18 - Semantic Networks, Inheritance, and CLOS

Objetivos: Nós construiremos redes semânticas emL isp:
- Suportadas por *property lists*
- A primeira implementação (decada de 70) do object system em Lisp incluem:
  - Hierarchical
  - Inheritance
  - Polymorphism
- O Common Lisp Object System (CLOS)
- Encapsulation
  - Inheritance
    - Programador de pesquisa de herança projetada
- Exemplo de implementação CLOS
  - Outras implementações em exercícios

Conteúdo: 18.1 Introdução
18.2 Object-Oriented Programming usando CLOS
18.3 Um exemplo CLOS: Uma simulação de um termóstato

18.1 Semantic Networks and Inheritance in Lisp

Este capítulo apresenta a implementação de semantic networks e
inheritance, em um ambiente completamente orientado objetos em Lisp. Como uma família das representações, semantic networks proporcionar uma base para uma grande variedade de inferências, e são amplamente utilizados em natural language processing e cognitive modeling. Não discutimos todos esses, mas se concentrar em uma abordagem de base para a construção de representações de rede usando *property lists*.
Após estes são discutidos e utilizado para definir uma simples semantic network, nós definimos uma função para *class inheritance*. Finalmente, uma vez que a semantic networks e a
inheritance são precursores importantes do design orientado a objetos, apresentamos CLOS, o Common Lisp Object System, sessão 18.2, e um examplo de sua implementação em implementation 18.3.

A Simple Semantic Network:
Lispé uma linguagem conveniente para representar qualquer estrutura gráfica, incluindo redes semânticas.Listas de fornecer a capacidade de criar objetos de complexidade arbitrária e esses objetos podem ser ligados a nomes, permitindo a fácil referência e a definição das relações entre elas. De fato, todas as estruturas de dados Lisp baseiam-se numa implementação interna como cadeias de ponteiros, aum isomorfo natural para representar graficamente estruturas.

Por exemplo, gráficos rotulados pode ser representado usando listas de associação: cada nó é uma entrada em uma lista de associação com todos os arcos fora desse nó armazenado no dado do nó como uma segunda lista associação. Arcos são descritos por um associação entrada de lista que tem o nome de arco como sua chave e que tem o arco destino como dado. Utilizando essa representação, a lista associação incorporada a função são utilizadas para encontrar o destino de um arco em particular a partir de um dado nó. Por exemplo, a, grafo orientado rotulado da Figura 18.1 é representado pela lista associação:

```
((a (1 . b))
        (b (2 . c))
        (c (2 . b) (3 . a)))
```

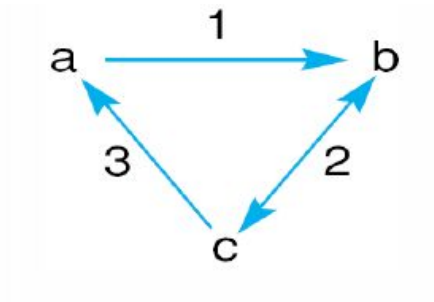

Figure 18.1 A simple labeled directed graph

Esta aproximação é a base de muitas implementações da rede. Outra forma de implementar redes semânticas é através do uso de *property lists.*
Essencialmente, *property lists*são um recurso incorporados ao Lisp que as permite relações nomeadas para ser anexado a símbolos. Ao invés de usar *setq* para ligar um lista de associação a um símbolo, com *property lists* podemos programar o direto fixação de atributos nomeados a objetos no ambiente global. Estes estão ligados ao símbolo não como um valor, mas como um componente adicional chamada de *property list.*
Funções para gerenciar *plists* são ***get, setf, remprop,*** e ***symbol-plist. get***, cuja sintaxe é

(get <symbol> <property-name>)

pode ser usado para recuperar a partir de uma propriedade a partir de <symbol> por sua <property-name>. Por exemplo, se o simbolo ***rose*** tem uma ***color*** de propriedade ***red*** e ***smell*** é uma propriedade de ***sweet***, então ***get*** teria o comportamento:

```
(get ‘rose ‘color)
red
(get ‘rose ‘smell)
sweet
```

(get ‘rose ‘party-affiliation)
nil

Como a última dessas chamadas para ***get*** ilustra, se for feita uma tentativa para recuperar uma propriedade inexistente, uma que não é na plist, ***get*** retorna o valor ***nil.***
Propriedades estão ligados a objetos usando a função ***setf***, que tem a sintaxe:

(setf)

setf é uma generalização de ***setq***. O primeiro argumento para ***setf*** é feita a partir de uma grande lista de formas mas específica. ***setf*** não utiliza o ***value***da forma, mas o local onde o ***value*** é guardado.A lista das formas inclui **car** e **cdr**. ***setf*** coloca o valor de seu segundo argumento nesse local. Por exemplo, podemos usar ***setf*** por muito tempo com as funções de lista para modificar as listas no ambiente global, como mostrado a seguir transcrito:

> (setq x ‘(a b c d e))
(a b c d e)
> (setf (nth 2 x) 3)
3
> x
(a b 3 d e)

Nós ***setf*** junto de ***get***, para alterar o valor de propriedades.Por exemplo, podemos definir as propriedades de uma ***rose*** por:

> (get ‘rose ‘color)
red
> (remprop ‘rose ‘color)
color
> (get ‘rose ‘color)
nil

symbol-plist tem como argumento um símbolo e retorna uma *plist*. Por exemplo:

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

Usando *plists*, é simples de implementar uma rede semântica. Por exemplo, o seguinte chamadas para ***setf*** implementa a rede semântica descrição de espécies de aves da Figura 2.1. As relações ***isa*** definem as ligações de herança.

(setf (get ‘animal ‘covering) ‘skin)

```
(setf (get ‘bird ‘covering) ‘feathers)
(setf (get ‘bird ‘travel) ‘flies)
(setf (get ‘bird ‘isa) animal)
(setf (get ‘fish ‘isa) animal)
(setf (get ‘fish ‘travel) ‘swim)
(setf (get ‘ostrich ‘isa) ‘bird)
(setf (get ‘ostrich ‘travel) ‘walk)
(setf (get ‘penguin ‘isa) ‘bird)
(setf (get ‘penguin ‘travel) ‘walk)
(setf (get ‘penguin ‘color) ‘brown)
(setf (get ‘opus ‘isa) ‘penguin)
(setf (get ‘canary ‘isa) ‘bird)
(setf (get ‘canary ‘color) ‘yellow)
(setf (get ‘canary ‘sound) ‘sing)
(setf (get ‘tweety ‘isa) ‘canary)
(setf (get ‘tweety ‘color) ‘white)
(setf (get ‘robin ‘isa) ‘bird)
(setf (get ‘robin ‘sound) ‘sings)
(setf (get ‘robin ‘color) ‘red)
```

Usando esta representação de redes semânticas, nós agora definimos funções de controle em herança hierárquica. Isto é simplesmente uma pesquisa junto ligações isa até um pai que é encontrado com a propriedade desejada. Os pais são pesquisados ao estilo um depth-first, a procura pára quando uma instância da propriedade é encontrado. Isto é típico dos algoritmos de herança fornecidos por muitos sistemas comerciais. Variações sobre esta abordagem incluem o uso de procura em largura como um estratégia de pesquisa de herança.
inherit-get é uma variação de ***get*** nas quais as primeiras tentativas tentam recuperar uma propriedade de um símbolo. se isto falha ***inherit-get*** chama ***get-from-parents*** para implementar a busca. ***get-from-parents*** toma como seu primeiro argumento seja um único pai ou uma lista de pais; o segundo argumento é um nome de propriedade.
Se **parents** for **nil** a busca pára com falha. Se **parents** for um atomo, então é chamado ***inherit-get*** onde o pai quer recuperar a propriedade do pai em si ou continuar a busca. Se **parents** for uma list ***get-from-parents*** chama a si mesmo de forma recursiva no **car** e no **cdr** da lista de **parents.** ***inherit-get*** é definido por:

```
(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)))))

Na próxima seção nós generalizamos nossas representações das coisas, aulas, e herança usando a biblioteca de programação orientada a objetos CLOS.

18.2 Programação Orientada a Objetos Usando CLOS

Objeto-orientação definida: Apesar das muitas vantagens da programação funcional, alguns problemas são mais bem conceituados em termos de objetos que possuem um estado que mudam ao longo do tempo. Programas de simulação são típicos desta. imagine tentar para construir um programa que irá prever a capacidade de um sistema de aquecimento a vapor aquecer um grande edifício: podemos simplificar o problema, pensando nele como um sistema de objetos (quartos, termostatos, caldeiras, tubos de vapor, etc.) que interagem para alterar a temperatura e o comportamento de um ao outro ao longo do tempo. Linguagens orientadas a objeto apoiar uma abordagem para a resolução de problemas que nos permite decompor um problema em objetos que interagem. Esses objetos têm uma estado que pode mudar ao longo do tempo, e um conjunto de funções ou métodos que definir comportamentos do objeto. Essencialmente, programação orientada a objetos permite-nos a resolver problemas através da construção de um modelo do domínio do problema como nós a entendemos. Esta abordagem baseada em modelo a a resolução de problemas é um natural apto para a inteligência artificial, uma metodologia de programação eficaz no seu direito próprio, e uma ferramenta poderosa para pensar sobre o problema complexo domínios.
Há um número de idiomas que suportam a programação orientado a objeto. Alguns dos mais importantes são Smalltalk, C ++, Java e o Common Lisp Object Systemt (CLOS).
À primeira vista, Lisp, com a sua raízes na programação funcional, e orientação a objetos, com sua ênfase sobre a criação de objetos que mantêm o seu estado ao longo do tempo, pode parecer mundos separados. No entanto, muitas características da linguagem, tais como a verificação de tipo dinâmico ea capacidade de criar e destruir objetos dinamicamente, torná-lo um ideal fundação para a construção de uma linguagem orientada a objeto. Certamente, foi Lisp a base para muitas das linguagens orientadas a objetos iniciais, como Smalltalk, Flavors, KEE, e ART. Como o padrão Common Lisp foi desenvolvido, a comunidade Lisp aceitou CLOS como a forma preferida para fazer a objetos programação em Lisp. A fim de apoiar plenamente as necessidades de programação orientada a objetos, umalinguagem de programação deve fornecer três capacidades: 1: *encapsulation*, 2) *polymorphism*, e 3) *inheritance.* O restante desta introdução descreve essas capacidades e uma introdução à maneira pela qual apoia CLOS elas.

Encapsulation: Todas as linguagens de programação modernas nos permitir criar estruturas de dados complexos que combinam itens de dados atômicas em um entidade única. Encapsulamento orientado a objeto é único em que ele combina ambos os itens de dados e os processos utilizados para a sua manipulação numa única estrutura, uma chamado *class*. Por exemplo, a
tipos de dados abstratos visto anteriormente (por exemplo, Seção 16.2) pode bastante adequadamente ser visto como classes. Em algumas linguagens orientadas a objeto, tais como Smalltalk, a encapsulação dos procedimentos (ou métodos como eles

são chamados na comunidade orientada a objetos) na definição de objeto é explícita. CLOS tem uma abordagem diferente, usando typechecking do Lisp para fornecer essa mesma capacidade. CLOS implementa métodos como *generic functions*. Estas funções verificar o tipo de seus parâmetros para garantia de que eles só podem ser aplicadas a casos de um certo tipo de classe. Isto dá-nos uma ligação lógica de métodos a seus objetos.

Polymorphism: A palavra polimórfica vem das raízes "Poly", o que significa muitos, e "morphos", significando forma. Uma função é polimórfico se tem muitos comportamentos diferentes, dependendo do tipos de seus argumentos. Talvez o exemplo mais intuitiva de funções polimórficas e sua importância é um programa de desenho simples. Assuma que definem-se objectos para cada uma das formas (quadrado, círculo, linha) que gostaria de desenhar. Uma maneira natural de implementar isso é definir um método chamado sorteio para cada classe de objeto. Embora cada método individual tem uma definição diferente, dependendo da forma que é desenhar, todos eles têm a mesma nome. Cada forma no nosso sistema tem um mesmo comportamento.Isto é muito mais simples e mais natural do que para definir um nome diferente function (draw-square, draw-circle, etc.) para cada forma. CLOS suporta polimorfismo através de funções genéricas. A função genérica é aquele cujo comportamento é determinado pelo tipo da sua argumentos. Em nosso exemplo de desenho, CLOS nos permite definir uma função genérica, para desenhar, que inclui o código para desenhar cada uma das formas definidas no programa. Na avaliação, ele verifica o tipo de seu argumento e automaticamente executa o código apropriado.

Inheritance: A herança é um mecanismo para a classe de apoio abstração em uma linguagem de programação. Ela nos permite definir geral classes que especificam a estrutura eo comportamento de sua especializações, assim como a classe "árvore" define os atributos essenciais de pinheiros, choupos, carvalhos e outras espécies diferentes. Na seção 18,1, nós construímos um algoritmo de herança para redes semânticas; esta demonstraram a facilidade de implementação de herança usando de técnicas incorporada do Lisp de estruturação de dados. CLOS nos fornece um algoritmo mais robusto, expressivo, com herança incorporada.

Definindo Classes e
Instâncias em CLOS: A estrutura de dados de base nos CLO é a classe. Uma classe é um especificação para um conjunto de instâncias de objetos. Nós definimos classes usando o macro ***defclass***. ***defclass*** tem a sintaxe:

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

<class-name> é um símbolo. Seguindo o nome da classe é uma lista de direta superclasses (chamada *superclass*); estes são os pais imediatos na hierarquia de herança da classe. Esta lista pode ser vazia. Seguindo a lista de classes pai é uma lista de zero ou mais ***slot-specifiers.*** *Um* ***slot-specifiers*** ou é o nome de um ***slot*** ou uma lista consistindo de um ***slot-name*** e zero ou mais ***slot-options:***

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

Por exemplo, podemos definir uma nova classe, ***rectangle*** que tem campos de valores para ***length*** e ***width:***

```
> (defclass rectangle ()
(length width))
#<standard-class rectangle>
```

make-instance nos permite criar instâncias de uma classe, tendo como argumento um nome de classe e retorna uma instância da classe. É as instâncias de uma classe que realmente armazenam valores de dados. Podemos ligar um símbolo, ***rect*** ao exemplo de ***rectangle*** usando ***make-instance*** e ***setq***:

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

As opções de campo em ***defclass*** definir as propriedades opcionais de slots. Opções de slot tem a sintaxe (onde "|" indica opções alternativas):

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

Declaramos opções de slot usando argumentos nomeados. Argumentos são uma forma de parâmetro opcional em uma função Lisp. A palavra-chave, que sempre começa com um ":", precede o valor para esse argumento. Opções de slot disponíveis incluem aqueles que fornecem acessores para um slot. A opção ***:reader*** define uma função chamada ***reader-function-name*** que retorna o valor de um slot de uma instância. A opção ***:writer*** define uma função chamada ***writer-function-name*** que vai escrever no slot. ***accessor*** define uma função que pode ler um valor de slot ou pode ser usado com ***setf*** para mudar seu valor.
No seguinte transcrição, nós definimos ***rectangle*** que tem slots para ***length*** e ***width,*** com os campos de ***accessors get-length*** e ***get-width***, respectivamente. Após ligar ***rect*** a ***rectancle*** usando ***make-instance,*** nos usamos ***accessor, get-length*** com ***setf*** para ligar o campo ***length*** ao valor 10. Finalmente, usamos ***accessor*** para ler esse valor:

```
> (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

Para além de definir ***accessors*** podemos acessar um slot usando a função primitiva ***slot-value. slot-value*** é definido para todos os slots'; toma como argumentos uma instância e um nome de slot e retorna o valor daquele slot. Podemos usar com ***setf*** para mudar o valor de um slot. Por exemplo, podemos usar ***slot-value*** para acessar ***width*** do campo ***rect.***

> (setf (slot-value rect ‘width) 5)
5
> (slot-value rect ‘width)
5

:allocation nos permite especificar a alocação de memória para um slot. ***allocation-type*** pode ser ***:instance*** ou ***:class.*** Se o tipo de alocação for ***:instance,*** então CLOS aloca um slot local para cada instância do tipo. Se a alocação for do tipo ***:class*** então todas as instâncias compartilham um único local para esse slot. Na alocação ***:class*** Todas as instâncias irão compartilhar o mesmo valor do slot; as alterações feitas para o slot por qualquer instância afetará todas as outras instâncias. Se você omitir o especificador ***:allocation,*** o padrão será ***:instance***.
:initarg nos permite especificar um argumento que podemos usar com ***make-instance*** para especificar um valor inicial de um slot. Por exemplo, podemos modificar a nossa definição de ***rectangle*** para permitir-nos para inicializar as instancias de ***length*** e ***width:***

> (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 nos permite especificar um formato que CLOS avalia em cada chamada para ***make-instance*** para calcular um valor inicial do slot. Por exemplo, se gostaríamos que o nosso programa para pedisse ao usuário por novos valores de cada nova instância do retângulo, podemos definir uma função para fazê-lo e incluí-lo em um ***:initform:***

```
> (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
```

Definindo Generic Functions e Methods: A função genérica é uma função cujo comportamento depende do tipo de seus argumentos. Em CLOS, funções genéricas contêm um conjunto de *métodos*, indexado pelo tipo de seus argumentos. Chamamos funções genéricas com uma sintaxe semelhante à de funções regulares; a função genérica recupera e executa o método associado com o tipo dos seus parâmetros.

CLOS utiliza a estrutura da hierarquia de classe na escolha de um método da função genérica; Se não existe um método definido directamente para uma discussão de uma determinada classe, ele usa o método associado ao ancestral "mais próximo" na hierarquia. Funções genéricas fornecem a maior parte das vantagens da Abordagens "mais puras" de métodos e passagem de mensagens, incluindo herança e sobrecarga. No entanto, eles estão muito mais próximos em espírito ao paradigma funcional de programação que constitui a base do Lisp. Por exemplo, nós podemos utilizar funções genéricas com ***mapcar, funcall,*** entre outras funções de alta-ordem do Lisp.

Nós definimos funções genéricas usando tanto ***defgeneric*** quanto ***defmethod***. ***defgeneric*** permite-nos definir uma função genérica e vários métodos usando um formulário.

defmethod permite-nos definir cada método separadamente, embora CLOS combina todos eles em uma única função genérica. ***defgeneric*** tem a sintaxe:

```
(defgeneric f-name lambda-list <method-description>*)
<method-description> ::= (:method specialized-lambdalist
 form)
```

defgeneric leva um nome da função, uma lista lambda lista seus argumentos, e uma série de zero ou mais descrições de método. Num método descrição, **specialized-lambda-list** é apenas como um lambda comum, lista uma definição de função, exceto que um parâmetro formal pode ser substituído com um (símbolo parameter-specializer) par: símbolo é o nome do parâmetro, e parameter-specializer é a classe dos argumentos. Se um argumento em um método não tem nenhum parameter-specializer, seus tipo padrãdo é T que é a classe mais geral de uma hierarquia de classe. Parâmetros de tipo T pode ligar-se a qualquer objeto. A lista lambda especializado de cada especificador de método deve ter o mesmo número de argumentos que a lista lambda na **defgeneric**. A **defgeneric** cria uma função genérica com o métodos especificados, substituindo quaisquer funções genéricas existentes.
Como um exemplo de uma função genérica, que pode definir classes de **rectangle** e **circle** e implementar os métodos adequados para a encontrar **area:**

```
(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)))

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

(setq rect (make-instance 'rectangle 'init-length 10
                          'init-width 5))

(setq circ (make-instance 'circle 'init-radius 7))
```

Podemos usar a função de área para calcular a área de uma forma:

```
> (area rect)
50
```

```
> (area circ)
153.93804002589985
```

Nós também podemos definir métodos usando **defmethod.** Sintaticamente, **defmethod** é semelhante ao **defun**, só que usa uma lista lambda especializada para declarar a classe à qual pertencem os seus argumentos. Quando se define um método utilizando **defmethod**, se não houver uma função genérica com aquele nome então **defmethod** cria um; se uma função genérica de que o nome já existe, **defmethod** adiciona um novo método para ele. Por exemplo, suponha que deseja adicionar a classe **square** para as definições acima, podemos fazer isso com:

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

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

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

defmethod não alterar a definições anteriores da função **area**; ele simplesmente adiciona um novo método para a função genérica:

```
> (area sqr)
36

> (area rect)
50

> (area circ)
153.93804002589985
```

Inheritance in CLOS: CLOS é uma linguagem de herança múltipla. Além de oferecer o programador um sistema representacional muito flexível, herança múltipla introduz o potencial para criar anomalias quando herdando slots e métodos. Se dois ou mais antepassados tem definido o mesmo método, é crucial para saber qual o método de qualquer instância desses antepassados iram herdar. CLOS resolver possíveis ambigüidades, definindo uma lista de classe de precedência, que é uma ordenação total de todas as classes dentro de uma hierarquia de classes.Cada **defclass** lista os pais diretos de uma classe da esquerda para a direita.
Usando a ordem dos pais diretos para cada classe, CLOS calcula um parcial ordenação de todos os antepassados na hierarquia de herança. A partir desta ordenação parcial, ele deriva o ordenamento total da lista de classe de precedência através de uma ordenação topológica. A lista de precedência segue duas regras:

1. Qualquer classe controladora direta precede qualquer antepassado mais distante.
2. Na lista de pais imediatos de **defclass**, cada classe precede aqueles à sua direita.

CLOS computa a lista de classe de precedência para um objeto topológico classificando suas classes ancestrais de acordo com o seguinte algoritmo.

1. Seja S o conjunto de C e todas as suas superclasses.
2. Para cada classe, c, em Sc, defina o conjunto de pares ordenados:

Rc = {(c, c1), (c1, c2), … (cn-1, cn)}

onde c1 através cn são os pais diretos de c na ordem eles são listados em **defclass**. Note-se que em cada Rc define-se uma ordem total.

3. Seja R a união do Rcs para todos os elementos de Sc. R pode ou não pode definir uma ordem parcial. Se não definir uma ordenação parcial, em seguida, a hierarquia é inconsistente e o algoritmo irá detectar isto.

4. Topologicamente classifique os elementos de R por:

a. Comece com uma lista de precedência vazio, P
b. Encontre uma classe em R não contendo antecessores. Adicione ao final de P e remova da classe de Sc e todos os pares contendo R. Se existem várias classes em Sc sem predecessores, selecione aquele que tem uma subclasse direta mais próxima ao fim na versão atual de P.
c. Repetir os dois passos anteriores até que nenhum elemento possa ser encontrado e que não tem qualquer antecessor em R.
d. Se Sc não estiver vazia, então a hierarquia é inconsistente; isto pode conter ambiguidades que não podem ser resolvidos usando esta técnica.

Como a lista de precedência resultante é uma ordenação total, ele resolve qualquer ordenações ambíguas que possam ter existido na hierarquia de classes. CLOS usa a lista de classe precedência na herança de faixas horárias e a seleção de métodos.
Na escolha de um método para aplicar a uma determinada chamada de uma função genérica, CLOS primeiro seleciona todos os métodos aplicáveis. Um método é aplicável a uma chamada de função genérica se cada um *parameter specializer* no método é coerente com o argumento correspondente na chamada de função genérica. Um *parameter specializer* é consistente com um argumento se o *specializer* corresponde à classe do argumento da classe ou de um dos seus antepassados.
CLOS em seguida, classifica todos os métodos aplicáveis, utilizando as listas de precedência dos argumentos.CLOS determina qual dos dois métodos devem vir em primeiro lugar esta ordenação por comparação da seu *parameter specializer* ao estilo da esquerda para a direita. Se o primeiro par correspondente de *parameter specializers* são iguais, CLOS compara o segundo, continuando deste modo até encontrar *parameter specializer* correspondentes que são diferentes. Destes dois, ela designa como mais

específica do método cujo parâmetro specializer aparece mais à esquerda na lista de precedência do argumento correspondente. Depois de pedir todos os métodos aplicáveis, a seleção do método padrão se aplica o método mais específico para os argumentos. Para mais detalhes, consulte Steele (1990).

18.3 Um exemplo CLOS: Uma simulação de um termóstato

As propriedades de programação orientada a objetos que o tornam um caminho natural para organizar implementações grandes e complexas de software são igualmente aplicáveis na concepção de bases de conhecimento. Além dos benefícios da herança de classe para representar o conhecimento taxonômico, a mensagem aspecto passando de sistemas orientados a objetos simplifica a representação de componentes que interagem.
Como um simples exemplo, considere a tarefa de modelação do comportamento de um aquecedor de vapor durante um pequeno edifício de escritórios. Podemos, naturalmente, ver este problema em termos de componentes que interagem. Por exemplo:

- Cada escritório tem um termostato que transforma o calor em que o escritório li gado e desligado; isso funciona independentemente dos termostatos em outros escritórios.
- Cada escritório tem um termostato que transforma o calor em que o escritório l igado e desligado; isso funciona independentemente dos termostatos em outros escritórios.
- Cada escritório tem um termostato que transforma o calor em que o escritório ligado e desligado; isso funciona independentemente dos termostatos em outros escritórios.
- A quantidade de vapor que o sistema pode rota para um único escritório é afetado pela procura total no sistema.

Estes pontos são somente algumas daquelas que devem ser tidas em conta em modelar o comportamento de um tal sistema; as possíveis interacções são extremamente complexo.
Uma representação orientada a objeto permite que o programador para se concentrar em descrever uma classe de objetos de cada vez. Nós representariamos termostatos, por exemplo, por a temperatura a que eles chamam por calor, juntamente com a velocidade com a qual eles respondem às mudanças de temperatura.
A planta de vapor pode ser caracterizada em termos da quantidade máxima de calor que pode produzir, a quantidade de combustível utilizada como uma função do calor produzido, a quantidade de tempo necessário para responder a um aumento da procura de calor, e a velocidade a que ele consome água.
Um quarto pode ser descrito em termos do seu volume, a perda de calor através das suas paredes e janelas, o ganho de calor do vizinho e a taxa à qual o radiador adiciona calor para o ambiente.
A base de conhecimento é construído de classes como **room** 2 **thermostat**, que definem as propriedades da classe, e casos como **room-322** 2 **thermostat-211**, qual o modelo situações individuais.

As interacções entre os componentes são descritos por mensagens entre instâncias. Por exemplo, a mudança de temperatura em **room** causaria uma mensagem a ser enviada para uma instância da classe **thermostat.** Se a nova **temperature** é baixa o suficiente, o **thermostat** iria mudar depois de um atraso apropriado. Isso faria com que uma mensagem seja enviada para o **heater** solicitando mais calor. Isto faria com que o **heater** consumisse mais óleo, ou, se já estiver operando em capacidade máxima, para encaminhar algum calor longe de outros quartos para responder à nova demanda. Isso faria com que outra **thermostat** ligasse, e assim por diante.
Usando esta simulação, que podem testar a capacidade do sistema para responder às alterações externas de temperatura, medir o efeito de perda de calor, ou determinar se o aquecimento projectada é adequada. Nós poderíamos usar esta simulação em um programa de diagnóstico para verificar se a hipótese de uma falha poderia realmente produzir um determinado conjunto de sintomas. Por exemplo, se temos razão para acreditar que um problema de aquecimento é causado por um tubo de vapor bloqueado, poderíamos introduzir tal falha na simulação e ver se ele produz os sintomas observados.
A coisa importante sobre esse exemplo é a maneira pela qual orientada a objeto permite uma abordagem que os engenheiros de conhecimento usam para lidar com a complexidade da simulação. Que lhes permite construir o modelo de uma peça de cada vez, incidindo apenas sobre os comportamentos de classes simples de objetos. A complexidade do comportamento do sistema emerge quando nós executamos o modelo
A base da nossa implementação CLOS deste modelo é um conjunto de objeto definições. **Thermostat** tem um único slot chamado **setting. setting** de cada instância é inicializada para 65 usando **initform**, **heater-thermostat** é uma subclasse de **thermostat** para controlar **heater** (ao contrário de condicionadores de ar); eles têm um único slot que será ligado a uma instância da classe **heater**. Note-se que slot de **heater** tem uma atribuição de classe; este capta a restrição de que o **thermostat** em diferentes salas de um edifício controlar os edifícios individualmente, **heater-obj**.

```
(defclass thermostat ()
  ((setting :initform 65
            :accessor therm-setting)))

(defclass heater-thermostat (thermostat)
  ((heater :allocation :class
           :initarg heater-obj)))
```

Um **heater** tem um estando (on e off) que é inicializado como **off**, e uma **location.** E há também um campo **rooms-heated,** que será vinculado a uma lista de objetos do tipo **room.** Note-se que as instâncias, como qualquer outra estrutura em Lisp,
podem ser elementos de uma lista.

```
(defclass heater ()
  ((state :initform ‘off
          :accessor heater-state)
   (location :initarg loc)
   (rooms-heated)))
```

room tem os para **temperature**, inicializadas a 65 graus; **thermostat**, que será ligado a uma instância de **thermostat**; e **name**, o nome de **room**.

Estas definições de classe que definem a hierarquia da Figura 18.2.

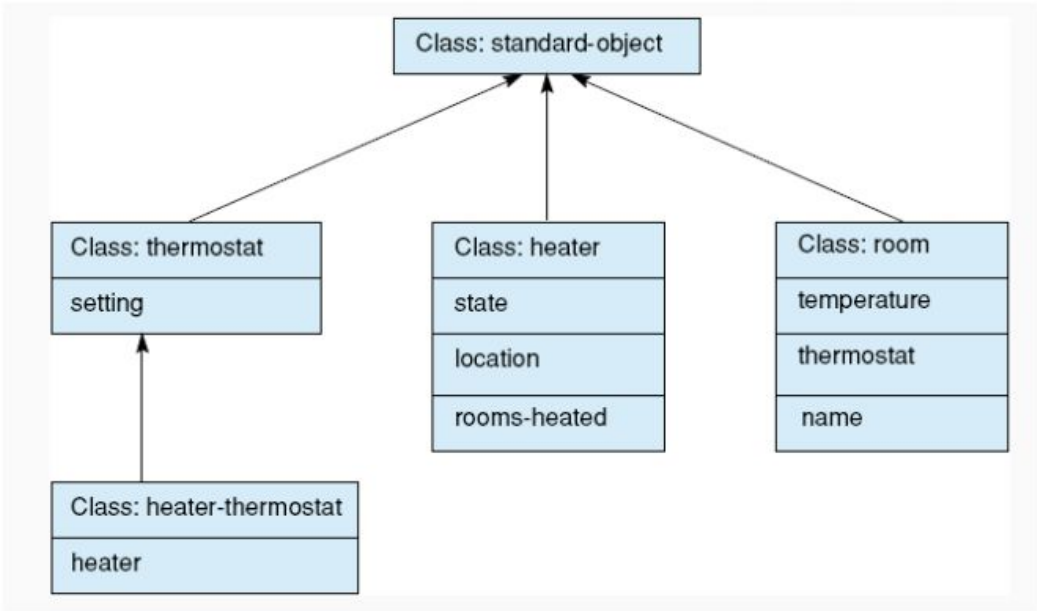

Figure 18.2. A class hierarchy for the room/heater/thermostat simulation.

Nós representamos nossa simulação em particular como um conjunto de instâncias dessas classes. Vamos implementar um sistema simples de um **room**, um **heater** e um **thermostat.**

```
(defparameter *office-heater* (make-instance 'heater 'loc 'office))

(defparameter *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*))
```

Figura 18.3 mostra a definição de casos, a atribuição de slots, e as ligações slots para valores.

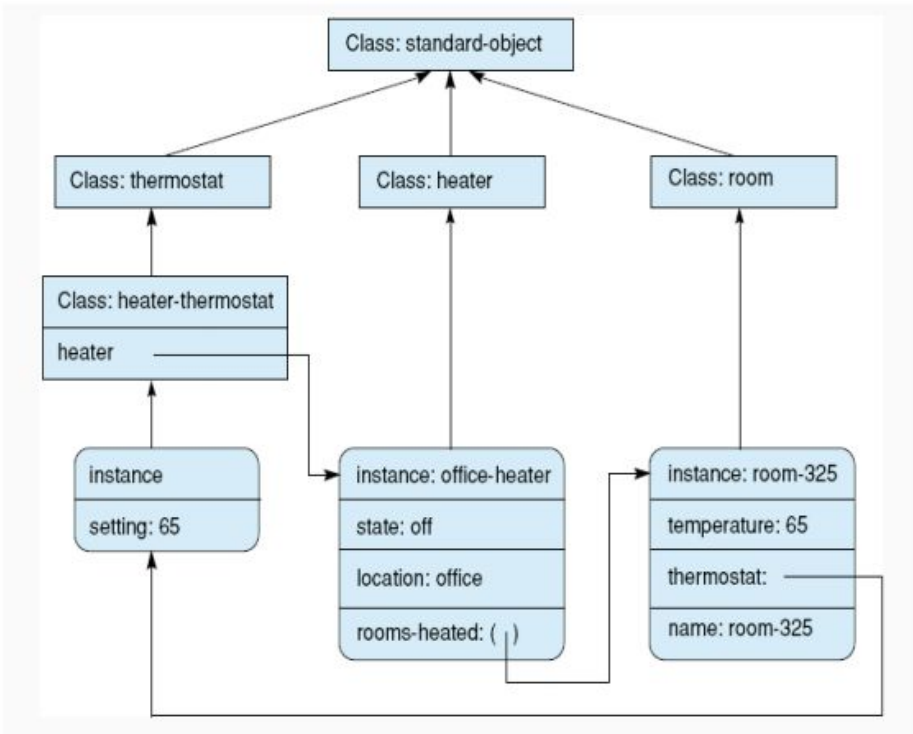

Figure 18.3. The creation of instances and binding of slots in the simulation.

Nós definimos o comportamento dos quartos através dos métodos **change-temp, check-temp,** e **change-setting. change-temp** mudam a **temperature** de **room** para um novo valor, imprime uma mensagem para o usuário, e chamadas **check-temp** para determinar o **heater**. Da mesma forma, **change-setting** muda o ajuste do termostato, **therm-setting** e chama **check-temp** que simula o **thermostat.** Se a **temperature** de **room** é menor do que a regulação do termostato, então é enviado ao **heater** uma mensagem para mudar para **on**, caso contrario, **off.**

```
(defmethod change-temp ((place room) temp-change)
  (let ((new-temp (+ (room-temp place) temp-change)))
    (setf (room-temp place) new-temp)
    (terpri)
    (prin1 "the temperature in ")
    (prin1 (room-name place))
    (prin1 " is now ")
    (prin1 new-temp)
    (terpri)
    (check-temp place)))
(defmethod change-setting ((room room) new-setting)
  (let ((therm (room-thermostat room)))
    (setf (therm-setting therm) new-setting)
    (prin1 "changing setting of thermostat in ")
    (prin1 (room-name room))
```

```
    (prin1 " to ")
    (prin1 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)))))
```

Os métodos de controlar o estado de aquecimento do aquecedor e alterar a temperatura das salas. **send-heater** toma como argumentos uma instancia de **heater** e a mensagem, **new-state.** Se **new-state** está **on** então o metodo **turn-on** é chamado para inicializar o **heater**; Se **new-state** é **off**, **heater** é desativado. Depois de ligar o aquecedor **send-heater** chama **heat-rooms** para aumentar a temperatura de cada quarto em um grau.

```
(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 (eql (heater-state heater) 'on)
             (turn-off heater)))))

(defmethod turn-on ((heater heater))
  (setf (heater-state heater) 'on)
  (prin1 "turning on heater in ")
  (prin1 (slot-value heater 'location))
  (terpri))

(defmethod turn-off ((heater heater))
  (setf (heater-state heater) 'off)
  (prin1 "turning off heater in ")
  (prin1 (slot-value heater 'location))
  (terpri))

;;; this function raises the temperature of a list of rooms
(defun heat-rooms (rooms amount)
  (cond ((null rooms) nil)
        (t (change-temp (first rooms) amount)
           (heat-rooms (rest rooms) amount))))
```

A transcrição a seguir ilustra o comportamento da simulação.

```
> (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