EECS3311 – Software Design

Tuples

Agents (functions as first class citizens)

- Integration
- Hold-Count (iteration)
- Undo/Redo with agents

Why agents?

Iteration:

$$\int_{a}^{b} my\_function\ (x)\ dx$$

GUI Callback:

button. click_actions.extend(**agent** *routine*)

$$\forall c: \text{CITIZEN}\ \exists\ p: \text{CITIZEN}\ |\ p \in c.parents \rightarrow c \in p.children$$

Why agents?

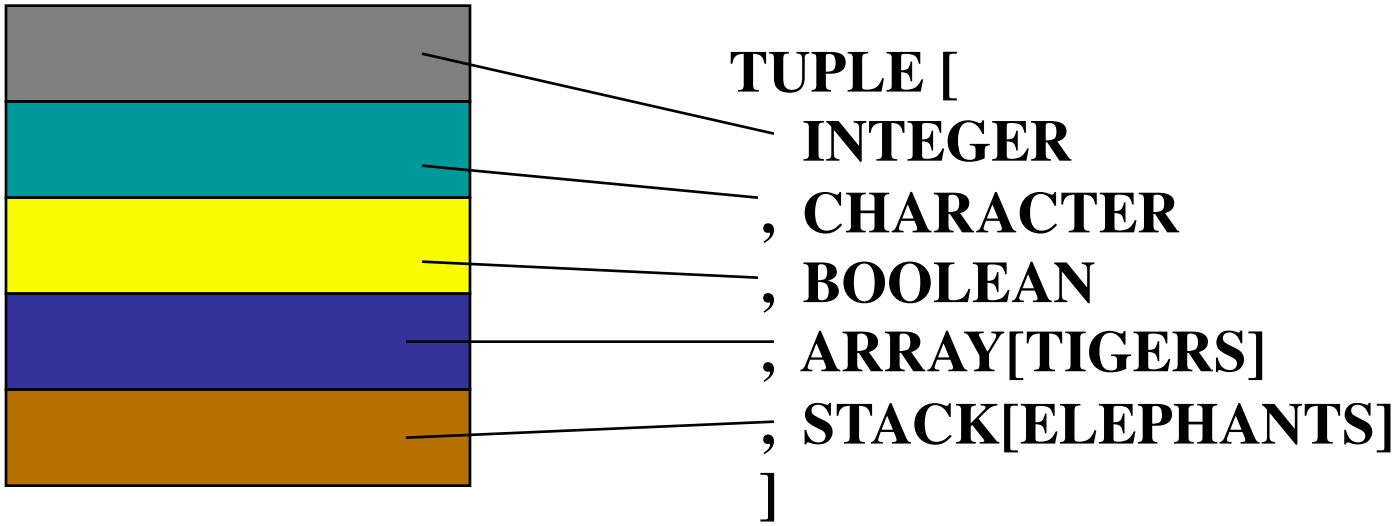
- Operations (routines) are not Data
  - object technology starts from the decision to separate these two aspects, and to choose object types (data), rather than the operations, as the basis for modular organization of a system, attaching each operation to the resulting modules -- the classes.
- But, in a number of applications, we may need **objects that represent operations**, so that we can store operations in object structures and execute them later
- This is similar to the notion of a function pointer in C, but it must be translated to the OO paradigm.

Tuples vs Array

- The generic array ARRAY[G] represents a sequence of elements, all of the same type.
  - e.g. a: ARRAY[INTEGER]
    a := <<2, 4, 6, 8>>
  - All elements are of the same (or conforming) type
- A typical tuple type is of the form TUPLE[X,Y,Z] denoting a tuple of at least three elements, such that
  - the type of the first conforms to X ,
  - the second to Y,
  - and the third to Z .

What is a tuple?

- Analogous to describing the types for the fields of a record
  - Example of a 5 field record with sufficient space in each field to hold items of the indicated type

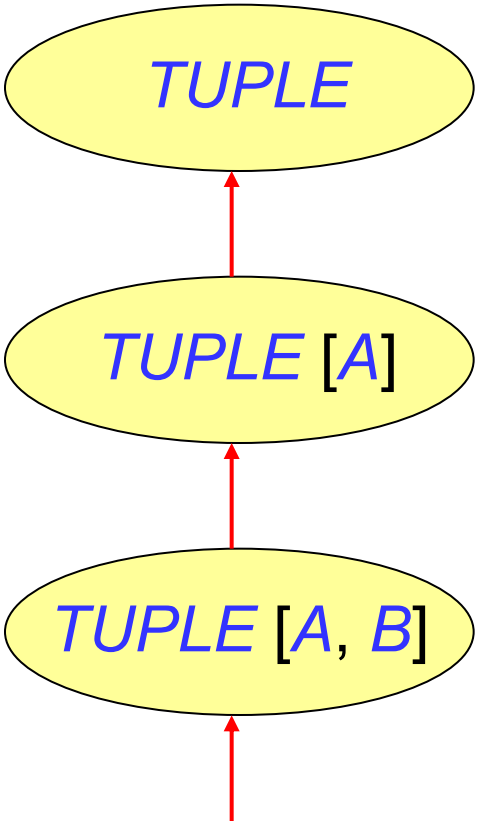


Tuple

- You may list any number of types in square brackets, including none at all.
- TUPLE, with no types in square brackets, denotes tuples of arbitrary length.
- TUPLE is an already defined class.

Tuple Types

- Syntax: *TUPLE* [*X*, *Y*, …]



Contract view of TUPLE

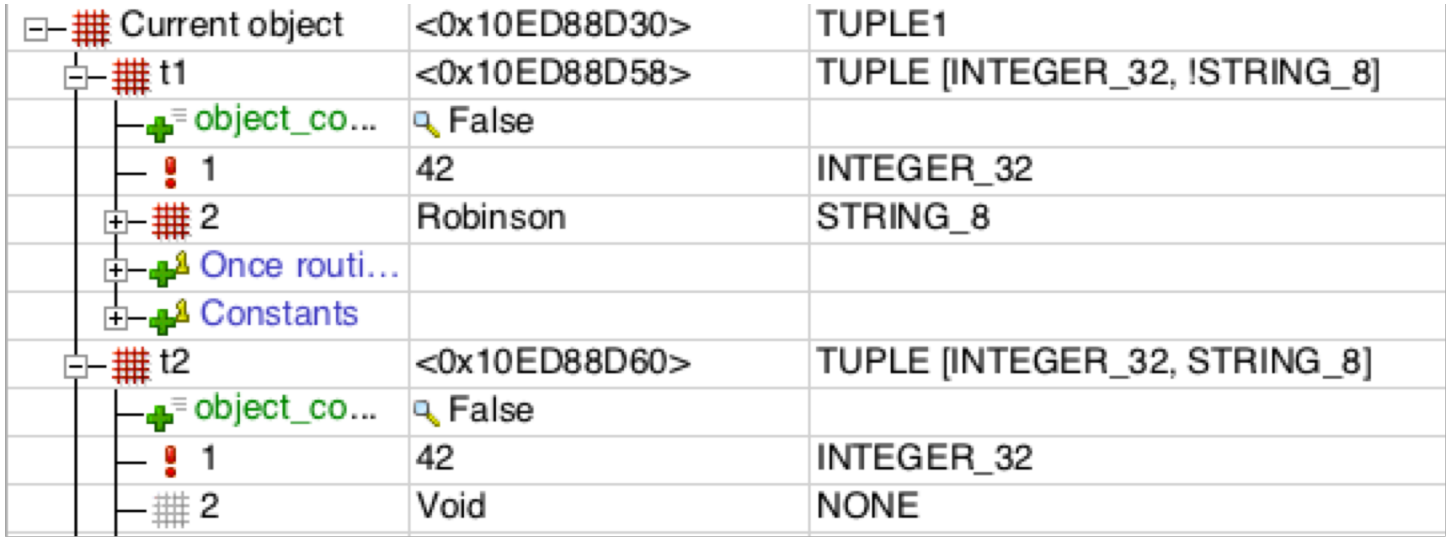
```
class TUPLE feature
    count: INTEGER
        -- number of fields in the tuple

    item alias (index: INTEGER): detachable ANY assign put
            -- Entry of key `index'.
        require
            valid_index: valid_index (index)

    put (v: ANY; index: INTEGER)
            -- Insert `v' at position `index'.
        require
            valid_index (index)
            valid_type_for_index (v, index)
end
```

Infix operators

```
item alias "[]" (index: INTEGER): detachable ANY assign put
        require
                valid_index: valid_index (index)
```

- Use either item or [] for accessing an entry in the tuple

```
class
    TUPLE1
create
    make
feature
    make
        local
            sd: detachable STRING
        do
            t1 := [42, "Robinson"]
--          t1 := [42, sd] -- VJAR error not compatible
            t2 := [42, sd]
        end
feature
    t1: TUPLE[id: INTEGER; name: STRING]

    t2: TUPLE[id: INTEGER; name: detachable STRING]

end
```



Current object	<0x10ED88D30>	TUPLE1
t1	<0x10ED88D58>	TUPLE [INTEGER_32, !STRING_8]
object_co...	False	
1	42	INTEGER_32
2	Robinson	STRING_8
Once routi...		
Constants		
t2	<0x10ED88D60>	TUPLE [INTEGER_32, STRING_8]
object_co...	False	
1	42	INTEGER_32
2	Void	NONE

TUPLE[BOOLEAN, INTEGER, STRING]

```
test_three_tuple: BOOLEAN
    local
        t3: TUPLE[BOOLEAN, INTEGER, STRING]
    do
        comment("test_three_tuple")
        t3  :=    [False,       342,       "Bzttt!"]
        Result :=
                        t3[1] = False
                and t3[2] = 342
                and t3[3] ~ "Bzttt!"
    end
```

Debugger view of the TUPLE[X,Y,Z]

t3	<0x11109EDA8>	TUPLE [BOOLEAN, INTEGER_32, !STRING_8]
object_co...	False	
1	False	BOOLEAN
2	342	INTEGER_32
3	Bzttt!	STRING_8

Integer Division as a Tuple

```
div (b, c: INTEGER): TUPLE [INTEGER, INTEGER]
        -- illustration of multi-functions
        -- (two return values) using tuples
        -- b divided by c yields quotient q and remainder r
    require
        c /= 0
    local
        r, q: INTEGER
    do
        q := b // c
        r := b \\ c
        Result := [q, r]
    end
```

Test *div*

```
test_integer_divison: BOOLEAN
        -- Calculate 23/4,
        -- 23 = 5*4 + 3
        -- a  =  q*b + r
    local
        td: TUPLE[INTEGER, INTEGER]
    do
        td := div (23, 4)
        Result  :=  td[1] = 5 and td[2] = 3
    end
```

Tuples and Void Safety

```
note
    description: "[
        Testing tuple for Void safety
        ]"
class
    FOO2 [G -> attached ANY create default_create end]
create
    make
feature
    item: G -- item must be attached with a default_create

    make(tuple: TUPLE[detachable G])
        do
            print(tuple.out); io.new_line -- this is ok

--          item := tuple[1]  -- VJAR error

            if attached {G} tuple[1] as l_item then
                item := l_item
                print(item.out) -- will not be Void
                io.new_line
            else
                create item
                -- without this we get VEVI error
                -- for item
            end
        end
```

```
TUPLE [INTEGER_32] [0x10CC8E048]
  1: INTEGER_32 = 42

42
```

```
t4: BOOLEAN
    local
        bar: FOO2 [INTEGER]
        cool: TUPLE [INTEGER]
    do
        comment ("t4: test FOO2[G] with tuple")
        cool := [42]
        create bar.make (cool)
        Result := bar.item = 42
    end
```

Agents
(Functional Programming in OO)

- See Touch of Class, Chapter 17
- An older article:
  - http://www.jot.fm/issues/issue_2004_04/article7/
- There is now new simplified notation

Agents

- Operations (i.e. routines) are not objects
  - object technology starts from the decision to separate these two aspects, and to choose object types, rather than the operations, as the basis for modular organization of a system, attaching each operation to the resulting modules -- the classes.

- In a number of applications, however, we may need objects that represent operations, so that we can include them in object structures that some other routine can call later.

Some function

```
do_something (flag: BOOLEAN; x: INTEGER) : REAL_64
        -- Return x/3 if `flag' holds true
    do
        if flag then
            Result := x/3
        end
    end
```

Some function

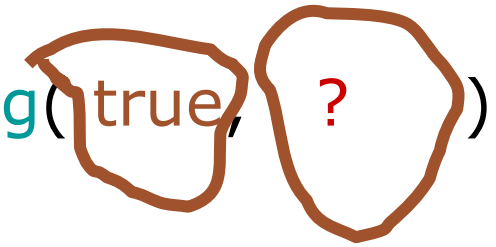
```
do_something (flag: BOOLEAN; x: INTEGER) : REAL_64
        -- Return x/3 if `flag' holds true
    do
        if flag then
            Result := x/3
        end
    end
```

```
test_function_do_something   : BOOLEAN
        local
            -- f1: FUNCTION[TUPLE[BOOLEAN,INTEGER],REAL_64]
            f1: FUNCTION[BOOLEAN,INTEGER,REAL_64]
            f2: FUNCTION[INTEGER, REAL_64]
            f3: FUNCTION[REAL_64]
            r1, r2, r3: REAL_64
        do
            -- store functions, but don't execute them
            f1 := agent do_something(?, ?)      -- two open args
            f2 := agent do_something (true, ?) -- one open arg
            f3 := agent do_something (false, 7)

            -- above is not executed until what follows

            r1 := f1(true, 7)  -- f3.item ([true, 7])
            Result := 2.333 <= r1 and r1 <= 2.334
            check Result end

            r2 := f2(7)
            Result := 2.333 <= r2 and r2 <= 2.334
            check Result end

            r3 := f3.item ([])
            Result := r3 = 0   -- default value
            check Result end

        end
```

f3.item is also ok

Closed and Open arguments

f := **agent** a.do_something(true, ?)



The agent expression **agent** *g* (*u*, **?**, *v*) denotes:

The one-argument function obtained from the three-argument function *g* by freezing its first and third arguments, to the values *u* and *v* respectively, and retaining only as a true argument the one at the second position, marked "**?**".

```
do_something (flag: BOOLEAN; x: INTEGER) : REAL_64
        -- Return x/3 if `flag' holds true
    do
        if flag then
            Result := x/3
        end
    end

do_something2 (x: INTEGER) : REAL_64
    do
        Result := do_something (true, x)
    end
```

Can obtain variants by freezing args

Agent definition vs. Agent call

```
test_function_do_something2: BOOLEAN
        local
            -- f1: FUNCTION[TUPLE[BOOLEAN,INTEGER],REAL_64]
            f1: FUNCTION[BOOLEAN,INTEGER,REAL_64]
            r: REAL_64
        do
            -- stores function, but don't execute it
            f1 := agent do_something(?, ?)    -- two open args

            -- above is not executed until what follows
            r := f1(true, 9) -- f1.call ([true,7]); r := f1.last_result

            Result := r = 3
        end
```

This is indeed what agents give us: the ability to build and dispatch objects representing operations ready to be executed, with a complete separation between:

- Agent *definition*: the place in the software that defines an agent around a routine *r*, through **agent** *r*, and of course must know about *r*.
- Agent *call*: any place in the software that receives an agent *a* and can apply features such as *call* to it, without knowing what routine it carries.

Function routine as an anonymous type

f: FUNCTION[BOOLEAN, INTEGER,REAL_64]

- f is
  - a function in some class
  - with argument TUPLE[BOOLEAN, INTEGER]
  - and return type REAL_64

```
do_something (flag: BOOLEAN; x: INTEGER) : REAL_64
        -- Return x/3 if `flag' holds true
    do
        if flag then
            Result := x/3
        end
    end
```

Reminder: constrained genericity

- *LIST* [*G*] (unconstrained): *G* represents arbitrary type. May use

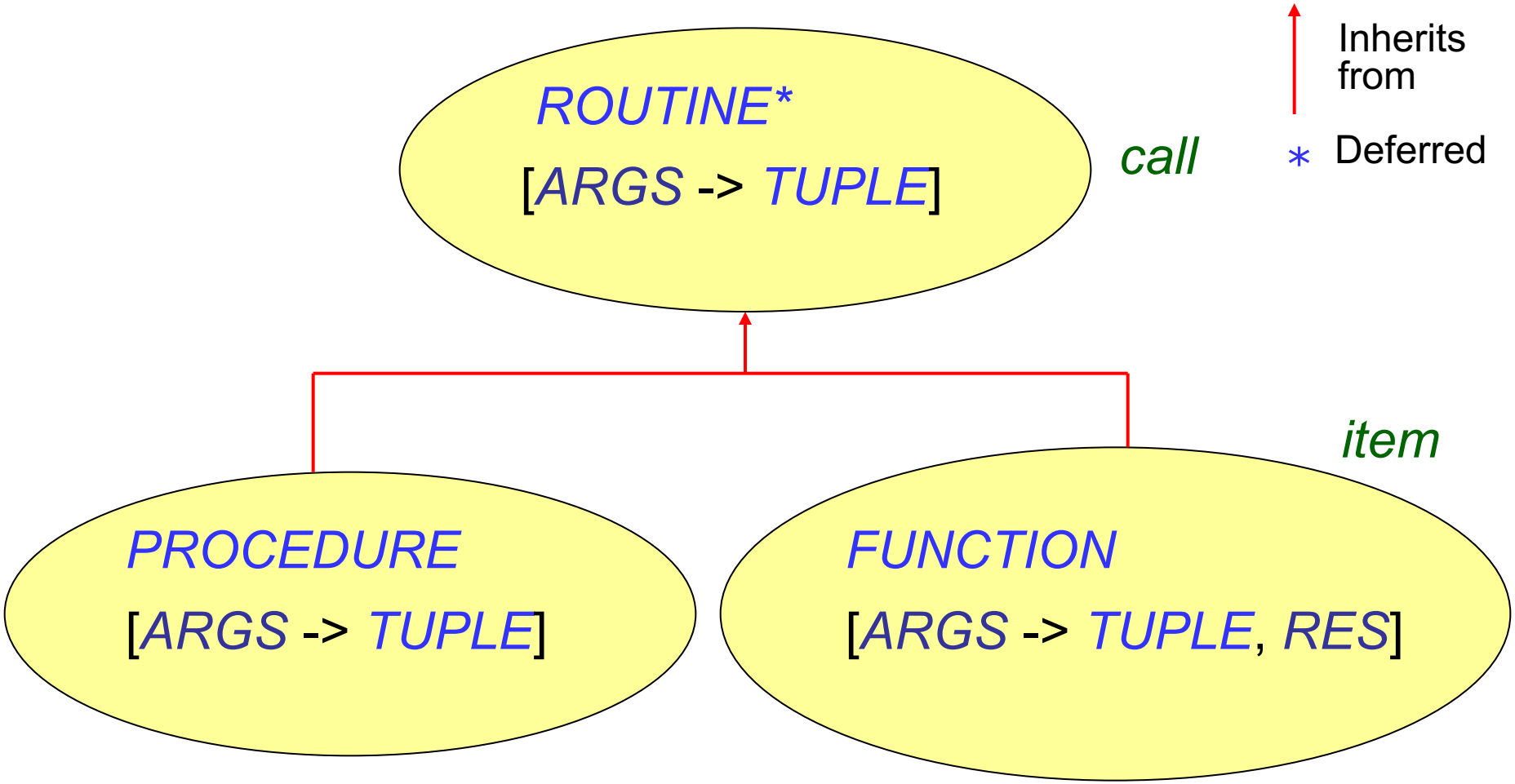
  LIST [*INTEGER*]
  LIST [*EMPLOYEE*]
  LIST [*SHIP*]

- *SORTABLE_LIST* [*G* -> *COMPARABLE*]
  (constrained by *COMPARABLE*)

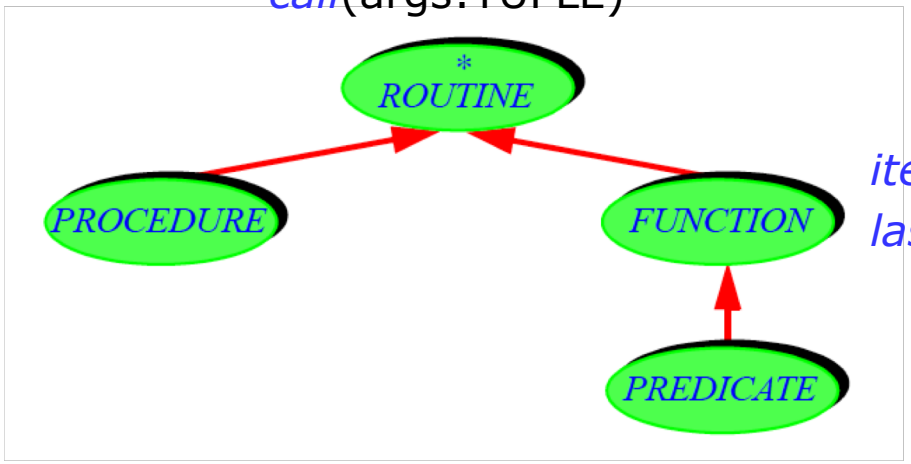
- *G* represents type descending from *COMPARABLE*

  SORTABLE_LIST [*INTEGER*]
  SORTABLE_LIST [*T*]
  --only if *T* descendant of *COMPARABLE*

Agent types: Kernel library classes

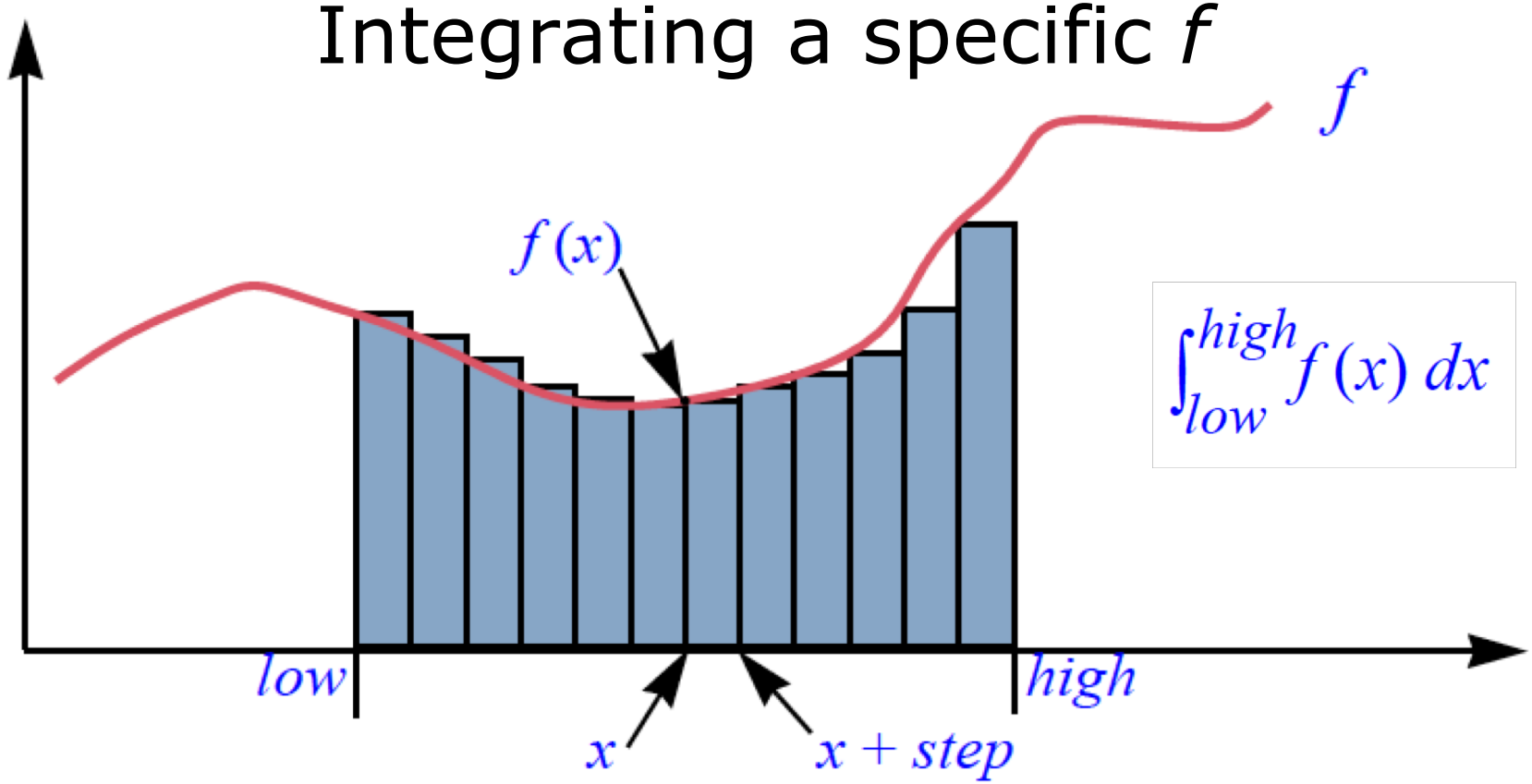


call(args:TUPLE)



item(args: TUPLE): RESULT_TYPE
last_result: **detachable** RESULT_TYPE

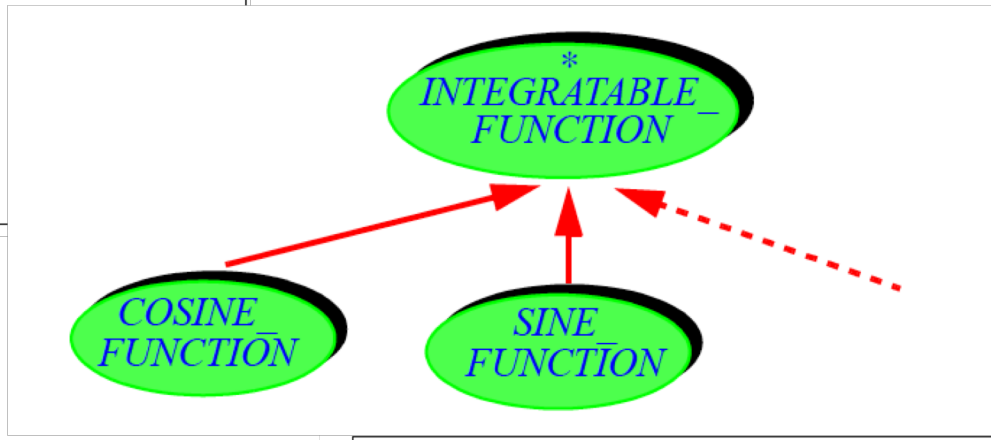
Integrating a specific *f*




```
do Result := from x := low until x >= high loop
    Result := Result + (4*x –x^2)*step  -- value of f at x
    x := x + step
end
```

Without agents

```
note
    description: "Functions that can be integrated over finite intervals"
deferred class INTEGRATABLE_FUNCTION feature
    item (x: REAL): REAL
            -- Function's value for x.
        deferred
        end
end
```




```
integral(f: INTEGRABLE_FUNCTION;
                low, hi:REAL): REAL
  do

    from ... until ... loop
      Result := Result + f.item(x) *step
    end
end
```

```
        -- In class COSINE_FUNCTION
item (x: REAL): REAL
        -- Function's value for x.
    do
        Result := cosine (x)
    end
```

```
f: INTEGRABLE_FUNCTION
r, low, hi: REAL

create {COSINE_FUNCTION}f.make
r := integral(f, low, hi)
```

Calling an agent: integration example

$$\int_{a}^{b} my\_function\ (x)\ dx$$

$$\int_{a}^{b} your\_function\ (x, u, v)\ dx$$

my_integrator.*integral* (**agent** *my_function(?)*, *a*, *b*)

my_integrator.*integral* (**agent** *your_function* (?, u, v), *a*, *b*)

The integral function

```
integral (f: FUNCTION [REAL, REAL];
          low, high: REAL): REAL
        -- Integral of f over the interval [low, high]
    local
        x: REAL
    do
        from x := low
        until x > high
        loop
            Result := Result + f(x) * step
            x := x + step
        end
    end

step: REAL
```

Where you call the function

Where you define the function

```
integral (agent {MATH}.cosine(?), 0.5, 1)


integral (agent {MATH}.tan(?), 0.2 , 0.7)
```

FUNCTION[ARGS, RESULT_TYPE]

```
call (args: OPEN_ARGS)
        -- Call routine with operands `args'.
    require
        valid_operands (args)
    do
        ...
    end

last_result: RESULT_TYPE
        -- Result of last call, if any
```

```
item (args: OPEN_ARGS): RESULT_TYPE
        -- Result of calling function with `args'
        -- Does call then sets last_result as a side effect
```

Features of routine classes

call (values: *TUPLE*)

item (values: *TUPLE*): *RESULT_TYPE*
-- In FUNCTION only

... *target*, *arguments*, *set_target*, *set_arguments*...

- Introspection features (in progress):

precondition: *FUNCTION* [*ARGUMENTS*, *BOOLEAN*]
postcondition
type: *TYPE*
Features of class *TYPE*: *heirs*, *parents*, *routines*
etc.

Counting Quantifier

$$\#i : INTEGER \bullet m \leq i \leq n \wedge P(i)$$

- is an INTEGER that denotes the number of different values in the range m .. n for which *P*(*i*) is true (note that is of type BOOLEAN).

Example of

- The number of items in the range 1 .. 6 that is greater than 5 is two

$$\{i \in 1..6 \mid i \geq 5\} = \{5,6\}$$

$$(\#i \in 1..6 : i \geq 5) = 2$$

Example

- The number of even numbers in the range [2 .. 6] is three

$$\{i : INTEGER \bullet 2 \leq i \leq 6 \wedge even(i)\} = \{2,4,6\}$$

$$(\#i : INTEGER \bullet 2 \leq i \leq 6 \wedge even(i)) = 3$$

Counting quantifiers using agents

```
class
    COUNTING
feature
    number_of (a: ARRAY[REAL]; predicate: FUNCTION [REAL, BOOLEAN]): INTEGER
            -- Number of  values in a.lower … a.upper
            -- that satisfy the predicate
        do
            across a as r loop
                if predicate(r.item) then
                    Result := Result + 1
                end
            end
        ensure
            class -- makes this query static
        end
end
```

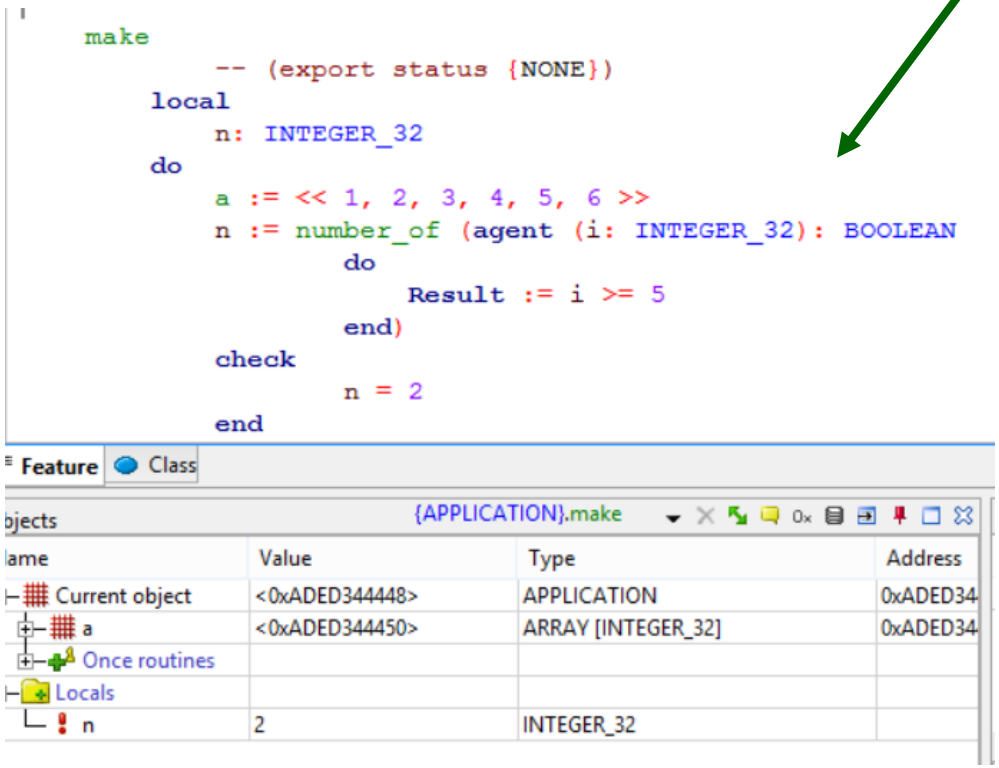
```
gt(r:REAL): BOOLEAN
    do
        Result := r >= 4.0
    end

test_counting: BOOLEAN
    local
        a: ARRAY[REAL]
        p: PREDICATE[REAL]
    do
        comment("test_counting: with an array")
        a := <<1.2, 3.9, 8.3, 16>>

        Result := {COUNTING}.number_of(a, agent gt) = 2
    end
```

agent gt(?)

$a = \langle 1.2, 3.9, 8.3, 16 \rangle$

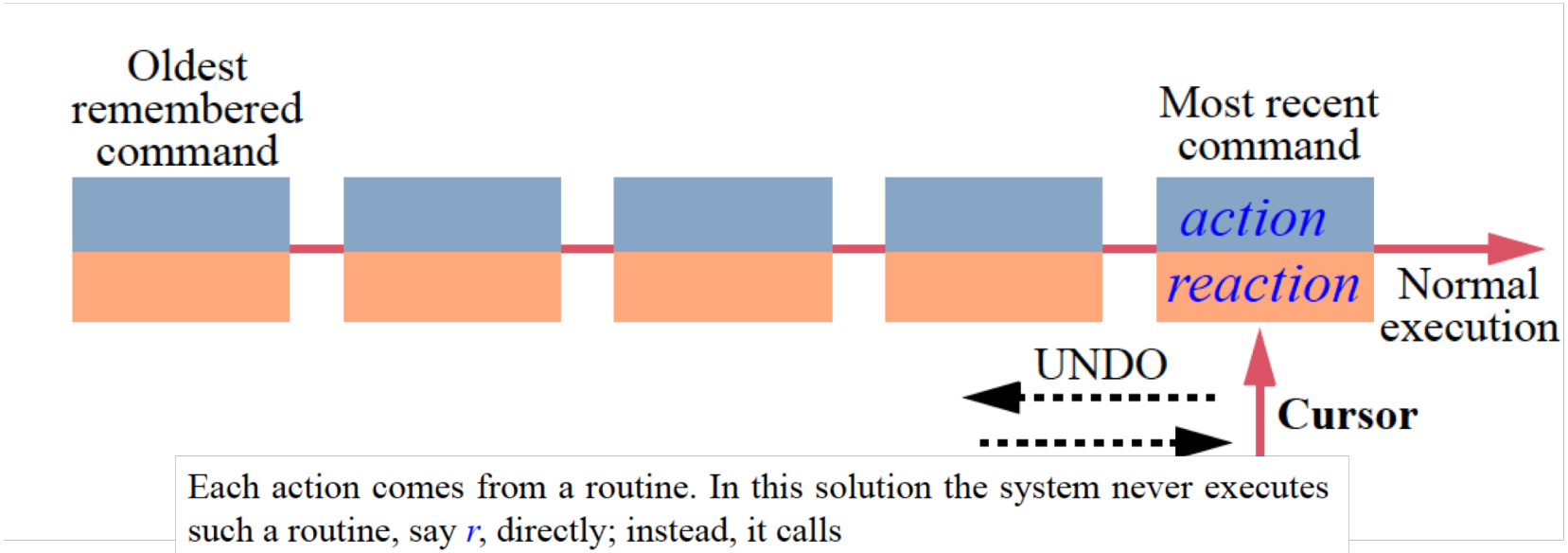
$(\#r \in a : r \geq 4) = 2$

Inline agent



Agents for undo/redo (TOC, p623)

A history list

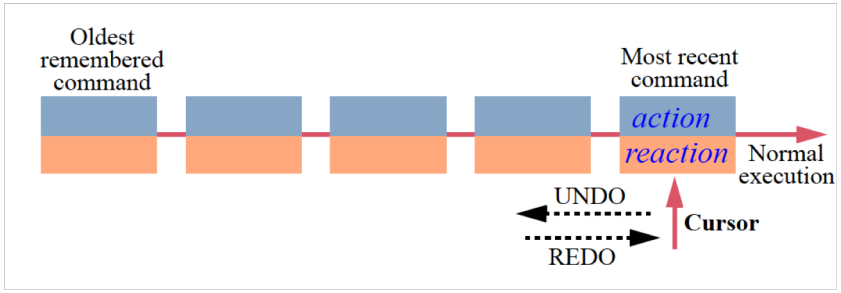


Each action comes from a routine. In this solution the system never executes such a routine, say *r*, directly; instead, it calls

```
execute (agent r , agent r_inverse)
```

where *execute* performs *call* (the mechanism for calling the routine associated with an agent, as previewed above) on its first argument, but also appends the object pair of its two arguments into the history list. Each pair in the history list contains two agents, one representing an action and the other — appearing as *reaction* in the figure — representing the reverse action; this assumes that for every routine *r* implementing a user command you also provide a routine *r_inverse* that cancels the action (otherwise you could not offer an undo-redo mechanism). Then if the user requests one or more "undo", you perform

Agents for undo/redo (TOC, p623)



Each action comes from a routine. In this solution the system never executes such a routine, say *r*, directly; instead, it calls

execute (**agent** *r* , **agent** *r_inverse*)

where *execute* performs *call* (the mechanism for calling the routine associated with an agent, as previewed above) on its first argument, but also appends the object pair of its two arguments into the history list. Each pair in the history list contains two agents, one representing an action and the other — appearing as *reaction* in the figure — representing the reverse action; this assumes that for every routine *r* implementing a user command you also provide a routine *r_inverse* that cancels the action (otherwise you could not offer an undo-redo mechanism). Then if the user requests one or more "undo", you perform

```
history.item.reaction.call ([])
history.back
```

undo

as many times as needed, but of course not going further back than the first item. For a "redo" request after one or more "undo", perform

```
history.forth
history.item.action.call ([])
```

redo

OPERATION with generic action/reaction

```
class OPERATION create
     make
feature {NONE}
     make (a_action: like action;
                a_reaction: like reaction)
          do
               action := a_action
               reaction := a_reaction
          end

feature
     execute
          do
               action.call
          end

     action: PROCEDURE

     reaction: PROCEDURE
end
```

Same as:
PROCEDURE[TUPLE]

Text editor requiring undo/redo

```
class OPERATION create
    make
feature {NONE}
    make (a_action: like action;
          a_reaction: like reaction)
        do
            action := a_action
            reaction := a_reaction
        end

feature
    execute
        do
            action.call
        end

    action: PROCEDURE

    reaction: PROCEDURE
end
```

```
feature -- text editor
    text: STRING

    num_chars: INTEGER
        -- number of characters of last insertion in `text'

    insert(s: STRING; i: INTEGER)
            -- put line `s' i times into `text'
        do
            across 1 |..| i as cr loop
                text.append (s + "%N")
            end
            num_chars := (s.count + 1)*i
        end

    reverse_insert(i: INTEGER)
            -- remove the last `i' lines in text
        do
            text.remove_tail (num_chars)
        end

    history: LIST[OPERATION]
```

Arguments of reverse_insert: TUPLE[INTEGER]

Arguments of insert: TUPLE[STRING, INTEGER]

```
force, extend (v: like item)
        -- Add `v' to end.
        -- Do not move cursor.
```

```
text: STRING

num_chars: INTEGER
    -- number of characters
    -- of last insertion in `text'

insert ..
reverse_insert ..
```

```
t1: BOOLEAN
    local
        o1, o2: OPERATION
    do
        text := ""

        -- store first text insert
        create o1.make (agent insert("first line", 2),
                        agent reverse_insert (num_chars))
        history.extend (o1)
        history.forth
        o1.execute -- now execute it

        -- store second text insert
        create o2.make (agent insert("second line", 3),
                        agent reverse_insert (num_chars))
        history.extend (o2)
        history.forth
        o2.execute -- now execute it

        -- now undo second insert
        history.item.reaction.call
        history.back

        -- now redo second insert
        history.forth
        history.item.action.call
```

Store action (for execute and a later redo) and reaction (for undo)

Execute action

Undo action

Debugger just before the first undo

```
first line
first line
second line
second line
second line
```

text	first line%Nfirst line%Nsecond line%Ns…	STRING_8	0x1091430E8
history[1]	<0x109143140>	OPERATION	0x109143140
action	<0x109143178>	PROCEDURE [!TUPLE]	0x109143178
reaction	<0x109143180>	PROCEDURE [!TUPLE]	0x109143180
Once routines			
history[2]	<0x109143148>	OPERATION	0x109143148
action	<0x1091431A8>	PROCEDURE [!TUPLE]	0x1091431A8
Agent	insert (s: STRING_8; i: INTEGER_…		
calc_rout_…	0x0	POINTER	
closed_op…	<0x1091431D8>	TUPLE [!TEST_UNDO, …	0x1091431D8
object_c…	False		
1	<0x1091430A0>	TEST_UNDO	0x1091430A0
2	second line	STRING_8	0x1091431F0
3	3	INTEGER_32	

reaction	<0x1091431B0>	PROCEDURE [!TUPLE]
Agent	reverse_insert (i: INTEGER_32)	
calc_rout_…	0x0	POINTER
closed_op…	<0x109143200>	TUPLE [!TEST_UNDO, …
object_c…	False	
1	<0x1091430A0>	TEST_UNDO
2	22	INTEGER_32

Keeping arguments open

- An agent can have both “closed” and “open” arguments.
- Closed arguments set at time of agent definition; open arguments set at time of each call.
- To keep an argument open, just replace it by a question mark:

```
u := agent a0.f (a1, a2, a3)
                -- All closed (as before)
w := agent a0.f (a1, a2, ?)
x := agent a0.f (a1, ?, a3)
y := agent a0.f (a1, ?, ?)
z := agent a0.f (?, ?, ?)
```

Agent types

- Reminder:

  PROCEDURE [*ARGS* –> *TUPLE*]
  a0: *C* having feature *f*

  f (*x1*: *T1*; *x2*: *T2*; *x3*: *T3*)
  -- in some class C
  do ... end

agent *a0*.*f* (*a1*, *a2*, *a3*) *PROCEDURE* [*TUPLE*] -- All closed

agent *a0*.*f* (*a1*, *a2*, ?) *PROCEDURE* [*TUPLE* [*T3*]]

agent *a0*.*f* (*a1*, ?, *a3*) *PROCEDURE* [*TUPLE* [*T2*]]

agent *a0*.*f* (*a1*, ?, ?) *PROCEDURE* [*TUPLE* [*T2*, *T3*]]

agent *a0*.*f* (?, ?, ?) *PROCEDURE* [*TUPLE* [*T1*, *T2*, *T3*]]

Calling an agent

a0: C; *a1*: *T1*; *a2*: T2; *a3*: *T3*

u :=**agent** *a0*.*f* (*a1*, *a2*, *a3*) *PROCEDURE* [*TUPLE*]

u.*call* ([])

v := **agent** *a0*.*f* (*a1*, *a2*, ?) *PROCEDURE* [*TUPLE* [*T3*]]

v.*call* ([*a3*])

w := **agent** *a0*.*f* (*a1*, ?, *a3*) *PROCEDURE* [*TUPLE* [*T2*]]

w.*call* ([*a2*])

x := **agent** *a0*.*f* (*a1*, ?, ?) *PROCEDURE* [*TUPLE* [*T2*, *T3*]]

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

y := **agent** *a0*.*f* (?, ?, ?) *PROCEDURE* [*TUPLE* [*T1*, *T2*, *T3*]]

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

Calling an agent: iterator

```
integer_list: LIST[INTEGER]
employee_list: LIST[EMPLOYEE]

all_positive, all_married: BOOLEAN


all_positive := integer_list.for_all (agent is_positive (?))


all_married :=
      employee_list.for_all (agent {EMPLOYEE}.is_married)
```

Iterators

- In class LINEAR [G], ancestor to all classes representing lists, sequences etc.

```
item: G -- Item at current position

for_all (test: FUNCTION [ANY, TUPLE [G], BOOLEAN]): BOOL
                -- Do all items satisfy test?
        do
                from
                        start
                        Result := True
                until
                        off or not Result
                loop
                        Result := test.item ([item])
                        forth
                end
        end
```

Iterators (cont'd)

for_all

there_exists

do_all

do_if

do_while

do_until

Advanced Info about Tuples & Agents

For more details on tuples and agents see

- See Touch of Class (online Steacie, ch 17)
  - Also explains how agents relate to the Lambda Calculus
- http://eiffel.eecs.yorku.ca (see tuples and agents)
  - Old Notation (in the above)
  - See above also for new notation