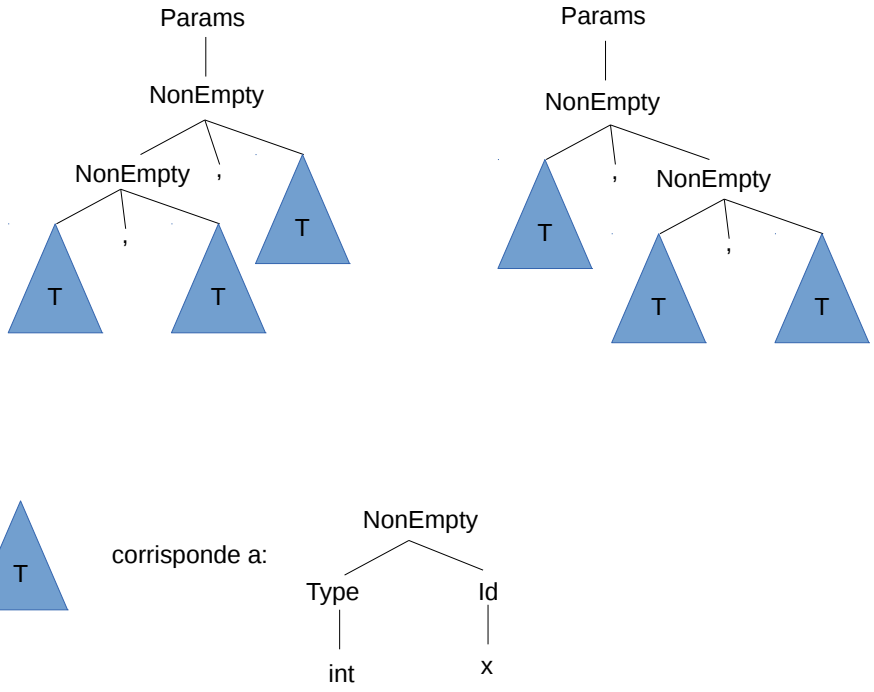
# Linguaggi e Programmazione Orientata agli Oggetti

## Soluzioni della prova scritta del 6 febbraio

a.a. 2013/2014

1. (a)
```
WEB_URL_RE = "[hH][tT][tT][pP](?<SSL>[sS])?://[a-zA-Z.]+(:(?<PORT>[0-9]+))?(/[a-zA-Z./]*)?";
```

   (b) Esistono due diversi alberi di derivazione per la stringa `int x, int x, int x`







   (c)
```
Params  ::=  Empty | NonEmpty
Empty ::= ϵ
NonEmpty ::= Param  | Param , NonEmpty
Param ::= Type Id
Type := int | char | boolean
Id ::= x | y | z
```

2. (a)
```
let rec explode = function
    [] -> ([],[])
  | (h1,h2)::tl -> let (t1,t2)= explode tl in (h1::t1,h2::t2);;
```

   (b)
```
let explode_accum l =
  let rec aux (l1,l2) = function
      [] -> (l1,l2)
    | (h1,h2)::tl -> aux (h1::l1,h2::l2) tl
  in let (l1,l2) = aux ([],[]) l
  in (List.rev l1,List.rev l2);;
```

   (c)
```
let explode_itlist l =
  it_list (function (a1,a2) -> function (e1,e2) -> (a1@[e1],a2@[e2])) ([],[]) l;;
```

   Nota: qui `l`, a sinistra e a destra dell'uguale, è brutto, ma serve per ragioni tecniche (ovvero, aggirare la value-restriction, di cui abbiamo solo accennato l'esistenza a lezione). Siccoma la questione è molto tecnica e fuori dal programma di LPO, nella correzione dello scritto non si è tenuto conto di questa restrizione e la presenza/assenza del parametro esplicito è stata ignorata.

3. Vedere il file `soluzioneEse3.jar`.

4. (a) `p1.m(null)`: `p1` ha tipo statico `P`, tutti i metodi in `P` sono accessibili e poiché il tipo di `null` è compatibile con ogni tipo reference, tutti e tre i metodi sono applicabili per sottotipo (in questo caso il metodo di arità variabile ha segnatura `m(Exp[])`). I tipi `Exp` e `Exp[]` sono entrambi più specifici di `Object`, ma nessuno dei due è più specifico dell'altro, quindi la chiamata è ambigua e viene segnalato un errore dal compilatore.

   (b) `(String) p1.m()`: `p1` ha tipo statico `P`, tutti i metodi in `P` sono accessibili e l'unico applicabile è quello di arità variabile (gli altri hanno arità 1), quindi la chiamata ha tipo statico `Object` e il cast a `String` è staticamente corretto poiché `String` $\leq$ `Object` (narrowing di un tipo reference) e tutta l'espressione ha tipo statico `String`.

   A tempo di esecuzione `p1` contiene un'istanza di `P`, quindi viene eseguito il corpo del metodo `m(Exp... exps)` in `P`. Viene restituita la stringa `"P.m(Exp...)"`, il cast ha successo, viene invocato il metodo `println(String)` di `java.io.PrintStream` che stampa la stringa `"P.m(Exp...)"` sullo standard output.

   (c) `h.m(e)`: le variabili `h` ed `e` hanno rispettivamente tipo statico `H` ed `EmptyStringExp`; tutti i metodi di `H` sono accessibili e quelli applicabili per sottotipo sono `m(Object)` (ereditato da `P`), `m(Exp)` e `m(AbstractExp)`. Poiché `AbstractExp` $\leq$ `Exp` $\leq$ `Object`, il metodo più specifico è `m(AbstractExp)` e quindi la chiamata è staticamente corretta.

   A tempo di esecuzione `h` contiene un'istanza di `H`, quindi viene eseguito il corpo del metodo `m(AbstractExp)` in `H`. La chiamata `super.m(e)` viene risolta staticamente selezionando il metodo `m(Exp)` in `P` (che è il più specifico tra quelli applicabili per sottotipo). Viene restituita la stringa `"P.m(Exp)"`, il cast a `String` ha successo, la stringa viene concatenata con `" H.m(AbstractExp)"` e, infine, il metodo `println(String)` di `java.io.PrintStream` stampa sullo standard output la stringa `"P.m(Exp) H.m(AbstractExp)"`.

   (d) `h.m(e, new MatchEmptyString())`: le variabili `h` ed `e` hanno rispettivamente tipo statico `H` ed `EmptyStringExp`, mentre l'espressione `new MatchEmptyString()` ha tipo statico `MatchEmptyString`; tutti i metodi di `H` sono accessibili, ma solo `m(Exp,Visitor<T>)` è applicabile per sottotipo (`EmptyStringExp` $\leq$ `Exp` e `MatchEmptyString` $\leq$ `Visitor<T>` per `T` $=$ `Boolean`)

   A tempo di esecuzione `h` contiene un'istanza di `H`, quindi viene eseguito il corpo del metodo `m(Exp,Visitor<T>)` in `H`. La chiamata `super.m(e)` viene gestita sia staticamente, sia dinamicamente, nello stesso modo del punto precedente, quindi viene restituita la stringa `"P.m(Exp)"`, il cast a `String` ha successo, la stringa viene concatenata con `" <T> H.m(Exp,Visitor<T>)"` e, infine, il metodo `println(String)` di `java.io.PrintStream` stampa sullo standard output la stringa `"P.m(Exp) <T> H.m(Exp,Visitor<T>)"`.

   (e) `p2.m(e)`: le variabili `p2` ed `e` hanno rispettivamente tipo statico `P` ed `EmptyStringExp`; tutti i metodi di `P` sono accessibili, e quelli applicabili per sottotipo sono `m(Object)` e `m(Exp)`. Poiché `Exp` $\leq$ `Object`, il metodo più specifico è `m(Exp)` e quindi la chiamata è staticamente corretta.

   A tempo di esecuzione `p2` contiene un'istanza di `H`, quindi viene eseguito il corpo del metodo `m(Exp)` in `H`. La chiamata `super.m(e)` viene risolta staticamente selezionando il metodo `m(Exp)` in `P` (che è il più specifico tra quelli applicabili per sottotipo). Viene restituita la stringa `"P.m(Exp)"`, il cast a `String` ha successo, la stringa viene concatenata con `" H.m(Exp)"` e, infine, il metodo `println(String)` di `java.io.PrintStream` stampa sullo standard output la stringa `"P.m(Exp) H.m(Exp)"`.

   (f) `p2.m(e2)`: le variabili `p2` ed `e2` hanno rispettivamente tipo statico `P` ed `Exp`; tutti i metodi di `P` sono accessibili, e quelli applicabili per sottotipo sono `m(Object)` e `m(Exp)`. Poiché `Exp` $\leq$ `Object`, il metodo più specifico è `m(Exp)` e quindi la chiamata è staticamente corretta.

   A tempo di esecuzione `p2` contiene un'istanza di `H` quindi viene eseguito il corpo del metodo `m(Exp)` in `H` e il risultato è lo stesso del punto precedente: viene stampa sullo standard output la stringa `"P.m(Exp) H.m(Exp)"`.