Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 13
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)
- (cat, dog)
- ()
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]`
- ...

Subtypes
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

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

![Diagram showing the relationship between EiffelBase class types: ROUTINE, PROCEDURE, FUNCTION, and PREDICATE. The diagram includes arrows indicating 'call' and 'item' connections.](image)
Agent Type Declarations

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

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

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

\[
\begin{align*}
u & := \text{agent } a0.f (a1, a2, a3) & \quad \text{-- All closed} \\
v & := \text{agent } a0.f (a1, a2, ?) \\
w & := \text{agent } a0.f (a1, ?, a3) \\
x & := \text{agent } a0.f (a1, ?, ?) \\
y & := \text{agent } a0.f (?, ?, ?) \\
z & := \text{agent } \{C\}.f (?, ?, ?) & \quad \text{-- All open}
\end{align*}
\]
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.

\[ y.call([a_1, a_2, a_3]) \]
\[ x.call([a_2, a_3]) \]
\[ w.call([a_2]) \]
\[ v.call([a_3]) \]
\[ u.call([],\ldots) \]
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.

```plaintext
do_all (do_this : PROCEDURE[ANY, TUPLE[G]])
  local
    i : INTEGER
  do
doi
  from
  i := 1
  until
  i > count
  loop
    do_this.call ([at (i)])
    i := i + 1
  end
end
```
For-all quantifiers over lists

\[
\text{for\_all} \ (\text{pred} : \text{PREDICATE} [\text{ANY, TUPLE}[G]]) : \text{BOOLEAN}
\]

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

\[
\text{do}
\]

\[
\text{Result} := \text{True}
\]

\[
\text{from}
\]

\[
i := 1
\]

\[
\text{until}
\]

\[
i > \text{count} \text{ or not Result}
\]

\[
\text{loop}
\]

\[
\text{Result} := \text{pred.item} ([\text{at} (i)])
\]

\[
i := i + 1
\]

\[
\text{end}
\]

\[
\text{end}
\]
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 \( \text{TUPLE} [A, B] \) conforms to \( \text{TUPLE} [A] \). This raises a problem. Consider the definition:

\[
f (\text{proc} : \text{PROCEDURE} [\text{ANY}, \text{TUPLE} [\text{INTEGER}]])
\]
\[
\text{do}
\]
\[
\text{proc.call} ([5])
\]
\[
\text{end}
\]

Are we allowed to call this on something of type \( \text{PROCEDURE} [\text{ANY}, \text{TUPLE} [\text{INTEGER}, \text{INTEGER}] ] \)?

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

Runtime error (compiles fine)