

---

# Fondamenti della Programmazione: Metodi Evoluti

Prof. Enrico Nardelli

Lezione 14: Agenti



1. Given a set of predefined functions, allow the user to choose one of them to be numerically integrated, according to user provided interval and integration step
2. Allow the user to input a polynomial function to be numerically integrated, as above

# Integration: a first solution

```
compute_integral (f: INTEGRABLE_FUNCTION;  
a, b, step: REAL): REAL
```

```
-- Integral of function  $f$   
-- over Interval  $[a, b]$ 
```

```
local
```

```
 $x$ : REAL;  $i$ : INTEGER
```

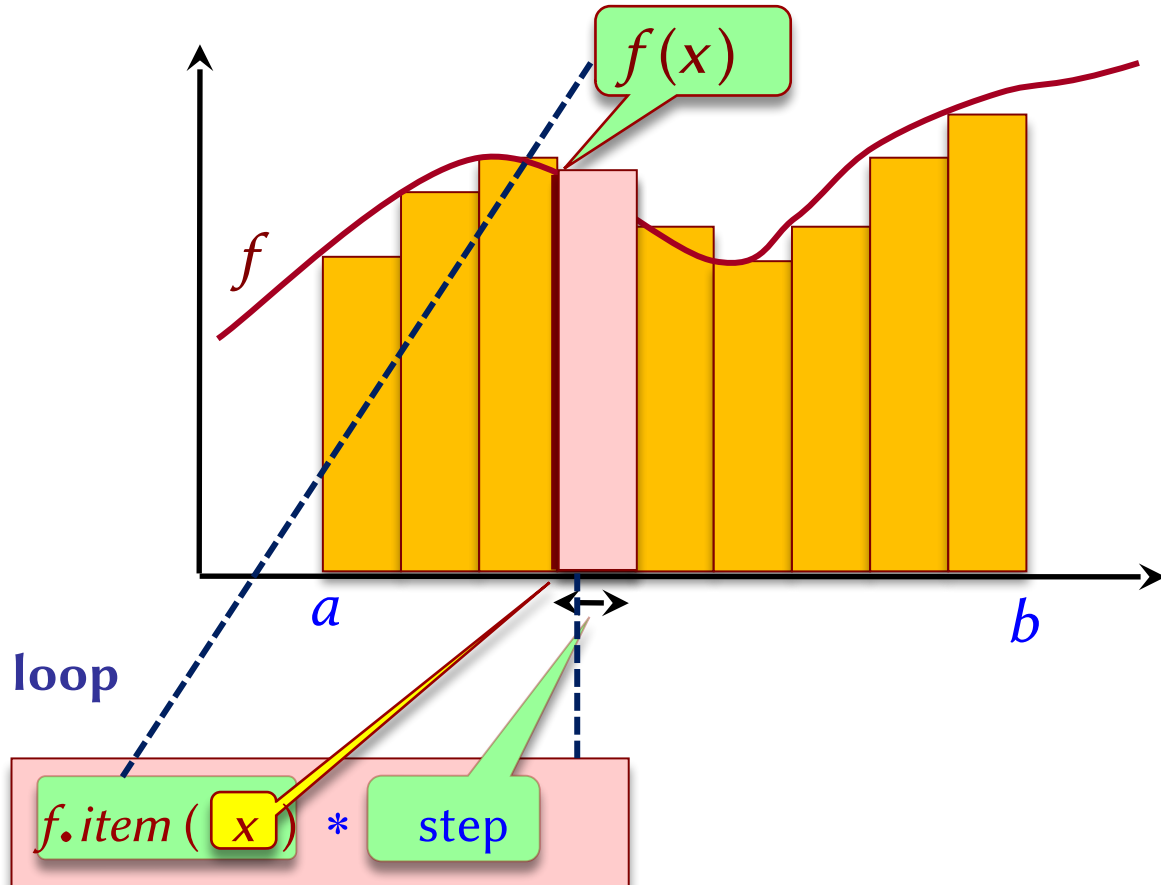
```
do
```

```
from  $x := a$   $i := 0$  until  $x \geq b$  loop
```

```
Result := Result +  
 $i := i + 1$ 
```

```
 $x := a + i * step$ 
```

```
end  
end
```



Numerical question: why not  
 $x := x + step$ ?

# Solution highlights

**deferred class** *INTEGRABLE\_FUNCTION* **feature**

*item* (*x*: *REAL*): *REAL*

...

For each function to be integrated one has to define an appropriate **class**, subclass of *INTEGRABLE\_FUNCTION* and implementing the required feature *item* which provides the current value of the function

**class** *CALCULUS* **feature**

*compute\_integral* (*f*: *INTEGRABLE\_FUNCTION*; *a*, *b*, *step*: *REAL*): *REAL*

...

*my\_calculus*: *CALCULUS*

...

*r* := *my\_calculus.compute\_integral* (*f\_to\_integrate*, *start*, *end*, *step*)

# Solution demo

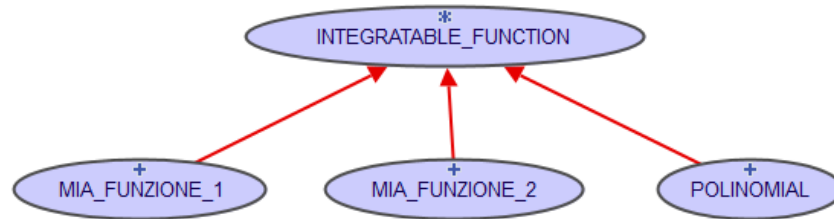
---

**See demo...**

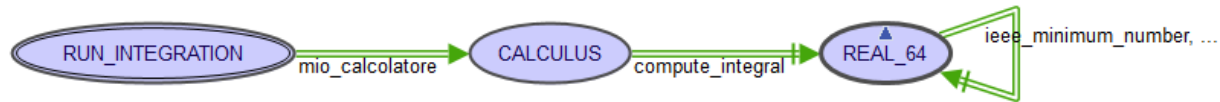
# Solution issues

Need to define a *wrapper class* (*INTEGRABLE\_FUNCTION*) only to provide the common ancestor for all functions to be integrated.

It has just one instance



It has no attributes



A better approach is to directly pass the function to be integrated as an argument to the integration procedure

The mechanism allowing this has to be provided by the programming language: in Eiffel it is the *agent mechanism*

# Agents (1)

**Agents** are objects whose unique purpose is to describe an operation

In Eiffel an operation is represented by a routine: a command (procedure) or a query (function)

Given a routine *r* (without arguments) the corresponding agent is **defined** by the expression

**agent** *r*

An agent can be assigned to an object (of an appropriate type to be seen next)

*a* := **agent** *r*

Now we can ask *a* to **execute** routine *r* through a predefined feature *call* (of an appropriate type: to be seen next)

*a.call*

like if we just wrote

*r*

## Agents (2)

If routine  $r$  takes two arguments then writing

$a.call(x, y)$

is the agent  $call$  producing the same effect as

$r(x, y)$

The old syntax used  $a.call([x, y])$   
Here  $(x, y)$  is indeed a tuple but  
with a simpler syntax

In such a way we can pass a routine  $r$  as an argument to another routine  $t$  so that the passed routine  $r$  is known inside  $t$  just through its formal name  $a$



## Agents (3)

---

Agents provide to **operations** the separation between  
definition of the operation (**agent definition**)  
execution of the operation (**agent call**)

Useful whenever an object has to apply an operation to  
other objects without prior knowledge of the specific  
operation

- **providing new operations to existing objects**

e.g.: iterating over a list and applying an action to every  
item, without knowing the action in advance

# Application of agents

---

**Numerical programming:** Applying a calculus operation to a **function**

**Iteration:** Applying an **operation** to all elements of a data structures

**Event-driven programming:** Applying a program **reaction** to an event (and being able to undo it)

**User interaction:** Being able to undo user **actions**

# A first kind of agent: function (1)

---

Assume class

*MY\_CLASS*

has feature

*my\_function* (*x: REAL*): *REAL* **do ... end**

and consider

*my\_object: MY\_CLASS*

We want to *agent-ize* *my\_object.my\_function*

Which is the language construct allowing to write

*a := agent my\_object.my\_function* ???

# A first kind of agent: function (2)

In general:

which is the type of an “*agent-ized*” function?

Described by ELFFEL generic class

***FUNCTION***[*A*, *R*]

*A* is constrained to be a type conforming to ***TUPLE***

*R* denotes the type of the result returned by the function

The old syntax used ***FUNCTION***(*B*, *A*, *R*) where *B* denoted the class providing the function passed as an agent, or its ancestor; often ***ANY*** was used.

# A first kind of agent: function (3)

Declaring variable

*a* : **FUNCTION**[*REAL*, *REAL*]

allows to write (agent **definition**)

*a* := **agent** *my\_object.my\_function*

not only by assignment (as written above) but also through parameter passing.

Then the request to the agent to execute feature **call**

*a.call* (*x*)

has the same effect as

*my\_object.my\_function* (*x*)

...but for the fact that *a.call* (*x*) does not return a result !

The old syntax used

**FUNCTION**(*ANY*, *TUPLE*[*REAL*], *REAL*)

## A first kind of agent: function (4)

Feature *call* of class *FUNCTION* [*A*, *R*] just executes the *agent*-ized routine and store its result in feature *last\_result*

Therefore

$$a.call(x)$$
$$s := a.last\_result$$

produces the same effect as

$$s := my\_object.my\_function(x)$$

A shortcut is to write (keeping the convention for accessing a generic item of a structure)

$$s := a.item(x)$$

# Integration: agent based solution

*compute\_integral* (*f*: **FUNCTION** [*REAL*, *REAL*]; *a*, *b*, *step*: *REAL*): *REAL*

-- Integral of function *f*  
-- over Interval [*a*, *b*]

**local**

*x*: *REAL*; *i*: *INTEGER*

**do**

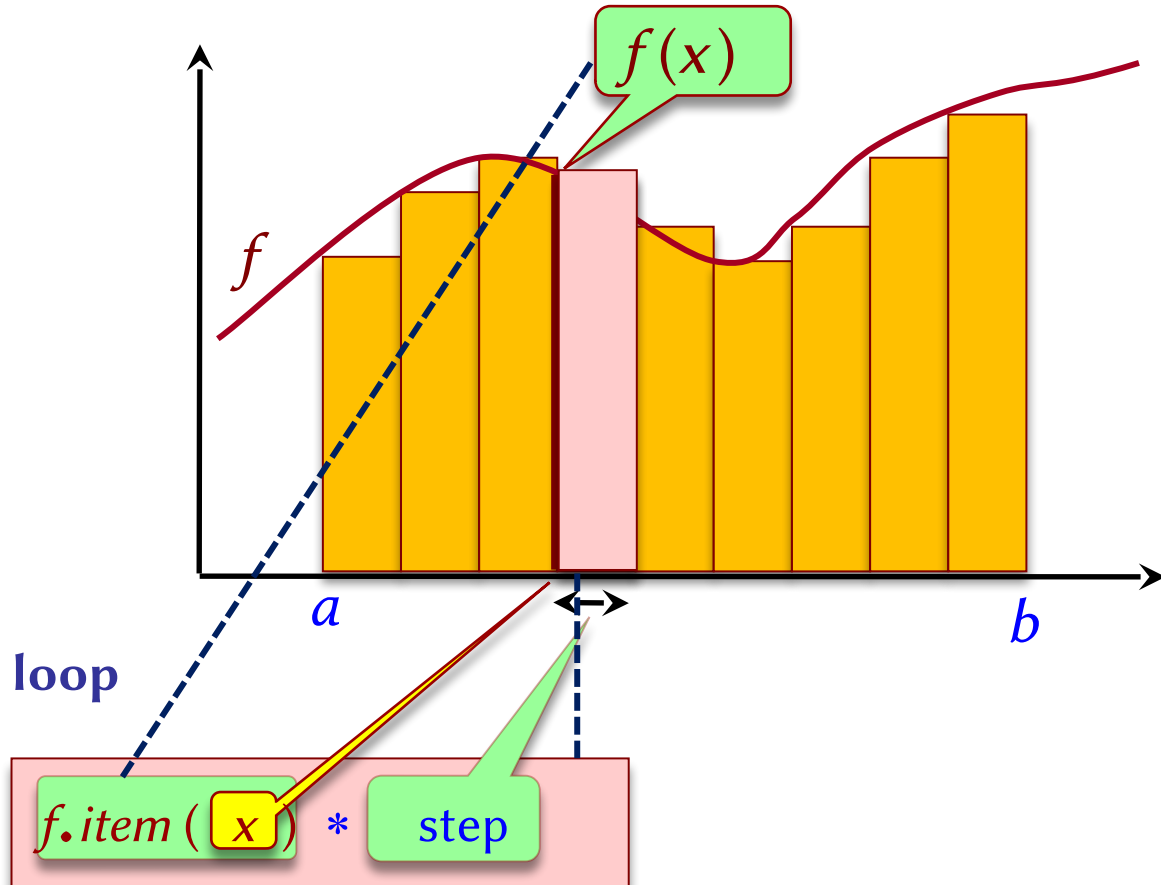
from *x* := *a* *i* := 0 until *x* ≥ *b* loop

**Result** := **Result** +  
*i* := *i* + 1

*x* := *a* + *i* \* *step*

**end**

**end**



Numerical question: why not  
 $x := x + step$ ?

# Agent-based solution demo

---

**See demo...**



# Agent-based solution highlight and comparison

**class** *CALCULUS* **feature**

*compute\_integral* (*f*: *FUNCTION*[*REAL*, *REAL*]; *a*, *b*, *step*: *REAL*): *REAL*

...

*my\_calculus*: *CALCULUS*

...

*r* := *my\_calculus.compute\_integral* (**agent** *f\_to\_integrate*, *start*, *end*, *step*)

a feature

Previous solution without agents

**class** *CALCULUS* **feature**

*compute\_integral* (*f*: *INTEGRATABLE\_FUNCTION*; *a*, *b*, *step*: *REAL*): *REAL*

...

**deferred class** *INTEGRATABLE\_FUNCTION* **feature**

*item* (*x*: *REAL*): *REAL*

...

*my\_calculus*: *CALCULUS*

...

*r* := *my\_calculus.compute\_integral* (*f\_to\_integrate*, *start*, *end*, *step*)

a class instance

# One more issue (1)

Declaring variable

$a : \textit{FUNCTION}[\textit{REAL}, \textit{REAL}]$

allows to write (agent **definition**)

$a := \textit{agent } my\_object.my\_function$

and we can then ask the agent  $a$  to compute  $my\_function$  with an argument  $x$  assigned by the routine using the agent itself

$s := a.item(x)$

What if we have a function with two arguments and we want the agent to compute it with one fixed parameter  $p$  defined by us?

# One more issue (2)

We declare variable

*a2* : **FUNCTION** [REAL, REAL, REAL]

and write (agent **definition**)

*a2* := **agent** *my\_object.my\_function* (?, *p*)

and we can then ask the agent *a2* to compute *my\_function* with first argument *x* assigned by the routine using the agent itself and second argument assigned to *p* by the agent definition

*s* := *a2.item* (*x*)

The old syntax used  
**FUNCTION**(ANY, TUPLE[REAL, REAL], REAL)

and we get the same effect as

*s* := *my\_object.my\_function* (*x*, *p*)

If no argument is used in the agent definition, then all the routine arguments have to be assigned by the routine using the agent

# Iteration examples

---

- Perform the following actions on a list of persons with name and age
  1. Print the name of each
  2. Increment age of each by a given amount

Implementation for each is straightforward

# Iteration examples (2)

---

**See demo...**

## Iteration examples (3)

---

For each problem the same “pattern” (**loop-and-for-each-do-something**) is applied again and again: can we abstract?

# Towards a generalization (1)

Assume *MY\_LIST\_CLASS* has a procedure able to iterate over all elements

```
do_for_each_item (action-on-some-argument ...)
  from start
  until after
  loop
    apply action-on-some-argument to current item
  forth
end
```

A routine passed as argument to a routine...

... so that it can be executed at run-time

# Towards a generalization (2)

Then for

*my\_list* : *MY\_LIST\_CLASS*

writing

*my\_list.do\_for\_each\_item* (*my\_procedure*)

would apply *my\_procedure* to each element in *my\_list*

The old syntax used  
*PROCEDURE*(*ANY*, *TUPLE*[*G*])

Descendants of *LINEAR* [*G*], like *LINKED\_LIST* [*G*], have such a routine! (a mechanism of this kind is often called **iterator**)

*do\_all* (*action* : *PROCEDURE*[*G*])

**from** *start*

**until** *after*

**loop**

*action.call* (*item*)

*forth*

**end**

The mechanism to  
*agent-ize* a  
procedure...

The argument passed to the *agent-ized* procedure is *item*, the *LINKED\_LIST* attribute denoting the current element of the list



# A second kind of agent: procedure (1)

In general:

which is the type of an “*agent-ized*” procedure?

Described by ELFFEL generic class

***PROCEDURE*[A]**

*A* is constrained to be a type conforming to ***TUPLE***

The old syntax used ***PROCEDURE*(B,A)** where *B* denoted the class providing the procedure passed as an agent, or its ancestor; often ***ANY*** was used.

## A second kind of agent: procedure (2)

Declaring object

$a : \textit{PROCEDURE}[G]$

allows to attach to  $a$  the object (agent **definition**)

$a := \textit{agent my\_object.my\_procedure}$

not only by assignment (as written above) but also through parameter passing.

Then the request to the agent to execute feature **call**

$a.\textit{call}(p)$

has exactly the same effect as

$\textit{my\_object.my\_procedure}(p)$

## A second kind of agent: procedure (3)

Then for

*my\_list* : *LINKED\_LIST* [*PERSON*]

writing

*my\_list.do\_all* (*my\_procedure*)

would apply *my\_procedure* to each element in *my\_list* , as if it were written

**from** *start*

**until** *after*

**loop**

*my\_procedure* (*item*)

*forth*

**end**

# Iteration solution with agents: first approach

How to use this with

*my\_list : LINKED\_LIST [PERSON]*

so that the iterator

*my\_list.do\_all(...)*

has the required behavior?

Which is the *my\_procedure* to be passed in an agent form so as to **print** or **increment\_age** ?

It has to be a procedure such that the current item of the list (an instance of *PERSON*) is its **argument**

And which is the **target** that has to call such a procedure?

# Iteration solution with agents: first approach

The answer is in the syntax of *agent*-ized procedure and *do\_all*

*my\_list.do\_all(agent my\_object.my\_procedure)*

*my\_procedure* has to be such that the current item of *my\_list* (which is an instance of *PERSON*) is its **argument**

*my\_object* cannot be a specific instance of *PERSON* since *my\_procedure* will have to be applied to each item of the list (and this is taken care by *do\_all*)

Hence *my\_object* has to be the current object (**Current**), which has to have a procedure able to call the appropriate procedure of *PERSON*

# Iteration examples

Given *my\_list* : *LINKED\_LIST* [*PERSON*]

we can write *my\_list.do\_all(...)* so as to implement:

- Perform the following actions on a list of persons with name and age
  1. **Print** the name of each
  2. **Increment age** of each by a given **amount**

The argument of action (**print**, **increment\_age**) is the current person, possibly with parameter(s) (**amount**)

But how to pass the proper argument?

# Iteration examples with agents

---



1. Perform the following operations on a list of persons with name and age
  1. Print the name of each
  2. Increment age of each by a given amount

# Iteration example solution (1<sup>st</sup> approach)

---

**See demo...**



# Agent-based solution (1<sup>st</sup> approach) highlight

Previous solution for integration was

**class** *CALCULUS* **feature**

*integral* (*f*: *FUNCTION*[*REAL*, *REAL*]; *a*, *b*, *step*: *REAL*): *REAL*

...

*my\_calculus*: *CALCULUS*

...

*r* := *my\_calculus.integral* (**agent** *f\_to\_integrate*, *start*, *end*, *step*)

a feature

In this case solution is

*my\_list.do\_all* (**agent** *print\_person*)

same as **agent** *print\_person*(?)

...

**feature** *print\_person* (*p*: *PERSON*) **do** *p.print\_me* **end**

...

**class** *PERSON* **feature**

*print\_me* **do** *print* ('my name is:' , *name*); ... **end**

Feature needed to use  
the *PERSON* feature  
*print\_me*

# Agent-based solution (1<sup>st</sup> approach) highlight

Also for case 2. the solution has the same structure

```
my_list.do_all (agent increment_age (?, delta_age) )
```

...

```
feature increment_age (p: PERSON; delta_age: INTEGER)
  do p.set_age (age + delta_age) end
```

...

```
class PERSON feature
  p.set_age (new_age: INTEGER) do age := new_age; ... end
```

Feature needed to use  
the *PERSON* feature  
*set\_age*

# Agent-based solution (1<sup>st</sup> approach) discussion

```

my_list.do_all (agent print_person)
...
feature print_person (p: PERSON) do p.print_me end
...
class PERSON feature
  print_me do print ('my name is:' , name); ... end
-----
my_list.do_all (agent increment_age (?, delta_age) )
...
feature increment_age (p: PERSON; delta_age: INTEGER)
  do p.set_age (age + delta_age) end
...
class PERSON feature
  p.set_age (new_age: INTEGER) do age:= new_age; ... end

```

In both cases we had to *wrap* the feature defined on PERSON (*print\_me*, *set\_age*) in a new feature (defined at the same level of the object on which iterator is applied) to be passed as an agent to the iterator

Can we avoid this wrapping and pass directly the original feature?

# Avoid wrapping procedures

We need a mechanism to pass, as an agent, a procedure defined for instances of a class *G* without *agent*-izing it by means of making reference to a specific instance *x* of *G*

(remember the agent definition is **agent** *my\_object.my\_procedure*)

Otherwise we cannot use that *agent*-ized procedure for arbitrary instances of *G*

Given procedure *my\_procedure* defined in class *G*, the agent **definition**

**agent** {*G*}.*my\_procedure*  
does the job.

It is called **agent definition with open target**

# Iteration examples with agents

---



- Perform the following operations on a list of persons with name and age
  1. Print the name of each
  2. Increment age of each by a given amount

# Iteration example solution (2<sup>nd</sup> approach)

---

**See demo...**

# Agent-based solution (2<sup>nd</sup> approach) discussion (1)

```
my_list.do_all (agent {PERSON}.print_me)
```

...

```
class PERSON feature
```

```
  print_me
```

```
  do print ('my name is:' , name); ... end
```

Previous solution (1st approach)

```
my_list.do_all (agent print_person)
```

...

```
feature print_person (p: PERSON) do p.print_me end
```

...

```
class PERSON feature
```

```
  print_me
```

```
  do print ('my name is:' , name); ... end
```

Agent mechanism  
allows to reuse  
*print\_me* (an  
existing procedure of  
*PERSON*) without  
having to **wrap** it

# Agent-based solution (2<sup>nd</sup> approach) discussion (2)

```
my_list.do_all (agent {PERSON}.increment_age (delta_age) )
```

...

```
class PERSON feature
```

```
  increment_age (delta_age: INTEGER)
```

```
    do age := age + delta_age end
```

Previous solution (1st approach)

```
my_list.do_all (agent increment_age (?, delta_age) )
```

...

```
feature increment_age (p: PERSON; delta_age: INTEGER)
```

```
  do p.increment_age (delta_age) end
```

...

```
class PERSON feature
```

```
  increment_age (delta_age: INTEGER)
```

```
    do age := age + delta_age end
```

If we can modify the code of *PERSON* and define a feature *increment\_age* then we can re-use it without having to **wrap** it



# Agent-based solution (2<sup>nd</sup> approach) discussion (3)

But if we cannot modify the code of the class *PERSON* then we have to use the 1st approach

Previous solution (1st approach)

```
my_list.do_all (agent increment_age (?, delta_age) )
```

...

```
feature increment_age (p: PERSON; delta_age: INTEGER)
  do p.set_age (age + delta_age) end
```

...

```
class PERSON feature
  set_age (new_age: INTEGER)
    do age := new_age; ... end
```

# Keeping arguments open

An agent can have both “closed” and “open” arguments

*closed* arguments are set at time of agent definition;

*open* arguments are set when requesting the agent to execute feature **call**

To keep an argument open, just replace it by a question mark:

$u := \text{agent } a0.f(a1, a2, a3)$  -- All closed

$w := \text{agent } a0.f(a1, a2, ?)$

$x := \text{agent } a0.f(a1, ?, a3)$

$y := \text{agent } a0.f(a1, ?, ?)$

$z := \text{agent } a0.f(?, ?, ?)$  -- All open. Same as  $z := \text{agent } a0.f$

# Calling an agent with open/closed arguments

$$f(x1: T1; x2: T2; x3: T3)$$

$$a1: T1; a2: T2; a3: T3$$

$u := \text{agent } a0.f(a1, a2, a3)$

$u.call([])$

$v := \text{agent } a0.f(a1, a2, ?)$

$v.call([a3])$

$w := \text{agent } a0.f(a1, ?, a3)$

$w.call([a2])$

$x := \text{agent } a0.f(a1, ?, ?)$

$x.call([a2, a3])$

$y := \text{agent } a0.f(?, ?, ?)$

$y.call([a1, a2, a3])$

$z := \text{agent } a0.f$

$z.call([a1, a2, a3])$

# Agents with open/closed target/arguments

---

*x.a\_feature* (agent *y.f*) -- closed/open

*x.a\_feature* (agent *y.f*(?, ?) ) -- closed/open

*x.a\_feature* (agent *y.f*(a, ?) ) -- closed/partial

*x.a\_feature* (agent *y.f*(a, b) ) -- closed/closed

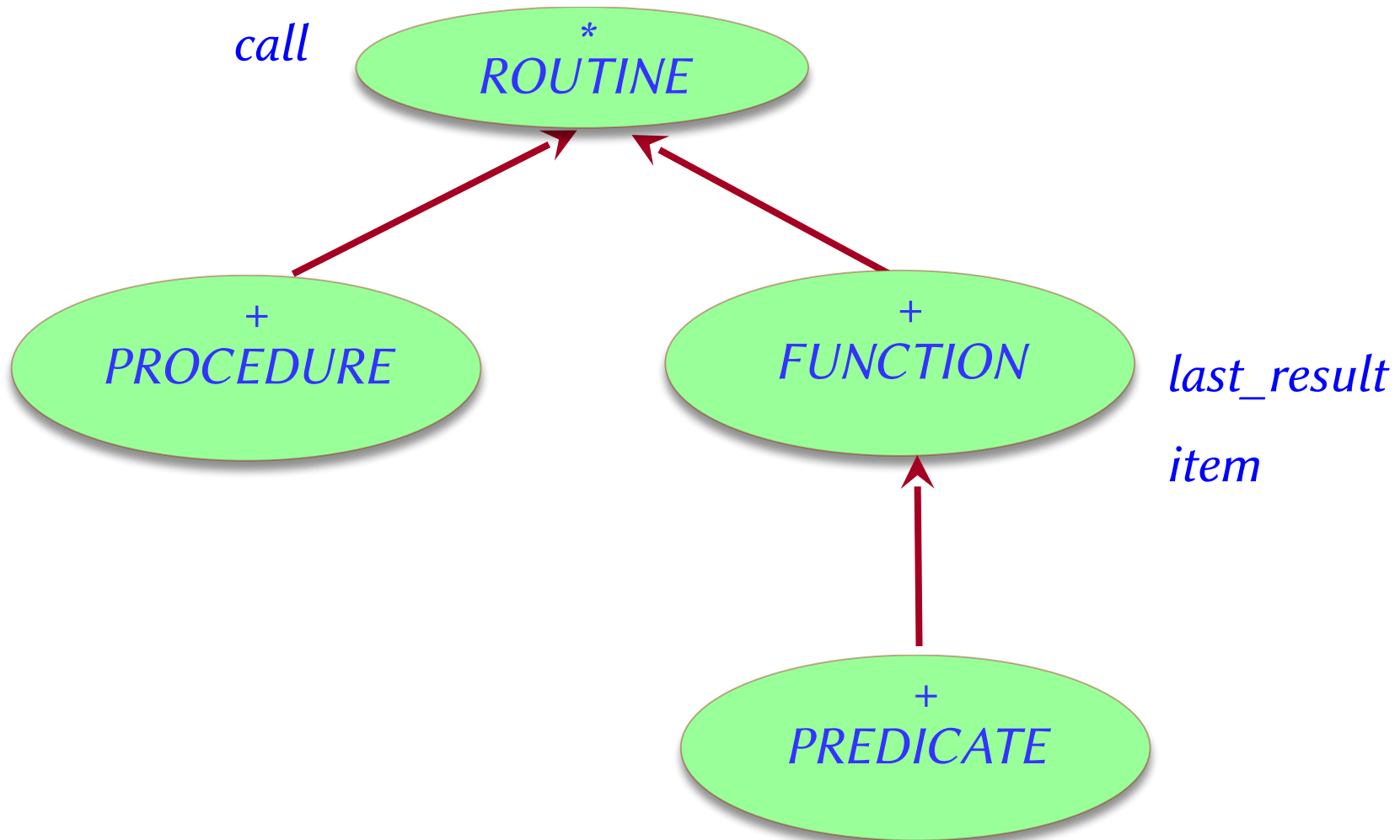
*x.an\_iterator* (agent {*C*}.*f*) -- open/open

*x.an\_iterator* (agent {*C*}.*f*(?, ?) ) -- open/open

*x.an\_iterator* (agent {*C*}.*f*(a, ?) ) -- open/partial

*x.an\_iterator* (agent {*C*}.*f*(a, b) ) -- open/closed

# Kernel library classes representing agents



# Further iterator routines in TRAVERSABLE

Other interesting iterators defined  
in *TRAVERSABLE* [G], parent of *LINEAR* [G]

Execute on all elements satisfying a given condition

*do\_if*(*action* : *PROCEDURE* [G];  
*test* : *PREDICATE* [G])

where *PREDICATE* [G] is a  
subclass of *FUNCTION* [G, *BOOLEAN*]

Test whether a property hold for **all** elements

*for\_all*(*test* : *PREDICATE* [G])

Test whether a property hold for **at least one** element

*there\_exists*(*test* : *PREDICATE* [G])

# Agent-like mechanisms in other languages

In non-O-O languages, e.g. C and Matlab, there is no notion of agent, but you can pass a routine as argument to another routine, as in

*integral* (& *f*, *a*, *b*)

where *f* is the function to integrate. & *f* (C notation for *function pointers*, one among many possible ones) is a way to refer to the function *f*. (We need some such syntax because just '*f*' could be a function call.)

Agents (or *delegates* in C# or *closures* in functional languages) provide a higher-level, more abstract and safer technique by wrapping the routine into an object with all the associated properties.