

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
 Department of Electrical Engineering and Computer Science
 6.001—Structure and Interpretation of Computer Programs
 Spring 2004

Recitation 16
Object-Oriented Programming

Scheme

1. Syntax

- (a) *. args* - In order to implement variable-number-of-arguments procedures (like `+`, `list`, `append`, or `map`). End the parameter list of a `lambda` with `. args`. The variable `args` will be bound to a list of all the remaining arguments.

```
((lambda (x . args) (append x y)) '(1 2) 3 4 5)
(define (do-stuff x y . rest) ...)
(define (add . args) (fold-right + 0 args))
((lambda args (cons 'yay args)) 3 4)
```

2. Procedures

- (a) `(apply proc args)`
 Applies *proc* to *args*. It's like having written `(proc arg0 arg1 arg2 ...)`.

Object System

```
(define (make-type self arg1 arg2 ... argn)
  (let ((super1-part (make-super1 self args)
        (super2-part (make-super2 self args)
          other superclasses
          other local state )
        (lambda (message)
          (case message
            ((TYPE)
             (lambda ()
              (type-extend 'type super1-part
                           super2-part ...))) )
            other messages and methods
            (else (get-method message super1-part
                               super2-part ...))))))
```

```
(define (create-type arg1 arg2 ... argn)
  (create-instance make-type arg1 arg2 ... argn))
```

Object Procedures

These are defined in `objsys.scm`.

1. `ask` - (`ask obj msg [args...]`)
Calls the method `msg` on the object `obj` with the (optional) extra arguments. It may be used with either an instance or a handler for the `obj`.
2. `get-method` - (`get-method msg obj [objs...]`)
Attempts to acquire a method for the given message from one or more handlers.
3. `type-extend` - (`type-extend type obj [objs...]`)
Returns a type list that includes the TYPEs of all the given handlers, with the given `type` on the front of the list.

Conventions

1. All objects follow the above object skeleton. It's a `make-class` procedure that produces a **handler** for the particular class.
2. Every class must implement the TYPE method and call `type-extend`.
3. Every class must inherit from some other object (have at least one *superpart*). If the class doesn't have an obvious superclass, it should probably inherit from the `root-object`.
4. Every method has a name in the `case` statement, and it returns a *procedure*.
5. Use `ask` to call methods on an object.
6. When calling other methods on the same object (or its superclasses), you should (`ask self ...`)
7. The exception to the above rule is when the call is in method M and is calling method M on the superclass:

```
((M)
 (lambda ()
  ...
  (ask super-part 'M)
  ...))
```

By rule 6, using the required (`ask self 'M`) instead of (`ask super-part 'M`), will infinite loop.

Problems

1. Write a `food` class

- Input state is the `name`, `nutrition` value, and `good-until` time.
- Additional state is the `age` of the food, initially 0.
- Methods are:
 - `NAME` - returns the name of the food
 - `AGE` - returns the age of the food
 - `SIT-THERE` - takes an amount of time, and increases the age of the food by the amount.
 - `EAT` - return the nutrition if the food is still good; 0 otherwise.

2. Write an `aged-food` class

- Input state is the same as the `food` class, with an additional parameter, which is the `good-after` time.
- Should inherit from the `food` class.
- Methods are:
 - `SMELL` - returns `#t` if it has aged enough to be good.
 - `EAT` - returns 0 if the food is not good yet; otherwise behaves like normal food.

3. Write a `vending-machine` class

- Input state is the same as the `food` class.
- Additional state is `age` of the `vending-machine`, initially 0.
- Methods are:
 - `SIT-THERE` - takes an amount of time, and increases the age of the vending-machine by *half* that amount (it's refrigerated!).
 - `SELL-FOOD` - returns a new food instance with the appropriate name, nutrition and good-until.

4. Write `mapn`, which allows an arbitrary number of input lists¹, for example:

```
(mapn (lambda (a b c) (list c (+ a b)))
      '(1 2 3)
      '(4 5 6)
      '(first second third))
;Value: ((first 5) (second 7) (third 9))
```

You may use the regular `map` in your implementation.

¹It turns out that the regular `map` actually works like the `mapn` you wrote here.