4c. Conditional Statements

Martin Alfaro
PhD in Economics

INTRODUCTION

Programs routinely must choose between alternative operations depending on how their execution unfolds. To handle these possibilities, programs rely on conditional statements, which enable the execution of specific code blocks only when certain conditions are satisfied.

Each code block within a conditional statement is referred to as a **branch**. Based on the number of branches, there are three types of conditional statements:

- **if-then statements**, which consist of a **single branch**. They run a specific operation only if a condition is met, with no operation performed otherwise.
- **if-else statements**, which consist of **two branches**. They run a specific operation if a condition is met, and another if the condition isn't satisfied.
- **if-else-if statements**, which consist of **three or more branches**. They comprise a series of conditions, with each branch executing a different code block.

Next, we cover each case in depth. The presentation builds heavily on the operators introduced in the previous section. If you haven't read it, I strongly recommend doing so before continuing.

IF-THEN STATEMENTS

If-then statements execute an operation only when a condition is met, doing nothing instead when the condition isn't satisfied. These statements can be constructed via:

- the `if` keyword,
- the logical operator `&&`,
- the logical operator `||`.

The approach via `if` keyword is self-explanatory. As for the logical operators, `&&` executes an operation if the condition is true, whereas `||` does it when the condition is *not* satisfied. In fact, `||` is equivalent to `&&` with its condition negated.

Below, we illustrate the syntax for each form. The examples rely on the `println` function, which displays the text passed as argument in the REPL.

```
x = 5

if x > 0
    println("x is positive")
end
```

```
x is positive
```

```
x = 5

(x > 0) && (println("x is positive"))
```

```
x is positive
```

```
x = 5

(x ≤ 0) || (println("x is positive"))
```

```
x is positive
```

Note that if-then statements imply that no action would've been taken if, for instance, we had used `x = -1` as a condition. It's only when `x > 0` that `println` is executed.

The `if` approach offers the most flexibility, making it ideal for complex conditional statements. However, it's somewhat verbose for simple conditional statements. For these cases, `&&` and `||` are preferred, as they help us keep the code streamlined.

A common application of `||` is in conjunction with the function `error` to handle errors. This construct immediately interrupts the script's execution when the condition isn't satisfied, displaying the provided message as the argument of `error`. For instance, consider a function `foo(x)` that requires non-negative values for `x`. To enforce this, you could include `x > 0 || error("x must be positive")` at the beginning of the function. If `foo(x)` is then called with a non-positive `x`, it'll immediately halt its execution and print the error message "x must be positive" in the REPL.

> **Remark**
>
> Note that `&&` and `||` behave like if-then statements when they combine a condition with an operation. This is different from using them exclusively with conditions, where all operands would be `Bool` values.

IF-ELSE STATEMENTS

If-else statements execute an operation when a condition is true and another operation when the condition is false. There are two forms to write these statements.

The first one is the most flexible and uses the `if` keyword in combination with `else`. The second method relies on the so-called **ternary operator**, which requires the keywords `?` and `:` via the syntax `<condition> ? <operation if true> : <operation if false>`. This is referred to as *the* ternary operator because it's the only operator in most programming languages that takes three arguments.

We illustrate the syntax of both approaches below.

```
x = 5

if x > 0
    println("x is positive")
else
    println("x is not positive")
end
```

```
x is positive
```

```
x = 5

x > 0 ? println("x is positive") : println("x is not positive")
```

```
x is positive
```

The function `ifelse` offers an alternative for constructing if-else expressions. This function takes three arguments: a condition, an expression to be evaluated if the condition is true, and another one if false. [1]

One advantage of using a function for an if-else statement is that it supports broadcasting. This is particularly helpful when creating vectors whose elements vary according to a condition, as demonstrated below.

```
x                    = [4, 2, -6]

are_elements_positive = ifelse.(x .> 0, true, false)
```

```
julia> are_elements_positive
3-element BitVector:
 1
 1
 0
```

```
x                   = [4, 2, -6]

x_absolute_value    = ifelse.(x .≥ 0, x, -x)
```

```
julia> x_absolute_value
3-element Vector{Int64}:
 4
 2
 6
```

> **Remark**
>
> Broadcasting `ifelse` **requires broadcasting both** `ifelse` **and the condition**. The first example, for instance, would throw an error if we execute `ifelse.(x>0, true, false)`. This is because `x > 0` would attempt to check if the *entire vector* is positive, which is an operation undefined in Julia.

IF-ELSE-IF STATEMENTS

So far, we've analyzed conditional statements that handle only two possibilities: one when the condition is met, and another if it isn't. This binary structure can be limiting when multiple alternatives need to be considered. Basically, it forces you to nest several `if` and `else` statements to manage additional conditions.

To simplify this process, we can use the `elseif` keyword to extend the `if` and `else` approach. This is illustrated below.

```
x = -10

if x > 0
    println("x is positive")
elseif x == 0
    println("x is zero")
end
```

```
x = -10

if x > 0
    println("x is positive")
elseif x == 0
    println("x is zero")
else
    println("x is negative")
end
```

```
x is negative
```

The first examples showcase the benefits provided by the approach. Specifically, `elseif` eliminates the need to explicitly specify actions for every possible scenario. Instead, it performs an action if `x` is positive, another action if `x` is zero, but it does nothing otherwise. In contrast, using `if` and `else` would require an exhaustive approach, where all possible contingents must be accounted for.

Likewise, the second example demonstrates that combinations of the `if`, `else`, and `elseif` keywords are possible.

FOOTNOTES

[1] The function `ifelse` does *not* behave as a **short-circuit operator**. This means that all the operations are computed, despite that only one of them will ultimately be returned as output.