Reusable visitor pattern

Visitor pattern is widely used. One limitation of the pattern is that all the visit-able items must be known in advance, because all visit-able items should have a process (a_visitor: VISITOR) feature in their classes.

In this exercise, you are asked to build a reusable visitor pattern using agents. Please download the Eiffel project associated with current exercise, and finish the class VISITOR by implementing the *visit* feature and *extend* feature.

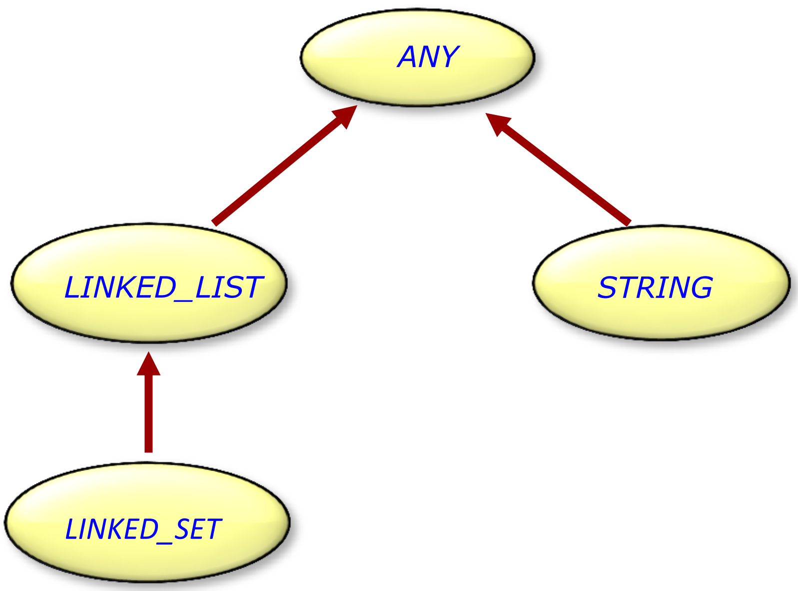
Once finished, the VISITOR class can be used in the following way:

```
class APPLICATION

create make

feature {NONE} -- Initialization

  make
    local
      l_visitor: VISITOR [ANY]
      l_str: STRING
      l_list: LINKED_LIST [ANY]
      l_any: ANY
      l_set: LINKED_SET [ANY]
    do
      create l_set.make
      l_str := "abc"
      create l_any
      create l_list.make

      create l_visitor.make
      l_visitor.extend (agent process_set)
      l_visitor.extend (agent process_string)
      l_visitor.extend (agent process_list)
      l_visitor.extend (agent process_any)

      l_visitor.visit (l_any)
      l_visitor.visit (l_list)
      l_visitor.visit (l_set)
      l_visitor.visit (l_str)
    end
```

```
    process_string (s: STRING)
      do
        io.put_string ("Processing string.%N")
      end

    process_list (l: LINKED_LIST [ANY])
      do
        io.put_string ("Processing list.%N")
      end

    process_set (l: LINKED_SET [ANY])
      do
        io.put_string ("Processing set.%N")
      end

    process_any (a: ANY)
      do
        io.put_string ("Processing any.%N")
      end

end
```

Hints

The idea of the reusable visitor pattern is to use type hierarchy of the visit-able items to decide which action to call to *visit* a given item. Using the above example, class ANY, LINKED_LIST, LINKED_SET and STRING have the following type conformance hierarchy (the hierarchy is simplified for clarity):



The reusable visitor pattern associates an agent for each type. That is: process_any for type ANY; process_list for type LINKED_LIST; process_set for type LINKED_SET and process_string for type STRING. When an item is to be processed, the visitor traverses the type conformance hierarchy until it finds an agent suitable for that item.

There are two important underlying mechanisms to support this visitor pattern: one is the ability to get type information of an object and to get type information of the open operand of an agent. In the partially implemented VISITOR class, feature type_of_object returns the type of a to-be-visited object; feature type_of_agent_argument returns the type of the open operand of an agent; feature type_conforms_to is used to decide whether one type conforms to another type.

The other supporting mechanism is the ability to represent the type conformance hierarchy. The hierarchy can be represented as a topological ordering. You can use a topological sorter, implemented in the DS_TOPOLOGICAL_SORTER class.

For example, if we want to sort ANY, LINKED_LIST, LINKED_SET and STRING in the order shown in the figure, we need to specify the following relation: ANY <- LINKED_LIST, LINKED_LIST <- LINKED_SET, STRING <- ANY. Using the DS_TOPOLOGICAL_SORTER, this translates into:

```
l_str := “string”
l_any := “any”
l_list := “list”
l_set := “set”

sorter: DS_TOPOLOGICAL_SORTER [STRING]
sorter.force (l_str)
sorter.force(l_any)
sorter.force(l_list)
sorter.force(l_set)
sorter.put_relation(l_any, l_str)
sorter.put_relation(l_any, l_list)
sorter.put_relation(l_list, l_set)
sorter.sort()
-- Access sorter.sorted_items.
```