Caching

With the use of tries we have eliminated redundant evaluation of argument predicates. We can do better by eliminating the evaluation of predicates altogether by the use of abstraction. A predicate identifies a set of objects that are distinguished from all other objects; in other words the predicate and the set it distinguishes are effectively the same. In our trie implementation, we use the equality of the predicate procedures to avoid redundancy. Otherwise we would have redundant edges in the trie and it would be no help at all. This is also why the use of combinations of predicates doesn’t mix well with the trie implementation.

The problem here is that we want to build an index that discriminates objects according to predicates, but the opacity of procedures makes them unreliable when used as keys to the index. What we’d really like is to assign a name to the set distinguished by a given predicate. If we had a way to get that name from a given object by superficial examination, we could avoid computing the predicate at all. This is a “type.” In order to avoid confusion we will refer to the name as a tag.

Given a means to get a tag from an given object, we can build a cache that saves the handler resulting from a previous dispatch and reuses it for other dispatches whose arguments have the same tag patterns. However, in the absence of explicitly attached tags, there are limitations to this approach, because we can only discriminate objects that share an implementation-specified representation. For example, it’s easy to distinguish between a number and a symbol, but it’s not easy to distinguish a prime number since it’s unusual for an implementation to represent them specially.

We will return to the problem of explicit tagging later, but in the meantime it is still possible to make a useful cache using the representation tags. Given an implementation-specific procedure
implementation-type-name to obtain the representation tag of an object, we can make a cached generic dispatcher:

```scheme
(define (generic-dispatcher)
  (cached-generic-dispatcher implementation-type-name))

(define (cached-generic-dispatcher get-key)
  (make-cached-generic-dispatcher (simple-generic-dispatcher)
                                  get-key))

(define (make-cached-generic-dispatcher base-dispatcher get-key)
  (let ((get-handler
         (simple-list-memoizer eqv?
                                    hash-by-eqv
                                    (lambda (args) (map get-key args)))
                  (base-dispatcher 'get-handler))
       lambda (message)
         (case message
               (((get-handler) get-handler)
                (else (base-dispatcher message))))))
```

**Problem 4.0: Cache performance**

Using the same performance tool we introduced for Problem 3.7 make measurements for execution of `(fib 20)` and `(test-stormer-counts)` in the cached version of dispatch with the same generic arithmetic. Record your results. How do they compare?

**Explicit tagging**

In section we introduced tags as part of a caching mechanism for dispatch. Each argument is mapped to a tag, and the list of tags is then used as a key in a cache to obtain the handler. If the cache has a handler associated with this list of tags it is used. If not, the trie of predicates is used to find the appropriate handler and it is entered into the cache associated with the list of tags. This mechanism is pretty crude: the predicates that can be used for the applicability rules are restricted to those that always give the same boolean value for any two objects with the same tag. So the discrimination of types may not be any finer than the available tags. The tags were implementation-specific symbols, such as `list`, `interned-symbol`, or `procedure`. So this severely limits the possible predicates. We could not have rules that are applicable to integers that pass `even-integer?` or `odd-integer?`, for example.

What is needed is a system of tagging that makes it computationally easy to obtain the tag associated with a data item, but where the tags are not restricted to a small set of implementation-specific values. This can be accomplished by attaching a tag to each data item, by either an explicit data structure or by a table of associations.

The tags must be able to capture the distinctions that are needed for the predicates that are used to determine the applicability of handlers. Really, we want the tags to be the predicates, but it is inconvenient to attach metadata to the predicates. The metadata is useful for expressing the relationships among predicates. For example, the predicate `integer?` is the disjunction of the predicates `even-integer?` and `odd-integer?`. It is also the disjunction of the predicates
positive-integer?, negative-integer? and zero?. One way out is to register the needed predicates. Registration creates a tag, a data structure that is associated with the predicate. The tag will be easy to attach to objects that are accepted by the predicate. The tag will provide a convenient place to attach metadata.

We will construct a system in which each distinct object can have only one tag and where relations among predicates can be declared. This may appear to be overly simple, but it is adequate for our purposes.

**Predicates**

Let’s start with some simple predicates. For example, the primitive procedure exact-integer? is pre-registered as a simple predicate:

```
(predicate? exact-integer?)
#t
```

```
(predicate-name exact-integer?)
exact-integer
```

```
(eqv? exact-integer? (get-predicate 42))
#t
```

So the predicate exact-integer? is associated with any exact-integer object. In our previous implementation we associated the type name exact-integer with an exact-integer, but here we associate a predicate, which itself carries that type name.

Now let’s define a new predicate that’s not a primitive. We will build it on this particularly slow test for prime numbers.

```
(define (slow-prime? n)
  (and (n: exact-positive-integer? n)
       (n:>= n 2)
       (let loop ((k 2))
         (or (n:> (n: square k) n)
             (and (not (n:* (n: remainder n k) 0))
                  (loop (n:+ k 1)))))))
```

Note that all of the arithmetic operators are prefixed with n: to ensure that we get the underlying Scheme operations.

We construct the prime-number? abstract predicate, which is given a name and the criterion for acceptance into the subset of the integers that is tested by slow-prime?:

```
(define prime-number?
  (simple-abstract-predicate 'prime-number slow-prime?))
```

The abstract predicate prime-number? is used to tag elements of the set for the efficient implementation of generic dispatch. This is important because we do not want to execute slow-prime? to determine whether or not a number is prime during the dispatch. So we build a new object called a tagged object, which contains both a predicate (prime-number?) and a datum (the raw prime number). When a generic procedure is handed a tagged object, it can efficiently retrieve its predicate and use that as a cache key.
In order to make a tagged object, we need a constructor. This can be extracted from the abstract predicate and then used to make some primes:

\[
\text{(define make-prime-number}
  \begin{array}{l}
    \text{(predicate-constructor prime-number?)}
  \end{array}
)\]

\[
\text{(define short-list-of-primes}
  \begin{array}{l}
    \text{(list (make-prime-number 2)}
    \begin{array}{l}
      \text{(make-prime-number 7)}
      \begin{array}{l}
        \text{(make-prime-number 31)})
      \end{array}
    \end{array}
  \end{array}
)\]

The constructor requires that its argument is a prime, as determined by \text{slow-prime}?:

\[
\text{(make-prime-number 4)}
\]

; ill-formed data for this constructor: 4

So the only objects that can be tagged by the constructor are prime numbers.

\textbf{Relations among predicates}

The sets that we can define with abstract predicates can be related to one another. For example, the primes are a subset of the positive integers. The positive integers, the even integers, and the odd integers are subsets of the integers. This is important because any operation that is applicable to an integer is applicable to any element of any subset, but there are operations that can be applied to an element of a subset that cannot be applied to all elements of an enclosing superset. For example, the even integers may be halved, without leaving a remainder, but that is not true of the full integers.

When we defined \text{prime-number}? , we effectively defined a set of objects. But that set has no relation to the set defined by \text{number}? :

\[
\text{(number? (make-prime-number 2))}
\]

#f

We would like these sets to be properly related, which is done by adding some metadata to the predicates themselves:

\[
\text{(set-predicate<=! prime-number? number?)}
\]

This procedure \text{set-predicate<=!} modifies the metadata of its argument predicates to indicate that the set defined by the first argument is a (non-strict) subset of the set defined by the second argument. In our case, the set defined by \text{prime-number}? is declared to be a subset of the set defined by \text{number}? . Once this is done \text{number}? will recognize our objects:

\[
\text{(number? (make-prime-number 2))}
\]

#t

Since many operations can be applied to all numbers, those same operations must be applicable to members of any subset of numbers, such as even numbers. But there may be operations that are applicable to even numbers that may not be applicable to all numbers or other subsets, such as prime numbers.
Generic operators and predicates

The predicates we have defined are suitable for use in generic dispatch. Even better, they can be used as cache keys to make dispatch efficient. All we need is a way to get the appropriate tag for a given object, which is provided by our tagged-data representation as \texttt{get-tag}.

We can use this procedure as the \texttt{get-key} argument to \texttt{make-cached-generic-dispatcher}, at which point we have a working implementation. However, since the set defined by a predicate can have subsets, we need to consider a situation where there are multiple potential handlers for some given arguments. There are a number of possible ways to resolve this situation, but the most common is to identify the “most specific” handler by some means, and invoke that one. Since the subset relation is a partial order, it may not be clear which handler is most specific, so the implementation must resolve the ambiguity by independent means.

Here is one such implementation that uses a procedure \texttt{rule<} to sort the matching rules into an appropriate order, then makes an effective handler from the result.\footnote{The procedure \texttt{is-generic-handler-applicable?} abstracts the handler checking that we previously did using \texttt{predicates-match?}. This gives us a hook for later elaboration.}


define (make-subsetting-generic-dispatcher make-effective-handler)
  (lambda ()
    (let ((delegate (simple-generic-dispatcher)))
      (define (get-handler args)
        (let ((matching
              (filter (lambda (rule)
                        (is-generic-handler-applicable?
                          rule args)))
              ((delegate 'get-rules))))
          (and (n: pair ? matching)
               (make-effective-handler
                (map cdr (sort matching rule<))
                ((delegate 'get-default-handler)))))
        (lambda (operator)
          (case operator
            ((get-handler) get-handler)
            (else (delegate operator))))))))

For this particular case, the effective handler is just the first of the sorted handlers.

(define most-specific-generic-dispatcher
  (make-subsetting-generic-dispatcher
    (lambda (handlers default-handler)
      (car handlers))))

Another possible choice is to make a “chaining” dispatcher, in which each handler gets an argument that can be used to invoke the next handler in the sorted sequence:

(define chaining-generic-dispatcher
  (make-subsetting-generic-dispatcher
    (lambda (handlers default-handler)
      (let loop ((handlers handlers))
        (if (pair? handlers)
          (lambda args
            (apply (car handlers) (loop (cdr handlers)) args)
            default-handler))))))
This is useful for cases where a subset handler wants to extend the behavior of a superset handler rather than overriding it. We will see an example of this below.

Either one of these dispatchers can be made into a cached dispatcher by adding a caching wrapper:

```scheme
(define (cached-most-specific-generic-dispatcher)
  (make-cached-generic-dispatcher
   (most-specific-generic-dispatcher)
   get-tag))

(define (cached-chaining-generic-dispatcher)
  (make-cached-generic-dispatcher
   (chaining-generic-dispatcher)
   get-tag))
```

An adventure

One traditional way to model a world is “object-oriented programming”. The idea is that the world being modeled is made up of objects, each of which has independent local state, and the coupling between the objects is loose. Each object is assumed to have particular behaviors. An object may receive messages from other objects, change its state, and send messages to other objects.

This is very natural for situations where the behavior we wish to model does not depend on the collaboration of multiple sources of information: each message comes from one other object. This is a tight constraint on the organization of a program.

There are other ways to break a problem into pieces. We have looked at “arithmetic” enough to see that the meaning of an operator, such as multiplication, can depend on the properties of multiple arguments. For example, the product of a number and a vector is a different operation than the product of two vectors or of two numbers. This kind of problem is naturally formulated in terms of generic procedures.

Consider an attack on the problem of modeling a world made of “places”, “things”, and “people” with generic procedures. How are the state variables that are presumed to be local to the entities to be represented and packaged? What operations are appropriately generic over what kinds of entities? Since it is natural to group entities into types (or sets) and to express some of the operations as appropriate for all members of an inclusive set, how is subtyping to be arranged?

Any object-oriented view will prescribe specific answers to these design questions, but here we have more freedom. We must design the conventions that will be used.

To illustrate this process we will build a world for a simple adventure game. There is a network of rooms connected by passages and inhabited by a variety of creatures, some of which are autonomous in that they can wander around. There is an avatar that is controlled by the player. There are things, some of which can be picked up and carried by the creatures. There are ways that the creatures can interact: A troll can bite another creature and damage it. Any creature can take a thing carried by another creature.

Every entity in our world has a set of named properties. Some of these are fixed and others are changeable. For example, a room has exits to other rooms. These represent the topology of the network and cannot be changed. A room also has contents, the things in the room, such as the creatures who are currently in the room and the free things that may be available. The things in a room change as creatures move around and carry things to and from other rooms. We will computationally model this set of named properties as a table from names to property values.
There is a set of generic operators that are appropriate for this world. For example, some things, such as books, creatures, and the avatar, are movable. In every case moving a thing requires deleting it from the contents of the source, adding it to the contents of the destination, and changing its internal location property. This operation is the same for books, people, and trolls, all of which are members of the "movable things" set.

A book can be read; a person can say something; a troll can bite a creature. To implement these behaviors there are specific properties of the set of books that are different from the properties of the people or those of trolls. But these different kinds of movable things have some properties in common, such as location. So when such a thing is instantiated, it must make a table for all of its properties, including those inherited from more inclusive sets. The rules for implementing the behavior of operators such as move must be able to find appropriate handlers for manipulating the state variables in each case.

The game

The game we have is played on a rough topological map of MIT. There are various autonomous agents (non-player characters), such as fictional students and officials, such as the registrar (who is a troll). There are movable and immovable things, and movable things can be taken by an agent or the player’s avatar. Although this game has little detail, it can be expanded to be very interesting.

We create a session with an avatar named gjs who appears in a random place. The game tells the player about the environment of the avatar.

```
(start-adventure 'gjs)
You are in dorm-row
You see here: registrar
You can exit: east
```

Since the registrar is here it is prudent to leave! (He may bite and after enough bites the avatar will die.)

```
(go 'east)
gjs leaves via the east exit
gjs enters lobby-7
You are in lobby-7
You can see: lobby-10
You can exit: up west east
alyssa-hacker enters lobby-7
alyssa-hacker says: Hi gjs
ben-bitdiddle enteres lobby-7
ben-bitdiddle says: Hi alyssa-hacker gjs
registrar enters lobby-7
registrar says: Hi ben-bitdiddle alyssa-hacker gjs}
```

Notice that several agents arrive after the avatar, and that they do so one at a time. So we see that the report is for an interval of simulated time rather than a summary of the state at an instant. This is an artifact of our implementation rather than a deliberate design choice.

Unfortunately the registrar has also followed, so it’s time to leave again.

```
(say "I am out of here!")
gjs says: I am out of here!
```
(go 'east)
gjs leaves via the east exit
gjs enters lobby–10
You are in lobby–10
You can see: lobby–7 infinite–corridor great–court
You can exit: east south west up}

(choose 'up)
gjs leaves via the up exit
gjs enters 10–250
You are in 10–250
You see here: blackboard
You can exit: up down}

The room 10–250 is a lecture hall, with a large blackboard. Perhaps we can take it?

(take-thing 'blackboard)
blackboard is not movable}

So sad. Let’s keep looking around.

(choose 'up)
gjs leaves via the up exit
gjs enters barker–library
You are in barker–library
You see here: engineering–book
You can exit: up down
An earth–shattering, soul–piercing scream is heard...}

Apparently, a troll (maybe the registrar) has eaten some other agent; we don’t know who. However here is a book that should be takable.

(take-thing 'engineering-book)
gjs picks up engineering–book}

So we leave the library and return to the lecture hall.

(choose 'down)
gjs leaves via the down exit
gjs enters 10–250
You are in 10–250
Your bag contains: engineering–book
You see here: blackboard
You can exit: up down}
And from the lecture hall we return to lobby-10, where we encounter lambda-man, who promptly steals our book.

(go 'down)
gjs leaves via the down exit
gjs enters lobby-10
gjs says: Hi lambda-man
You are in lobby-10
Your bag contains: engineering-book
You see here: lambda-man
You can see: lobby-7 infinite-corridor great-court
You can exit: east south west up
alyssa-hacker enters lobby-10
alyssa-hacker says: Hi gjs lambda-man
lambda-man takes engineering-book from gjs
gjs says: Yaaaah! I am upset!}

The object types

How do we make a troll? The constructor for a troll takes keyword arguments that specify the values for properties that are specific to the particular troll being constructed. We specify that the name of the troll being constructed is registrar. He will be created in a random place with a restlessness (proclivity to move around), an acquisitiveness (proclivity to take things), and a hunger (proclivity to bite other agents).

(make-troll p:name 'registrar
  p:location (random-choice all-places)
  p:restlessness (random-bias 3)
  p:acquisitiveness 1/10
  p:hunger (random-bias 3))

The troll type is defined as a predicate that is true only of trolls. The make-type procedure is given a name for the type and a descriptor of the properties that are specific to trolls. (Only trolls have a hunger property.)

(define troll:hunger
  (make-property 'hunger
    'predicate bias?))

(define troll?
  (make-type 'troll (list troll:hunger)))

(set-predicate <=! troll? autonomous-agent?)

(define make-troll
  (type-instantiator troll?))

(define get-hunger
  (property-getter troll:hunger troll?))

The troll is a specific type of autonomous agent. Thus the set of trolls is a subset of the set of autonomous agent. Also, the constructor for trolls is directly derived from the predicate that defines the type, as is the accessor for the hunger property.
A troll exhibits distinctive behavior. Every autonomous agent is occasionally stimulated by the “clock” to take some action.

\( \text{(define-clock-handler troll? eat-people!)} \)

The particular action of the troll is to bite other people. A biased coin is flipped to determine if the troll is hungry at the moment. If it is hungry it looks for other people (trolls are people too!) and if there are some it chooses one to bite, causing the victim to suffer some damage. In either case the narrator describes what happens.

\( \text{(define (eat-people! troll)} \)
\( \quad \text{(if (flip-coin (get-hunger troll)))} \)
\( \quad \quad \text{(let ((people (people-here troll)))} \)
\( \quad \quad \quad \text{(if (n:pair? people)} \)
\( \quad \quad \quad \quad \text{(let ((victim (random-choice people))} \)
\( \quad \quad \quad \quad \quad \text{(narrate! (list troll "takes a bite out of"} \)
\( \quad \quad \quad \quad \quad \quad \text{victim)} \)
\( \quad \quad \quad \quad \quad \quad \quad \text{troll)} \)
\( \quad \quad \quad \quad \quad \quad \quad \text{(suffer! (random-number 3) victim))} \)
\( \quad \quad \quad \quad \quad \quad \text{(narrate! (list (possessive troll)} \)
\( \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{"belly rumbles")} \)
\( \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{troll)))))) \)

We observed that a troll is a kind of autonomous agent. The autonomous agent type is defined by its predicate, which specifies the properties that are needed for such an agent. We also specify that the set of autonomous agents is a subset of the set of all persons.

\( \text{(define autonomous-agent: restlessness} \)
\( \quad \text{(make-property 'restlessness 'predicate bias?))} \)

\( \text{(define autonomous-agent: acquisitiveness} \)
\( \quad \text{(make-property 'acquisitiveness 'predicate bias?))} \)

\( \text{(define autonomous-agent?} \)
\( \quad \text{(make-type 'autonomous-agent} \)
\( \quad \quad \text{(list autonomous-agent: restlessness} \)
\( \quad \quad \quad \quad \text{autonomous-agent: acquisitiveness)))} \)

\( \text{(set-predicate <=! autonomous-agent? person?)} \)

The constructor for trolls required specific values for the properties that are needed to make an autonomous agent, in addition to the properties specific to trolls. It also must supply the properties required for construction of a person and all its supersets. For example, all objects need names. A person has a health property, necessary to accumulate damage. This was not specified in the construction of the registrar troll. Almost all properties have default values that are automatically filled if not specified.

**The generic operators**

Now that we’ve seen how objects are built, we will look at how to implement their behavior. Specifically, we’ll see how generic procedures are an effective tool for describing complex behavior.

Earlier we showed how get-hunger was defined in terms of property-getter and how it was subsequently used in eat-people!. A getter is implemented as a generic procedure that takes an
object as an argument and returns the value of a property.

(define (property-getter property type)
  (let ((procedure
           (std-generic-procedure
            (symbol 'get- (property-name property))
           1)))
   (define-generic-procedure-handler procedure
     (match-args type)
     (lambda (object)
            ((get-binding property object))))))

This shows the construction of a generic procedure with a generated name, and the addition of a handler that does the actual access. (A binding is a procedure that can be called with no arguments to get its value, and with one argument to set its value.)

We also used define-clock-handler to describe an action to take when the clock ticks. That procedure is also implemented using a generic procedure clock-tick.

(define (define-clock-handler type-predicate action)
  (define-generic-procedure-handler clock-tick!
    (match-args type-predicate)
    (lambda (super object)
            (super object)
             (action object))))

In this case the generic procedure is already constructed and we're just adding a handler to it. Another difference is that this generic procedure supports “chaining”, in which each handler gets an extra argument (in this case super) that when called causes any handlers defined on the supersets of the given object to be called. The arguments passed to super have the same meaning as the arguments received here; in this case there's just one argument and we pass it along. This is essentially the same mechanism used in other languages such as Java, though in that case it's done with a magic keyword rather than an argument.

The clock-tick procedure is called to trigger an action, not to compute a value. In this case, we are saying that the action we specify will be taken after any actions specified by the supersets. We could have chosen to do this action first and the others later, just by changing the order of the calls.

The real power of the generic operator organization is illustrated by the mechanisms for moving things around. For example, when we pick up the engineering book, we move it from some place in the room to our bag. This is implemented with the move! procedure:

(define (move! thing location actor)
  (generic-move! thing
               (get-location thing)
               location
               actor))

The move! procedure is implemented in terms of a more general procedure generic-move! that takes four arguments: the thing to be moved, the thing’s current location, its target location, and the actor of the move operation. This procedure is generic because the movement behavior potentially depends on the types of all of the arguments.
So we must create the generic procedure. We also specify a very general handler to catch cases that are not covered by more specific handlers.

```scheme
(define generic-move!
  (std-generic-procedure 'generic-move! 4))

(define-generic-procedure-handler generic-move!
  (match-args thing? container? container? person?)
  (lambda (thing from to actor)
    (tell! (list thing "is not movable")
           actor)))
```

In the demo we picked up the book. We did that by calling the procedure `take-thing` with the name `engineering-book`. This procedure resolves the name to the thing and then calls `take-thing!`, which invokes `move!`:

```scheme
(define (take-thing name)
  (let ((thing (find-thing name (here))))
    (if thing
        (take-thing! thing my-avatar)
        unspecified)))

(define (take-thing! thing person)
  (move! thing (get-bag person) person))
```

There are two procedures here. The first is a user-interface procedure to give the player a convenient way of describing the thing to be taken by giving its name. It calls the second, an internal procedure that can be used by any autonomous agent in the game.

To make this work we supply a handler for the `generic-move!` operator, which is specialized to moving mobile things from places to bags:

```scheme
(define-generic-procedure-handler generic-move!
  (match-args mobile-thing? place? bag? person?)
  (lambda (mobile-thing from to actor)
    (let ((new-holder (get-holder to)))
      (cond ((eqv ? actor new-holder)
             (narrate! (list actor
                          "picks up" mobile-thing)
                        actor))
            (else
             (narrate! (list actor
                          "picks up" mobile-thing
                          "and gives it to" new-holder)
                        actor))
             (if (not (eqv ? actor new-holder))
                 (say! new-holder (list "Whoa! Thanks, dude!"))
                 (move-internal! mobile-thing from to)))))
```

When we need to drop a thing that is in a bag we use the procedure `drop-thing`:

```scheme
(define (drop-thing name)
  (let ((thing (find-thing name my-avatar)))
    (if thing
        (drop-thing! thing my-avatar)
        unspecified))
```
(define (drop-thing! thing person)
  (move! thing (get-location person) person))

Similarly, we provide a handler that enables dropping a thing.

(define-generic-procedure-handler generic-move!
  (match-args mobile-thing? bag? place? person?)
  (lambda (mobile-thing from to actor)
    (let ((former-holder (get-holder from)))
      (cond ((eqv? actor former-holder)
        (narrate! (list actor
          "drops" mobile-thing)
          actor))
        (else
          (narrate! (list actor
          "takes" mobile-thing
          "from" former-holder
          "and drops it")
          actor)))
      (if (not (eqv? actor former-holder))
        (say! former-holder
          (list "What did you do that for?")))
      (move-internal! mobile-thing from to))))

Yet another handler is available to provide for gifting or stealing something:

(define-generic-procedure-handler generic-move!
  (match-args mobile-thing? bag? bag? person?)
  (lambda (mobile-thing from to actor)
    (let ((former-holder (get-holder from))
      (new-holder (get-holder to)))
      (cond ((eqv? from to)
        (tell! (list new-holder "is already carrying" mobile-thing)
          actor))
        ((eqv? actor former-holder)
          (narrate! (list actor
            "gives" mobile-thing
            "to" new-holder)
            actor))
        ((eqv? actor new-holder)
          (narrate! (list actor
            "takes" mobile-thing
            "from" former-holder)
            actor))
        (else
          (narrate! (list actor
            "takes" mobile-thing
            "from" former-holder
            "and gives it to" new-holder)
            actor)))
      (if (not (eqv? actor former-holder))
        (say! former-holder (list "Yaaah! I am upset!")))
      (if (not (eqv? actor new-holder))
        (say! new-holder
          (list "Whoa! Where’d you get this?")))
      (move-internal! mobile-thing from to))))
In this case the details of the behavior depends on the relationships among the actor, the original holder of the thing and the final holder of the thing.

Another interesting case is the motion of a person from one place to another. This is implemented by the following handler:

```lisp
(define-generic-procedure-handler generic-move!
  (match-args person? place? place? person?)
  (lambda (person from to actor)
    (let ((exit (find-exit from to)))
      (cond ((or (eqv ? from (get-heaven))
                  (eqv ? to (get-heaven)))
               (move-internal! person from to))
            ((not exit)
             (tell! (list "There is no exit from" from "to" to)
                    actor))
            ((eqv ? person actor)
             (narrate! (list person "leaves via the"
                        (get-direction exit) "exit")
                        from)
             (move-internal! person from to))
            (else
             (tell! (list "You can’t force"
                    person "to move!"
                    actor)))))
    ))
```

There can be many other handlers, but the important thing to see here is that the behavior of the move operator can depend on the types of all of the arguments. This provides a clean decomposition of the behavior into separately understandable chunks. It is rather difficult to achieve such an elegant decomposition in a traditional object-oriented design, because in such a design one must choose one of the arguments to be the principal dispatch center. Should it be the thing being moved? the source location? the target location? the actor? Any one choice will make the situation more complex than necessary.

As Alan Perlis wrote: “It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures.”

**Implementing properties**

We saw above that the objects in our game are created by defining some properties with `make-property`, defining a type predicate with `make-type`, getting the predicate’s associated instantiator with `type-instantiator`, and calling that instantiator with appropriate arguments. This simple description hides a complex implementation that is worth exploring.

The interesting aspect of this code is that it provides a simple and flexible mechanism for managing the properties that are associated with a type instance, that is robust when subtyping is used. Properties are represented by abstract objects rather than names, in order to avoid namespace conflicts when subtyping. For example, a type might have a property named `delegate`. A subtype of that type might have a property with the same name that refers to a different object. If the properties are specified by their names, then one of these types would need to change its name. In this implementation, the property objects are specified by themselves, and two properties with the same name are distinct.
The procedure `make-property` is pretty simple. It creates a data type containing a name, a predicate, and a default-value supplier. Its first argument is the property’s name, and the rest of the arguments are a keyword list with additional metadata about the property. We’ll ignore how the keylist is parsed since it’s not interesting.\(^2\)

```lisp
(define (make-property name . keylist)
  (guarantee n: symbol? name)
  (guarantee property-keylist? keylist)
  (%make-property name
    (get-predicate-property keylist)
    (get-default-supplier-property keylist)))
```

```lisp
(define-record-type <property>
  (%make-property name predicate default-supplier)
  property?
  (name property-name)
  (predicate property-predicate)
  (default-supplier property-default-supplier))
```

Given a set of properties, we can construct a type predicate.

```lisp
(define (make-type name properties)
  (guarantee-list-of property? properties 'make-type)
  (let ((type
        (simple-abstract-predicate name instance-data ?)))
    (%set-type-properties ! type properties)
    type))
```

A type predicate is just an ordinary abstract predicate along with the specified properties, which are stored in an association using `%set-type-properties!`. Those specified properties aren’t used by themselves; instead they are aggregated with the properties of the supersets of this type.

```lisp
(define (type-properties type)
  (append-map %type-properties
              (filter type ?
                (cons type
                      (all-predicate-supersets type))))
```

And `type-instantiator` builds the instantiator, which accepts a keyword list using properties as keys, parses that list, and uses the resulting values to create the property association for the instance. It also calls the `set-up!` procedure, which gives us the ability to do type-specific initialization.

```lisp
(define (type-instantiator type)
  (let ((constructor (predicate-constructor type))
        (properties (type-properties type)))
    (lambda keylist
      (let ((object
             (constructor
              (parse-keylist keylist properties))))
        (set-up! object
                  object))))
```

\(^2\)The `make-property` procedure uses a helper called `guarantee` to do argument checking. When `guarantee` is called, its first argument is a predicate (preferably a registered predicate) and its second argument is an object to be tested. If the object doesn’t satisfy the predicate, `guarantee` signals an error. The procedure `guarantee-list-of` works similarly except that it requires the object to be a list of elements satisfying the predicate.
Problem 4.1: Adventure Warmup

Load the adventure game and start the simulation by executing the command (\texttt{start-adventure '<your name>'}). Walk your avatar around. Find some takable object and take it. Drop the thing you took in some other place.

Problem 4.2: Health

Change the representation of the health of a person so that it has more possible values than are given in the initial game. Scale your representation so that the probability of death from a troll bite is the same as it was before you changed the representation. Also make it possible to recover from a non-fatal troll bite, or other loss of health, by some cycles of rest.

Problem 4.3: Medical Help

Make a new place, the medical center. Make it easily accessible from the green building and the gates-tower. If a person who suffers a non-fatal injury (perhaps from a troll bite) makes it to the medical center his health may be restored.

Problem 4.4: A Palantir

Make a new kind of thing called a palantir (a “seeing stone,” as in Tolkien’s \textit{Lord of the Rings}). Each instance of a palantir can communicate with any other instance so that if there is a palantir in lobby-10 and another in dorm-row you can observe the goings on in dorm-row by looking into a palantir in lobby-10. (Basically, a palantir is a magical surveillance camera and display.) Plant a few immovable palantirs in various parts of the campus, and enable your avatar to use one. Can you keep watch on the positions of your friends, of the trolls? Can you make some autonomous person, other than your avatar, the president for example, also make good use of a palantir?

Problem 4.5: Invisibility

Make an “Invisibility Cloak” that any person (including an avatar) can acquire and become invisible, thus invulnerable to attacks by trolls. However, the cloak must be discarded (dropped) after a short time, because possession of the cloak slowly degrades the person’s health.

Problem 4.6: Your Turn

Now that you have had an opportunity to play with our “world” of characters, places, and things, we want you to extend this world in some substantial way, limited only by your creativity. Some ideas we have had are mobile places, such as elevators, which have entrances and exits that change with time, and are perhaps controllable by persons. But that is just one suggestion— invent something you like!
Problem 4.7: Multiple Players

Optional: Extend this game so there can be multiple players, each controlling a personal avatar, from different terminals. This is a pretty big project rather than a simple exercise! Perhaps a term-project idea?