Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 6
Today

- Conditional
- Loop
- Linked list
Inside the routine body

- The body of each routine consists of instructions (command calls, creations, assignments, etc.)
- In the programs you’ve seen so far they were always executed in the order they were written

```python
create passenger.make_with_route(Route3, 1.5)
passenger.go
passenger.set_reiterate(True)
Paris.put_passenger(passenger)
create tram.make_with_line(Line1)
tram.start
Paris.put_tram(tram)
```

- Programming languages have structures that allow you to change the order of execution
Structured programming

- If the order of execution could be changed arbitrarily, it would be hard to understand programs.
- In **structured programming** instructions can be combined only in three ways:
  - Compound
  - Conditional
  - Loop

- Each of these blocks has a single entrance and exit and is itself an instruction.
Conditional

- Basic syntax:
  
  ```
  if c then
    i_1
  else
    i_2
  end
  ```

- c is a boolean expression (e.g., entity, query call of type `BOOLEAN`)

- else-part is optional:
  ```
  if c then
    i_1
  end
  ```
Compilation error? Runtime error? (1)

\[ f(x, y: \text{INTEGER}): \text{INTEGER} \]

do

\[
\text{if } \left( x \div y \right) \text{ then}
\]

1

\[
\text{else}
\]

0

end

dc
Compilation error? Runtime error? (2)

\[ f(x, y: \text{INTEGER}): \text{INTEGER} \]

\[
\text{do} \\
\text{if (False) then} \\
\quad \text{Result} := x \div y \\
\text{end} \\
\text{if (}x \neq 0\text{) then} \\
\quad \text{Result} := y \div x \\
\text{end} \\
\text{end}
\]

Everything is OK (during both compilation and runtime)
Calculating function’s value

\[ f(\text{max}: \text{INTEGER}; s: \text{STRING}): \text{STRING} \]

\[
\begin{align*}
\text{do } & \\
\text{if } s.\text{is}\_\text{equal}(\text{"Java"}) & \text{ then} \\
\text{Result} & := \text{"J**a"} \\
\text{else } & \\
\text{if } s.\text{count} > \text{max} & \text{ then} \\
\text{Result} & := \text{"<an unreadable German word>"} \\
\text{end } & \\
\text{end } & \\
\text{end } & \\
\end{align*}
\]

Calculate the value of:

- \( f(3, \text{"Java"}) \rightarrow \text{"J**a"} \)
- \( f(20, \text{"Immatrikulationsbestätigung"}) \rightarrow \text{"<an unreadable German word>"} \)
- \( f(6, \text{"Eiffel"}) \rightarrow \text{Void} \)
What does this routine do?

\[ \text{abs (} x : \text{REAL)} : \text{REAL} \]

\[
\text{do}
\]

\[
\text{if (} x \geq 0 \text{) then}
\]

\[
\text{Result} := x
\]

\[
\text{else}
\]

\[
\text{Result} := -x
\]

\[
\text{end}
\]

\[
\text{end}
\]
Write a routine...

- ... that computes the maximum of two integers:
  \[\text{max}(a, b: \text{INTEGER}): \text{INTEGER}\]

- ... that increases time by one second inside class \text{TIME}:
  \begin{verbatim}
  class \text{TIME}
  
  hour, minute, second: \text{INTEGER}

  second_forth
  do ... end

  ...

  end
  \end{verbatim}
Comb-like conditional

If there are more than two alternatives, you can use the syntax:

```
if \( c_1 \) then
    i_1
elseif \( c_2 \) then
    i_2
... 
elseif \( c_n \) then
    i_n
else
    i_e
end
```

instead of:

```
if \( c_1 \) then
    i_1
else
    if \( c_2 \) then
        i_2
    else
        ...
    else
        ...
    end
    ... 
else
    ...
    if \( c_n \) then
        i_n
    else
        i_e
end
end
```
If all the conditions have a specific structure, you can use the syntax:

```
inspect expression
when const_1 then
  i_1
when const_2 then
  i_2
...
when const_n1 .. const_n2 then
  i_n
else
  i_e
end
```
Lost in conditions

Rewrite the following multiple choice:

- using a comb-like conditional
- using nested conditionals

```python
inspect user_choice
when 0 then
    print("Here is your hamburger")
when 1 then
    print("Here is your Coke")
else
    print("Sorry, not on the menu today!")
end
```
Lost in conditions: solution

if user_choice = 0 then
  print ("Here is your hamburger")
elseif user_choice = 1 then
  print ("Here is your Coke")
else
  print ("Sorry, not on the menu today!")
end

if user_choice = 0 then
  print ("Here is your hamburger")
else
  if user_choice = 1 then
    print ("Here is your Coke")
  else
    print ("Sorry, not on the menu today!")
  end
end
Loop

Syntax:

```
from initialization invariant inv until exit_condition loop body variant var end
```

- `from` (Optional)
- `initialization`
- `invariant inv`
- `until exit_condition`
- `loop body`
- `variant var`
- `end`

- Instruction
- Boolean expression
- Instruction
- Integer expression
How many times will the body of the following loop be executed?

```plaintext
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
Simple loop (2)

And what about this one?

\[ i \text{: INTEGER} \]

\[ \ldots \]

\[ \text{from} \]

\[ i := 10 \]

\[ \text{until} \]

\[ i < 1 \]

\[ \text{loop} \]

\[ \text{print ("I will not say bad things about assistants")} \]

\[ \text{end} \]

Caution!
Loops can be infinite!
What does this function do?

\[
\text{factorial} \ (n: \ \text{INTEGER}): \ \text{INTEGER} \ \text{is}
\]

require
\[
\quad n \geq 0
\]
local
\[
\quad i: \ \text{INTEGER}
\]
do
\[
\quad \text{from}
\]
\[
\quad i := 2
\]
\[
\quad \text{Result} := 1
\]
until
\[
\quad i > n
\]
loop
\[
\quad \text{Result} := \text{Result} \ast i
\]
\[
\quad i := i + 1
\]
end
end
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 correct variant can not be infinite (why?)
What are the invariant and variant of the “factorial” loop?

```plaintext
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!
Implement a function that calculates Fibonacci numbers, using a loop

\[ \text{fibonacci}(n: \text{INTEGER}): \text{INTEGER} \]

\[ \quad -- \text{n-th Fibonacci number} \]

\[ \text{require} \]
\[ \quad n\_\text{non\_negative}: n \geq 0 \]

\[ \text{ensure} \]
\[ \quad \text{first\_is\_zero}: n = 0 \text{ implies } \text{Result} = 0 \]
\[ \quad \text{second\_is\_one}: n = 1 \text{ implies } \text{Result} = 1 \]
\[ \quad \text{other\_correct}: n > 1 \text{ implies } \text{Result} = \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2) \]
Writing loops (solution)

```plaintext
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
      invariant
         a = fibonacci (i - 1)
         b = fibonacci (i)
      until
         i = n
      loop
         Result := a + b
         a := b
         b := Result
         i := i + 1
      variant
         n - i
   end
end
```
Two kinds of queues

Electronic queue (like in the post office)

No! We would have to take tickets again!

May I stand here? Pleeease...

No! We would have to take tickets again!
Two kinds of queues

Live queue

Chuck is after me

Dan is after me

... 

Sure! Just remember that Dan is after you
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

(data) (LINKABLE) (LINKABLE) (LINKABLE) (LINKABLE) (LINKABLE)
class INT_LINKABLE
create put
feature
  item: INTEGER

  put (i: INTEGER)
    do item := i end

right: INT_LINKABLE

  put_right (other: INT_LINKABLE)
    do right := other end
end
class INT_LINKED_LIST
feature
  first_element: INT_LINKABLE
    -- First cell of the list
  last_element: INT_LINKABLE
    -- Last cell of the list
  count: INTEGER
    -- Number of elements in the list
...
end
**INT_LINKED_LIST**: inserting at the end

```
4 \rightarrow 8 \rightarrow 15
```

- `count`: 4
- `first_element`: 4
- `last_element`: 16

```
(\text{INT\_LINKABLE}) \quad (\text{INT\_LINKABLE}) \quad (\text{INT\_LINKABLE})
```
extend(v: INTEGER)
   -- Add v to end.
   local
     new: INT_LINKABLE
   do
     create new.put(v)
     if first_element = Void then
       first_element := new
     else
       last_element.put_right(new)
     end
     last_element := new
     count := count + 1
   end
**INT_LINKED_LIST: search**

\[ has(v: \text{INTEGER}): \text{BOOLEAN} \]

-- Does list contain \( v \)?

**local**

\[ \text{temp}: \text{INT_LINKABLE} \]

**do**

**from**

\[ \text{temp} := \text{first\_element} \]

**until**

\( (\text{temp} = \text{Void}) \) or Result

**loop**

if \( \text{temp.item} = v \) then

\[ \text{Result} := \text{True} \]

end

\[ \text{temp} := \text{temp.right} \]

end

end
Write a routine that
- calculates the sum of all positive values in a list
  
  ```plaintext
  sum_of_positive: INTEGER
  do ... end
  ```

- inserts an element after the first occurrence of a given value and does nothing if the value is not found
  
  ```plaintext
  insert_after (i, j: INTEGER)
  do ... end
  ```
**INT_LINKED_LIST: sum_of_positive**

`sum_of_positive: INTEGER`

-- Sum of positive elements

local

`temp: INT_LINKABLE`

do

from

`temp := first_element`
until

`temp = Void`

loop

if `temp.item > 0` then

`Result := Result + temp.item`

end

`temp := temp.right`

end

end
**INT_LINKED_LIST: insert_after**

```plaintext
insert_after(i, j: INTEGER)  
-- Insert `j` after `i` if present

local
  temp, new: INT_LINKABLE

do
  from
    temp := first_element
  until
    temp = Void or else temp.item = i
  loop
    temp := temp.right
  end
if temp /= Void then
  create new.put(j)
  new.put_right(temp.right)
  temp.put_right(new)
  count := count + 1
  if temp = last_element then
    last_element := new
  end
end
end
```