ELTE | FACULTY OF INFORMATICS

PROGRAMMING

Lecture 5

Zsuzsa Pluhár

pluharzs@inf.elte.hu

Type definition

The type

- Value set
- Operators

Example: Array

- Value set:

Dimension: length of the array

interval of the index

type of the index

type of the elements

```
Type TA:Array[i1..in:Titem]
```

- Operators:

$\bullet := \bullet$ initialization

$\bullet[\bullet]$ referencing to an element

Type definition - Array

	Type definition	Specification	Algorithm
Value set	**Type** TA:**Array**[i1..in:Titem]	$X[1..]\in\mathbb{S}^*$ $X[1..n]\in\mathbb{N}^n$	X:**Array**[1.. :**Titem**] X:**Array**[1..n:**Integer**]
Operators	•:=•		X[]:=Y[] or X[]:=(1,2,3)
	•[•]	X[i]	X[i]

in C#:

```
int[] X=new int[n];
…
X[i]=3;
```

Type definition - struct

Structs (also called records, or classes) group elements which don't need to all be the same type, and are accessed by field (member) name

- Value set:

```
Type TR:Record(
  fn1.:Tf1,
  ...
  fnn:Tfn)
```

$$\textbf{Type}\ TR{:}\textbf{Record}(\ \underbrace{fn_1}_{\text{name of the field}}.{:}\underbrace{Tf_1}_{\text{type of the field}},\ \ldots,\ fn_n{:}Tf_n)$$

- Operators:

$\bullet{:}{=}\bullet$ — initialization

$\bullet.fn_i$ — referencing to a field

Type definition - struct

the direct product of fields

	Type definition	Specification	Algorithm
Value set	**Type** TR:**Record**(fn_1.:Tf_1, ... fn_n:Tf_n)	TR=($fn_1 \times fn_2 \times ... \times fn_n$), $fn_1=\mathbb{S}_1$, ... $fn_n=\mathbb{S}_n$, a∈TR	TR:**Record**[fn_1:**Titem**$_1$, ...,fn_n:**Titem**$_n$]
Operators	•:=•		a:=(1,...,"str1")
	•.fn_i	a.fn_i	a.fn_i

in C#:

```
struct TR{
  public int x;
  public string y;
};
```

```
...
TR a;
...
a.x=1;
a.x=„str1";
```

Struct – example$_1$

Task: We know the monthly income and expenses of a person. Let's calculate by how much money his asset will change by the end of the year

Specification

Input: $n \in \mathbb{N}$, $X_{1..n} \in \boxed{\mathtt{Tin}^n}$, $\boxed{\mathtt{Tin} = (in \times out), in, out = \mathbb{N}}$

Output: $sum \in \mathbb{Z}$

Precondition:

Postcondition: $sum = \left(\sum_{i=1}^{n} (X[i].in - X[i].out) \right)$

Algorithm

```
sum:=0
i=1..n
    sum:=sum+(X[i].in-X[i].out)
```

Struct – example$_2$

Task: We have birth dates from N people, let's find who has birthday in the year first!

Specification

Input: $n \in \mathbb{N}$, $D_{1..n} \in \boxed{\mathtt{Date}^n}$, $\boxed{\mathtt{Date} = (\mathtt{month} \times \mathtt{day}), \mathtt{month}, \mathtt{day} = \mathbb{N}}$

Output: $\mathtt{first} \in \mathbb{N}$

Precondition: $n > 0$ and $\forall i(1 \leq i \leq n): D[i].\mathtt{month} \in [1..12]$ and $D[i].\mathtt{day} \in [1..31]$

Postcondition: $1 \leq \mathtt{first} \leq n$ and
$\forall i(1 \leq i \leq n):$ $D[\mathtt{first}].\mathtt{month} < D[i].\mathtt{month}$ **or**
$(D[\mathtt{first}].\mathtt{month} = D[i].\mathtt{month}$ **and** $D[\mathtt{first}].\mathtt{day} \leq D[i].\mathtt{day})$

Struct – example$_2$

…

Postcondition: `1≤first≤n and ∀i(1≤i≤n): D[first]≤D[i])`

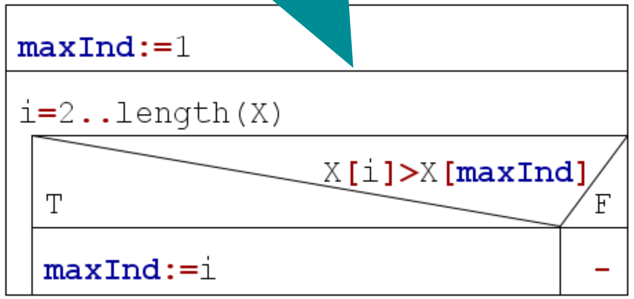
Definition:

`≤:Date×Date → 𝕃`

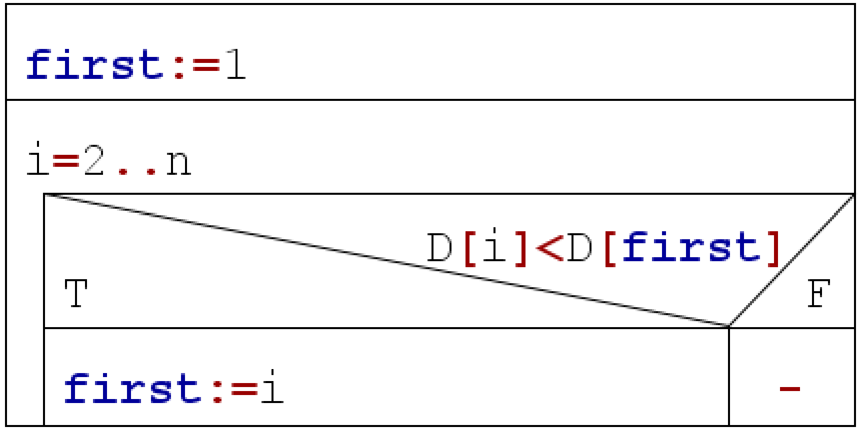
`d1≤d2 ↔ d1.month<d2.month or`

`d1.month=d2.month and d1.day≤d2.day`

maximum selection



length(X)	→	n
X[ ]	→	D[ ]
maxInd	→	first



Type struct

- **Structs** (also called records, or classes) similarly to arrays, are **compound data types**. However, the **elements are** not **referred** to **by** an index, but by their **name**.(a.x, a.y).
- It groups elements which don't need to all be the same type, and are accessed by field (member) name

type identifier

Specification : $TR=(fn_1 \times fn_2 \times \ldots \times fn_n)$, $fn_1=\mathbb{S}_1, \ldots$ $fn_n=\mathbb{S}_n$,
$a \in TR$

Using, referencing : $a.fn_i$

field identifier

data identifier

Text type

Similarities between text and array

- consists of elements of the same type,
- can be indexed;

Differences:

- the array is parameterized with the type of its elements (and index), whilst text always consists of characters,
- in an algorithm, text index always starts by 1, but array start index depends on declaration,
- array length is constant, but text length can vary; in the case of text, we have `length()` function and + operator
- modifying an element is problematic with text type

Text type

Most important text (string) and character operations

- create an empty text; e.g.: a:="";
- read in (until a character)
- write out
- return the number of characters (length())
- concatenate two texts (+)
- reference to the ith character (str[i])
- find a location of a character in a string (find())
- return a substring (from…to)
- replace or convert characters in a string
- return the character type (isalpha, isdigit, islower, ...)

Text – example$_1$

Task: Let's reverse the order of letters in a word (text)!

Specification

Input: $O \in \mathbb{T}$

Output: $R \in \mathbb{T}$

Precondition: -

Postcondition: `length(R)=length(O) and`

`∀i(1≤i≤length(O)): R[i]=O[length(O)-i+1]`

Pre-defined function:

`length:`$\mathbb{T} \rightarrow \mathbb{N}$

`length(t): character count of t`

Text – example$_1$

Sequence calculation PoA!

Task: Let's reverse the order of letters in a word (text)!

- We cannot modify the i^{th} character of a text, if it does not exist. We need the **+** operator.

Algorithm

```
R:=""
i=length(O)..1 by -1
   R:=R+O[i]
```

```
R:=""
i=1..length(O)
   R:=O[i]+R
```

Text – example$_2$

Task: Let's define the initials of an English name (e.g. Joe Black → JB)!

Specification

IsCorrect: $\mathbb{T} \to \mathbb{L}$

if all the initial letters are capital letters, but just the initials, nothing else

Input: `name`$\in\mathbb{T}$

Output: `ini`$\in\mathbb{T}$

Precondition: `IsCorrect(name)`

IsCapital: $\mathbb{C}\text{h} \to \mathbb{L}$

Postcondition:

$$length(ini) = \left(\sum_{\substack{i=1 \\ IsCapital(name[i])}}^{length(name)} (1) \right)$$

$\subseteq$: subsequence operator

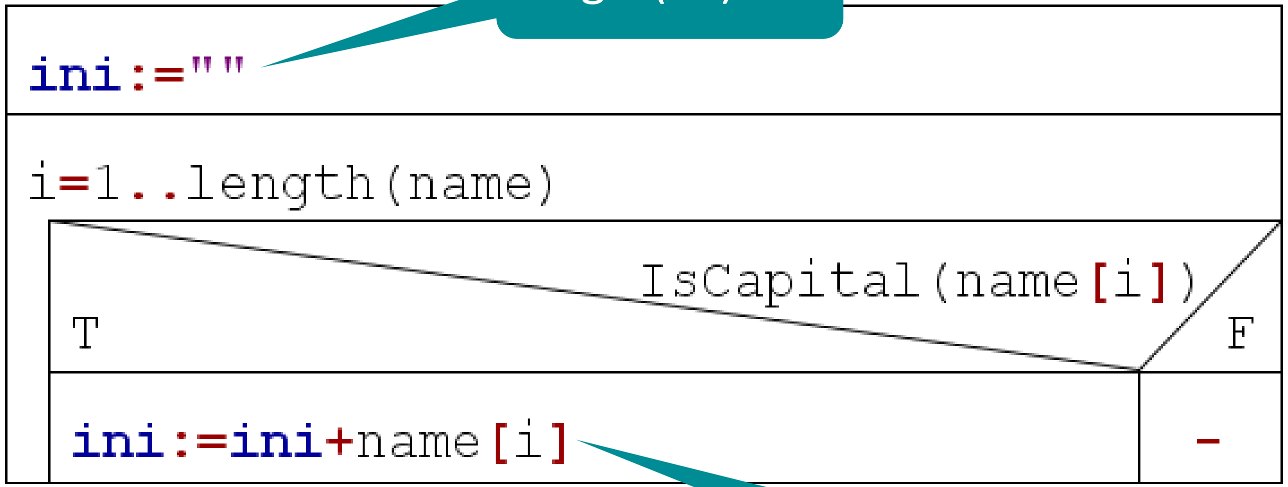
and $\forall$i(1≤i≤length(ini)): IsCapital(ini[i]) and ini $\subseteq$ name

Text – example$_2$

Task: Let's define the initials of an English name (e.g. Joe Black → JB)!

Algorithm

length(ini)=0



$Ini+Name_i \equiv Ini_1+\ldots+Ini_{length(Ini)}+Name_i$
$\rightarrow\ length(Ini+Name_i)=length(ini)+1$

Create the table

Text – example$_3$

Task: Let's define the initials of a Hungarian name (e.g. Szabó Éva → SzÉ)!

Specification

Input: $\texttt{name} \in \mathbb{T}$

Output: $\texttt{ini} \in \mathbb{T}$

Precondition: $\texttt{IsCorrect(name)}$

Postcondition:

Complete

the postcondition

Problem: in Hungarian we have double letters, where the second one is not a capital in the initial

Text – example$_3$

Task: Let's define the initials of a Hungarian name (e.g. Szabó Éva → SzÉ)!

Idea – representing data:

`DoubleL[1..8]`$\in\mathbb{T}^8$ `= ("Cs","Dz","Gy","Ly","Ny","Sz","Ty","Zs")`

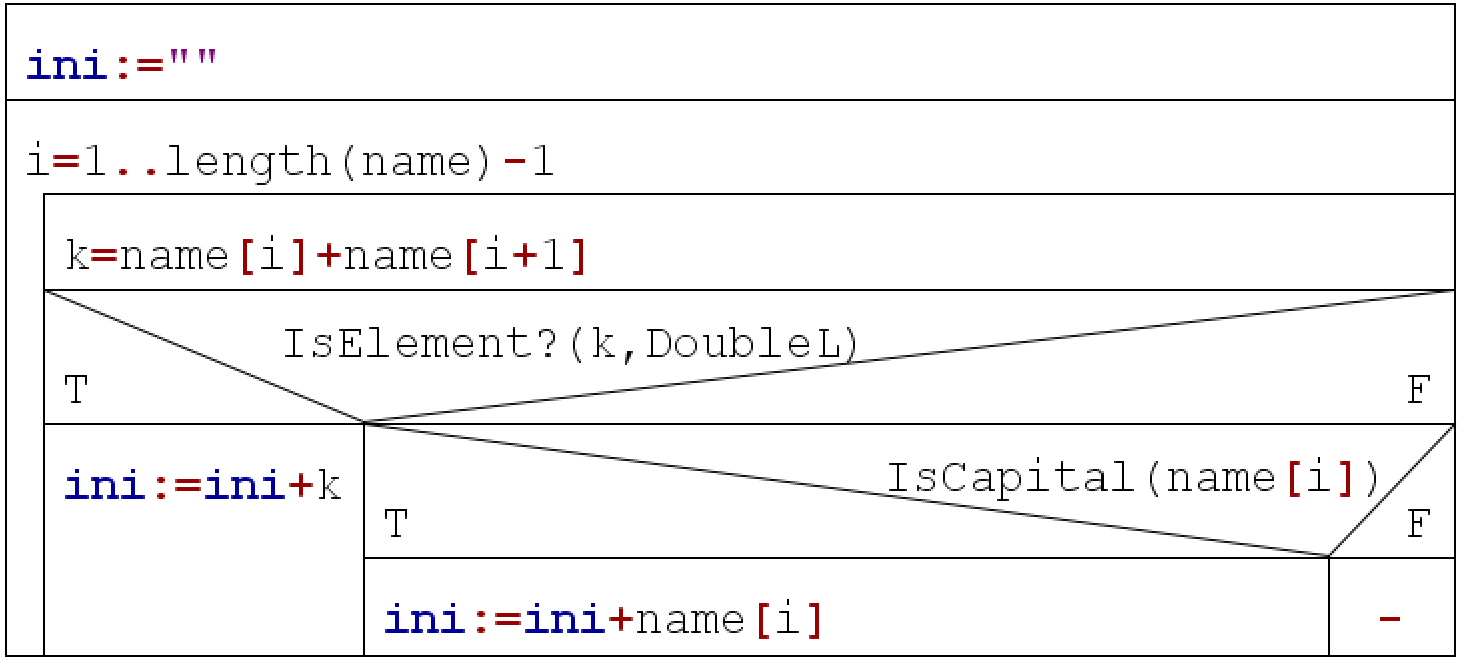
Question: How would you deal with „Dzs”?

A draft for the **algorithm**:
Let's define all two-character parts of Name, then check which one is part of `DoubleL` array, or starts with a capital letter

Text – example$_3$

Task: Let's define the initials of a Hungarian name (e.g. Szabó Éva → SzÉ)!

Algorithm:

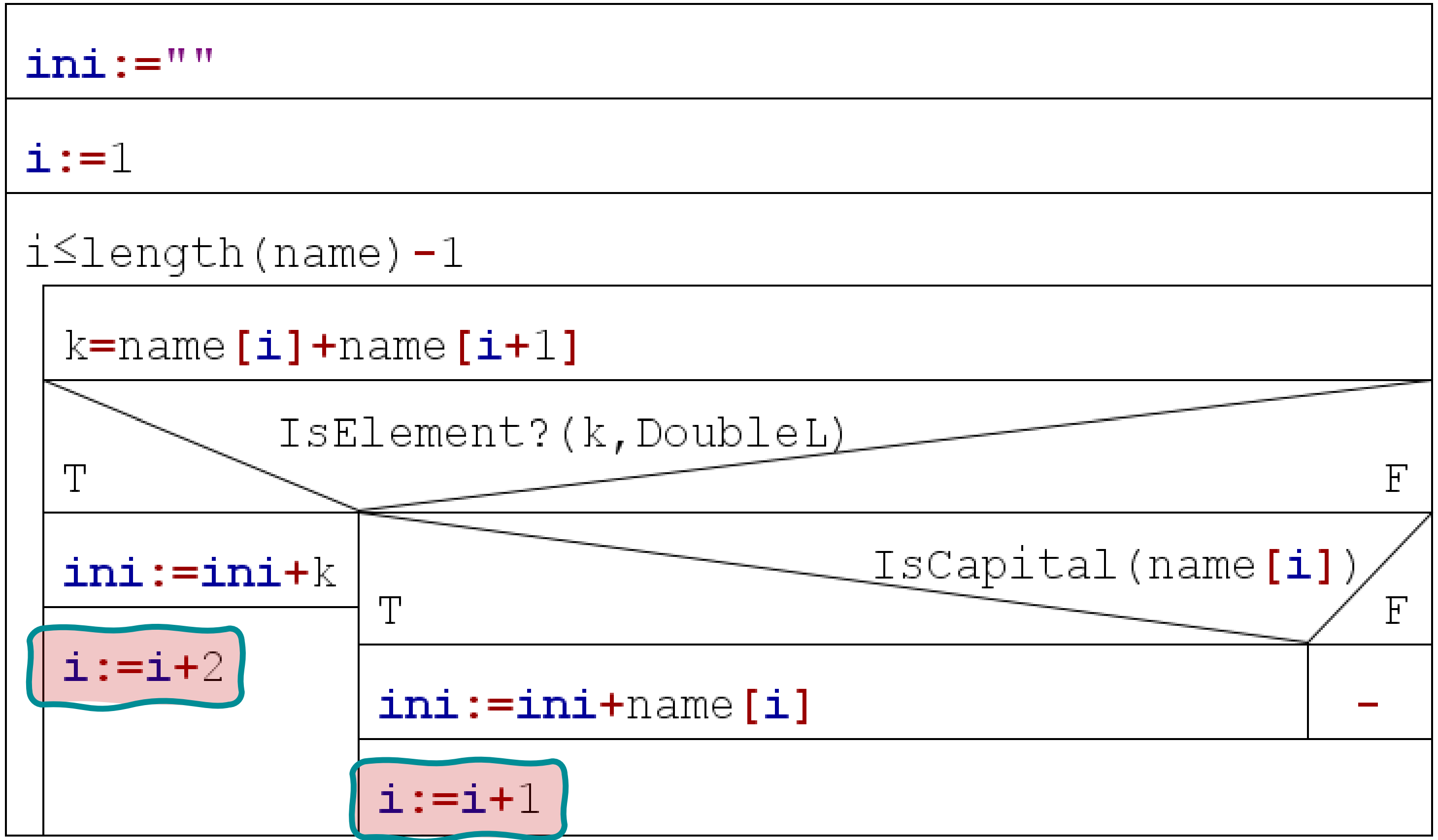


Questions: 1) Can a name be „Nagy A”? 2) Can we change the order of conditional statements? 3) Is it an optimal solution?

Text – example$_3$

Task: Let's define the initials of a Hungarian name (e.g. Szabó Éva → SzÉ)!

Algorithm:



Text – example$_4$

Task: We have two letters, let's decide which one precedes the other in the English alphabetical order!

Specification

Input: `a,b`$\in\mathbb{C}$`h`

Output: `aPrecedes`$\in\mathbb{L}$

Precondition: `isLetter(a), isLetter(b)`

Postcondition: `aPrecedes=a`$<_E$`b`

Definition: `x`$<_E$`y if and only if ???`

Assumption:

there exists function `IsLetter:`$\mathbb{C}\text{h}\rightarrow\mathbb{L}$

Text – example$_4$

Task: We have two letters, let's decide which one precedes the other in the English alphabetical order!

Idea for the solution:

Let's store all the characters in the right order, and if we find A precedes B in this array, then output is true $\rightarrow$ use Selection PoA twice

Definition:

`Letters[1..2*26]`$\in \mathbb{C}h^{2*26}$`=("a","A","b","B",…,"z","Z")`

`x`$<_E$`y` if and only if `i<j : x=Letters[i] and y=Letters[j]`

Another solution based on text type:

`Letters`$\in \mathbb{T}$`="aAbB…zZ"`

Text – example$_4$

Task: We have two letters, let's decide which one precedes the other in the English alphabetical order!

Algorithm:

```
i:=1
Letters[i]≠a
    i:=i+1
j:=1
Letters[j]≠b
    j:=j+1
aPrecedes:=(i<j)
```

Questions:

What if the precondition was not met?

Could be "a"="A"?

```
(i-1) div 2 < (j-1) div 2
```

Type definition - set

Value set:

- The iteration of the base set (that is defined by the element type) – „which items can be in the set?”
- The element type is usually a finite, discrete type, sometimes even the count of elements is limited (<256)
- If it does not exist in the language, then the implementation might allow more elements

Type definition - set

Operations (implementation)

- ToSet (puts an element into the set): $S:=S\cup\{e\}$
- FromSet (discards an element from the set): $S:=S\setminus\{e\}$
- Read (reading in the set)
- Write (writing out the set)
- Empty (creating an empty set) or Empty'SetType pre-defined constant
- Empty? (boolean function)

Type definition - set

Operations (mathematical)

- Intersection: $A \cap B$
- Union: $A \cup B$
- Difference: $A \setminus B$
- Complement: A' (not always implementable)
- Element of set: $a \in A$
- Subset: $A \subseteq B$, $A \subset B$

Type definition - set

Representation

- Listing the elements
- A boolean vector - bitmap

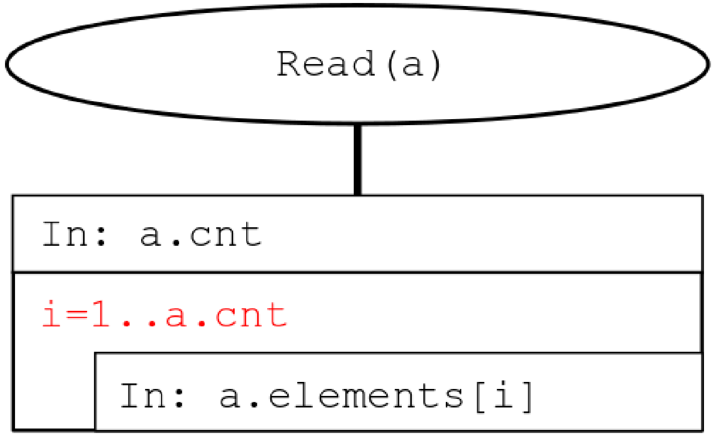
Set – by listing elements

Representation: by listing the elements

```
Set(ElementType)=
  Record(cnt:integer,
  elements:Array(1..MaxCnt:ElementType))
```

We give the set by listing its elements in an array whose length is the same as the count of elements of the set (more precisely, in the first Cnt elements).

Set – by listing elements - READ

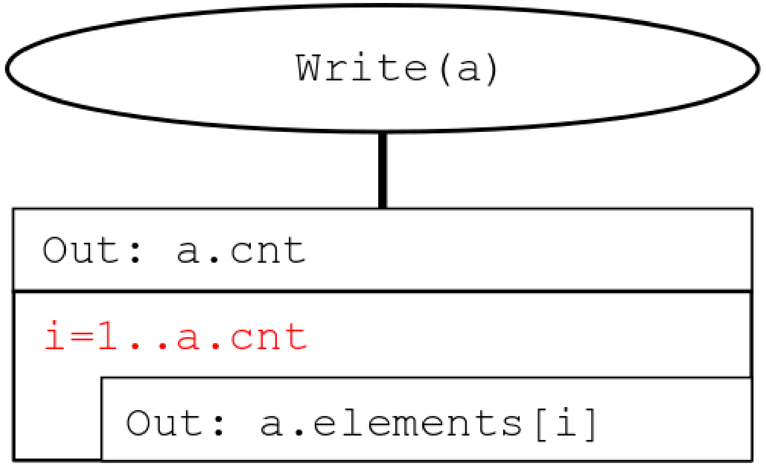


We assume that we read in a **set**.

Calculation of the operation need:

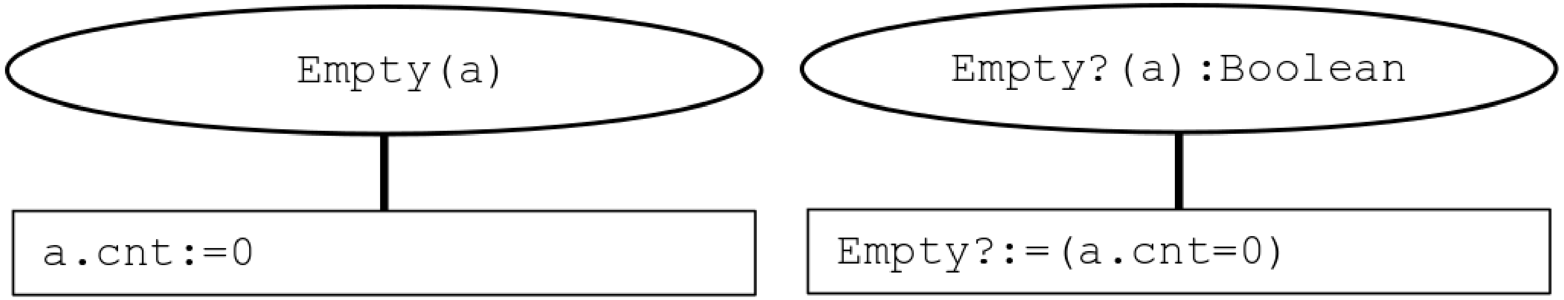
The loop will run as many times as many elements there are in the set – thus, the runtime is proportional to the count of elements of the set.

Set – by listing elements - WRITE



Calculation of the operation need:
The loop will run as many times as many elements there are in the set – thus, the runtime is proportional to the count of elements of the set.

Set – by listing elements – EMPTY, EMPTY?

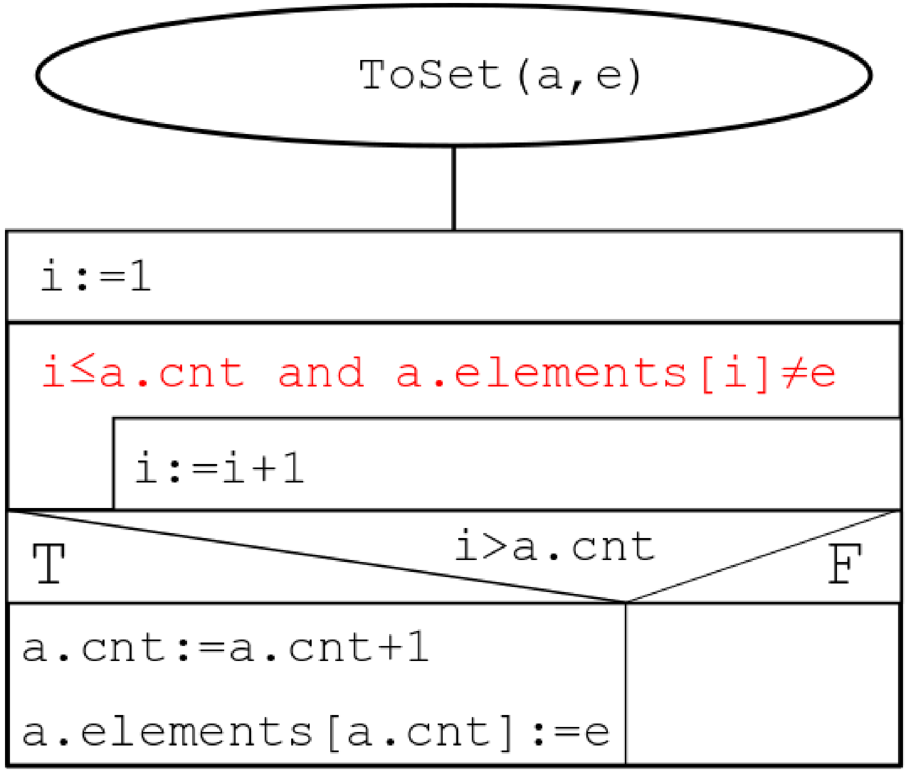


Calculation of the operation need:
It does not depend on the count of elements of the set.

Set – by listing elements – ToSet

Applying the **Decision** PoA

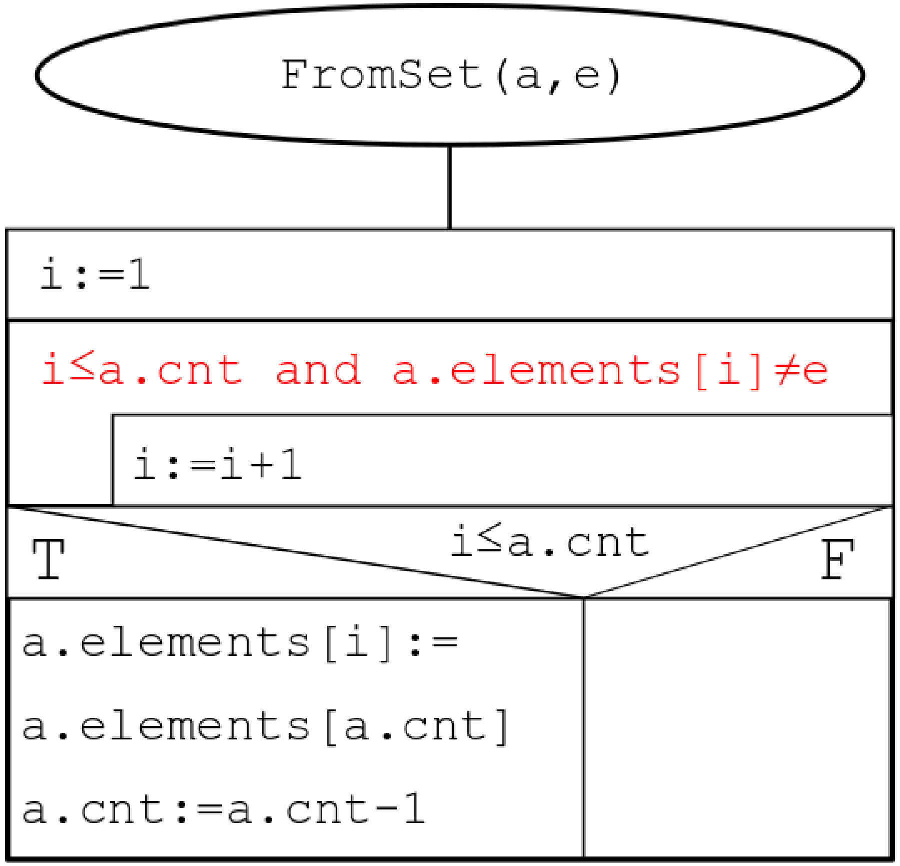
Calculation of the operation need:
The loop will run as many times as many elements there are in the set – thus, the runtime is proportional to the count of elements of the set.



Set – by listing elements – FromSet

Applying the **Search** PoA

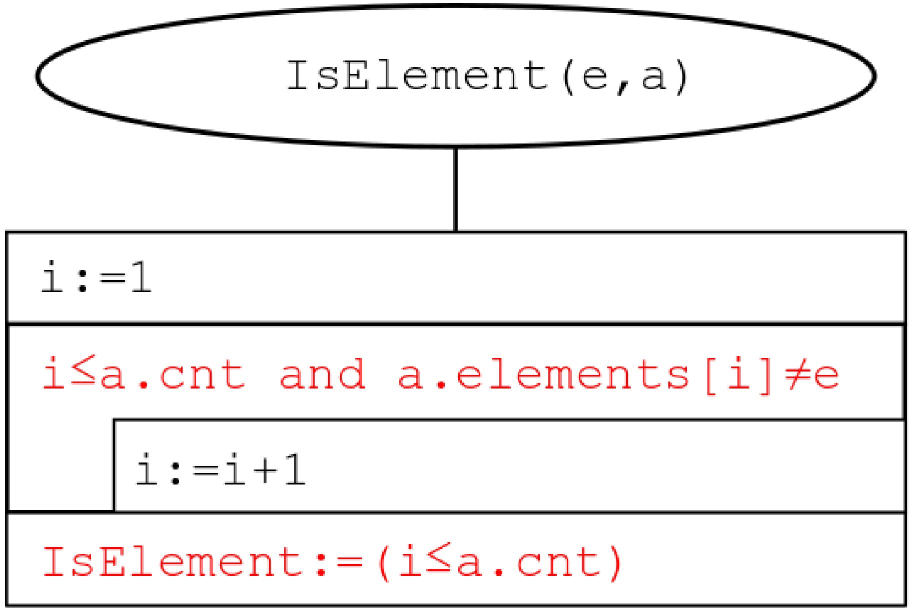
Calculation of the operation need:
The loop will run as many times as many elements there are in the set – thus, the runtime is proportional to the count of elements of the set.



Set – by listing elements – IsElement

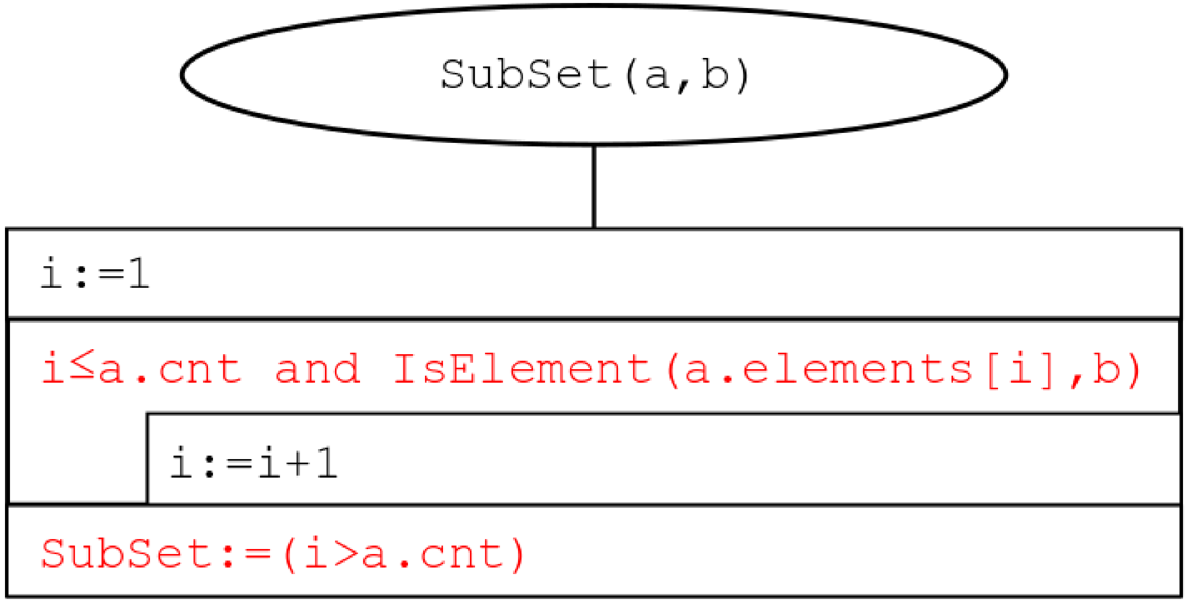
Applying the **Decision** PoA

Calculation of the operation need:
The loop will run as many times as many elements there are in the set – thus, the runtime is proportional to the count of elements of the set.



Set – by listing elements – SubSet

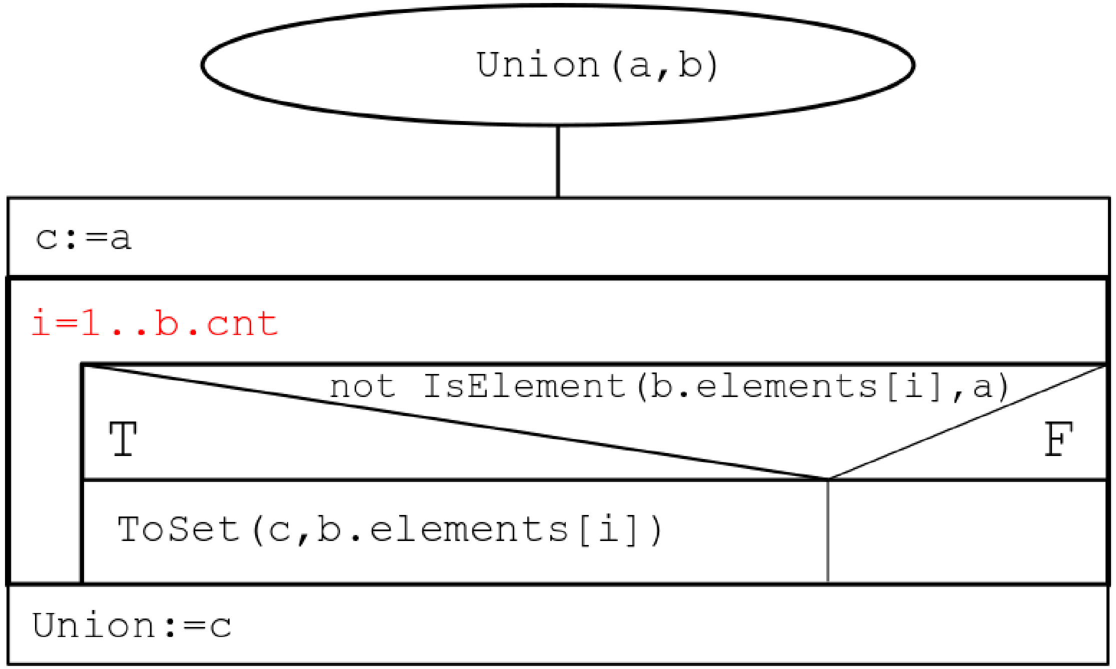
Applying the **Decision** PoA
with decision in the conditional
statement



Calculation of the operation need:
The loop will run as many times as many elements there are in set A, the IsElement function as many times as many elements there are in set B, thus, the runtime is proportional to the product of the count of elements in the two sets.

Set – by listing elements – SubSet

Applying the **Copy, Multiple item selection, Decision** PoA's

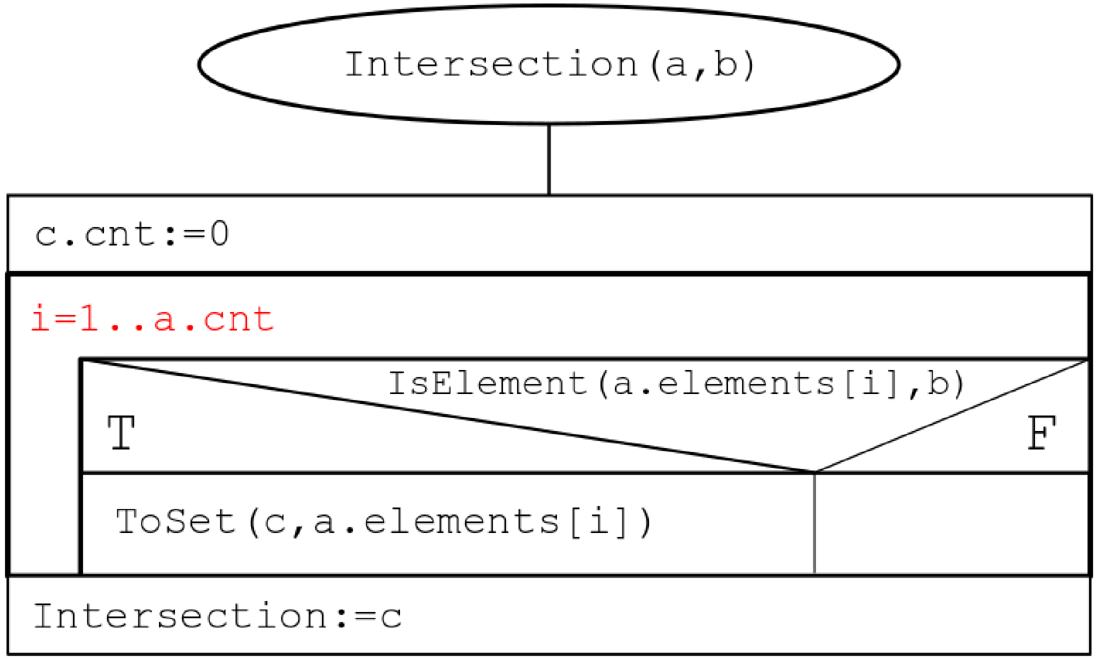


Calculation of the operation need:
The loop will run as many times as many elements there are in set B, the IsElement function as many times as many elements there are in set A, thus, the runtime is proportional to the product of the count of elements in the two sets.

The set type – by listing elements – Intersection

Applying the **Multiple item selection, Decision**

PoA's



Calculation of the operation need:
The loop will run as many times as many elements there are in set A, the IsElement function as many times as many elements there are in set B, thus, the runtime is proportional to the product of the count of elements in the two sets.

The set type – by listing elements

Notes:

- Problem: it is not checked if only elements that should be in the set are actually in the set.
- No limit on the type of elements stored in the set, as we can store **anything** in an vector
- **No limit** on the **count of elements** of the base set that the elements of the set derive from. We only limit the count of elements of the specific sets.

The set type – as boolean vector

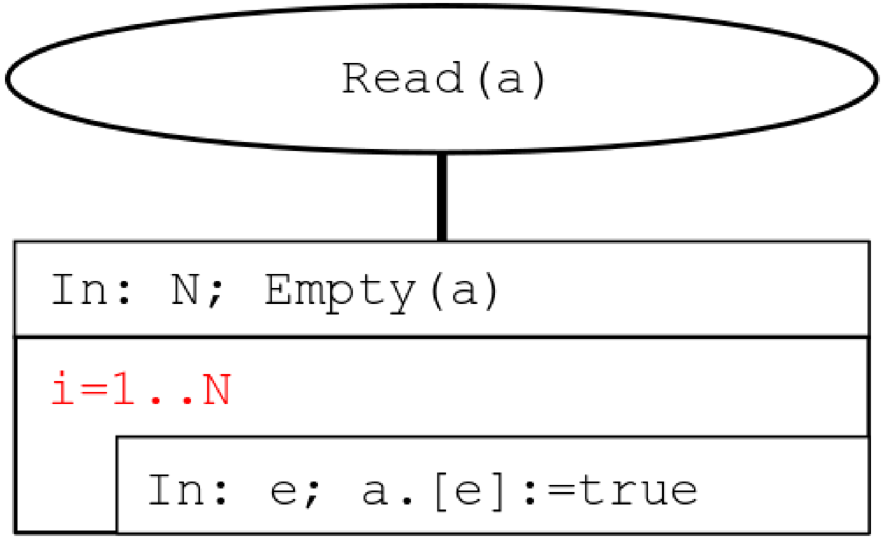
Bit map – boolean vector:

`Set(ElementType)=Array(Min’ElementType..Max’ElementType: Boolean)`

We interpret the set as a vector of `{true,false}` elements, where we use the **value** of the element as **index**.

Such a set is always **sorted**.

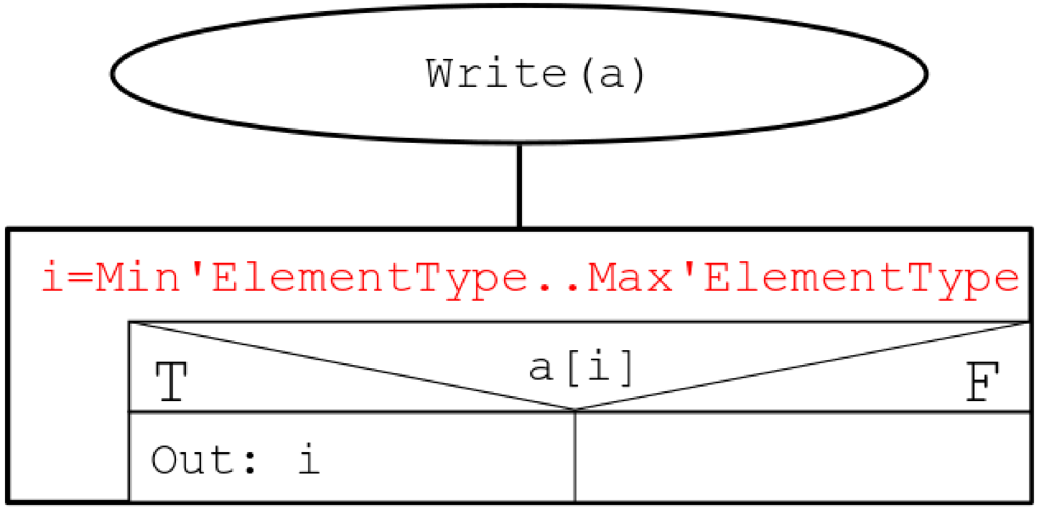
The set type – as boolean vector - READ



Calculation of the operation need:

The operation need of Empty(a) and the loop. The loop will run as many times as many elements there are in the set – thus, the runtime is proportional to the count of elements of the set

The set type – as boolean vector - WRITE

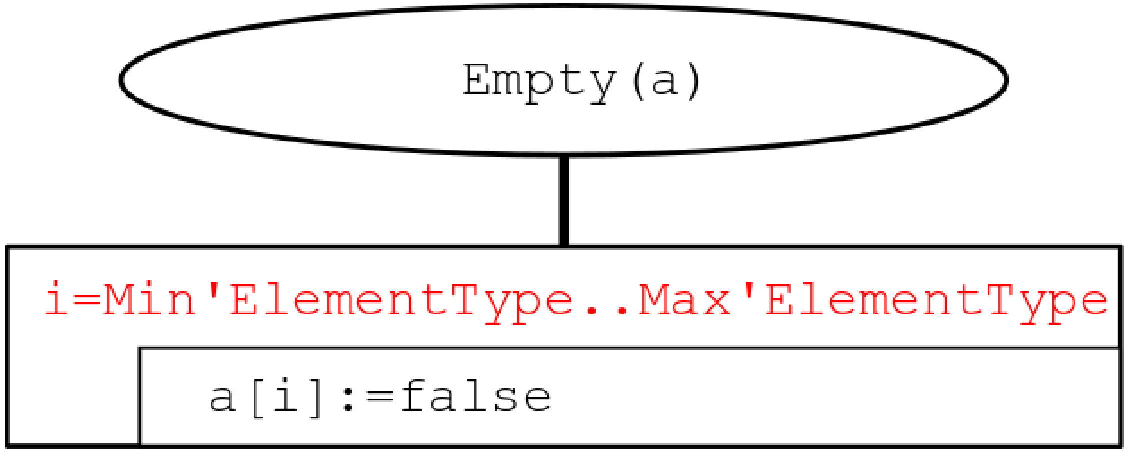


Calculation of the operation need:

The loop will run as many times as many elements there might be in the set – thus, the runtime is proportional to the cardinality of element type of the set.

What if we stored the maximum and minimum element of the set?

The set type – as boolean vector - EMPTY



Calculation of the operation need:

The loop will run as many times as many elements there might be in the set – thus, the runtime is proportional to the cardinality of element type of the set.

The set type – as boolean vector – EMPTY?

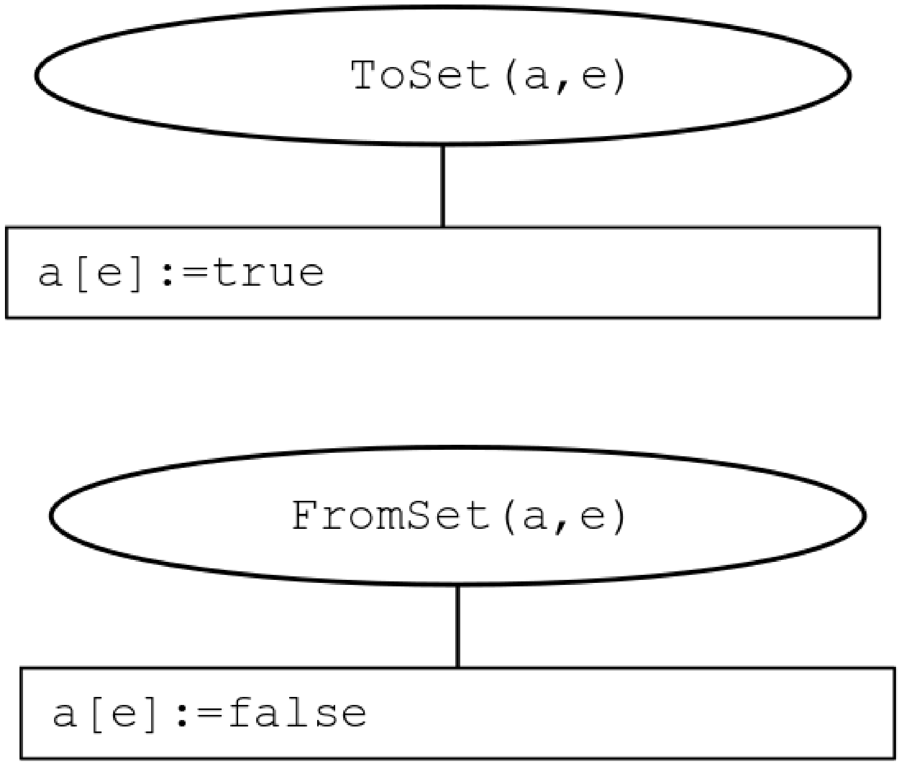
Applying the **Decision** PoA

```
Empty?(a):Boolean
i:=Min'ElementType
i≤Max'ElementType and not a[i]
    i:=i+1
Empty?:=(i>Max'ElementType)
```

Calculation of the operation need:

The loop will run as many times as many elements there might be in the set – thus, the runtime is proportional to the cardinality of element type of the set.

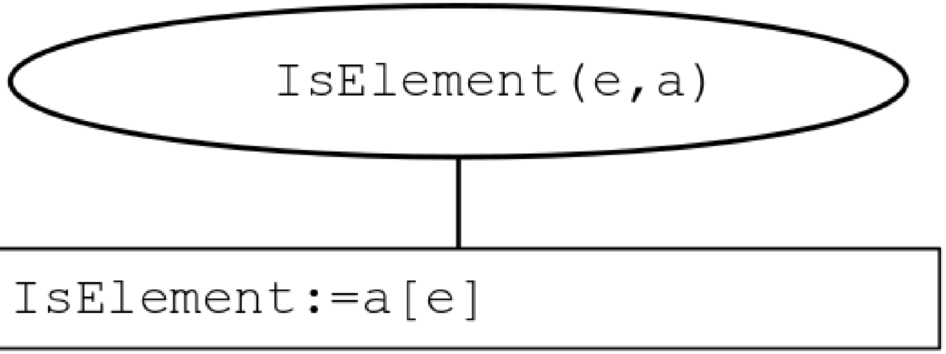
The set type – as boolean vector – ToSet, FromSet



Calculation of the operation need:

It does not depend on the count of elements of the set.

The set type – as boolean vector – IsElement



Calculation of the operation need:

It does not depend on the count of elements of the set.

The set type – as boolean vector – SubSet

Applying the **Decision**
PoA

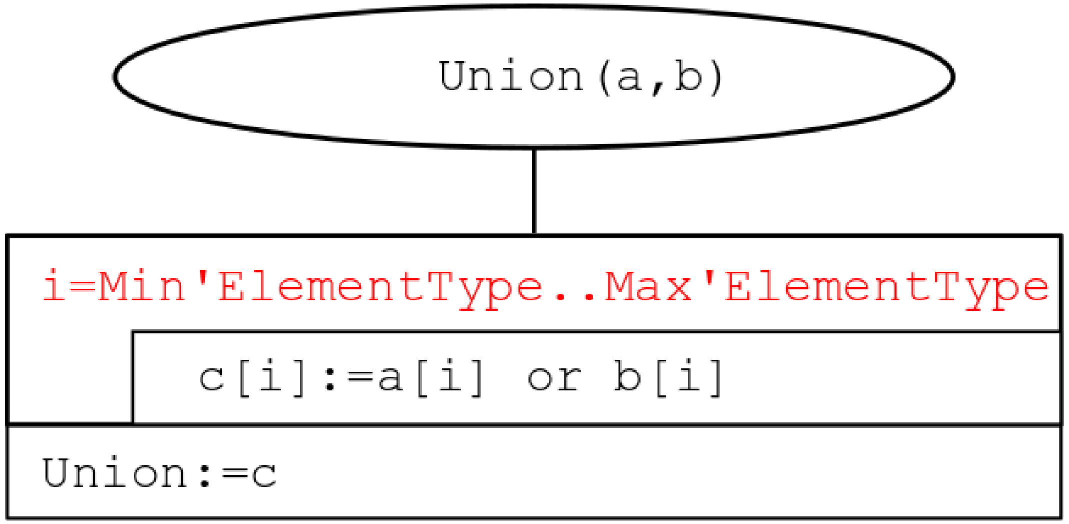
```
SubSet(a,b)
i:=Min'ElementType
i≤Max'ElementType and not (a[i] and not b[i])
    i:=i+1
SubSet:=(i>Max'ElementType)
```

Calculation of the operation need:

The loop will run as many times as many elements there might be in the set – thus, the runtime is proportional to the cardinality of element type of the set.

The set type – as boolean vector – Union

Applying the
Copy PoA

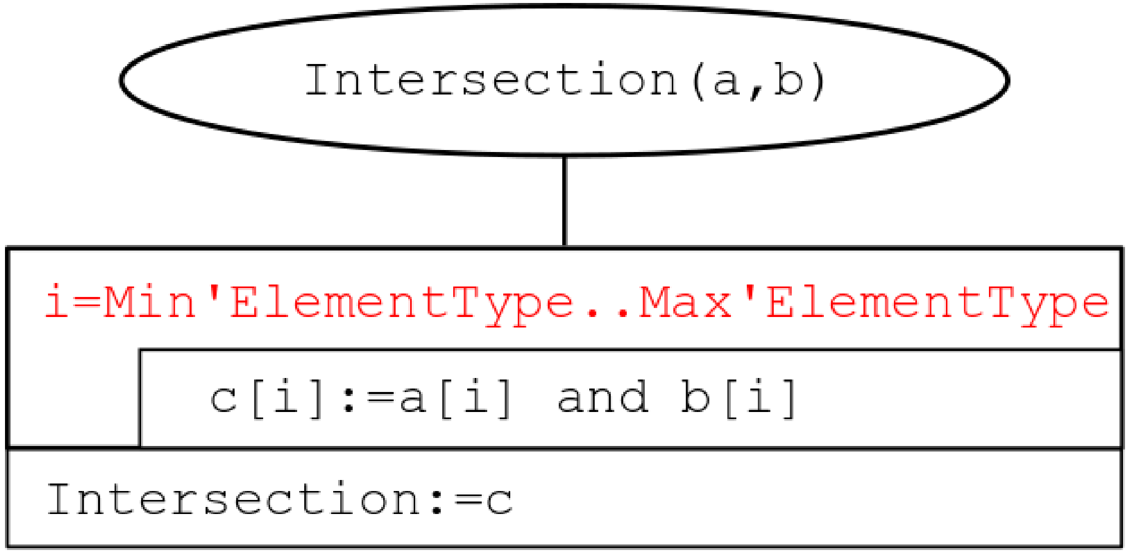


Calculation of the operation need:

The loop will run as many times as many elements there might be in the set – thus, the runtime is proportional to the cardinality of element type of the set.

The set type – as boolean vector – Intersection

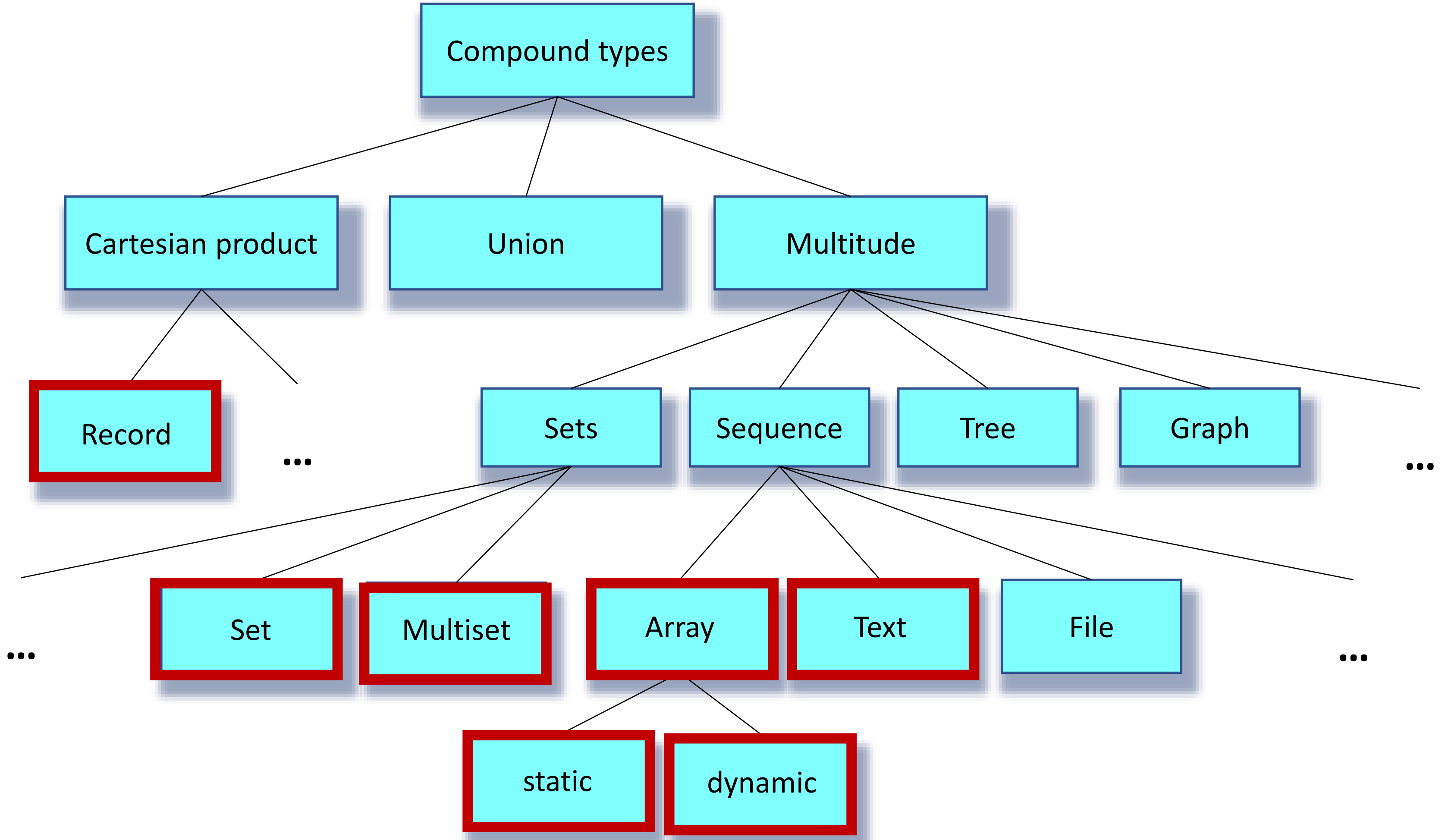
Applying the **Copy** PoA

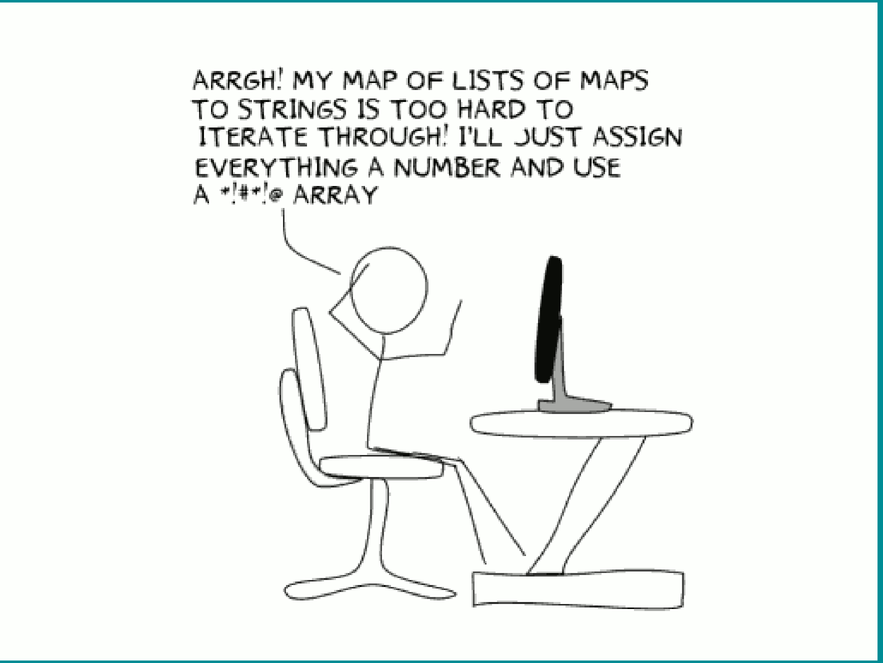


Calculation of the operation need:

The loop will run as many times as many elements there might be in the set – thus, the runtime is proportional to the cardinality of element type of the set.

Compound types





Thank you for your attention!