18 Semantic Networks, Inheritance, and CLOS

Chapter Objectives
We build Semantic Networks in Lisp:
   Supported by property lists
First implementation (early 1970s) of object systems
Object systems in Lisp include:
   Encapsulation
   Inheritance
   Hierarchical
   Polymorphism
The Common Lisp Object System (CLOS)
Encapsulation
   Inheritance
   Inheritance search programmer designed
Example CLOS implementation
   Further implementations in exercises
18.1 Introduction
18.2 Object-Oriented Programming Using CLOS
18.3 CLOS Example: A Thermostat Simulation

18.1 Semantic Networks and Inheritance in Lisp
This chapter introduces the implementation of semantic networks and inheritance, and a full object-oriented programming system in Lisp. As a family of representations, semantic networks provide a basis for a large variety of inferences, and are widely used in natural language processing and cognitive modeling. We do not discuss all of these, but focus on a basic approach to constructing network representations using property lists. After these are discussed and used to define a simple semantic network, we define a function for class inheritance. Finally, since semantic networks and inheritance are important precursors of object-oriented design, we present CLOS, the Common Lisp Object System, Section 18.2, and an example implementation in 18.3.

A Simple Semantic Network
Lisp is a convenient language for representing any graph structure, including semantic nets. Lists provide the ability to create objects of arbitrary complexity and these objects may be bound to names, allowing for easy reference and the definition of relationships between them. Indeed, all Lisp data structures are based on an internal implementation as chains of pointers, a natural isomorph to graph structures.

For example, labeled graphs may be represented using association lists: each node is an entry in an association list with all the arcs out of that node stored in the datum of the node as a second association list. Arcs are described by an
association list entry that has the arc name as its key and that has the arc
destination as its datum. Using this representation, the built-in association list
functions are used to find the destination of a particular arc from a given node.
For example, the labeled, directed graph of Figure 18.1 is represented by the
association list:

\[
((a \ (1 \ . \ b)) \\
(b \ (2 \ . \ c)) \\
(c \ (2 \ . \ b) \ (3 \ . \ a)))
\]

![Figure 18.1 A simple labeled directed graph](image)

This approach is the basis of many network implementations. Another way to
implement semantic networks is through the use of property lists.

Essentially, property lists are a built-in feature of Lisp that allows named
relationships to be attached to symbols. Rather than using \texttt{setq} to bind an
association list to a symbol, with property lists we can program the direct
attachment of named attributes to objects in the global environment. These
are bound to the symbol not as a value but as an additional component called
the property list.

Functions for managing property lists are \texttt{get}, \texttt{setf}, \texttt{remprop}, and
\texttt{symbol-plist.get}, which has the syntax:

\[
\text{(get <symbol> <property-name>)}
\]

may be used to retrieve a property from \texttt{<symbol>} by its \texttt{<property-name>}.
For example, if the symbol \texttt{rose} has a \texttt{color} property of \texttt{red} and
a \texttt{smell} property of \texttt{sweet}, then \texttt{get} would have the behavior:

\[
\text{(get 'rose 'color)}
\text{red} \\
\text{(get 'rose 'smell)}
\text{sweet} \\
\text{(get 'rose 'party-affiliation)}
\text{nil}
\]

As the last of these calls to \texttt{get} illustrates, if an attempt is made to retrieve a
nonexistent property, one that is not on the property list, \texttt{get} returns a value
of \texttt{nil}.

Properties are attached to objects using the \texttt{setf} function, which has the
syntax:

\[
\text{(setf <form> <value>)}
\]
setf is a generalization of setq. The first argument to setf is taken from a large but specific list of forms. setf does not use the value of the form but the location where the value is stored. The list of forms includes car and cdr. setf places the value of its second argument in that location. For example, we may use setf along with the list functions to modify lists in the global environment, as the following transcript shows:

```
? (setq x '(a b c d e))
(a b c d e)
? (setf (nth 2 x) 3)
3
? x
(a b 3 d e)
```

We use setf, along with get, to change the value of properties. For instance, we may define the properties of a rose by:

```
> (setf (get 'rose 'color) 'red)
red
> (setf (get 'rose 'smell) 'sweet)
sweet
```

remprop takes as arguments a symbol and a property name and causes a named property to be deleted. For example:

```
> (get 'rose 'color)
red
> (remprop 'rose 'color)
color
> (get 'rose 'color)
nil
```

symbol-plist takes as argument a symbol and returns its property list. For example:

```
> (setf (get 'rose 'color) 'red)
red
> (setf (get 'rose 'smell) 'sweet)
sweet
> (symbol-plist 'rose)
(smell sweet color red)
```

Using property lists, it is straightforward to implement a semantic network. For example, the following calls to setf implement the semantic network description of species of birds from Figure 2.1. The isa relations define inheritance links.

```
(setf (get 'animal 'covering) 'skin)
(setf (get 'bird 'covering) 'feathers)
(setf (get 'bird 'travel) 'flies)
(setf (get 'bird 'isa) animal)
```
Using this representation of semantic nets, we now define control functions for hierarchical inheritance. This is simply a search along isa links until a parent is found with the desired property. The parents are searched in a depth-first fashion, and search stops when an instance of the property is found. This is typical of the inheritance algorithms provided by many commercial systems. Variations on this approach include the use of breadth-first search as an inheritance search strategy.

**inherit-get** is a variation of **get** that first tries to retrieve a property from a symbol; if this fails, **inherit-get** calls **get-from-parents** to implement the search. **get-from-parents** takes as its first argument either a single parent or a list of parents; the second argument is a property name. If the parameter **parents** is nil, the search halts with failure. If **parents** is an atom, it calls **inherit-get** on the parent to either retrieve the property from the parent itself or continue the search. If **parents** is a list, **get-from-parents** calls itself recursively on the car and cdr of the list of **parents**. The tree walk based function **inherit-get** is defined by:

```lisp
(defun inherit-get (object property)
    (or (get object property)
        (get-from-parents (get object 'isa) property))))

(defun get-from-parents (parents property)
    (cond ((null parents) nil)
          ((atom parents)
           (inherit-get parents property))
          (t (or (get-from-parents (car parents) property)
                 (get-from-parents (cdr parents) property)))))
```
In the next section we generalize our representations for things, classes, and inheritance using the CLOS object-oriented programming library.

### 18.2 Object-Oriented Programming Using CLOS

In spite of the many advantages of functional programming, some problems are best conceptualized in terms of objects that have a state that changes over time. Simulation programs are typical of this. Imagine trying to build a program that will predict the ability of a steam heating system to heat a large building: we can simplify the problem by thinking of it as a system of objects (rooms, thermostats, boilers, steam pipes, etc.) that interact to change the temperature and behavior of each other over time. Object-oriented languages support an approach to problem solving that lets us decompose a problem into interacting objects. These objects have a state that can change over time, and a set of functions or methods that define the object’s behaviors. Essentially, object-oriented programming lets us solve problems by constructing a model of the problem domain as we understand it. This model-based approach to problem solving is a natural fit for artificial intelligence, an effective programming methodology in its own right, and a powerful tool for thinking about complex problem domains.

There are a number of languages that support object-oriented programming. Some of the most important are Smalltalk, C++, Java and the Common Lisp Object System (CLOS). At first glance, Lisp, with its roots in functional programming, and object orientation, with its emphasis on creating objects that retain their state over time, may seem worlds apart. However, many features of the language, such as dynamic type checking and the ability to create and destroy objects dynamically, make it an ideal foundation for constructing an object-oriented language. Indeed, Lisp was the basis for many of the early object-oriented languages, such as Smalltalk, Flavors, KEE, and ART. As the Common Lisp standard was developed, the Lisp community has accepted CLOS as the preferred way to do object-oriented programming in Lisp.

In order to fully support the needs of object-oriented programming, a programming language must provide three capabilities: 1) **encapsulation**, 2) **polymorphism**, and 3) **inheritance**. The remainder of this introduction describes these capabilities and an introduction to the way in which CLOS supports them.

**Encapsulation.** All modern programming languages allow us to create complex data structures that combine atomic data items into a single entity. Object-oriented encapsulation is unique in that it combines both data items and the procedures used for their manipulation into a single structure, called a **class**. For example, the abstract data types seen previously (e.g., Section 16.2) may quite properly be seen as classes. In some object-oriented languages, such as Smalltalk, the encapsulation of procedures (or methods as they are called in the object-oriented community) in the object definition is explicit. CLOS takes a different approach, using Lisp’s type-checking to provide this same ability. CLOS implements methods as
generic functions. These functions check the type of their parameters to
guarantee that they can only be applied to instances of a certain
class. This gives us a logical binding of methods to their objects.

**Polymorphism.** The word polymorphic comes from the roots
“poly”, meaning *many*, and “morphos”, meaning *form*. A function is
polymorphic if it has many different behaviors, depending on the
types of its arguments. Perhaps the most intuitive example of
polymorphic functions and their importance is a simple drawing
program. Assume that we define objects for each of the shapes
(square, circle, line) that we would like to draw. A natural way to
implement this is to define a method named draw for each object
class. Although each individual method has a different definition,
depending on the shape it is to draw, all of them have the same
name. Every shape in our system has a draw behavior. This is much
simpler and more natural than to define a differently named
function (draw-square, draw-circle, etc.) for every shape. CLOS
supports polymorphism through generic functions. A generic
function is one whose behavior is determined by the types of its
arguments. In our drawing example, CLOS enables us to define a
generic function, draw, that includes code for drawing each of the
shapes defined in the program. On evaluation, it checks the type of
its argument and automatically executes the appropriate code.

**Inheritance.** Inheritance is a mechanism for supporting class
abstraction in a programming language. It lets us define general
classes that specify the structure and behavior of their
specializations, just as the class “tree” defines the essential attributes
of pine trees, poplars, oaks, and other different species. In Section
18.1, we built an inheritance algorithm for semantic networks; this
demonstrated the ease of implementing inheritance using Lisp’s
built-in data structuring techniques. CLOS provides us with a more
robust, expressive, built-in inheritance algorithm.

The basic data structure in CLOS is the **class**. A **class** is a
specification for a set of object instances. We define classes using the
**defclass** macro. **defclass** has the syntax:

```
(defclass <class-name> (<superclass-name>*))
```

*<class-name>* is a symbol. Following the class name is a list of direct
superclasses (called **superclass**); these are the class’s immediate
parents in the inheritance hierarchy. This list may be empty. Following
the list of parent classes is a list of zero or more **slot-specifiers**. A
**slot-specifier** is either the name of a **slot** or a list consisting of a
**slot-name** and zero or more **slot-options**:

```
slot-specifier ::= slotname |
(slot-name [slot-option])
```

For instance, we may define a new class, **rectangle**, which has slots
values for **length** and **width**:
> (defclass rectangle()
  (length width))
#<standard-class rectangle>
make-instance allows us to create instances of a class, taking as its
argument a class name and returning an instance of that class. It is the
instances of a class that actually store data values. We may bind a symbol,
rect, to an instance of rectangle using make-instance and
setq:

> (setq rect (make-instance 'rectangle))
#<rectangle #x286AC1>

The slot options in a defclass define optional properties of slots. Slot
options have the syntax (where “|” indicates alternative options):

slot-option ::= :reader <reader-function-name>|
                :writer <writer-function-name>|
                :accessor <reader-function-name>|
                :allocation <allocation-type>|
                :initarg <initarg-name>|
                :initform <form>

We declare slot options using keyword arguments. Keyword arguments are
a form of optional parameter in a Lisp function. The keyword, which
always begins with a “:”, precedes the value for that argument. Available
slot options include those that provide accessor s to a slot. The :
reader option defines a function called reader-function-name
that returns the value of a slot for an instance. The :writer option
defines a function named writer-function-name that will write to
the slot. :accessor defines a function that may read a slot value or may
be used with setf to change its value.

In the following transcript, we define rectangle to have slots for
length and width, with slot accessors get-length and get-
width, respectively. After binding rect to an instance of rectangle
using make-instance, we use the accessor, get-length, with
setf to bind the length slot to a value of 10. Finally, we use the
accessor to read this value.

> (defclass rectangle ()
  ((length :accessor get-length)
   (width :accessor get-width)))
#<standard-class rectangle>
> (setq rect (make-instance 'rectangle))
#<rectangle #x289159>
> (setf (get-length rect) 10)
10
> (get-length rect)
10
In addition to defining **accessors**, we can access a slot using the primitive function **slot-value**. **slot-value** is defined for all slots; it takes as arguments an instance and a slot name and returns the value of that slot. We can use it with **setf** to change the slot value. For example, we could use **slot-value** to access the **width** slot of **rect**:

```lisp
> (setf (slot-value rect 'width) 5)
5
> (slot-value rect 'width)
5
```

**allocation** lets us specify the memory allocation for a slot. **allocation-type** may be either **:instance** or **:class**. If allocation type is **:instance**, then CLOS allocates a local slot for each instance of the type. If allocation type is **:class**, then all instances share a single location for this slot. In **:class** allocation, all instances will share the same value of the slot; changes made to the slot by any instance will affect all other instances. If we omit the **allocation** specifier, allocation defaults to **:instance**.

**initarg** allows us to specify an argument that we can use with **make-instance** to specify an initial value for a slot. For example, we can modify our definition of **rectangle** to allow us to initialize the **length** and **width** slots of instances:

```lisp
> (defclass rectangle ()
  ((length :accessor get-length :initarg init-length)
   (width :accessor get-width :initarg init-width)))
#<standard-class rectangle>
> (setq rect (make-instance 'rectangle
  'init-length 100 'init-width 50))
#<rectangle #x28D081>
> (get-length rect)
100
> (get-width rect)
50
```

**initform** lets us specify a form that CLOS evaluates on each call to **make-instance** to compute an initial value of the slot. For example, if we would like our program to ask the user for the values of each new instance of rectangle, we may define a function to do so and include it in an **initform**:

```lisp
> (defun read-value (query) (print query)(read))
read-value
> (defclass rectangle ()
  ((length :accessor get-length
    :initform (read-value "enter length")))
```
(width :accessor get-width
   :initform (read-value "enter width")))

<standard-class rectangle>
> (setq rect (make-instance ‘rectangle))

“enter length” 100
“enter width” 50

<rectangle #x290461>
> (get-length rect)
100
> (get-width rect)
50

A generic function is a function whose behavior depends upon the type of
its arguments. In CLOS, generic functions contain a set of methods, indexed
by the type of their arguments. We call generic functions with a syntax
similar to that of regular functions; the generic function retrieves and
executes the method associated with the type of its parameters.

CLOS uses the structure of the class hierarchy in selecting a method in a
generic function; if there is no method defined directly for an argument of
a given class, it uses the method associated with the “closest” ancestor in
the hierarchy. Generic functions provide most of the advantages of
“purer” approaches of methods and message passing, including inheritance
and overloading. However, they are much closer in spirit to the functional
programming paradigm that forms the basis of Lisp. For instance, we can
use generic functions with mapcar, funcall, and other higher-order
constructs in the Lisp language.

We define generic functions using either defgeneric or defmethod.
defgeneric lets us define a generic function and several methods using
one form. defmethod enables us to define each method separately,
although CLOS combines all of them into a single generic function.
defgeneric has the (simplified) syntax:

(defun defgeneric f-name lambda-list <method-description>*)

<method-description> ::= (:method specialized-lambda-
   list form)

defgeneric takes a name of the function, a lambda list of its
arguments, and a series of zero or more method descriptions. In a method
description, specialized-lambda-list is just like an ordinary
lambda list in a function definition, except that a formal parameter may
be replaced with a (symbol parameter-specializer) pair: symbol is the name
of the parameter, and parameter-specializer is the class of the argument. If
an argument in a method has no parameter specializer, its type defaults to
t, which is the most general class in a CLOS hierarchy. Parameters of type
t can bind to any object. The specialized lambda list of each method
specifier must have the same number of arguments as the lambda list in
the defgeneric. A defgeneric creates a generic function with the
specified methods, replacing any existing generic functions.
As an example of a generic function, we may define classes for `rectangle` and `circle` and implement the appropriate methods for finding areas:

```lisp
(defclass rectangle ()
  ((length :accessor get-length :initarg init-length)
   (width :accessor get-width :initarg init-width)))

(defclass circle ()
  ((radius :accessor get-radius :initarg init-radius)))

(defun area (shape)
  (:method ((shape rectangle))
    (* (get-length shape) (get-width shape)))
  (:method ((shape circle))
    (* (get-radius shape) (get-radius shape) pi)))
```

```lisp
(setq rect (make-instance 'rectangle 'init-length 10 'init-width 5))
(setq circ (make-instance 'circle 'init-radius 7))
```

We can use the `area` function to compute the area of either shape:

```lisp
> (area rect)
50
> (area circ)
153.93804002589985
```

We can also define methods using `defmethod`. Syntactically, `defmethod` is similar to `defun`, except it uses a specialized `lambda` list to declare the class to which its arguments belong. When we define a method using `defmethod`, if there is no generic function with that name, `defmethod` creates one; if a generic function of that name already exists, `defmethod` adds a new method to it. For example, suppose we wish to add the class `square` to the above definitions, we can do this with:

```lisp
(defclass square ()
  ((side :accessor get-side :initarg init-side)))

(defmethod area ((shape square))
  (* (get-side shape) (get-side shape)))
```

```lisp
(setq sqr (make-instance 'square 'init-side 6))
```

`defmethod` does not change the previous definitions of the `area` function; it simply adds a new method to the generic function:

```lisp
> (area sqr)
36
```
Inheritance in CLOS

CLOS is a multiple-inheritance language. Along with offering the program designer a very flexible representational scheme, multiple inheritance introduces the potential for creating anomalies when inheriting slots and methods. If two or more ancestors have defined the same method, it is crucial to know which method any instance of those ancestors will inherit. CLOS resolves potential ambiguities by defining a class precedence list, which is a total ordering of all classes within a class hierarchy.

Each defclass lists the direct parents of a class in left-to-right order. Using the order of direct parents for each class, CLOS computes a partial ordering of all the ancestors in the inheritance hierarchy. From this partial ordering, it derives the total ordering of the class precedence list through a topological sort. The precedence list follows two rules:

1. Any direct parent class precedes any more distant ancestor.
2. In the list of immediate parents of defclass, each class precedes those to its right.

CLOS computes the class precedence list for an object by topologically sorting its ancestor classes according to the following algorithm. Let C be the class for which we are defining the precedence list:

1. Let $S_C$ be the set of $C$ and all its superclasses.
2. For each class, $c$, in $S_C$, define the set of ordered pairs:
   $$R_c = \{(c, c_1), (c_1, c_2), \ldots (c_{n-1}, c_n)\}$$
   where $c_1$ through $c_n$ are the direct parents of $c$ in the order they are listed in defclass. Note that each $R_c$ defines a total order.
3. Let $R$ be the union of the $R_c$s for all elements of $S_C$. $R$ may or may not define a partial ordering. If it does not define a partial ordering, then the hierarchy is inconsistent and the algorithm will detect this.
4. Topologically sort the elements of $R$ by:
   a. Begin with an empty precedence list, $P$.
   b. Find a class in $R$ having no predecessors. Add it to the end of $P$ and remove the class from $S_C$ and all pairs containing it from $R$. If there are several classes in $S_C$ with no predecessor, select the one that has a direct subclass nearest the end in the current version of $P$.
   c. Repeat the two previous steps until no element can be found that has no predecessor in $R$. 

In the example:

- `(area rect)`
- `50`
- `(area circ)`
- `153.93804002589985`
d. If $S_e$ is not empty, then the hierarchy is inconsistent; it may contain ambiguities that cannot be resolved using this technique.

Because the resulting precedence list is a total ordering, it resolves any ambiguous orderings that may have existed in the class hierarchy. CLOS uses the class precedence list in the inheritance of slots and the selection of methods.

In selecting a method to apply to a given call of a generic function, CLOS first selects all applicable methods. A method is applicable to a generic function call if each parameter specializer in the method is consistent with the corresponding argument in the generic function call. A parameter specializer is consistent with an argument if the specializer either matches the class of the argument or the class of one of its ancestors.

CLOS then sorts all applicable methods using the precedence lists of the arguments. CLOS determines which of two methods should come first in this ordering by comparing their parameter specializers in a left-to-right fashion. If the first pair of corresponding parameter specializers are equal, CLOS compares the second, continuing in this fashion until it finds corresponding parameter specializers that are different. Of these two, it designates as more specific the method whose parameter specializer appears leftmost in the precedence list of the corresponding argument. After ordering all applicable methods, the default method selection applies the most specific method to the arguments. For more details, see Steele (1990).

### 18.3 CLOS Example: A Thermostat Simulation

The properties of object-oriented programming that make it a natural way to organize large and complex software implementations are equally applicable in the design of knowledge bases. In addition to the benefits of class inheritance for representing taxonomic knowledge, the message-passing aspect of object-oriented systems simplifies the representation of interacting components.

As a simple example, consider the task of modeling the behavior of a steam heater for a small office building. We may naturally view this problem in terms of interacting components. For example:

- Each office has a thermostat that turns the heat in that office on and off; this functions independently of the thermostats in other offices.
- The boiler for the heating plant turns itself on and off in response to the heat demands made by the offices.
- When the demand on the boiler increases, there may be a time lag while more steam is generated.
- Different offices place different demands on the system; for example, corner offices with large windows lose heat faster than inner offices. Inner offices may even gain heat from their neighbors.
The amount of steam that the system may route to a single office is affected by the total demand on the system. These points are only a few of those that must be taken into account in modeling the behavior of such a system; the possible interactions are extremely complex. An object-oriented representation allows the programmer to focus on describing one class of objects at a time. We would represent thermostats, for example, by the temperature at which they call for heat, along with the speed with which they respond to changes in temperature.

The steam plant could be characterized in terms of the maximum amount of heat it can produce, the amount of fuel used as a function of heat produced, the amount of time it takes to respond to increased heat demand, and the rate at which it consumes water.

A room could be described in terms of its volume, the heat loss through its walls and windows, the heat gain from neighboring rooms, and the rate at which the radiator adds heat to the room.

The knowledge base is built up of classes such as room and thermostat, which define the properties of the class, and instances such as room-322 and thermostat-211, which model individual situations.

The interactions between components are described by messages between instances. For example, a change in room temperature would cause a message to be sent to an instance of the class thermostat. If this new temperature is low enough, the thermostat would switch after an appropriate delay. This would cause a message to be sent to the heater requesting more heat. This would cause the heater to consume more oil, or, if already operating at maximum capacity, to route some heat away from other rooms to respond to the new demand. This would cause other thermostats to turn on, and so forth.

Using this simulation, we can test the ability of the system to respond to external changes in temperature, measure the effect of heat loss, or determine whether the projected heating is adequate. We could use this simulation in a diagnostic program to verify that a hypothesized fault could indeed produce a particular set of symptoms. For example, if we have reason to believe that a heating problem is caused by a blocked steam pipe, we could introduce such a fault into the simulation and see whether it produces the observed symptoms.

The significant thing about this example is the way in which an object-oriented approach allows knowledge engineers to deal with the complexity of the simulation. It enables them to build the model a piece at a time, focusing only on the behaviors of simple classes of objects. The full complexity of the system behavior emerges when we execute the model.

The basis of our CLOS implementation of this model is a set of object definitions. Thermostats have a single slot called setting. The setting of each instance is initialized to 65 using initform. heater-thermostat is a subclass of thermostat for controlling heaters (as opposed to air conditioners); they have a single slot that will be bound to an instance of the heater class. Note that the heater slot has a class allocation; this captures the constraint that the thermostats in different rooms of a building control the single building's heater-obj.
A heater has a state (on or off) that is initialized to off, and a location. It also has a slot, rooms-heated, that will be bound to a list of objects of type room. Note that instances, like any other structure in Lisp, may be elements of a list.

(room has slots for temperature, initialized to 65 degrees; thermostat, which will be bound to an instance of thermostat; and name, the name of room.)

These class definitions define the hierarchy of Figure 18.2.
We represent our particular simulation as a set of instances of these classes. We will implement a simple system of one room, one heater, and one thermostat:

(setf office-heater (make-instance 'heater 'loc 'office))

(setf room-325 (make-instance 'room
  'therm (make-instance 'heater-thermostat
        'heater-obj office-heater)
        'name 'room-325))

(setf (slot-value office-heater 'rooms-heated) (list room-325))

Figure 18.3 shows the definition of instances, the allocation of slots, and the bindings of slots to values.

We define the behavior of rooms through the methods change-temp, check-temp, and change-setting. change-temp sets the temperature of a room to a new value, prints a message to the user, and calls check-temp to determine whether the heater should come on. Similarly, change-setting changes the thermostat setting, therm-setting, and calls check-temp, which simulates the thermostat. If the temperature of the room is less than the thermostat setting, it sends the heater a message to turn on; otherwise it sends an off message.
(defmethod change-temp ((place room) temp-change)
  (let ((new-temp (+ (room-temp place) temp-change)))
    (setf (room-temp place) new-temp)
    (terpri)
    (prinl "the temperature in")
    (prinl (room-name place))
    (prinl " is now ")
    (prinl new-temp)
    (terpri)
    (check-temp place)))

(defmethod change-setting ((room room) new-setting)
  (let ((therm (room-thermostat room)))
    (setf (therm-setting therm) new-setting)
    (prinl "changing setting of thermostat in")
    (prinl (room-name room))
    (prinl " to ")
    (prinl new-setting)
    (terpri)
    (check-temp room)))

(defmethod check-temp ((room room))
  (let* ((therm (room-thermostat room))
         (heater (slot-value therm 'heater)))
    (cond ((> (therm-setting therm) (room-temp room))
           (send-heater heater 'on))
          (t (send-heater heater 'off)))))

The heater methods control the state of the heater and change the temperature of the rooms. send-heater takes as arguments an instance of heater and a message, new-state. If new-state is on it calls the turn-on method to start the heater; if new-state is off it shuts the heater down. After turning the heater on, send-heater calls heat-rooms to increase the temperature of each room by one degree.

(defmethod send-heater ((heater heater) new-state)
  (case new-state
    (on (if (equal (heater-state heater) 'off)
          (turn-on heater))
        (heat-rooms (slot-value heater
                     'rooms-heated) 1))
(off (if (equal (heater-state heater) 'on)
   (turn-off heater))))
(defmethod turn-on ((heater heater))
  (setf (heater-state heater) 'on)
  (prinl “turning on heater in”)
  (prinl (slot-value heater ‘location))
  (terpri))
(defmethod turn-off ((heater heater))
  (setf (heater-state heater) 'off)
  (prinl “turning off heater in”)
  (prinl (slot-value heater ‘location))
  (terpri))
(defun heat-rooms (rooms amount)
  (cond ((null rooms) nil)
    (t (change-temp (car rooms) amount)
      (heat-rooms (cdr rooms) amount))))

The following transcript illustrates the behavior of the simulation.

> (change-temp room-325 5)
“the temperature in “room-325” is now “60
“turning on heater in “office
“the temperature in “room-325” is now “61
“the temperature in “room-325” is now “62
“the temperature in “room-325” is now “63
“the temperature in “room-325” is now “64
“the temperature in “room-325” is now “65
“turning off heater in “office
nil
> (change-setting room-325 70)
“changing setting of thermostat in “room-325” to ”70
“turning on heater in “office
“the temperature in “room-325” is now “66
“the temperature in “room-325” is now “67
“the temperature in “room-325” is now “68
“the temperature in “room-325” is now “69
“the temperature in “room-325” is now “70
“turning off heater in “office
nil
Exercises

1. Create two semantic network representations (Section 17.1) for an application of your choice. Build the representation first using association lists and then build it using property lists. Comment on the differences in these two approaches for representing semantic information.

2. Add to the CLOS simulation of Section 18.3 a cooling system so that if any room’s temperature gets above a certain temperature it starts to cool. Also add a “thermal” factor to each room so that it heats and cools as a function of its volume and insulation value.

3. Create a CLOS simulation in another domain, e.g., a building that has both heating and cooling. You can add specifics to each room such as an insulation value that mitigates heat/cooling loss. See the discussion at the beginning of Section 18.3 for parameters you might build in to your augmented system.

4. Create a CLOS simulation for an ecological situation. For example, you might have classes for grass, wolves, cattle, and weather. Then make a set of rules that balances their ecological survival across time.