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.

```scheme
((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

```scheme
(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)
    (make-handler
     'type
     (make-methods
      'METHOD-NAME
      method-lambda
      ...)
     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. **make-handler** - `(make-handler type method-list [parts...])
   Make a handler (a message-passing procedure) that handles an object of the given type, methods, and parents.

3. **make-methods** - `(make-methods name method name method ...)
   Builds an association-list out of the method names and lambdas. Used to build the method-list for `make-handler`.

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 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`.

3. Use `ask` to call methods on an object.

4. When calling other methods on the same object (or it’s superclasses), you should `(ask self ...)

5. The exception to the above rule is when the call is in method M and is calling method M on the superclass:

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

   By rule 4, 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
     - NUTRITION - returns the nutrition 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:
     - **SNIFF** - 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 **decaying-food** class
   
   - Input state is the same as the **aged-food** class.
   - Should inherit from the **aged-food** class.
   - Methods are:
     - **NUTRITION** - nutrition is inversely proportional to age, after the good-after time.
4. 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-HERE - 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.