1.1

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.

Funzione Pair, restituisce TRUE se il numero è pari, FALSE altrimenti.

```
(define Pair(lambda (n)
(cond   ((eq? n 0) #t)
        ((eq? n 1) #f)
        (#t (Pair (- n 2))))))
```

Funzione ExtractPair, estrae i numeri pari da una lista e li restituisce in output

```
(define ExtractPair (lambda (l)
(cond   ((null? l) '())
        ((Pair (car l)) (cons(car l) ( ExtractPair (cdr l))))
        (#t ( ExtractPair (cdr l))))))
```

Funzione Sum, funzione per effettuare la somma, in questo caso utilizzata per sommare tutti i numeri pari

```
(define Sum (lambda (l)
(cond   ((null? l) 0)
        (#t (+ (car l) (Sum(cdr l)))))))
```

Funzione Count per contare gli elementi di una lista

```
(define Count (lambda (l)
(cond   ((null? l) 0)
        (#t (+ 1 (Count(cdr l)))))))
```

Funzione Check che restituisce il primo numero pari della lista o lista vuota altrimenti

```
(define Check (lambda (l)
(cond   ((null? l) '())
        ((Pair (car l)) (car l))
        (#t (Check (cdr l))))))
```

Funzione AVG per il calcolo della media di una lista.
Se la lista è vuota restituisce 0, successivamente tramite la funzione Check controlla che esista almeno un numero pari nella lista, in caso contrario restituisce 0.
Questo controllo è stato inserito per evitare di ricevere l'errore “Division by 0” nel quale si incappava se la funzione Count restituiva 0 venendogli passata una lista di soli numeri dispari.

```
(define AVG(lambda (l)
(cond   ((null? l) 0)
        ((null? (Check l)) 0)
        (#t (/ (Sum(ExtractPair l)) (Count(ExtractPair l)))))))
```

Alcune esecuzioni di prova:

```
// Normale esecuzione
> (AVG '(1 2 3))
2

// Divisione reale
> (AVG '(1 2 6 6))
4.6

// Lista vuota
> (AVG '())
0

// Tutti numeri dispari
> (AVG '(1 3 5))
0
```

1.2

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 pair(l) = check(l, 0, 0);


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

Alcune esecuzioni:

```
// Dispari maggiori
- pair([1,2,3]);
val it = false : bool

// Pari maggiori
- pair([1,2,4]);
val it = true : bool

// Pari = Dispari
- pair([1,2]);
val it = false : bool

// Lista vuota
- pair([]);
val it = false : bool
```

1.3
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.

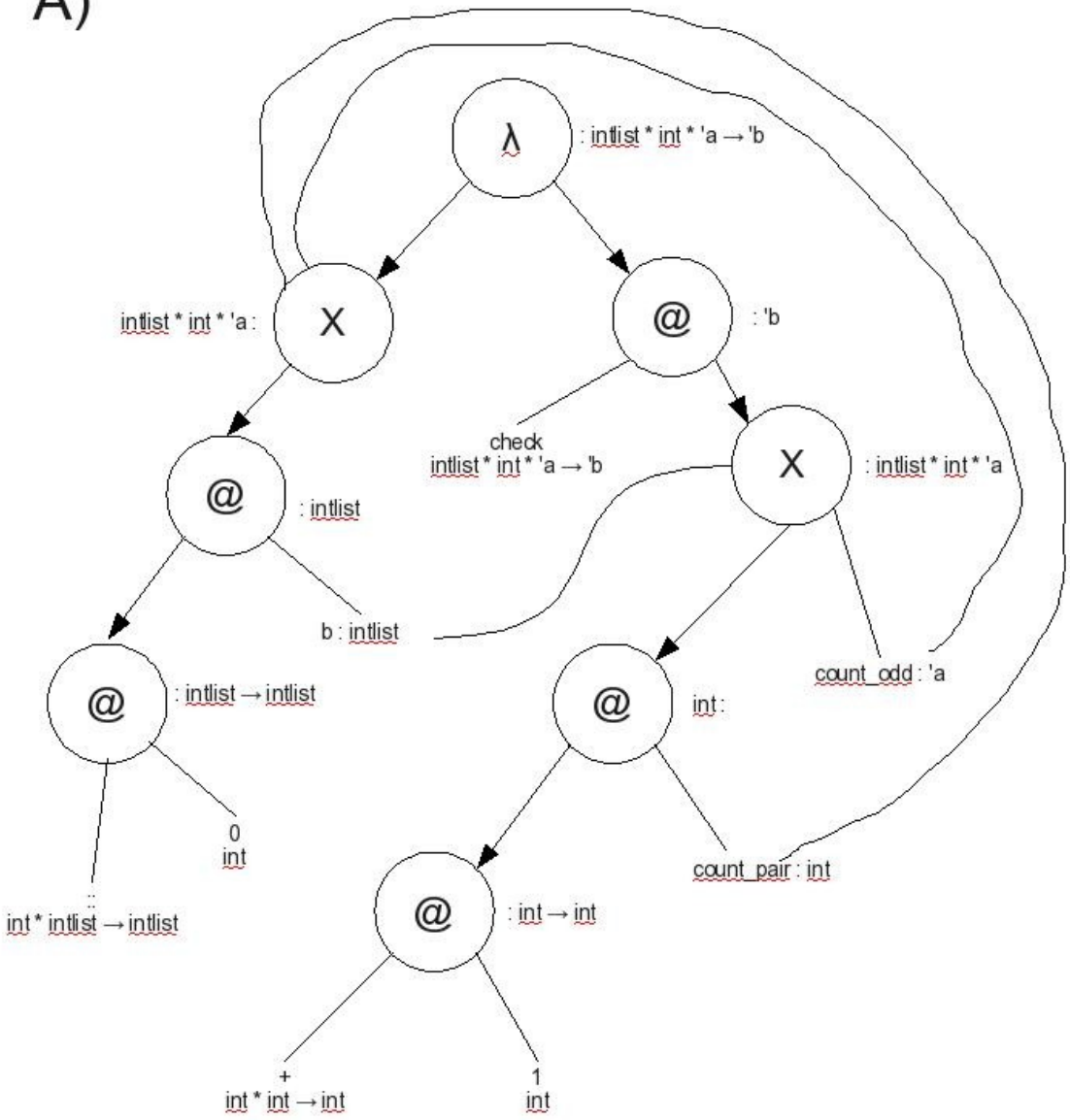
A fun check (0::b, count_pair, count_odd) = check(b, count_pair+1, count_odd)

B check (1::b, count_pair, count_odd) = check(b, count_pair, count_odd+1)

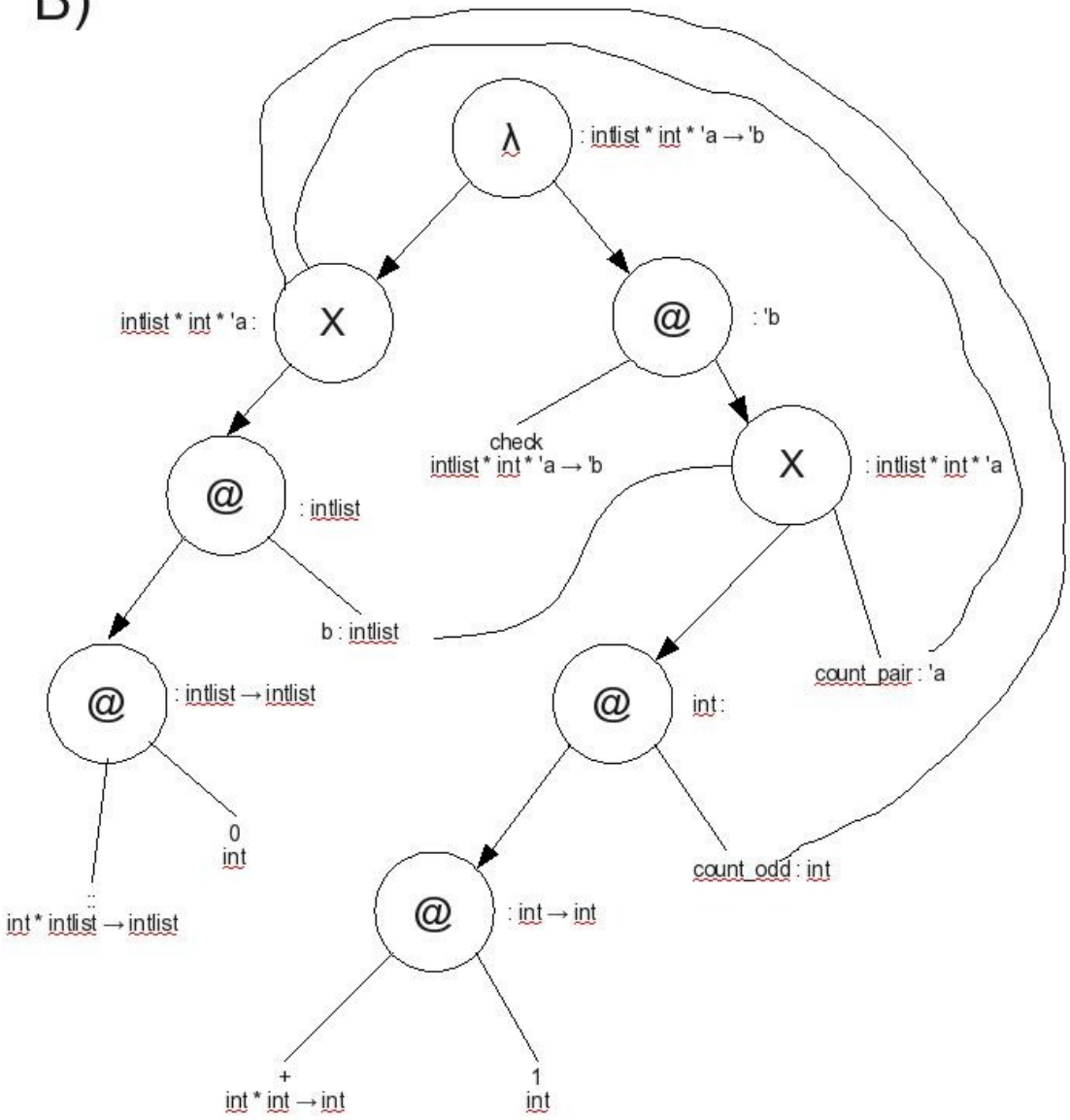
C check (a::b, count_pair, count_odd) = check(a-2::b, count_pair, count_odd)
D check (nil, count_pair, count_odd) = count_pair > count_odd;

E fun pair(l) = check(l, 0, 0);

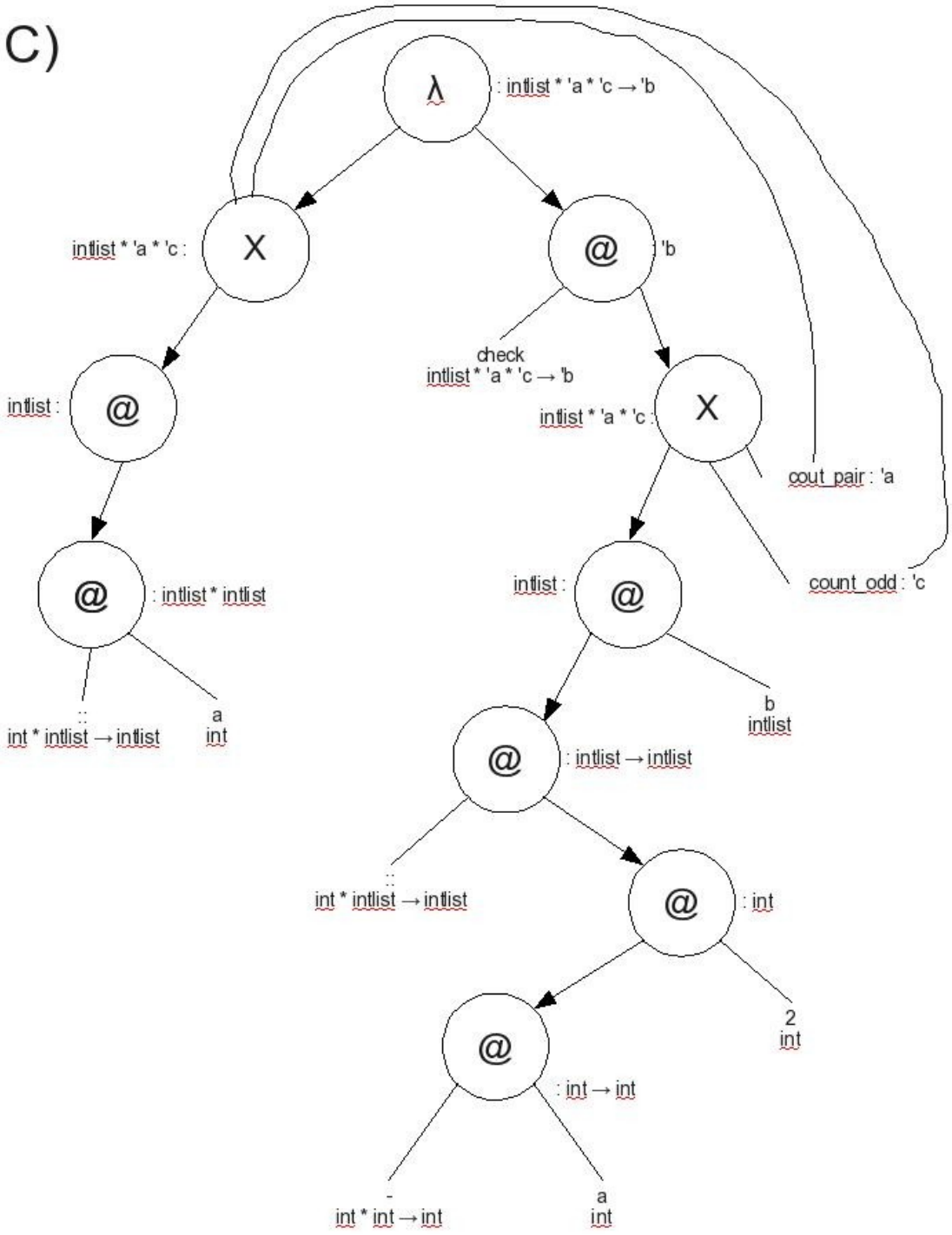
A)


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_odd : 'a
0
int
::
int * intlist → intlist
@
: int → int
count_pair : int
+
int * int → int
1
int

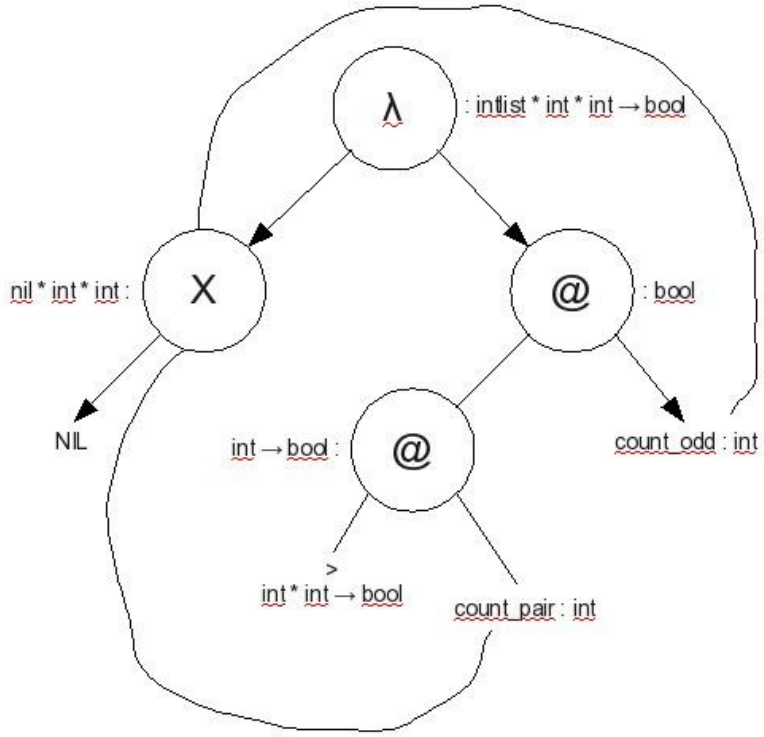
B)


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
::
int * intlist → intlist
@
: int → int
count_odd : int
+
int * int → int
1
int

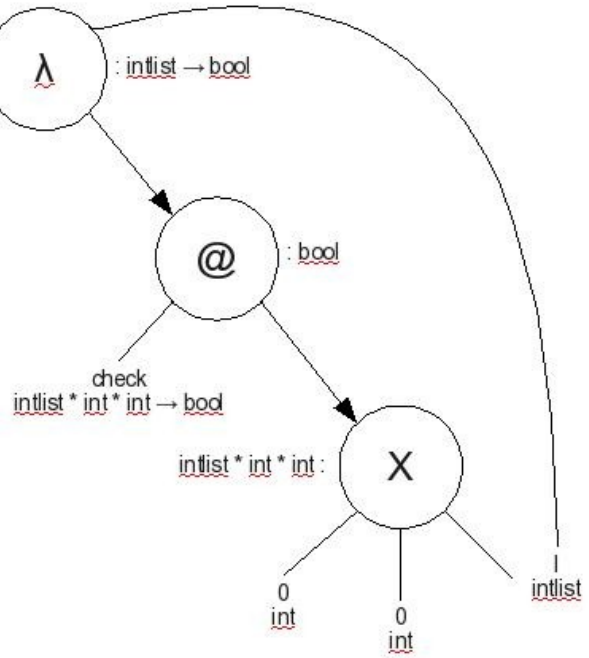
C)


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

1.4

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

SCOPE 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

2.1

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.

Il subtyping è una relazione su interfacce, invece l'ereditarietà è una relazione tra implementazioni.
Nello specifico essere un sottotipo significa che se un oggetto A ha tutte le funzionalità di un'altro oggetto B, noi possiamo usarlo (o meglio sostituirlo) al posto di B.
Il meccanismo alla base del subtyping è il principio della sostituzione, ovvero
“se A è un sottotipo di B, allora un'istanza di A può essere usata in qualsiasi contesto in cui il programma si aspetta di trovare un'istanza di B”.
Questo ci dà il vantaggio di poter effettuare a runtime operazioni uniformi su dati differenti.

L'ereditarietà permette di definire un nuovo oggetto sulla base di uno già esistente, permettendo il riutilizzo del codice “comune”, infatti la sottoclasse "eredita" implicitamente tutte le caratteristiche (attributi e operazioni) della classe base, che possono anche essere modificate.

La differenza sostanziale tra subtyping e ereditarietà è che la prima è una relazione su interfacce, la seconda su implementazioni.

C++ e Java supportano entrambe le implementazioni ma in C++ A è un sottotipo di B solo se eredita pubblicamente.

2.2

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.

In C++ abbiamo 2 tipi di polimorfismo quello parametrico con i template e quello dato dall'utilizzo dell'ereditarietà con le funzioni virtuali.

I template che sono un meccanismo di parametrizzazione dei tipi che ci permettono di chiamare una funzione passandogli dei tipi differenti ad ogni chiamata.
Questo è un ottimo esempio di “programmazione generica” e ci fornisce una migliore prestazione nell'esecuzione, ma purtroppo non di spazio in quanto a link time viene effettuata una copia del codice per ogni tipo al fine di allocare opportunamente spazio in memoria per i tipi passati.

L'ereditarietà pubblica attraverso le funzioni virtuali in C++ è un'altro tipo di polimorfismo.
L'anteposizione della parola virtual al nome di una funzione ci consente il dynamic lookup (capacità di scegliere un metodo relativamente ad un oggetto dinamicamente) a runtime.

Infine in C++ esiste anche l'overloading (detto anche polimorfismo ad hoc) che ci permette, a condizione che il numero e/o il tipo dei parametri siano differenti, di dichiarare funzioni con lo stesso nome.

2.3

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

In Java il polimorfismo ci permette di risolvere a runtime una chiamata a funzione.
La selezione della classe alla quale fà capo la funzione avviene dinamicamente a seconda del tipo della variabile.

Un'altro tipo di polimorfismo in java è dato dalle interfacce.
Queste permettono tramite la parola chiave “implements” di definire un “protocollo” senza scendere nei dettagli dell'implementazione (infatti un'interfaccia non permette l'implementazione dei corpi dei metodi).
Una classe che implementa un'interfaccia è obbligata a implementare tutti i metodi contenuti nell'interfaccia.

2.4

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 al momento della chiamata a funzione viene passato come parametro il riferimento all'oggetto, in questo modo la funzione verrà risolta in base al tipo dell'oggetto puntato da device.

Nel programma B invece durante il passaggio dei parametri è come se avvenisse un assegnamento del tipo ComunicationDevices devices = user1, risolvendo le chiamate a funzione sempre in modo statico col tipo ComunicationDevices.

2.5

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

```
class A { }
class B extends A { }

B[] arrayB = new B[5];
A[] arrayA = arrayB;

arrayA[0] = new A();
```

Questo codice genere un errore a runtime in quanto dopo l'assegnamento A[] arrayA = arrayB; l'array arrayA punterà ad oggetti di tipo B ai quali non si possono assegnare oggetti di tipo A perchè B è una sottoclasse di A e non il contrario.