Linguaggi di Programmazione
AA 2013-2014
Progetto Gennaio 2014
OOΛ′

Marco Antoniotti, Pietro Braione e Giuseppe Vizzari, DISCo

3 Gennaio 2014

1 Scadenza

La consegna di questo elaborato è fissata per il 31 Gennaio 2014 alle ore 23:59 GMT+1.

2 Introduzione

Ai tempi di Simula e del primo Smalltalk, molto molto tempo prima di Python, Ruby, Perl e SLDJ, i programmatori Lisp già producevano una pletora di linguaggi *object oriented*. Il vostro progetto consiste nella costruzione di un sistema siffatto.

OOΛ′ è un semplice linguaggio object-oriented con eredità singola. Il suo scopo è didattico e mira soprattutto ad evidenziare aspetti dell'implementazione di linguaggi object-oriented: (1) il problema di dove e come recuperare i valori ereditati, (2) come rappresentare i metodi e le loro chiamate e (3) come manipolare il codice nei metodi stessi.

Dovrete produrre una versione di **OOΛ′** in Common Lisp.

3 OOΛ′ in Common Lisp

Le primitive di **OOΛ′** sono tre: `define-class`, `new` e `get-slot`.

1. `define-class` definisce la struttura di una classe e la memorizza in una locazione centralizzata (una variabile globale).

   La sua sintassi è:

   ```
   '(' define-class <class-name> <parent> <slot-value>* ')'
   ```

   dove `<class-name>` e `<parent>` sono simboli, e ogni `slot` (o *campo*) è una coppia siffatta:

   ```
   slot-value ::= <slot-name> <value>
              |   <method-name> <method-exp>
   method-exp ::= '(' method <arglist> <form>* ')'
   ```

dove `<slot-name>` e `<method-name>` sono simboli, `<value>` è un oggetto qualunque, `<arglist>` è una lista di parametri standard Common Lisp e `<form>` è una qualunque espressione Common Lisp. Si noti che `<class-name>`, `<parent>`,`<slot-name>`, `<value>`, `<method-name>`, e `<method-exp>` sono espressioni valutate.

Il valore ritornato da `define-class` è `<class-name>`.

2. `new`: crea una nuova *istanza* di una classe. La sintassi è:

```
'(' new <class-name> [<slot-name> <value>]* ')'
```

dove `<class-name>` e `<slot-name>` sono simboli, mentre `<value>` è un qualunque valore Common Lisp. *Attenzione: le parentesi quadre sono solo parte della grammatica.*

Il valore ritornato da `new` è la nuova istanza di `<class-name>`. Naturalmente `new` deve controllare che gli slot siano stati definiti per la classe.

3. `get-slot`: estrae il valore di un campo da una classe. La sintassi è:

```
'(' 'get-slot' <instance> <slot-name> ')'
```

dove `<instance>` è una istanza di una classe e `<slot-name>` è un simbolo. Il valore ritornato è il valore associato a `<slot-name>` nell'istanza (tale valore potrebbe anche essere ereditato dalla classe o da uno dei suoi antenati). Se `<slot-name>` non esiste nella classe dell'istanza (ovvero se non è ereditato) allora viene segnalato un errore.

3.1 Esempi

Creiamo una classe `person`

```
CL prompt> (define-class 'person nil 'name "Eve" 'age "undefined")
PERSON
```

Ora creiamo una sottoclasse `student`

```
CL prompt> (define-class 'student 'person
                         'name "Eva Lu Ator"
                         'university "Berkeley"
                         'talk '(method ()
                                        (princ "My name is ")
                                        (princ (get-slot this 'name))
                                        (terpri)
                                        (princ "My age is ")
                                        (princ (get-slot this 'age))))
STUDENT
```

Ora possiamo creare delle istanze delle classi `person` e `student`.

```
CL prompt> (defparameter eve (new 'person))
EVE

CL prompt> (defparameter adam (new 'person 'name "Adam"))
ADAM
```

```
CL prompt> (defparameter s1 (new 'student 'name "Eduardo De Filippo" 'age 108))
S1

CL prompt> (defparameter s2 (new 'student))
S2
```

...e possiamo anche ispezionarne i contenuti.

```
CL prompt> (get-slot eve 'age)
"undefined"

CL prompt> (get-slot s1 'age)
108

CL prompt> (get-slot s2 'name)
"Eva Lu Ator"

CL prompt> (get-slot eve 'address)
Error: unknown slot.
```

3.2 "Metodi"

Un linguaggio ad oggetti deve fornire la nozione di *metodo*, ovvero di una funzione in grado di eseguire codice associato[1] alla "classe". Ad esempio, considerate la classe `student` definita sopra.

Domanda: come invochiamo il metodo `talk`?

In Java, una volta definita la variabile `s1`, il metodo verrebbe invocato come `s1.talk`[2]. In Common Lisp ciò sarebbe...inelegante. In altre parole vogliamo mantenere la notazione funzionale e rendere possibili chiamate come la seguente:

```
CL prompt> (talk s1)
My name is Eduardo De Filippo
My age is 108
108
```

Naturalmente, se non c'e un metodo appropriato associato alla classe dell'istanza, l'invocazione deve generare un errore.

```
CL prompt> (talk eve)
Error: no method or slot named TALK found.
```

Infine, il metodo deve essere invocato correttamente su istanze di sotto-classi. Ad esempio:

```
CL prompt> (define-class 'studente-bicocca 'studente
                         'talk '(method ()
                                        (princ "Mi chiamo ")
                                        (princ (get-slot this 'name))
                                        (terpri)
                                        (princ "e studio alla Bicocca.")
```

[1] Tralasciamo, per questo progetto le questioni di *incapsulamento* e *visibilità*.

[2] In C++ come `s1.talk`, `s1->talk` o `(*s1).talk` a seconda del tipo associato alla variabile `s1`.

```
                                    (terpri))
                            'university "UNIMIB")
STUDENTE-BICOCCA

CL prompt> (defparameter ernesto (new 'studente-bicocca
                                      'name "Ernesto"))
ERNESTO

CL prompt> (talk ernesto)
Mi chiamo Ernesto
e studio alla Bicocca
NIL
```

Il problema è quindi *come definire automaticamente la funzione* ***talk*** *in modo che sia in grado di riconoscere le diverse istanze.*

3.3 Suggerimenti ed Algoritmi

Per realizzare il progetto vi si consiglia di implementare ogni classe ed istanza come delle list con alcuni elementi di base e con una *association list* che contiene le associazioni tra campi e valori.

Si suggerisce di realizzare ***prima*** il meccanismo di manipolazione dei campi (o "slot") in modo che i meccanismi di ereditarietà funzionino correttamente. Solo ***dopo*** questo passo, è possibile attaccare il problema della definizione corretta dei metodi.

Il codice riportato di seguito vi sarà utile per implementare `define-class` e la manipolazione dei metodi.

```
(defparameter *classes-specs* (make-hash-table))

(defun add-class-spec (name class-spec)
  (setf (gethash name *classes-specs*) class-spec))

(defun get-class-spec (name)
  (gethash name *classes-specs*))
```

`make-hash-table` e `gethash` manipolano le hash tables in Common Lisp. La forma di `class-spec` è un dettaglio implementativo.
Si consiglia di rappresentare le istanze come liste[3] dalla forma

```
'(' ool-instance <class> <slot-value>* ')'
```

3.3.1 Come si recupera il valore di uno slot in un'istanza

Per recuperare il valore di uno slot in un'istanza, metodo o semplice valore[4] che sia, prima si guarda dentro all'istanza, se si trova un valore allora lo si ritorna, altrimenti si recupera la specifica della classe dell'istanza e si cerca il valore lì, se si trova un valore lo si ritorna, altrimenti lo si ricercherà nella specifica del genitore della classe e così via. Se non c'è genitore si genera un errore.

Domanda 1. Posso associare dei metodi ad un'istanza?

Domanda 2. Ho bisogno di un `Object`?

[3] Vi sono alternative.
[4] Fa differenza?

Domanda 3. Cosa succede in questo caso?

```
cl-prompt> (get-slot s1 'talk) ; S1 è l'istanza "Eduardo"
```

3.3.2 Come si "installano" i metodi

Per la manipolazione dei metodi dovete usare la funzione qui sotto per generare il codice necessario (n.b.: richiamata all'interno della `define-class` o della `new`).

```
(defun process-method (method-name method-spec)
  #| ... and here a miracle happens ... |#
  (eval (rewrite-method-code method-name method-spec)))
```

Notate che `rewrite-method-code` prende in input il nome del metodo ed una S-expression siffatta `'(' method <arglist> <form>* ')'` (si veda la definizione di `define-class`) e la riscrive in maniera tale da ricevere in input anche un parametro `this`.

Ovviamente, `rewrite-method-code` fa metà del lavoro. L'altra metà la fa il codice che deve andare al posto di `#| ... and here a miracle happens... |#`. Questo codice deve fare due cose

1. creare una funzione `lambda` anonima che si preoccupi di recuperare il codice vero e proprio del metodo nell'istanza, e di chiamarlo con tutti gli argomenti del caso;

2. associare la suddetta `lambda` al nome del metodo.

Il punto (1) è relativamente semplice ed è una semplice riscrittura di codice; la funzione anonima creata è a volte chiamata funzione-*trampolino* (*trampoline*) per motivi che si chiariranno da sè durante la stesura del codice. Il punto (2) richiede una spiegazione. Il sistema Common Lisp deve avere da qualche parte delle primitive per associare del codice ad un nome. In altre parole, dobbiamo andare a vedere cosa succede sotto il cofano di `defun`.

Senza entrare troppo nei dettagli, si può dire che la primitiva che serve a recuperare la *funzione* associata ad un nome è `fdefinition`.

```
CL prompt> (fdefinition 'first)
#<Function FIRST>
```

Questa primitiva piò essere usata con l'operatore di assegnamento `setf` per associare (ovvero, *assegnare*) una funzione ad un nome.

```
CL prompt> (setf (fdefinition 'plus42) (lambda (x) (+ x 42)))
#<Anonymous function>

CL prompt> (plus42 3)
45
```

Con questo meccanismo il "miracolo" in `process-method` diventa molto semplice da realizzare.
Si noti che non dovrebbe importare quante volte si "definisce" un metodo. Il codice di base di un metodo dovrebbe sempre essere lo stesso (più o meno una decina di righe ben formattate).

Domanda: perché bisigna usare `fdefinition` invece di richiamare direttamente `defun`?

Attenzione: il linguaggio **OOΛ′** è senz'altro divertente ma ha molti problemi semantici. Sono tutti noti! Nei test NON si analizzerà il comportamento del codice in casi patologici, che non rappresentano l'obiettivo del progetto.

4 Da consegnare...

- Uno `.zip` o `.tgz` file dal nome `<Cognome>-<Nome>-<matricola>-ool.zip` che contenga una cartella dal nome `<Cognome>-<Nome>-<matricola>-oolp`.
- Nella cartella dovete avere una sottocartella di nome `Lisp`.
- Nella directory `Lisp` dovete avere:
  - un file dal nome `ool.lisp` che contiene il codice di `define-class`, `new`, e `get-slot`. Inoltre dovete avere il codice che realizza la definizione dei metodi.
    - Le prime linee del file **devono essere dei commenti con il seguente formato**, ovvero devono fornire le necessarie informazioni secondo le regole sulla collaborazione pubblicate su Moodle.

      ```
      ;;;; <Cognome> <Nome> <Matricola>
      ;;;; <eventuali collaborazioni>
      ```

      Il contenuto del file deve essere ben commentato.
  - Un file `README` in cui si spiega come si possono usare i predicati definiti nel programma.

ATTENZIONE! Consegnate solo dei files e directories con nomi fatti come spiegato. Niente spazi extra **e soprattutto niente `.rar` or `.7z` o `.tgz` – solo `.zip`**

4.1 Valutazione

Il programma verrà valutato sulla base di una serie di test standard. In particolare si valuterà la copertura e correttezza di `define-class`, `new`, `get-slot` e dei metodi.