Compilatore di Functional and Object Oriented Language

Functional and Object Oriented Language

- Sviluppo di un compilatore per il Functional and Object Oriented Language (FOOL).
- La sintassi del linguaggio è definita nel file FOOL.g4 allegato a queste slides.
- I metodi di visita utilizzabili nelle diverse fasi del compilatore sono dichiarati nel file BaseASTVisitor.java allegato a queste slides.

Functional and Object Oriented Language

- Il linguaggio è una estensione della versione di base sviluppata in laboratorio con:
  - operatori aggiuntivi "<=", ">=", "||", "&&", "/", "-" e "!", con stesso significato che hanno in C/Java;
  - gestione dell'object orientation.

Il compilatore

- Descritto in modo dettagliato come estensione del compilatore sviluppato in laboratorio per la versione di base del linguaggio.
- Il compilatore quindi produce codice per la Stack Virtual Machine (SVM) sviluppata a lezione: senza bisogno di modificarla!

Estensione Object Oriented

Commenti Preliminari su Estensione OO

- Descritta come estensione della versione base del linguaggio fatta in laboratorio
  - parti con sfondo rosa da considerare solo per sviluppo linguaggio FOOL con ereditarietà
- Si usa lo heap per allocare: gli oggetti e le dispatch tables (spiegate in precedenza)
- Lo heap viene allocato da indirizzi bassi verso indirizzi alti (registro $hp inizialmente è 0)
- Per semplicità non implementeremo deallocazione oggetti (es. garbage collector)

Elementi Sintattici Nuovi in FOOL.g4

- dichiarazioni (solo ambiente globale e all'inizio)

  class ID1 [extends ID2] (..campi dichiarati come parametri..) {
  .. metodi dichiarati come funzioni ..
  }

  dove extends ID2 è opzionale e ID2 è ID di una classe

- espressioni
  - ID1.ID2(..)
  - new ID(..) dove ID è ID di una classe
  - null
- tipi (tipo dei riferimenti)
  - ID dove ID è ID di una classe

Esempio: Dichiarazione Classe

```
let
  class A (a:int, b:bool) {
    fun n:int(...) ... ;
    fun m:bool(...) ... ;
  }
in
  ... ;
```

- Campi "a" e "b" dichiarati con sintassi di parametri
- Esempio creazione oggetto di classe A: new A(5,true)
  - costruisce un oggetto che ha campi a=5 e b=true
  - restituisce il riferimento (di tipo A) all'oggetto

- Oggetti, una volta creati, sono immutabili
  - campi non modificabili
- Campi accessibili (leggibili) solo da dentro la classe A (o da dentro una classe che eredita da A)
  - tramite il nome, es. a+5
- Metodi invocabili da dentro classe (che eredita da) A
  - come funzioni, es. n(...)

o anche dall'esterno

  - con notazione x.n(...) dove "x" contiene riferimento di tipo A

Esempio: Ereditarietà

```
let
  class A (a:int, b:bool) {
    fun n:int(...) ... ;
    fun m:bool(...) ... ;
  }
  class B extends A (c:int) {
    fun l:int(...) ... ;
  }
in
  ... ;
```

- Esempio creazione oggetto di classe B: new B(5,true,7)
  - costruisce un oggetto che ha campi a=5, b=true e c=7
  - restituisce il riferimento (di tipo B) all'oggetto

Esempio: Ereditarietà e Overriding di Campi

```
let
  class A (a:int, b:bool) {
    fun n:int(...) ... ;
    fun m:bool(...) ... ;
  }
  class B extends A (c:int, a:bool/* overriding */) {
    fun l:int(...) ... ;
  }
in
  ... ;
```

- L'overriding di campi modifica il tipo di un campo ma non estende l'elenco dei campi
- Es. creazione oggetto di classe B: new B(false,true,7)
  - costruisce un oggetto che ha campi a=false, b=true e c=7

Layouts

- layout oggetti in HEAP:

[PRIMA POSIZIONE LIBERA HEAP]	<- $hp subito dopo allocazione oggetto
dispatch pointer	[offset 0] <- object pointer
valore primo campo dichiarato	[offset -1]
.	
.	
valore ultimo (n-esimo) campo	[offset -n]

Layouts

- layout dispatch tables in HEAP:

```
[PRIMA POSIZIONE LIBERA HEAP]  <- $hp subito dopo allocazione tabella
addr ultimo (m-esimo) metodo   [offset m-1]
.
.
addr primo metodo dichiarato   [offset 0] <- dispatch pointer
```

Layouts

- layout degli AR (amb. globale/funzioni/metodi)
  - invariato
    - dichiarazioni classi in ambiente globale occupano lo spazio di un indirizzo: il dispatch pointer della classe
    - sono insieme alle altre dichiarazioni dell'ambiente globale (variabili e funzioni): in nostro layout offset iniziale è -2
- Nota: in caso di AR di un metodo
  - il suo Access Link (AL) contiene l'object pointer dell'oggetto ("this" in C/Java) su cui lo si ha invocato (ambiente delle dichiarazioni nel corpo della classe)

Estensione Object Oriented
Generazione Enriched AST

Abstract Syntax Tree

- Dichiarazioni
  - ClassNode
    - mettere i figli campi in campo "fields" e i figli metodi in campo "methods", in ordine di apparizione
    - in campo "superID" mettere ID di classe da cui eredita (null se non eredita)
  - FieldNode (come ParNode)
  - MethodNode (come FunNode)
- Nuove e vecchie (VarNode, FunNode, ParNode) ereditano da classe abstract DecNode
  - contenente campo type e metodo getType()
    - dove memorizzare il tipo dell'ID (messo in Symbol Table)

Abstract Syntax Tree

- Espressioni (a lato si indica elemento sintattico)
  - IdNode ID
  - CallNode ID()
  - ClassCallNode ID.ID()
  - NewNode new ID()
  - EmptyNode null
- Tipi (in AST/restituiti da type checking)
  - RefTypeNode ID
    - contiene l'ID della classe come campo
  - EmptyTypeNode (tipo di null)
    - non in AST ma restituito da typeCheck() di EmptyNode

Symbol Table: struttura STentry invariata

- Offset in STentry calcolato diversamente per:
  - classi/funzioni/variabili
  - parametri
  - campi
  - metodi

  in base al rispettivo layout
- Tipo in STentry per metodi è ArrowTypeNode

Symbol Table: STentry per i nomi delle Classi

- Nesting level è 0 (ambiente globale)
- Offset: da -2 decrementando ogni volta che si incontra una nuova dichiarazione di classe
  - in base alla sintassi, dichiarazioni di funzioni/variabili appaiono in seguito nell'ambiente globale
- Tipo:
  - ClassTypeNode che ha come campi:
    - ArrayList<TypeNode> allFields
      (tipi dei campi, inclusi quelli ereditati, in ordine di apparizione)
    - ArrayList<ArrowTypeNode> allMethods
      (tipi funzionali metodi, inclusi ereditati, in ordine apparizione)

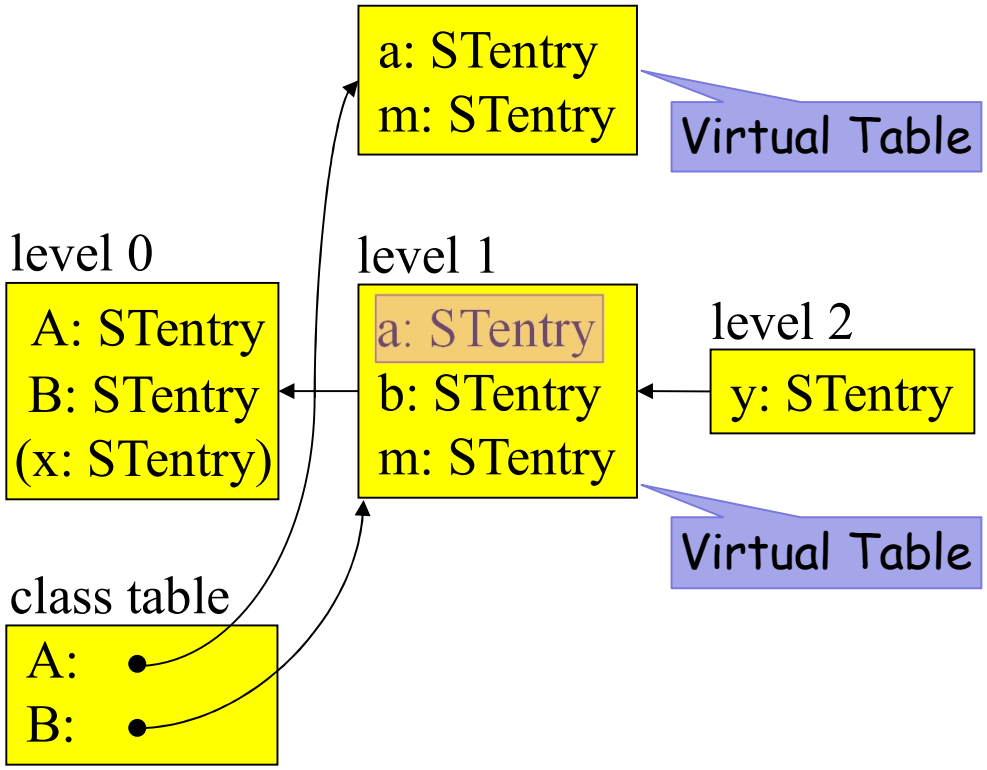
Symbol Table: dentro classi diviene Virtual

- Quando si visita lo scope interno di una classe, la Symbol Table per il livello corrispondente (livello 1 da noi) deve includere anche le
  - STentry per i simboli (metodi e campi) ereditati su cui non è stato fatto overriding
- Per questo motivo tale tabella viene chiamata Virtual Table

Symbol Table: aggiunta della Class Table

- In aggiunta a Symbol Table multilivello uso anche una Class Table
  - mappa ogni nome di classe nella propria Virtual Table
    - Map< String, Map<String,STentry> > classTable
  - serve per preservare le dichiarazioni interne ad una classe (campi e metodi) una volta che il visitor ha concluso la dichiarazione di una classe
    - le rende accessibili anche in seguito tramite il nome della classe, es.
      - uso di un metodo tramite notazione ID1.ID2(..)
      - calcolo Virtual Table di classe che eredita

Symbol Table: esempio




```
let
  class A (a: int) {
    fun m:int() ... ;
  }
  class B extends A (b: int) {
    fun m:bool()
      let
        var y:bool = true;
      in ... ;
  }
  var x:int = 5;
in
  ... ;
```

we are here

Symbol Table: gestione Class e Virtual Tables

- In visita di dichiarazione di classe (ClassNode):
  - nella Symbol Table (livello 0) viene aggiunto il nome della classe mappato ad una nuova STentry
    - se non si eredita, il tipo è un nuovo oggetto ClassTypeNode con una lista inizialmente vuota in allFields e allMethods
    - altrimenti, il tipo viene creato copiando il tipo della classe da cui si eredita (si deve creare copia di **tutto il contenuto** dell'oggetto ClassTypeNode e non copiare il solo riferimento)
  - nella Class Table viene aggiunto il nome della classe mappato ad una nuova Virtual Table
    - se non si eredita, essa viene creata vuota
    - altrimenti, viene creata copiando la Virtual Table della classe da cui si eredita (si deve creare copia di **tutto il contenuto** della Virtual Table e non copiare il solo riferimento)

Symbol Table: gestione Class e Virtual Tables

- All'entrata dentro la dichiarazione della classe:
  - viene creato un nuovo livello per la Symbol Table
    - ma anziché creato vuoto, viene posto essere la nuova Virtual Table creata (ogni livello è un riferimento!)
- All'interno della dichiarazione della classe:
  - Virtual Table e oggetto ClassTypeNode (contenuto dentro la STentry del nome della classe) vengono aggiornati tutte le volte che si incontra
    - la dichiarazione di un campo (parametro della classe)
      - direttamente senza visita di FieldNode
    - la dichiarazione di un metodo, tramite visita di MethodNode
- All'uscita dalla dichiarazione della classe
  - inalterato: rimosso livello corrente Symbol Table

Symbol Table: dichiarazioni campi e metodi

1. Aggiornamento Virtual Table
   - come inserimento in livello corrente Symbol Table di dichiarazioni di parametri (campi) e funzioni (metodi),

a parte

- se nome di campo/metodo è già presente, non lo considero errore, ma overriding: sostituisco nuova STentry alla vecchia **preservando l'offset** che era nella vecchia STentry
  - non consentire overriding di un campo con un metodo o viceversa
- altrimenti, invariato: uso contatore di offset e lo decremento/incremento

   - come contatore di offset per i metodi usare il campo decOffset del visitor (acceduto anche in visita di MethodNode)

Symbol Table: dichiarazioni campi e metodi

2. Aggiornamento ClassTypeNode
   - fatto interamente nel codice della visita di ClassNode
   - considero tipo e offset del campo/metodo dichiarato che ho messo dentro la sua STentry al punto 1.
     - per i campi aggiorno array allFields settando la posizione -offset-1 al tipo (in nostro layout offset primo campo è -1)
     - per i metodi aggiorno array allMethods settando la posizione offset al tipo funzionale (in layout offset primo metodo è 0)
       - aggiungere campo "offset" a MethodNode e, durante la sua visita, settare tale campo a offset messo in sua STentry
       - dopo la visita a MethodNode usare il suo campo "offset" per aggiornare l'array allMethods

Symbol Table: dichiarazioni campi e metodi

- Inizializzazione contatore di offset per campi/metodi
  - se non si eredita, settato inizialmente in base a layouts di oggetti e dispatch tables
    - in nostri layouts: -1 per campi e 0 per metodi
  - altrimenti, settato in base a ClassTypeNode in STentry della classe da cui si eredita: primo offset libero in base a lunghezza di allFields e di allMethods
    - nostri layouts: -lunghezza-1 per campi e lunghezza per metodi

Symbol Table: decorazione nodi AST (usi di ID)

- IdNode e CallNode ID e ID()
  - invariati: STentry di ID in campo "entry"
- ClassCallNode ID1.ID2()
  - STentry di ID1 in campo "entry"
    - cercata come per ID in IdNode e CallNode (discesa livelli)
  - STentry di ID2 in campo "methodEntry"
    - cercata nella Virtual Table (raggiunta tramite la Class Table) della classe del tipo RefTypeNode di ID1
      - se ID1 non ha tale tipo si ha una notifica di errore

Symbol Table: decorazione nodi AST (usi di ID)

- NewNode new ID()
  - STentry della classe ID in campo "entry"
    - ID deve essere in Class Table e STentry presa direttamente da livello 0 della Symbol Table
- ClassNode class ID1 extends ID2 ....
  - STentry della classe ID2 in campo "superEntry"
    - ID2 deve essere in Class Table e STentry presa direttamente da livello 0 della Symbol Table

Estensione Object Oriented
Type Checking

Struttura Super Type

- Campo statico superType di TypeRels che mappa ID di classe in ID di sua super classe
  - Map<String,String> superType
    - struttura che definisce la gerarchia dei tipi riferimento (RefTypeNode)
    - da aggiornare quando si visita ClassNode tramite campo superID

Subtyping

- isSubtype() in TypeRels estesa considerando:
  - un tipo riferimento RefTypeNode sottotipo di un altro in base alla funzione superType
    - raggiungibilità applicandola multiple volte
  - un tipo EmptyTypeNode sottotipo di un qualsiasi tipo riferimento RefTypeNode
  - un tipo funzionale ArrowTypeNode sottotipo di un altro (necessario per overriding tra metodi) in base alla:
    - relazione di co-varianza sul tipo di ritorno
    - relazione di contro-varianza sul tipo dei parametri

Dichiarazioni

- FieldNode
  - non usato (come ParNode)
- MethodNode
  - identico a FunNode
- ClassNode (dopo aggiornamento superType)
  - si richiama sui figli che sono metodi
  - confronta suo tipo ClassTypeNode in campo "type" con quello del genitore in campo "superEntry" per controllare che eventuali overriding siano corretti
    - scorre tipi in array allFields/allMethods del genitore e controlla che il tipo alla stessa posizione nel proprio array allFields/allMethods sia sottotipo

Espressioni

- IdNode ID
  - invariato (ID non deve essere di tipo funzionale), con aggiunta che non sia il nome di una classe (di tipo ClassTypeNode)
- CallNode ID()
  - invariato (ID deve essere di tipo funzionale)
- ClassCallNode ID1.ID2()
  - come CallNode
    - symbol table visitor ha già controllato che ID1 sia di tipo RefTypeNode

Espressioni

- NewNode new ID()
  - controlla parametri come CallNode, e torna un RefTypeNode
    - recupera i tipi dei parametri tramite allFields del ClassTypeNode in campo "entry"
- EmptyNode null
  - ritorna tipo EmptyTypeNode

Estensione Object Oriented
Code Generation

Dispatch Tables

- Quando si genera il codice per la dichiarazione di una classe viene creata la sua Dispatch Table (seguendo le regole spiegate a lezione)
- Il codice generato la alloca nello heap e mette il relativo dispatch pointer in AR dell'amb. globale
  - sarà reperibile all'offset della classe
- E' quindi comodo accedere direttamente ad indirizzo (fp) dell'AR dell'ambiente globale
  - tale indirizzo in base a nostro layout dell'ambiente globale è costante MEMSIZE (valore iniziale di $fp)

Struttura Dati per Dispatch Tables

- Per ogni classe si costruisce la relativa Dispatch Table (un ArrayList di String)
  - etichette (indirizzi) di tutti i metodi, anche ereditati, ordinati in base ai loro offset
    - cioè stesso ordine di allMethods nel ClassTypeNode
- Le Dispatch Table di tutte le classi vengono create staticamente dal compilatore
  - in ordine di dichiarazione classi nell'ambiente globale
  - memorizzate in campo privato dispatchTables del visitor
    - List< List<String> > dispatchTables

Dichiarazioni

- FieldNode
  - non usato (come ParNode)
- MethodNode
  - genera un'etichetta nuova per il suo indirizzo e la mette nel suo campo "label" (aggiungere tale campo)
  - genera il codice del metodo (invariato rispetto a funzioni) e lo inserisce in FOOLlib con putCode()
  - ritorna codice vuoto (null)

Dichiarazioni

- ClassNode
  - ritorna codice che alloca su heap la dispatch table della classe e lascia il dispatch pointer sullo stack,
  - ciò viene fatto come descritto in seguito

Dichiarazione Classe: costruzione Dispatch Table

1. creo la Dispatch Table e la aggiungo a dispatchTables
   - se non si eredita, essa viene inizialmente creata vuota
   - altrimenti, viene creata copiando la Dispatch Table della classe da cui si eredita (si deve creare copia di **tutto il contenuto** della Dispatch Table e non copiare il solo riferimento)
     - la individuo in base a offset classe da cui eredito in "superEntry"
       - per layout ambiente globale: posizione -offset-2 di dispatchTables
2. considero in ordine di apparizione i miei figli metodi (in campo methods) e, per ciascuno di essi,
   - invoco la sua visit()
   - leggo l'etichetta a cui è stato posto il suo codice dal suo campo "label" ed il suo offset dal suo campo "offset"
   - aggiorno la Dispatch Table creata settando la posizione data dall'offset del metodo alla sua etichetta

Dichiarazione Classe: codice ritornato

1. metto valore di $hp sullo stack: sarà il dispatch pointer da ritornare alla fine
2. creo sullo heap la Dispatch Table che ho costruito: la scorro dall'inizio alla fine e, per ciascuna etichetta,
   - la memorizzo a indirizzo in $hp ed incremento $hp

Espressioni: codice ritornato

- EmptyNode null
  - mette sullo stack il valore -1
    - sicuramente diverso da object pointer di ogni oggetto creato
- IdNode ID
  - invariato
    - indipendentemente dal fatto che, risalendo la catena statica, giunga ad AR in stack o ad oggetto in heap comunque prendo il valore che c'è all'offset della STentry

Espressioni: codice ritornato

- ClassCallNode ID1.ID2()
  - inizia la costruzione dell'AR del metodo ID2 invocato:
    - dopo aver messo sullo stack il Control Link e il valore dei parametri
      - fin qui il codice generato è invariato rispetto a CallNode
    - recupera valore dell'ID1 (object pointer) dall'AR dove è dichiarato con meccanismo usuale di risalita catena statica (come per IdNode) e lo usa:
      - per settare a tale valore l'Access Link mettendolo sullo stack e, duplicandolo,
      - per recuperare (usando l'offset di ID2 nella dispatch table riferita dal dispatch pointer dell'oggetto) l'indirizzo del metodo a cui saltare

Espressioni: codice ritornato

- CallNode ID()
  - controllo se ID è un metodo (offset ≥ 0)
    - se non lo è, invariato
    - se lo è, modificato:
      - dopo aver messo sullo stack l'Access Link impostandolo all'indirizzo ottenuto tramite risalita della catena statica (in base a differenza di nesting level di ID) e aver duplicato tale indirizzo sullo stack
        - fin qui il codice generato è invariato
        - si noti che in questo caso tale indirizzo è l'object pointer
      - recupera (usando l'offset di ID nella dispatch table riferita dal dispatch pointer dell'oggetto) l'indirizzo del metodo a cui saltare

Espressioni: codice ritornato

- NewNode new ID()
  - **prima**:
    - si richiama su **tutti** gli argomenti in ordine di apparizione (che mettono ciascuno il loro valore calcolato sullo stack)
  - **poi**:
    - prende i valori degli argomenti, uno alla volta, dallo stack e li mette nello heap, incrementando $hp dopo ogni singola copia
    - scrive a indirizzo $hp il dispatch pointer recuperandolo da contenuto indirizzo MEMSIZE + offset classe ID
    - carica sullo stack il valore di $hp (indirizzo object pointer da ritornare) e incrementa $hp
  - nota: anche se la classe ID non ha campi l'oggetto allocato contiene comunque il dispatch pointer
    - == tra object pointer ottenuti da due new è sempre falso!

Estensione Object Oriented
Ottimizzazioni

Ridefinizione Erronea di Campi e Metodi

- Rende possibile rilevare la ridefinizione (erronea) di campi e metodi con stesso nome effettuata all'interno della stessa classe
  - la trattavamo come fosse un overriding
- In symbol table visitor, mentre si scorrono le dichiarazioni di campi e metodi di una classe,
  - usare un campo del visitor contenente un oggetto HashSet<String> creato vuoto all'entrata nella classe
  - ad ogni dichiarazione di campo o metodo:
    - controllare se il suo nome è già presente nella HashSet
      - se lo è notificare l'errore, altrimenti aggiungerlo alla HashSet
    - gestire la dichiarazione come in precedenza

Type Checking Più Efficiente per ClassNode

- Migliora l'efficienza nel type checking della dichiarazione delle classi
  - effettua il controllo di correttezza (subtyping) solo per i campi/metodi su cui è stato fatto overriding
- Nuovo funzionamento type checking descritto in slide successiva:
  - richiede di recuperare l'offset ed il tipo per ogni suo figlio campo o metodo
    - aggiungere campo "offset" a FieldNode (come già fatto per MethodNode) e, nel symbol table visitor, settarlo a offset messo in STentry

Type Checking Più Efficiente per ClassNode

- Si richiama sui figli che sono metodi (invariato)
- In caso di ereditarietà controlla che l'overriding sia corretto
  - Chiamato parentCT il tipo (un ClassTypeNode) in "superEntry"; per ogni proprio figlio campo/metodo:
    - calcola la posizione che, in allFields/allMethods di parentCT, corrisponde al suo offset
      - in nostri layouts: -offset-1 per campi e offset per metodi
    - se la posizione è inferiore a lunghezza di allFields/allMethods di parentCT (overriding), controlla che il tipo del figlio sia sottotipo del tipo in allFields/allMethods in tale posizione

Type Checking con Lowest Common Ancestor

- Rende possibile utilizzare nei rami then ed else di un "if-then-else" due espressioni
  - non solo quando sono una sottotipo dell'altra,
  - ma anche quando hanno un lowest common ancestor

- Type checking di IfNode
  - chiama lowestCommonAncestor (nuovo metodo statico da aggiungere a TypeRels) sui tipi ottenuti per le espressioni nel then e nell'else:
    - se ritorna null il typechecking fallisce, altrimenti restituisce il tipo ritornato

Type Checking con Lowest Common Ancestor

metodo:
TypeNode lowestCommonAncestor(TypeNode a,TypeNode b)

- per a e b tipi riferimento (o EmptyTypeNode)
  - se uno tra "a" e "b" è EmptyTypeNode torna l'altro; altrimenti
  - all'inizio considera la classe di "a" e risale, poi, le sue superclassi (tramite la funzione "superType") controllando, ogni volta, se "b" sia sottotipo (metodo "isSubtype") della classe considerata:
    - torna un RefTypeNode a tale classe qualora il controllo abbia, prima o poi, successo, null altrimenti
- per a e b tipi bool/int
  - torna int se almeno uno è int, bool altrimenti
- in ogni altro caso torna null