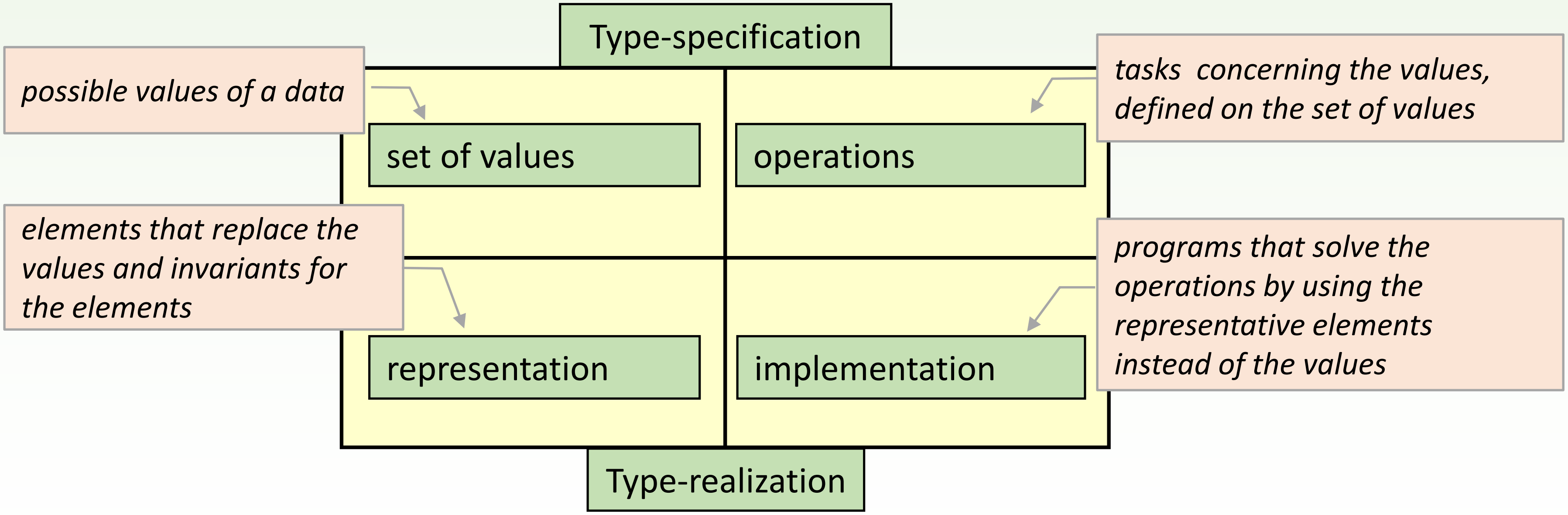
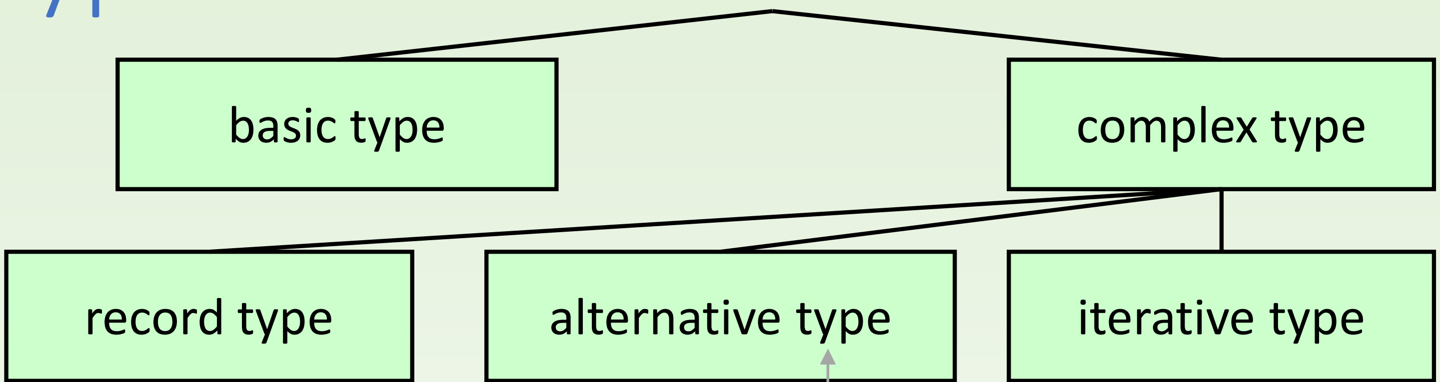
Collections, enumerators, basic algorithms (algorithmic patterns)

Datatype

- Type of a data (specifically an object) is defined by the set of its values and its operations. It is called specifiation.
- Type realization shows how the values could be represented and how programs solve or implement the operations.



Type structure



a value is represented by a group of values of other types

$T = rec(s_1:T_1, \ldots, s_n:T_n)$
i^{th} component of t:T *is* $t.s_i$

a value is represented by a finite collection of values of another type, the elements of the collection are of the same type

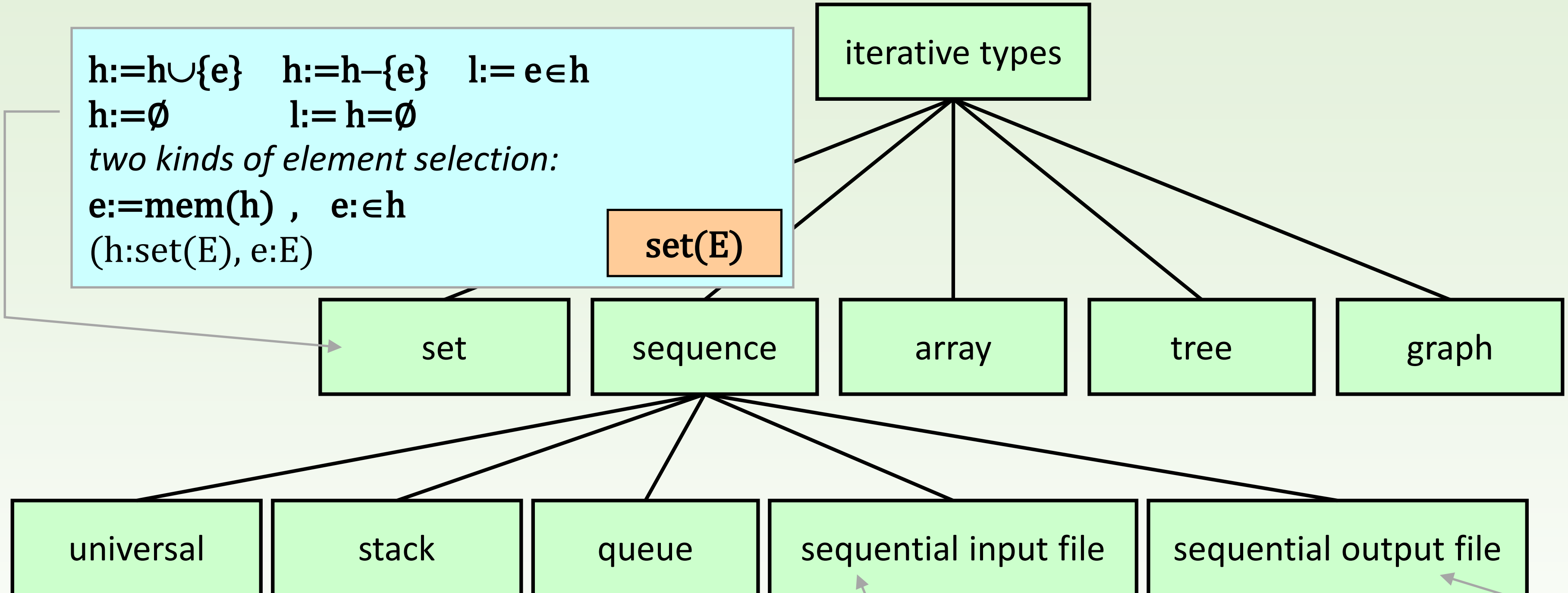
T = it(E)

relation of elements that represent the values

a value is represented by one of the values of other types

$T = alt(s_1:T_1, \ldots, s_n:T_n)$
if type of t:T *is* T_i, *then* $t.s_i$ *is true*

Well-known iterative types



set(E)

$h:=h\cup\{e\}$ $h:=h-\{e\}$ $l:=e\in h$

$h:=\emptyset$ $l:=h=\emptyset$

two kinds of element selection:

$e:=mem(h)$, $e:\in h$

(h:set(E), e:E)

infile(E)

st, e, f : read

f : *sequential input file* (**infile(E)**)

e : *the read element* (**E**)

st: *indicates the success of reading* (**Status**)

Status ={norm, abnorm}

sf, df, f := read(f)

if f=<> **then** sf := abnorm

else sf:=norm : df := f_1 : f := $<f_2, \ldots, f_{|f|}>$

outfile(E)

f:=< >

f : write(s)

f : *sequential output file* (**outfile(E)**)

s: sequence to write (**E***)

f := write(f, s)

f := f ⊕ s

Processing a collection

- Collection (container, collection, iteration) is an object, capable of storing elements. It provides operations for archiving and searching elements.
  - Like complex types, especially the iterations: set, sequence (stack, queue, file), array, tree, graph.
  - There are so-called virtual collections, too, the elements of which do not have to be stored: e.g. items of an integer-type interval or prime divisors of a natural number.
- Processing a collection means processing its elements.
  - Find the biggest item of a set.
  - How many negatives are in a sequence of numbers?
  - Traverse backwards every second item of an [*m* .. *n*] interval.
  - Sum the prime divisors of a *n* natural number.

Enumeration

- Enumeration of the *E*-type elements of a collection can be considered as a sequence in set E^*. The operations of the traversal are the following:
  - *first()* : selects the first item of the enumeration, it actually starts the enumeration
  - *next()* : selects the next item of the enumeration
  - *l* := *end()* (l:$\mathbb{L}$) : shows if the enumeration has ended
  - *e*:= *current()* (e:E): gets the current item of the enumeration

States of the enumeration

- An enumeration has different states (*pre-start*, *in process*, *finished*): the operations are only reasonable in certain states (otherwise, their effect is not defined).
- The processing algorithm guarantuees that the operations are executed only (in that state) when they are reasonable.

t : enor(E)

t.first()

¬t.end()

Process(t.current())

t.next()

```
for(t.first(); !t.end(); t.next())
{
    process(t.current());
}
```

$e \in t$

Process(e)

foreach (forall) loop

```
for( auto e : t )
{
    process(e);
}
```

Enumeration with object

- Enumeration is done by a distinct object separated from the collection. There can be more enumerator objects for one collection.
- Type of the enumerator object is denoted by *enor*(*E*).
- Realization of an enumerator object depends on the type of the collection.
    - As the enumerator object has to know the traversed collection, its representation contains a reference of the collection.
    - Implementations of the operations usually need auxiliary data.
- It is worthy to create the enumerator by a method of the collection so that the collection is aware of being traversed.

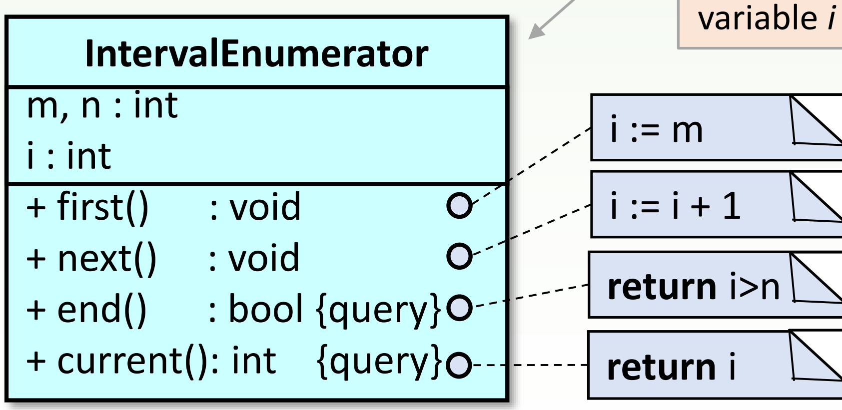
Classic enumerator of an interval

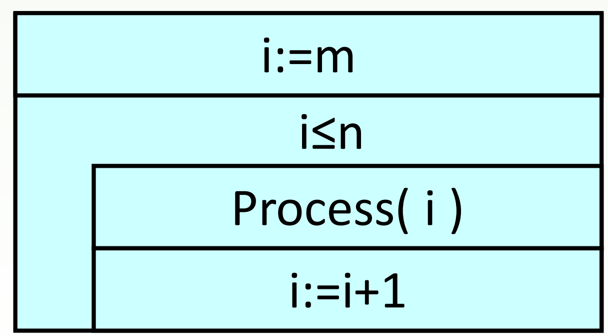
Enumeration of integers in an interval in ascending order.

enor($\mathbb{Z}$)

$\mathbb{Z}^*$	first()	next()	l:= end()	e:= current()
m , n : $\mathbb{Z}$ i : $\mathbb{Z}$	i:=m	i:=i+1	l:= i>n l:$\mathbb{L}$	e:=i e:$\mathbb{Z}$

Usually, this class is not instantiated, the enumerator is variable *i* itself.

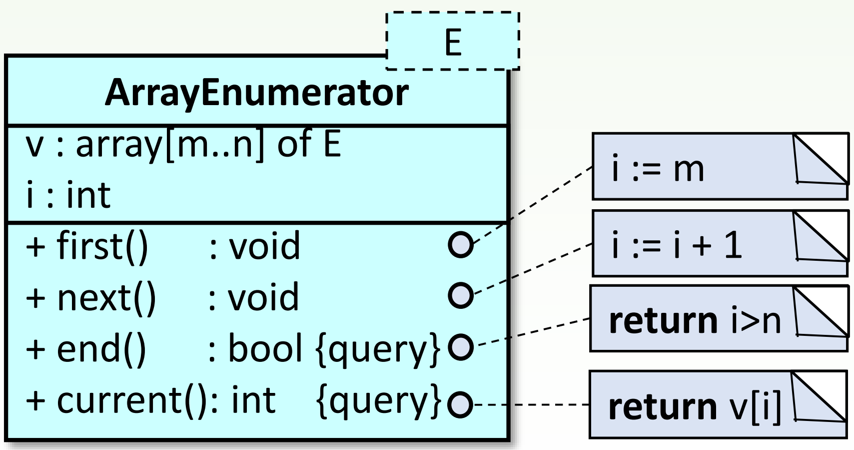


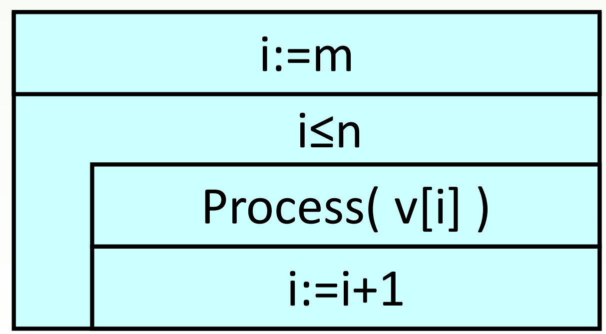


Classic enumerator of a vector

Enumeration of the items of a vector containing values from E,
from the beginning to the end

enor(*E*)				
E*	first()	next()	l:= end()	e:= current()
v : $E^{m..n}$ i : $\mathbb{Z}$	i:=m	i:=i+1	l:= i>n l:$\mathbb{L}$	e:=v[i] e:E

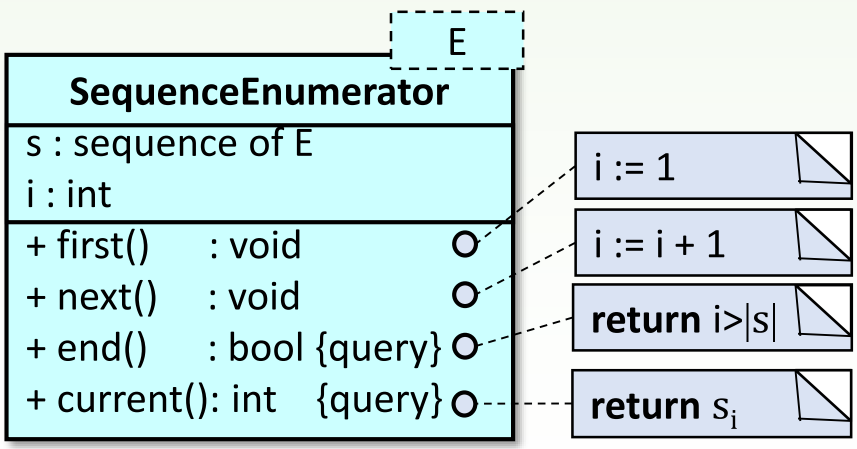


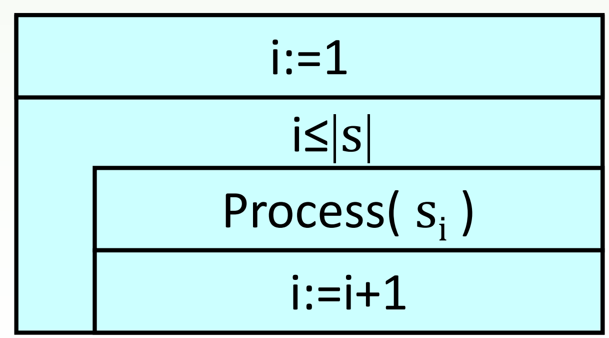


Classic enumerator of a sequence

Enumeration of a finite sequence of values from E, from the beginning to the end

enor(*E*)				
E*	first()	next()	l:= end()	e:= current()
s : E^* i : $\mathbb{Z}$	i:=1	i:=i+1	l:= i>|s| l:$\mathbb{L}$	e:= s_i e:E

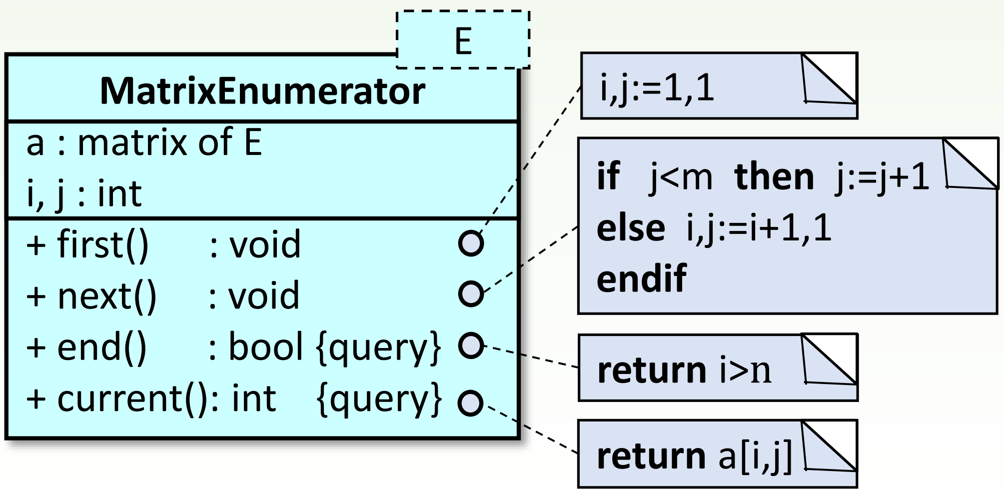


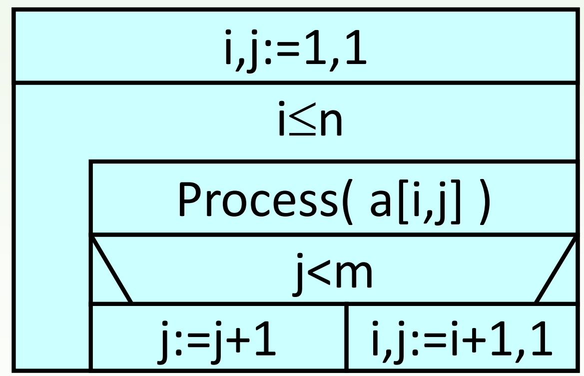


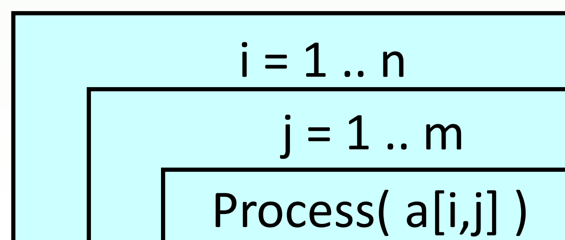
Row major enumerator of a matrix

Enumeration of the items of a matrix with values from E in row major order.

enor(E)				
E*	first()	next()	l:= end()	e:= current()
a : $E^{n\times m}$ i,j : $\mathbb{Z}$	i,j:=1,1	**if** j<m **then** j:=j+1 **else** i,j:=i+1,1	l:= i>n l:$\mathbb{L}$	e:= a[i,j] e:E





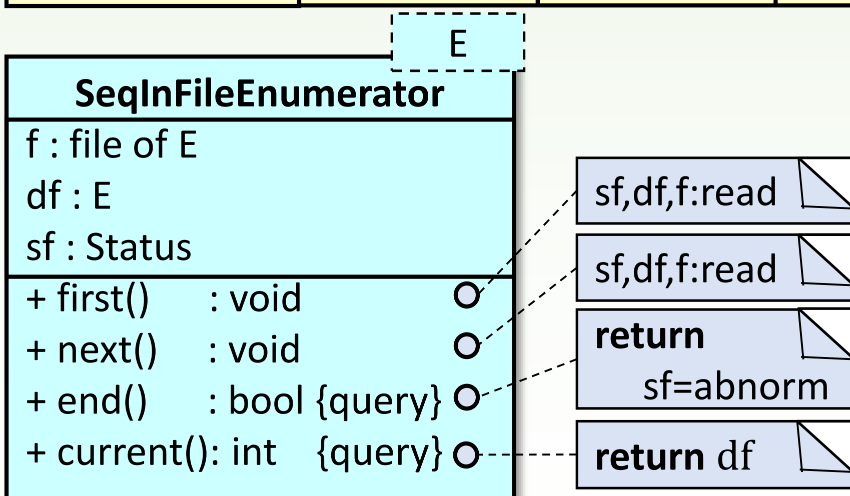


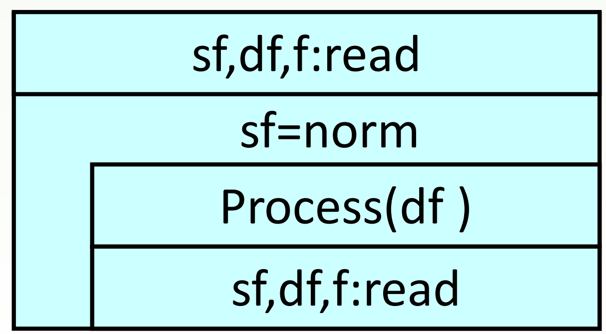
Enumerator of a sequential input file

Enumeration of the items of a sequential input file with values from E.

enor(*E*)

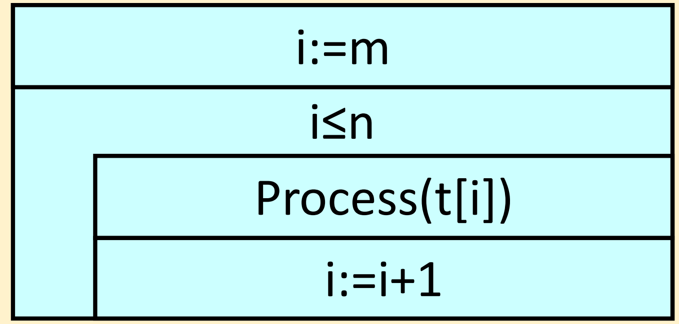
E*	first()	next()	l:= end()	e:= current()
f : infile(E) df : E sf : Status	sf,df,f:read	sf,df,f:read	l:= sf=abnorm l:$\mathbb{L}$	e:= df e:E



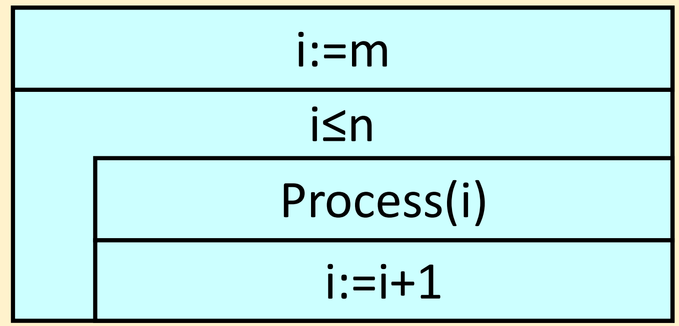


Generalization of the algorithmic patterns

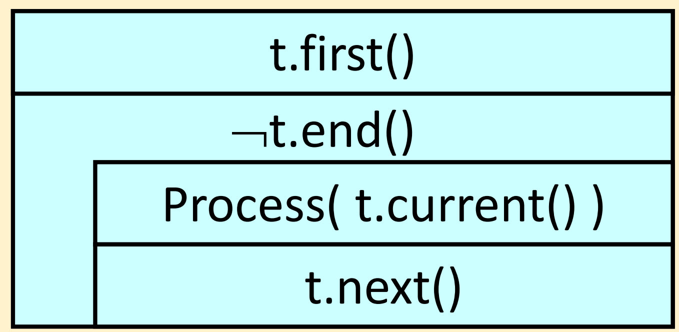
- Algorithmic patterns for arrays:
  - $t : E^{m..n}$ $(E^{1..n} = E^n)$
  - $f : E \to H$, $cond: E \to \mathbb{L}$



- Algorithmic patterns for functions defined on intervals:
  - $[m .. n]$
  - $f:[m .. n] \to H$, $cond: [m .. n] \to \mathbb{L}$



- Algorithmic patterns for enumerators:
  - $t : enor(E)$
  - $f : E \to H$, $cond: E \to \mathbb{L}$



Summation

Sum the values assigned to the elements of an enumeration.

A : t:enor(E), s:H

Pre : t = t'

Post: $s = \Sigma_{e \in t'} f(e)$

$\Sigma_{e \in t'} f(e) = (\ldots(f(e_1) + f(e_2)) + \ldots) + f(e_n)$, where $e_1, \ldots , e_n$ are the elements of enumeration *t'*

$f : E \to H$

$+ : H \times H \to H$

$0 \in H$ *with left neutral element*

Special case: conditional summation

$$\Sigma_{\substack{e \in t' \\ \text{cond}(e)}} g(e), \text{ so } f(e) = \begin{cases} g(e), & \text{if cond}(e) \\ 0, & \text{otherwise} \end{cases}$$

s := 0

t.first()

¬t.end()

s := s + f(t.current())

t.next()

Counting

Count the items with certain condition in an enumeration.

A : t:enor(E), c:$\mathbb{N}$

Pre : t = t'

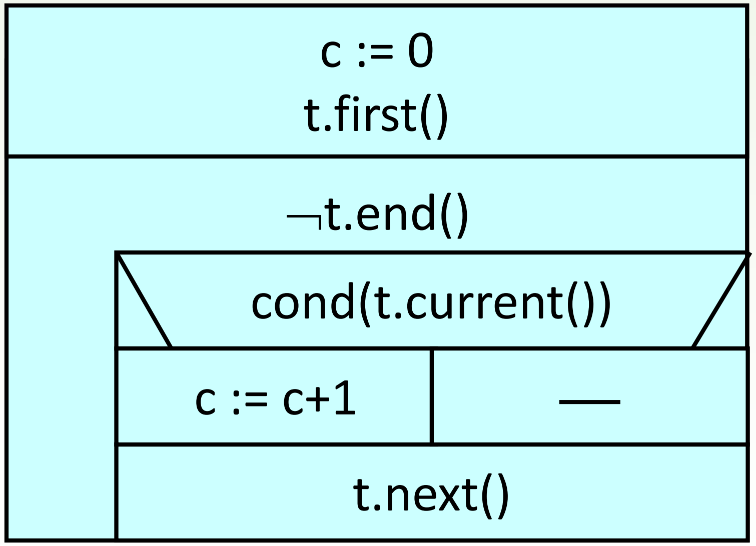
Post: $c = \sum_{\substack{e \in t' \\ \text{cond}(e)}} 1$

cond: $E \to \mathbb{L}$

Sum defined on the set of natural numbers

Counting is a special summation

$$\sum_{e \in t'} f(e), \text{ thus } f(e) = \begin{cases} 1, & \text{if cond}(e) \\ 0, & \text{otherwise} \end{cases}$$



Maximum search

Give the item of the highest value of an enumeration according to a given point of view.

A : t:enor(E), elem:E, max:H

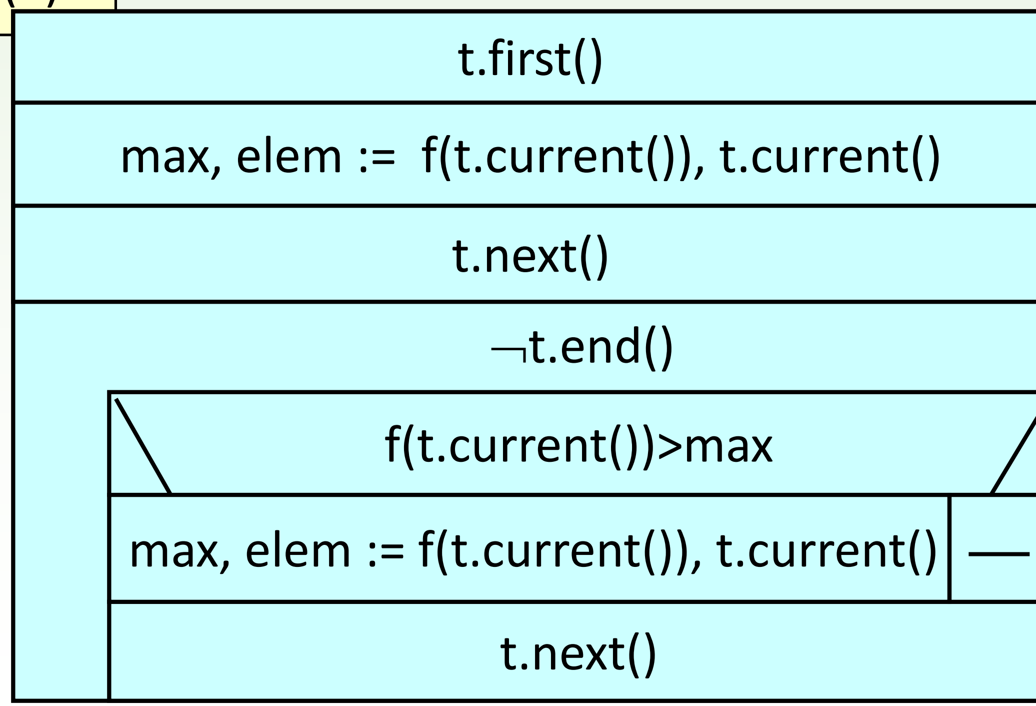
Pre : $t = t' \wedge |t|>0$)

Post: (max , elem)=$\mathbf{MAX}_{e \in t'}$ f(e)

f:E → H

items of set H *can be sorted*

$max = f(elem) = \mathbf{MAX}_{e \in t'} f(e) \wedge elem \in t'$



- MIN instead of MAX
- elem *can be skipped,* max *not*

Selection (success is sure)

Find the first item of a given condition in an enumeration if it is sure that the enumerator contains such element.

A: t:enor(E), elem:E

Pre: $t = t' \wedge \exists e \in t : cond(e)$

Post: (elem, t) = **SELECT** $_{e \in t'}$ cond(e)

cond: $E \to \mathbb{L}$

At the end of the selection, the state of the enumerator is still „*in process*", as it has remaining items

Searches the first item (it will be the *elem*) of enumerator *t'* for which the condition is satisfied.
Formally:
$cond(e_i) \wedge \forall_{k=1..i-1} \neg cond(e_k) \wedge elem = e_i$,
where e_1, e_2 , … are items of enumerator *t'*

t.first()
$\neg$cond(t.current())
t.next()
elem := t.current()

Linear search (success is unsure)

Find the first item of a given condition in an enumeration.

A : t:enor(E), l:$\mathbb{L}$, elem:E

Pre : t = t'

Post: (l, elem, t) = **SEARCH** $_{e\in t'}$ cond(e)

cond:E → $\mathbb{L}$

Enumeration of *t* only finishes if the search is unsuccessful, otherwise it remains in state „*in process*".

Searches the first item (it will be the *elem*) of enumerator *t'* for which the condition is satisfied.
If the search is successful, the value of *l* changes to true, otherwise it remains false.
Formally:
$l = \exists_{e\in t'}\, cond(e) \wedge (\, l \rightarrow cond(e_i) \wedge \forall_{k=1..i-1}\, \neg cond(e_k) \wedge elem=e_i$, where $e_1, \dots, e_n$ are items of *t'*.

It is used for **decision**, too:
l = **SEARCH** $_{e\in t'}$ cond(e) or l = $\exists_{e\in t'}$ cond(e)

l := false; t.first()
¬l ∧ ¬t.end()
elem := t.current()
l := cond(elem)
t.next()

Optimistic linear search

Check if a given condition stands for every element of an enumeration. If not, give the first item that violates it.

A : t:enor(E), l:$\mathbb{L}$, elem:E

cond:E → $\mathbb{L}$

Pre : t = t'

Post : (l, elem, t) = ∀**SEARCH** $_{e\in t'}$ cond(e)

Enumeration of *t* only finishes if the search is successful, otherwise it remains in state *„in process"*.

If the condition stands for every item of the enumerator, then *l* remains true. Otherwise it changes to false. In this case, *elem* contains the first item of the enumeration that violates the condition. Formally:
$l = \forall_{e\in t'}\, cond(e) \wedge (\neg l \rightarrow \neg cond(e_i) \wedge \forall_{k=1..i-1}\, cond(e_k) \wedge elem{=}e_i$, where $e_1, \ldots, e_n$ are elements of t'

```
l := true; t.first()
l ∧ ¬t.end()
    elem := t.current()
    l := cond(elem)
    t.next()
```

It is used for **decision**, too:
l = ∀**SEARCH** $_{e\in t'}$ cond(e) or l = ∀ $_{e\in t'}$ cond(e)

Conditional maximum search

Give the item of, according to a given point of view, the highest value in those items of an enumeration that satisfy a condition.

A : t:enor(E), l:$\mathbb{L}$, elem:E, max:H

Pre : t = t'

Post : (l, max, elem) = $\mathbf{MAX}_{\substack{e \in t' \\ cond(e)}}$ f(e)

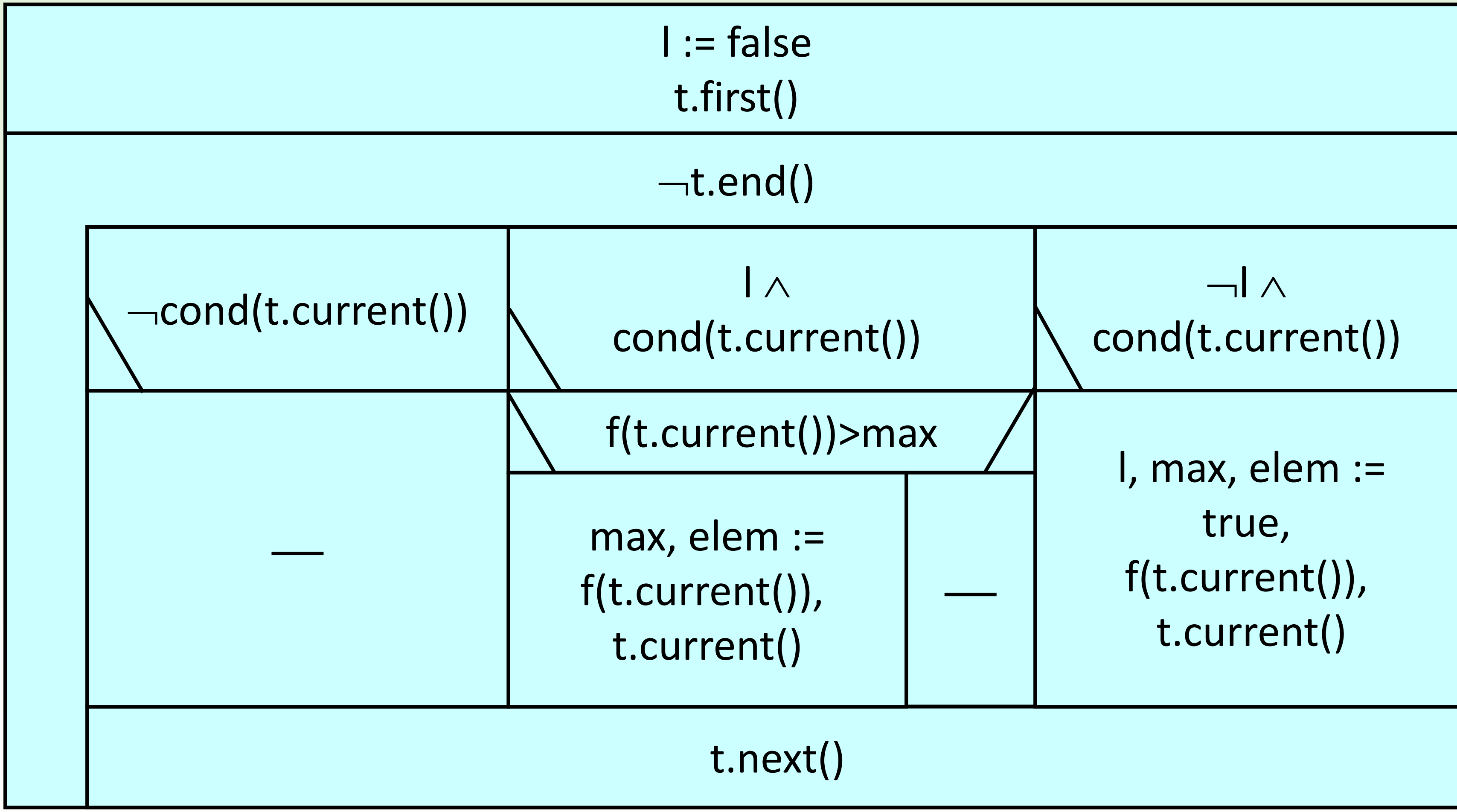
f:E → H

cond:E → $\mathbb{L}$

items of set H *can be sorted*

$l = \exists_{e \in t'}\, cond(e) \wedge (l \longrightarrow max = f(elem) = \mathbf{MAX}_{\substack{e \in t' \\ cond(e)}} f(e) \wedge elem \in t')$

Conditional maximum search



- MIN *instead of* MAX
- elem *can be skipped,* max *not*

Steps of analogy

1. Forebode the algorithmic pattern that solves (part of) the task.
2. Specify the task by executable postcondition that predicts the solution.
3. Give the differences between the task and the algorithmic pattern:
   - type of the *enumerator* with the type of its items
   - concrete representatives of the *functions* ($f:[m..n]\rightarrow H$, $cond:[m..n]\rightarrow \mathbb{L}$)
   - *operation* of H, if needed
     - ($\mathbb{Z}$, >) or ($\mathbb{Z}$, <) instead of (H, >)
     - ($\mathbb{Z}$, +, 0) or ($\mathbb{R}$, *, 1) or ($\mathbb{L}$, $\wedge$, true) instead of (H, +, 0)
   - *renaming of the variables*
4. By applying the differences in the general algorithm of the algorithmic pattern, the solution of the task is given.

Aspects in testing

- *Enumerator*-based (for all of the patterns)
  - *length*: 0, 1, and more items
  - *first* and *last*: when the special element (to be summed or satisfying a condition or the maximal) is at the beginning or at the end of the enumerator.
- *Role*-based
  - *search*: exists or not exists an item satisfying *cond*
  - *max. search*: one maximum or more maxima
  - *summation*: loading
- Particularities of the operations that calculate functions *cond*(*i*) and *f*(*i*).