ETH Zürich

Chair of Software Engineering

Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 11

Today

- Basic Data-structures
  - Arrays
  - Linked Lists
  - Hashtables
- Tuples
- Agents
- Agents and Data-structures

Arrays

An array is a very fundamental data-structure, which is very close to how your computer organizes its memory. An array is characterized by:

- Constant time random reads
- Constant time random writes
- Costly to resize (including inserting elements in the middle of the array)
- Must be indexed by an integer
- Generally very space efficient

In Eiffel the basic array class is generic, *ARRAY [G]*.

Using Arrays

Hands-On

Which of the following lines are valid?
Which can fail, and why?

- my_array : ARRAY [STRING] — Valid, can't fail
- my_array ["Fred"] := "Sam" — Invalid
- my_array [10] + "'s Hat" — Valid, can fail
- my_array [5] := "Ed" — Valid, can fail
- my_array.force ("Constantine", 9) — Valid, can't fail

Which is not a constant-time array operation?

Linked Lists

- Linked lists are one of the simplest data-structures
- They consist of linkable cells

```
class LINKABLE [G]

create
    set_value

feature
    set_value (v : G)
        do
            value := v
        end

    value : G

    set_next (n : LINKABLE[G])
        do
            next := n
        end

    next : LINKABLE [G]
end
```

Using Linked Lists

Hands-On

Supposing you keep a reference to only the head of the linked list, what is the running time (using big O notation) to:

- Insert at the beginning — O (1)
- Insert in the middle — O (n)
- Insert at the end — O (n)
- Find the length of the list — O (n)

What simple optimization could be made to make end-access faster?

Hashtables

Hashtables provide a way to use regular objects as keys (sort of like how we use INTEGER "keys" in arrays). This is essentially a trade-off:

- we have to provide a *hashing function* ☹
- hashing function should be good (minimize collision) ☹
- our hashtable will always take up more space than it needs to ☹

Good points about Hashtables

Hands-On

Hashtables aren't all that bad though, they provide us with a great solution: they can store and retrieve objects quickly by key! This is a *very* common operation.

For each, list what the key and values could be:

- A telephone book — Name → Telephone Number
- The index of a book — Concept → Page
- Google search — Search String → Websites

Would you use a hashtable or an array for storing the pages of a book?

Tuples

- 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]
  ...

Labeled Tuples

- Tuples may be declared with labelled arguments:

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

- Same as an unlabeled tuple:

  TUPLE [STRING, INTEGER]

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

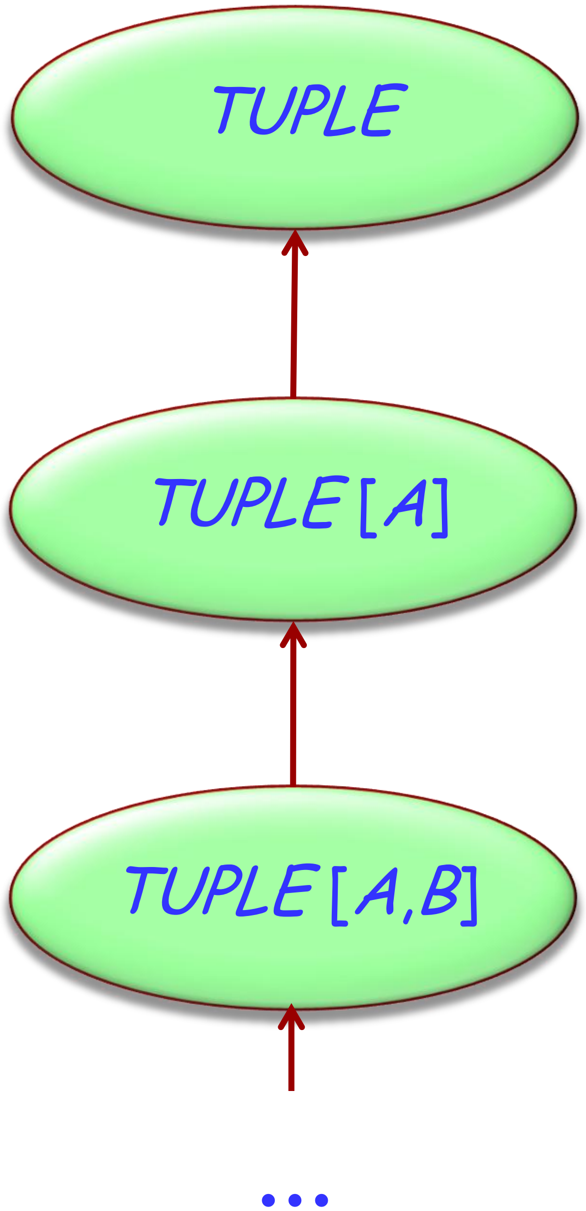
  May use

  io.print (tuple.food)

  instead of

  io.print (tuple.item(1))

Tuple Inheritance



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 (create t2)

Implicit creation (t2 := [10, 20])

Runtime error, but will compile (print (t0.item (3).out))

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

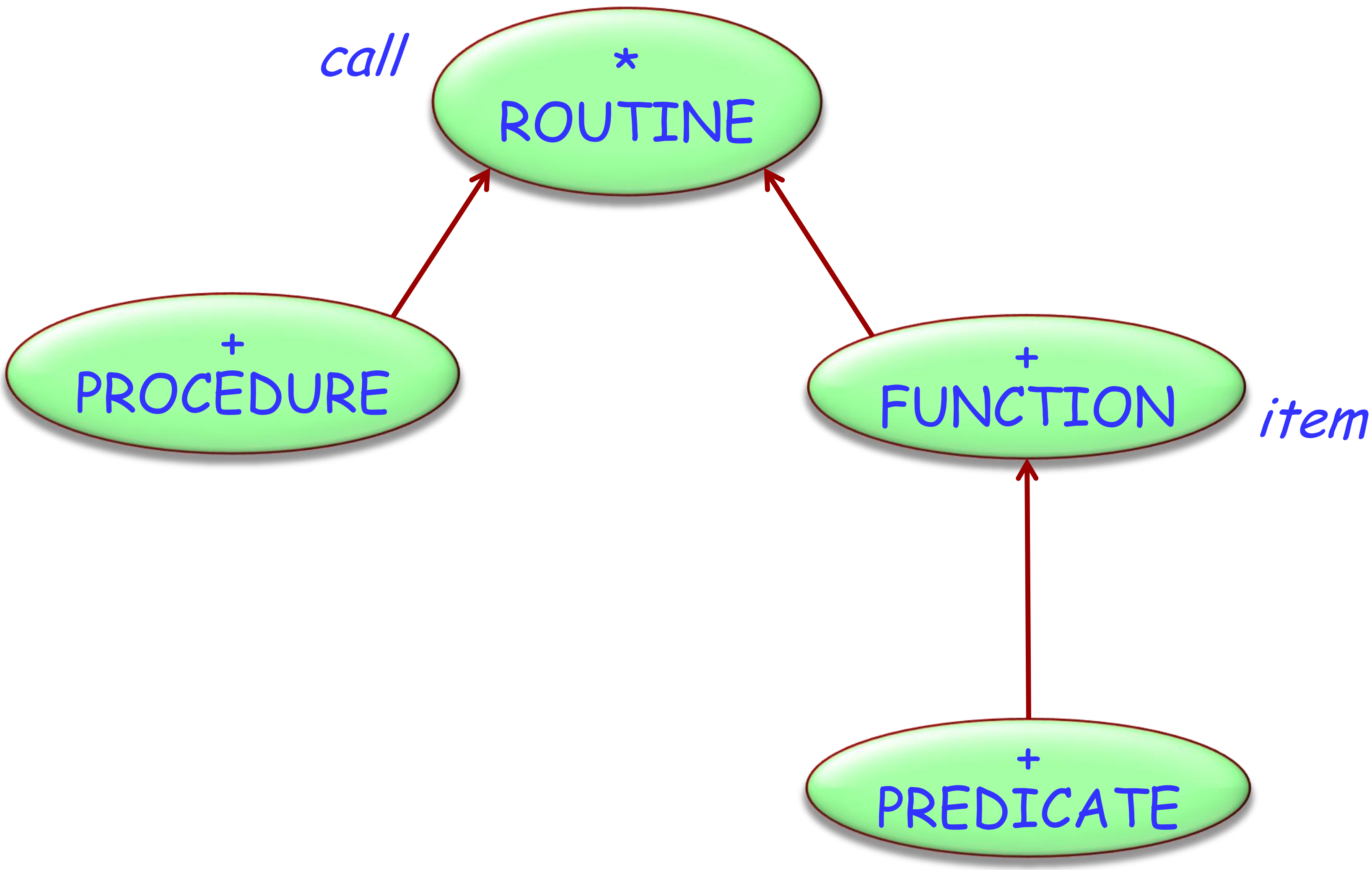
- Every agent has an associated routine, the one that the agent wraps and is able to invoke

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

- This is called agent definition

- What's the type of an_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 set at agent definition time
  - open arguments 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
w := agent a0.f (a1, a2, ?)
x := agent a0.f (a1, ?, a3)
y := agent a0.f (a1, ?, ?)
z := agent a0.f (?, ?, ?) -- All open
```

Agent Calls

An agent invokes its routine using feature "call"

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

```
u := agent a0.f (a1, a2, a3)
v := agent a0.f (a1, a2, ?)
w := agent a0.f (a1, ?, a3)
x := agent a0.f (a1, ?, ?)
y := agent a0.f (?, ?, ?)
```

PROCEDURE [C, TUPLE]

PROCEDURE [C, TUPLE [T3]]

PROCEDURE [C, TUPLE [T2]]

PROCEDURE [C, TUPLE [T2, T3]]

PROCEDURE [C, TUPLE [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]])
        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!