Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 13

Tuples

In mathematics, computer science, linguistics, and philosophy a tuple is an ordered list of elements. In set theory, an (ordered) n-tuple is a sequence (or ordered list) of elements, where n is a non-negative integer.

Wikipedia, Tuple

For example:

(2, 1, 4, 5)

(cat, dog)

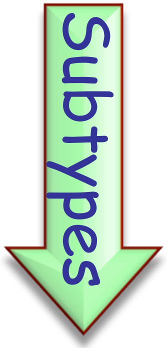
()

Tuples in Eiffel

- A tuple of type *TUPLE [A, B, C]* is a sequence of at least three values, first of type *A*, second of type *B*, third of type *C*.
- In this case possible tuple values that conform are:
  - *[a, b, c], [a, b, c, x]*,...

  where a is of type *A*, b of type B, c of type *C* and x of some type X

- Tuple types (for any types *A*, *B*, *C*, ...):



TUPLE
TUPLE [A]
TUPLE [A, B]
TUPLE [A, B, C]
...

Tuple conformance

```
tuple_conformance
    local
        t0: TUPLE
        t2: TUPLE [INTEGER, INTEGER]
    do
        create t2
        t2 := [10, 20]
        t0 := t2
        print (t0.item (1).out + "%N")
        print (t0.item (3).out)
    end
```

Not necessary in this case

Implicit creation

Runtime error, but will compile

Labeled Tuples

- Tuples may be declared with labeled arguments:

```
tuple: TUPLE [food: STRING; quantity: INTEGER]
```

- Same as an unlabeled tuple:

```
TUPLE [STRING, INTEGER]
```

but provides easier (and safer!) access to its elements:

May use

```
Io.print (tuple.food)
```

instead of

```
Io.print (tuple.item (1))
```

Assigning tuple values

```
t1 : TUPLE [INTEGER, INTEGER, STRING]
n : INTEGER
…
if attached {INTEGER} t1.item (1) as comp then
    n := comp
end

-- or

n := t1.integer_32_item (1)
```

What are agents in Eiffel?

- Objects that represent operations

- Can be seen as operation wrappers

- Similar to
  - delegates in C#
  - anonymous inner classes in Java < 7
  - closures in Java 7
  - function pointers in C
  - functors in C++

Agent definition

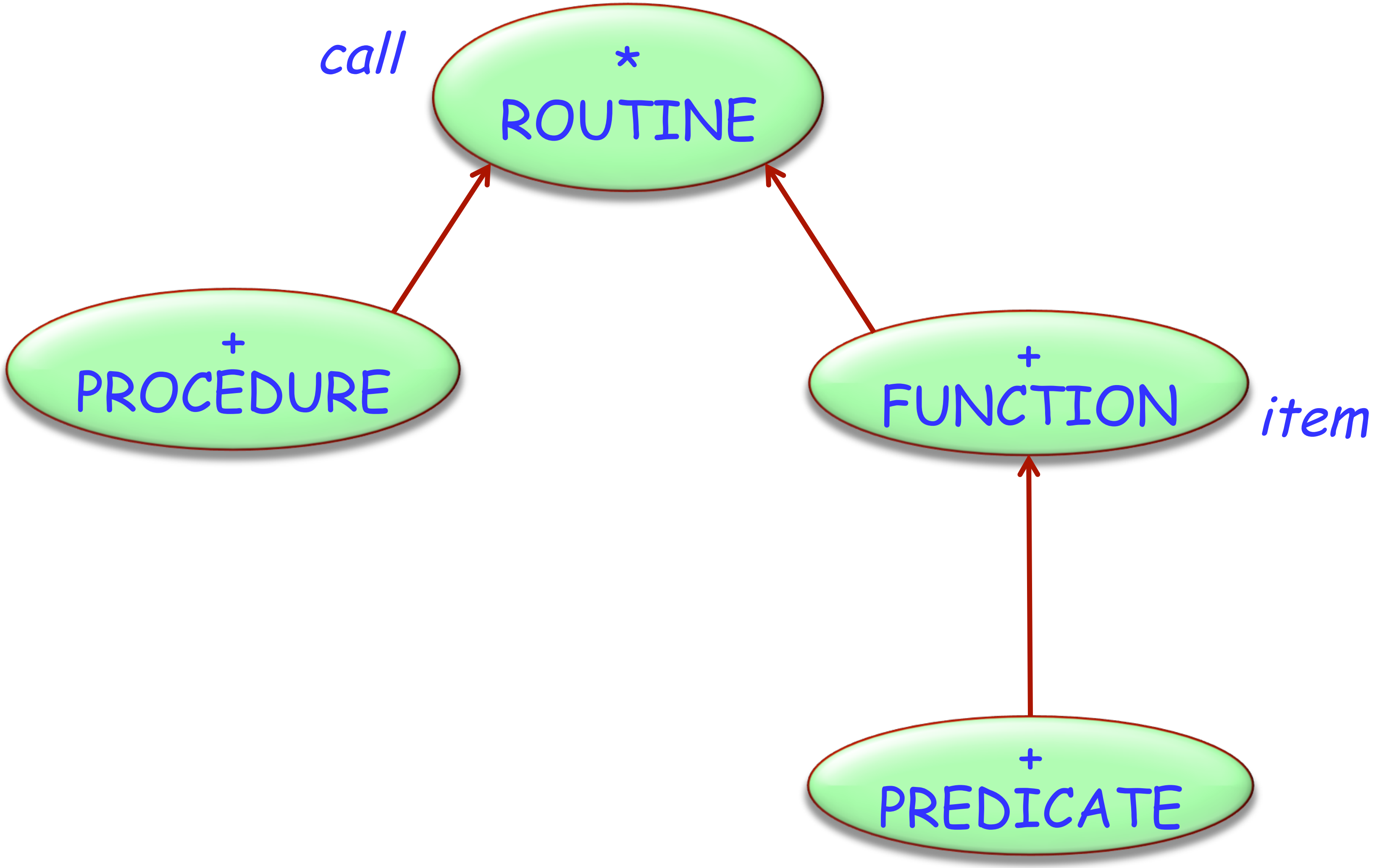
- Every agent has an associated routine, which the agent wraps and is able to invoke

- To get an agent, use the **agent** keyword
  e.g. a_agent := **agent** my_routine

- This is called agent definition

- What's the type of a_agent?

EiffelBase classes representing agents



Agent Type Declarations

p: PROCEDURE [ANY, TUPLE]

Agent representing a procedure belonging to a class that conforms to ANY. At least 0 open arguments

q: PROCEDURE [C, TUPLE [X, Y, Z]]

Agent representing a procedure belonging to a class that conforms to C. At least 3 open arguments

f: FUNCTION [ANY, TUPLE [X, Y], RES]

Agent representing a function belonging to a class that conforms to ANY. At least 2 open arguments, result of type *RES*

Open and closed agent arguments

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

```
u := agent  a0.f (a1, a2, a3)        -- All closed
v := agent a0.f (a1, a2, ?)
w := agent a0.f (a1, ?, a3)
x := agent a0.f (a1, ?, ?)
y := agent a0.f (?, ?, ?)
z := agent {C}.f (?, ?, ?)           -- All open
```

Agent Calls

An agent invokes its routine using the feature "call"

```
f (x1: T1; x2: T2; x3: T3)
    -- defined in class C with
    -- a0: C; a1: T1; a2: T2; a3: T3
```

Arguments in excess, if any, are ignored

u := **agent** a0.f (a1, a2, a3)	PROCEDURE [C, TUPLE]
v := **agent** a0.f (a1, a2, ?)	PROCEDURE [C, TUPLE [T3]]
w := **agent** a0.f (a1, ?, a3)	PROCEDURE [C, TUPLE [T2]]
x := **agent** a0.f (a1, ?, ?)	PROCEDURE [C, TUPLE [T2, T3]]
y := **agent** a0.f (?, ?, ?)	PROCEDURE [C, TUPLE [T1,T2,T3]]
z := **agent** {C}.f (?, ?, ?)	PROCEDURE [C, TUPLE [C,T1,T2,T3]]

What are the types of the agents?

Doing something to a list

Hands-On

Given a simple ARRAY [G] class, with only the features `count' and `at' , implement a feature which will take an agent and perform it on every element of the array.

```
do_all (do_this : PROCEDURE[ANY, TUPLE[G]])
        local
                i : INTEGER
        do
                from
                        i := 1
                until
                        i > count
                loop
                        do_this.call ([at (i)])
                        i := i + 1
                end
        end
```

For-all quantifiers over lists

Hands-On

```
for_all (pred : PREDICATE [ANY, TUPLE[G]]): BOOLEAN
        local
                i : INTEGER
        do
                Result := True
                from
                        i := 1
                until
                        i > count or not Result
                loop
                        Result := pred.item ([at (i)])
                        i := i + 1
                end
        end
```

Using inline agents

We can also define our agents as-we-go!

Applying this to the previous `for_all' function we made, we can do:

```
for_all_ex (int_array : ARRAY [INTEGER]): BOOLEAN
        local
                greater_five: PREDICATE [ANY, TUPLE [INTEGER]]
        do
                greater_five := agent (i : INTEGER): BOOLEAN
                                do
                                        Result := i > 5
                                end
                Result := int_array.for_all (greater_five)
        end
```

Problems with Agents/Tuples

We have already seen that TUPLE [A,B] conforms to TUPLE [A]. This raises a problem. Consider the definition:

```
f (proc : PROCEDURE [ANY, TUPLE [INTEGER]])
        do
                proc.call ([5])
        end
```

Are we allowed to call this on something of type PROCEDURE [ANY, TUPLE [INTEGER, INTEGER]] ?

Yes! Oh no… that procedure needs at least TWO arguments!

Runtime error (compiles fine)