Fondamenti della Programmazione: Metodi Evoluti

Prof. Enrico Nardelli

Lezione 9: Istruzioni di controllo

In this lecture

- Basic control structures: sequence, conditional, loop
- Decision structures: variants of conditional instruction
- Repeating operations: the loop
- Loops as approximation strategy: the loop invariant
- What does it take to ensure that a loop terminates?
- A look at the general problem of loop termination
- Lower-level control structures and the rationale for the "control structures of Structured Programming"
- Undecidability of the Halting Problem

The notion of algorithm

General definition:

> An **algorithm** is the specification of a process to be carried out by a computer

Not quite an algorithm

5 properties of an algorithm

1. Defines data to which process will be applied

2. Every elementary step taken from a set of well-specified actions

3. Describes ordering(s) of execution of these steps

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

5. For any data, guaranteed to terminate after finite number of steps

Algorithm vs program

"Algorithm" usually considered a more abstract notion, independent of platform, programming language etc.

In practice, the distinction tends to fade, since:

- Algorithms need a precise notation
- Programming languages becoming more abstract

However:

- In software systems, data (objects) are just as important as algorithms
- A software system typically contains many algorithms and object structures

What makes up an algorithm

Basic actions:

- Feature call $x.f(a)$
- Assignment
- …

(Actually, not much else!)

Sequencing of these basic actions:

CONTROL STRUCTURES

Control structures

A control structure

A program construct that describes the scheduling of basic actions

Three fundamental control structures:

- Sequence
- Loop
- Conditional

They are the

"Control structures of Structured Programming"

... more at the end of the lecture ...

Control structures as problem-solving techniques

Sequence:

"To achieve C from A, first achieve an intermediate goal B from A, then achieve C from B"

Loop:

solve the problem on successive approximations of its input set

Conditional:

solve the problem separately on two or more subsets of its input set

The sequence (or Compound)

instruction $_1$;

instruction $_2$;

... ;

instruction $_n$;

The empty instruction IS an instruction!

Conditional instruction (full form)

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

A variant of the conditional (short form)

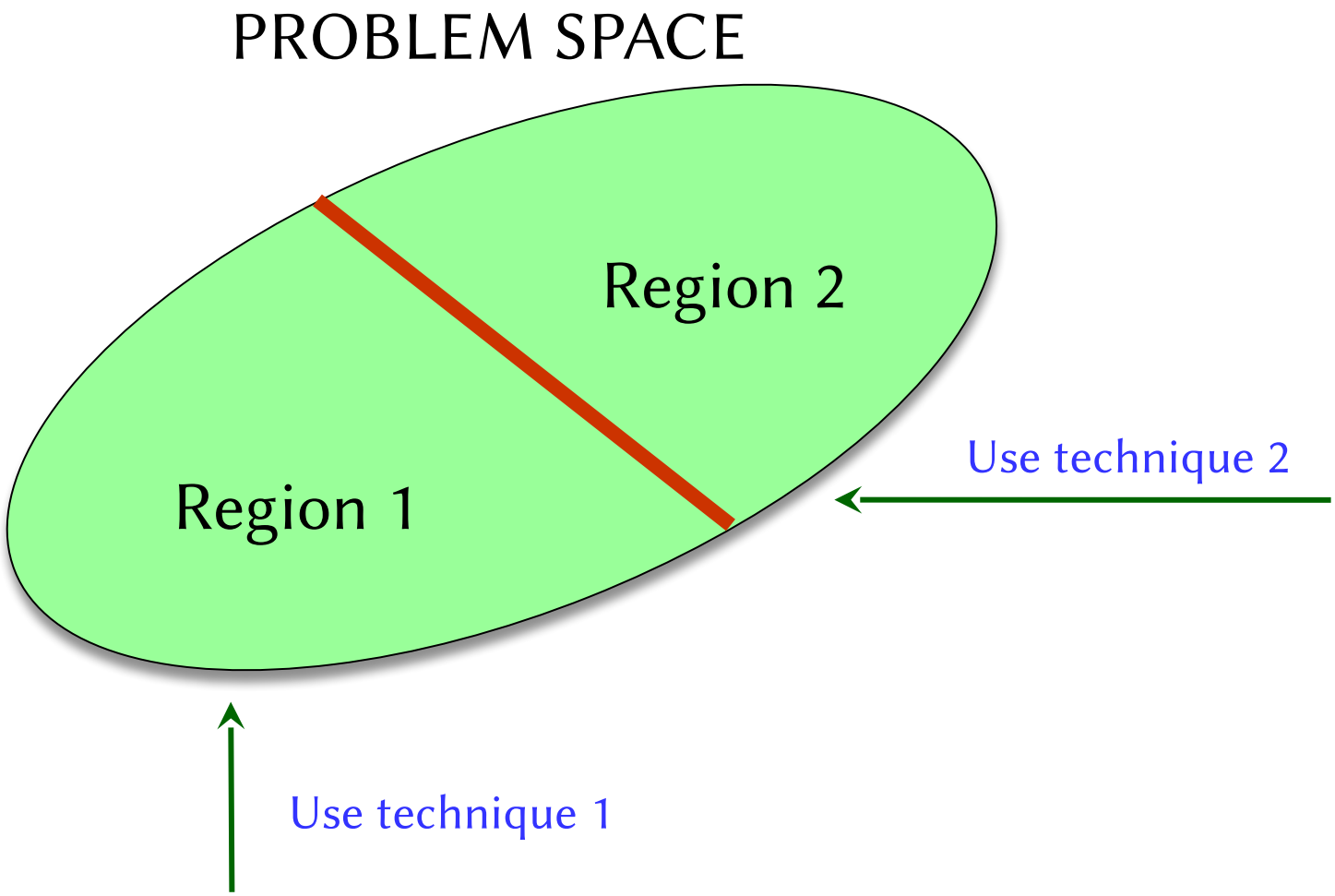
```
if Condition then
    Instructions
end
```

Means the same as

```
if Condition then
        Instructions
else

end
```

Empty clause

The conditional as problem-solving technique



A query computing the greatest of two integers

```
greater (a, b : INTEGER): INTEGER
            -- The higher of a and b.
    do
        if
            a > b
        then
            Result := a
        else
            Result := b
        end
    end
```

Typical use

i, j, k: *INTEGER*

...

$m :=$ *greater* $(25, 32)$

$n :=$ *greater* $(i + j, k)$

Better O-O style

In a class *DATE*:

Modeling a relevant concept

later (*d*: *DATE*): *DATE*

```
    -- The latest between Current and d.
    do
        if
            d.as_integer > Current.as_integer
        then
            Result := d
        else
            Result := Current
        end
    end
```

Feature call on a target

A possible variant

later (*d*: *DATE*): *DATE*

-- The latest between *Current* and *d*.

do

```
Result := Current
if d.as_integer > Current.as_integer then
        Result := d
end
```

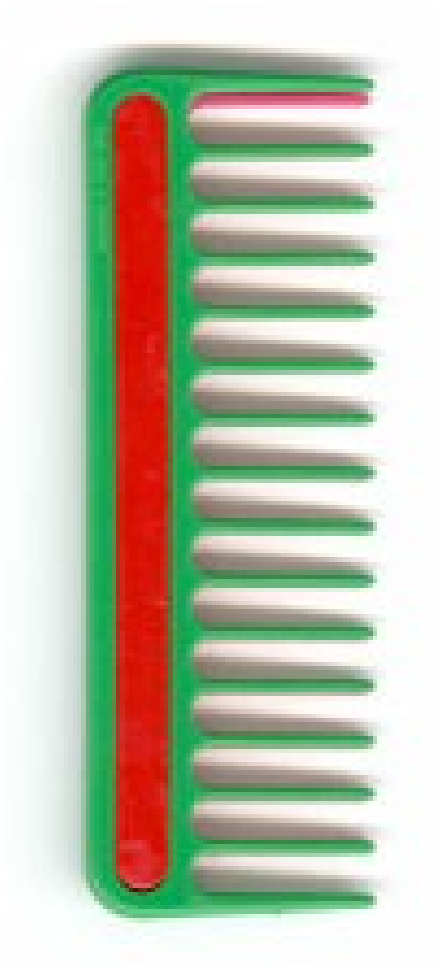
end

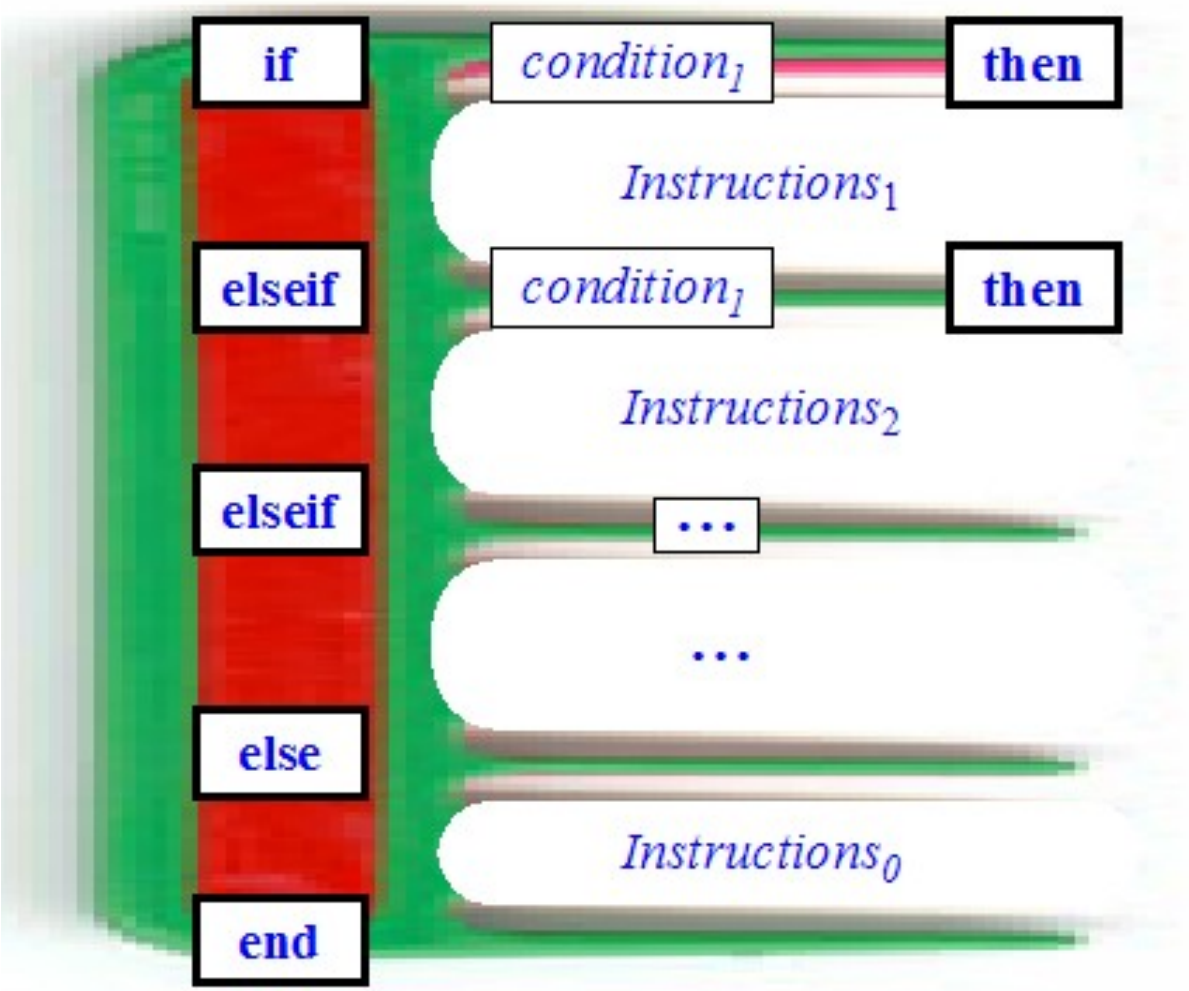
Nesting

```
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
```

Nested structure

Comb-like structure





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

EXACTLY one among *Instructions*$_i$ is executed even if more *Condition*$_i$ are true

When multiple *Condition*$_i$ are true the FIRST branch is taken

Independent conditionals

```
if Condition1 then
        Instructions1
end

if Condition2 then
        Instructions2
end

if Condition3 then
        Instructions3
end

if
        ...

end
```

Any *Instructions*$_i$ whose *Condition*$_i$ is true is executed

When multiple *Condition*$_i$ are true ALL branches are taken

Also available in Eiffel: «Inspect» (Multi-branch)

```
inspect
        choice
when "D" then
        Instruction_i
when "F", "H" then
        Instruction_j
when "K".."M" then
        Instruction_k
...
else
        Instruction_0
end
```

CHARACTER or INTEGER

Have to be disjoint, hence at most one is executed...

multiple choices

interval

Compiler will complain if not disjoint.

...or this is executed.

It's optional but a runtime exception is raised if missing and no other clause is used

Syntactical variations (1)

A **conditional expression** is an **expression** whose value depends on the result of a conditional instruction

```
a_greeting := if time < noon then
                  "Good morning"
              else
                  "Good afternoon"
              end
```

a ***detachable STRING***

Also with the **elseif** variant

```
a_greeting := if time < noon then
                  "Good morning"
              elseif time < evening then
                  "Good afternoon"
              else
                  Void
              end
```

if the various expressions do not have the same type their lowest common ancestor type is used

Syntactical variations (2)

Works also with the **inspect** instruction

```
a_textual_number := inspect a_number
                    when 1 then "one"
                    when 2 then "two"
                    else  "do not know"
                    end
```

More control structure topics

- Loops and their invariants
- See what it takes to ensure that a loop terminates
- Look at the general problem of loop termination
- Examine lower-level control structures: “Goto” and flowcharts; see rationale for the “control structures of Structured Programming”
- Prove the undecidability of the Halting Problem

Loop, short form

```
from
        Initialization          -- Compound

until
        Exit_condition                  -- Boolean_expression
loop
        Body                            -- Compound

end
```

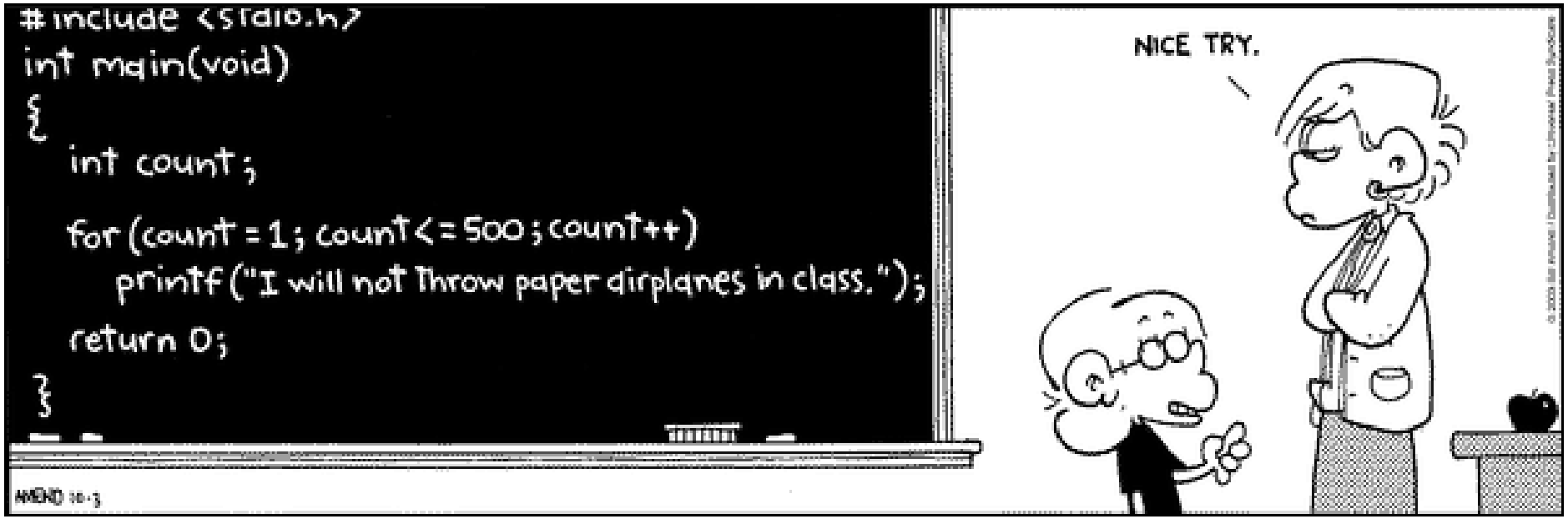
Loop, full form

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

Loop, full form (old syntax)

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

Another loop syntax



Forms of loop (in different languages)

```
from
        Instructions
until
        Condition
loop
        Instructions
end
```

```
while Condition do
        Instructions
end
```

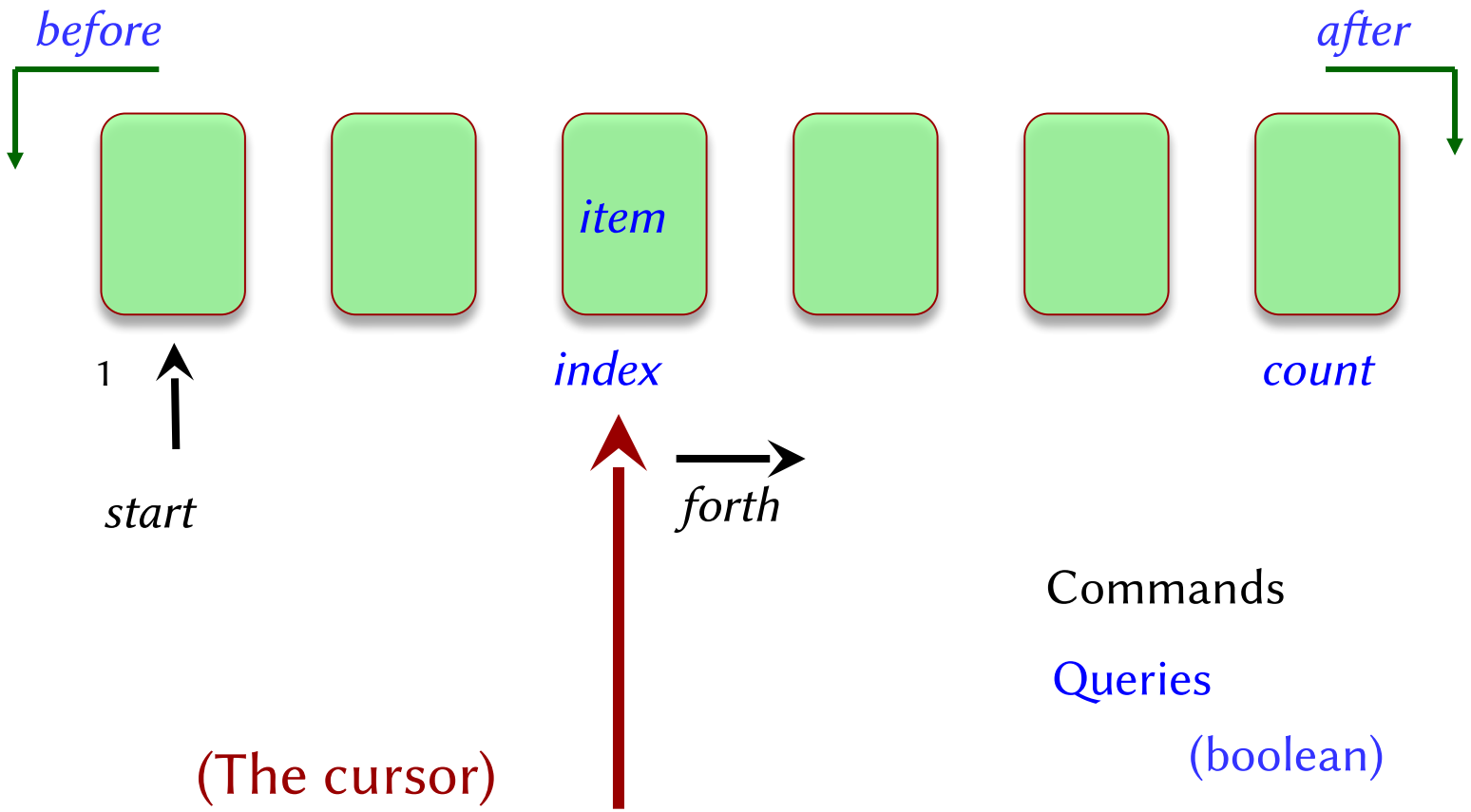
```
repeat
        Instructions
until
        Condition
end
```

```
for i : a..b do
        Instructions
end
```

```
for (Instruction; Condition; Instruction) do
        Instructions
end
```

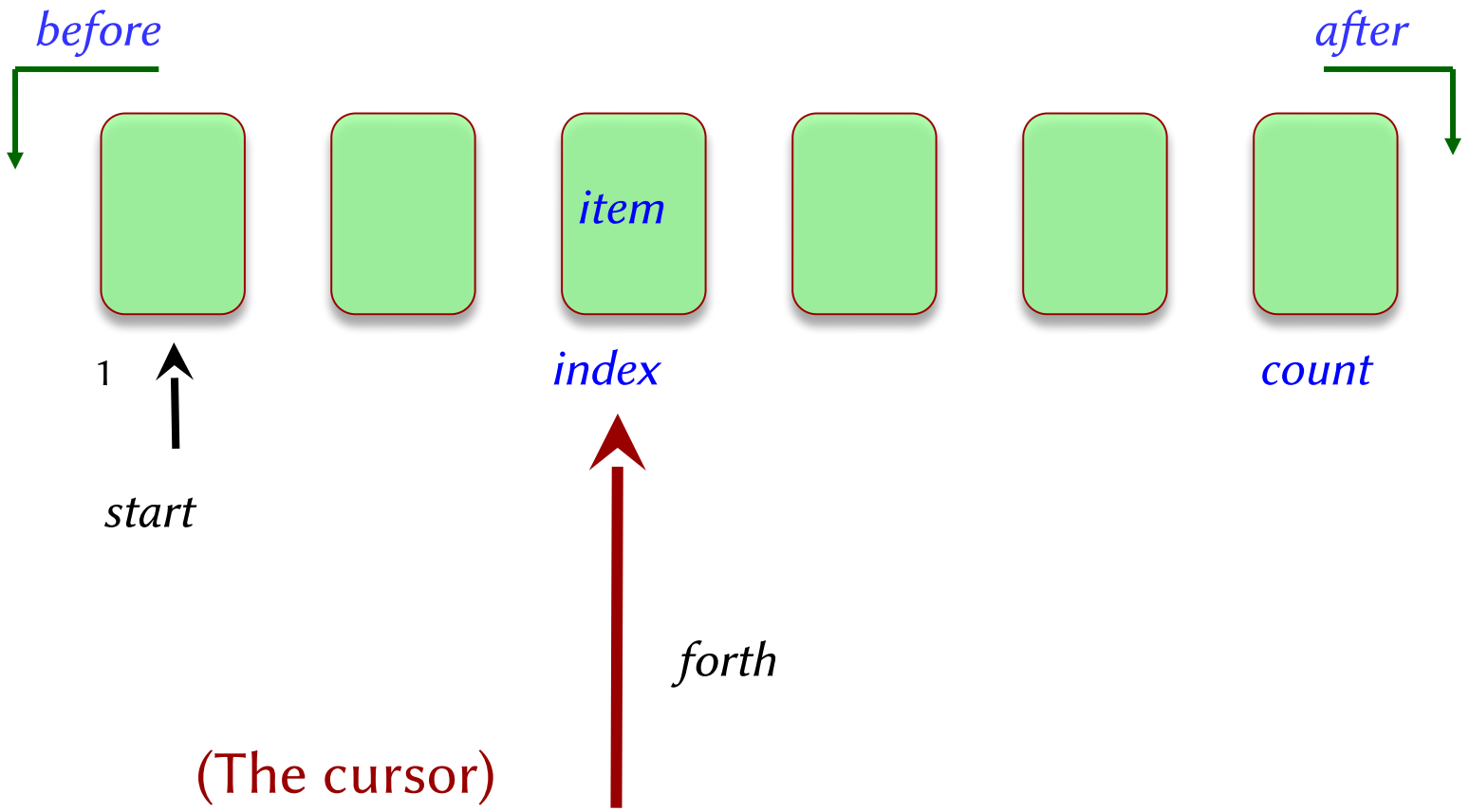
Operations on a list

LINKED_LIST (predefined class in Eiffel Studio)



Moving in a list

LINKED_LIST (predefined class in Eiffel Studio)



A problem with the cursor

For *my_list* instance of *LINKED_LIST*

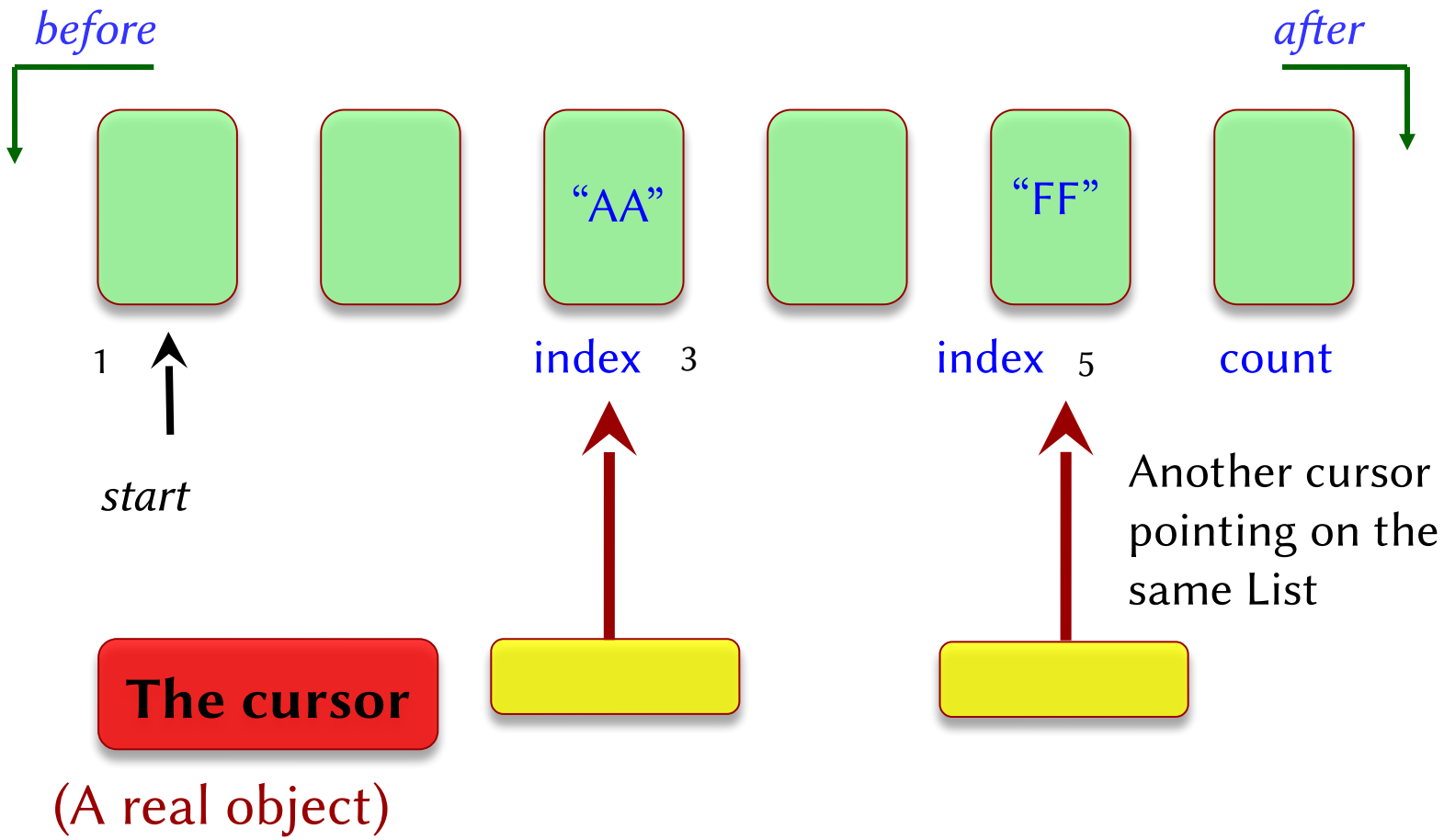
```
has_duplicates: BOOLEAN
        -- Has my_list duplicate values?
    local
        s : LINKABLE
    do
        from
            my_list.start
        until
            my_list.after  or Result
        loop
            s := my_list.item
            my_list.forth
            -- Check if s occurs again in the line:
            my_list.search (s)
            Result := not my_list.after
        end
    end
```

The position of the internal cursor must always be saved and restored

A feature looking for an element in a list

search moves the cursor to the position of *s* or to *after*

Adding an external cursor



Iterating over all items in a list (2)

Using an **external** cursor:

```
local
    c : like my_list.new_cursor
do
    from
        c := my_list.new_cursor
    until
        c.after
    loop
        -- "Do something with c.item"
        c.forth
    end
end
```

The cursor variable

The same type of *my_list.new_cursor*

A newly generated pointer pointing to the first element

Also the new cursor has this feature returning its status

The new cursor can be moved with *start* and *forth*

The solution with external cursor

```
has_duplicates: BOOLEAN
        -- Has my_list duplicate values?
    local
        s : LINKABLE
        c : like my_list.new_cursors
    do
        from
            my_list.start
        until
            my_list.after  or Result
        loop
            s := my_list.item
            my_list.forth
            -- Check if s occurs again in the line:
            c := my_list.cursor
            my_list.search (s)
            my_list.go_to (c)
            Result := not my_list.after
        end
    end
```

Save the cursor

Restore the cursor

Looping over all list items

For *my_list* instance of *LINKED_LIST*

```
from
        my_list.start
until
        my_list.after
loop
        -- "Do something with my_list.item"
        -- Display current item
        print(my_list.item)

        my_list.forth
end
```

A feature finding the maximum in a list of reals (1)

For *my_list* a list of *REAL*, instance of *LINKED_LIST*, and *REAL* has a *greater* feature

```
feature
    highest_value_in_list: REAL
    do
        from
            my_list.start
        until
            my_list.after
        loop
            Result := item.greater (Result)
            my_list.forth
    end
```

The greater of two values, e.g.

greater (8.5, 10.2) = 10.2

Iterating over all items in a list (3)

```
across
        my_list as ml
loop
        -- "Do something with ml.item"
end
```

The same effect as the previous version, but shorter!

Key points:

- iterates over **all** elements of the list
- defines a **new** cursor (allows multiple concurrent iterations of the same structure)

It's **not yet** part of the Eiffel Standard definition

A feature finding the maximum in a list of reals (2)

For *my_list* a list of *REAL*, instance of *LINKED_LIST*, and *REAL* has a *greater* feature

```
feature
    highest_value_in_list: REAL
    do
        across
            my_list as ml
        loop
            Result := ml.item.greater (Result)
    end
```

The greater of two values, e.g.

greater (8.5, 10.2) = 10.2

Comparison between the two versions

```
across
    my_list as ml

loop
    Result :=
        ml.item.greater (Result)

end
```

```
from
    my_list.start
until
    my_list.after
loop
    Result :=
        my_list.item.greater (Result)
    my_list.forth
end
```

Expressions iterating over all elements of a list

```
across
        my_list as ml
loop
        -- "Do something with ml.item"
end
```

Remember it's an **instruction**, but it has also two **expression** variants yielding a boolean value (only for objects of *ITERABLE* type):

across *my_list* **as** *ml* **all** *ml.item* > 0 **end**

true if and only if **all** elements are positive.

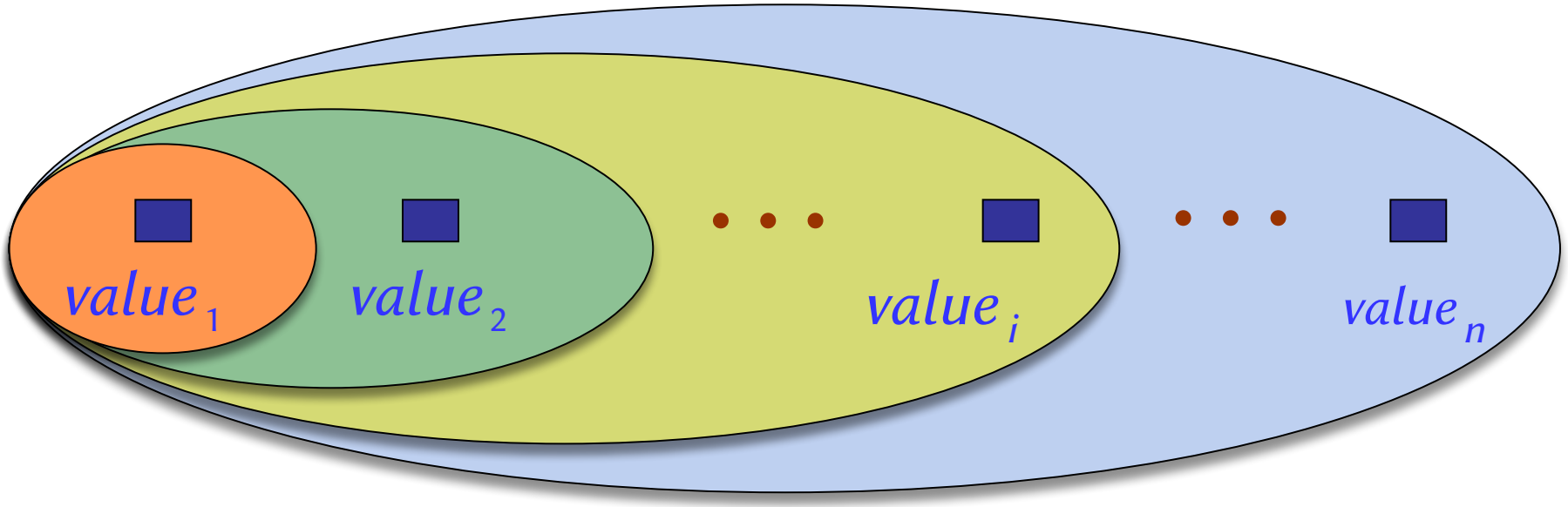
across *my_list* **as** *ml* **some** *ml.item* > 0 **end**

true if and only if **at least one** is positive.

Useful in invariants, but also in other contexts

Remember these are expressions and cannot stand on their own

Loop as approximation strategy



Loop body:

Result := *my_list.item.greater* (**Result**)
i := *i* + 1

Result = *value* 1 = Max (*values* 1..1)

Result = Max (*values* 1..2)

Slice

Result = Max (*values* 1..*i*)

Result = Max (*values* 1..*n*)

The loop invariant

Computing the maximum: postcondition?

```
do
        from
                my_list.start

        until
                my_list.after
        loop
                Result := my_list.item.greater (Result)

                my_list.forth
        end

ensure
        -- Result is the greatest among all values

end
```

How to implement it?

Computing the maximum: invariant?

```
do
        from
                my_list.start

        until
                my_list.after
        loop
                Result := my_list.item.greater (Result)

                my_list.forth
        end
ensure
        across my_list as ml  all  ml.item.value <= Result
end
```

Invariant ???

Loop invariant

(Do not confuse with class invariant)

It is a property that:

- Is true after initialization (**from** clause)
- Is preserved by every loop iteration (**loop** clause), i.e. is true at the end of each loop iteration
- Hence it's true also ***after*** the last loop iteration
- Ensures the desired result when the exit condition (**until** clause) *becomes* true

The loop invariant

What is always true for the set of previous values?

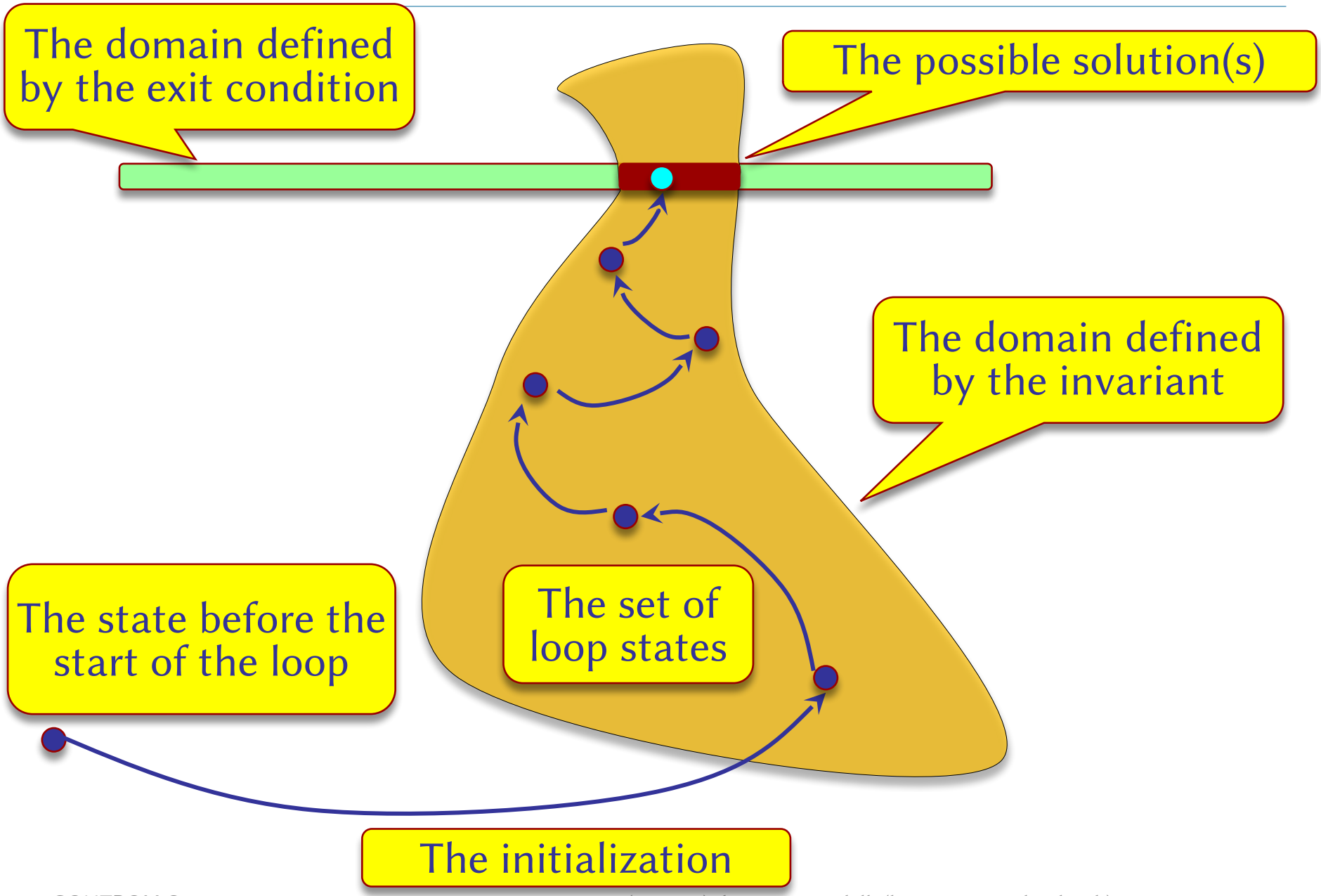
How to parametrically describe the set of previous values?

```
from
    my_list.start
invariant
        -- Result is the greatest among previous values
    my_list.index >= 1
    my_list.index <= my_list.count + 1
until
    my_list.after
loop
    Result := my_list.item.greater (Result)

    my_list.forth
end
```

Something is missing?

```
ensure
        across my_list as ml  all  ml.item <= Result
```

How the loop invariant works



The loop invariant

equivalent to **Result** = Max (*values* 1 . . *index-1*)

```
from
    my_list.start
invariant
        -- Result is the greatest among previous values
    my_list.index >= 1
    my_list.index <= my_list.count + 1
until
    my_list.after
loop
    Result := my_list.item.greater (Result)

    my_list.forth
end
```

```
ensure
    across my_list as ml  all  ml.item <= Result
```

The effect of the loop

```
from
        my_list.start
invariant
        -- Result is the greatest among previous values
        my_list.index >= 1
        my_list.index <= my_list.count + 1
until
        my_list.after
loop
        Result := my_list.item.greater (Result)

        my_list.forth
end
```

Invariant satisfied after initialization

equivalent to **Result** = Max (*values* $_{1\,..\,index-1}$)

Invariant satisfied after each iteration

Exit condition satisfied at end

At the end: invariant **and** exit condition implies:

- all items visited (*after*)
- **Result** is the greatest value

How do we know a loop terminates?

```
from
        my_list.start
invariant
        my_list.index >= 1
        my_list.index <= my_list.count + 1
        -- If there is any previous item,
        -- Result is the greatest of them
until
        my_list.after
loop
        Result := my_list.item.greater (Result)

        my_list.forth

end
```

Loop variant

Integer expression that must:

- Be non-negative after initialization (**from**)
  - it's always checked after initialization even if no execution of the body takes place
- **Decrease** (i.e. by at least one), while remaining non-negative, after every iteration of the body (**loop**)
- Be non-negative right after exit (**until**)
- Nice if it's 0 (zero) after exit but it's NOT needed

The variant for our example

```
from
        my_list.start
invariant
        my_list.index >= 1
        my_list.index <= my_list.count + 1
        -- If there is any previous item,
        -- Result is the greatest of them
until
        my_list.after
loop
        Result := my_list.item.greater (Result)

        my_list.forth
variant

        my_list.count – my_list.index  + 1
end
```

The most general form for a loop instruction

across
expression **as** *identifier*
from
compound

Requires an *ITERABLE*

At least one is needed, both may be present

invariant
assertion

optional

until
boolean_expression
loop
compound

Required in absence of **across**, may be used with **across**

Always present

variant
integer_expression

optional

An example of LOOPing across ITERABLEs

For class *INTEGER* the operator |..| provides an interval, e.g. 6 |..| 9 denotes the closed interval 6,7,8,9

An interval of integers is an *ITERABLE*

Hence it's possible to iterate across integer intervals like in this example

```
across 6 |..| 9 as ic
loop
        print (ic.item.out + "%N")
end
```

Another example of LOOPing across ITERABLEs

Cursors of *LINKED_LIST* are also *ITERABLE*

Then it's possible to use them to iterate across lists in ways different from the standard one.

For *my_list* a list of *STRING* istance of *LINKED_LIST* we can write

```
across my_list.new_cursor.reversed as ml
loop
        print (ml.item + "%N")
end
```

to print the string in the list in reversed order

An example using both across and from

For *my_list* a list of STRING we want to count how many strings there are **before** a "stop" string

```
across my_list as ml
from
        sum := 0
until
        ml.item ~ "stop"
loop
        sum := sum + 1
end
```

The presence of **across** avoid the need to explicitly write **start**, **after**, and **forth**

What if "stop" is not found?

Combines the automatic iteration across the entire list with the possibility of an early exit

The most general form for a loop expression

across
expression **as** *identifier*

Requires an *ITERABLE*

all
boolean_expression
some
boolean_expression

Only one of these

What happens at the machine level?

Unconditional branch:

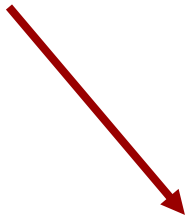
BR *label*

Conditional branch:

BZ *label_true* *label_false*

The equivalent of if-then-else

if *a* = 0 **then** *Compound_1* **else** *Compound_2* **end**



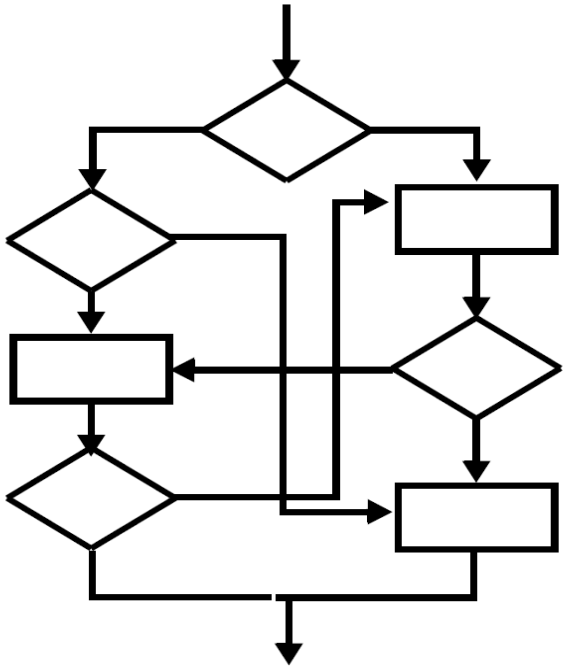
```
062     BZ  111  082

082     ... Code for Compound_2 ...
        BR 125

111     ... Code for Compound_1 ...

125     ... Code for continuation of program ...
```

“Goto considered harmful”

Dijkstra, 1968

Arbitrary Goto instructions lead to messy, hard to maintain programs (“spaghetti code”)

Böhm and Jacopini theorem: any program that can be expressed with **goto** instructions and conditionals can also be expressed without **goto**s, using sequences and loops



For an example of transformation see *Touch of Class*

The Goto today

Almost universally decried

Still exists in some programming languages

Also hides under various disguises, e.g. **break**

```
for
        ...
        if c then break end
        ...
end
```

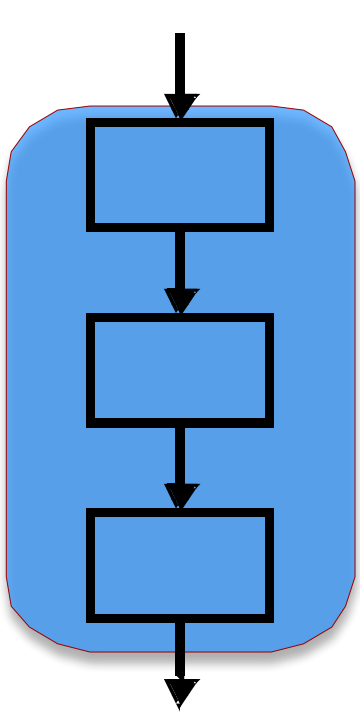
Stay away from **goto** in any form!

In programming languages: the Goto

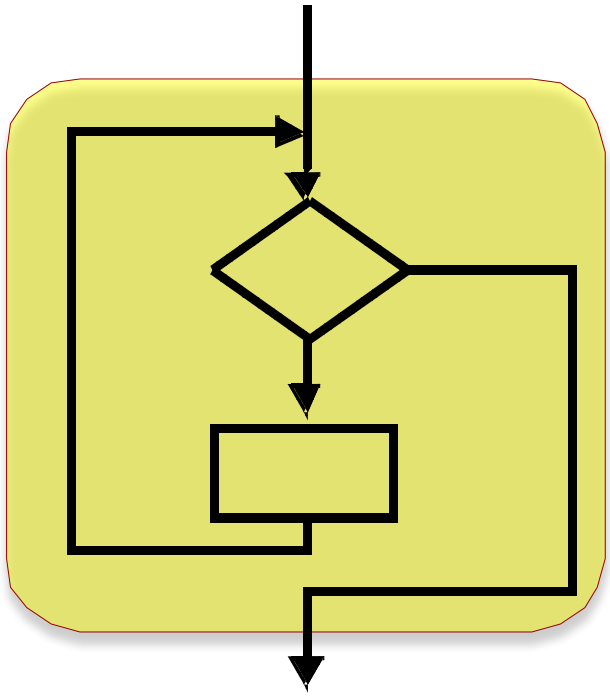
```
                test condition goto else_part
                Compound_1
                goto continue
else_part :     Compound_2
continue :      ... Continuation of program ...
```

One-entry, one-exit

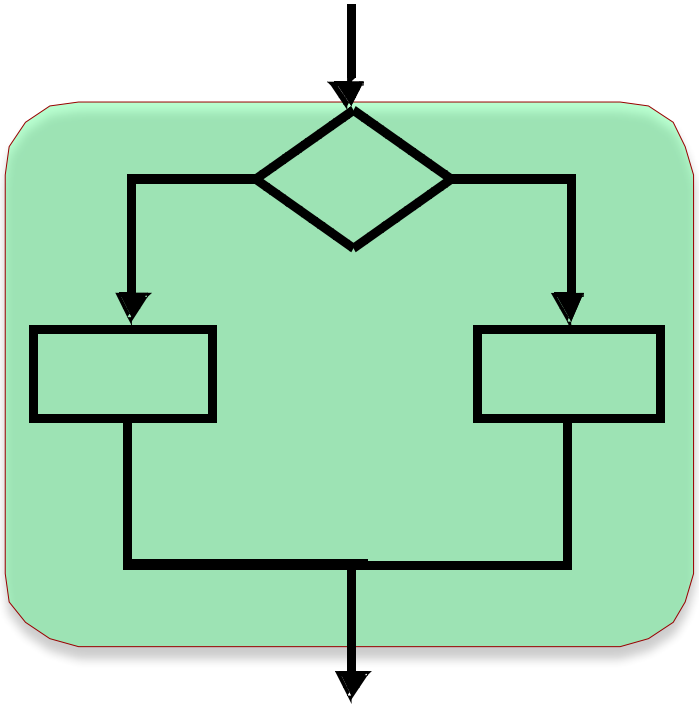
Always use 1-entry 1-exit control structures!



(Compound)



(Loop)



(Conditional)

The general termination problem

Can EiffelStudio find out if your program will terminate?

No, it can't ☹

No other program, for any other realistic programming language, can! ☹☹☹

The halting problem and undecidability

("*Entscheidungsproblem*", Alan Turing, 1936.)

It is **not** possible to devise an effective procedure that will find out if an arbitrary program will terminate on arbitrary input

(or if an arbitrary program with no input will terminate)

The halting problem in Eiffel (1)

Assume we have a feature in a root class *TURING*

```
terminates (file_name : STRING): BOOLEAN
        -- Does program in file file_name terminate?
            do
                ... Your algorithm here ...
            end
```

It returns **true** if program stored in *file_name* terminates

And then ...

The halting problem in Eiffel... (2)

Write the following root procedure for the class *TURING* stored in /usr/home/turing.e:

```
what_do_you_think
        -- Terminate only if not.
do
    from
    until
        not terminates ("/usr/home/turing.e")
    loop
    end
end
```

What would happen when you run it?

Paradoxes in logic

The Russell paradox

- Some sets are members of themselves, e.g. the set **T** of all infinite sets is infinite, i.e. **T** contains **T**
- Others are **not** members of themselves, e.g. the set **F** of all finite sets is not finite, i.e. **F** does not contain **F**
- Consider now the set **S** of **all** sets that are **not** members of themselves
  - Is **S** inside **S**? If yes its definition would be wrong (the **not** part)
  - Is **S** outside **S**? If yes its definition would be wrong (the **all** part)

The barber paradox (Russell)

- In various cities some inhabitants do their own hair and some use one of the many hairdressers
- In Rome, while some inhabitants do their own hair, there is a single hairdresser, who is defined as the person who does the hair of all the inhabitants who do not do their own hair
- Who does Rome hairdresser's hair?

The liar's paradox

The oldest (600 aC) version (but not a paradox!)

Epimenides is Cretan and says: "All Cretans are liars"

If we assume Epimenides' sentence is true

then "All Cretans are liars" would imply Epimenides (a Cretan!) is a liar, hence Epimenides' sentence would be false: a contradiction.

However, if we assume Epimenides' sentence is false, this woud imply some Cretans tell the truth (not Epimenides!) and some are liars (Epimenides!)

Hence Epimenides' sentence has a non-contradictory interpretation

A later (400 aC) version (a true paradox!)

A person says: "I am lying".

If the sentence is true then she is lying, then the sentence "I am lying" must be false,

then the person is telling the truth, hence she is not lying: a contradiction

If the sentence is false then she is not lying, then the sentence "I am lying" must be true,

then the person is not telling the thruth, hence she is lying: a contradiction

The Grelling paradox

An adjective in English is defined to be:

- "Autological" if it holds for itself / describes itself (e.g. "polysyllabic")
- "Heterological" if it does NOT describe itself

What is the status of adjective "Heterological"?

- It should be "Autological" but it cannot be "Autological" by its definition
- It cannot be "Heterological" by its definition

Other forms: are the following sentences true or not?

This sentence does not speak about itself

It's false that this sentence appears in red

This sentence is false

The halting problem in practice

Some programs must not terminate

e.g.: operating systems

Some programs should terminate in all cases

If they don't … that's a bug!

Use **variant** to ensure loop termination

The undecidability of the halting problem does not prevent to prove that a **specific** program can terminate

What we have seen

Iteration in Eiffel

- Loop
- Across
- Inspect

GoTo and structured programming

Undecidability of the halting (and some paradoxes)