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'estensione "object oriented" di Common Lisp, chiamata **OOΛ**, e di un'estensione "object oriented" di Prolog, chiamata **OOΠ**.

OOΛ è un linguaggio object-oriented con eredità *multipla*. 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.

3 OOΛ in Common Lisp

Le primitive di **OOΛ** sono essenzialmente quattro: `def-class`, `make`, `field` e `fields`.
Attenzione: nel seguito le parentesi quadre sono solo parte della grammatica.

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

   La sua sintassi è:

   ```
   '(' def-class <class-name> <parents> <part>* ')'
   ```

   dove `<class-name>` è un simbolo, `<parents>` è una *lista* (possibilmente vuota) di simboli, mentre `<part>` é un insieme di `field` (o *campo*) o è un insieme di definizioni di *metodo*:

   ```
   part   ::= '(' fields <field>* ')'
          |   '(' methods <method>* ')'
   field  ::= <field-name>
          |   '(' field <field-name> <value> [<field-type>] ')'
   ```

```
method ::= '(' <method-name> <arglist> <form>* ')'
field-type ::= T | <type>
```

dove `<field-name>` e `<method-name>` sono simboli, `<value>` è un'espressione costante autovalutante qualunque, `<arglist>` è una lista di parametri standard Common Lisp e `<form>` è una qualunque espressione Common Lisp. Ci sono due modi di definire un campo: nel caso semplice, `field` é sono un simbolo il cui valore iniziale é `NIL` e il cui tipo é `T`; nel secondo caso, si possono specificare un valore iniziale ed un tipo (opzionale) per il campo. Se `<field-type>` non é presente, in suo default é `T`; altrimenti deve essere il nome di una classe definita con `def-class` o un tipo numerico Common Lisp[1].

Se `<parent>` è `NIL`, allora la classe non ha genitori[2].

Notate che `def-class` è una funzione normale e quindi gli argomenti che appaiono nelle sue chiamate sono valutati regolarmente. Ne consegue che negli esempi sotto `<field-name>` e `<method-name>` sono spesso delle espressioni costituite da un simbolo quotato (o da una keyword).

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

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

   ```
   '(' make <class-name> [<field-name> <value>]* ')'
   ```

   dove `<class-name>` e `<field-name>` sono simboli, mentre `<value>` è un qualunque valore Common Lisp.

   Il valore ritornato da `make` è la nuova istanza di `<class-name>`. Naturalmente `make` deve controllare che i 'fields' siano stati definiti per la classe.

3. `is-class`: restituisce `T` se l'atomo passatogli è il nome di una classe. La sintassi è:

   ```
   '(' is-class <class-name> ')'
   ```

   dove `<class-name>` è un simbolo.

4. `is-instance`: restituisce `T` se l'oggetto passatogli è l'istanza di una classe. La sintassi è:

   ```
   '(' is-instance <value> [<class-name>]')'
   ```

   dove `<class-name>` è un simbolo o `T` (passato per default come parametro `&optional`) e `<value>` è un valore qualunque. Se `<class-name>` è `T` allora basta che `<value>` sia un'istanza qualunque, altrimenti deve essere un'istanza di una classe che ha `<class-name>` come superclasse.

5. `field`: estrae il valore di un campo da una classe. La sintassi è:

   ```
   '(' field <instance> <field-name> ')'
   ```

[1] Notate che qui potete finire molto velocemente in una serie di problemi non banali; i nostri test controllano solo casi semplici e molto intuitivi.

[2] Come in C++ e Common Lisp non esiste una classe particolare che fa da "radice" come `java.lang.Object` in Java.

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

6. `field*`: estrae il valore da una classe percorrendo una catena di attributi. La sintassi è:

```
'(' field* <instance> <field-name>+ ')'
```

dove `<instance>` è un'istanza di una classe e `<field-name>+` è una lista non vuota di simboli, che rappresentano attributi nei vari oggetti recuperati. Il risultato è il valore associato all'ultimo elemento di `<field-name>+` nell'ultima istanza (tale valore potrebbe anche essere ereditato dalla classe o da uno dei suoi antenati). Se uno degli elementi di `<field-name>+` non esiste nella classe dell'istanza, la funzione segnala un errore.
NB. Vale l'equivalenza:

```
CL prompt> (eq (field (field (field I 's1) 's2) 's3)
               (field* I 's1 's2 's3))
T
```

3.1 Esempi

Creiamo una classe `person`

```
CL prompt> (def-class 'person nil '(fields (name "Eve") (age 21 integer)))
PERSON
```

Ora creiamo una sottoclasse `student`

```
CL prompt> (def-class 'student '(person)
                      '(fields
                         (name "Eva Lu Ator")
                         (university "Berkeley" string))
                      '(methods
                         (talk (&optional (out *standard-output*))
                             (format out "My name is ~A~%My age is ~D~%"
                                     (field this 'name)
                                     (field this 'age)))))
STUDENT
```

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

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

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

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

CL prompt> (defparameter s2 (make 'student))
```

```
S2

CL prompt> (defparameter s3 (make 'student 'age "42"))
Error: value "42" for field AGE is not of type INTEGER.
```

...e possiamo anche ispezionarne i contenuti.

```
CL prompt> (field eve 'age)
21

CL prompt> (field s1 'age)
108

CL prompt> (field s2 'name)
"Eva Lu Ator"

CL prompt> (field eve 'address)
Error: unknown field.
```

Un esempio più "complesso":

```
CL prompt> (def-class 'p-complex nil
                      '(fields
                        (:phi 0.0 real)
                        (:rho 1.0 real))
                      '(methods
                        (sum (pcn)
                             (let ((r1   (field this :rho))
                                   (phi1 (field this :phi))
                                   (r2   (field pcn :rho))
                                   (phi2 (field pcn :phi))
                                   )
                               ;; Do the math!
                               (make 'p-complex #| fix :rho and :phi |#)))
                        (mult (pcn) #|...|#)
                        (div (pcn) #|...|#)
                        (as-complex ()
                           (complex #| fix realpart and imagpart |#))
                       ))
P-COMPLEX ;; Complex numbers in polar format.

CL prompt> (defparameter pc1 (make 'p-complex :phi (/ pi 2) :rho 42.0))
PC1

CL prompt> (mult pc1 (make 'p-complex :rho 1.0 :phi pi))
;; the printed representation of the multiplication of PC1 with the new complex number.
```

Come potete vedere, le *keywords* del CL funzionano perfettamente come "field names"; ma non usate le keywords come nomi di metodi.
Nell'esempio precedente `PC1` $= \rho(\cos\phi + i\sin\phi)$ e la moltiplicazione di due numeri complessi in forma polare ha la rappresentazione *rettangolare*:

$$\rho_{\texttt{pc1}}\rho_{\text{new}}(\cos(\phi_{\texttt{pc1}} + \phi_{\text{new}}) + i\sin(\phi_{\texttt{pc1}} + \phi_{\text{new}})).$$

3.2 "Metodi"

Un linguaggio ad oggetti fornisce la nozione di *metodo*, ovvero di una funzione in grado di eseguire codice associato[3] alla "classe". Ad esempio, considerate la classe `student` definita sopra.

Domanda: come invochiamo il metodo `talk`?

In Java, C++, Python e Ruby, una volta definita la variabile `s1`, in il metodo verrebbe invocato come `s1.talk()`[4]. 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 field named TALK found.
```

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

```
CL prompt> (def-class 'studente-bicocca '(student)
                      '(methods
                        (talk ()
                           (format t "Mi chiamo ~A, e studio alla Bicocca~%"
                                (field this 'name))))
                      '(fields (university "UNIMIB")))
STUDENTE-BICOCCA

CL prompt> (defparameter ernesto (make '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 "field") 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 `def-class` e la manipolazione dei metodi.

[3] Tralasciamo, per questo progetto le questioni di *incapsulamento* e *visibilità*, nonchè le questioni legate al polimorfismo, viste in Julia.

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

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

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

(defun 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[5] dalla forma

```
'(' oolinst <class> <field-value>* ')'
```

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

Per recuperare il contenuto di uno field in un'istanza, metodo o semplice valore[6] che sia, prima si guarda dentro all'istanza, se si trova un valore allora lo si ritorna, altrimenti si cerca tra le superclassi (se ce ne sono). Dato che potenzialmente ci sono N superclassi, con un'intero grafo di superclassi sopra, allora bisogna decidere come attraversare il grafo. Si veda la Domanda 4 qui sotto al proposito.

Naturalmente se non si trova alcun valore per lo field cercato (ovvero, non c'è lo field), si dovrà richiamare la funzione `error`.

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

Domanda 2. Ho bisogno di un `Object`?

Domanda 3. Cosa succede in questo caso?

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

Domanda 4. Supponiamo di avere queste classi:

```
(def-class person () '(fields (:age 42) (:name "Lilith")))

(def-class superhero '(person) '(fields (:age 4092)))

(def-class doctor '(person))

(def-class fictional-character '(person) '(fields (:age 60)))

(def-class time-lord '(doctor superhero fictional-character))
```

Definiamo ora un'istanza

```
(defparameter the-doctor (make 'time-lord :name "Dr. Who"))
```

Quale dovrà essere il risultato di

```
(field the-doctor :age)
```

[5] Vi sono alternative.
[6] Fa differenza?

?
NB. Ci sono almeno tre modi di rispondere a questa domanda. L'algoritmo che dovrete implementare dovrà recuperare il valore dello field `:age` attraversando il grafo delle superclassi in profondità partendo dalla prima super classe[7]. Nel caso qui sopra, la risposta dovrà quindi essere `42`.

Tipo associato ad un 'field'. Ogni volta che si associa un valore ad un campo (e.g., via `make`) bisogna controllare che il tipo del campo sia corretto, ovvero che l'oggetto sia di un tipo compatibile con il tipo dichiarato del campo. Il tipo del campo é esso stesso ereditato.

Una complicazione la si ha quando si definiscono delle sottoclassi. Il tipo di uno slot non puó essere piú "ampio" in una sottoclasse[8], quindi, al momento della creazione di una classe bisogna controllare che ció non accada.

```
CL-prompt> (def-class using-integers ()
               '(fields (x 42 integer)))
USING-INTEGERS

CL-prompt> (def-class using-reals (using-integers)
               '(fields (x 42.0 real)))
Error: type of slot 'X' is a supertype of inherited type.

CL-prompt> (make 'using-integers 'x "42")
Error: value "42" for field X is not of type INTEGER.
```

La funzione Common Lisp `subtypep` vi tornerà utile in questo caso.

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 `def-class` o della `make`).

```
(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 `'(' <arglist> <form>* ')'` (si veda la definizione di `def-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`.

[7] Ne consegue che l'ordine delle superclassi è importante.
[8] Note on **"variance".**

```
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: perchè bisogna 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 distolgono l'attenzione dal principale obiettivo didattico del progetto.

4 OOΠ in Prolog

Anche le primitive di **OOΠ** sono essenzialmente quattro: `def_class`, `make`, `field` e `fields`.

1. il predicato `def_class` definisce la struttura di una classe e la memorizza nella "base di conoscenza" di Prolog.

   La sua sintassi è:

   ```
   def_class '(' <class-name> ',' <parents> ')'
   def_class '(' <class-name> ',' <parents> ',' <parts> ')'
   ```

   dove `<class-name>` é un atomo (simbolo) e `<parents>` è una *lista* (possibilmente vuota) di atomi (simboli), e `<parts>` è una *lista* di termini siffatti:

   ```
   part   ::= <field> | <method>
   field  ::= field '(' <field-name>, <value> ')'
          |   field '(' <field-name>, <value>, <type> ')'
   method ::= method '(' <method-name>, <arglist> ',' <form> ')'
   ```

   dove `<field-name>` e `<method-name>` sono simboli, `<value>` è un oggetto qualunque, `<arglist>` è una *lista* di parametri standard Prolog e `<form>` è una qualunque congiunzione di predicati Prolog. Il simbolo `this` all'interno di `<form>` si riferisce all'istanza stessa.

   Si noti che le regole di unificazione Prolog si applicano a `<value>` ed al corpo del metodo.

   `def_class` modifica la base dati del sistema Prolog mediante `assert*`.

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

```
make '(' <instance-name> ','
        <class-name> ','
       '[' [ <field-name> '=' <value>
           [ ',' <field-name> '=' <value> ]* ]*
       ']'
    ')'
```

dove `<class-name>` e `<field-name>` sono simboli, mentre `<value>` è un qualunque termine Prolog.

Il comportamento di `make` cambia a seconda di che cosa è il primo argomento `<instance-name>`.

- Se `<instance-name>` è un simbolo o una variabile istanziata con un simbolo allora lo si associa, nella base dati, al termine che rappresenta la nuova istanza.
- Se `<instance-name>` è una variabile non istanziata allora questa viene unificata con il termine che rappresenta la nuova istanza.
- Altrimenti `<instance-name>` deve essere un termine che unifica con la nuova istanza appena creata.

`make` modifica la base dati del sistema Prolog mediante `assert*` avviene solo nel primo caso appena descritto; gli altri due casi servono essenzialmente per motivi di debugging.

Create sia il predicato `make/2` and il predicato `make/3`.

`make(foo, fooclass)` $\equiv$ `make(foo, fooclass, [])`

Il comportamento relativamente complesso di `make` è necessario per facilitare la programmazione di oggetti composti, ad esempio, strutture ad albero o grafo.

3. `is_class`: ha successo se l'atomo passatogli è il nome di una classe. La sintassi è:

```
is_class '(' <class-name> ')'
```

dove `<class-name>` è un simbolo.

4. `is_instance`: ha successo se l'oggetto passatogli è l'istanza di una classe. La sintassi è:

```
is_instance '(' <value> ')'
is_instance '(' <value> ',' <class-name> ')'
```

dove `<class-name>` è un simbolo e `<value>` è un valore qualunque. `is_instance/1` ha successo se `<value>` è un'istanza qualunque; `is_instance/2` ha successo se `<value>` è un'istanza di una classe che ha `<class-name>` come superclasse.

5. `inst`: recupera un'istanza dato il nome con cui è stata creata da `make`[9].

```
inst '(' <instance-name> ',' <instance> ')'
```

[9]Questo predicato è necessario in Prolog dato che, al contrario di Common Lisp non abbiamo "variabili".

dove `<instance-name>` è un simbolo e `<instance>` è un termine che rappresenta un'istanza (o, ovviamente, una variabile logica).

6. `field`: estrae il valore di un campo da una classe. La sintassi è:

   ```
   field '(' <instance> ',' <field-name> ',' <result> ')'
   ```

   dove `<instance>` è un'istanza di una classe e `<field-name>` è un simbolo. Il valore recuperato viene unificato con `<result>` ed è il valore associato a `<field-name>` nell'istanza (tale valore potrebbe anche essere ereditato dalla classe o da uno dei suoi antenati). Se `<field-name>` non esiste nella classe dell'istanza il predicato fallisce.

7. `fieldx`: estrae il valore da una classe percorrendo una catena di attributi. La sintassi è:

   ```
   fieldx '(' <instance> ',' <field-names> ',' <result> ')'
   ```

   dove `<instance>` è un'istanza di una classe e `<field-names>` è una lista non vuota di simboli, che rappresentano attributi nei vari oggetti recuperati. Il valore recuperato viene unificato con `<result>` ed è il valore associato all'ultimo elemento di `<field-names>` nell'ultima istanza (tale valore potrebbe anche essere ereditato dalla classe o da uno dei suoi antenati). Se uno degli elementi di `<field-names>` non esiste nella classe dell'istanza il predicato fallisce.
   NB. Vale l'equivalenza (e simili):

   ```
   ?- inst(instance_1, I1),
   |  field(I1, s1, V1),
   |  field(V2, s2, V2),
   |  field(V2, s3, R),
   |  fieldx(I1, [s1, s2, s3], R).
   ```

Per semplificare, dovrete anche implementare `make/2` che non prevede di modificare i valori degli field.

4.1 Esempi

Creiamo una classe `person`

```
?- def_class(person, [], [field(name, 'Eve'), field(age, 21, integer)]).
```

Ora creiamo una sottoclasse `student`

```
?- def_class(student, [person],
             [field(name, 'Eva Lu Ator'),
              field(university, 'Berkeley'),
              method(talk, [],
                           (write(My name is '),
                            field(this, name, N),
                            writeln(N),
                            write('My age is ')
                            field(this, age, A),
                            writeln(A)))]).
```

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

```
?- make(eve, person).
result...

?- make(adam, person, [name = 'Adam']).
result...

?- make(s1, student, [name = 'Eduardo De Filippo', age = 108]).
result...

?- make(s2, student).
result...

?- make(s3, student, [name = 'Harry Potter', age = "12"]).
fail          % Si puo' stampare un messaggio di errore prima di fallire.
```

...e possiamo anche ispezionarne i contenuti.

```
?- field(eve, age, A).
A = 21

?- field(s1, age, A).
A = 108

?- field(s2, name, N).
N = 'Eva Lu Ator'

?- inst(s2, Eva_the_student), field(Eva_the_student, name, N).
N = 'Eva Lu Ator'

?- field(eve, address, Address).
false
```

4.2 "Metodi"

I "metodi" in **OOΠ** sono normali predicati Prolog che mettono in relazione un'istanza ed una serie di altri argomenti. Anche in questo caso non ci preoccupiamo di *incapsulamento* e *visibilità*. Anche per **OOΠ** valgono le stesse domande poste per **OOΛ**. Considerate la classe `student` definita sopra.

Domanda: come invochiamo il metodo `talk`?

In **OOΠ** vogliamo poter valutare il *predicato* `talk` come nell'esempio qui sotto:

```
?- talk(s1).
My name is Eduardo De Filippo
My age is 108
result...
```

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

```
?- inst(eve, Eve), talk(Eve).
false
```

Come per **OOΛ**, in **OOΠ** il metodo deve essere verificabile correttamente su istanze di sotto-classi. Ad esempio:

```
?- def_class(studente_bicocca, [studente],
             [method(talk, [],
                          (write('Mi chiamo '),
                           field(this, name, N),
                           writeln(N),
                           writeln('e studio alla Bicocca.'))),

              method(to_string, [ResultingString],
                               (with_output_to(string(ResultingString),
                                              (field(this, name, N),
                                               field(this, university, U),
                                               format('#<~w Student ~w>',
                                                      [U, N])))))

              field(university, 'UNIMIB')]).
result...

?- make(ernesto, studente_bicocca, [name = 'Ernesto']).
true

?- inst(ernesto, E), talk(E). % O anche: ?- talk(ernesto).
Mi chiamo Ernesto
e studio alla Bicocca
true

?- to_string(ernesto, S).
S = "#<UNIMIB Student Ernesto>"
```

Anche in questo caso, il problema è *come definire automaticamente i predicato* `talk` *e* `to_string` *in modo che siano in grado di riconoscere le diverse istanze.* Notate come il predicato `to_string` abbia due argomenti ed il secondo sia usato come output (la semantica, dopotutto, è quella Prolog).

4.3 Suggerimenti ed Algoritmi

Il suggerimento più importante è di ragionare in termini di *base di dati*. Ogni classe ed ogni istanza può essere mappata su una serie di predicati interni. Ad esempio, un predicato potrebbe essere `field_value_in_class/3`. Altri predicati utili potrebbero essere `instance_of/2`, `superclass/2`

4.3.1 Come si recupera il valore di uno field in un'istanza

In Prolog potrebbe essere più semplice assicurarsi che ogni field sia direttamente associato all'istanza creata da `make`. Ciò significa che al momento di verificare un'istanziazione di `make` bisognerà preoccuparsi di ereditare tutti i valori ed i metodi. A questo proposito, si consiglia di studiare i predicati `findall/3` e/o `bagof/3`.

Tipi dei 'fields'. Anche in questo caso dovete controllare il tipo associato ai 'fields' nello stesso modo descritto per il Common Lisp. Dovrete peró implementare l'equivalente di `subtypep` almeno per i tipi numerici.

4.3.2 Come si "installano" i metodi

Per manipolare i metodi in Prolog bisogna seguire una serie di passi alternativi.

Innanzitutto, nel "corpo" di un metodo, ovvero nella congiunzione di letterali, l'atomo `this` non può essere lasciato immutato. Ad esso va sostituita una variabile logica. Quindi è necessario avere un predicato che possa sostituire un atomo con un altro termine (inclusa una variabile logica); questo predicato è concettualmente semplice, ma richiede attenzione nella sua implementazione.

Una volta processato il corpo del predicato/metodo, basterà fare una `assert*` appropriata per installare il predicato.

Attenzione: per trattare correttamente l'"invocazione" *dinamica* dei predicati/metodi sarà necessario asserire dei tests che precedono ogni chiamata vera e propria: il loro compito è di implementare correttamente l'algoritmo di selezione del predicato/metodo da eseguire. Inoltre, bisognerà accertarsi che *solo* il predicato/metodo corretto venga eseguito. A questo proposito, si suggerisce di studiare il predicato di sistema `clause/2` in congiunzione con il predicato `call`.

Attenzione: anche il linguaggio **OOΠ** è senz'altro divertente ma ha molti problemi semantici – più di **OOΛ**. Anche in questo caso sono tutti noti e nei tests NON si analizzerà il comportamento del codice in casi patologici, che non rappresentano l'obiettivo del progetto.