Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 13
Today

- Mock exam 2 review
- Tuples and agents
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.

For example:

\[(2, 1, 4, 5)\]
\[(\text{cat, dog})\]
\[()\]
Tuples in Eiffel

- A tuple of type \texttt{TUPLE [A, B, C]} is a sequence of at least three values, first of type \texttt{A}, second of type \texttt{B}, third of type \texttt{C}.
- In this case possible tuple values that conform are:
  - \([a, b, c], [a, b, c, x],...\)
  - where \(a\) is of type \texttt{A}, \(b\) of type \texttt{B}, \(c\) of type \texttt{C} and \(x\) of some type \texttt{X}

- Tuple types (for any types \texttt{A, B, C, ... }):
  \[
  \text{TUPLE}
  \text{TUPLE [A]}
  \text{TUPLE [A, B]}
  \text{TUPLE [A, B, C]}
  ...
  \]
tuple_conformance

**local**

\[ t0: \text{TUPLE} \]
\[ t2: \text{TUPLE} \ [\text{INTEGER, INTEGER}] \]

**do**

\[ \text{create} \ t2 \]
\[ t2 := [10, 20] \]
\[ t0 := t2 \]

\[ \text{print} \ (t0.\text{item}(1).\text{out} + "\%N") \]
\[ \text{print} \ (t0.\text{item}(3).\text{out}) \]

**end**

Not necessary in this case

Implicit creation

Runtime error, but will compile
Labeled Tuples

- Tuples may be declared with labeled arguments:
  \[
  \text{tuple: TUPLE [food: STRING; quantity: INTEGER]}
  \]

- Same as an unlabeled tuple:
  \[
  \text{TUPLE [STRING, INTEGER]}
  \]
  but provides easier (and safer!) access to its elements:
  May use
  \[
  \text{Io.print (tuple.food)}
  \]
  instead of
  \[
  \text{Io.print (tuple.item (1))}
  \]
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

- **ROUTINE**
  - **PROCEDURE**
  - **FUNCTION**
  - **PREDICATE**

- *call*
- **item**
Agent Type Declarations

\( p: \text{PROCEDURE [ANY, TUPLE]} \)
Agent representing a procedure belonging to a class that conforms to ANY. At least 0 open arguments

\( q: \text{PROCEDURE [C, TUPLE [X, Y, Z]]} \)
Agent representing a procedure belonging to a class that conforms to C. At least 3 open arguments

\( f: \text{FUNCTION [ANY, TUPLE [X, Y], RES]} \)
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”

\[ f(x_1: T_1; x_2: T_2; x_3: T_3) \]
-- defined in class \( C \) with
-- \( a_0: C; a_1: T_1; a_2: T_2; a_3: T_3 \)

\[ u := \text{agent } a_0.f(a_1, a_2, a_3) \]
\[ v := \text{agent } a_0.f(a_1, a_2, ?) \]
\[ w := \text{agent } a_0.f(a_1, ?, a_3) \]
\[ x := \text{agent } a_0.f(a_1, ?, ?) \]
\[ y := \text{agent } a_0.f(?, ?, ?) \]
\[ z := \text{agent } \{C\}.f(?, ?, ?) \]

What are the types of the agents?

---

Arguments in excess, if any, are ignored

PROCEDURE \([C, \text{TUPLE}]\)
PROCEDURE \([C, \text{TUPLE }[T_3]]\)
PROCEDURE \([C, \text{TUPLE }[T_2]]\)
PROCEDURE \([C, \text{TUPLE }[T_2, T_3]]\)
PROCEDURE \([C, \text{TUPLE }[T_1, T_2, T_3]]\)
PROCEDURE \([C, \text{TUPLE }[C,T_1, T_2, T_3]]\)
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.

\[\text{do}_\text{all} \ (\text{do}_\text{this} : \text{PROCEDURE}[\text{ANY}, \text{TUPLE}[G]])\]

\[
\text{local}
\]

\[
\text{i} : \text{INTEGER}
\]

\[
\text{do}
\]

\[
\text{do} \ \ \text{from} \ 
\]

\[
\text{i} := 1
\]

\[
\text{until} \ 
\]

\[
\text{i} > \text{count}
\]

\[
\text{loop}
\]

\[
\text{do}_\text{this}.\text{call} \ ([\text{at} (i)])
\]

\[
\text{i} := \text{i} + 1
\]

\[
\text{end}
\]

\[
\text{end}
\]
For-all quantifiers over lists

```plaintext
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:

```plaintext
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
        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:

```plaintext
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!