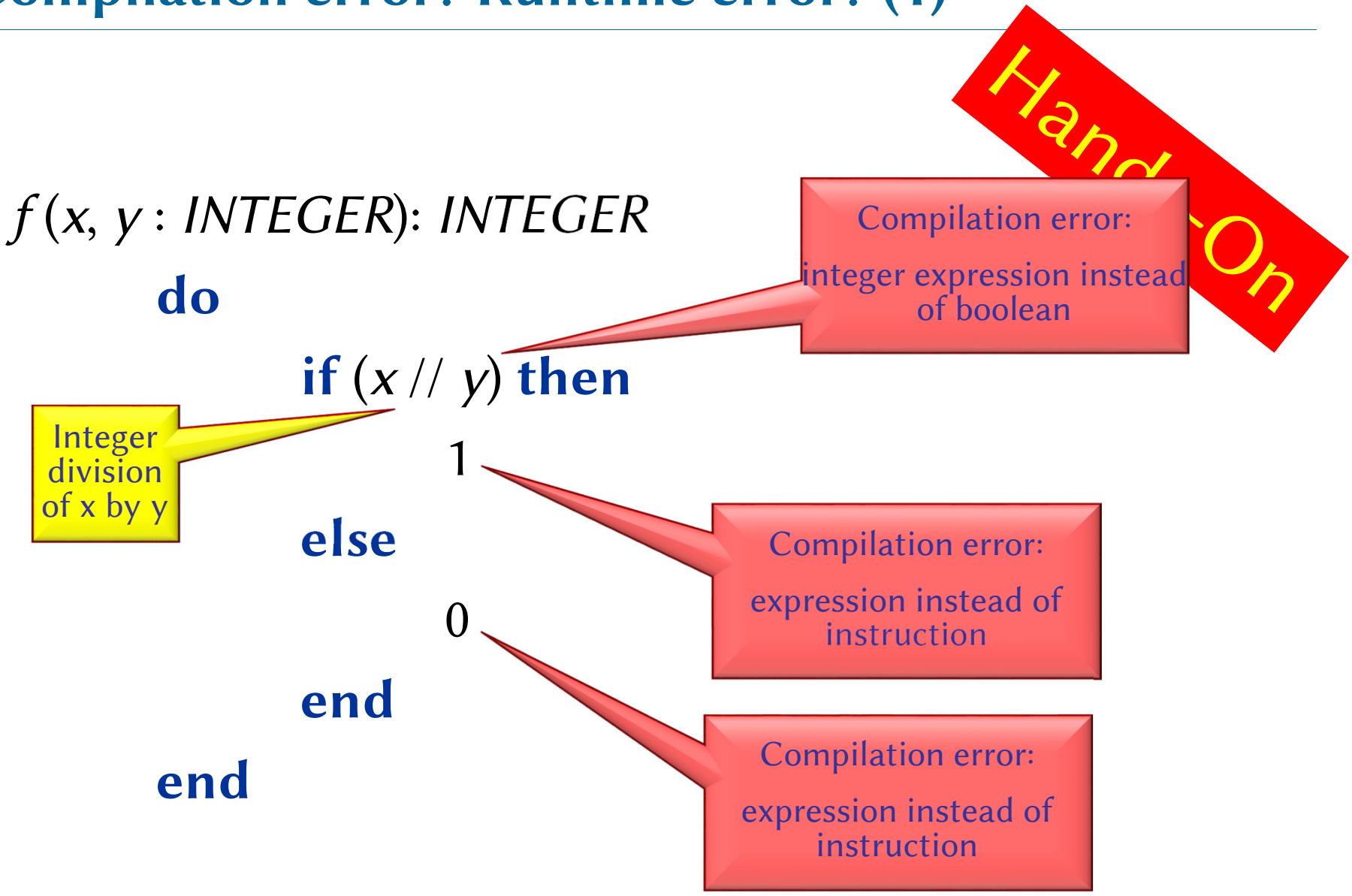
Fondamenti della Programmazione: Metodi Evoluti

Prof. Enrico Nardelli

Esercitazione 5

Compilation error? Runtime error? (1)



Compilation error? Runtime error? (2)


```
f (x, y : INTEGER): INTEGER
        do
                if (False) then
                        Result := x // y
                end
                if (x /= 0 ) then
                        Result := y // x
                end
        end
```

Everything is OK

(during both compilation and runtime)

Calculating function's value


```
f (max : INTEGER; s : STRING): STRING
        do
                if s.is_equal ("Java") then
                        Result := "J**a"
                else
                        if s.count > max then
                                Result := "<an unreadable German word>"
                        end
                end
        end
```

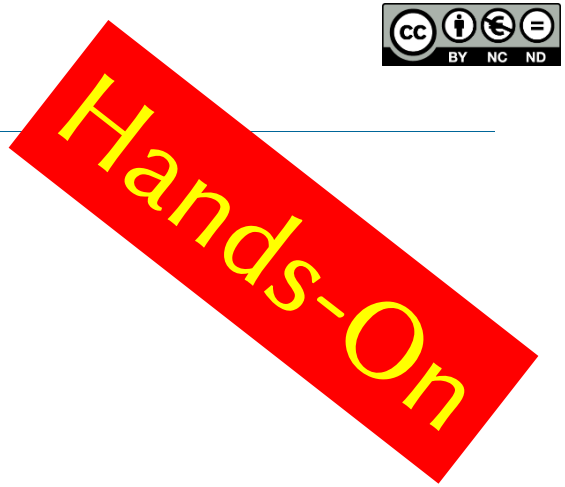
Calculate the value of:

- f(3, "Java") → "J**a"
- f(20, "Immatrikulationsbestätigung") → "<an unreadable German word>"
- f(6, "Eiffel") → **Void**

What does this routine do?


```
abs (x : REAL): REAL
        do
                if (x >= 0) then
                        Result := x
                else
                        Result := -x
                end
        end
```

Write a routine...



➢ ... that increases time by one second inside class *TIME*:

```
class TIME
        hour, minute, second : INTEGER

        second_forth
                do ... end

        ...
end
```

Increasing seconds...

```
class TIME
    second_forth
        do
            if second = max_second then
                second := 0
                minute_forth
            else
                second := second + 1
            end
        ensure
            second = (old second ) + 1 or else second = 0
        end
end
```

Increasing minutes... and so on...

```
class TIME
    minute_forth
        do
            if minute = max_minute then
                minute := 0
                hour_forth
            else
                minute := minute + 1
            end
        ensure
            minute = (old minute) + 1 or else minute = 0
        end
end
```

Simple loop (1)

Hands-On

How many times will the body of the following loop be executed?

i: *INTEGER*

...

from

    i := 1

until

    i > 10

loop

    print ("I will not say bad things about assistants")

    i := *i* + 1

end

In Eiffel we usually start counting from 1

10

Simple loop (2)

Hands-On

And what about this one?

i : *INTEGER*

...

from

    i := 10

until

    i < 1

loop

    print (“I will not say bad things about assistants”)

end

Caution!

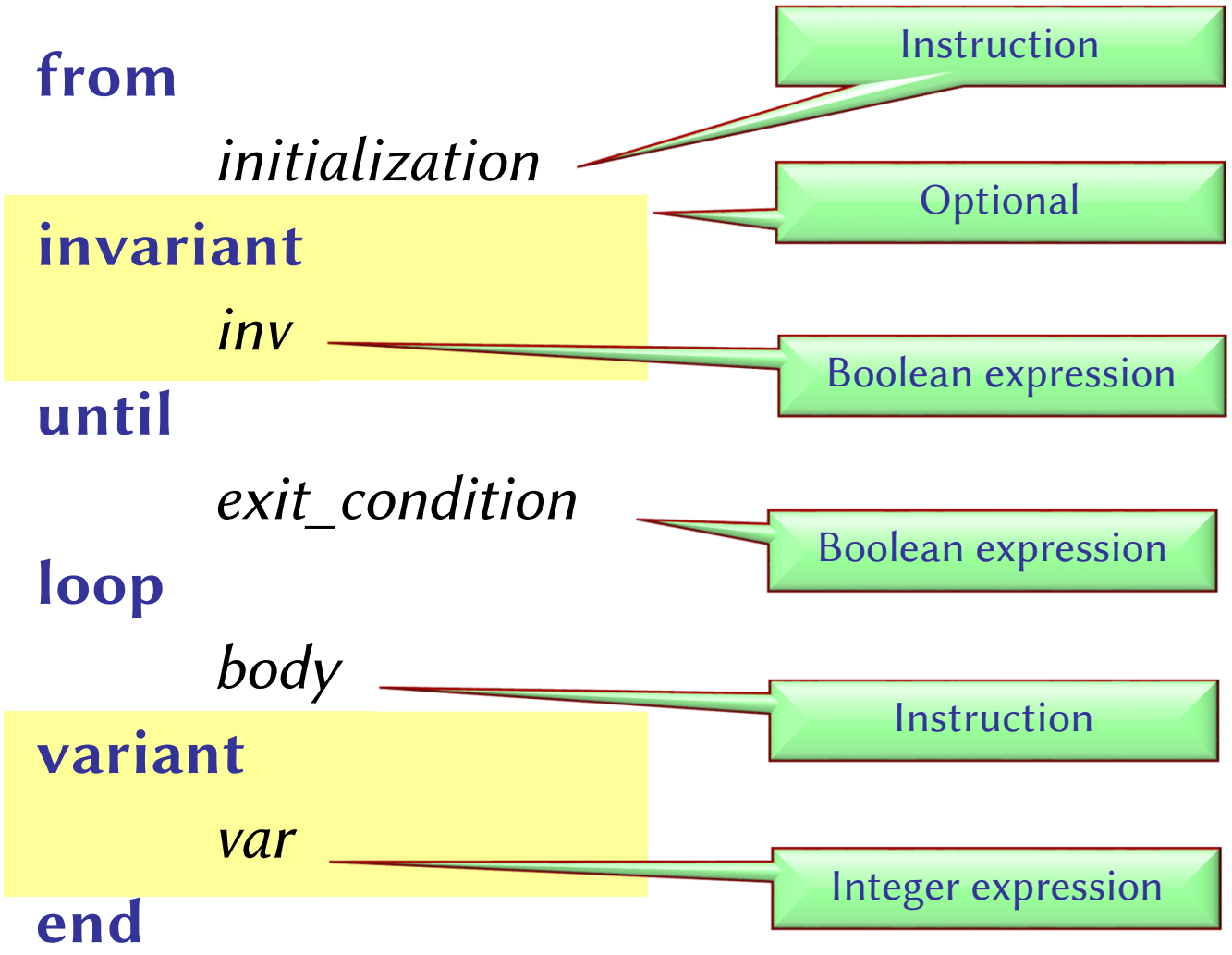
Loops can be infinite!

What does this function do?


```
factorial (n : INTEGER): INTEGER
            local
                    i : INTEGER
            do
                    from
                            i := 2
                            Result := 1
                    until
                            i > n
                    loop
                            Result := Result * i
                            i := i + 1
                    end
            end
```

Loop, full form

Syntax:



Invariant and variant

Loop invariant (do not mix with class invariant)

- holds after execution of **from** clause and after each execution of **loop** clause
- captures how the loop iteratively solves the problem: e.g. "to calculate the sum of all *n* elements in a list, on each iteration *i* (*i* = 1..*n*) the sum of first *i* elements is obtained"

Loop variant

- integer expression that is nonnegative after execution of **from** clause and after each execution of **loop** clause and strictly decreases with each iteration
- a loop with a variant can not be infinite (why?)

Invariant and variant

Hands-On

What are the invariant and variant of the "factorial" loop?

```
from
        i := 2
        Result := 1
invariant
        Result = factorial (i - 1)
until
        i > n
loop
        Result := Result * i
        i := i + 1
variant
        n − i + 2
end
```

Result = 6 = 3!

When features are used in contracts, ***their*** contracts are **NOT** checked

$n - i + 1$
it's enough, if we call factorial with $n > 0$

Contracts

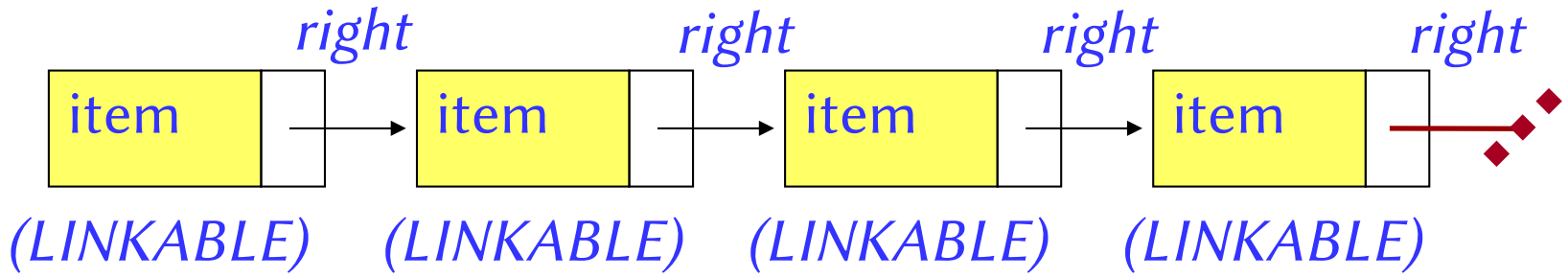
Which are the contracts?

```
require
        n >= 0
do
        from
                i := 2
                Result := 1
        invariant
                Result = factorial (i - 1)
        until
                i > n
        loop
                Result := Result * i
                i := i + 1
        variant
                n − i + 2
        end
ensure
        n > 0  implies Result = n *  factorial (n - 1)
end
```

LINKABLE (a predefined Eiffel class)

To make it possible to link infinitely many similar elements together, each element should contain a reference to the next element

```
class LINKABLE
        feature
                ...
                right : LINKABLE
end
```



INT_LINKABLE

```
class INT_LINKABLE
create set_value
feature -- Access
        value : INTEGER
        next : detachable INT_LINKABLE
feature -- Initialization
        set_value (a_value : INTEGER)
                do value := a_value
                ensure value = a_value
                end
feature -- Processing
        link_to (other : detachable INT_LINKABLE)
                -- connect to `other'
                do next := other
                ensure next = other
                end
end
```

No need of requiring *other* /= **Void**

INT_LINKED_LIST

```
class INT_LINKED_LIST
feature -- Access
        first_element : detachable INT_LINKABLE
                -- First element of the list

        last_element : detachable INT_LINKABLE
                -- Last element of the list

        active_element : detachable INT_LINKABLE
                -- Current element of the list

        count : INTEGER
                -- Number of elements in the list
        ...
invariant
        ...
end
```

INT_LINKED_LIST

```
class INT_LINKED_LIST
feature – Cursor movement
    start
        -- Set `active_element' to the first element
        do
            active_element := first_element
        ensure active_element = first_element
        end
    last
        -- Set `active_element' to the last element
        do
            active_element := last_element
        ensure active_element = last_element
        end
    forth
        -- Set `active_element' to the next element, if exists
        do
            if attached active_element as ae then
                active_element := ae.next
            end
        ensure attached old active_element as ae implies active_element = ae.next
        end
end
```

INT_LINKED_LIST class invariant

Is it Void Safe?

invariant

count >= 0

last_element /= **Void implies** *last_element.next* = **Void**

...

...

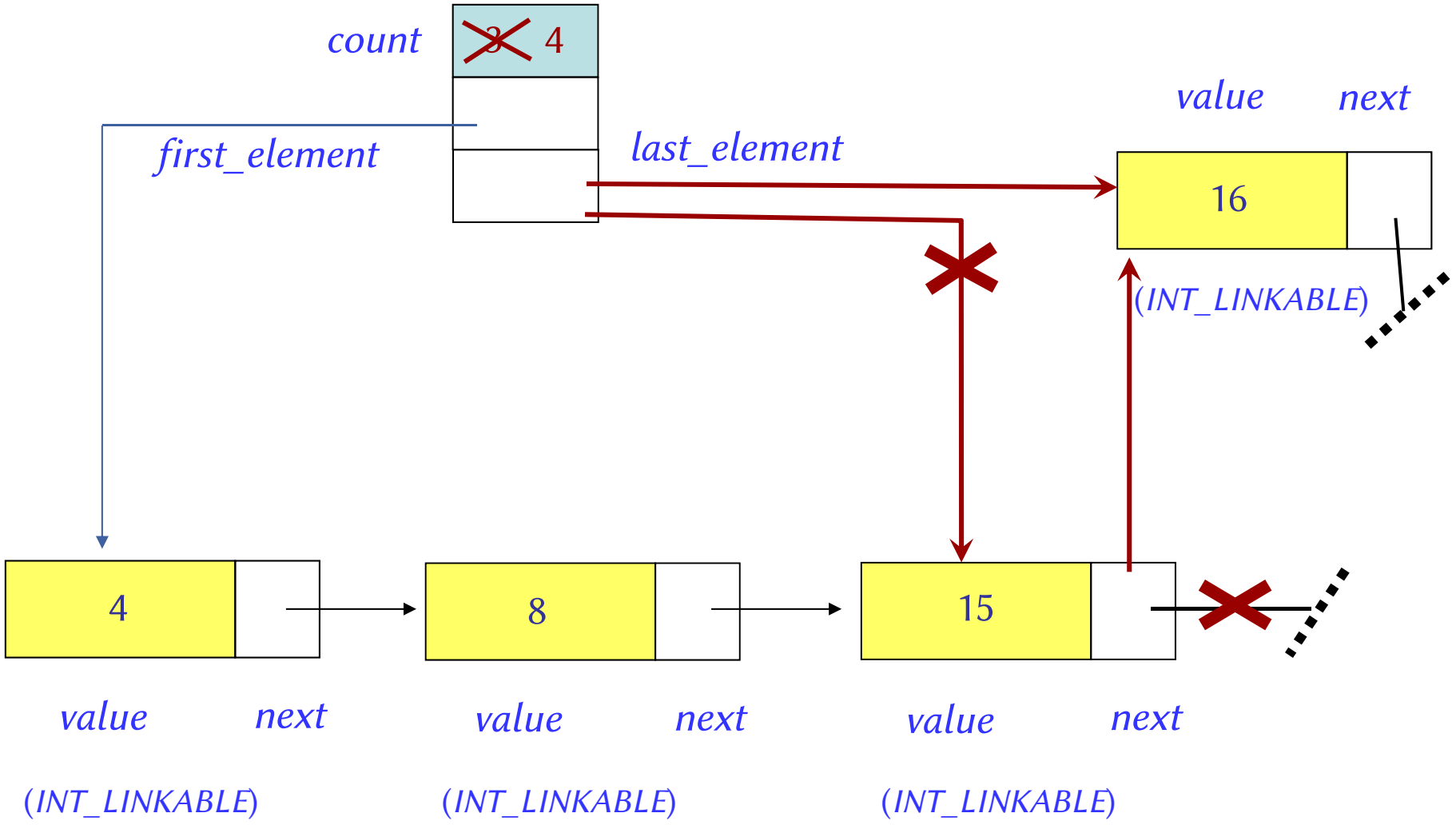
...

end

INT_LINKED_LIST class invariant


```
invariant
    count >= 0
    attached last_element as le implies le.next = Void
    count = 0  implies
        (first_element = last_element) and
        (first_element = Void) and
        (first_element = active_element)
    count = 1  implies
        (first_element = last_element) and
        (first_element /= Void) and
        (attached active_element as ae implies ae = first_element)
    count = 2  implies
        (first_element /= last_element) and
        (first_element /= Void) and
        (last_element /= Void) and
        (attached active_element as ae implies ae = first_element or ae = last_element) and
        (attached first_element as fe implies fe.next = last_element)
    count > 2  implies
        (first_element /= last_element) and
        (first_element /= Void) and
        (last_element /= Void) and
        (attached first_element as fe implies fe.next /= last_element)
end
```

INT_LINKED_LIST: inserting at the end



INT_LINKED_LIST: inserting at the end

```
append (a_value : INTEGER)
    -- Add `a_value' after `last_element'.
local
    new_element: like first_element
do
    create new_element.set_value (a_value)
    if count = 0 then
        first_element := new_element
    else
        if attached last_element as le then
            le.link_to (new_element)
        end
    end
    last_element := new_element
    count := count + 1
ensure
    count = old count + 1
    attached last_element as le implies le.value = a_value
    attached old last_element as ole implies ole.next = last_element
end
```

Is it

Void Safe?

INT_LINKED_LIST: searching for an item

```
has (a_value: INTEGER): BOOLEAN
        -- Does list contain `a_value '?
    local
        previous_element, current_element : like first_element
    do
        from
            current_element := first_element;
            previous_element := Void
        invariant
            not Result implies
                (previous_element /= Void implies
                        previous_element.value /= a_value )
            Result implies
                (previous_element /= Void implies
                        previous_element.value = a_value )
        until
            (current_element = Void) or Result
        loop
            if current_element.value = a_value then
                Result := True
            end
            previous_element := current_element
            current_element := current_element.next
        end
    end
```

For the **variant** contract
k: INTEGER

k := 0

k := *k*+1

variant
count - k

Is it

Void Safe?

INT_LINKED_LIST: finding an item

```
get_element (a_value: INTEGER): detachable INT_LINKABLE
    -- Return the first element containing `a_value', if exists
local
    current_element, previous_element : like first_element
do
    from
        current_element := first_element;
        previous_element := Void
    invariant
        previous_element /= Void implies previous_element.value /= a_value
    until
        current_element = Void or else current_element.value = a_value
    loop
        previous_element := current_element
        current_element := current_element.next
    end
    Result := current_element
ensure
    Result /= Void implies Result.value = a_value
    Result = Void implies not has(a_value)
end
```

Is it

Void Safe?

For the **variant** contract introduce the same changes as in previous slide

INT_LINKED_LIST

Write a routine that

- inserts value *a_value* after the first occurrence of a value *target* and add *a_value* at the end if the value *target* is not found
- do not reuse existing feature *has* and *get_element* (generally not a good choice!)

insert_after (*a_value*, *target*: *INTEGER*)

do ... **end**

INT_LINKED_LIST: insert_after (1)

```
insert_after (a_value, target : INTEGER)
    -- Insert `a_value' right after first occurrence of `target', if exists,
    -- otherwise add `a_value' after `last_element'.
    local
        current_element, new_element : like first_element
    do
        create new_element.set_value (a_value)
        from current_element := first_element
        until current_element = Void or else current_element.value = target
        loop current_element := current_element.next
        end
        if current_element /= Void then
            new_element.link_to (current_element.next)
            current_element.link_to (new_element)
            if current_element = last_element then
                last_element := new_element
            end
        else -- list does not contain `target'
            ...
        end
        count := count + 1
    ensure
        count = old count + 1
        not old has(target) implies
                (attached last_element as le implies le.value = a_value)
        old has(target) implies (attached get_element(target) as ge implies
                (attached ge.next as gen implies gen.value = a_value) )
    end
```

Refactor into feature *link_after*

INT_LINKABLE (final version)

```
class INT_LINKABLE
create set_value
feature -- Access
        value : INTEGER
        next : detachable INT_LINKABLE
feature -- Initialization
        set_value (a_value : INTEGER)
                do value := a_value
                ensure value = a_value
                end
feature -- Processing
        link_to (other : detachable INT_LINKABLE)
                -- Connect to `other'.
                do next := other
                ensure next = other
                end
        link_after (other : INT_LINKABLE)
                -- insert after `other' preserving what was after it.
                do
                        link_to (other.next)
                        other.link_to (Current)
                ensure
                        next  = old other.next
                        other.next  = Current
                end
end
```

Same as *next* := *other.next* ? — Yes!

Same as *other.next* := **Current** ? — No!

INT_LINKED_LIST: insert_after (2)

```
insert_after (a_value, target : INTEGER)
    -- Insert `a_value' right after first occurence `target', if exists,
    -- otherwise add `a_value' after `last_element'.
    local
        current_element, new_element : like first_element
    do
        create new_element.set_value (a_value)
        from current_element := first_element
        until current_element = Void or else current_element.value = target
        loop current_element := current_element.next
        end
        if current_element /= Void then
            new_element.link_after (current_element)
            if current_element = last_element then
                last_element := new_element
            end
        else – list does not contain `target'
            ...
        end
        count := count + 1
    ensure
        count = old count + 1
        not old has(target) implies
                (attached last_element as le implies le.value = a_value)
        old has(target) implies (attached get_element(target) as ge implies
                    (attached ge.next as gen implies gen.value = a_value) )
    end
```

INT_LINKED_LIST: insert_after (3)

```
insert_after (a_value, target : INTEGER)
    -- Insert `a_value' right after first occurence `target', if exists,
    -- otherwise add `a_value' after `last_element'.
    local
        current_element, new_element : like first_element
    do
        create new_element.set_value (a_value)
        ...
        if current_element /= Void then
        ...
        else – list does not contain `target'
            if count = 0 then
                first_element := new_element
            else
                if attached last_element as le then
                    new_element.link_after(le)
                end
            end
            last_element := new_element
        end
        count := count + 1
    ensure
        ...
    end
```

Same as *le.link_to (new_element)* ?

Yes!

INT_LINKED_LIST

Write a routine that

- calculates the sum of all positive values in a list

sum_of_positive: *INTEGER*

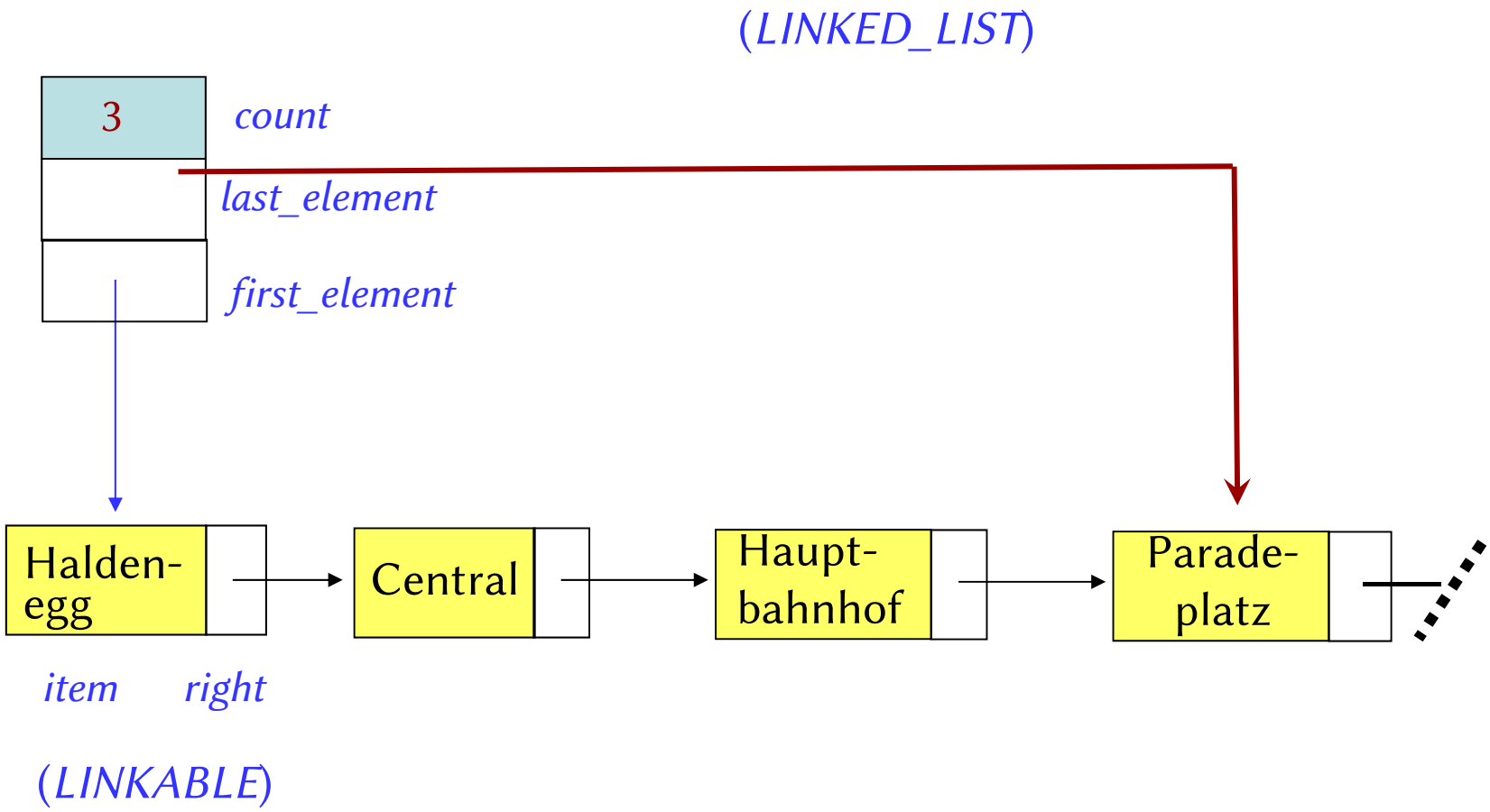
do ... **end**

INT_LINKED_LIST: sum_of_positive

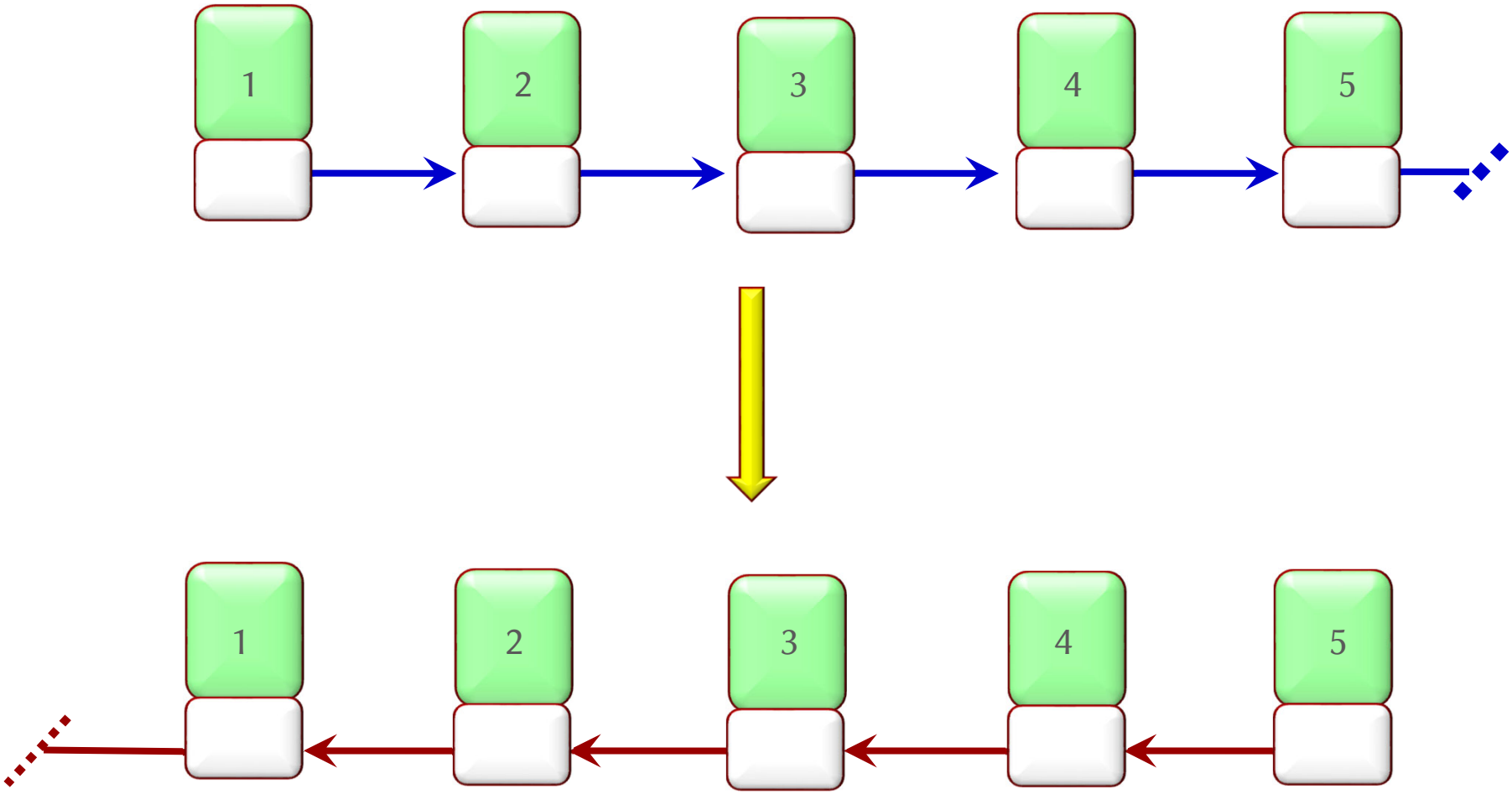
```
sum_of_positive: INTEGER
        -- The sum of positive elements
    local
        current_element : like first_element
    do
        from
            current_element := first_element
        until
            current_element = Void
        loop
            if current_element.value > 0 then
                Result := Result + current_element.value
            end
            current_element := current_element.next
        end
    ensure
        Result >= 0
    end
```

Exercise (uses loops)

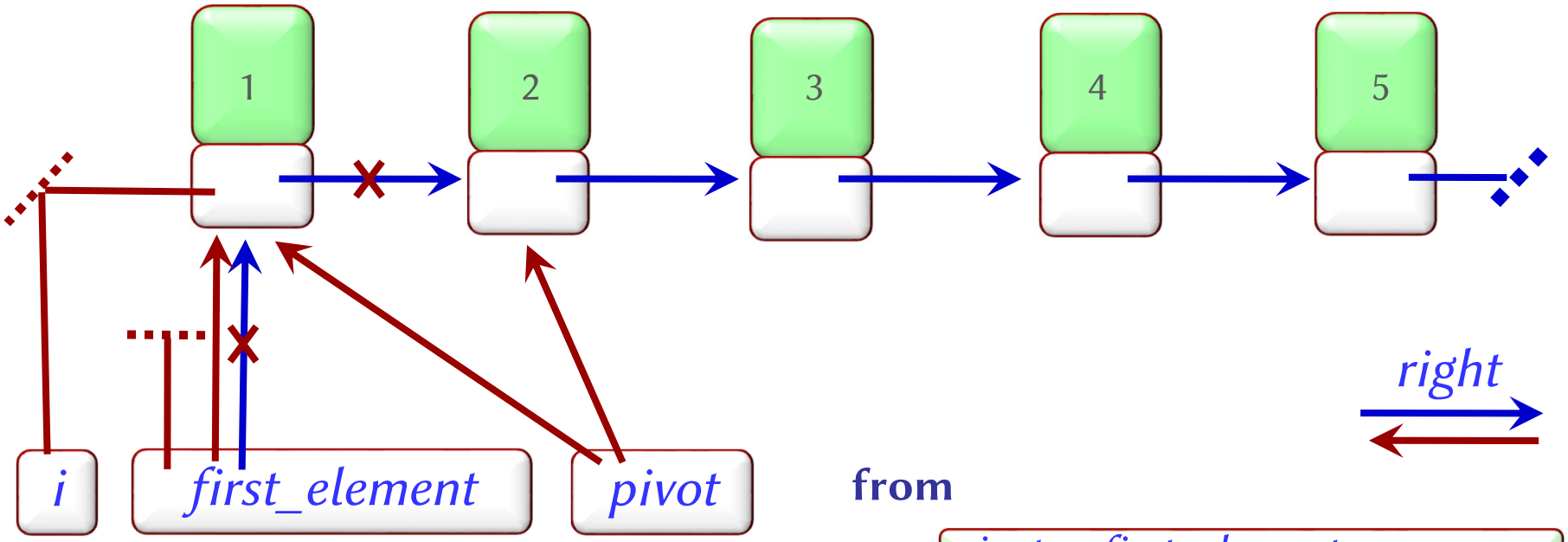
Reverse a list!



Reversing a list

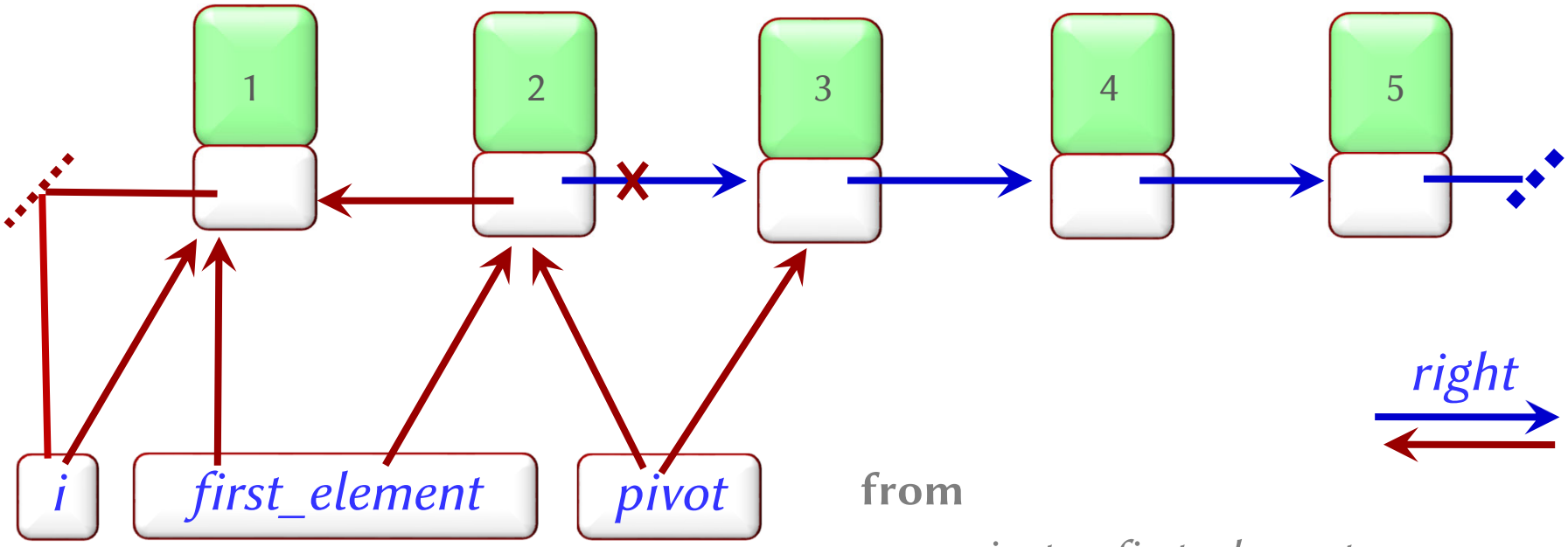


Reversing a list



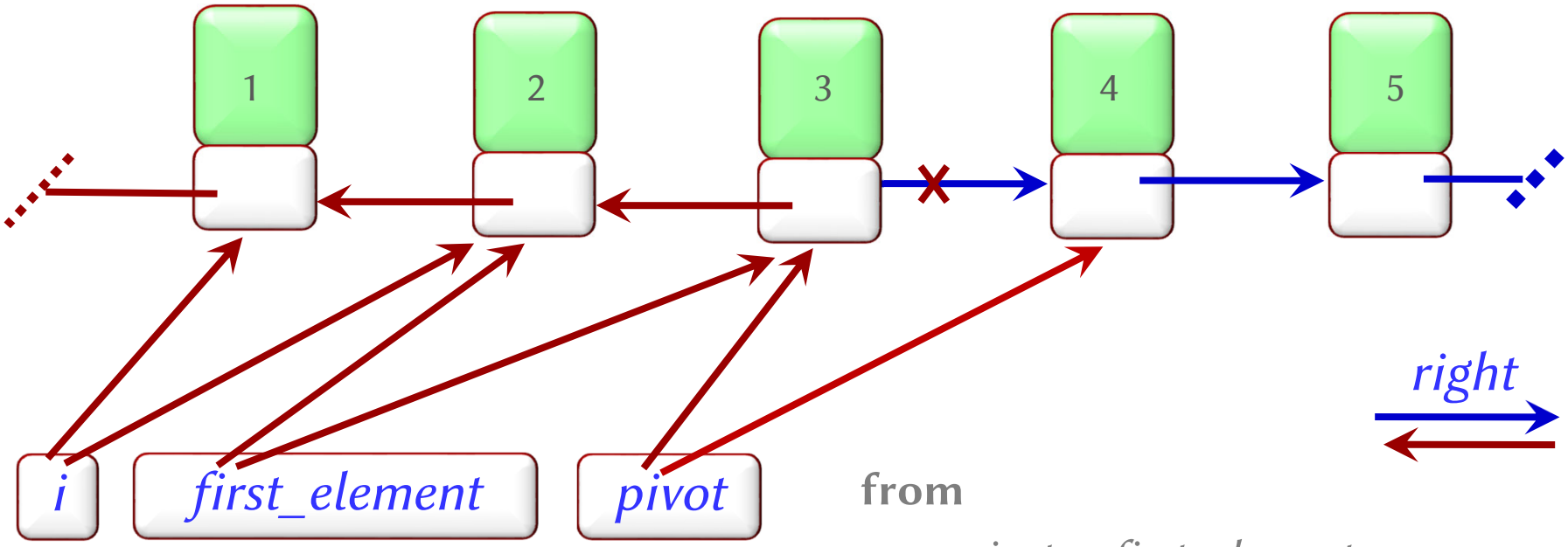

```
from
        pivot := first_element
        first_element := Void
until pivot = Void loop
        i := first_element
        first_element := pivot
        pivot := pivot.right
        first_element.put_right (i)
end
```

Reversing a list



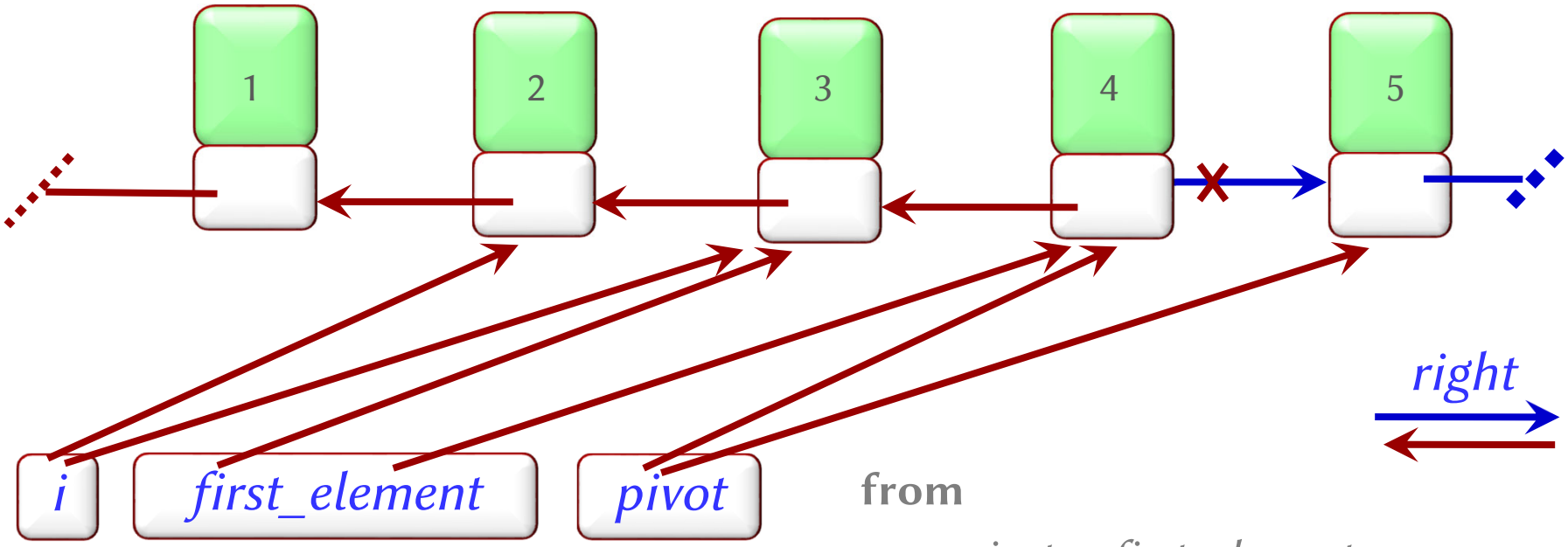

```
from
        pivot := first_element
        first_element := Void
until pivot = Void loop
        i := first_element
        first_element := pivot
        pivot := pivot.right
        first_element.put_right (i)
end
```

Reversing a list



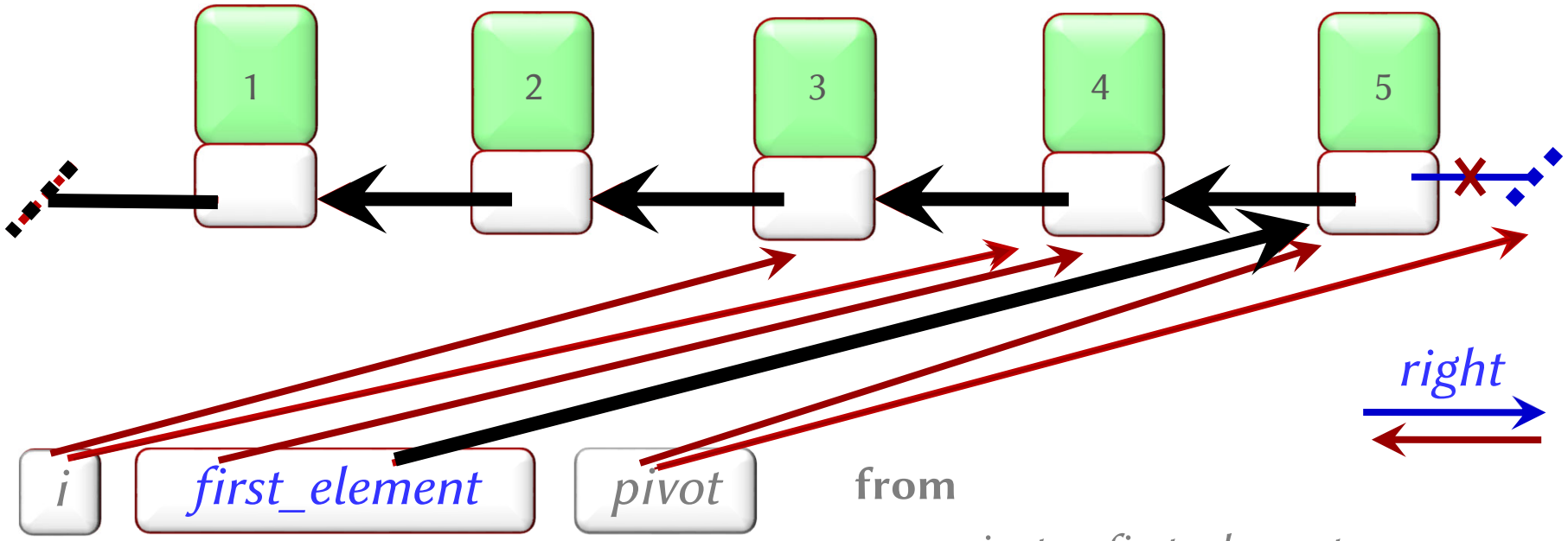

```
from
        pivot := first_element
        first_element := Void
until pivot = Void loop
        i := first_element
        first_element := pivot
        pivot := pivot.right
        first_element.put_right (i)
end
```

Reversing a list



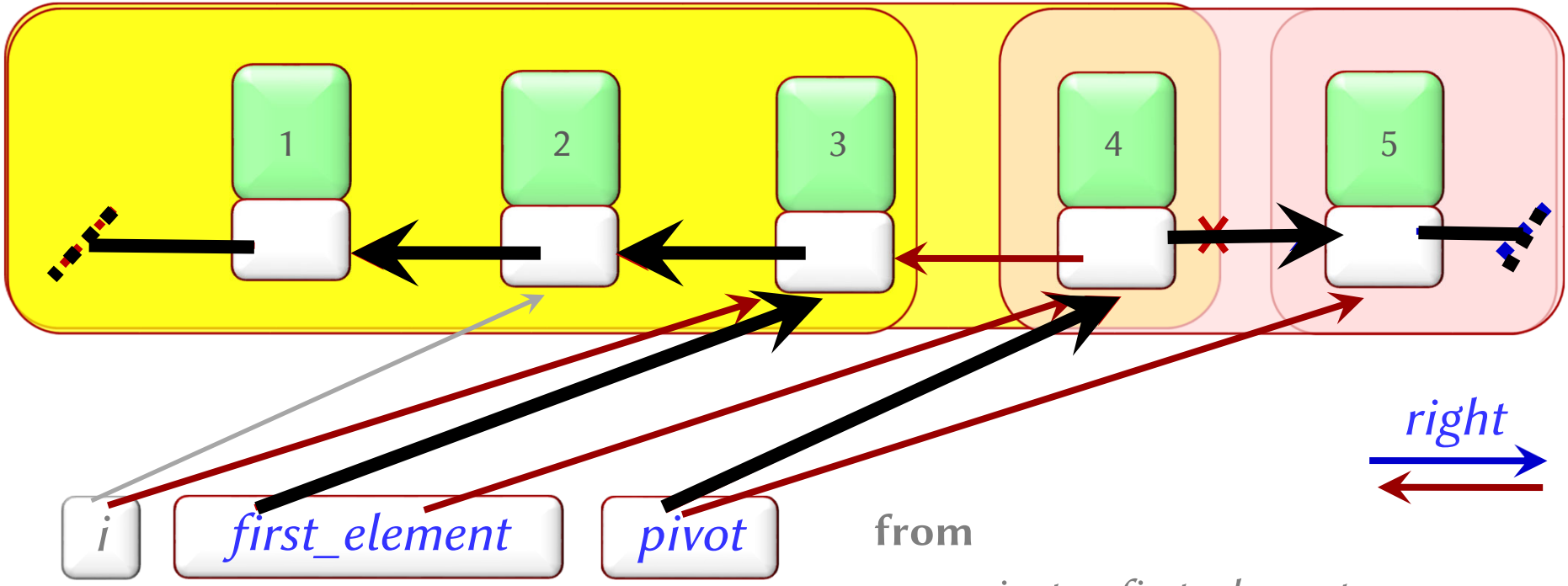

```
from
        pivot := first_element
        first_element := Void
until pivot = Void loop
        i := first_element
        first_element := pivot
        pivot := pivot.right
        first_element.put_right (i)
end
```

Reversing a list




```
from
        pivot := first_element
        first_element := Void
until pivot = Void loop
        i := first_element
        first_element := pivot
        pivot := pivot.right
        first_element.put_right (i)
end
```

The loop invariant




```
from
        pivot := first_element
        first_element := Void
until pivot = Void loop
        i := first_element
        first_element := pivot
        pivot := pivot.right
        first_element.put_right (i)
end
```

Invariant: following *right*, from *first_element*, initial items are in inverse order; from *pivot*, rest of items in original order

Fibonacci numbers

Implement a function that calculates Fibonacci numbers, using a loop

fibonacci (*n*: *INTEGER*): *INTEGER*

-- n-th Fibonacci number

require

n_non_negative: *n* >= 0

ensure

first_is_zero : *n* = 0 **implies Result** = 0

second_is_one : *n* = 1 **implies Result** = 1

other_correct : *n* > 1 **implies Result** = *fibonacci* (*n* - 1) + *fibonacci* (*n* - 2)

end

Fibonacci numbers (solution)

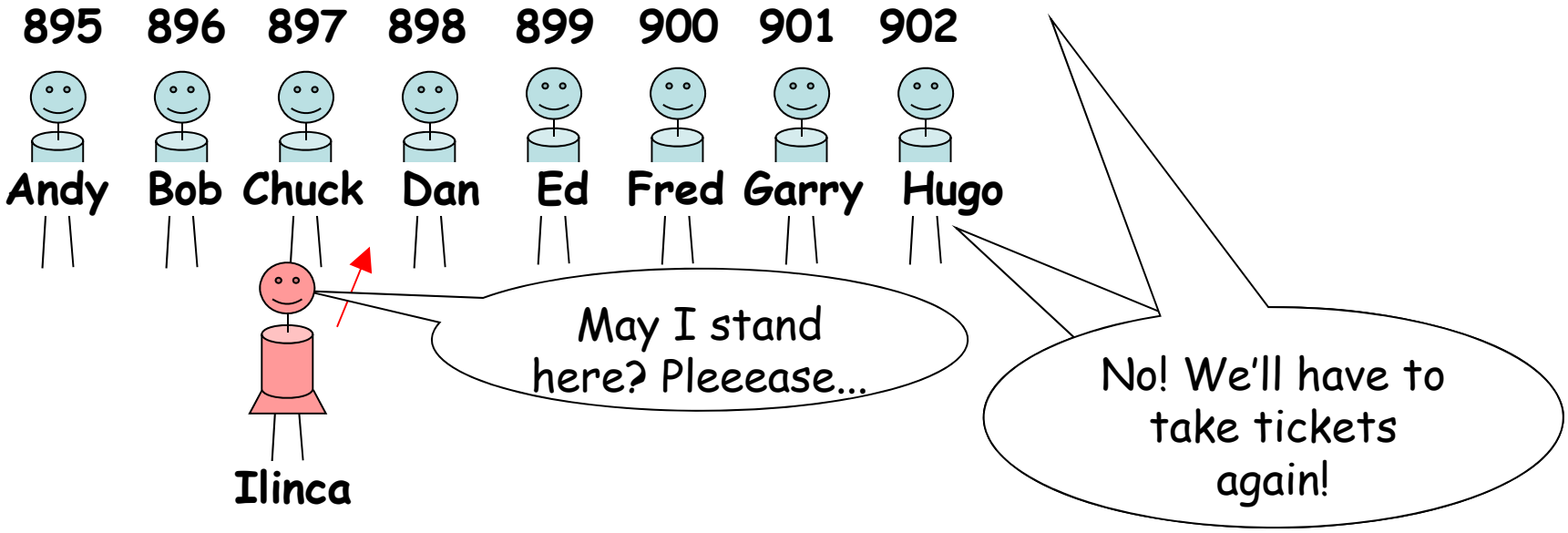
Hands-On

```
fibonacci (n : INTEGER): INTEGER
        local
                a, b, i : INTEGER
        do
                if n <= 1 then
                        Result := n
                else
                        from
                                a := fibonacci (0)
                                b := fibonacci (1)
                                i := 1
                                Result := 1
                        invariant
                                a = fibonacci (i - 1)
                                b = fibonacci (i )
                                Result = fibonacci (i )
                        until
                                i = n
                        loop
                                Result := a + b
                                a := b
                                b := Result
                                i := i + 1
                        variant
                                n - i
                        end
                end
        end
```

Invariants **AND** exit condition provide the required solution

Two kinds of queues

"Inflexible" queue (like in the post office)



Two kinds of queues

"Adaptable" queue