```
;;;
;;;    The Oz Interactive Fiction System
;;;       Copyright (c) 1990 by Carnegie Mellon University
;;;

;;;
;;;  Class Lyotard (emotional cat written in Hap and Em)
;;;
;;;==================================================================
;;;  Change Log:
;;;       9/25/91  abl  created by abl.
;;;
;;;==================================================================
;;;----------------------------------------------------------------------
;;;Problems:
;;;
;;;Notes:
;;;
;;;----------------------------------------------------------------------

(eval-when (compile load eval)
  (defmacro matcher-time-recently ()
    '(from (- (call what-time-is-it $actor) 3)
      through (call what-time-is-it $actor)))
  (defmacro matcher-time-during-last (number)
    '(from (- (call what-time-is-it $actor) ,number)
      through (call what-time-is-it $actor)))
  (defmacro matcher-condition-willing-to-interact (actor)
    '(or (and (> (call relationship $em ,actor 'like) 0)
              (not (call has-feature $em 'ignoring))
              (not (match (call feel-fear $em)
                          (list-containing (list-of ,actor ?v))))
              (not (match (call feel-anger $em)
                          (list-containing (list-of ,actor ?v)))))
         (< $food $famished-threshold)))
  (defmacro matcher-condition-feel-aggressive (reason-pat intensity-pat)
    '(or (match (call feel-fear $em)
                (list-containing (list-of ,reason-pat ,intensity-pat)))
         (match (call feel-anger $em)
                (list-containing (list-of ,reason-pat ,intensity-pat)))))
  )

(class lyotard
  (hap-em-cat-actor)
  (nouns "Lyotard" "kitty" "cat")
  (slots
   (sex 'male)
   (size 12)
   (weight 15)
   (strength 10)
   (hair-color 'black)
   (eye-color 'green)
   (voice 'lyotard)
   (emotional-annoyance-threshold 2)
   (emotional-serious-threshold 5)
   (emotional-overload-threshold 20)
   (famished-threshold 10)
   (hunger-threshold 20)
   (satisfied-threshold 45)
   (full-threshold 60)
   ;;  current amount of food (start off almost hungry)
   (food 50)

   (production-memory
    (hap-program

;;------------------------------------------
;;  Sets up global constants and variables
;;------------------------------------------
(sequential-production set-up-global-constants-and-variables ()
  ;;  initialize global constants and variables for processing
  (mental-act (setf $$me (call node-self (slot sr-sight $actor))))
  ;;  say something so the inferences in the ism can learn my own voice.
  (act actor self verb '*say
       object (make speech text "I am learning my voice"
                    voice (slot voice $actor)))
  (with ignore-failure
    (let ((a 1))
      (act actor self verb '*go object *north*)
      (act actor self verb '*go object *east*)
      (act actor self verb '*go object *west*)
      (act actor self verb '*go object *south*)
      (act actor self verb '*go object *south*)
      (act actor self verb '*go object *north*)
      (act actor self verb '*go object *east*)
      (act actor self verb '*go object *south*)))
  (mental-act (progn ;(make player name 'player voice 'player
                     ;       nouns '("Player"))
                     (setf actor-list (list player lyotard))
                     (place player 'in kitchen))))

;;------------------------------------------
;;  wait-a-while <ticks>
;;------------------------------------------
(sequential-production wait-a-while (ticks)
  (cond ((= $$ticks 0) (with (success-test t) (wait)))
        (t (act actor self verb '*wait))))

;;------------------------------------------
;;  trapped
;;------------------------------------------
(sequential-production trapped ()
  (act actor self verb '*meow))

;;------------------------------------------
;;  want-to-eat
;;------------------------------------------

;;  set up a demon to eat when you can, and then actively position
;;  yourself so that you can.  It succeeds only by satisfying the
;;  success test that invokes it.
(parallel-production want-to-eat ()
  (with (priority-modifier .5) effect-only
    (demon (and (or (and (or (can-see (a food ?f color ?c) in
                                      (a container ?container closed nil)
                                      on (a solid) supporting (node $$me))
                             (can-see (a food ?f color ?c) in
                                      (a container ?container closed nil)
                                      in (a container) containing
                                      (node $$me)))
                         (or (not (match ?container (an actor)))
                             (and (match ?container (an actor))
                                  (did-see (an act actor ?container
                                                verb '*give
                                                object ?f)))))
                    (can-see (a food ?f color ?c) in (a container)
                             containing (node $$me))
                    (can-see (a food ?f color ?c) on (a solid)
                             supporting (node $$me)))
                (member ?c '(black brown red)))
```

```
      (act actor self verb '*eat object $$f)
      (mental-act (setf $food (+ $food 10)))))
    (subgoal position-self-to-eat))

  (sequential-production wait-while-eating ()
    (with (success-test
           ;; no food nearby
           (or (not (and (or (and (or (can-see (a food ?f color ?c) in
                                                (a container ?container closed nil)
                                                on (a solid) supporting (node $$me))
                                       (can-see (a food ?f color ?c) in
                                                (a container ?container closed nil)
                                                in (a container) containing
                                                (node $$me)))
                                  (or (not (match ?container (an actor)))
                                      (and (match ?container (an actor))
                                           (did-see (an act actor ?container
                                                        verb '*give
                                                        object ?f)))))
                             (can-see (a food ?f color ?c) in (a container)
                                      containing (node $$me))
                             (can-see (a food ?f color ?c) on (a solid)
                                      supporting (node $$me)))
                         (member ?c '(black brown red))))
               ;; full
               (> $food $satisfied-threshold)))
           (wait)))

  (sequential-production position-self-to-eat ()
         (specificity 1)
         (precondition
          (and $$bowl-empty-or-full-and-cant-reach
               (know-of-in-ism (a player ?p))
;               (willing-to-interact ?p)
               (or (know-of-in-ism (a container ?loc) containing
                                   (a food-bowl ?fb))
                   (know-of-in-ism (a solid ?loc) supporting
                                   (a food-bowl ?fb)))))
    (subgoal find $$p)
    (subgoal tell-human 'come $$loc $$p)
    ;; in case can't quite get to the food bowl.
    (with ignore-failure
      (subgoal goto $$loc 'quickly))
    ;; make sure at least made it to the right room or abandon this
    ;; approach
    (with (success-test
           (and (can-see (a location ?l) location (a food-bowl))
                (can-see (a location ?l) location (node $$me))))
      (subgoal fail))
    (subgoal tell-player-to-feed-me-at-bowl))

  (sequential-production tell-player-to-feed-me-at-bowl ()
         (precondition (or (and (can-see (a container ?place)
                                         containing (node $$me))
                                (match 'in ?in/on))
                           (and (can-see (a solid ?place)
                                         supporting (node $$me))
                                (match 'on ?in/on))))
         (context-condition (can-see (a food-bowl) nearby (node $$me)))
    (with (success-test (can-see (a player) nearby (node $$me)))
      (subgoal wait-a-while (+ (random 3) 2)))
    (subgoal tell-player-to-feed-me-at-bowl-when-player-here))

  (sequential-production tell-player-to-feed-me-at-bowl-when-player-here ()
         (precondition (can-see (a player ?p) nearby (node $$me)))
```

```
    (subgoal tell-human 'feed-me $$fb $$p $$in/on $$place)
    (subgoal wait-while-eating)
    (with (success-test (> $food $satisfied-threshold))
      (subgoal fail)))

  (sequential-production position-self-to-eat ()
         (precondition
          (and (or (know-of-in-ism (a food color ?c) on
                                   (a solid ?place size ?size))
                   (and (or (know-of-in-ism (a food color ?c) in
                                            (a container ?cont closed nil
                                               opening-size ?osize) on
                                               (a solid ?place size ?size))
                            (know-of-in-ism (a food color ?c) in
                                            (a container ?cont closed nil
                                               opening-size ?osize) in
                                               (a container ?place opening-size ?size)))
                        (< ?osize $size)
                        (> ?osize (/ $size 4)))
                   (know-of-in-ism (a food color ?c) in
                                   (a container ?place opening-size ?size)))
               (member ?c '(brown black red) :test #'eq)
               (> ?size $size)))
     (subgoal goto $$place nil)
     (subgoal wait-while-eating)
     (with (success-test (> $food $satisfied-threshold))
       (subgoal fail)))

  (sequential-production position-self-to-eat ()
         (specificity 2)
         (precondition (and (not $$bowl-empty-or-full-and-cant-reach)
                            (or (know-of-in-ism (a container ?loc)
                                                containing
                                                (a food-bowl ?fb))
                                (know-of-in-ism (a solid ?loc) supporting
                                                (a food-bowl ?fb)))))
    (with ignore-failure
      (subgoal goto $$loc nil))
    (cond
     ((and (can-see (a food color ?c) in (node $$fb))
           (member ?c '(brown black red) :test #'eq)
           (or (can-see (node $$fb) in (a container) containing (node $$me))
               (can-see (node $$fb) on (a solid) supporting (node $$me))))
      (with (success-test t) (wait)))
     (t (with (success-test (not $$bowl-empty-or-full-and-cant-reach))
          (mental-act (setf $$bowl-empty-or-full-and-cant-reach t)))
        (subgoal complain (+ (random 3) 3))))
    (subgoal wait-while-eating)
    (with (success-test (> $food $satisfied-threshold))
      (subgoal fail)))

  (sequential-production complain (ticks)
    (cond
     ((= $$ticks 0) (with (success-test t) (wait)))
     (t (par-cond
         (t (act actor self verb '*meow))
         (t (act actor self verb '*meow))
         (t (act actor self verb '*meow))
         (t (act actor self verb '*wait)))
        (subgoal complain (- $$ticks 1)))))

  ;;-----------------------------------------
  ;; walk-around-destructively <~ticks>
  ;;-----------------------------------------
  (sequential-production walk-around-destructively-once (~ticks)
```

```
    (cond
     ((= $$~ticks 0) (with (success-test t) (wait)))
     (t (subgoal walk-around-destructively-once)
        (subgoal walk-around-destructively (- $$~ticks 1)))))

  (sequential-production walk-around-destructively-once ()
         (precondition (and (can-see (a solid ?obj size ?size) on (a solid)
                                     supporting (node $$me))
                            (< ?size (* $size 2))))
    (with (importance 6)
      (subgoal destruct $$obj)))

  (sequential-production walk-around-destructively-once ()
         (precondition (and (can-see (a solid ?target size ?t-size)
                                     on (a solid ?surface size ?s-size))
                            (< ?t-size (* $size 2))
                            (> ?s-size $size)))
    (subgoal goto $$surface 'nil)
    (with (importance 6)
      (subgoal destruct $$target)))

  (sequential-production walk-around-destructively-once ()
         (precondition (and (or (can-see (a container ?target size ?t-size)
                                         on (a solid ?surface size ?s-size))
                                (can-see (a container ?target size ?t-size)
                                         in (a container ?surface
                                               opening-size ?s-size)))
                            (< ?t-size (* $size 2))
                            (> ?s-size $size)))
    (subgoal goto $$surface 'nil)
    (with (importance 6)
      (subgoal destruct $$target)))

  (sequential-production destruct (obj)
    (par-cond
     (t (act actor self verb '*push object $$obj))
     (t (act actor self verb '*nudge object $$obj))
     (t (act actor self verb '*hit object $$obj))))

  ;;-----------------------------------------
  ;;  clean-self <ticks>
  ;;-----------------------------------------
  (sequential-production clean-self (ticks)
    (cond
      ((= $$ticks 0) (with (success-test t) (wait)))
      (t (with (importance 2)
           (act actor self verb '*lick object self))
         (subgoal clean-self (- $$ticks 1)))))
  ;;-----------------------------------------
  ;;  amuse-self
  ;;-----------------------------------------
  (sequential-production amuse-self ()
    (par-cond
     ((and (call feel-anger $em) (call has-feature $em 'energetic))
      (subgoal walk-around-destructively (+ (random 3) 3)))
;     ((and (call has-feature $em 'curious)
;           (call has-feature $em 'energetic))
;       (subgoal walk-around-destructively) (+ (random 3) 3))
     ((and (call has-feature $em 'friendly)
           (know-of-in-ism (a toy-mouse ?mouse)))
      (subgoal find $$mouse)
      (subgoal play-with-object $$mouse (+ (random 3) 3)))
     ((know-of-in-ism (a ball ?ball))
        ;;      (call has-feature $em 'energetic)
      (subgoal find $$ball)
      (subgoal play-with-object $$ball (+ (random 3) 3)))
     (t ;;(and (or (call has-feature $em 'content)
        ;;            (call feel-happy $em))
        ;;        (not (call has-feature $em 'energetic)))
      (subgoal sit-in-comfy-place (+ (random 3) 3)))
     (t ;;(and (or (call has-feature $em 'content)
        ;;            (call feel-happy $em))
        ;;        (not (call has-feature $em 'energetic))
        ;;        (call has-feature $em 'curious))
      (subgoal watch-something (+ (random 3) 3)))
     ((call has-feature $em 'energetic)
      (subgoal explore (+ (random 3) 3)))
     ((not (did-see (an act actor $$me verb '*lick object $$me)
                    (during-last 20)))
      (subgoal clean-self (+ (random 3) 3)))
     ((call has-feature $em 'ignoring)
      (subgoal clean-self (+ (random 3) 3)))
     ;;  willing to interact here implies liking, not afraid of,
     ;;  not angry at, and not ignoring--cause if hunger was the
     ;;  cause, we would still be trying to get food.
     ((and (can-see (a player ?p)) (willing-to-interact ?p))
      (subgoal want-to-be-petted $$p))
     ((and (know-of-in-ism (a player ?p)) (willing-to-interact ?p)
           (call has-feature $em 'energetic))
      (subgoal find $$p)
      (subgoal want-to-be-petted $$p))))

;;-----------------------------------------
;;  want-to-be-petted <actor>
;;-----------------------------------------
(parallel-production want-to-be-petted (actor)
  (with (priority-modifier .5) effect-only
    (demon follow-human
           (and (not (and (can-see (a location ?l) location (node $$actor))
                          (can-see (a location ?l) location (node $$me))))
                (can-see (a location ?loc) location (node $$actor)))
      (subgoal goto $$loc 'nil)))
  (subgoal tell-human 'pet-me $$me $$actor))

;;-----------------------------------------
;;  sit-in-comfy-place <~ticks>
;;-----------------------------------------
(sequential-production sit-in-comfy-place (ticks)
       (precondition
        (and (collect-from
              (or (and (know-of-in-ism (a solid ?p hardness ?h))
                       (numberp ?h)
                       (< ?h 4))
                  (and (know-of-in-ism (a solid ?p warmth ?w))
                       (numberp ?w)
                       (> ?w 6))
                  (and (call feel-fear $em)
                       (know-of-in-ism (a container ?p brightness ?b))
                       (numberp ?b)
                       (< ?b 4)))
              (in ?places ?p))
             ?places))
  (let ((place (random-element $$places)))
    (subgoal goto $$place 'nil)
    (par-cond
     (t (with ignore-failure
              (act actor self verb '*sit-down)))
     (t (with ignore-failure
              (act actor self verb '*lie-down))))
    (subgoal hangout $$ticks $$place)))
```

```
(sequential-production hangout (ticks place)
       (context-condition
        (or (can-see (node $$place) containing (node $$me))
            (can-see (node $$place) supporting (node $$me))))
  (cond ((= $$ticks 0) (with (success-test t) (wait)))
        (t (with (importance 3)
             (par-cond
              (t (act actor self verb '*wait))
              (t (act actor self verb '*wait))
              (t (act actor self verb '*wait))
              (t (act actor self verb '*wait))
              (t (act actor self verb '*wait))
              (t (subgoal clean-self (+ (random 2) 2)))))
           (subgoal hangout (- $$ticks 1) $$place))))

;;------------------------------------------
;;  watch-moving-object <object> <ticks>
;;------------------------------------------
(sequential-production watch-moving-object (object ticks)
       (context-condition (and (can-see (node $$object))
                               (did-see (a solid moving ?amount-moving)
                                        (node $$object) recently)
                               ?amount-moving))
  (cond
   ((= $$ticks 0) (with (success-test t) (wait)))
   (t (par-cond
       (t (act actor self verb '*lookat object $$object)
          (subgoal watch-moving-object $$object (- $$ticks 1)))
       (t (act actor self verb '*lookat object $$object)
          (subgoal watch-moving-object $$object (- $$ticks 1)))
       (t (act actor self verb '*lookat object $$object)
          (subgoal watch-moving-object $$object (- $$ticks 1)))
       ((and (match $$object (a solid size ?size)) (< ?size $size))
        (subgoal play-with-object $$object (+ (random 4) 3)))))))

;;------------------------------------------
;;  play-with-object <object> <ticks>
;;------------------------------------------
(sequential-production play-with-object (object ticks)
       (precondition (can-see (node $$object) nearby (node $$me)))
  (cond
   ((= $$ticks 0) (with (success-test t) (wait)))
   (t (cond
       ((can-see (a solid ?object) (node $$object))
        (subgoal play-with-object-once $$object)
        (subgoal play-with-object $$object (- $$ticks 1)))
       (t (act actor self verb '*lookaround))))))

(sequential-production play-with-object-once (object)
       (precondition (or (can-see (node $$object) in (a container ?place))
                         (can-see (node $$object) on (a solid ?place))))
  (subgoal goto $$place 'nil) ; has appropriate success-test built in.
  (subgoal play-with-object-once-nearby $$object))

(sequential-production play-with-object-once-nearby (object)
       (precondition (can-see (node $$object) nearby (node $$me)))
  (par-cond
   ((and (match $$object (a solid moving ?amount-moving))
         ?amount-moving)
    (subgoal watch $$object 1))
   ((and (match $$object (a solid moving ?amount-moving))
         ?amount-moving)
    (act actor self verb '*pounce-on object $$object))
   (t (act actor self verb '*nudge object $$object))
   (t (act actor self verb '*pounce-on object $$object))
   ((and (feel-aggressive ?reason ?amount)
         (> ?amount $emotional-annoyance-threshold))
    (act actor self verb '*bite object $$object))
   ((and (feel-aggressive ?reason ?amount)
         (> ?amount $emotional-annoyance-threshold))
    (act actor self verb '*bite object $$object))
   ((and (feel-aggressive ?reason ?amount)
         (> ?amount $emotional-annoyance-threshold))
    (act actor self verb '*scratch object $$object))))

;;------------------------------------------
;;  watch-something <ticks>
;;------------------------------------------
(sequential-production watch-something (ticks)
       (specificity 1)
       (precondition
        (and (collect-from
              (or (and (can-see (a solid ?s moving ?m))
                       ?m (not (eq ?m '*not-legal-property*)))
                  (and (can-see (an actor ?s)) (not (match ?s $$me)))
                  (can-see (a window ?s)))
              (in ?stuff ?s))
             ?stuff))
  (subgoal watch (random-element $$stuff) $$ticks))

(sequential-production watch-something (ticks)
       (precondition (and (know-of-in-ism (a solid ?place size ?size
                                                  warmth ?warmth
                                                  brightness ?bri)
                                               nearby (a window ?win))
                          (numberp ?warmth) (numberp ?bri)
                          (numberp ?size)
                          (> ?warmth 6)
                          (> ?bri 6)
                          (> ?size $size)))
  (subgoal goto $$place nil)
  (subgoal watch $$win $$ticks))

(sequential-production watch-something (ticks)
       (precondition (know-of-in-ism (a player ?player)))
  (subgoal find $$player)
  (subgoal watch $$player $$ticks))

(sequential-production watch (obj ticks)
       (context-condition (can-see (node $$obj)))
  (cond ((= $$ticks 0) (with (success-test t) (wait)))
        (t (act actor self verb '*lookat object $$obj)
           (subgoal watch $$obj (- $$ticks 1)))))

;;------------------------------------------
;;  threaten <actor> <intensity>
;;------------------------------------------
(sequential-production threaten (actor intensity)
  (par-cond
   ((> $$intensity $emotional-overload-threshold)
    (act actor self verb '*scratch object $$actor))
   ((> $$intensity $emotional-overload-threshold)
    (act actor self verb '*bite object $$actor))
   (t (act actor self verb '*arch-back))
   (t (act actor self verb '*hiss-at object $$actor))))

;;------------------------------------------
;;  release-frustration <amount>
;;------------------------------------------
```

```
(sequential-production release-frustration (amount)
  (par-cond
   ((< $$amount $emotional-serious-threshold)
    (subgoal clean-self (+ (random 3) 3)))
   ((and (> $$amount $emotional-serious-threshold)
         (know-of-in-ism (a toy-mouse) (node ?mouse)))
    (subgoal find $$mouse)
    (subgoal get $$mouse)
    (subgoal drop $$mouse)
    (subgoal transfer-aggression-to-mouse $$amount (+ (random 3) 3)))))

;;-----------------------------------------
;;  transfer-aggression-to-mouse <amount> <ticks>
;;-----------------------------------------
(sequential-production transfer-aggression-to-mouse (amount ticks)
       (precondition (know-of-in-ism (a toy-mouse ?mouse)))
  (with (success-test (can-see (a toy-mouse) in (node $$me)))
    (subgoal get $$mouse))
  (subgoal aggress-mouse $$mouse $$amount $$ticks))

(sequential-production aggress-mouse (mouse amount ticks)
       (context-condition (can-see (node $$mouse) nearby (node $$me)))
  (cond
    ((= $$ticks 0) (with (success-test t) (wait)))
    (t (with (importance 2)
         (par-cond
          (t (act actor self verb '*bite object $$mouse))
          (t (act actor self verb '*lick object $$mouse)
             ;;  reduce frustration
             )
          (t (act actor self verb '*scratch object $$mouse))))
       (subgoal aggress-mouse $$mouse $$amount (- $$ticks 1)))))

;;-----------------------------------------
;;  attack-once <object>
;;-----------------------------------------
(sequential-production attack-once (object)
       (context-condition (can-see (node $$object) nearby (node $$me)))
  (par-cond
   (t (act actor self verb '*bite object $$object))
   (t (act actor self verb '*scratch object $$object))))

;;-----------------------------------------
;;  escape-run-away <object> <intensity>
;;-----------------------------------------
;;  if fear is great enough, run away.
(sequential-production escape-run-away (object intensity)
       (precondition (and (> $$intensity $emotional-annoyance-threshold)
                          (< $$intensity $emotional-overload-threshold)))
       (specificity 1)
  (with (success-test (not (can-see (node $$object) nearby (node $$me))))
    (subgoal run-away $$object))
  (act actor self verb '*lookaround adverb 'nervously))


;;  if fear is too great, do the best you can.
(sequential-production escape-run-away (object intensity)
       (specificity 0)
   (subgoal paralized-with-fear (+ (random 3) 1)))

(sequential-production paralized-with-fear (ticks)
   (cond ((= $$ticks 0) (mental-act (setf $quivering nil)))
         (t (mental-act (setf $quivering t))
            (act actor self verb '*squish-down)
            (subgoal paralized-with-fear (- $$ticks 1)))))

;;-----------------------------------------
;;  run-away from object
;;-----------------------------------------
(sequential-production run-away (object)
       (precondition
        (and
         (collect-from
          (or (and (know-of-in-ism (a container ?place brightness ?b
                                      opening-size ?s))
                   (numberp ?b) (numberp ?s) (< ?b 4) (> ?s $size))
              (know-of-in-ism (a location ?place) goable (a location)
                              location (node $$me)))
          (in ?places-to-run-to ?place))
         ?places-to-run-to))
  (subgoal goto (random-element $$places-to-run-to) 'quickly))

;;-----------------------------------------
;;  find <object>
;;-----------------------------------------
(sequential-production find (object)
       (precondition
        (or (know-of-in-ism (a container ?loc) containing (node $$object))
            (know-of-in-ism (a solid ?loc) supporting (node $$object))))
       (specificity 1)
  (with ignore-failure
    (subgoal goto $$loc nil))
  (with (success-test (can-see (node $$object) nearby (node $$me)))
    (subgoal fail)))

(sequential-production find (object)
  (with (success-test (can-see (node $$object) nearby (node $$me)))
    (subgoal explore (+ (random 3) 3)))
  (with (success-test (can-see (node $$object) nearby (node $$me)))
    (subgoal fail)))

;;-----------------------------------------
;;  get <object>
;;-----------------------------------------
;;  assumes that you are at the location of <object>
(sequential-production get (object)
       (precondition
        (or (can-see (node $$object) on (a solid) supporting (node $$me))
            (can-see (node $$object) in (a container) containing
                     (node $$me))
            (and (or (can-see (node $$object) in
                              (a container ?container closed nil)
                              on (a solid) supporting (node $$me))
                     (can-see (node $$object) in
                              (a container ?container closed nil)
                              in (a container) containing
                              (node $$me)))
                 (or (not (match ?container (an actor)))
                     (and (match ?container (an actor))
                          (did-see (an act actor ?container
                                       verb '*give
                                       object $$object)))))))
       (specificity 1)
  (act actor self verb '*take object $$object))

;;  can't get to it, but can get to what it is on so push that.
(sequential-production get (object)
       (precondition
        (and (or (can-see (node $$object) on (a solid ?p size ?size)
                          nearby (node $$me))
```

```lisp
                  (can-see (node $$object) in
                           (a container ?p opening-size ?size) nearby
                           (node $$me)))
              (< ?size $size)
              (or (can-see (node ?p) on (a solid) supporting (node $$me))
                  (can-see (node ?p) in (a container) containing
                           (node $$me)))))
  (with (success-test
         (not (or (can-see (node $$object) on (node $$p))
                  (can-see (node $$object) in (node $$p)))))
        (subgoal push $$p (+ (random 3) 3)))
  (subgoal get $$object))

;;  go get player, and try to get player to get it for me.
(sequential-production get (object)
       (precondition
        (and (or (can-see (node $$object) on (a solid ?p size ?size)
                           nearby (node $$me))
                  (can-see (node $$object) in
                           (a container ?p opening-size ?size) nearby
                           (node $$me)))
              (< ?size $size)
              (know-of-in-ism (a player ?player))
              (not (and (can-see (a location ?l) location (node ?player))
                        (can-see (node ?l) location (node $$me))))
              (willing-to-interact ?player)
              (call has-feature $em 'energetic)))
  (with (success-test (and (can-see (node $$l) location (node $$player))
                           (can-see (node $$l) location (node $$me))))
    (subgoal bring-player $$l))
  (with (success-test
         (not (or (can-see (node $$object) on (node $$p))
                  (can-see (node $$object) in (node $$p)))))
    (subgoal tell-human 'get $$object $$player 'in $$l))
  (subgoal get $$object))

;;  try to get player to get it for me if he is around
(sequential-production get (object)
       (precondition
        (and (or (can-see (node $$object) on (a solid ?p size ?size)
                           nearby (node $$me))
                  (can-see (node $$object) in
                           (a container ?p opening-size ?size) nearby
                           (node $$me)))
              (< ?size $size)
              (and (can-see (a location ?l) location (node ?p))
                   (can-see (node ?l) location (node $$me)))
              (know-of-in-ism (a player ?player))
              (willing-to-interact ?player)))
  (with (success-test
         (not (or (can-see (node $$object) on (node $$p))
                  (can-see (node $$object) in (node $$p)))))
    (subgoal tell-human 'get $$object $$player 'in $$l))
  (subgoal get $$object))

;;-----------------------------------------
;;  push <object> <ticks>
;;-----------------------------------------
(sequential-production push (object ticks)
       (context-condition
        (or (can-see (node $$object) on (a solid) supporting (node $$me))
            (can-see (node $$object) in (a container) containing
                     (node $$me))))
  (cond ((= $$ticks 0) (with (success-test t) (wait)))
        (t (act actor self verb '*push object $$object)
           (subgoal push $$object (- $$ticks 1)))))

;;-----------------------------------------
;;  open-door door
;;-----------------------------------------
(sequential-production open-door (door)
       (precondition (and (can-see (a player ?p) nearby (node $$me))
                          (willing-to-interact ?p)
                          (can-see (a location ?l) location (node $$me))))
       (context-condition (willing-to-interact ?p))
  (subgoal tell-human 'open $$door $$p 'in $$l))

(sequential-production open-door (door)
       (precondition (and (not (can-see (a player) nearby (node $$me)))
                          (know-of-in-ism (a player ?p))
                          (willing-to-interact ?p)
                          (can-see (a location ?l) location (node $$me))))
       (context-condition (willing-to-interact $$p))
  (with (success-test (can-see (node $$l) location (node $$p)))
    (subgoal bring-player $$l))
  (subgoal tell-human 'open $$door $$p 'in $$l)
  (with (success-test (match $$door (a door closed nil)))
    (subgoal fail)))

;;-----------------------------------------
;;  bring-player <location> <tries>
;;-----------------------------------------
(sequential-production bring-player (location tries)
       (precondition (know-of-in-ism (a player ?p)))
  (cond ((= $$tries 0)
         (with (success-test (can-see (node $$location) location
                                      (node $$p)))
           (subgoal fail)))
        (t (subgoal find $$p)
           (subgoal tell-human 'come $$location $$p)
           (subgoal lead $$p $$location)
           (with (success-test (can-see (node $$location)
                                        location (node $$p)))
             (subgoal bring-player $$location (- $$tries 1))))))

(parallel-production lead (player location)
  ;;  this demon allows the cat to wait for the human to catch up
  ;;  while leading him.  We put three of them here because
  ;;  lyotard will only wait three times before giving up.
  (with (priority-modifier .5)
    (demon (not (and (can-see (a location ?l) location (node $$player))
                     (can-see (a location ?l) location (node $$me))))
      (with (success-test
             (and (can-see (a location ?l) location (node $$player))
                  (can-see (a location ?l) location (node $$me))))
        (subgoal wait-a-while (+ (random 3) 2)))))
  (with (priority-modifier .5)
    (demon (not (and (can-see (a location ?l) location (node $$player))
                     (can-see (a location ?l) location (node $$me))))
      (with (success-test
             (and (can-see (a location ?l) location (node $$player))
                  (can-see (a location ?l) location (node $$me))))
        (subgoal wait-a-while (+ (random 3) 2)))))
  (with (priority-modifier .5)
    (demon (not (and (can-see (a location ?l) location (node $$player))
                     (can-see (a location ?l) location (node $$me))))
      (with (success-test
             (and (can-see (a location ?l) location (node $$player))
                  (can-see (a location ?l) location (node $$me))))
        (subgoal wait-a-while (+ (random 3) 2)))))
```

```
  ;; if the cat has waited 3 times and the human still isn't
  ;; following, then give up.
  (with (priority-modifier .25)
    (demon (not (and (can-see (a location ?l) location (node $$player))
                     (can-see (a location ?l) location (node $$me))))
      (subgoal fail)))
  (subgoal goto $$location nil))

;;-----------------------------------------
;; tell-human <what> <object> <who>
;; tell-human <what> <object> <who> <in/on> <place>
;;
;; tell <who> to do <what> to <object> (can convey as long as
;; human is near), or
;; tell <who> to do <what> to <object> and stay <in/on> <place>
;; while telling him.
;;
;;    <what> = 'open | 'come | 'get | 'feed-me | 'pet-me
;;-----------------------------------------
(parallel-production tell-human (what object who in/on place)
       (precondition (can-see (node $$who) nearby (node $$me)))
       (context-condition (can-see (node $$who) nearby (node $$me)))
  (with (priority-modifier .5) effect-only
    (demon (not (or (and (eq $$in/on 'in)
                         (can-see (node $$me) in (node $$place)))
                    (and (eq $$in/on 'on)
                         (can-see (node $$me) on (node $$place)))))
      (with (success-test
             (or (and (eq $$in/on 'in)
                      (can-see (node $$me) in (node $$place)))
                 (and (eq $$in/on 'on)
                      (can-see (node $$me) on (node $$place)))))
        (subgoal goto $$place nil))))
  (subgoal tell-human-for-time $$what $$object $$who (+ (random 3) 3)))

(sequential-production tell-human (what object who)
       (precondition (can-see (node $$who) nearby (node $$me)))
       (context-condition (can-see (node $$who) nearby (node $$me)))
  (subgoal tell-human-for-time $$what $$object $$who (+ (random 3) 3)))

(sequential-production tell-human-for-time (what object who ticks)
       (precondition (> $$ticks 0))
       (context-condition (willing-to-interact $$who))
  (subgoal tell-human-once $$what $$object $$who)
  (cond
    ((= $$ticks 0) (with (success-test t) (wait)))
    (t (subgoal tell-human $$what $$object $$who (- $$ticks 1)))))

;; pet-me
(sequential-production tell-human-once (what me who)
       (precondition
        (and (eq $$what 'pet-me)
             (can-see (an actor ?human) (node $$who) nearby (node $$me))))
  (par-cond
   (t (act actor self verb '*meow))
   (t (act actor self verb '*rub object $$human))
   (t (act actor self verb '*rub object $$human))
   (t (act actor self verb '*rub object $$human))
   (t (act actor self verb '*rub object $$human))
   (t (act actor self verb '*rub object $$human))
   (t (act actor self verb '*lick object $$human))
   (t (act actor self verb '*circle-near object $$human))
   (t (act actor self verb '*nudge object $$human))
   (t (act actor self verb '*lookat object $$human))
   ((can-see (node $$human) on (a furniture-for-people))
```

```
    (act actor self verb '*jump-in object $$human))))


;; feed-me
(sequential-production tell-human-once (what bowl who)
       (precondition (and (eq $$what 'feed-me)
                          (can-see (a food-bowl ?fb) (node $$bowl))))
  (par-cond
   (t (act actor self verb '*meow))
   (t (act actor self verb '*lookat object $$fb))
   (t (act actor self verb '*circle-near object $$fb))))

;; open the door
(sequential-production tell-human-once (what object who)
       (precondition (and (eq $$what 'open)
                          (can-see (a door ?d) (node $$object)
                                   nearby (node $$me))))
  (par-cond
   (t (act actor self verb '*scratch object $$d))
   (t (act actor self verb '*meow))
   (t (act actor self verb '*meow))
   (t (act actor self verb '*meow))
   (t (act actor self verb '*meow))
   (t (act actor self verb '*lookat object $$d))
   (t (act actor self verb '*go-toward object $$d))
   (t (act actor self verb '*circle-near object $$d))))

;; follow me
(sequential-production tell-human-once (what object who)
       (precondition
        (and (eq $$what 'follow)
             (can-see (a location ?l-me) location (node $$me))
             (know-of-in-ism (a location ?l) location (node $$object))
             (match ?path (call map-room-path self ?l-me ?l))
             (not (null ?path))))
  (par-cond
   (t (act actor self verb '*meow))
   (t (act actor self verb '*meow))
   (t (act actor self verb '*meow))
   (t (act actor self verb '*go-toward object (car $$path)))
   (t (act actor self verb '*go-toward object (car $$path)))
   (t (act actor self verb '*go-toward object (car $$path)))
   (t (act actor self verb '*lookbackat object $$who))))

;; get object
(sequential-production tell-human-once (what object who)
       (precondition (and (eq $$what 'get)
                          (can-see (a solid ?s) (node $$object)
                                   nearby (node $$me))))
  (par-cond
   (t (act actor self verb '*meow))
   (t (act actor self verb '*circle-near object $$s))
   (t (act actor self verb '*go-toward object $$s))))

;;-----------------------------------------
;; explore <ticks>
;;-----------------------------------------
(sequential-production explore (ticks)
  (cond ((= $$ticks 0) (with (success-test t) (wait)))
        (t (subgoal go-random)
           (subgoal explore (- $$ticks 1)))))

;;-----------------------------------------
;; go random place from here
;;-----------------------------------------
```

```lisp
;;  The precondition collects all of the places that the cat can
;;  go to from where it is:
;;      other rooms, objects to jump on, containers to go in, and
;;      object already on (the cat will jump off), a container the
;;      cat is alread in (the cat will exit it.).
(sequential-production go-random ()
       (precondition
        (collect-from
         (or (and (know-of-in-ism (a location ?place) goable
                                  (a location) containing (node $$me))
                  ;; something you are afaid of isn't in there is it
                  (match (call feel-fear $em)
                         (list-containing (list-of ?obj ?amount)))
                  (> ?amount $emotional-serious-threshold)
                  (not (can-see (node ?obj) in (node ?place))))
             (and (can-see (a door ?place closed t) connected-to
                           location (node $$me))
                  (call has-feature $em 'energetic))
             ;; places to jump in or on.
             (and (or (can-see (a solid ?place size ?size) in
                               (a container) containing (node $$me))
                      (can-see (a solid ?place size ?size) on
                               (a solid) supporting (node $$me)))
                  (not (match ?place (an actor)))
                  (or (and (> ?size $size) (< ?size (* $size 10)))
                      (and (match ?place (a container opening-size ?opening))
                           (> ?opening $size))))
             ;; go off of or out of.
             (can-see (a solid ?place) supporting (node $$me))
             (and (can-see (a container ?place closed nil) containing (node $$me))
                  (not (match ?place (a location)))))
         (in ?all-places-to-go ?place)))
   (cond
    ((null $$all-places-to-go)
     (subgoal trapped))
    (t (par-cond
        ;; probably backtracking, so maybe look around here a while
        ((= (length $$all-places-to-go) 1) (subgoal go-random-here))
        ((= (length $$all-places-to-go) 1) (subgoal go-random-here))
        ((= (length $$all-places-to-go) 1) (subgoal go-random-here))
        (t (let ((place (random-element $$all-places-to-go)))
             (cond
              ((can-see (node $$place) supporting (node $$me))
               (act actor self verb '*jump-off object $$place))
              ((and (can-see (node $$place) containing (node $$me))
                    (not (match $$place (a location))))
               (act actor self verb '*go-out object $$place))
              ((match $$place (a door closed t))
               (with ignore-failure
                 (subgoal go $$place nil)))
              (t (subgoal go $$place nil)))))))))

(sequential-production go-random-here ()
       (precondition (or (can-see (a container ?place) containing
                                  (node $$me))
                         (can-see (a solid ?place) supporting
                                  (node $$me))))
  (par-cond
   ((match $$place (a container))
    (act actor self verb '*walk-around-in object $$place))
   ((not (match $$place (a container))) ; on a solid instead
    (act actor self verb '*walk-along object $$place))
   (t (act actor self verb '*lookaround object $$place))))

;;------------------------------------------
;;  go <somewhere> <quality> (nil or quickly)
;;------------------------------------------
;;  always goes in a container rather than on it.

;;  walk
;;  there is a small chance of pushing an object that you are around.
(sequential-production go (place how)
       (precondition (and (match $$how nil)
                          (can-see (an anything ?p) (node $$place))))
  (par-cond
   ;; if you are going somewhere, are angry and see an object
   ;; just ripe to push off, maybe try to push it off.
   ((and (can-see (a solid ?obj size ?size) on (a solid)
                  supporting (node $$me))
         (call feel-anger $em)
         (< ?size (* $size 2)))
    (act actor self verb '*push object $$obj))
   ((match $$place (a location))
    (act actor self verb '*go-to object $$place))
   ((or (match $$place (a door closed nil))
        (match $$place (a doorway)))
    (act actor self verb '*go-through object $$place))
   ((and (match $$place (a door closed t))
         (call has-feature $em 'energetic))
    (with (success-test (can-see (a door closed nil) (node $$place)))
      (subgoal open-door $$place))
    (act actor self verb '*go-through object $$place))
   ((and (match $$place (a container closed nil opening-size ?osize))
         (> ?osize $size)
         (not (match $$place (a location))))
    (act actor self verb '*go-in object $$place))
   ((and (match $$place (a solid size ?size))
         (not (match $$place (a container)))
         (> ?size $size)
         (not (> ?size (* $size 10))))
    (act actor self verb '*jump-on object $$place))))

;; run
(sequential-production go (place how)
       (precondition (match $$how 'quickly))
  (par-cond
   ((match $$place (a location))
    (act actor self verb '*run-to object $$place))
   ((or (match $$place (a door closed nil))
        (match $$place (a doorway)))
    (act actor self verb '*run-through object $$place))
   ((and (match $$place (a door closed t))
         (call has-feature $em 'energetic))
    (with (success-test (can-see (a door closed nil) (node $$place)))
      (subgoal open-door $$place))
    (act actor self verb '*run-through object $$place))
   ((and (match $$place (a container closed nil opening-size ?osize))
         (> ?osize $size)
         (not (match $$place (a location))))
    (act actor self verb '*run-in object $$place))
   ((and (match $$place (a solid size ?size))
         (not (match $$place (a container)))
         (> ?size $size)
         (not (> ?size (* $size 10))))
    (act actor self verb '*jump-on object $$place))))

;;-------------------------
;;  goto
;;-------------------------
```

```
(sequential-production goto (target how)
       (precondition
        (and (can-see (a location ?l-me) location (node $$me))
             (know-of-in-ism (a location ?l-target) location
                             (node $$target))))
  (with (success-test (or (can-see (a location) containing (node $$me))
                          (can-see (node $$l-target) location
                                   (node $$me))))
    (subgoal goto-location $$l-me $$how))
  (with (success-test (can-see (node $$l-target) location (node $$me)))
    (subgoal goto-location $$l-target $$how))
  (with (success-test (or (can-see (node $$target) containing
                                   (node $$me))
                          (can-see (node $$target) supporting
                                   (node $$me))))
    (subgoal goto-place-in-location $$target $$how)))

;;  in a location, goto another location
(sequential-production goto-location (location how)
       (precondition
        (and (can-see (a location ?l-me) containing (node $$me))
             (match $$location (a location))))
       (context-condition (can-see (a location) containing (node $$me)))
  (cond
   ((match $$l-me $$location) (with (success-test t) (wait)))
   (t (let ((path (call map-room-path self $$l-me $$location)))
        (par-cond
         ((null $$path) (subgoal fail))
         ($$path (subgoal go (car $$path) $$how)
                 (subgoal goto $$location $$how)))))))

;;  not directly in a location, goto my location (jump off and out
;;  of things)
(sequential-production goto-location (location how)
       (precondition
        (and (can-see (a location ?l-me) location (node $$me))
             (not (can-see (node ?l-me) containing (node $$me)))
             (match $$location (a location))
             (or (can-see (a solid ?place) supporting (node $$me))
                 (can-see (a container ?place) containing (node $$me)))))
       (context-condition (can-see (node $$l-me) location (node $$me)))
  (cond
    ((match $$place (a container))
     (par-cond
      ((match $$how nil)
       (act actor self verb '*go-out object $$place))
      ((match $$how 'quickly)
       (act actor self verb '*run-out object $$place))))
    ((match $$place (a solid))
     (act actor self verb '*jump-off object $$place)))
  (with (success-test (can-see (node $$location) containing (node $$me)))
    (subgoal goto-location $$location $$how)))

;;  already in the correct location.  now going on or in something.
(sequential-production goto-place-in-location (target how)
       (precondition ;  in the same location, & bind place w/in loc.
        (and (can-see (a location ?l) location (node $$me))
             (can-see (a location ?l) location (node $$target))
             (or (can-see (a solid ?place) supporting (node $$me))
                 (can-see (a container ?place) containing (node $$me)))))
       (context-condition
        (and (can-see (node $$l) location (node $$me))
             (can-see (node $$l) location (node $$target))))
  (cond
   ((match $$place $$target) (with (success-test t) (wait)))
    (t (let ((path (call map-inroom-path self $$place $$target)))
         (par-cond
          ((null $$path) (subgoal fail))
          ($$path (subgoal go (car $$path) $$how)
                  (subgoal goto-place-in-location $$target $$how)))))))

;;-------------------------
;;  drop object
;;-------------------------

(sequential-production drop (object)
       (precondition (can-see (an any-object ?loc) supporting
                              (node $$me)))
       (context-condition (can-see (node $$object) in (node $$me)))
  (act actor self
       verb '*put
       object $$object
       preposition 'on
       iobject $$loc))

(sequential-production drop (object)
       (precondition (can-see (an any-object ?loc) containing
                              (node $$me)))
       (context-condition (can-see (node $$object) in (node $$me)))
  (act actor self
       verb '*put
       object $$object
       preposition 'in
       iobject $$loc))

;;-------------------------
;;  say-to
;;-------------------------

(sequential-production say-to (receiver words)
       (precondition (and $$receiver
                          (can-see (node $$receiver) nearby (node $$me))))
  (act actor self
       verb '*say
       object (make speech text $$words voice
                    (slot voice $actor))
       preposition 'to
       iobject $$receiver))

(sequential-production say-to (receiver words)
       (precondition (match $$receiver nil))
  (act actor self
       verb '*say
       object (make speech text $$words voice
                    (slot voice $actor))))

)) ;  end of production memory

(active-plan-tree
 (start-goals
  (persistent
   (with (priority 7)
     (demon stop-purring-demon
            (and $purring
                 (or (not (or (call has-feature $em 'content)
                              (call feel-happy $em)))
                     (call feel-fear $em)
                     (call feel-anger $em)))
       (mental-act (setf $purring nil))))
   (with (priority 6)
```

```
  (demon respond-negatively-to-contact
         (and (or (can-see (node $$me) in (a player ?player))
                  (and (did-see (an act actor (a player ?player)
                                    verb ?verb
                                    object $$me))
                       (member ?verb
                               '(*pet *take *rub *lick *push *pounce-on
                                 *nudge *polish  *touch *taste)
                               :test #'eq)))
              (match (call feel-anger $em)
                     (list-containing (list-of ?player ?amount))))
    (subgoal attack-once $$player)
    (subgoal escape-run-away $$player $$amount)))
(with (priority 5)
  (demon eat-if-hungry
         (and (< $food $hunger-threshold)
              (match nil ?bowl-empty-or-full-and-cant-reach))
      (with (importance 20)
            (success-test (> $food $satisfied-threshold))
        (subgoal want-to-eat))))
(with (priority 4)
  (demon respond-to-aggressive
         (and (call has-feature $em 'energetic)
              (feel-aggressive ?a ?v)
              (> ?v $emotional-serious-threshold)
              (can-see (node ?a) nearby (node $$me)))
    (with (success-test (or (not (can-see (node $$a) nearby (node $$me)))
                            (not (and (call has-feature $em 'energetic)
                                      (feel-aggressive $$a ?v)
                                      (> ?v $emotional-serious-threshold)))))
      (subgoal threaten $$a $$v))))
(with (priority 4)
  (demon run-away-demon
         (and (can-see (an any-object ?a) nearby (node $$me))
              (match (call feel-fear $em)
                     (list-containing (list-of ?a ?v))))
     (with (success-test (not (did-see (node $$a) nearby
                                       (node $$me) recently)))
       (subgoal escape-run-away $$a $$v))))
(with (priority 3)
  (demon start-purring-demon
         (and (not $purring) (or (call has-feature $em 'content)
                                 (call feel-happy $em)))
    (mental-act (setf $purring t))))
(with (priority 2)
  (demon respond-to-being-held-demon
         (and (can-see (node $$me) in (a player ?player)))
    (cond
     ((call has-feature $em 'ignoring)
      (act actor self verb '*jump-down))
     ((> (call relationship $em $$player 'like) 0)
      ;; succeed this goal for Em, to make the cat happy.
      (with (success-test t) (importance 3)
            (subgoal being-held-and-like))
      (par-cond
       ((not $purring) (mental-act (setf $purring t)))
       (t (act actor self verb '*lookat object $$player adverb 'lazily))
       (t (act actor self verb '*close-eyes adverb 'lazily))))
     (t
      (par-cond
       (t (act actor self verb '*lookat object $$player))
       (t (act actor self verb '*wait))
       (t (act actor self verb '*lookaround)))))))
(with (priority 2)
  (demon respond-to-contact-demon
         (and (did-see (an act actor (a player ?player)
                                 verb ?verb
                                 object $$me))
              (member ?verb
                      '(*pet *rub *lick *push *nudge *polish
                        *touch *taste)
                      :test #'eq))
     (cond
      ((call has-feature $em 'ignoring)
       (subgoal go-random))
      ((and (member $$verb '(*pet *rub))
            (> (call relationship $em $$player 'like)))
       ;; succeed this goal for Em to make the cat happy.
       (with (success-test t) (importance 3)
             (subgoal being-petted-and-like))
       (with (success-test $purring)
         (mental-act (setf $purring t)))
       (par-cond
        (t (act actor self verb '*lookat object $$player adverb 'lazily))
        (t (act actor self verb '*close-eyes adverb 'lazily))))
      (t (act actor self verb '*lookat object $$player adverb 'strangely)
         (subgoal go-random)))))
(with (priority 1)
  (demon release-frustration-demon
         (and (match (call feel-anger $em)
                     (list-containing (list-of ?a ?v1)))
              (match (call feel-fear $em)
                     (list-containing (list-of ?a ?v2)))
              (match (max ?v1 ?v2) ?v)
              (> ?v $emotional-annoyance-threshold))
    (subgoal release-frustration $$v)))
(with (priority 1)
  (demon watch-moving-objects-demon
         (and (can-see (a solid ?obj moving ?amount-moving)
                       nearby (node $$me))
              ?amount-moving)
    (subgoal watch-moving-object $$obj (+ (random 4) 3))))
(with (priority 1)
  (demon listen-for-sounds-demon
         ;; this would need to be more complicated if there
         ;; were other cats who could meow and hiss.
         (and (did-hear (a sound ?sound))
              (not (match ?sound (a meow)))
              (not (match ?sound (a hiss))))
    (par-cond listen-to-sound
      (t (act actor self verb '*listen))
      ((hear-speech ?maker ?a ?b ?c ?sound)
       (subgoal watch $$maker (+ (random 4) 3))))))
(with (priority 0) (subgoal amuse-self))
)
(one-shot
 (with (priority 10) (subgoal set-up-global-constants-and-variables)))))

) ; end of slots

(methods

 ;; use up food as time goes by.
 (tick ()
   (setf $food (- $food 1)))

 ;; this maps a path from room to room
 (map-room-path (start end)
   (let ((start (cond ((ism-node-p start) start)
                      ((sr-node-p start) (slot ism-node start))
```

```
                        (t nil)))
           (end  (cond ((ism-node-p end) end)
                       ((sr-node-p end) (slot ism-node end))
                       (t nil))))
       (cond
         ((and start end)
          (best-path (list #'(lambda (node)
                                   (call net-node-nodes-related-by node 'goable))
                               #'(lambda (edge) edge))
                     start end nil nil nil))
         (t nil))))

   ;;  this maps a path from object to object within the room.
   ;;  not finished.  need to redo map.
   (map-inroom-path (start end)
     (let ((start (cond ((ism-node-p start) start)
                        ((sr-node-p start) (slot ism-node start))
                        (t nil)))
           (end  (cond ((ism-node-p end) end)
                       ((sr-node-p end) (slot ism-node end))
                       (t nil))))
       (cond
         ((and start end)
          (best-path (list #'(lambda (node)
                                   (append
                                    (call net-node-nodes-related-by node 'down-in)
                                    (call net-node-nodes-related-by node 'down-on)
                                    (call net-node-nodes-related-by node 'up-in)
                                    (call net-node-nodes-related-by node 'up-on)))
                               #'(lambda (edge) edge))
                     start end nil nil nil))
         (t nil))))

   ) ;  end of methods
  ) ;  end class
```