Unit 1: Programming (scripting) Bootcamp

Table of contents

In this unit we'll study the very basics of programming using the Julia language, though the concepts apply to pretty much every major programming language with slight modifications to the syntax. The unit ends with suggested class micro-projects and practice questions for the upcoming quiz. The pace and nature of this unit is aimed at students with little or no programming experience. The units afterwards pick up the pace significantly.

What to expect from this unit:

- Variables
- Types
- Using in-built functions (e.g. `println`) and macros (e.g. `@show`)
- Logical statements (AND, OR, etc..)
- Conditional statements (`if-elseif-else`)
- For loops
- While loops
- Creating your own functions with (`function`)

1 Getting Started

Install Julia for your particular type of computer following the steps listed on the Julia website. Then open a terminal on your computer and start Julia with the command ''julia''. This should give you about the following output.

```
dietmar@Nina:~/github/MATH2504/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems$ julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.12.6 (2026-04-09)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org release
|__/                   |

julia>
```

1.1 Command line interface (REPL) - using Julia as a calculator

The prompt `julia>` represents the Julia command line interface called **REPL** (Read-Eval-Print-Loop). Start by **using it as a calculator**, e.g. at the Julia prompt enter

```
5+6
```

```
11
```

or

```
sin(25/180)
```

```
0.13844278873711527
```

Here the function `sin` from the standard library is called.

1.2 Generating output - Hello World!

Another function from the standard library can be called as follows:

```
println("Hello World!")
```

```
Hello World!
```

We are calling (or invoking) the function `println` with the argument `"Hello World!"`. The name of the function refers to ''Print (and go into the next) line''. It takes the argument (the text `"Hello World!"` in this case) and writes it into the next line after the command, finally ending the line and moving the cursor into the next line.

The `println` function can also take several arguments.

```
println("hello", " ", "world")
```

```
hello world
```

Here is `print` without a new line.

```
print("hello")
print(" ")
print("world")
```

```
hello world
```

The symbol `\n` represents a `new line`-command,

```
println("hello\nworld") #notice the \n (newline)
println("Here is the next line")
```

```
hello
world
Here is the next line
```

Note that **the symbol # in Julia is used to declare a single-line comment** (everything after # is part of a comment and doesn't affect program execution).

Round brackets after a name are the application of a function, kind of like in mathematics. Keep in mind that round brackets also have other uses: For example, dealing with the order of precedence of computations:

```
1 + 1 * 4
```

```
5
```

as compared to

```
(1 + 1) * 4
```

```
8
```

or something more involved such as

```
(1/sin(25/180)+2.3)
```

```
9.523200349560057
```

1.3 Macros

Note that '@show' does something very similar to `println`. It is a **macro (all macros have @ in their names)**.

```
@show(3/425)
s="3.5";
@show s;
println(s)
```

```
3 / 425 = 0.007058823529411765
s = "3.5"
3.5
```

Macros are evaluated at compile time (when translating the Julia code into machine code) and therefore have access to more information, e.g. the variable name of their argument. For that reason @show can also output the variable name 's' which is convenient for debugging, whereas prinln only has access to the content '3.5' of its argument.

1.4 Using code editors

The *REPL* is great for using Julia as a calculator. **Multi-line input (scripts, programs)** can be **pasted line-by-line and sometimes even at once directly into the REPL**, but it is more convenient to **store it into a .jl file**, and then

- invoke the file line by line either from the terminal running 'julia filename.jl',
- or from the REPL using 'include("filename.jl")'.

Or, you use a **dedicated editor**, e.g.

- install VSCode and, once you started the editor, install the Julia language extension. (If you can't make it work, we'll show you in the tutorials how to do this) and
- the Jupyter notebook interface which can be installed by running, line-after-line, in the REPL
  ```
  using Pkg
  Pkg.add("IJulia")
  using IJulia
  notebook()
  ```

1.5 Input a variable through user interaction

Store the following 3 lines into the file 'test.jl' and run it either by running the command 'julia test.jl' from the terminal or through include("test.jl") from the REPL, or directly from VSCode.

```
println("What is your name: ")
username=readline()
println("Hi ",  username, ", how are you?")
```

This short script outputs the question about your name, and then used the function `readline` to read in your name and store it into the variable `name`. Finally it outputs a greeting which includes your particular name.

Try running this simple script in several ways:

- REPL (by running Julia and using the REPL, line after line)
- Jupyter (in a Jupyter notebook with Ctrl+Enter),
- In a file (by creating a file such as my_code.jl and either running `julia my_code.jl` from the terminal or include("my_code.jl") within the REPL)
- In VS-Code with the Julia extension.

2 Variables and types

In the last example the variable `username` acted as a container for the name the user would input. Indeed, one may think of variables as a **storage box**, with a name (tag) on it. The computer's memory is the **storage facility** in which you can store all your boxes.

Figure 1: Variables are boxes in a storage facility (memory)

For example the command

```
x = 3
```

```
3
```

stores the value 3 in the box tagged `x`.

Later, you may retrieve the value stored in the variable `x`, for example the command

```
x + 25
```

```
28
```

retrieves the value stored in x and adds 25 to it.

The **assignment operator =** is also used to assign the value (content) of one variable to another variable, or assign the result of a computation to a variable,

```
y=x
z=y+5
println("Now x = ", x, ", y = ", y, " and z = ",z, ".")
```

```
Now x = 3, y = 3 and z = 8.
```

Algebraic operations on numbers

```
x = 3.0
y = 4.2
x*y, x+y, x-y, x/y, x^y
```

```
(12.600000000000001, 7.2, -1.2000000000000002, 0.7142857142857143, 100.90420610885693)
```

```
u = 33
v =53.234
@show(u % 4) #u modulo 3
@show (u ÷ 7) #integer division without rest
@show (rem(u,7)) #this returns the rest after division
@show(v % 7)
```

```
u % 4 = 1
u ÷ 7 = 4
rem(u, 7) = 5
v % 7 = 4.234000000000002
```

```
4.234000000000002
```

Mathematical functions

```
x = 2
√x #\sqrt + [TAB]
```

```
1.4142135623730951
```

```
sqrt(x)
```

```
1.4142135623730951
```

```
y = -x
```

```
-2
```

```
sqrt(y)
```

```
DomainError: DomainError(-2.0, "sqrt was called with a negative real argument but will only return
a complex result if called with a complex argument. Try sqrt(Complex(x)).")
DomainError with -2.0:
sqrt was called with a negative real argument but will only return a complex result if called with
a complex argument. Try sqrt(Complex(x)).
Stacktrace:
 [1] throw_complex_domainerror(f::Symbol, x::Float64)
   @ Base.Math ./math.jl:33
 [2] sqrt
   @ ./math.jl:627 [inlined]
 [3] sqrt(x::Int64)
   @ Base.Math ./math.jl:1546
 [4] top-level scope
   @
~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-1.qmd:211
```

```
z = sqrt(y + 0im)
```

```
0.0 + 1.4142135623730951im
```

Let's see some mathematical in-built functions:

```
exp(1)
```

```
2.718281828459045
```

```
 #\euler + [TAB],
```

```
 = 2.7182818284590...
```

```
 #\pi + [TAB]
```

```
 = 3.1415926535897...
```

```
log( ^2), log2(2^5), log10(10^3), sin(3 ), cos(0.1), tan( /3)
```

```
(2.0, 5.0, 3.0, 3.6739403974420594e-16, 0.9950041652780258, 1.7320508075688767)
```

```
abs(-3.5), abs(3.5), abs(3 + 4im)
```

```
(3.5, 3.5, 5.0)
```

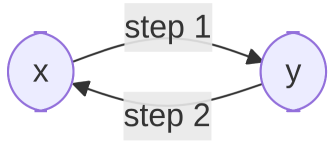
```
factorial(4)
```

```
24
```

2.1 Swapping values

In Julia, there is special notation ("syntactic sugar") for swapping values

If you do it in that `naive` way, then the following will happen.

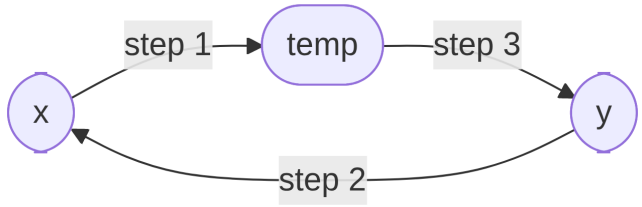



```
x = 20
y = 5

#swapping x and y:  attempt 1
x = y
y = x

@show x; #This is the @show macro
@show y;
```

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

So the value in `x` is overwritten and lost as soon as it is overwritten by the value in `y`.

This can be fixed by storing the value in `x` temporarily in another variable.




```
x = 20
y = 5

#swapping x and y:  attempt 2
temp = x
x = y
y = temp

@show x;
@show y;
```

```
x = 5
y = 20
```

To simplify the notation for this, in Julia we can do it without using the variable `temp`:

```
x = 20
y = 5

x, y = y, x #Julia specific solution

@show x;
@show y;
```

```
x = 5
y = 20
```

2.2 Numeric Types

Variables have a type - think of it as the size and shape of the box. The command

```
typeof(x)
```

```
Int64
```

gives the type of the variable x (an integer value of size 8 bytes). The size can be obtained using the function `sizeof`, e.g.

```
sizeof(x)
```

```
8
```

The type Int64 is the standard type for integers. Particular integer numbers are written without `.`:

```
typeof(32)
```

```
Int64
```

Another big class of numeric variables are floating-point numbers. Specific floating-point numbers are written with a `.`, e.g.

```
y=34.23
typeof(y), sizeof(y)
```

```
(Float64, 8)
```

defines a Floating point number of size 8 bytes = 64 bits. Floating point numbers are also the result when we divide integer numbers, e.g.

```
r=3/4
println(r)
typeof(r)
```

```
0.75
```

```
Float64
```

and the result of many functions which typically return floating point numbers such as

```
result=cos(2)
result, typeof(result)
```

```
(-0.4161468365471424, Float64)
```

2.3 Variables of type String (i.e. text) and Char(acter)

Variables can not only contain numbers, but pretty much every type of data, for example letters and text. The data type of single letters is called `Char`, which is short for character. The values of type `Char` are written with simple `'`, for example

```
c = 'a'
typeof(c)
```

```
Char
```

```
y = ' ' # \eta + [TAB]
typeof(y), sizeof(y)
```

```
(Char, 4)
```

On the other hand, the data type for text is called String (`"a string of characters!"`). Keep in mind that double `"` are used to write Strings.

```
str = "Hello"
typeof(str)
```

```
String
```

The length of the string (number of characters) can be obtained from the built-in function `length`:

```
length(str)
```

```
5
```

But you can't do absolute value of a string:

```
abs("3.5")
```

```
MethodError: MethodError(abs, ("3.5",), 0x00000000000097a9)
MethodError: no method matching abs(::String)
The function `abs` exists, but no method is defined for this combination of argument types.
Closest candidates are:
  abs(::Bool)
   @ Base bool.jl:155
  abs(::Pkg.Resolve.VersionWeight)
   @ Pkg
~/.julia/juliaup/julia-1.12.6+0.x64.linux.gnu/share/julia/stdlib/v1.12/Pkg/src/Resolve/versionweights.jl:32
  abs(::Missing)
   @ Base missing.jl:101
  ...
Stacktrace:
 [1] top-level scope
   @
~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-1.qmd:380
```

And for strings even some algebraic operations are defined,

```
s1 = "hello " #notice the extra space
s2 = "world"
@show typeof(s1);
s1*s2 #In Python it would have been x+y for concatenation
```

```
typeof(s1) = String
```

```
"hello world"
```

Even the exponential operator is defined for strings:

```
s = "hello "
y = 5
x^y
```

```
3125
```

These are examples of **operator (method) overloading** in Julia: For different data types, different versions of a function/method/operator can be defined.

```
s^(y-1)*s[1:end-1]
```

```
"hello hello hello hello hello"
```

Since the operator ^ has precedence of *, this is the same as

```
(s^(y-1))*s[1:end-1]
```

```
"hello hello hello hello hello"
```

2.4 Common errors: type mismatch

What if we did the brackets the other way?

```
s^((y-1)*s[1:end-1])
```

```
MethodError: MethodError(*, (4, "hello"), 0x00000000000097ab)
MethodError: no method matching *(::Int64, ::String)
The function `*` exists, but no method is defined for this combination of argument types.
Closest candidates are:
  *(::Any, ::Any, ::Any, ::Any...)
   @ Base operators.jl:642
  *(::Missing, ::Union{AbstractChar, AbstractString})
   @ Base missing.jl:174
  *(::Real, ::Dates.Period)
   @ Dates
~/.julia/juliaup/julia-1.12.6+0.x64.linux.gnu/share/julia/stdlib/v1.12/Dates/src/periods.jl:91
  ...
Stacktrace:
 [1] top-level scope
   @
~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-1.qmd:412
```

Here, Julia tries to identify an instance of the method "*(.,.)" which takes an Integer and a String as arguments (this is Julia's **"multiple dispatch"** paradigm), but doesn't succeed. The closest candidates are listed in the error message, but none of them fits (e.g., any argument of type Int64 can be converted into types Real, Number, Any, etc).

2.5 Type conversions and string interpolation

For type conversions, in particular between integers and floating point numbers, use `Float64` and `Int` as though they were functions:

```
a=4;
x=Float64(a);
@show typeof(a) a;
@show typeof(x) x;
```

```
typeof(a) = Int64
a = 4
typeof(x) = Float64
x = 4.0
```

```
y=sqrt(5);
b=round(y);
@show typeof(y) y;
@show typeof(b) b;
```

```
typeof(y) = Float64
y = 2.23606797749979
typeof(b) = Float64
b = 2.0
```

Here, `b` is still of type `Float64`. Only after applying `Int` it becomes an integer

```
basInt=Int(b)
@show basInt;
@show typeof(basInt);
```

```
basInt = 2
typeof(basInt) = Int64
```

The standard method to convert numbers into Strings uses the function **`string`** (lowercase!). E.g.

```
a=3
f=3.333
str="A string which includes a="*string(a)*" and f="*string(f)
```

```
"A string which includes a=3 and f=3.333"
```

As special type of type conversion is **String interpolation** using the symbol `$` which automatically converts a variable (and even an entire expression) into a String and inserts it into an existing string, e.g.:

```
i = 234
f = pi/2
println("An example of an integer is i=$i and an example for a floating-point number is f=$f.")
```

```
An example of an integer is i=234 and an example for a floating-point number is f=1.5707963267948966.
```

2.6 Indexing, Vectors and Matrices

Particular characters in a String can be retrieved using **`indexing`**. **Julia indexing is 1-based**, i.e. index 1 refers to the first letter, etc., whereas in other programming languages the first entry has index `0` be default.

```
str2 = "This is an example string!"
```

```
"This is an example string!"
```

The following diagram illustrates how to retrieve characters from a string by index:

```
String: "This is an example string!"

Index position (1-based):
   1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
   T  h  i  s     i  s     a  n     e  x  a  m  p  l  e     s  t  r  i  n  g  !

Examples:
  str2[1]      →  'T'    (first character)
  str2[7]      →  's'    (character at index 7)
  str2[end]    →  '!'    (last character)
  str2[end-5:end]  →  "tring!"  (substring from index 20 to 25)
  str2[10:15]  →  "nample"    (substring from index 10 to 15)
```

```
c1=str2[1]
println("The first letter in the string is: ", c1)
typeof(c1)
```

```
The first letter in the string is: T
```

Char

```
c7=str2[7]
println("and the 7th letter in the string is: ", c7)
```

```
and the 7th letter in the string is: s
```

The special index **end** can be used to the last character of the string

```
clast=str2[end]
println("and the final letter in the string is: ", clast)
```

```
and the final letter in the string is: !
```

Once can even retrieve sub-sections of a string, which gives another string, e.g.

```
string_end=str2[end-5:end]
println("and the final section of length 6 is: ", string_end)
```

```
and the final section of length 6 is: tring!
```

or

```
middle=str2[10:15]
println("and a middle section of length 6 is: ", middle)
```

```
and a middle section of length 6 is: n exam
```

Another example of types of variables that admit indexing are vectors

```
u=[1,2,3]
u[2]=5
v=u
```

```
3-element Vector{Int64}:
 1
 5
 3
```

and matrices

```
M=[1 2 3; 4 5 6; 7 8 9]
M*v
```

```
3-element Vector{Int64}:
 20
 47
 74
```

```
M[1,1]=0
M*v
```

```
3-element Vector{Int64}:
 19
 47
 74
```

Note that dealing with such higher-dimensional objects is one of the big strengths of Julia and will be discussed in the coming units. For now, vectors (and matrices) only serve as examples of composite variables.

2.7 Mutable vs immutable composite types

A big difference between Strings on the one hand and collections such as Vectors as well as Matrices on the other hand is that Strings in Julia are immutable whereas Vectors and Matrices are mutable. This means that Strings can not be modified in hindsight, whereas Vectors (and Matrices) can be modified after creation.

```
str="Here, I am creating a string variable"
str[5]='c'
```

```
MethodError: MethodError(setindex!, ("Here, I am creating a string variable", 'c', 5),
0x00000000000097b7)
MethodError: no method matching setindex!(::String, ::Char, ::Int64)
The function `setindex!` exists, but no method is defined for this combination of argument types.
Stacktrace:
 [1] top-level scope
   @
~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-1.qmd:540
```

This creates an error whereas for a Matrix

```
v=[1,2,3]
v[1]=10
@show(v)
M=[1.0 2.0 3; 4 5 6; 7 8 9]
M[1, 2]=200.0
@show(M)
```

```
v = [10, 2, 3]
M = [1.0 200.0 3.0; 4.0 5.0 6.0; 7.0 8.0 9.0]
```

```
3×3 Matrix{Float64}:
 1.0  200.0  3.0
 4.0    5.0  6.0
 7.0    8.0  9.0
```

2.8 Storage-by-reference vs storage-by-value

Variables of type String are more complex than those for simple numbers. They are composed of a sequence of simpler components of type `Char`. In particular, the length of a String can vary, and therefore also the amount of memory it occupies.

For these reasons, Strings are examples of **composite types**, while the simple datatypes such as those for basic integers and floating point numbers, and the one of characters are called **primitive types**. This can be verified using the built-in function `isbits` which returns `true` (a logical constant - see below) for primitive types, and false for composite types.

```
isbits(x), isbits(c), isbits(str)
```

```
(true, true, false)
```

This has important consequences for how variables are stored in memory. Variables of **primitive types**, i.e. for which the variable **directly** refers to the value of the variable, are stored IN the variable - "in the box" denoted by the name of the variable (see figure above).

Variables of **composite types**, on the other hand, such as Strings do not refer directly to their content, but to the address (**reference**, pointer) in memory where the content is stored. Think of the reference as the shelf number in the computer memory which plays the role of a storage facility. For example in the figure above, the String variable `str` contains the reference (address) `5` which is the position in memory where the actual string containing "Heelo" is stored

This has striking consequences. For example, when assigning the content of one primitive variable to another variable, only the content is copied

```
x=10
y=x
@show(y)
```

```
y = 10
```

```
10
```

so we can change the content of one variable without modifying the content of the other one.

```
x=100
println(y)
```

```
10
```

On the other hand, for a composite type such as vectors and matrices,

```
u=[1,2,3]
v=u
```

```
3-element Vector{Int64}:
 1
 2
 3
```

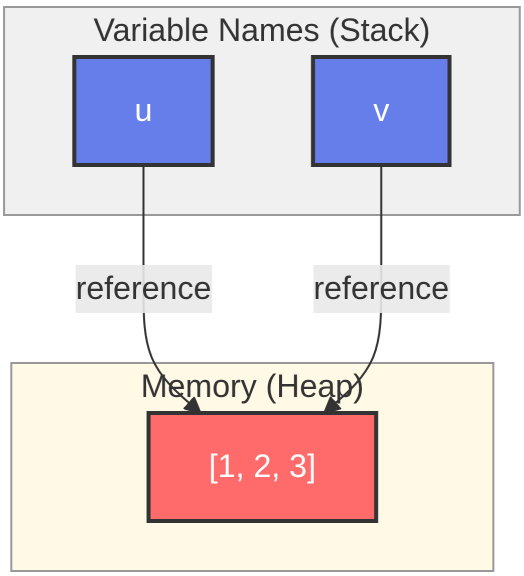
This defines a vector of length 3 and copies the reference to this vector from u into v.

When we modify the content of u using indexing, we also modify the content of v:

```
u[1]=10
@show(v)
```

```
v = [10, 2, 3]
```

```
3-element Vector{Int64}:
 10
  2
  3
```



The reason is that using indexing, we directly modify the section in memory, to which both references in `u` and in `v` are pointing.

3 Logical statements and conditional execution

The data type Bool has only two possible values,

```
true, false #In Python it is True and False (caps first letter)
```

```
(true, false)
```

```
typeof(true)
```

```
Bool
```

Internally, those two values are represented like the integer values `0` and `1` (actually in programming languages which are closer to the hardware representation of the program, this is exactly how logical values are treated).

```
@show Int(true);
@show Int(false);
```

```
Int(true) = 1
Int(false) = 0
```

3.1 Logical operators

```
5 == 2 + 3 # check for equality
```

```
true
```

```
5 != 2 + 3 # check for not being equal (!=)
```

```
false
```

```
false && false, false && true, true && false, true && true #logical AND
```

```
(false, false, false, true)
```

```
false || false, false || true, true || false, true || true #logical OR
```

```
(false, true, true, true)
```

```
(2 != 3) || (2 == 3)
```

```
true
```

```
!(2 == 3) # ! (not)
```

```
true
```

Logical AND (`&&`) has precedence of logical OR (`||`) as can be seen in

```
true || false && false
```

```
true
```

For the next block, you need to have the package `Random` installed: To install it, change to package mode using `]`, and use `add Random`.

The REPL modes are

- ; to access the shell/terminal temporarily,
- ? for help mode,
-] for package mode,
- Ctrl-C or Backspace brings you back to the standard "Julian" prompt.

```
using Random

x = rand(1:100) #random number within 1, 2, ..., 100
y = rand(1:100) #random number within 1, 2, ..., 100

@show x, y;

(x == y) || (x != y)
```

```
(x, y) = (40, 93)
```

```
true
```

```
x < y, x<=y, x   y, x > y, x   y # \le or \ge + [TAB]
```

```
(true, true, true, false, false)
```

```
(x == y) && (x != y)
```

```
false
```

3.2 Logical operators for composite types

```
x=[1,2,3]
y=[1,2,3]
x==y
```

```
true
```

compares the content of x to the content of y. To check whether x and y are physically identical (same reference) use

```
@show x===y;
z=y
@show z===y;
```

```
x === y = false
z === y = true
```

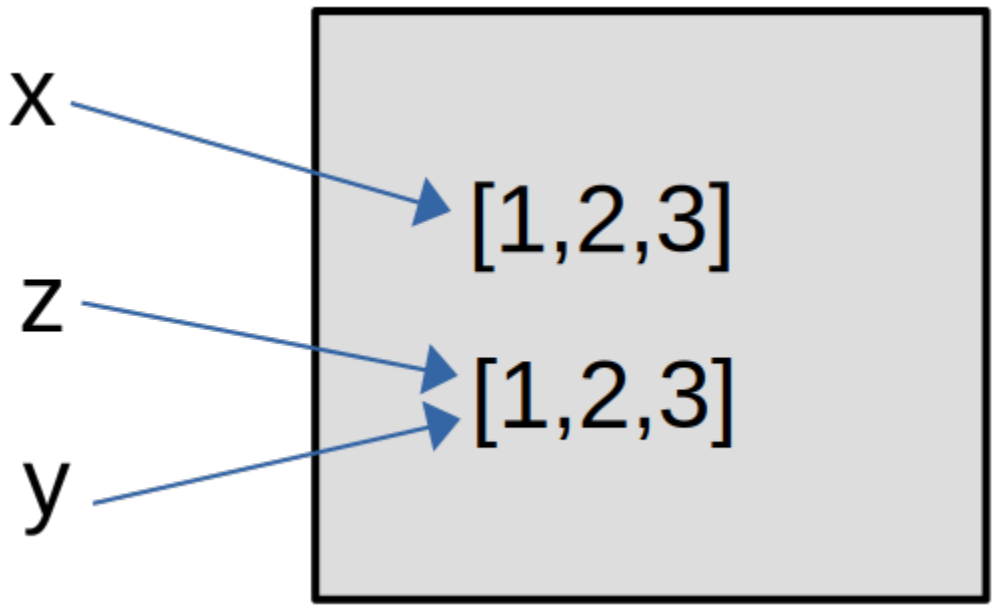


Figure 2: composite types

This is again a consequence of **storage-by=reference**. Initially the vector `[1,2,3]` is **physically saved in memory twice**. The reference to the first one is stored in the variable `x`, the reference to the second one is stored in `y`. For that reason the comparison `x==y` evaluates to `true`, this is because the content of the two vectors is equal. But when comparing the two variables using the comparison operator `===`, the result is `false`, because physically the two variables "point to" two different positions in memory.

Note that **for Strings the result is different**,

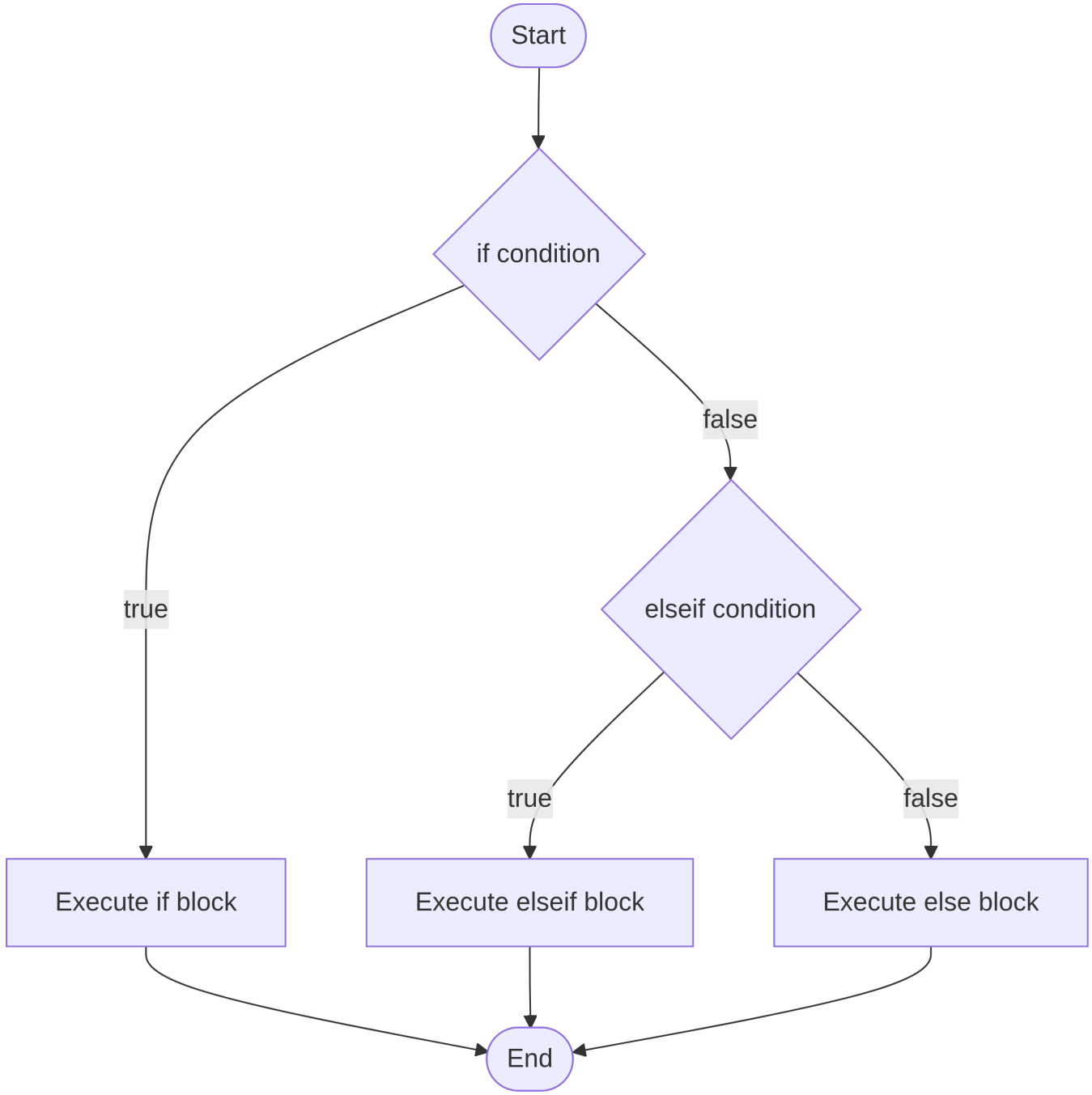
```
s1="some text"
s2="some text"
@show(s1==s2)
@show(s1===s2)
```

```
s1 == s2 = true
s1 === s2 = true
```

```
true
```

because **Strings are immutable**. This enables the compiler to **optimise memory usage by storing the String constant `"some text"` only once in memory!**

3.3 Conditional execution (`if-elseif-else`)




```
if 2 < 3 && 2 > 3
   println("The world has gone crazy")
end
```

```
if 2 < 3 || 2 > 3
   println("The world makes sense")
end
```

```
The world makes sense
```

```
x = 25.3
if x < 30
   println("It is less than 30")
else
   println("It is greater or equal to 30")
end
```

```
It is less than 30
```

```
x = 25.3
if x < 20
   println("It is less than 20")
elseif x < 30
   println("It is less than 30 but not less than 20")
else
   println("It is greater or equal to 30")
end
```

```
It is less than 30 but not less than 20
```

Let's use it to compute an absolute value.

```
x = -3 # some input
if x < 0
   println(-x)
else
   println(x)
end
```

```
3
```

3.4 Compact ways of formulating if clauses

Use either the **ternary operator** `condition ? dothisiftrue : dothisiffalse`

```
x = -3 # some input
absx = x < 0 ? -x : x
```

```
3
```

is the equivalten to

```
x = -3 # some input
if x<0
   absx=-x
else
   absx=x
end
```

```
3
```

or using logical AND (`&&`), making use of Julia's **lazy evalution of logical operators**,

```
x = -3 # some input
x < 0 && println("This is a negative number!")
```

```
This is a negative number!
```

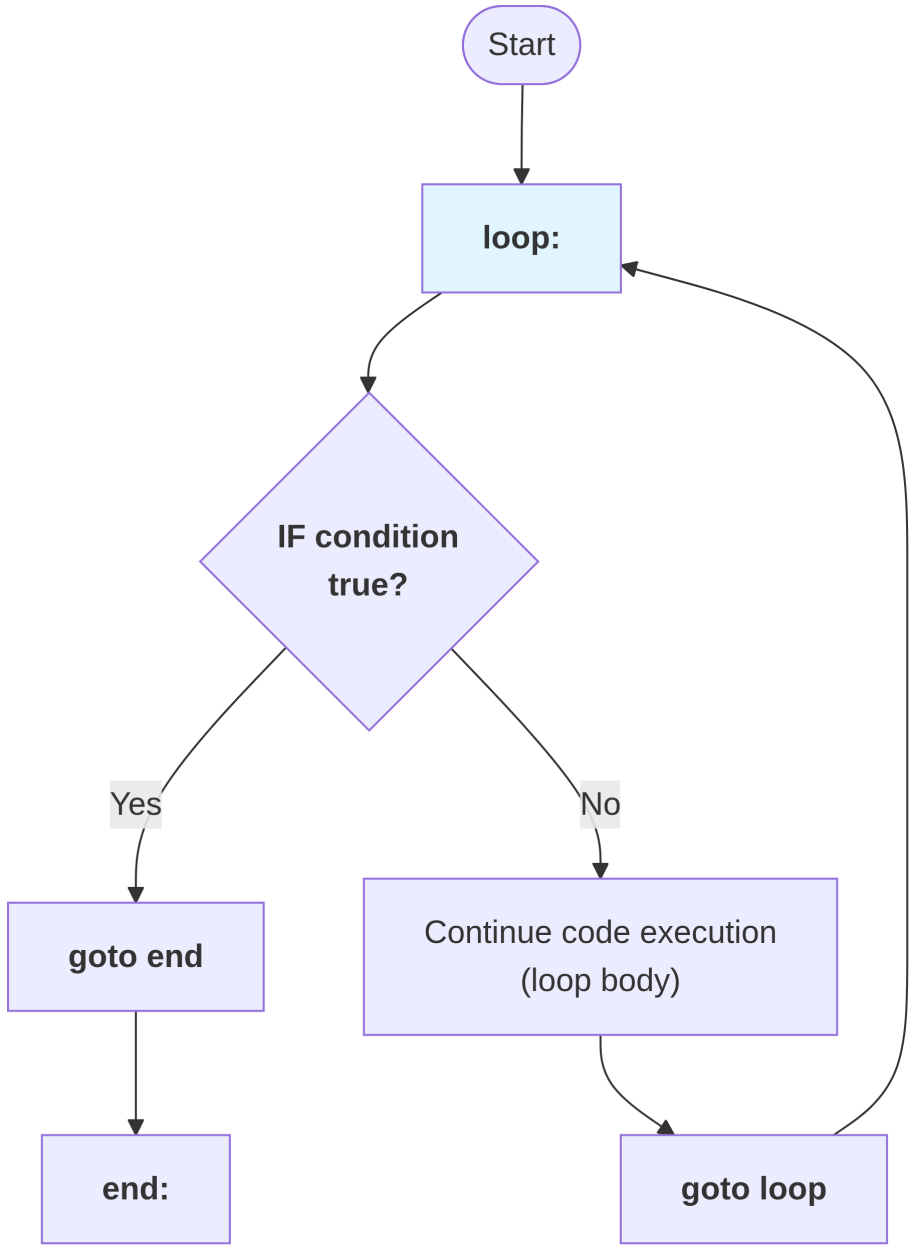
Here, the second condition for the AND clause `&&` is only evaluated if the first condition evaluates to true. Effectively this behaves like a **single-line if statement**.

4 Loops

You might want to execute a specific section of your code more than once, for example for a certain number of times, or until a condition is reached. The section of a program which is structured in this way is called a **loop**.

In principle, a loop can be implemented by combining an

1. if-statement which checks whether the loop should be executed one more time or not, and a
2. goto statement by which program execution jumps back to the beginning of the loop section.

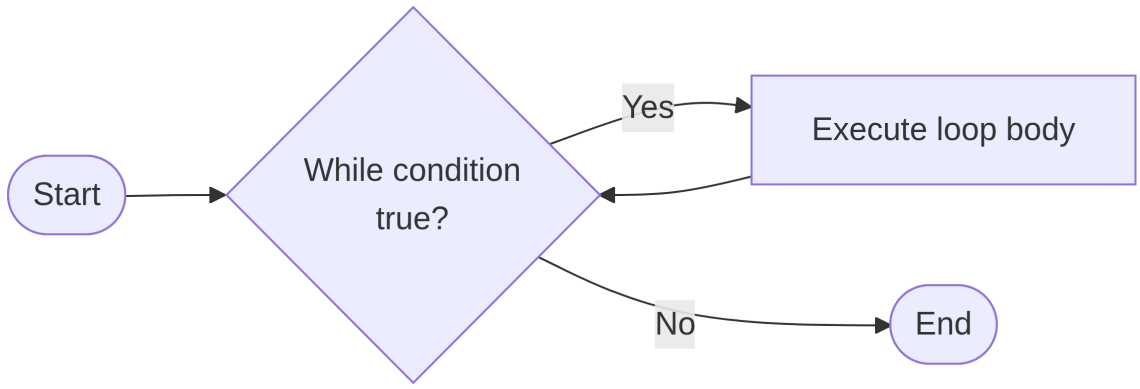


In many programming languages, particularly older ones, such `goto` statements exist, but they make code error-prone as well as hard to read and to maintain. For that reason more modern programming languages either don't include any type of `goto` command, or just in the form of macros which is the case for Julia.

It is **pretty much always better to implement loops through dedicated control structures,**

namely `for` loops and `while` loops.

4.1 While loops



In while loops, the loop section is written as a `while block` with the termination condition (actually it should be rather called continuation condition) at the BEGINNING of the loop section. Any initialisation has to happen before the `while` loop.

```
i = 1
while i   3   # do the following as long as i is smaller or equal than 3
   global i
   println(i)
   i = i + 1
end
```

```
1
2
3
```

In particular, if the continuation condition is never satisfied, the loop will not be executed, not even once.

```
i = 5
while i   3   # do the following as long as i is smaller or equal than 3
   global i
   println(i)
   i = i + 1
end
```

On the other hand, once can also write infinite loops which by themself won't terminante, unless the user presses `Ctrl-C` (or kills the program by some other method).

```
i=0
while true   # do it forever!
   global i
   println(i)
   i = i + 1
end
```

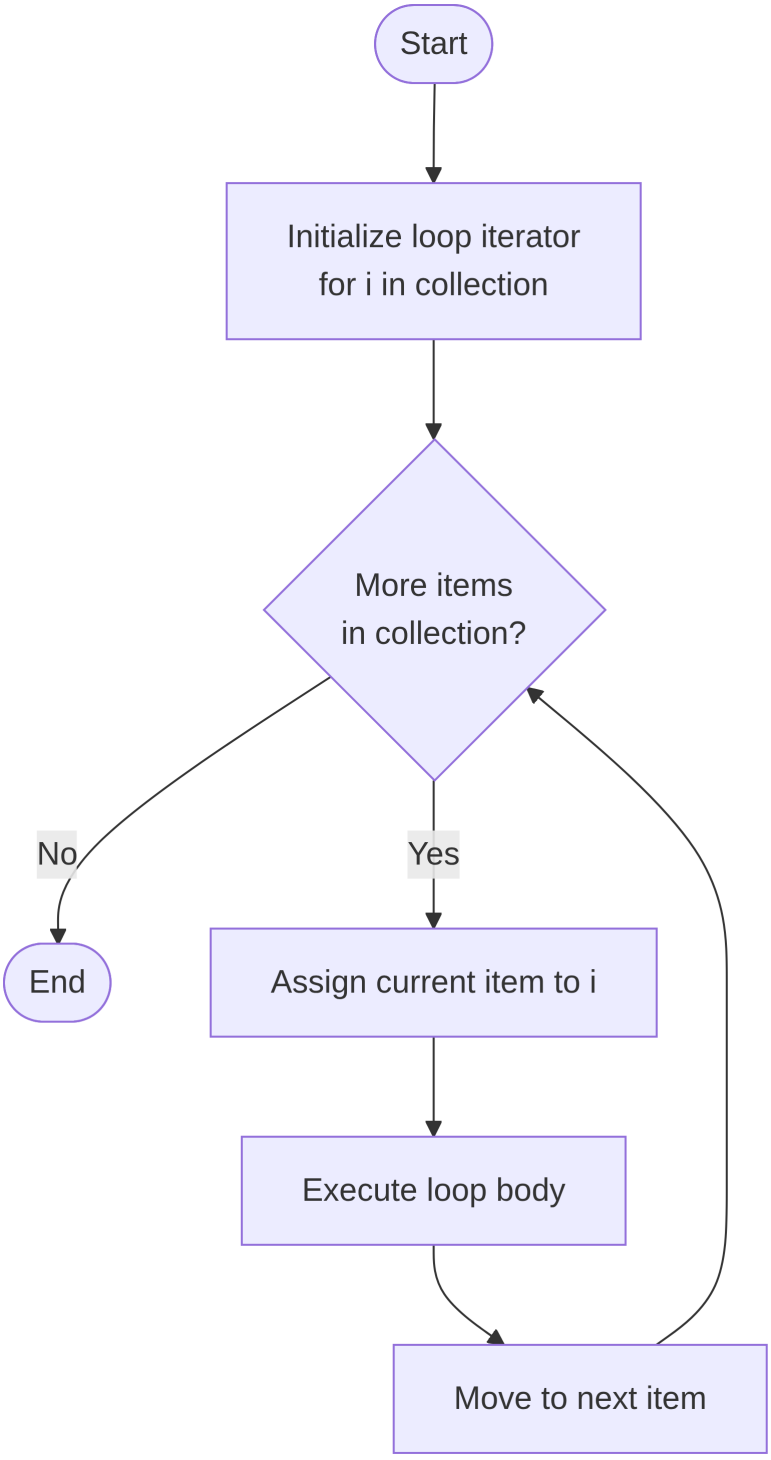
As an example, where we compute the beginning of the hailstone sequence: If a number is even, half it, if it is odd, multiply by 3 and add 1. Stop when you reach 1.

```
n = 7
while n != 1
   #global n
   print(n, ", ")
```

```
   if n % 2 == 0 # modulo 2...testing of n is even
     n = n ÷ 2 # \div + [TAB] Integer division
   else
     n = 3n + 1
   end
end
println(n)
```

```
7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
```

4.2 For loops



In `for` loops, the loop section is written as a `for...end` block with a counter and an ordered index set in the `for` line at the beginning of the block.

The `for` line serves three functions:

1. Initialise the counter

2. Check whether the end of the index set is reached and if so, leave the loop
3. Otherwise update (e.g. increase) the counter after every iteration

```
for i in 2:5
   println("hello nr ", i)
end
```

```
hello nr 2
hello nr 3
hello nr 4
hello nr 5
```

```
for i   1:3 # \in + [TAB]
   println("hello nr ", i)
end
```

```
hello nr 1
hello nr 2
hello nr 3
```

If the counter is not used in the loop, then _ can be used as place holder:

```
for _   1:3 # \in + [TAB]
   println("hello")
end
```

```
hello
hello
hello
```

A more general way of writing thisis

```
for i in eachindex(1:3)
   println("hello nr ", i)
end
```

```
hello nr 1
hello nr 2
hello nr 3
```

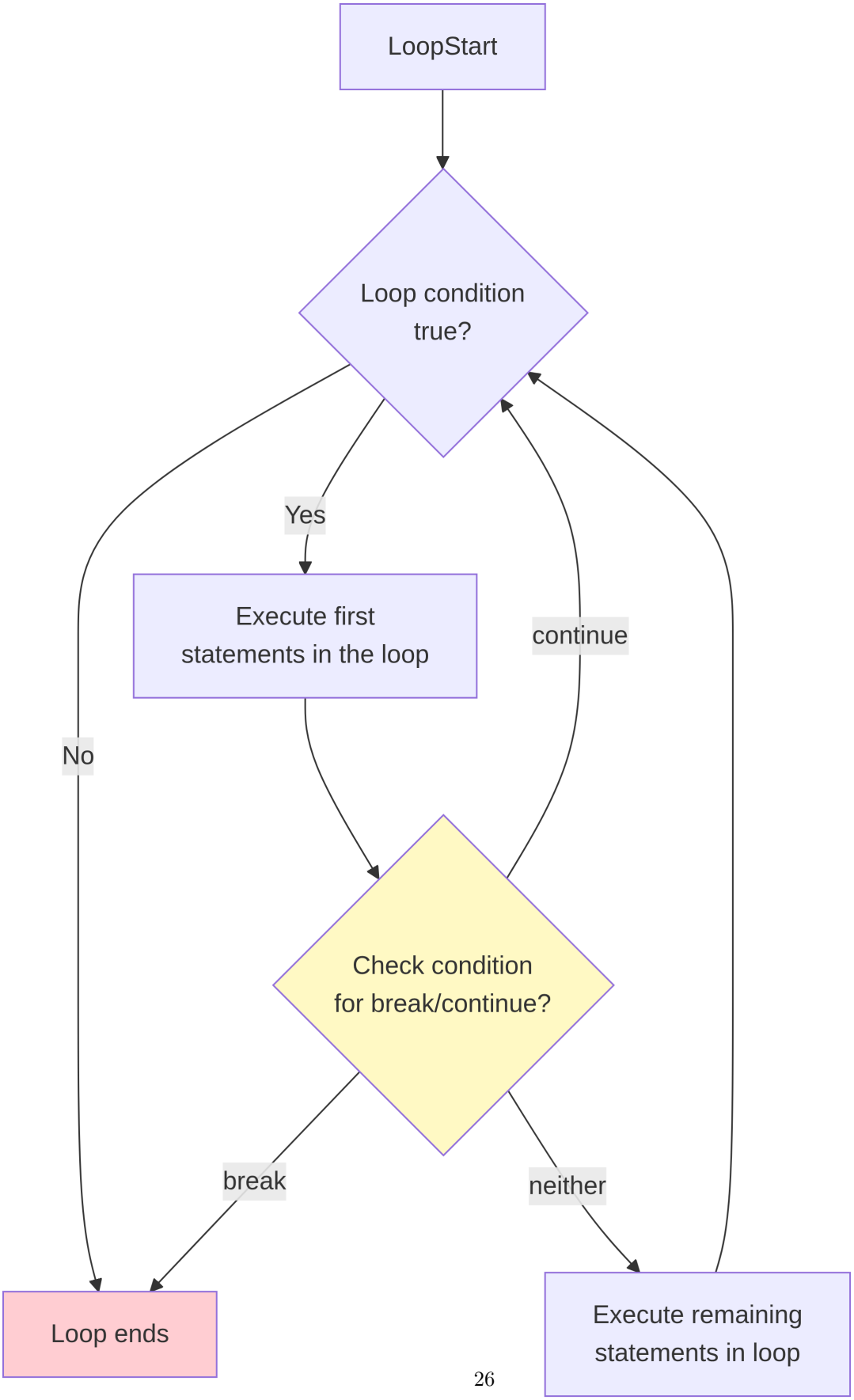
For example, summation of all elements in a finite sequence:

```
total = 0
max_val = 10
for i in 1:max_val
   global total # This line can be ignored by now
   total = total + i
end
println("The total is: $total")
println("And using a formula: ", max_val*(max_val+1)/2)
```

```
The total is: 55
And using a formula: 55.0
```

Notice the above `55` vs. `55.0`. This is because of division which converts an integer to a float.

4.3 `break` and `continue`



The archaic form or writing loops using `if` and some sort of `goto` command is arguably more flexible, and sometimes this type of flexibility is very helpful. For that reason, many programming languages such as Julia implement the commands `continue` and `break` by which the loop execution can be controlled from whithin the loop.

Using the command `continue` within a loop, code executation **continues at the start of the loop**.

```
for i in 1:13
   if i<10
      continue
   else
      println("hello ", i)
   end
end
```

```
hello 10
hello 11
hello 12
hello 13
```

Using the command `break` within a loop, **loops can be left at any time**.

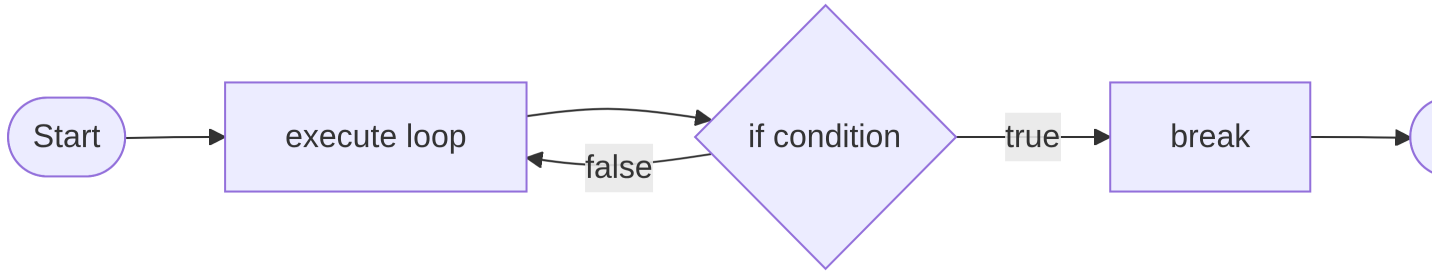
```
for i  1:13 # \in + [TAB]
   if i<5
      println("hello $i")
   else
      break
   end
end
```

```
hello 1
hello 2
hello 3
hello 4
```

```
for i  1:13 # \in + [TAB]
   if i>10
      break
   else
      println("hello $i")
   end
end
```

```
hello 1
hello 2
hello 3
hello 4
hello 5
hello 6
hello 7
hello 8
hello 9
hello 10
```

4.4 Howto implement do-while loops in Julia



In do-while loops, the continuation condition is at the end of the loop. Therefore, the loop is executed at least once before the continuation condition is evaluated! No explicit syntax for do-while loops exists in Julia, since this particular behavior can be reproduced by **combining an infinite `while` loop with `if` and `break` commands**:

```
i=0
while true
    # do stuff
    println(i)
    i = i + 1
    if i>5
      break
    end
    # shorter:  i>5 || break
end
```

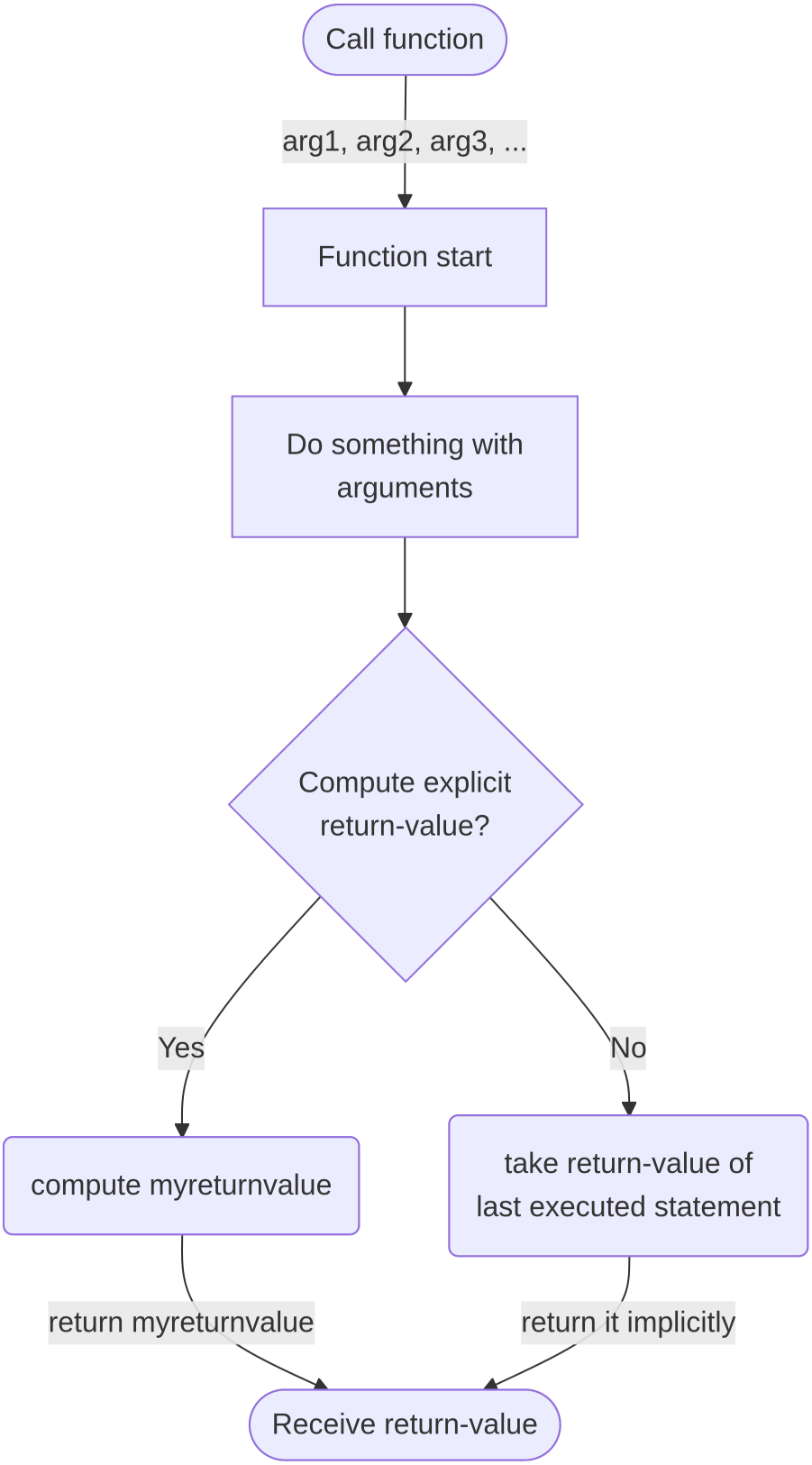
```
0
1
2
3
4
5
```

This is to demonstrate that with a do-while loop, the loop section is always executed at least once, even if the termination condition is `true` from the start:

```
i=10
while true
    # do stuff
    println(i)
    i = i + 1
    if i>5
      break
    end
    # shorter:  i>5 || break
end
```

```
10
```

5 Creating your own functions



We already had logic for an absolute value function (of real values). Now let's make a callable function out of it:

```
function my_abs(x)
   if x < 0
      return -x
   else
      return x
   end
end
```

```
my_abs (generic function with 1 method)
```

Note that using the command `return` the function terminates and the argument of the `return` command becomes the **return value** of the function. We can also implement this as,

```
function my_abs(x)
   if x < 0
      return -x
   end
   return x
end
```

```
my_abs (generic function with 1 method)
```

Or even,

```
function my_abs(x)
   if x < 0
      return -x
   end
   x
end
```

```
my_abs (generic function with 1 method)
```

which uses that **functions by default return the output of the last command.**

```
@show my_abs(-3.5);
@show my_abs(2.3);
@show my_abs(0);
```

```
my_abs(-3.5) = 3.5
my_abs(2.3) = 2.3
my_abs(0) = 0
```

Functions don't have to have an argument or a return value:

```
function print_my_details()
   println("Name: Jacob")
   println("Occupation: diesel mechanic")
   return nothing
end

print_my_details()
```

```
Name: Jacob
Occupation: diesel mechanic
```

5.1 Using mathematical notation to define functions

When you can implement a function in one line, you can avoid using the `function` keyword and instead use **mathematical notation for defining functions**:

Instead of

```
function times_1(x)
   return 2x
end
times_1(343)
```

```
686
```

you could also just define a function as follows:

```
times_2(x) = 2x
```

```
times_2 (generic function with 1 method)
```

These two function definitions are actually perfectly equivalent. Once functions are defined, you can for example compose them as follows:

```
@show times_2(times_2(10)) #this uses the function twice.
@show (times_2  times_2)(10) # \circ...alternative way of writing function composition
```

```
times_2(times_2(10)) = 40
(times_2  times_2)(10) = 40
```

```
40
```

Note that on many occasion when there is no ambiguity, Julia permits omitting the multiplication symbol `*` in order to mimic mathematical notation even more closely.

6 Example: Collatz conjecture, hailstone sequence

Let's formulate the computation of the first 10 elements of the hailstone sequence as a function:

```
#here, we "wrap" the code we had before in a function
function hailstone(n_start)
   n = n_start
   while n != 1
      print(n, ", ")
      if n % 2 == 0 #is n even?
         n = n ÷ 2
      else
         n = 3n + 1
      end
   end
   println(n)
   return nothing
end

hailstone(7)
```

```
7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
```

Note that the function actually only outputs the elements of the hailstone sequence, but it doesn't return them.

```
println("Hailstone sequences: ")
for n in 1:10
```

```
   hailstone(n)
end
```

```
Hailstone sequences:
1
2, 1
3, 10, 5, 16, 8, 4, 2, 1
4, 2, 1
5, 16, 8, 4, 2, 1
6, 3, 10, 5, 16, 8, 4, 2, 1
7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
8, 4, 2, 1
9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
10, 5, 16, 8, 4, 2, 1
```

The Collatz Conjecture says that for every starting value the sequence eventually hits 1. Let's try to disprove it by seeing if this program gets stuck.

First I extend the definition of the function hailstone above, so it returns the length of the hailstone sequence rather than printing it.

```
function hailstone_length(n_start)
   n = n_start
   len = 1
   while n != 1
      len += 1 # short notation for n = n + 1
      # print(n, ", ")
      if n % 2 == 0
         n = n ÷ 2
      else
         n = 3n + 1
      end
   end
   #println(n)
   #println()
   return len
end

hailstone_length(7)
```

```
17
```

```
for n in 1:10^3
   if n>1
      print(", ")
   end
   print(hailstone_length(n))
end
println()
```

```
1, 2, 8, 3, 6, 9, 17, 4, 20, 7, 15, 10, 10, 18, 18, 5, 13, 21, 21, 8, 8, 16, 16, 11, 24, 11, 112, 19, 19
```

It didn't get stuck (try even changing `10^3` to `10^6`). If the loop would "get stuck", it wouldn't terminate unless we press `Ctrl-C`!

Now let's see what was the longest sequence.

```
length_of_longest = 0
n_of_longest = 0
for n in 1:10^3
```

```
   global length_of_longest, n_of_longest
   seq_len = hailstone_length(n)
   if seq_len > length_of_longest
      length_of_longest = seq_len
      n_of_longest = n
   end
end

println("The longest hailstone sequence is of length $length_of_longest when you start at
$n_of_longest.")
```

```
The longest hailstone sequence is of length 179 when you start at 871.
```

7 Quiz practise questions

1. Write a function called `fizz_buzz` that accepts a number and
   - If the number is divisible by 3 but not 5, prints "Fizz" (without quotes)
   - If the number is divisible by 5 but not 3, prints "Buzz" (without quotes)
   - If the number is divisible by 3 and 5, print "FizzBuzz" (without quotes)
   - Otherwise, print the number itself
2. Write a Julia function which returns the number of spaces in the String provided as argument to the function. Test the function using the argument "Julia uses the Multiple-Dispatch paradigm!"
3. Rewrite the following piece of code

   ```
   cn=1
   println(cn)
   for i=1:5
      cn=sin(cn)
      println(cn)
   end
   ```

   using a
   - `while` loop and
   - using and `if` clause to mimic a `do-while-loop`. Aim at avoiding redundant statements, yet, both versions of the script should generate the exactly same output as the code above.
4. Write a short script which prints the first 30 elements of the recursion relation $c_{n+1} = 4c_n - 3c_{n-1}$ starting with $c_1 = 0$ and $c_2 = 1$. At the end the script should also print the sum $c_1 + c_2 + ...c_{30}$.
5. Sum of squares: Write a function called `sum_of_squares` that accepts the natural number n and returns the sum $1^2 + 2^2 + \cdots + (n-1)^2 + n^2$.
6. What is the type of each of the following Julia expressions?

   ```
   3
   3.0
   '3'
   3>0
   "3"
   ```

7. Fibonacci Sequence: Calculate the first element of the Fibonacci sequence $a_0 = 1, a_1 = 1, a_{n+2} = a_{n+1} + a_n$ greater than 1000, and store it in a variable called `an`.
8. Consider the Julia code:

```
a = [1, 4, 9]
s = a[1]*a[2]*a[3]
```

What is the type and value of `s`?

9. You want to write a function `my_minimum` that gets a vector of numbers and returns the minimal value in the array. Do not use the `minimum` in-built function as part of your answer.

10. You want to write a Julia function `is_sub_str` which accepts two Strings `str` and `substr`, and returns `true` is `substr` is a sub-string of `str` (do not use any library functions, but only the tools developed in this Unit 1).

11. Which of the following correctly defines a Julia function that returns the square of its argument?

    A.

```
square(x) = x*x
```

    B.

```
function square(x)
   x*x
end
```

    C. Both A and B

    D. Neither

12. Consider the Julia function, `tamid_nahon` which gets two boolean values as inputs, `a`, and `b`.

```
function tamid_nahon(a, b)
   return !(a && b) == !a || !b
end
```

    For what combinations of `a` and `b` is the return value `true`? For what is it `false`?

13. Write a function `round_to_nearest_int` that takes a floating-point number and returns it as an integer type, rounded to the nearest integer. Test it with values like `3.7`, `-2.3`, and `5.5`.

14. Without running the code, predict the output of the following expression and explain why:

```
result = true || false && false
```

    What is the value of `result`? Where could you add parenthesis to change the result?

15. Write a Julia expression which computes the exponential of pi/2 and stores the result in a variable named `res`. What is the type of this variable?

16. Which operator tests whether two values are numerically equal?

    A. `=`

    B. `==`

    C. `:=`

    D. `===`

17. Write a script which asks the user to input a loginname and then prints a statement like the following: "Your loginname includes xxx integer numbers between 0 and 9".

18. Write a function substrings which takes the String `str` as argument and returns three substrings, one which starts like `str`, one which ends like `str` and one take from somewhere in the middle. The lengths of these substrings should be random every time the function is executed.

19. Write a function `count_down` that takes a positive integer `n` and prints all numbers from `n` down to 1, each on a separate line, using a while loop. After printing all numbers, it should print "Liftoff!".

20. What is printed?

```
A = [1, 2, 3]
B = A
B[2] = 100

println(A)
```

21. Write a function `print_odd_numbers` that takes an integer `n` and prints all odd numbers from 1 to `n` (inclusive). Use a for loop and the `continue` statement to skip even numbers.

22. Consider the Julia function

```
function myfun2(a)
   i=10
   while i<115
      if i >= a
         break
      end
      println(i)
      i = i+1
   end
   return i
end
```

A. What is the output written by `println` when calling `myfun2(10)`?

B. What is the return value of `myfun2(11)`?

23. What is printed?

```
str1 = "hello"
str2 = str1
str1 = "world"
println(str2)
```

Would the behavior be different if we used vectors instead of strings?

24. Consider the following code:

```
vec1 = [1, 2, 3]
vec2 = vec1
vec3 = [1, 2, 3]
```

Which of the following comparisons will return `true` and which will return `false`? Explain why.

- `vec1 == vec2`
- `vec1 === vec2`
- `vec1 == vec3`
- `vec1 === vec3`

25. Rewrite the following if-else statement using the ternary operator:

```
if x >= 0
   result = sqrt(x)
else
   result = 0.0
end
```

26. Write two functions: `double(x)` that returns `2*x` and `add_five(x)` that returns `x + 5`. Then use function composition to create a single expression that first doubles a number and then adds five to the result.

27. Given `name = "Julia"`, which expression produces `Hello Julia!`?

    A. `"Hello name!"`

    B. `"Hello $name!"`

    C. `'Hello $name!'`

    D. `println(name)`

28. Write a function `digit_sum` that takes a positive integer `n` and returns the sum of its digits. For example, `digit_sum(123)` should return `6` (because $1 + 2 + 3 = 6$). Hint: Use the modulo operator `%` and integer division `÷`.

29. Write a function `check_brackets` that takes a string containing a mathematical expression and checks whether opening and closing brackets are balanced (i.e. for every opening bracked there is a closing one).

30. What is printed?

```julia
x = 5
if x > 3
   println("A")
elseif x > 5
   println("B")
else
   println("C")
end
```

31. What is printed?

```julia
s = 0
for i in 1:4
   s += i
end
println(s)
```

32. Write a Julia function `is_even(n)` that returns `true` if `n` is even and `false` otherwise.