Scheme

1. Syntax

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

\[
((\text{lambda} (x . \text{args}) (\text{append} x y)) '(1 2) 3 4 5)
\]

\[
(\text{define} (\text{do-stuff} x y . \text{rest}) ...)
\]

\[
(\text{define} (\text{add} . \text{args}) (\text{fold-right} + 0 \text{args}))
\]

\[
((\text{lambda} \text{args} (\text{cons} \text{’yay} \text{args})) 3 4)
\]

2. Procedures

(a) \(\text{apply proc args}\)

Applies \(\text{proc}\) to \(\text{args}\). It’s like having written \((\text{proc} \text{arg0} \text{arg1} \text{arg2} ...).\)

Object System

\[
(\text{define} (\text{make-type} \text{self} \text{arg1} \text{arg2} \ldots \text{argn})
\]

\[
(\text{let} ((\text{super1-part} (\text{make-super1} \text{self} \text{args})))
\]

\[
(\text{super2-part} (\text{make-super2} \text{self} \text{args}))
\]

\[
\text{other superfices}
\]

\[
\text{other local state}
\]

\[
(\text{make-handler}
\]

\[
\text{'type}
\]

\[
(\text{make-methods}
\]

\[
\text{'METHOD-NAME}
\]

\[
\text{method-lambda}
\]

\[
\ldots
\]

\[
\text{super1-part super2-part} \ldots))))
\]

(\text{define} (\text{create-type} \text{arg1} \text{arg2} \ldots \text{argn})
\]

(\text{create-instance} (\text{make-type} \text{arg1} \text{arg2} \ldots \text{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:

   ```scheme
   ((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 \texttt{name}, \texttt{nutrition} value, and \texttt{good−until} time.
   - Additional state is the \texttt{age} of the food, initially 0.
   - Methods are:
     - \texttt{NAME} - returns the name of the food
     - \texttt{NUTRITION} - returns the nutrition of the food
     - \texttt{AGE} - returns the age of the food
     - \texttt{SIT−THERE} - takes an amount of time, and increases the age of the food by the amount.
     - \texttt{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-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.

5. Write **mapn**, which allows an arbitrary number of input lists\(^1\), for example:

\[
\begin{align*}
\text{(mapn (lambda (a b c) (list c (+ a b)))} & \\
  '(1 2 3) & \\
  '(4 5 6) & \\
  '(\text{first second third}) & \\
; \text{Value: ((first 5) (second 7) (third 9))}
\end{align*}
\]

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

\(^1\)It turns out that the regular **map** actually works like the **mapn** you wrote here.