics for incorporating conditions into our programs. Formally, conditions are defined as functions and operators that return true or false as their output. A common example of a condition is `x > y`.

To get the most out of this section, you should keep in mind the classification of operators discussed here. This establishes that operators can be categorized according to their number of operands. Specifically, **unary operators** act on a single operand and precede it (i.e. `<operator>x`), whereas **binary operators** take two operands and are placed between them (i.e. `x <operator> y`).

CONDITIONS

Conditions are represented as values with type `Bool`, evaluating to either `true` or `false`. These values are internally represented as integers restricted to `1` and `0`.

The representation of Boolean values in the REPL varies depending on their dimension: scalar `Bool` values are displayed as `true` and `false`, while `Bool` vectors use `1` and `0`. This is illustrated below.

```
x = 2

#`y` provides `true` or `false` as its output
y = (x > 0)
```

```
julia> y
true
```

```
x = 2

#'z' provides 'true' and 'false' as its output, represented by 1s and 0s
z = [x > 0, x < 0]
```

```
julia> z
2-element Vector{Bool}:
 1
 0
```

> **Warning!**
> Parentheses are optional when writing single conditions, allowing us to write `y = x > 0` rather than `y = (x > 0)`. Nonetheless, the former

syntax is somewhat ambiguous, with the risk of being potentially misinterpreted as `(y = x) > 0`. To avoid confusion, it's a good practice to always include parentheses. This is especially true when working with multiple conditions, where outcomes can be drastically altered.

The condition in the previous example was defined via the operator `>`. More generally, conditions accept **comparison operators**, which are *binary operators* that compare values of various types (e.g., numbers and strings). The next list defines the most common ones.

Comparison Operator	Meaning
`x == y`	equal
`x ≠ y` or `x != y`	not equal
`x < y`	lower than
`x ≤ y` or `x <= y`	lower or equal than
`x > y`	greater than
`x ≥ y` or `x >= y`	greater or equal than

Remark

The non-standard characters appearing in the table can be written using tab completion:

- `≠` via `\ne`, which stands for "not equal",
- `≥` via `\ge`, which stands for "greater or equal",
- `≤` via `\le`, which stands for "lower or equal".

Remark

Comparison operators are also available as functions. For instance, the following expressions are all valid:

```
==(1,2)     # same as 1 == 2
 ≠(1,2)     # same as 1 ≠ 2
 ≥(1,2)     # same as 1 ≥ 2
>=(1,2)     # same as 1 ≥ 2
 >(1,2)     # same as 1 > 2
```

LOGICAL OPERATORS

Logical operators allow us to combine multiple conditions into a single one. Formally, they take `Bool` expressions as their operands, and return another `Bool` as their output. The following are the main logical operators used in Julia.

Logical Operator	Meaning
`x && y`	`x` and `y`
`x || y`	`x` or `y`
`!x`	negation of `x`

Notice that `&&` and `||` follow the syntax rules of *binary* operators.

```
x = 2
y = 3

# are both variables positive?
z1 = (x > 0) && (y > 0)

# is either `x` or `y` (or both) positive?
z2 = (x > 0) || (y > 0)
```

```
julia> z1
true
julia> z2
true
```

Another operator taking conditions as their operands is the "not" operator, represented by `!`. This is a unary operator that inverts a condition's value, changing `true` to `false` and vice versa. To use it, you simply place `!` at the start of the condition (i.e., before the parentheses).

As an illustration, the variables `y1` and `y2` below become equivalent via `!`.

```
x = 2

# is `x` positive?
y1 = (x > 0)

# is `x` not less than zero nor equal to zero? (equivalent)
y2 = !(x ≤ 0)
```

```
julia> y1 #identical output as 'y2'
true
```

LOGICAL OPERATORS AS SHORT-CIRCUIT OPERATORS

A key feature of `&&` and `||` is that they're **short-circuit operators**. This means that, once an operand is evaluated, the remaining operands are evaluated only if the previous operands didn't establish the truth or falseness of the expression. Specifically:

- `(x > 0) || (y > 0)`
  This expression is true when *at least one condition* is satisfied. Thus, Julia begins by analyzing `x > 0`. If this expression is true, it immediately returns `true`, without evaluating any subsequent expression. Only when `x > 0` is false will Julia evaluate `y > 0`.

- `(x > 0) && (y > 0)`
  This expression is true if *both conditions* are satisfied. Thus, Julia begins by analyzing `x > 0`. If this expression is false, it immediately returns `false`, without evaluating any subsequent expression. Only when `x > 0` is true will Julia evaluate `y > 0`.

Since not all operands are always evaluated, it's possible to get a result even if some operands contain invalid expressions. This is shown in the next example, where we include invalid Julia code as a condition.

```
x = 10
```

```
julia> (x < 0) && (this-is-not-even-legitimate-code)
false
julia> (x > 0) && (this-is-not-even-legitimate-code)
ERROR: UndefVarError: `this` not defined
```

```
x = 10
```

```
julia> (x > 0) || (this-is-not-even-legitimate-code)
true
julia> (x < 0) || (this-is-not-even-legitimate-code)
ERROR: UndefVarError: `this` not defined
```

PARENTHESIS IN MULTIPLE CONDITIONS

The inclusion of parentheses isn't crucial when working with only two conditions. This is because expressions like `(x > 0) && (y > 0)` can be safely written as `x > 0 && y > 0`, without much risk of confusion.

On the contrary, when dealing with three or more conditions, the lack of parentheses can drastically impact the expected behavior of an expression. The following example illustrates this point.

```
x = 5
y = 0
```

```
julia> x < 0 && y > 4 || y < 2
true
```

```
x = 5
y = 0
```

```
julia> (x < 0) && (y > 4 || y < 2)
false
```

```
x = 5
y = 0
```

```
julia> (x < 0 && y > 4) || (y < 2)
true
```

In the example, the expression without parenthesis is equivalent to the last tab's, since `&&` **has higher precedence than** `||` in Julia: when both `&&` and `||` are used, `&&` will be evaluated first.

To avoid confusion when more than two conditions are incorporated, **we'll always add parentheses**. This improves readability and spares us the need to memorize specific rules. The next optional subsection covers Julia's precedence rules in more detail. However, if you'll consistently enclose conditions in parentheses, you can safely skip it.

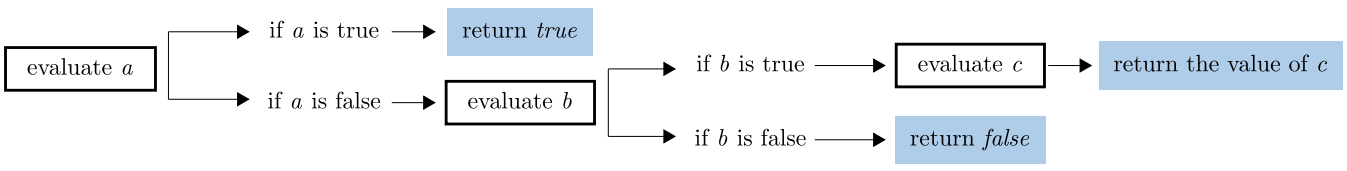
MULTIPLE CONDITIONS WITHOUT PARENTHESES (*OPTIONAL*)

To simplify the explanation, let's focus on cases with three conditions. These conditions will be represented through `Bool` variables `a`, `b`, and `c`, with each variable possibly representing expressions like `x > 0`.

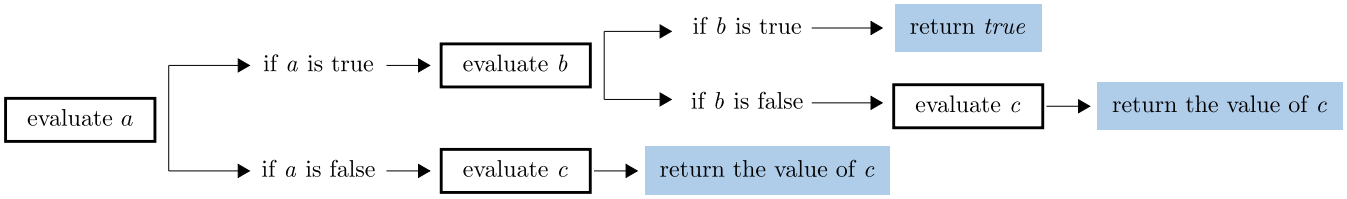
To understand how Julia groups three conditions without parentheses, there are two rules you need to know. First, `&&` has higher precedence than `||`. This means that `a && b || c` is equivalent to `(a && b) || c`, whereas `a || b && c` is equivalent to `a || (b && c)`. Second, `&&` and `||` are short-circuit operators. Thus, `a && b` immediately returns `false` if its first operand `a` is false, without evaluating the second operand `b`. Likewise, `a || b` returns `true` if the first operand `a` is true, without evaluating the second operand `b`.

The following diagrams describe the process for evaluating `a && b || c` and `a || b && c`, based on these two rules.

CASE 1: `a || b && c` is equivalent to `a || (b && c)`



CASE 2: `a && b || c` is equivalent to `(a && b) || c`



To illustrate the rules in practice, let's go through several examples that combine true/false values for `a`, `b`, and `c`. In these examples, we'll use the invalid expression `does-not-matter`. This is to emphasize that some conditions aren't necessarily evaluated thanks to the short-circuit behavior of `&&` and `||`.

```
julia> false || true && true
true
julia> false || true && false
false
julia> true || does-not-matter
true
```

```
julia> true && false || true
true
julia> true && false || false
false
julia> false && does-not-matter || true
true
```

FUNCTIONS TO CHECK CONDITIONS ON VECTORS: "ALL" AND "ANY"

Julia provides two built-in functions called `all` and `any` to evaluate multiple conditions in a collection. The function `all` returns `true` if *every condition* is true, whereas `any` returns `true` if *at least one condition* is true. The functions **require either directly specifying the conditions through a Boolean vector or defining the condition to check through a function**. Next, we cover each case separately.

VECTORS FOR REPRESENTING MULTIPLE CONDITIONS

In the following, we demonstrate the syntax of `all` and `any` when they take a Boolean vector as their argument.

```
a                = 1
b                = -1

# function indicating whether all elements satisfy the condition
are_all_positive = all([a > 0, b > 0])

# function indicating whether at least one element satisfies the condition
is_one_positive  = any([a > 0, b > 0])
```

```
julia> are_all_positive
false
julia> is_one_positive
true
```

The function `all` returns `true` only when all the conditions are satisfied, thus requiring that each vector's entry is positive. This doesn't hold in the example, since `b = -1`. Conversely, `any` returns `true` when at least one of the conditions holds, thus requiring at least one element in the vector to be positive. This is satisfied in the example, since `a = 1`.

As we indicated, `all` and `any` do not support passing multiple conditions as separate arguments. This entails that expressions like `all(a > 0, b > 0)` aren't allowed. Nevertheless, this restriction actually makes the functions more flexible, as they **enable the use of broadcasting operations for checking multiple conditions**. For example, the following code snippet implements the same operations as above, but through a vector `x`.

```
x                = [1, -1]

are_all_positive = all(x .> 0)
is_one_positive  = any(x .> 0)
```

```
julia> are_all_positive
false
julia> is_one_positive
true
```

FUNCTIONS FOR REPRESENTING MULTIPLE CONDITIONS

In addition to expressing conditions through vectors, `all` and `any` allow **passing a function to represent the condition to check**. The syntax for this is `all(<function>, <array>)` and `any(<function>, <array>)`, where `<function>` can be an anonymous function. The following examples demonstrate how to implement `all(x .> 0)` and `any(x .> 0)` using this approach.

```
x                = [1, -1]

are_all_positive = all(i -> i > 0, x)
is_one_positive  = any(i -> i > 0, x)
```

```
julia> are_all_positive
false
julia> is_one_positive
true
```

By passing a function as an argument, `all` and `any` can additionally be employed **to evaluate the same condition across multiple vectors**. This is achieved by broadcasting `all` and `any`.

```
x                = [1, -1]
y                = [1,  1]

are_all_positive = all.(i -> i > 0, [x,y])
is_one_positive  = any.(i -> i > 0, [x,y])
```

```
julia> are_all_positive # all elements in 'y' are positive, but not in 'x'
2-element BitVector:
 0
 1
julia> is_one_positive # at least one element of 'x' or 'y' is positive
2-element BitVector:
 1
 1
```