Introduction to Programming

Control Structures I

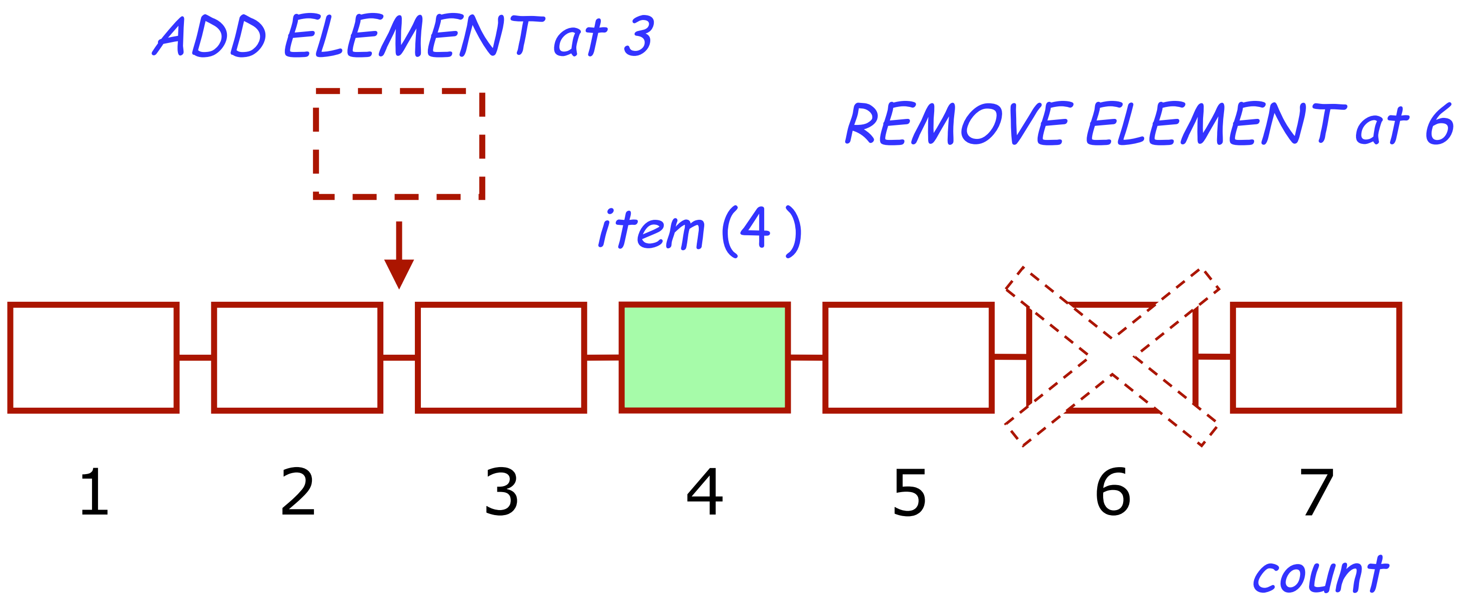
Lecture 4 - Manuel Mazzara

Agenda

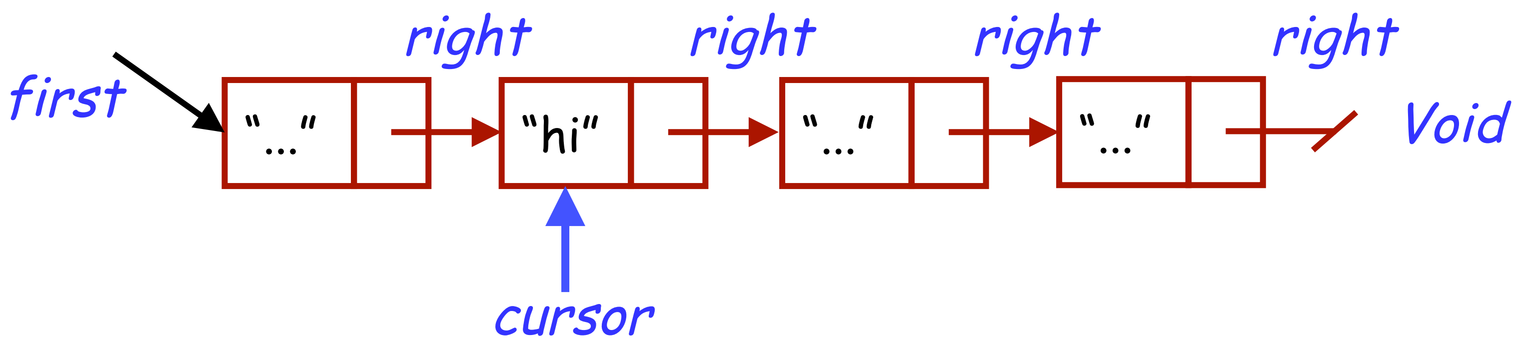
- The need for control structure (flashback cinema technique)
- The notion of algorithm
- Basic control structures: sequence, conditional, loop
- Decision structures: variants of conditional instruction
- Repeating operations: the loop
- Rationale for the “Control Structures of Structured Programming”
- Examples on List data structure

Programming Scenario: Data Structures (list)

- A **list** is a container storing items, identified by an integer index, where items can be **inserted** or **removed** at any position



(Singly) Linked List (1)

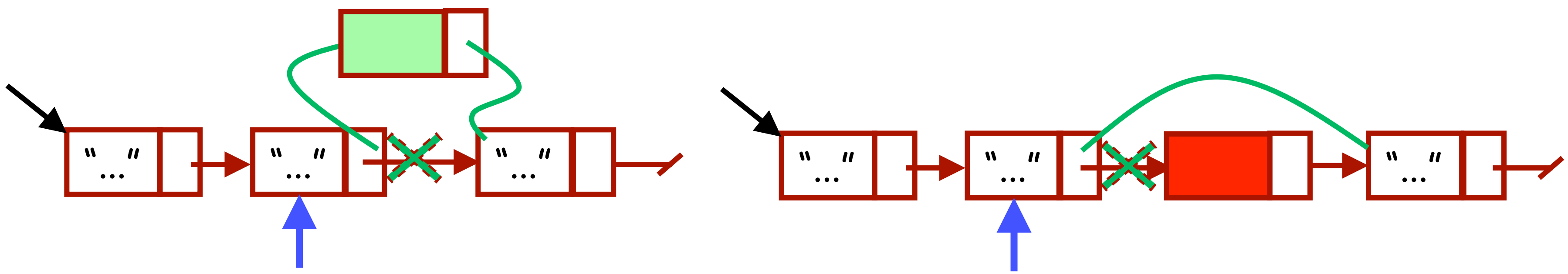


Stores each item in a cell, which knows where to look for the next cell

(Singly) Linked List (2)

- One of the several possible implementations of ”list”
- Each node has a pointer to the next node
  - void to end the list
- It can be used as a stack or LIFO queue
- “Fast” operations - O(1)
  - <u>Adding</u> a new first element
  - <u>Removing</u> the existing first element
  - <u>Examining</u> the first element

(Singly) Linked List (3)



No direct access by index; to point to an item use a **cursor**, which moves left to right

Once the position is found, **inserting/removing** in the middle is easy, but you need to find it!

List traversal

- Traversing a list : visit <u>every node</u> in the list
- When you ***visit*** a **node**, you want to:
  - (at least) **access** (i.e. read) the data stored in that node
  - **modify** the data stored in that node
  - Possibly both

- How do you do this traversal?
- How do you move the cursor from one node to the other?
- Should you repeat the same code fragment over and over again?
- And for how many times? How do you know how many nodes are there?

We need loops (pseudocode example)

Node *cursor*; // support variable used to "visit" a Node object

cursor = first; // Start with a visit to the first node

while (*cursor* is not void) **Control Structure**

{

//**read/update variables** inside the Node (do what you want on the current node!)

cursor = *cursor.right*;

// *cursor* will point to the next node

// i.e.: *cursor* will "visit" the next node in the list

}

The notion of Algorithm

- You will see more in detail at year 2: Theoretical Computer Science
- An algorithm is the specification of a (computational) process to be carried out by a computer
  - Example: Sum of two integer numbers

- Do you know where the world “algorithm” come from?

- How the word “logarithm” and “algorithm” are related?
  - They are not!

Algorithm, intuitively


PREPARAZIONE E TEMPI DI COTTURA
ZUBEREITUNG - PREPARATION

Versate le verdure ancora surgelate in 1 litro abbondante d'acqua fredda con 2 cucchiai d'olio, salate e cuocete secondo i tempi indicati.

Tiefgefrorene Gemüse in einen Liter kaltes Wasser geben, 2 Esslöffel Öl und Salz hinzufügen.

Verser les légumes surgelés dans 1 litre d'eau froide, ajouter deux cuillers à soupe d'huile et du sel.

Pour the still frozen vegetables in one liter of cold water, add two tablespoons of oil and salt

Properties of an algorithm

1. Defines the **data** on which the algorithm will be applied (data and algorithm go side by side in computer science)

2. Every **elementary step** taken from a set of well-specified (basic) actions

3. Describes **ordering(s) of execution** of these steps (composed actions)

4. Properties 2 and 3 based on precisely **defined conventions**, suitable for an automatic device

5. It has a **finite number of steps**

6. For any data, guaranteed to terminate after **finite number of steps**
   - Not everybody agrees on this 6th property

Algorithm vs. Program

- "Algorithm" usually considered a **more abstract** notion, independent of platform, programming language etc.
- In practice, the distinction tends to fade:
  - Algorithms need a **precise notation**
  - Programming languages becoming more abstract
- In programs, **data** (objects) are just as important as **algorithms**
- A program typically contains many algorithms and object structures
  - We will see an example later on today
- **Program = algorithm(s) + data structure(s)**

Constituent parts of an algorithm

- Basic/elementary steps
  - Assignment
  - Feature call ***x.f(a)***
  - If OOP, otherwise simple procedure calls
- **Control Structures**
  - Sequencing/composition of these basic steps

Control Structures

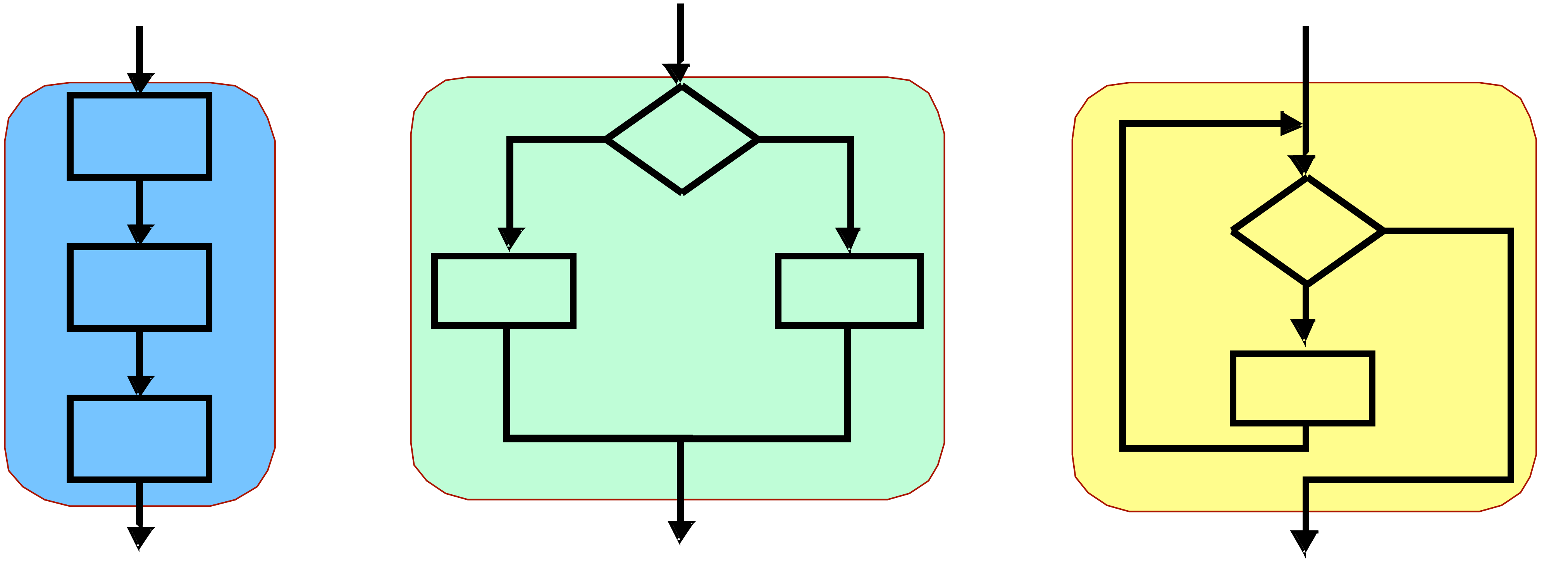
- Construct that describes the scheduling of basic actions

- There are three fundamental control structures in programing languages

    - Sequence
    - Loop
    - Conditional

“Control structures of Structured Programming”

One-entry, one-exit



(Compound) (Conditional) (Loop)

Böhm-Jacopini theorem (1966)

- *Computable functions* can be implemented by the use of the following control structures:
  - Executing one subprogram, and then another subprogram (**sequence/compound**)
  - Executing one of two subprograms according to the value of a boolean expression (**selection/conditional**)
  - Executing a subprogram until a boolean expression is true (**iteration/loop**)

- The statement is simplified and no proof presented
  - Get the idea!

Control structures as problem solving

- Sequence: to achieve C from A, first achieve an intermediate goal B from A, then achieve C from B
- Conditional: solve the problem separately on two or more subsets of its input set
- Loop: solve the problem on successive approximations of its input set
- Let us look at this into details

The sequence (compound)

instruction $_1$

instruction $_2$

…

instruction $_n$

Semicolon as optional separator

$instruction_1$ $\boxed{;}$

$instruction_2$ $\boxed{;}$

... $\boxed{;}$

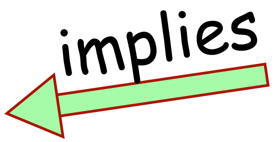
$instruction_n$

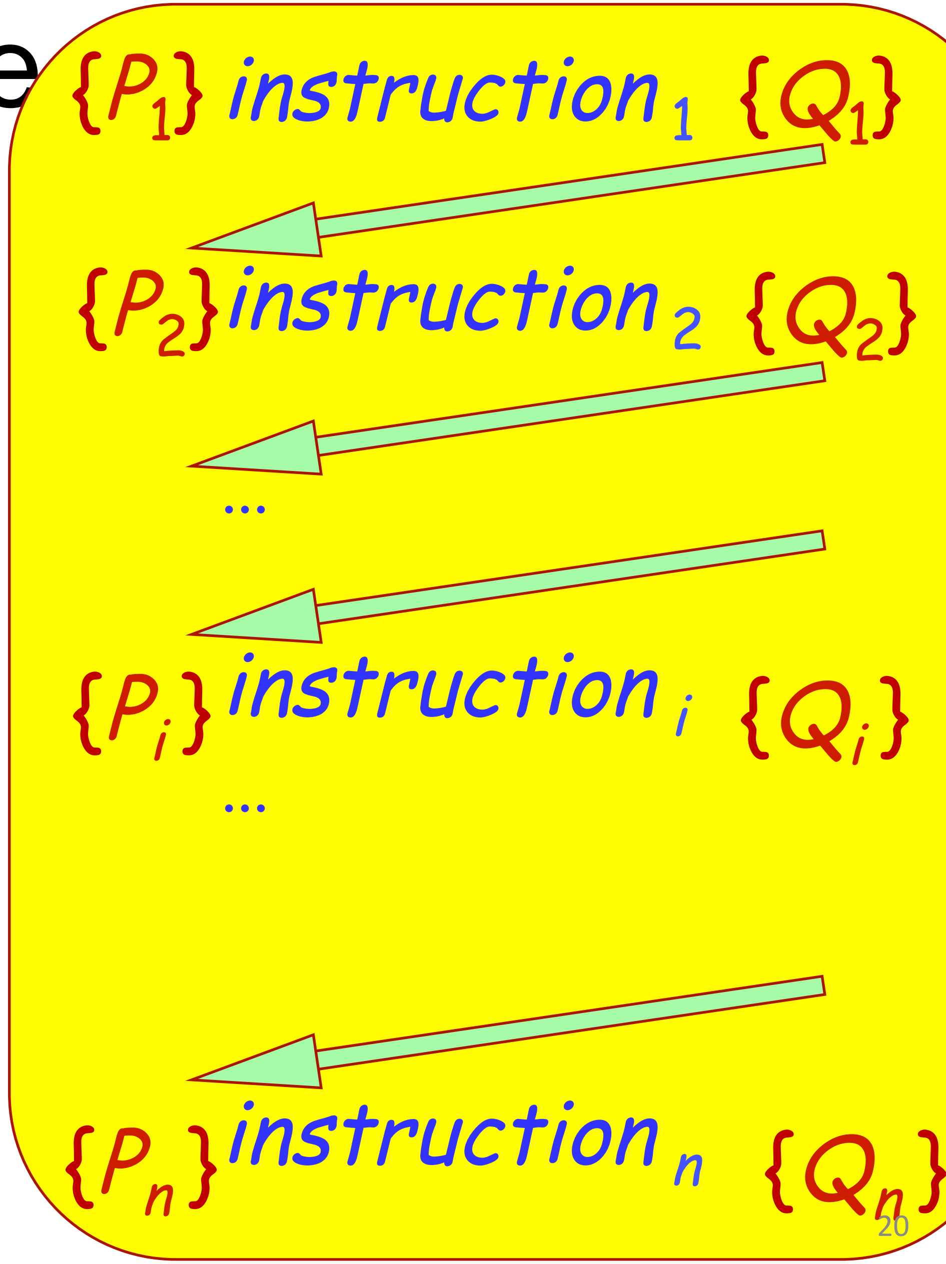
Correctness of sequence

Precondition of *instruction 1* must hold initially

Postcondition of each *instruction i* must imply precondition of each *instruction i + 1*

Final effect is postcondition of *instruction n*





Conditional instruction

```
if
        Condition               -- Boolean_expression
then
        Instructions            -- Compound
else
        Other_instructions      -- Compound

end
```

Computing the greater of two numbers

```
if
        a > b
then
        max   := a
else
        max   := b
end
```

Computing the greater of two numbers (query)

```
maximum (a, b : INTEGER): INTEGER
            -- The larger value between a and b.
    do
        if
                a > b
        then
                Result := a
        else
                Result := b
        end
    end
```

Example

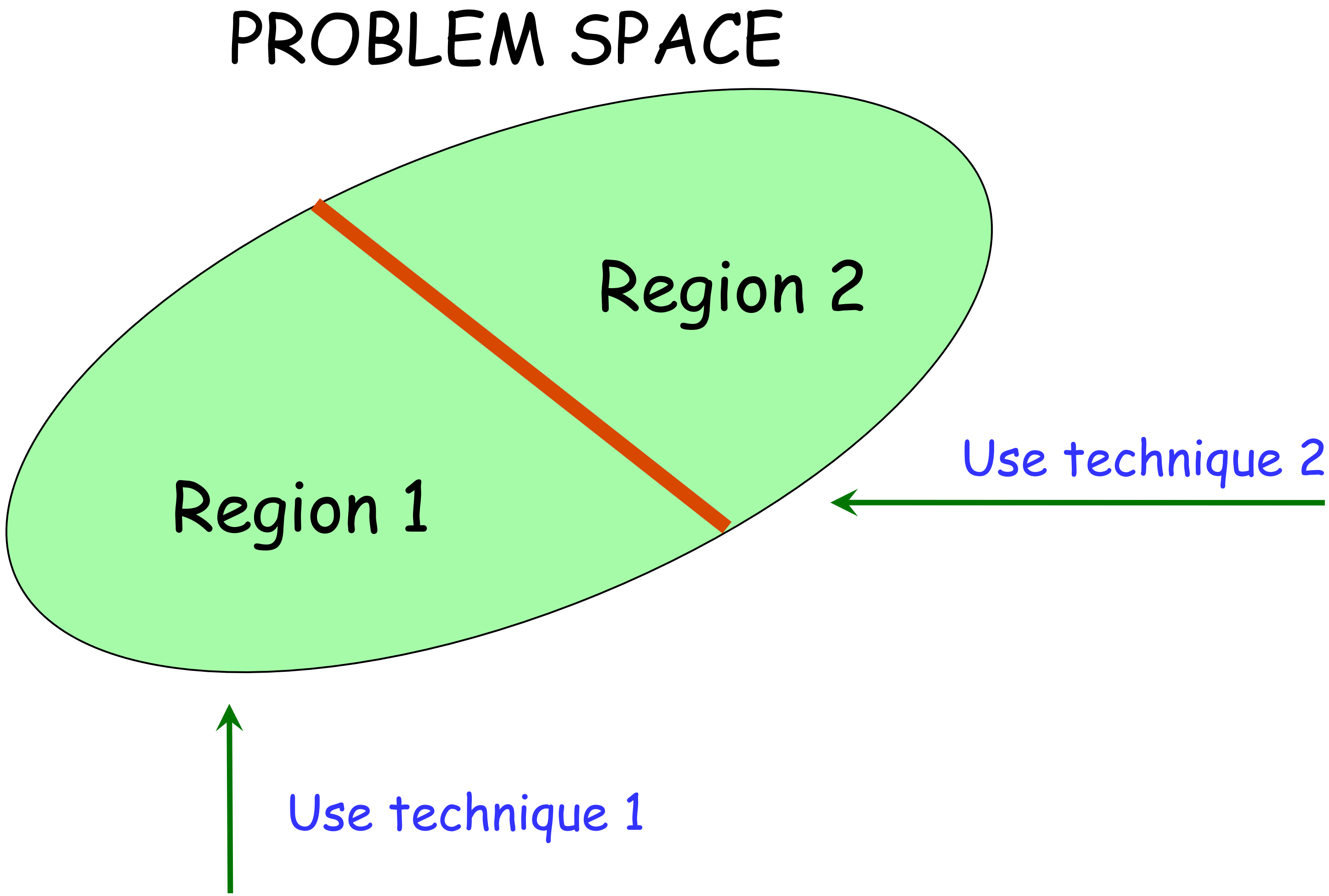
i, j, k, m, n: INTEGER

…

m := maximum (25, 32)

n := maximum (i + j, k)

Conditional as problem-solving technique



Basic form of conditional

```
if Condition then
        Instructions
else
        Other_instructions
end
```

Variant

```
if  Condition then
        Instructions
end
```

=

```
if  Condition then
        Instructions
else

end
```

Empty clause

“Then” without “else” clause

```
if date > due_date then
        penalty := ...
        amount := amount + penalty
end
```

Nested vs. comb-like structure

Nesting

```
if Condition1 then
        Instructions 1
else
        if Condition2 then
                Instructions 2
        else
                if Condition3 then
                        Instructions 3
                else
                        if Condition3 then
                                Instructions 4
                        else
                        end                    . . .
                end
        end
end
```

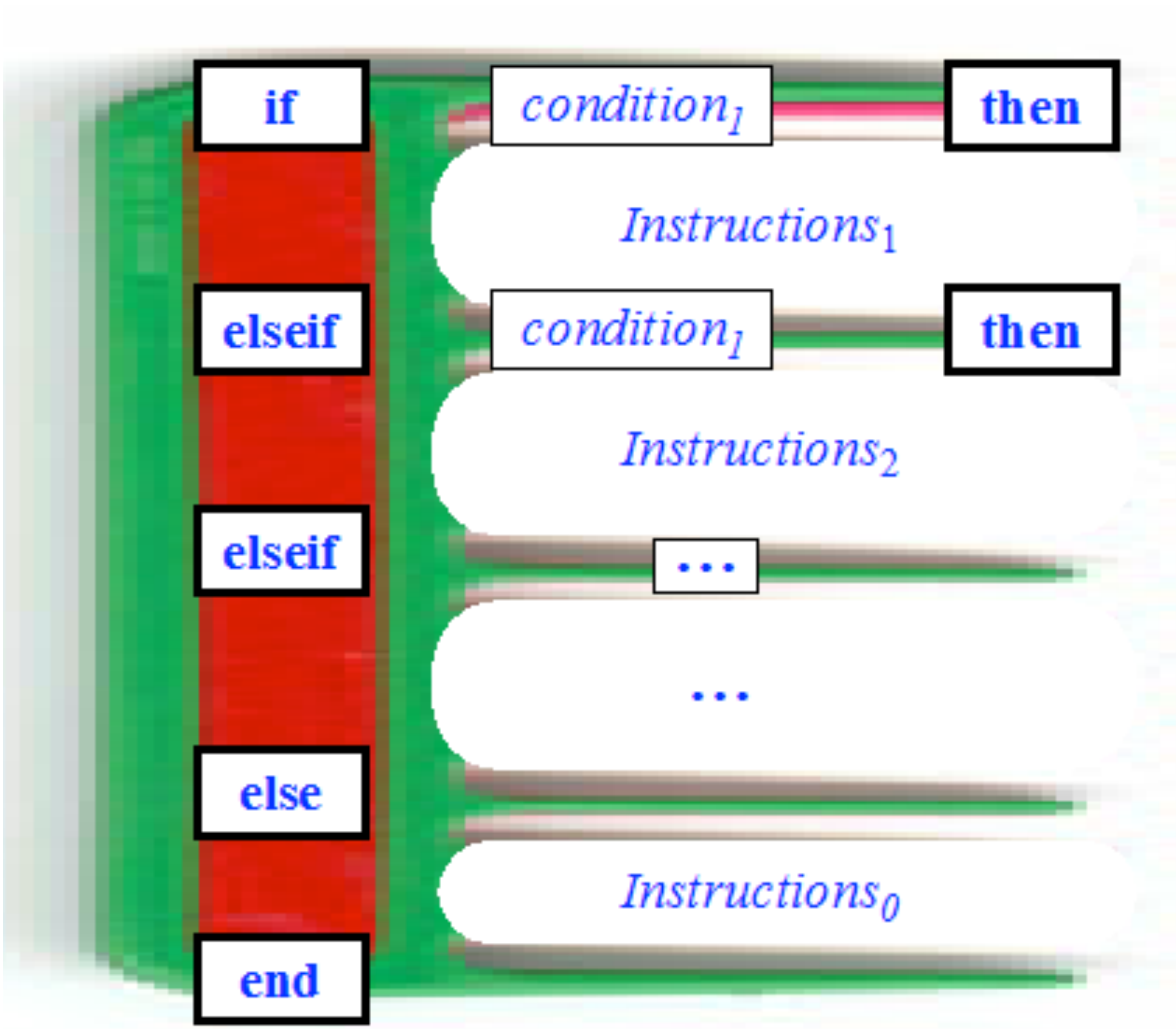
Comb-like conditional

if *Condition*$_1$ **then**
    Instructions$_1$
elseif *Condition*$_2$ **then**
    Instructions$_2$
elseif *Condition*$_3$ **then**
    Instructions$_3$
elseif
    ...
else
    Instructions$_0$
end

Always one

Zero or one

Comb-like structure



Zero or more

More on Control Structures

- Loops and their invariants
- Loop termination
- General problem of termination will be discussed in TCS course next year
- Lower-level control structures: “Goto” and flowcharts
- Rationale for the “control structures of Structured Programming”

Loop

```
from
        Initialization                  -- Compound

until
        Exit_condition                  -- Boolean_expression
loop
        Body   -- Compound. Executed until Exit_condition is false
end
```

Example

```
from
        i := 1
    until
        i > n
    loop
        sum := sum + i
                i := i + 1
end
```

Loop, full form

```
from
        Initialization              -- Compound
invariant
        Invariant_expression        -- Boolean_expression
variant
        Variant_expression          -- Integer_expression
until
        Exit_condition              -- Boolean_expression
loop
        Body                        -- Compound
end
```

Loop Variant and Invariant

- Loop Invariant and Variant are optional

- Will be studied in detail later on in the course
  - Just a quick tour here

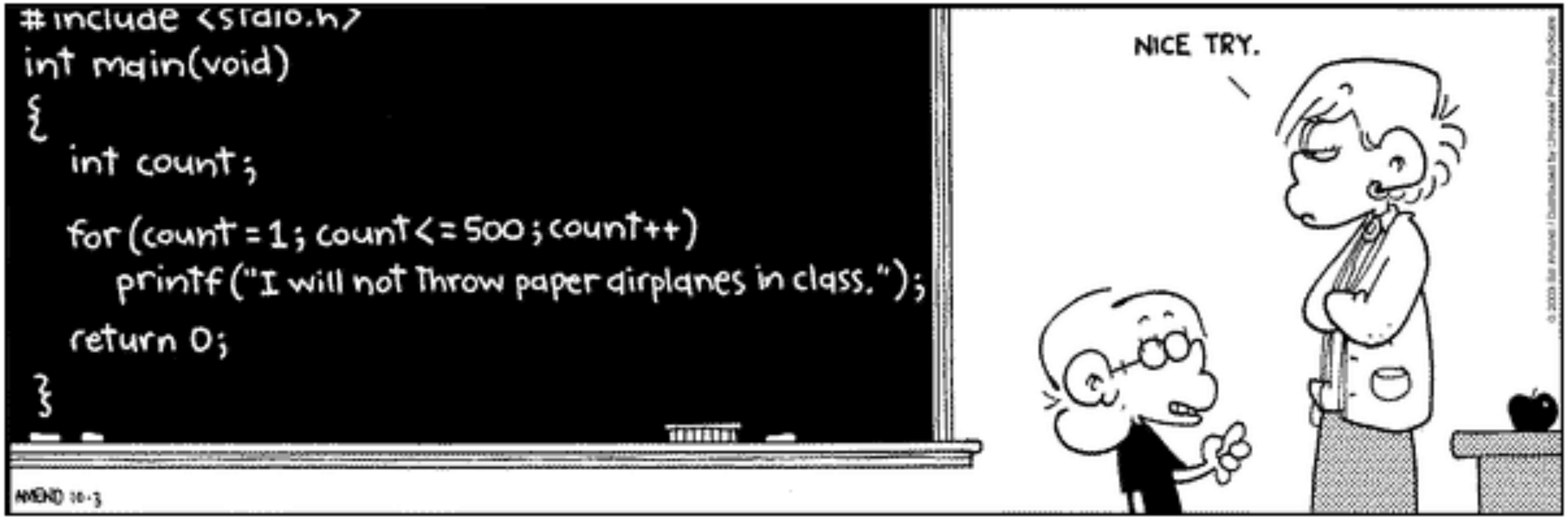
- **Invariant**: a property which will be **true after initialization** and preserved after **every iteration** of the body

- **Variant**: will help us determine that the loop **terminates**
  - We will see this later on in the course

Example

```
loop_example
        -- A loop example...
        local
            count: INTEGER
        do
            from
                count := 1
            invariant
                count >= 1
                count <= 101
            variant
                101 - count
            until
                count > 100
            loop
                io.put_integer (count)
                io.put_new_line
                count := count + 1
        end
    end
```

Other loop syntaxes



Different Syntaxes

```
from                -- Eiffel
        Initialization
until
        Condition
loop
        Body
end
```

```
for (Initialization; Condition; Advance) do
        Body
end
```

```
for i : a..b do
        Body
```

```
repeat
        Body
until
        Condition
end
```

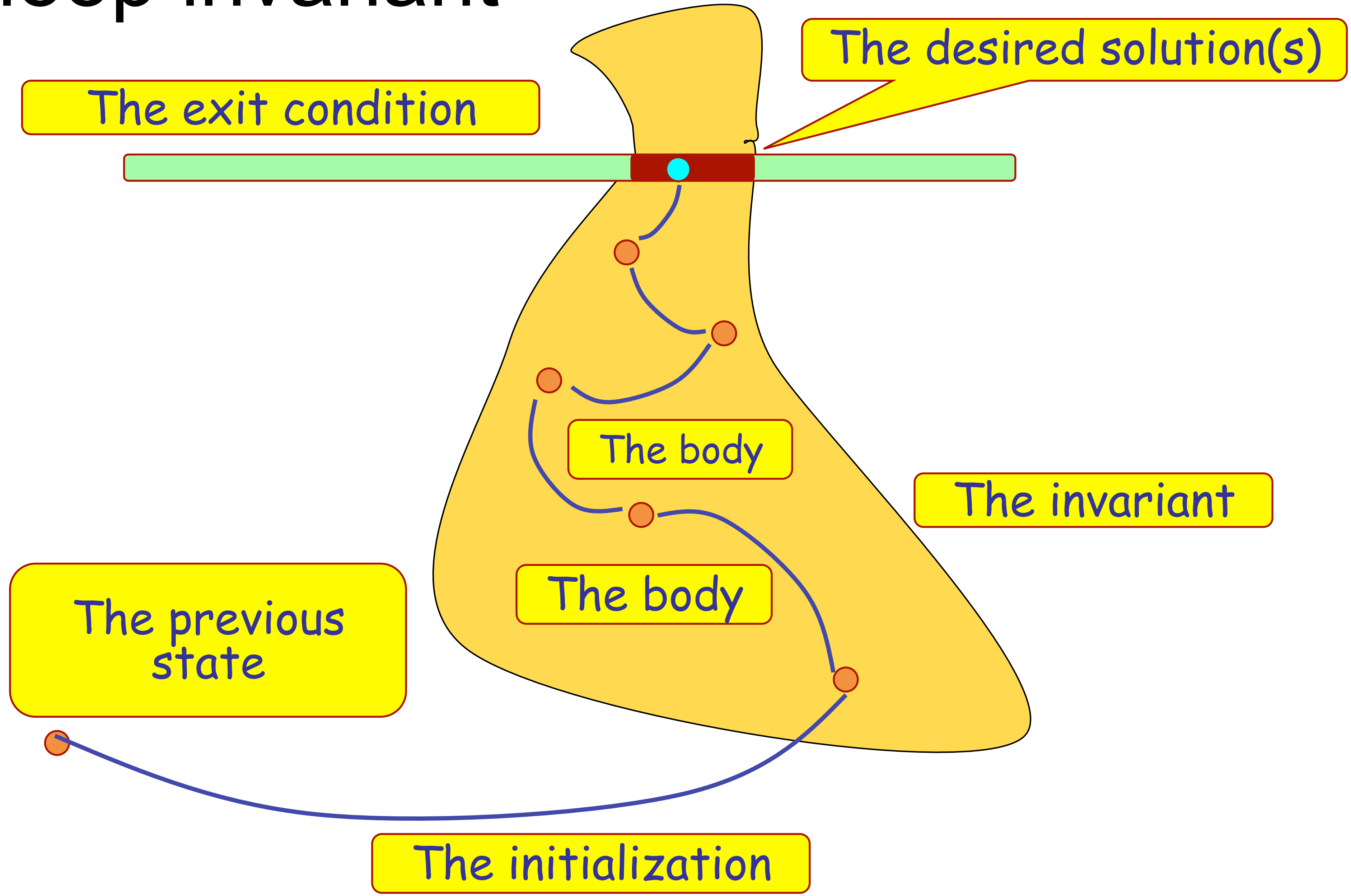
```
across                        -- Eiffel
        data_structure as var
loop
        Body -- Using var
end
```

```
while Condition do
        Body
end
```

Loop, full form

```
from
        Initialization              -- Compound
invariant
        Invariant_expression        -- Boolean_expression
variant
        Variant_expression          -- Integer_expression
until
        Exit_condition              -- Boolean_expression
loop
        Body                        -- Compound
end
```

he loop invariant



The solution

The solution is the **intersection** between the **invariant** and some condition (the **exit condition**)

Recap

- Invariants and variants will be the topic of a future class
    - Control Structures II

- So far
    - The need for control structures
    - The notion of algorithm
    - The notion of control structure
    - Correctness of an instruction
    - Control structure: sequence
    - Control structure: conditional
    - Nesting, and how to avoid it