**A.1**
*Scrivere in Scheme una funzione Media che, data una lista di numeri naturali, calcola la media aritmetica dei numeri pari nella lista.*
*Ad esempio, se in input viene passata la lista (6 3 5 8 9 64 45 32 4 77 89), in output viene restituito il risultato di (6 + 8 + 64 + 32 + 4)/5.*

The isEven function returns TRUE if the number is even, FALSE otherwise:

```
(define isEven(lambda (number)
        (cond ((eq? number 0) #t)
        ((eq? number 1) #f)
        (#t (isEven (- number 2))))
))
```

The getEvens function extracts a list of even numbers and returns it in output:

```
(define getEvens (lambda (list)
        (cond ( (null? list) '() )
        ((isEven (car list)) (cons(car list) ( getEvens (cdr list))))
        (#t ( getEvens (cdr list))))
))
```

The sum function sums all the even numbers given in input:

```
(define sum (lambda (list)
        (cond ((null? list) 0)
        (#t (+ (car list) (sum(cdr list)))))
))
```

The count function counts the elements of the given input list:

```
(define count (lambda (list)
        (cond ((null? list) 0)
        (#t (+ 1 (count(cdr list)))))
))
```

The check function returns the first even element of the list or an empty list:

```
(define check (lambda (list)
        (cond ((null? list) '())
        ((isEven (car list)) (car list))
        (#t (check (cdr list))))
))

(define Media(lambda (list)
        (cond ((null? list) 0)
        ((null? (check list)) 0)
        (#t (/ (sum(getEvens list)) (getEvens list)))))
))
```

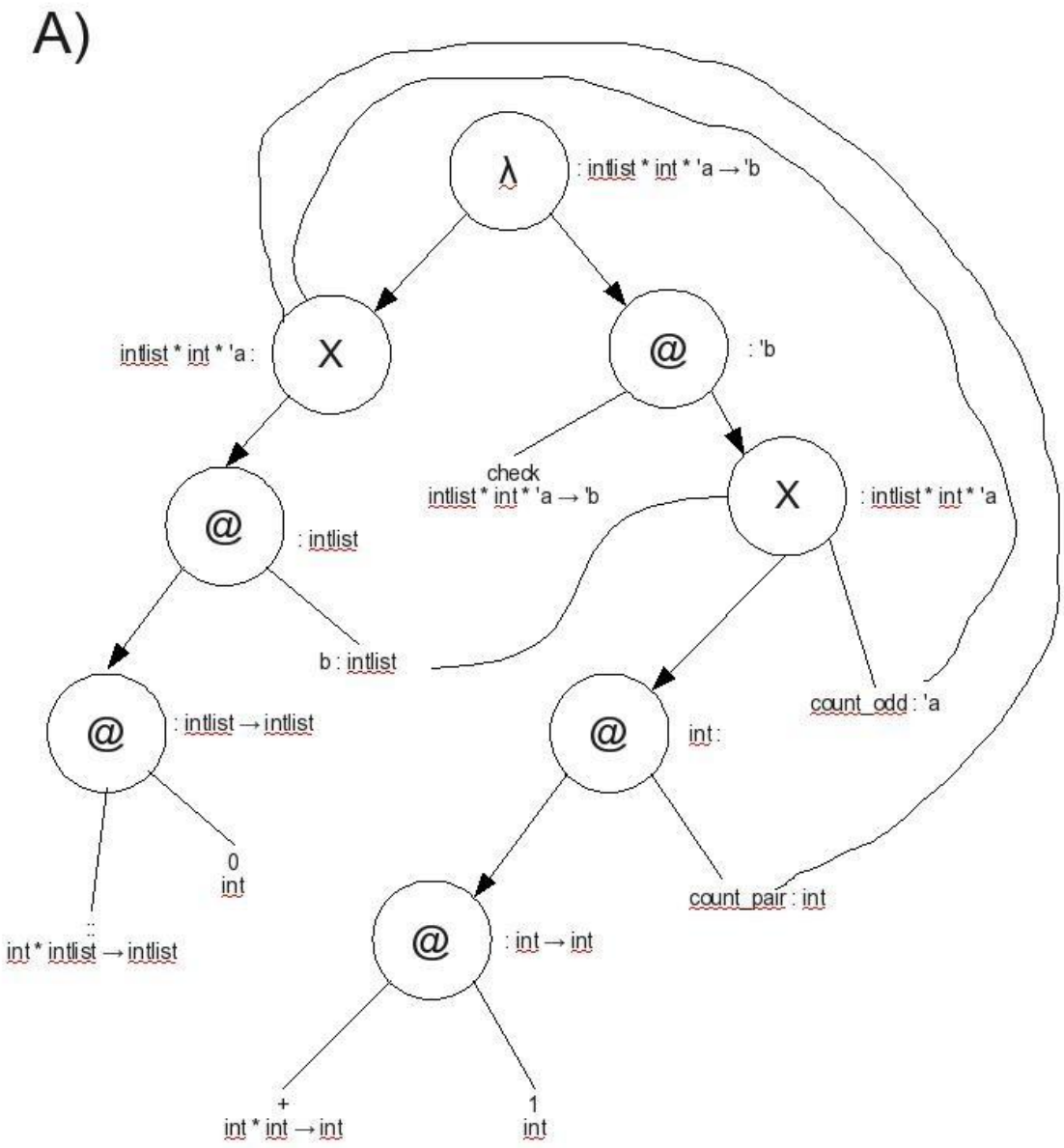
**A.2**
*Scrivere in ML una funzione Pari che, data una lista di numeri naturali, ritorna true se nella lista ci sono più numeri pari che numeri dispari; ritorna false altrimenti.*
*Ad esempio, se in input viene passata la lista (6 3 5 8 9 64 45 32 4 77 89), in output viene restituito false.*

```
fun     check (0::b, count_pair, count_odd) = check(b, count_pair+1, count_odd) |
        check (1::b, count_pair, count_odd) = check(b, count_pair, count_odd+1) |
        check (a::b, count_pair, count_odd) = check(a-2::b, count_pair, count_odd) |
        check (nil, count_pair, count_odd) = count_pair > count_odd
;

fun Pari(l) = check(l, 0, 0);

val check = fn : int list * int * int -> bool

val Pari = fn : int list -> bool
```

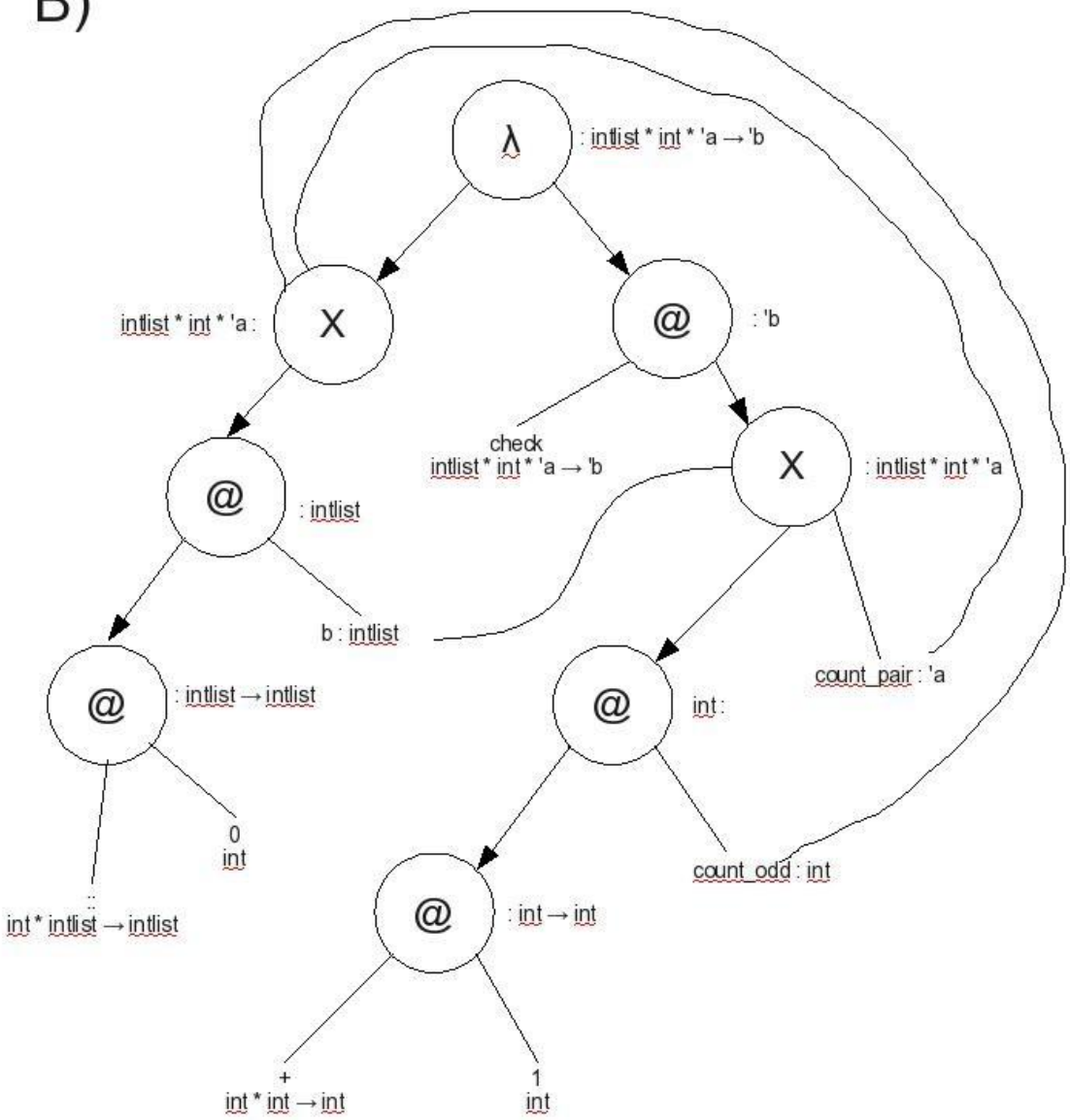
**A.3**

*Mostrare i syntax tree costruiti dall'algoritmo di inferenza dei tipi di ML per ognuna delle funzioni (ovvero per la funzione Pari e per tutte le eventuali funzioni ausiliarie) scritte per svolgere l'esercizio 2.*

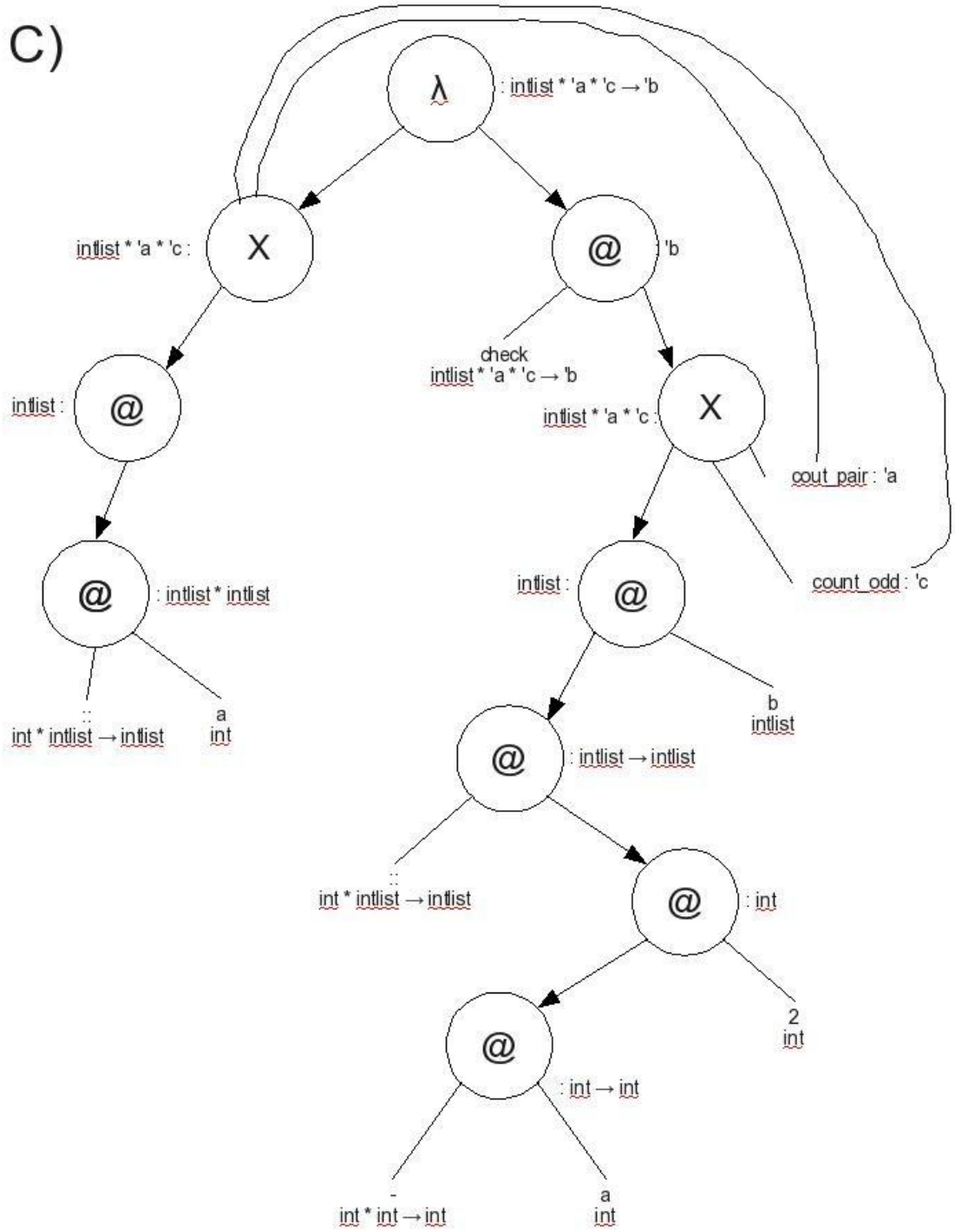
*A) fun check (0::b, count_pair, count_odd) = check(b, count_pair+1, count_odd)*
*B) check (1::b, count_pair, count_odd) = check(b, count_pair, count_odd+1)*
*C) check (a::b, count_pair, count_odd) = check(a-2::b, count_pair, count_odd)*
*D) check (nil, count_pair, count_odd) = count_pair > count_odd;*
*E) fun Pari(l) = check(l, 0, 0);*



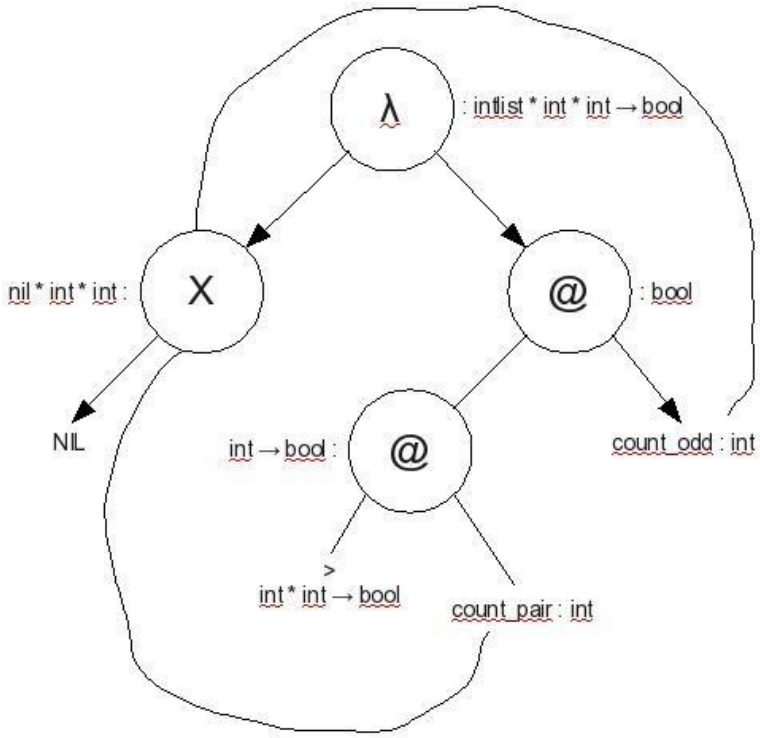
B)


λ
: intlist * int * 'a → 'b
intlist * int * 'a :
X
@
: 'b
check
intlist * int * 'a → 'b
@
: intlist
X
: intlist * int * 'a
b : intlist
@
: intlist → intlist
@
int :
count_pair : 'a
0
int
count_odd : int
::
int * intlist → intlist
@
: int → int
+
int * int → int
1
int

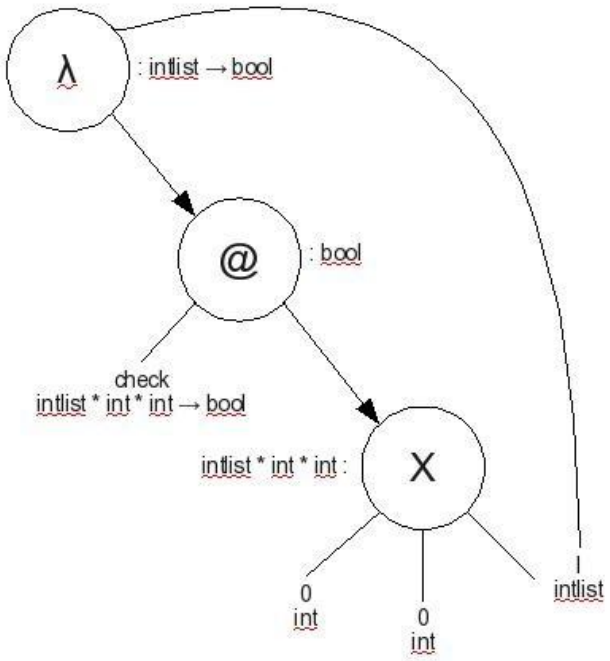
C)


λ
: intlist * 'a * 'c → 'b
intlist * 'a * 'c :
X
@
'b
check
intlist * 'a * 'c → 'b
intlist :
@
intlist * 'a * 'c :
X
cout_pair : 'a
@
: intlist * intlist
intlist :
@
count_odd : 'c
::
int * intlist → intlist
a
int
b
intlist
@
: intlist → intlist
::
int * intlist → intlist
@
: int
@
2
int
@
: int → int
-
int * int → int
a
int

**D)**


λ
: intlist * int * int → bool
nil * int * int :
X
@
: bool
NIL
int → bool :
@
count_odd : int
>
int * int → bool
count_pair : int

**E)**


λ
: intlist → bool
@
: bool
check
intlist * int * int → bool
intlist * int * int :
X
0
int
0
int
I
intlist

**A.4**

*Descrivere, in entrambi i casi di scope statico e dinamico, lo stack di run-time dopo la chiamata della funzione t dentro la chiamata della funzione f dentro la chiamata della funzione g.*

```
val z=5;
fun f(x) = z*x;
fun h(x) = z+2*x;
fun g(t,h) = let val z=f+3*z in f(t(h)*z) end;
g(h,2);
```

SCOPE STATICO

| | | |
|---|---|---|
| 1 | Control Link | 0 |
| | Access Link | 0 |
| | z | 5 |
| 2 | Control Link | 1 |
| | Access Link | 1 |
| | f | C1 |
| 3 | Control Link | 2 |
| | Access Link | 2 |
| | h | C2 |
| 4 | Control Link | 3 |
| | Access Link | 3 |
| | g | C3 |
| 5 | Control Link | 4 |
| | Access Link | 4 |
| | t | C2 |
| | h | 2 |
| | z | 17 |
| | Ris | 765 |
| 6 | Control Link | 5 |
| | Access Link | 2 |
| | x | 9*17 = 153 |
| | Ris | 765 |
| 7 | Control Link | 6 |
| | Access Link | 3 |
| | x | 2 |
| | Ris | 9 |
| C1 | 2 | Code f |
| C2 | 3 | Code h |
| C3 | 4 | Code g |

DINAMICO

| | | |
|---|---|---|
| 1 | Control Link | 0 |
| | Access Link | 0 |
| | z | 5 |
| 2 | Control Link | 1 |
| | Access Link | 1 |
| | f | C1 |
| 3 | Control Link | 2 |
| | Access Link | 2 |
| | h | C2 |
| 4 | Control Link | 3 |
| | Access Link | 3 |
| | g | C3 |
| 5 | Control Link | 4 |
| | Access Link | 4 |
| | t | C2 |
| | h | 2 |
| | z | 17 |
| | Ris | 6069 |
| 6 | Control Link | 5 |
| | Access Link | 5 |
| | x | 21*17 = 357 |
| | Ris | 6069 |
| 7 | Control Link | 6 |
| | Access Link | 6 |
| | x | 2 |
| | Ris | 4+17 = 21 |
| C1 | 2 | Code f |
| C2 | 3 | Code h |
| C3 | 4 | Code g |

**B.1**
*Quali sono, in generale, le differenze sostanziali tra la relazione di sotto-tipo e la relazione di ereditarietà? Java e C++ supportano entrambe le relazioni? Se si, ci sono differenze nel modo con cui queste relazioni sono supportate dai due linguaggi? Discutere brevemente e motivare la risposta.*

Le relazioni di sottotipo caratterizzano una entità (per esempio un Oggetto nei linguaggi OOP o una Classe nel linguaggio OWL) vincolando quest'ultima all'implementazione di specifiche proprietà (spesso interfacce) che ne espongono le funzionalità o ne definiscono gli attributi.
L'implementazione di questi vincoli secondo una logica compositiva-additiva, dove
{ Ra = {...}, Rb={...} e Va = Ra, Vb = Va U Rb } determina una relazione di sotto-tipizzazioni per cui l'entità che implementa Vb è sia Va che Vb vincolata.

Più astrattamente, sfruttando “Il Principio di Sostituzione di Liskov” (LSP):
Se per ogni oggetto o1 di tipo S esiste un oggetto o2 di tipo T tale che per ogni programma P definito in termini di T, il comportamento di P è immutato quando o2 è sostituito a o1, allora S è sottotipo di T.

Esempio puro di tale principio è il Duck Typing.

La relazione di ereditarietà è una estensione delle relazioni di sottotipo, specificamente progettata per permettere il riutilizzo delle implementazioni ed evitare quindi duplicazioni delle stesse. Alla sottotipizzazione quindi, aggiunge il principio DRY (Don't Repeat Yourself anche conosciuto come Single Point of Truth).

Il C++ ed il Java supportano entrambe le implementazioni ma, in C++, B è sottotipo di A solo se il primo eredita pubblicamente dal secondo.

**B.2**
*Quali forme di polimorfismo supporta il linguaggio C++? Discutere brevemente e motivare la risposta.*

Il C++ implementa due macro-forme di polimorfismo:
- parametrico, mediante l'uso dei Template o dell'overloading delle funzioni;
- ereditario, mediante l'uso delle funzioni virtuali (overriding).

I Template permettono la “programmazione generica”;
Le funzioni Virtuali permettono la specializzazione delle implementazioni e la loro conseguente invocazione dinamica (a runtime);

**B.3**
*Quali forme di polimorfismo supporta Java? Discutere brevemente e motivare la risposta.*

Il Java (si fa riferimento all'ultima versione attualmente disponibile, Java6 o Java 1.6) implementa le stesse forme di polimorfismo del C++ (si veda la risposta B.2) ma differentemente da quest'ultimo, l'overriding delle funzioni è implicitamente dinamico e quindi non è richiesto l'uso dello specificatore virtual (inesistente in Java).
Questo per semplificare il linguaggio e perchè Java non è stato progettato per fornire il livello di prestazioni richiesto dal C++.

**B.4**
*Una volta mostrato quale è l’output dei due programmi che seguono, si spieghi perché l’output del programma A è differente dall’output del programma B (si faccia attenzione al fatto che i due programmi differiscono per UN SOLO carattere).*

Output Programma A:
“””
This is a Mobile Phone...GSM Supported
This is a Mobile Phone...CDMA Supported
“””

Output Programma B:
“””
This is a common device...
This is a common device...
“””

Nel programma A la risoluzione del metodo “which” da invocare sull'istanza passata alla funzione “whichPhoneUserIsUsing” è effettuata dinamicamente a runtime poiché l'instanza è gestita per riferimento.

Nel programma B la risoluzione del metodo “which” da invocare sull'istanza passata alla funzione “whichPhoneUserIsUsing” è effettuata staticamente a compiletime e quindi indissolubilmente legata al tipo del parametro, ovvero “CommunicationDevices”.

**B.5**
*Considerando il linguaggio Java, si mostri (tramite scrittura di un frammento di codice Java) e si discuta un caso in cui l'utilizzo corretto a compile-time della relazione di sotto-tipo produce un errore a run-time.*

```
1> class A { }
2> class B extends A { }
3>
4> B[] Ab = new B[5];
5> A[] Aa = Ab;
6>
7> Aa[0] = new A();
```

La riga 7 genera un errore a runtime poiché la riga 5 modifica, da A a B, il tipo puntato da "Aa". Conseguentemente l'assegnamento di una istanza di tipo genitore ad una di tipo figlio genera una eccezione a tempo di esecuzione (il compilatore non è in grado di accorgersene ma la VM si).