

PROGRAMMAZIONE II - a.a. 2019-20

Esercitazione Soluzione— 12 dicembre 2019

Esercizio 1

Si estenda il linguaggio didattico funzionale in modo da includere espressioni e valori di tipo **record**. Un *valore (di tipo) record* è un dato strutturato composto da un numero finito di coppie *nome-valore*, detti **campi**. Analogamente, una *espressione (di tipo) record* è composta da un numero finito di coppie *nome-valore*. La valutazione di una espressione record produce un valore record.

Un identificatore può esser legato a un valore record tramite il costrutto **let**: nel seguente esempio (in sintassi OCaml-like) si evidenzia che l'espressione record che compare nel **let** è valutata in un valore record

```
# let rect = record{base = 5 * 5, altezza = 10 - 6}
val rect = record{base = 25, altezza = 4}
```

Sui record è definita l'operazione di selezione di una componente. Continuando l'esempio precedente

```
# let b = rect.base
val b = 25
```

1. Si estenda la sintassi astratta del linguaggio didattico funzionale in modo da includere valori ed espressioni record e l'operatore di selezione.

Prima soluzione con lista di coppie

```
type exp = ...
  | Rec of (ide * exp) list
  | Select of exp * ide

type evT = ...
  | RecVal of (ide * evT) list
  ...
```

```
let rec eval (e : exp) (r : evT env) : evT = match e with
  ...
  Rec(lst) -> RecVal(evalList lst r) |
  ...
  Select(rd, id) ->
    (match (eval rd r) in
      RecVal(rdPairs) -> lookup id rdPairs |
      _ -> failwith("wrong select value")) |
  ...

and evalList (lst : (ide * exp) list) (r : evT env) : (ide * evT) list = match lst with
  [ ] -> [ ] |
  (id, arg) :: rest -> (id, eval arg r) :: evalList rest r |
and lookup (id : ide) (lst : (ide * evT) list) : evT = match lst with
  [ ] -> Unbound |
  (id1, val):: rest -> if (id = id1) then val else (lookup id rest);;
```

Seconda soluzione con funzione

```
type exp = ...
  | Rec of (ide * exp) list
  | Select of exp * ide

type evT = ...
  | RecVal of (evT env)
  ...
```

```

let rec eval (e : exp) (r : evT env) : evT = match e with
  ...
  Rec(lst) -> RecVal(evalEnv lst r) |
  ...
  Select(rd, id) ->
    (match (eval rd r) in
      RecVal(r1) -> (applyenv r1 id) |
      _ -> failwith("wrong select value")) |
  ...
and evalEnv (lst : (ide * exp) list) (r : evT env) : (evT env) = match lst with
  [ ] -> (emptyEnv Unbound) |
  (id, arg) :: rest -> (bind id (eval arg r) (evalEnv rest r));;

```

Esercizio 2

Si estenda il linguaggio didattico funzionale introducendo il tipo di dato `IntSet` che permette di dichiarare insieme di interi di cardinalità finita. In aggiunta, il linguaggio è esteso con le operazioni primitive `insert myset elem` e `remove myset elem` che permettono di operare su insiemi finiti di interi.

1. Si mostri come deve essere modificato l'interprete del linguaggio didattico funzionale.

Si presenta una soluzione minimale: `insert` aggiunge sempre un intero alla lista, mentre `remove` ne rimuove tutte le occorrenze.

```

type exp = ...
  | IntSet of seq | insert of exp * exp | remove of exp * exp
  ...
and seq = Empty | Item of exp * seq

type evT = ...
  | IntSetVal of int list

let rec eval (e : exp) (r : evT env) : evT = match e with
  ...
  | IntSet e1 -> IntSetVal (evalSeq e1 r)
  | insert e1 e2 = (match (eval e2 r, eval e1 r) with
    IntSetVal s, Int i -> IntSetVal (i :: s)
    | _, _ -> failwith("IntSet error") )
  | remove e1 e2 = (match (eval e2 r, eval e1 r) with
    IntSetVal s, Int i -> IntSetVal (removeSeq s i)
    | _, _ -> failwith("IntSet error") )
  ...
and let rec evalSeq (s : seq) (r : evT env) : int list = match s with
  Empty -> []
  | Item (e1, s1) -> (eval e1 r)::(evalSeq s1 r)
  | _ -> failwith("IntSet error")
and let rec removeSeq (s : int list) (i: int) : int list = match s with
  [] -> []
  | i1 :: s1 -> if (i = i1) then (removeSeq s1 i) else i1 :: (removeSeq s1 i)

```

Esercizio 3

Si consideri il seguente programma OCaml

```

let z =1;;

let f1 = fun x y -> x + y *z;;

let rec apply_n_times f n x =
  if n<=0 then x
  else apply_n_times f (n-1) (f x);;

let rec map_n_times g n = function
| [] -> []
| h1::ls -> (apply_n_times g n h1) :: map_n_times g n ls;;

let z = 2;;

let ff = f1 1;;

map_n_times ff z [10;20];;

```

1. Si simuli la valutazione del programma mostrando la struttura della pila dei record di attivazione.
2. Si determini il valore calcolato dal programma. *Il valore calcolato è [12;22]*

Esercizio 4

Si consideri il seguente programma scritto in una notazione Java-like.

```

class A {
  private int a = 1;
  public int m1(int x) { return a + x + 1; }
  public int m2(int x) { return a + x + 2; }
}

class B extends A {
  private int b = 2;
  public int m1(int x) { return super.m1(x) + b; }
}

class C extends B {
  private int c = 3;
  public int m2(int x) { return m1(x) + c; }
}

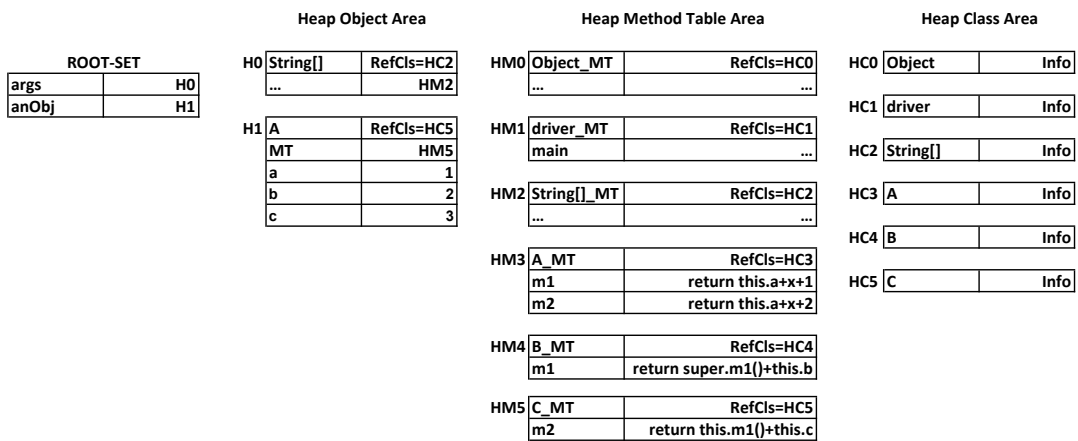
class driver {
  public static void main (String args[]) {
    A anObj = null;
    if (args[0] == "uno") anObj = new B();
    else anObj = new C();
    System.out.println(anObj.m1(5) + anObj.m2(5)); //(1)
  }
}

```

1. Si descriva l'ordine di caricamento delle classi durante l'esecuzione del comando `java driver "due"`.

```
Object driver String[] A B C
```

2. Si simuli la struttura del runtime quando l'esecuzione dell'istruzione (1) termina. In particolare si descriva la struttura degli oggetti sullo heap e la struttura delle tabelle dei metodi delle classi.



3. Supponendo di avere ereditarietà multipla e di estendere il programma con la seguente dichiarazione

```
class D extends B, C {
    private int d = 4;
    public int m3(int x) { return x * c; }
}
```

si descriva, motivando la risposta, la struttura della tabella dei metodi di un oggetto di tipo D.