Pseudocode I

- Our algorithms written in pseudocode will consist of two type of instructions:
  - standard instructions (assignment, conditional, repetitive, etc.)
  - non-standard instructions (written in plain English to describe parts of the algorithm that are not developed yet). These non-standard instructions will start with @.

Pseudocode II

- One line comments in the code will be denoted by //
- For reading data we will use the standard instruction **read**
- For printing data we will us the standard instruction **print**
- For assignment we will use $\leftarrow$
- For testing the equality of two variables we will use $=$

Pseudocode III

- Conditional instruction will be written in the following way (the *else* part can be missing):

```
if condition then
    @instructions
else
    @instructions
end-if
```

Pseudocode IV

- The *for* loop (loop with a known number of steps) will be written in the following way:

```
for i ← init, final, step execute
    @instructions
end-for
```

- *init* - represents the initial value for variable i
- *final* - represents the final value for variable i
- *step* - is the value added to i at the end of each iteration. *step* can be missing, in this case it is considered to be 1.

Pseudocode V

- The *while* loop (loop with an unknown number of steps) will be written in the following way:

while condition **execute**
  @instructions
end-while

Pseudocode VI

- Subalgorithms (subprograms that do not return a value) will be written in the following way:

subalgorithm name(formal parameter list) **is:**
  @instructions - subalgorithm body
end-subalgorithm

- The subalgorithm can be called as:

name (actual parameter list)

Pseudocode VII

- Functions (subprograms that return a value) will be written in the following way:

function name (formal parameter list) **is:**
  @instructions - function body
  name ← v *//syntax used to return the value v*
end-function

- The function can be called as:

result ← name (actual parameter list)

Pseudocode VIII

- If we want to define a variable i of type Integer, we will write: i : *Integer*
- If we want to define an array a, having elements of type T, we will write: $a : T[]$
  - If we know the size of the array, we will use: $a : T[Nr]$ - indexing is done from 1 to Nr
  - If we do not know the size of the array, we will use: $a : T[]$ - indexing is done from 1

Pseudocode IX

- A struct (record) will be defined as:

```
Array:
    n: Integer
    elems: T[]
```

- The above struct consists of 2 fields: *n* of type Integer and an array of elements of type T called *elems*
- Having a variable *var* of type Array, we can access the fields using . (dot):
  - var.n
  - var.elems
  - var.elems[i] - the i-th element from the array

Pseudocode X

- For denoting pointers (variables whose value is a memory address) we will use $\uparrow$:
  - p: $\uparrow$ Integer - p is a variable whose value is the address of a memory location where an Integer value is stored.
  - The value from the address denoted by p is accessed using $[p]$
- Allocation and de-allocation operations will be denoted by:
  - allocate(p)
  - free(p)
- We will use the special value NIL to denote an invalid address

Specifications I

- An operation will be specified in the following way:
  - **pre:** - the preconditions of the operation
  - **post:** - the postconditions of the operation
  - **throws:** - exceptions thrown (optional - not every operation can throw an exception)
- When using the name of a parameter in the specification we actually mean its value.
- Having a parameter i of type T, we will denote by $i \in T$ the condition that the value of variable i belongs to the domain of type T.

Specifications II

- The value of a parameter can be changed during the execution of a function/subalgorithm. To denote the difference between the value before and after execution, we will use the ' (apostrophe).

- For example, the specification of an operation *decrement*, that decrements the value of a parameter x (x : *Integer*) will be:
  - **pre:** x $\in$ Integer
  - **post:** x' = x - 1

Generic Data Types I

- We will consider that the elements of a container ADT are of a generic type: *TElem*
- The interface of the *TElem* contains the following operations:
  - assignment $(e_1 \leftarrow e_2)$
    - **pre:** $e_1, e_2 \in TElem$
    - **post:** $e_1' = e_2$
  - equality test $(e_1 = e_2)$
    - **pre:** $e_1, e_2 \in TElem$
    - **post:**

$$equal \leftarrow \begin{cases} True, & \text{if } e_1 \text{ } equals \text{ } e_2 \\ False, & \text{otherwise} \end{cases}$$

Generic Data Types II

- When the values of a data type can be compared or ordered based on a relation, we will use the generic type: *TComp*.
- Besides the operations from TElem, TComp has an extra operation that compares two elements:
  - compare(e_1, e_2)
    - **pre:** $e_1, e_2 \in TComp$
    - **post:**

$$compare \leftarrow \begin{cases} true & \text{if } e_1 <= e_2 \\ false & \text{if } e_1 > e_2 \end{cases}$$

- For simplicity, sometimes instead of calling the *compare* function, we will use the notations $e_1 \leq e_2$, $e_1 = e_2$, $e_1 \geq e_2$