5. CLOS
Orientação a Objetos em LISP

Traduzido de:

- Brief Guide to CLOS

- Written by Jeff Dalton, University of Edinburgh, <J.Dalton@ed.ac.uk>
  Modified by Bruno Haible <haible@ma2s2.mathematik.uni-karlsruhe.de>

Notação: "Thing*" significa zero ou mais ocorrências de thing; "thing+" significa uma ou mais. Chaves { e } são usadas para agrupar, como em {a b}+, o que significa **`a b, a b a b, a b a b a b`**, etc.

Nota específica a CLISP: antes de usar CLOS, você precisa executar **`(USE-PACKAGE "CLOS")`**.

5.1. Definindo Classes

Você define uma classe com o form **`DEFCLASS`**:

```
(DEFCLASS node-de-classe (nome-de-superclasse*)
   (descrição-de-slots*)
   opções-de-classe*
  )
```

Para coisas simples, ignore por enquanto as **`opções-de-classe`**.

Uma descrição-de-slot tem a forma (slot-name slot-option*), onde cada opção é uma palavra-chave seguida de um nome, expressão ou o que quer que seja. As opções mais úteis são

- :ACCESSOR function-name
- :INITFORM expression
- :INITARG symbol

DEFCLASS é semelhante a **DEFSTRUCT**.

A sintaxe é um pouco diferente e você tem mais controle sobre quais coisas são chamadas. Por exemplo, considere o **DEFSTRUCT**:

```
(defstruct person
   (name 'bill)
   (age 10)
)
```

DEFSTRUCT iria automaticamente definir campos com expressões para computar valores iniciais default, funções de acesso como **PERSON-NAME** para acessar e setar os valores dos campos, e uma **MAKE-PERSON** que pegaria argumentos de inicialização com palavras-chave como em:

```
(make-person :name 'george :age 12)
```

Um DEFCLASS que proveria funções de acesso similares, etc, seria:

```
(defclass person ()
  ((name :accessor person-name
         :initform 'bill
         :initarg :name
   )
   (age :accessor person-age
        :initform 10
        :initarg :age
) ))
```

Observe que **DEFCLASS** lhe permite controlar quais coisas são chamadas. Você não precisaria chamar o método de acesso para **name PERSON-NAME**. Você poderia simplesmente chamá-lo **NAME**.

Em geral, você deverá pegar nomes que façam sentido dentro da sua aplicação ao invés da estrutura rígida do **DEFSTRUCT**.

Você não precisa prover todas as opções para cada campo (slot, membro). Talvez você não deseje que seja possível inicializar um determinado slot quando chamar **MAKE-INSTANCE** (explicado adiante). Neste caso, não proveja um :**INITARG**. Talvez também não haja um valor default que faça sentido, talvez os valores significativos sejam somente definidos por subclasses. Neste caso, não crie um :**INITFORM**.

Observe que uma classe, uma vez definida, também é um objeto. Para obter um objeto-classe a partir de seu nome, use **(FIND-CLASS name)**. Ordinarily, you won't need to do this.

5.2. Instâncias

Você pode criar uma instância de uma classe com **MAKE-INSTANCE**. É similar às funções **MAKE-x** definidas por **DEFSTRUCT**, porém permite que você passe a classe a ser instanciada como argumento:

```
(MAKE-INSTANCE class {initarg value}*)
```

Ao invés do objeto classe ele mesmo (é o que o símbolo de nome de classe está denotando: uma variável global que representa o objeto-classe) você pode simplesmente usar o seu nome. Por exemplo:

```
(make-instance 'person :age 100)
```

Esta pessoa teria idade 100 e nome Bill por default.

É muitas vezes uma boa idéia que você defina os seus próprios métodos construtores, ao invés de chamar **MAKE-INSTANCE** diretamente, porque assim você pode encapsular detalhes de implementação e não precisa utilizar parâmetros-de-palavra-chave para nada. Você poderia desejar definir:

```
(defun make-person (name age)
  (make-instance 'person :name name :age age)
)
```

caso queira que o nome e a sejam requeridos através de parâmetros posicionais.

Os métodos de acesso podem ser usados para pegar e setar valores de slots (variáveis de instância):

```
<cl> (setq p1 (make-instance 'person :name 'jill :age 100))
#<person @ #x7bf826>

<cl> (person-name p1)
jill

<cl> (person-age p1)
100

<cl> (setf (person-age p1) 101)
101

<cl> (person-age p1)
101
```

Note que quando você usa **DEFCLASS**, as instâncias são impressas usand-se a notação **#<...>**, ao invés de **#s(person :name jill :age 100)**.
Mas você pode modificar a maneira pela qual instâncias são impressas através da definição de métodos na função genérica **PRINT-OBJECT**.

Slots podem também ser acessados por nome usando-se **(SLOT-VALUE instance slot-name)**:

```
<cl> (slot-value p1 'name)
jill

<cl> (setf (slot-value p1 'name) 'jillian)
jillian

<cl> (person-name p1)
jillian
```

Você pode descobrir detalhes a respeito de uma instância chamando **DESCRIBE**:

```
<cl> (describe p1)
 #<person @ #x7bf826> is an instance of class
      #<clos:standard-class person @ #x7ad8ae>:
 The following slots have :INSTANCE allocation:
 age      101
 name     jillian
```

5.3. Herança de opções de slot

As classes acima não possuíam superclasses. Por causa disso havia um "()" após, por exemplo **"defclass person"**. Na verdade, isto significa, ao contrário de linguagens como C++, que o objeto **tem** uma superclasse: a classe **STANDARD-OBJECT**.

Quando há superclasses explícitas, uma subclasse pode especificar um slot que já foi antes especificado para a superclasse. Quando isto acontece, a informação definida nas opções-de-slot será combinada. Para as opções de slot listadas, ou há uma sobrecarga ou uma união das opções:

:ACCESSOR -- união
:INITARG -- união
:INITFORM -- sobrecarga

Isto é o que você deveria esperar. A subclasse pode modificar o valor default inicial através da sobrecarga do **:INITFORM**, e pode adicionar opções aos argumentos de inicialização e aos métodos de acesso (acessores).

No entanto, a ***união*** para acessores é somente uma conseqüência de como funções genéricas trabalham. Se elas podem ser aplicadas a instances de uma classe C, elas também poderão ser aplicadas a instâncias de subclasses de C.

(Métodos de acesso são genéricos, isto se tornará mais claro mais tarde)

Aqui alguns exemplos de subclasses:

```
<cl> (defclass teacher (person)
        ((subject :accessor teacher-subject
                  :initarg :subject
     ) ))
#<clos:standard-class teacher @ #x7cf796>

<cl> (defclass maths-teacher (teacher)
        ((subject :initform "Mathematics"))
     )
#<clos:standard-class maths-teacher @ #x7d94be>

<cl> (setq p2 (make-instance 'maths-teacher
                  :name 'john
                  :age 34
     )        )
#<maths-teacher @ #x7dcc66>

<cl> (describe p2)
#<maths-teacher @ #x7dcc66> is an instance of
 class #<clos:standard-class maths-teacher @ #x7d94be>:
 The following slots have :INSTANCE allocation:
 age        34
 name       john
 subject    "Mathematics"
```

Note que classes imprimem-se como `#<clos:standard-class maths-teacher @ #x7d94be>`. A notação `#<...>` usualmente tem a forma:

```
#<classe-do-objeto ... mais informação ...>
```

Assim uma instância de `maths-teacher` se imprime `#<MATHS-TEACHER ...>`. A notação das classes acima indica que são instâncias de `STANDARD-CLASS`. `DEFCLASS` define classes padrão.
`DEFSTRUCT` define classes-estrutura.

5.4. Herança Múltipla

Uma classe pode possuir mais de uma superclasse. Com a herança simples (somente uma superclasse), é muito fácil ordenar-se superclasses do mais geral para o mais específico. Esta é a regra:

Regra 1: Cada classe é mais específica que suas superclasses.

In multiple inheritance this is harder. Suppose we have

```
(defclass a (b c) ...)
```

Classe A é mais específica do que B ou C (do ponto de vista de instâncias de A), mas o que ocorre se algo (um :INITFORM, ou um método) é especificado por B e C? Qual sobrecarrega qual? A regra em CLOS é a de que superclasses listadas mais tarde são mais específicas do que superclasses listadas antes. Assim:

Regra 2: Para uma classe dada, superclasses listadas antes são mais específicas do que aquelas listadas mais tarde.

Essas regras são usadas para computar uma ordem linear para uma classe e todas as suas superclasses, da mais específica até a menos específica. Esta ordem é a lista de precedência de classes desta classe.

As duas ordens acima não são sempre suficientes para determinar uma ordem única, no entanto, assim CLOS tem um algoritmo para resolver a maioria dos problemas.

Isto pelo menos garante que todas as implementações vão produzir sempre a mesma ordem, mas geralmente é considerado uma idéia ruim programadores confiarem somente nesta ordem.

Se a ordem de algumas superclasses é importante, isto pode ser definido a mão diretamente na definição de classe.

5.5. Funções e Métodos Genéricos

Funções genéricas (métodos genéricos) em CLOS são a coisa mais próxima de mensagens que se tem. A invés de se escrever:

```
(SEND instance mensagem arg*)
```

você escreve:

```
(mensagem instance arg*)
```

As mensagens/operações são funções genéricas cujo comportamento pode ser definido para instâncias de classes particulares através da definição de ***métodos***.

(DEFGENERIC function-name lambda-list) pode ser usado para definir uma função genérica. Você não precisa chamar **DEFGENERIC**, no entanto, porque **DEFMETHOD** automaticamente define a função genérica, caso não tenha sido definida ainda. Por outro lado, é boa prática de programação utilizar-se **DEFGENERIC** como uma declaração de que uma operação existe e possui certos parâmetros.

De qualquer forma, todas as coisas interessantes ocorrem nos métodos. Um método é definido por:

```
(DEFMETHOD generic-function-name specialized-lambda-list
     form*
  )
```

Isto pode parecer meio complicado, mas se você olhar para a definição de uma função com **DEFUN** escrita da mesma forma:

```
(DEFUN function-name lambda-list form*)
```

Uma "**lambda list**" é somente uma lista de parâmetros formais, mais coisas como **&OPTIONAL** ou **&REST**. É por causa dessas extensões que dizemos "**lambda-list**" invés de "(**parâmetro***)" quando descrevemos a sintaxe. Mais detalhes em CLtL.

Assim uma função normal possui uma lista lambda como (var1 var2 ...). Um método possui uma na qual cada parâmetro deve ser "especializado" para uma classe particular. Assim se parece com:

```
((var1 class1) (var2 class2) ...)
```

O especializador é opcional. Omití-lo significa que o método pode ser aplicado a instâncias de qualquer classe, incluindo classes que não foram definidas por **DEFCLASS**. Por exemplo:

```
(defmethod muda-matéria ((teach teacher) nova-matéria)
     (setf (teacher-subject teach) nova-matéria)
)
```

Aqui, a **nova-matéria** poderia ser qualquer objeto. Se você desejar restringí-la, faça algo como:

```
(defmethod change-subject ((teach teacher)
                           (new-subject string))
     (setf (teacher-subject teach) new-subject)
  )
```

Ou você poderia definir uma classe de matérias.

Metodos em OOP "clássica" só especializam o primeiro parâmetro. Em CLOS, você pode especializar mais de um. Se você o faz assim, alguns autores chamam este método de multi-método.

Um método definido para uma classe C sobrecarrega qualquer método definido para superclasses de C. O método de C é "mais específico" do que métodos de mesmo nome de superclasses.

Para multi-métodos, a determinação de qual método é mais específico envolve mais de um parâmetro. Estes são considerados da esquerda para a direita.

```
(defmethod test ((x number) (y number))
  '(num num)
)

(defmethod test ((i integer) (y number))
  '(int num)
)

(defmethod test ((x number) (j integer))
  '(num int)
)

(test 1 1)       =>  (int num), not (num int)
(test 1 1/2)     =>  (int num)
(test 1/2 1)     =>  (num int)
(test 1/2 1/2)   =>  (num num)
```

5.6. Combinação de Métodos

Quando mais de uma classe define um método para uma função genérica (função aqui é visto como o nome do método) e mais de um método for aplicável a um conjunto de argumentos, os métodos aplicáveis são combinados em um único “método efetivo”. Cada definição de método individual é então somente parte da definição do método efetivo.

Um tipo de combinação de métodos é sempre suportado por CLOS. è chamado de combinação de métodos padrão. Também é possível definir-se novos tipos de combinação de métodos. Combinação de métodos padrão envolve quatro tipos de métodos:

1. Métodos primários do corpo principal do método efetivo.
   Somente o método primário mais específico é chamado, mas ele pode chamar o mais específico seguinte através de:

   ```
   (call-next-method)
   ```

2. Métodos **`:BEFORE`** são todos chamados antes do método primário, com o método **`:BEFORE`** mais específico chamado primeiro.

3. :AFTER methods are all called after the primary method, with the most specific :AFTER method called last.

4. Métodos **`:AROUND`** são executados antes de outros métodos. Da mesma forma que com métodos primários, somente o mais específico é chamado e o resto pode ser invocado por **`CALL-NEXT-METHOD`**. Quando o **`:AROUND`** menos específico chama **`CALL-NEXT-METHOD`**, o que ele chama é a combinação de **`:BEFORE, :AFTER`**, e métodos primários.

Métodos **:BEFORE, :AFTER**, e **:AROUND** são indicados através do qualificador correspondente na definição do método. Métodos **:BEFORE, :AFTER** são os mais fáceis de se usar, e um exemplo simples pode mostrar como funcionam:

```
(defclass food () ())

(defmethod cook :before ((f food))
  (print "A food is about to be cooked.")
)
(defmethod cook :after ((f food))
  (print "A food has been cooked.")
)
(defclass pie (food)
   ((filling :accessor pie-filling :initarg :filling
             :initform 'apple))
)
(defmethod cook ((p pie))
  (print "Cooking a pie")
 (setf (pie-filling p) (list 'cooked (pie-filling p)))
)
(defmethod cook :before ((p pie))
  (print "A pie is about to be cooked.")
)
(defmethod cook :after ((p pie))
  (print "A pie has been cooked.")
)
(setq pie-1 (make-instance 'pie :filling 'apple))
```

E agora:

```
<cl> (cook pie-1)
"A pie is about to be cooked."
"A food is about to be cooked."
"Cooking a pie"
"A food has been cooked."
"A pie has been cooked."
(cooked apple)
```

5.7. Quick Reference

Colchetes ("[" and "]") indicam elementos opcionais. Uma barra vertical ("|") indica uma alternativa. Thing* significa zero ou mais ocorrências de thing, thing+ uma ou mais ocorrências de thing. "{" e "}" são usadoa para agrupamento.

Definindo uma classe:

```
(DEFCLASS class-name (superclass-name*)
  (slot-description*)
)
```

Slot descriptions:

```
(slot-name slot-option*)

:ACCESSOR function-name
:INITFORM expression
:INITARG keyword-symbol
```

Instâncias:

```
(MAKE-INSTANCE class {initarg value}*)
```

Definições de métodos:

```
(DEFMETHOD generic-function-name [qualifier]
                specialized-lambda-list
  form*
)
```

onde
- generic-function-name é um símbolo;
- **qualifier** é **:BEFORE, :AFTER, :AROUND**, ou algo omitido;
- **specialized-lambda-list** é

```
( {variable | (variable class-name)}* )
```

Algumas funções:

```
(DESCRIBE instance)
(DESCRIBE class)

(FIND-CLASS class-name) -> classe

(CLASS-NAME class) -> símbolo

(CLASS-PRECEDENCE-LIST class) -> lista de classes
[Chamado (CLOS::CLASS-PRECEDENCE-LIST class) em CLISP.]

(CLASS-DIRECT-SUPERCLASSES class) -> lista de classes
[Chamado (CLOS::CLASS-DIRECT-SUPERCLASSES class) em CLISP.]

(CLASS-DIRECT-SUBCLASSES class) -> list of classes
[Não existe em CLISP.]
```

5.8. Exercício

Reimplemente o seu programa da Mercearia utilizando CLOS. A sua modelagem deve ter pelo menos as seguintes classes:

- A mercearia
- O estoque (lista de produtos que pode se atualizar, se ordenar, etc)
- A lista de compras.

Opção: Implemente o seu programa para resolver um problema de matemática discreta em CLOS.