Report on Languages for Object-Oriented Programming

Summary of the Tutorial

- **The Class Model in TOOL:**

  1. Classes, instances and generic functions:
     - object is an instance of a class
     - every class has its superclass: `<object>`. And you can assign a class's superclass when it's generating.
     - generic-function belongs to the class `<object>`. You can append new method to the generic-function. Each method is specified to a class.

  2. Eval Apply
     - the difference between eval and tool-eval is that the tool-eval has 4 more forms:
       `define-generic-function` `define-method` `define-class` `make`
     - apply adds a new procedure:
       `generic-function?`

  3. New Data Structure
     - **class representation:**
       `class name` `list of slots` `list of ancestors`
     - **generic-function representation**
       `name of the function` `list of methods defined for it`
     - **method representation**
       `specializer` `procedure`
       And the specializer is a list of classes

  4. Definitions:

     1. Defining Generic functions and methods:
        `(define-generic-function name)`
        name is the name of a new generic-function. In this procedure, this name will be combined to a new generic-

function instance. And the instance will be generated by `(make-generic-function name)`.
`(define-method generic-function (params-and-classes) . body)`
params-and-classes is a list of param-class pairs.

2. Defining classes and instances:

   - **class**: we need to collect all the slots from its ancestor, so there would be a procedure called `collect-slots`
   - **make**: This one will be taken care of by the procedure '(make class slots-name-and-values).

5. Applying Generic Function:

   - `apply-generic-function` takes `generic-function` and `arguments` as arguments
   - First it will extract the applicable method from the methods' list which is suitable for the arguments' classes:
     `compute-applicable-methods-using-classes`

     1. In order to get the most proper method, we will sort it by `method-more-specific`
     2. We also need to check the classes whether they fit in the method by `method-applies-to-classes`. This method bases on:

        1. All the classes **supplied** need to be the exact classes or the **subclasses** of the **required** classes.
        2. There would be no extra classes.
        3. In order to check the ancestor, we will use the "ancestor-link".

   - Next when we get the `method`, we will use `tool-apply` upon it:
     `(tool-apply (method-procedure (car methods)) arguments)`

6. Classes for Scheme Data:
   TOOL take the original Scheme object as its object. This would be done by keeping the `scheme-object-classes` set.

7. Initial environment and driver loop
   At first, the interpreter will bind the `true`, `false` and some initial values.

Answers to Exercises:

- Tutorial Exercise 1:

  1. **Any fault in Louis plan?**
     If he put the predicate `application?` at the first place, then the program will end in error. Take the `(define x 3)` as an instance:
     The sentence `(eval (operator exp) env)` will be extended as `(eval 'define env)`, and the eval will explain the `define` as a variable.
  2. If we change every application's form to `call+application`, then we can add a new predicate like `call?` which can specify an application at first. Then it will avoid the error in prob.1

- Tutorial Exercise 2:

  1. class definition:
     `(define-class <vector> <object> xcor ycor)`
     It's superclass is for there's no direct superclass to it.
     And it has two slots: xcor and ycor.
  2. methods:
     - Add method for 2 vectors

```
(define-method + ((v1 <vector>) (v2 <vector>))
                  (make <vector>
                  (xcor (+ (get-slot v1 'xcor) (get-slot
v2 'xcor))
                  (ycor (+ (get-slot v1 'ycor) (get-slot
v2 'ycor))
                  ))))
```

     - Mul method for 2 vectors(The inner method is similar)

```
(define-method * ((v1 <vector>) (v2 <vector>)) ...)
```

     - Dot method:

```
  (define-method · ((v1 <vector>) (v2 <vector>))
  (+ (* (get-slot v1 'xcor) (get-slot v2 'xcor))
     (* (get-slot v1 'ycor) (get-slot v2 'ycor))
     ))
```

- `number*vector` & `vector*number`:

```
  (define-method * ((v1 <vector>) (num <number>)))
  (define-method * ((num <number>) (v1 <vector>)))
```

- The generic function `length`:

```
  (define-generic-function length)
  (define-method length ((num <number>))
                        num)
  (define-method length ((vec <vector>))
                        (make <number>
                            (sqrt
                            (* (get-slot vec 'xcor)
                               (get-slot vec 'xcor))
                            (* (get-slot vec 'ycor)
                               (get-slot vec 'ycor)))
```

3. Tutorial Exercise 3:

    1. Guess: I think the `paramlist-element-class` will not call the `tool-eval`.
    A procedure will call it once there's something that the procedure cannot give out the meaning directly. For example, in the line 2 of the `eval-define-method` procedure, it calls the `tool-eval` for the generic function should be explained by the eval procedure.
    2. Real: the tool-eval is called.
    Here is the code:

```
  (define (paramlist-element-class p env)
    (let ((class (tool-eval (paramlist-element-class-name
p) env)))
      (if (class? class)
          class
          (error "Unrecognized class -- DEFINE-METHOD >> "
 class))))
```

- `paramlist-element-class-name` is a procedure which analyze the class name of the param. It will pass a quoted variable to the `tool-eval` and it will return the text-part of the quoted variable.
  - For example, if the quoted one is `'<object>`, then it will return `<object>`.
- The reason for the `tool-eval` 's appearance, I think, is just to reuse the code. If I don't use `tool-eval`, I could just do `text-of-quotation` to the class, but this is not so general. What's more is that, the `tool-eval` could help us to identify the parameter. If we do `text-of-quotation` to the parameter, without checking out if it's in a quoted form, then it will generate trouble.

4. Tutorial Exercise 4: Explain how the generic function dispatch the `say` procedure.
   Take the `(say fluffy 37)` as an example:
   1. Enter the `tool-eval` and explain the `say` as a generic function.
   2. `tool-apply` will do the generic-functions apply.
   3. The class of the two arguments are `<cat>` and `<number>`, for all the methods in the `say` 's methods, the sort method will put the `(say <cat> <number>)` to the first place.

5. Tutorial Exercise 5:
   - First we need to define a generic function `print`

     ```
     (define-generic-function print)
     ```

   - Second, add the method with arguments whose class is `<vector>` :

     ```
     (define-method print ((v1 <vector>))
        (display (xcor v1))
        (display " ")
        (display (ycor v1)))
     ```