

Riassunto PR2

Classe : Modello per la creazione degli oggetti \Rightarrow Oggetto : Istanza di una classe

Una classe contiene:

- Costruttore : Istanzia l'oggetto
- Metodi : Operazioni
- Variabili d'istanza: Variabili che indicano lo stato locale degli oggetti:
 - Public
 - private

public: Visibile fuori dall'oggetto

private: Visibile solo all'interno dell'oggetto

Gli valori Null: Indica che un oggetto/variabile di tipo riferimento, è null

Heap: le variabili dentro lo heap vengono automaticamente istanziate:

int = 0
boolean = False
riferimento = Null

→ Variabili d'istanza quando si crea un oggetto

→ Variabili di classe, quando si cerca una classe

Metodi statici: Metodi implementati dall'oggetto che non dipendono dai valori delle variabili d'istanza (dipendono solo dalle variabili passate per parametro)

Usabili da tutti gli oggetti \Leftrightarrow Non dipendono dallo stato dell'oggetto

Uguaglianza:

$==$: Restituisce TRUE se due variabili denotano lo stesso RIFERIMENTO

equals: Restituisce TRUE se due variabili denotano oggetti IDENTICI (stesso stato locale) tra stringhe

ASM (Abstract Stack machine): Modello computazionale che permette di descrivere la notazione di stato modificabile

Contiene:

- Workspace: Memorizzazione dei programmi in esecuzione

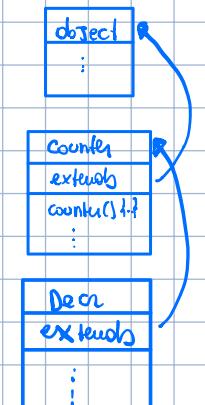
- Stack: Gestione delle variabili primitive e identificatori \Rightarrow le variabili dei metodi vengono allocate nello stack

\hookrightarrow Il codice del main viene salvato sullo stack quando

creo un oggetto / invoco un metodo, per poter mettere le istruzioni da eseguire nel workspace

- * - Heap: Memoria dinamica

- **tabelle dei metodi**: Contiene il codice dei metodi definiti nella classe, e tutte le componenti statiche definite nella classe stessa.
 - La tabella contiene un puntatore alla classe padre
 - L'insieme delle tabelle è un albero
 - Sono allocate nello Heap

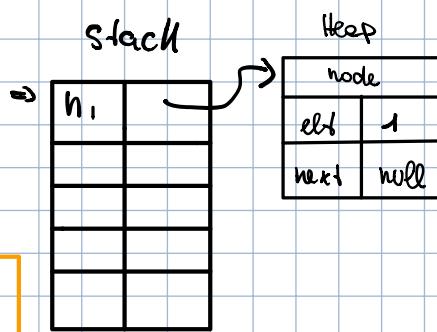


Esempi:

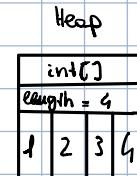
```
class node{
    private int elt;
    private node next;
}
```

Work space

node n1 = new node(1, null)
 \uparrow \uparrow
 elt next



```
int[] array;
array = new int[4]
array[0] = 1
:
array[3] = 4
```



Ereditarietà

Una classe B è sottotipo di A solo se l'oggetto che soddisfa l'interfaccia di B soddisfa anche quella di A

class B extends A() { \Rightarrow la classe B è sottotipo della classe A
 \vdots
 $\}$

B può usare il costruttore di A utilizzando il metodo **super()**

\downarrow
 Richiama il costruttore della classe supertipo

this: Le parole this fa riferimento all'oggetto corrente

Metodi:

Equals: Verifica se due oggetti hanno lo stesso riferimento (stesso oggetto) (in object)

hashCode: Restituisce, dato un oggetto, il suo valore hash da usare in una tabella di hash
Stesso valore per 2 oggetti identici

clone: Genera una copia dell'oggetto (la classe deve avere "implementable")

toString: Genera una stringa contenente il tipo e l'hash code dell'oggetto

tipi statici e dinamici:

- **Statico:** tipo che viene dato in compilazione del programma.

- **Dinamico:** tipo che viene dato durante il run-time

B prova = new C();
 $\xrightarrow{\text{Statico} = B}$ tipo che sta prima dell'assegnamento o quando la dichiara
 $\xrightarrow{\text{Dinamico} = C}$ tipo che sta dopo l'assegnamento

Importante:

1) Il tipo dinamico è **SEMPRE** un sotto-tipo del tipo statico

2) Se due classi hanno gli stessi metodi, l'invocazione sceglie sempre il metodo del tipo dinamico.

3) Quando passo per parameter un'istanza ad un metodo, viene considerato il tipo statico

Eccezioni

\hookrightarrow **Separation of concern:** Separare il codice di gestione degli errori dal codice "Normale"

Controllate (checked): Devono essere gestite esplicitamente dal programmi.

\hookrightarrow controllate dal compilatore

1) Istruzione racchiusa in un blocco try-catch-finally \rightarrow Catturare l'eccezione per gestire un comportamento anomalo

2) Nel caso la causa dell'eccezione chiama la funzione chiamo. throw

→ Sollevare l'eccezione programmando "un'uscita di emergenza" nell'esecuzione del programma

Non controllate (unchecked): Situazioni anomale provocano a run-time la terminazione (anomala) del programma in esecuzione



estendono Error e RuntimeException

Create un'eccezione:

```
public class // extends Exception / RuntimeException / Error {  
    public // () {}  
    super();  
}  
  
public // (String msg) {  
    super(msg);  
}
```

Comando throw: lancia l'eccezione specificata. Nell'header del metodo ci deve essere quale eccezione lancia (throws // ...)



throw new <Eccezione> (Riaccoglio di errore);

Defensive Programming: Verificare l'assenza di errore ogni volta ciò è possibile e a riportarli usando il meccanismo delle eccezioni

Tipi Generici: <T> => tipo generico che può assumere ogni tipo

$\langle E, F \rangle \Rightarrow$ coppie di tipi generic

Dichiarazione: class ... $\langle T \rangle$ implements ... $\langle T \rangle \{ \dots \}$

Utilizzo: $T \dots ;$

Vincoli di tipo: • $\langle typeVar \rangle$ extends SuperType (upperbound)

• $\langle typeVar \rangle$ extends classA of InterfaceB of interfaceC of ... (multiple upperbound)

• $\langle typeVar_1 \rangle$ extends supertype1, $\langle typeVar_2 \rangle$ extends supertype2 ...

• $\langle typeVar \rangle$ super SubType (lower Bound)

Regole in Java: Se type2 è un sottotipo di type3 e type2 \neq type3

↓
⇒ type1 $\langle type2 \rangle$ non è un sottotipo di type1 $\langle type3 \rangle$

Esempi:

1) Integer sottotipo di Number

2) ArrayList < E > sottotipo di List < E >

3) List < E > sottotipo di Collection < E >

4) List < Integer > non è sottotipo di List < Number >

Varianza per tipi:

1) Un operatore A è covariante se: T sottotipo S ⇒ A(T) sottotipo A(S) ($A(T) \subset A(S)$)

2) Un operatore A è contravariante se: T sottotipo S ⇒ S(T) supertipo A(S) ($A(S) \subset A(T)$)

3) Un operatore A è Bivariant se: È sia covariante che contravariante

4) Un operatore A è invariante se: Non è sia covariante che bivariante

Java utilizza queste definizioni se si confrontano 2 tipi uguali

Esempio: Se LargeBag < T > estende Bag < T >

⇒ LargeBag < Integer > è sottotipo di Bag < Integer >

Convenzioni

T = type

E = element

K = key

V = Value

Caso Particolare:

Se type1 è sottotipo di type2 \Rightarrow type1[] sottotipo type2[]

type Erasure: Tutti i tipi generici sono trasformati in Object nel processo di compilazione

Esempio: List<String> L1 = "();

List<Integer> L2 = "();

L1.getclass() == L2.getclass() // true

Generici e Casting: Node<E> n = (Node<E>) obj;

tipo dell'argomento non esiste
a run-time

Design by contracts (Specific)

Si descrive cosa deve fare e non come si fa (realizza)

1) Preconditions: Vincolo sulle proprietà che devono valere prima dell'invocazione del metodo



@ requires: Descrivono le condizioni di validità che il cliente deve garantire
↳ Preferibile sollevare un errore quando non vengono rispettati i requires (Fail Fast)

Postconditions: Vincolo sulle proprietà che devono valere al momento della terminazione

@ modifies: Elenco gli oggetti che sono modificati durante l'esecuzione del metodo, gli oggetti non presenti nell'elenco non sono modificati

@ throws: Elenco le possibili eccezioni

@ effects: Descrivono le proprietà dello stato finale

@ return: Descrivono i valori restituiti dal metodo

Sia M un'implementazione e S una specifica, Π soddisfa S solo se:

Tutti i comportamenti di Π sono permessi da S .

2) Astezione sulle strutture d'implementazione :

Il cliente deve solo comprendere le modalità di interazione senza dover comprendere i dettagli del codice

Metodologia: Scrivere prima di tutto le specifiche permette di utilizzarle come guida all'implementazione.

Comparare Specifiche

Una specie S_1 è più debole di S_2 se per ogni implementazione Π

M noddinga S₂ → II noddinga S₁

Realizzazione una specifica: Un mix tra specifiche forti e deboli

- Utile per il clienti
 - Favorisce l'implementazione
 - Favorisce il riuso del codice e la modularità

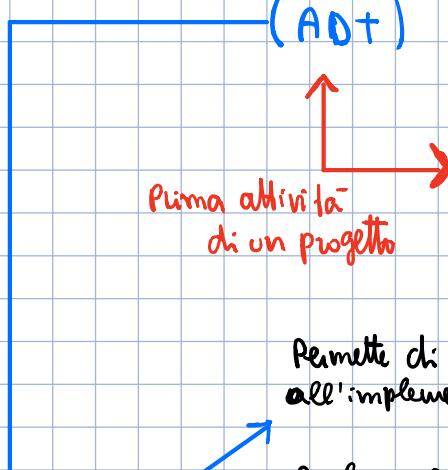
Astazioni:

1) Astrazione procedurale :

- Attivazione i dettagli d'implementazione delle procedure
 - Specifica come strumento di articolazione
 - Verifica dell'implementazione rispetto alle specifiche

2) Abstrakte nui dati:

- Analisi dei dettagli delle rapp. dei dati
 - Specifica come strumento di astrazione



Permette di ribadire le scelte specifiche all'implementazione del dato

Risolvere errori di programmazione molto facili
soltanto l'implementazione del dato

Modificare algoritmi: Maggiore performance

Adattare algoritmo a specifici contesti

ADT: Rappresentazione astratta di una struttura dati, modello che specifica:

- tipo dei dati memorizzati;
- Operazioni che possono essere eseguite dal cliente sui dati

Abstraction barrier: Fa da barriera tra il cliente e l'implementazione

L'unico modo per accedere e utilizzare le operazioni della astrazione

 **Stato astratto**:

- Non è lo stato concreto dei dati
- Permette di specificare le astrazioni procedurali
- Stato concreto non fa parte della specifica

Specifico ADT (abstraction data type)

Ammettono metodi che modificano (set)



1. overview - Definisce se il nuovo ADT è mutable o immutable
- Definisce il modello astratto da utilizzare nella specifica del tipo di dato astratto

typical element Specifica l'elemento tipo dell'ADT. Di solito si usano le var di istanza

2. abstract state Specificare le astrazioni procedurali per descriverlo

3. creators Restituiscono un nuovo oggetto del ADT **Costruttore**

4. Producers Operazioni (dell'ADT) che restituiscono valori:

5. Mutators Modificano lo stato concreto ADT

6. Observers Restituiscono informazioni sullo stato ADT

} **Metodi**

ADT Più comuni: Ogni struttura ha i metodi: isEmpty();
 isNotEmpty();

1) **Pila**: Struttura dati dove gli oggetti possono essere inseriti ed estatti secondo

(stack) un comportamento LIFO (last in, last out)

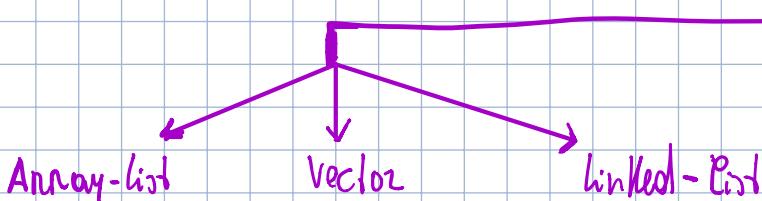
- Push(): inserisce l'oggetto in cima alla pila
- pop(): Elimina l'oggetto in cima alla pila
- top(): Ispeziona l'oggetto in cima alla pila senza esilarlo

2) Coda (queue): gli oggetti possono essere inseriti ed estratti secondo un comportamento FIFO (First in, First out)

- Enqueue(): inserisce l'elemento in fondo alla coda
- Dequeue(): Elimina l'oggetto inserito per primo in coda
- getFront(): Esamina il primo oggetto senza esilarlo

3) Collezioni: gruppo di elementi

- insieme: Gruppo di elementi: tutti distinti fra loro (set)
- lista: Gruppo di elementi in cui posso avere elementi ripetuti: (list)



strumenti essenziali per realizzare la specifica:

1) INVARIANTE: - Condizioni da rispettare per le variabili d'istanza

- stabilisce valore concreto dei valori dell'astrazione

Guida per chi implementa / modifica / verifica l'implementazione delle astrazioni

object → boolean

Nessun oggetto deve violare
nei invarianti

2) ABSTRACT FUNCTION: Stabilisce come interpretare la struttura dati concreta della implementazione

- Definita solo sui valori che rispettano l'invariante di rappresentazione
- Ogni operazione deve fare "la cosa giusta" con le rapp. concrete
- Associa le rapp. concrete ai valori astratti

Object → Abstract value

Defence Programming: Progettare codice in modo che

- 1) Alla chiamata dei metodi:
 - Verifica rep invariant
 - Verifica pre-conditioni
- 2) All'uscita dei metodi
 - Verifica rep invariant
 - Verifica post-conditioni

Espone la rappresentazione:

Bisogna progettare l'astrazione in modo da evitare di espone la rappresentazione.

Metodologia: 1) Fare una copia del dato senza espone l'implementazione

- Copy in: Parametri che diventano valori delle rapp.
- Copy out: Risultati che sono parte dell'implementazione

Non bisogna fare una copia shallow (sui puntatori) (Aliasing)

Esempio: --- ↴

```
private Point s,e;  
Public ... (Point s,Point e) {  
    this.s = New Point(s.x,s.y);  
    this.e = New Point(e.x,e.y);  
    Public ... getStart() {  
        return new Point(this.s.x,this.s.y);
```

← Copy

↳

2) Strutture non modificabili:

Pro

- Non può essere spezzato né inviolabile
- Più efficiente del copy out per dimensioni elevate
- Si usano librerie standard

- Aliasing non è un problema

- Non è necessario fare copie

- Rep invariant non può essere noto

Contro

- Permette di vedere le rappresentazioni

Principio di sostituzione:

- 1) Un oggetto del sotto-tipo può sostituire un oggetto del super-tipo senza influire sul comportamento dei programmi
- 2) Il sotto-tipo deve soddisfare le specifiche del super-tipo



↳ Prequisiti richiesti ad una superclasse devono essere almeno altrettanto vincolanti di quelli richiesti dalle sottoclassi: $\text{Pre}_{\text{super}} \Rightarrow \text{Pre}_{\text{sub}}$

3) Le postcondizioni e gli invarianti di una sottoclasse devono essere almeno altrettanto vincolanti di quelle delle superclassi: $\text{Post}_{\text{sub}} \Rightarrow \text{Post}_{\text{super}}$

4) Le eccezioni non devono essere più ampie nelle sottoclassi

Regole che devono essere rispettate

1) Regola della segnatura:

- Gli oggetti sub-type devono avere tutti i metodi del super-type
- Le signature dei metodi del sottotipo devono essere compatibili con quelle del super-tipo

▼
Una sottoclasse può ricevere un metodo restituendo un valore di un sottotipo
rispetto a quello previsto dal metodo della superclasse
(covariant return type)

2) Regola dei metodi:

- le chiamate dei metodi del sottotipo devono comportarsi come quelle del supertipo

$$\text{Pre}_{\text{super}} \Rightarrow \text{Pre}_{\text{sub}} \quad \text{e} \quad \text{Post}_{\text{sub}} \sqcup \text{Pre}_{\text{super}} \Rightarrow \text{Post}_{\text{super}}$$

3) Regola delle proprietà:

- Il sottotipo deve preservare tutte le proprietà che devono essere ereditate sul supertipo

- Creatori e produttori del sottotipo stabiliscono l'invariante
- tutti i metodi (anche i costitutori) del sottotipo
preservano l'invariante

Collections: Gruppo di oggetti omogenei (dello stesso tipo)

→ **array:** Collezione modificabile, lineare, di dimensione non modificabile

JAVA COLLECTION FRAMEWORK = Definisce una gerarchia di interface
e classi che realizzano una varietà
di collezioni

↓
VANTAGGIO

- Uso di algoritmi testati
- Efficienza implementazioni
- Interoperabilità
- Riuso del software

1) SFRUTTA i meccanismi di astrazione per:

- Specifica (documentazione delle interface)

- Parametrizzazione (uso dei generici)

Realizzazioni varie tipologie d'astrazione

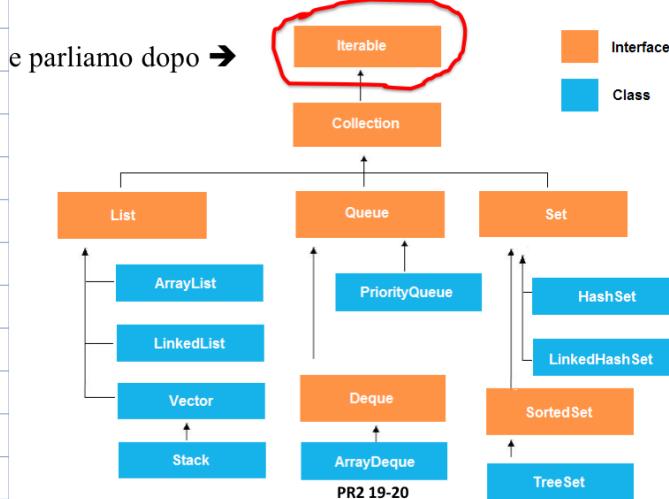
- Astrazione procedurale (Def. nuove operazioni)
- Astrazione dati (Definizione di nuovi tipi = ADT)
- Iterazione astratta
- Gerarchie di tipo (implements e extends)

2) Contiene realizzazioni di algoritmi efficienti di utilità generale

Interfacce :

- Collection <E> : Definisce operazioni basiche su collezioni
- Set <E> : Collezione senza duplicati
- List <E> : Sequenza lineare di elementi
- Queue <E> : Supporta politica FIFO (First in First out)
- Map <K, T> : Definisce un' associazione chiavi (k) → valori (t)

Gerarchia



CLASSI CONCRETE

- `ArrayList<E>`, `Vector<E>` : implementazione list <E> su array.
Sostituisce l'array di supporto con uno più grande quando è pieno
- `LinkedList<E>` : implementazione di list <E> basata su doubly-linked list. Usa un record `Node<E>` che ha: `Node<E> prev`, `E item`, `< Node<E> next`
- `TreeSet` : implementa set <E> con ordine crescente degli elementi
- `hashSet`, `linkedHashSet` : implementano set <E> con tabella di hash

Proprietà classi concrete:

	ArrayList	Vector	LinkedList	HashMap	LinkedHashMap	HashTable	TreeMap	HashSet	LinkedHashSet	TreeSet
Allows Null?	Yes	Yes	Yes	Yes (But One Key & Multiple Values)	Yes (But One Key & Multiple Values)	No	Yes (But Zero Key & Multiple Values)	Yes	Yes	No
Allows Duplicates?	Yes	Yes	Yes	No	No	No	No	No	No	No
Retrieves Sorted Results?	No	No	No	No	No	No	Yes	No	No	Yes
Retrieves Same as Insertion Order?	Yes	Yes	Yes	No	Yes	No	No	No	Yes	No
Synchronized?	No	Yes	No	No	No	Yes	No	No	No	No

Iteratori: è un'interfaccia che permette di estrarre uno alla volta gli elementi di una collezione, senza esporre la rappresentazione.

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove();  
}
```

Esempio di utilizzo:

```
// creo un iteratore sulla collezione  
Iterator<Integer> it = myIntCollection.iterator();  
while (it.hasNext()) { // finché ci sono elementi  
    int x = it.next(); // prendo il prossimo  
    ... // uso x  
}
```

Metodi:

```
public interface Iterator<E> {  
    boolean hasNext();  
    /* returns: true if the iteration has more elements. (In other words, returns  
     * true if next would return an element rather than throwing an exception.) */  
    E next();  
    /* returns: the next element in the iteration.  
     * throws: NoSuchElementException - iteration has no more elements. */  
    void remove();  
    /* Removes from the underlying collection the last element returned by the  
     * iterator (optional operation).  
     * This method can be called only once per call to next.  
     * The behavior of an iterator is unspecified if the underlying collection is  
     * modified while the iteration is in progress in any way other than by calling  
     * this method. */
```

?

it.next(): Pone l'attenzione sul prossimo elemento della collezione

it.hasNext(): Verifica se c'è un elemento successivo nella collezione.

Esempio:

- Esempio: stampa degli elementi di una qualsiasi collezione

```
public static <E> void print(Collection<E> coll) {  
    Iterator<E> it = coll.iterator();  
    while (it.hasNext()) // finché ci sono elementi  
        System.out.println(it.next());  
}
```

Creazione iterator: `Iterator <--> nome = NomeCollezione.iterator();`

Osservazione: La collezione non può essere modificata durante l'iterazione.

Unico comando ammesso è il remove dell'iterator

Foreach:

for (\in elem : coll) --; // per ogni elemento delle collezioni



usabile su array / collezione o oggetto che implementa Iterable<E>

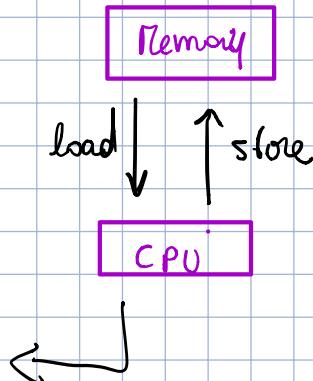
Una classe astratta non può essere initializzata

Secondo Parke

Modello Von Neumann

Memoria: Dove sono memorizzati i programmi e i dati
CPU: Dove vengono eseguiti i programmi

Fetch: Prelevare l'istruzione dalla memoria
Decode: Interpretare l'istruzione
Data fetch: Prelevare i dati dalla mem. su cui eseguire le operazioni
Execute: Esecuzione dell'operazione
Store: Inserire in mem. il risultato dell'operazione



JAVA VIRTUAL MACHINE

Linguaggio sorgente



Codice Macchina

1. compilatore

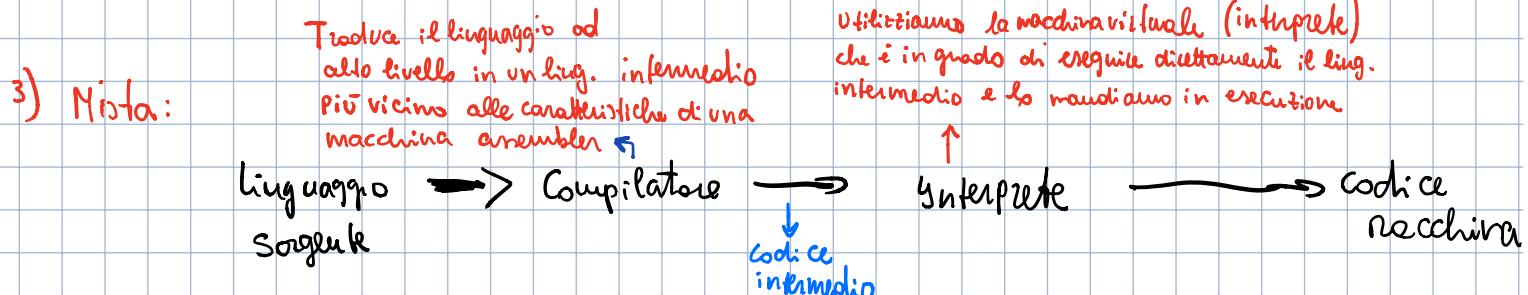
2. Interpretazione

possibili:

3 soluzioni per passare dal linguaggio naturale al codice macchina

3. Miste

- 1) Compilazione : traduce il codice sorgente in linguaggio macchina
- 2) Interpretazione : implementazione che esegue le istruzioni del linguaggio sorgente
(macchina virtuale) → Esempio: JVM che è una macchina in grado di eseguire immediatamente il bytecode di JAVA



Aspetti Significativi

- 1) Data Abstraction
 - Organizzazione i dati
 - Option types - Null values
 - Pairs - Record types
 - Lists - Recursive types
 - Pointers - References
 - Abstract data types
 - generics parametric vs
 - Polymorphism subtyping

Nascondere la scatola dell'implementazione

- 2) Control Abstraction
 - for - while - iterations
 - Case - switch
 - exceptions
 - threads

Funzione e procedura

Nascondere i dettagli dell'implementazione

- 3) Modular Abstraction

- Modules
- Namespaces Objects

- classes
- inheritance Interfaces
- Information hiding

Macchina astratta: Sistema virtuale che rapp. il comportamento di una macchina fisica individuando precisamente l'insieme delle risorse necessarie per l'esecuzione di programmi

Per lui:

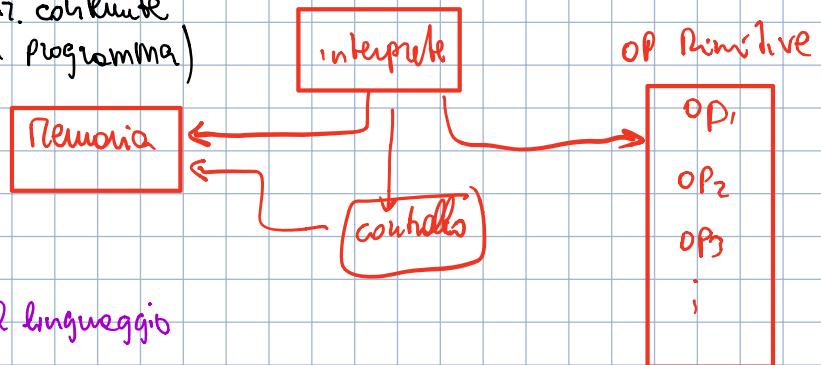
Collezione di strutture dati e algoritmi in grado di man. e eseguire programmi

Componenti:

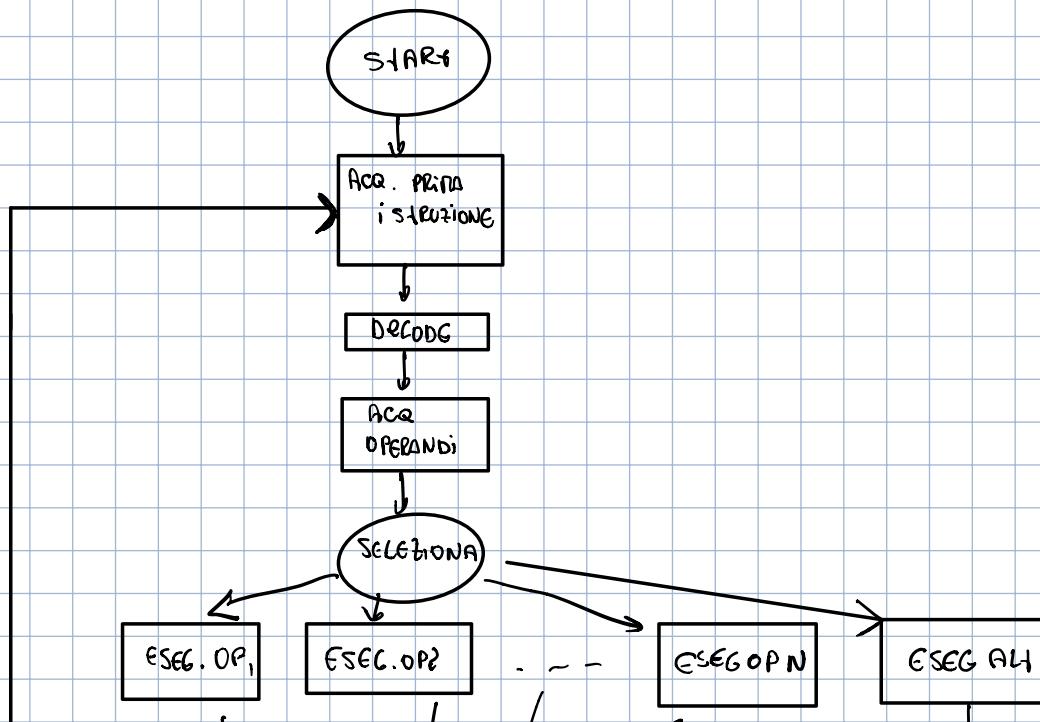
- Interpret (esegue le ist. contenute in un programma)
- Memoria
- controllo
- OP. Primitive

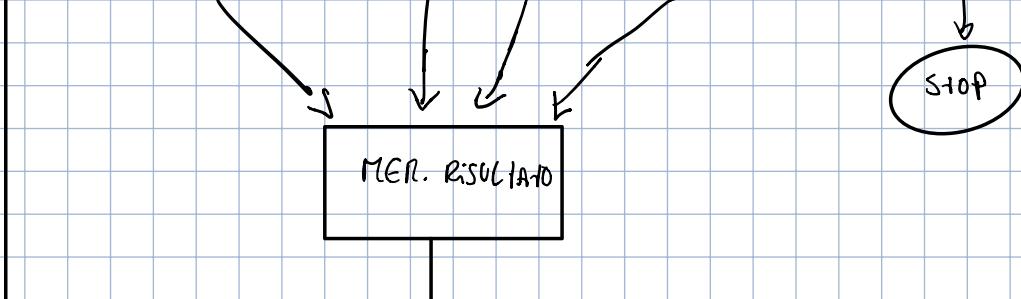
for,
while,
if ecc...

controllo
sia come
vergono
prese dalla mem i dati.



INTERPRETE





Linguaggio Macchina

→ Linguaggio eseguibile direttamente dalla macchina M

(Linguaggio macchina di M)

- M = macchina astratta
- L_M = linguaggio di M
- Programmi sono particolari dati su cui lavora l'interprete
- Alle componenti di L corrispondono componenti di L_M
 - tipi di dato primitivi
 - controllare acquisizioni e transf. dati
- I componenti di L sono realizzati mediante strutture dati e algoritmi implementati nel linguaggio macchina di una macchina ospitante M_0 già esistente

↳ La realizzazione di L può coincidere con l'interprete M_0
(ESTENSIONE)

↳ Può essere diverso dall'interprete M_0
(modo INTERPRETATIVO su M_0)

Macchina che ha L come ling. macchina
↗

- Implement. di L = Realizzazione di M_L su macchina ospite M_0
 M_L e M_0 hanno lo stesso interprete

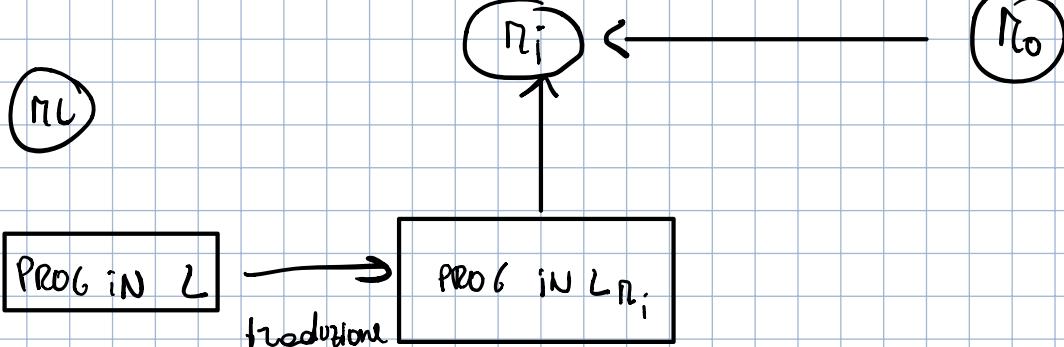
INTERPRETE PURO: M_L è realizzata su M_0 in modo interpretativo

↳ Scarsa efficienza → vengono usate istruz. di low per fare ciò che voglio fare

nel linguaggio L

COMPILATORE PURO : M_L non viene realizzata, i prog. tradotti sono eseguiti su M_0 * traduce il ling. L in un linguaggio l_0
→ problema della din. del codice prodotto

MACCHINA INTERMEDIA



L_{R_i} = ling. intermedio R_i = Recchini intermedio

$M_L = M_i$ interpretazione pura

→ possibile solo ne la diff tra M_0 e M_L
è molto limitata

$M_0 = R_i$ traduzione pura

II

L ling. annesso di M_0

A seconda ↗ spostiamo M_i verso M_0

o M_L .

↓
In tutti gli altri casi ho sempre una M_i che estende la M_0 in alcuni componenti

Il compilatore (R_i realizzata per estensione su M_0) ⇒ Stesso interprete

Differenza tra M_i e M_0 se coincidono gli interpreti



R+S (supp. a tempo di esecuzione)



collezione di strutture dati e sottoprogrammi che devono essere

caricati in memoria prima dell'esecuzione delle istruzioni

caricati su R_0 per permettere l'esecuzione del codice
prodotto dal compilatore

$$M_i = R_0 + RfS$$

Quindi: $L_{R_i} = L_{R_0} + \text{chiamate a RTS}$

IMPLEMENTAZIONI MISTE

Quando l'interprete di M_i \neq quello di R_0 esiste un ciclo
di interpretazione di L_{R_i} realizzato su R_0

- 1) Per ottenere un codice tradotto più compatto
- 2) Per facilitare la portabilità su altre macchine
- 3) Reimplementazione di L_{R_i}
- 4) Non si deve per forza reimplementare traduttore

L'implementazione di Java è mista

- traduzione prog da L a L_{R_i}
- programmi L_{R_i} sono interpretati su R_0

Riassunto implementazioni

- 1) Interpreti puro: $M_c = M_i$ e interprete di L realizzato in modo interpretativo su R_0
- 2) Compilatore puro: Macchina M_i realizzata per esecuzione su $R_0 \Rightarrow$ stesso interprete e RTS
- 3) Mista: traduzione prog da L_R a L_{R_i} e i programmi in L_{R_i} sono interpretati su R_0

Implementazione di Java \Rightarrow Implementazione mista (traduzione in bytecode ed esecuzione con la JVM)

- Programmi bytecode interpretati
- Interpreti di JVM opera su struttura a stack \rightarrow Operatori che operano su uno stack influiscono su neg.
- Esistenza Garbage Collector \Rightarrow Gestione mem. dinamica lasciata all'infrastruttura e non al programmatore

Le operazioni vengono
nuove nuove

Ocaml

- **typechecking:** Gli errori di tipo non vengono dati a run-time ma vengono controllati prima (compilazione)
- **Regole di esecuzione:** Regole di controllo che guidano l'esecuzione del programma sulla macchina astratta che è in grado di eseguire OCaml come linguaggio assembler

Valori: Esempio che non deve essere valutata ulteriormente (già valutata)

Esempio: 34 valore

34117 espressione ancora da valutare

Notazioni:

1) $\langle \text{exp} \rangle \rightarrow \langle \text{Val} \rangle$

Indica che l'espressione e quando viene valutata produce Val

2) $\text{eval}(e) = v$

Indica che se applico l'interprete delle espressioni di e , mi restituisce il valore v

let = Notazione di blocco

let $x = e_1$ in e_2

Voglio usare il valore dell'espressione di e_1 in e_2

Esempi:

- 1) let $x=2$ in $x+x \Rightarrow$ Usa il valore $x=2$ in $x+x$
- 2) let inc $x = x+1$ in inc 10 \Rightarrow definizione di una funzione
- 3) let $y = \text{"pro grammatione"}$ in (let $z = "2"$ in y^z)

Regole di valutazione

Valutazione let $x = e_1$ in e_2 , passaggi:

- 1) eval(e_1) = $v_1 \rightarrow$ Valuto e_1 (chiamo l'interprete della PA di e_1)
- 2) subst(e_2, x, v_1) = e_2' \rightarrow sostituisco che occorre di x in e_2 con il valore valutato da e_1
 ↓
 sostituisco solo le var libere (Non definite) (compilazione)
- 3) eval(e_2') = $v \rightarrow$ Valuto l'espressione dopo la subst (Richiamo l'interprete)
- 4) eval(let $x = e_1$ in e_2) = $v \rightarrow$ Il valore di tutto il let sarà il valore trovato al punto 3

Potrei interpretarlo anche così:

$$\frac{\text{eval}(e_1) = v_1 \quad \text{subst}(e_2, x, v_1) = e_2' \quad \text{eval}(e_2') = v}{\text{eval}(\text{let } x = e_1 \text{ in } e_2) = v}$$

let Annidati (come se ci fossero parentesi graffe fra i blocchi let)

let $x = 42$ in $x + (\text{let } x = \text{"riso"} \text{ in int-of-string } x)$

Non viene sostituito $x=42$ nel secondo let perché esistono un altro let non è visibile dal primo. Quindi viene valutato il secondo e poi il primo

Viene chiamato overlapping

Dichiarazioni di funzioni (Si usa il let)

let $f(x:\text{int}) : \text{int} =$

let $y = x * 10$ in

$y * y;$

Dichiamo una funzione che prende un parametro
X intero e restituisce un intero

Funzione ricorsiva

se ne per def. ricorsiva

let nec pow $x y =$

if $y = 0$ then 1

else $x * \text{pow } x(y-1)$

Valutazione di una funzione:

le funzioni sono valori.

↳ Mecanismo di ordine superiore: posso utilizzare una funzione in tutti quei posti dove mi aspetto un valore

Potenziale: posso creare funzioni anonymous

Esempi: let double $(x:\text{int}): \text{int} = 2 * x;$

1) let twice $((f:\text{int} \rightarrow \text{int}), x:\text{int}) : \text{int} = f(f x);;$

parametri:

W
valore
di ritorno

2) let quad $(x:\text{int}) : \text{int} = \text{twice}(\text{double}, x);;$

Usare una funzione in 2 tempi diversi

let $f x y = x^*x + y^*y$; → crea la fun f

let $g = f 3$; → crea la fun g con un par di: f già settato

$g 4$; → chiama g sull'altro parametro

Funzione composta

let compose $f g x = f(g x)$

Aplica prima g con param x
Aplica f con param g(x)

Funzioni anonime

$(\text{fun } x \rightarrow x+1) 3$; ⇒ restituisce 4 Usa x in $x+1$

let $f x = x+1$; identica a let $f = \text{fun } x \rightarrow x+1$

Polimorfismo

let $f x = x$;

Quando definisco una funzione senza specificare i tipi
di cosa li considero che possono prendere un qualunque tipo

Y inferenza di tipo : OCRL infiere il tipo più generale

OCRL list

- Diclaraazione

let $l = [1; 2; 3]$;

lista vuota
↑

let $l = []$;

- Aggiungere in testa

let $l_5 = 5 :: l$;

let $l = 5 :: 4 :: 3 :: 2 :: 1 :: []$

lista: [1, 2, 3, 4, 5]

- Accedere a una lista : Si usa il pattern matching



1) Può essere nil, []

2) Può essere ottenuta mediante un op. di cons di un elemento a una lista

Pattern matching

match e with

| P₁ → e₁

P₁, ..., P_n sono detti pattern

| P₂ → e₂

| :

| P_n → e_n

Esempi:

let empty lst = match lst with

| [] → true

Lista vuota

| h::t → false

Elemento cons

Un'altra lista

1° elemento nello
della lista lista

Può essere vuota

quindi considero le liste da 1 elem
in poi

let rec sum xs = match xs with

| [] → 0 \rightarrow Se è vuota somma 0

| h::t → h + sum t \rightarrow Se ha più di un elemento, estrai il
primo e sommalo alla ric sulla lista
restante

Patterni:

1) a :: [] matcha le liste con 1 elemento

2) a :: b matcha le liste con almeno 1 elemento

3) a :: b :: [] matcha le liste con esattamente 2 elementi

4) a :: b :: c matcha le liste con almeno 2 elementi

s) `a::b::c::d` matcha le liste con almeno 3 elementi

Option type

`let rec max-list = function`

`| [] → None` → Nessun valore

`| h::t → match max-list t with`

`| None → Some h` → Qualche h

`| Some x → Some (max h x)`

↓
Se esiste qualche
x

↓
Qualche risultato
dell'espressione

Hereditari

`let rec map f = function`

`| [] → []`

`| x::xs → (f x)::(map f xs)`

Applico la funzione f a tutti gli elementi della lista

Definire nuovi tipi in OCaml : si usa "type"

→ nome tipo

`type giorno =`

`| lunedì |`

`| ... | domenica`

I nostri:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

type tree =

| Empty

→ Def. di un albero

| Node of tree → int * tree

1) Creatione: let $t = \text{Node}(\text{Node}(\text{Empty}, 1, \text{Empty}), 3, \text{Node}(\text{Empty}, 2, \text{Node}(\text{Empty}, 4, \text{Empty})))$

2) Ricerca: let rec contains t n

begin match t with

| Empty \rightarrow false

| Node(lt, x, rt) \rightarrow if $x = n$ then true
else if $n < x$ then (contains lt n)
else (contains rt n);

3) Insert: let rec insert t n =

begin match t with

| Empty \rightarrow Node(Empty, n, Empty)

| Node(lt, x, rt) \rightarrow if $x = n$ then t

else if $x < n$ then Node(insert lt n, x, rt)
else Node(lt, x, insert rt n)

3) Delete: let rec delete t n =

begin match t with

| Empty \rightarrow Empty

| Node(lt, x, nt) \rightarrow if $x = n$ then

begin match

| (Empty, Empty) \rightarrow Empty

| (Node, Empty) \rightarrow lt

| (Empty, node) \rightarrow nt

| - \rightarrow let m = tree-max lt in

Node(delete lt m,
m, nt)

else if $n < x$

then Node(delete lt n, x, nt)

else Node(lt, x, delete nt);;

Generici in OCaml

let rec length (l: 'a list) : int =

begin match l with

| [] \rightarrow 0

| _ :: tl \rightarrow 1 + (length tl);;

Funzione utilizzabile per una lista di un tipo qualsiasi

Fa la stessa cosa se non definisco il tipo dei parametri e ritorno

Generic tree:

type 'a tree =

| Empty

| Node of 'a tree * 'a * 'a tree

Collection (Set) → Un insieme di dati omogenei senza duplicati:

↳ lista dove non ho duplicati e non importa l'ordine

Interface in OCaml: tremafe "module"

module type Set = sig

type 'a set

val empty : 'a set

val add : 'a → 'a set → 'a set

:

File: x.mli

Indice che è un'interfaccia

Utilizzarne l'interfaccia → come se dicessi myset implements set

module myset : set = struct

:

implementazione operandi

:

;;

dot notation : permetti di utilizzare le api di OCaml

let s1 = Myset.add 3 myset.empty

Open: Posso usare open per non utilizzare la dot notation

open FlySet

let s1 : int set = Empty

Uguaglianza

$= =$ Uguaglianza su puntatori o valori primitivi

$=$ Uguaglianza nulla struttura

Exceptions

- Un'eccezione può essere lanciata tramite il comando
`raise`

`if m=0 then Raise name eccez.`

- Catturare un'eccezione tramite `try` e `with`

`try f x y with name eccez.` \rightarrow result

Introduzione nuove eccezioni:

- `Exception Error`
- `Exception Unix_Error of string`

Tipo di implementazione di OCAML : Mista

Realizzare un interprete in OCaml

Rappresentazione intermedia : Si utilizza OCAML

(Linguaggio Intermedio)

Albero di sintassi astratta

type op = Plus | times | Minus ...

type exp =

int - e of int \Rightarrow espressione intera \rightarrow int

| OP - e of exp * OP * exp \Rightarrow operatore \Rightarrow exp op exp

| Var - e of variable \Rightarrow uso di una variabile

| let - e of variable * exp * exp \Rightarrow Dichiarazione di una variabile

type value = exp

Esempio rappresentazione : 3 + 17

let l1 = int_e 3

let l2 = int_e 17

let l3 = OP_e (l1, plus, l2)

Eval (Valutazione):

- Se $e = n \Rightarrow \text{eval}(e) = n$

- Se $e = e_1 + e_2 \Rightarrow \text{eval}(e_1) = v_1, \text{eval}(e_2) = v_2$ e infine
 $\text{eval}(e_1 + e_2) = v_1 + v_2 = v$

- Se $e = e_1 * e_2 \Rightarrow eval(e_1) = v_1, eval(e_2) = v_2$ e inline
 $eval(e_1 * e_2) = v_1 * v_2 = v$

Decodifica operazioni

let is_value (e: exp) : bool = match e with
| Int_e → true
| OP_e → false
| let_e → false
| Vcn_e → false;;

Ci dice se l'espressione passata è un valore o un'operazione

let eval_OP (v1: exp)(op: operand)(v2: exp) : exp =
match v1, op, v2 with

| Int_e i, Plus, Int_e j → Int_e (i+j)
| Int_e i, Times, Int_e j → Int_e (i*j)
| Int_e i, Minus, Int_e j → Int_e (i-j)
| _, (Plus | Minus | times

if is_value v1, ff is_value v2

then raise failwith "type_error"

else raise failwith "Not_value"

Verifica se è un'operazione ammessa e nel caso esegue l'operazione

let substitute (v : value) (x : variable) (e : exp) : exp =

let rec subst (e : exp) : exp =

match e with

| int - e \rightarrow e

| OP - e (e_1, op, e_2) \rightarrow OP - e (subst e_1 , op , subst e_2)

| Var - e $y \rightarrow$ if $x = y$ then v else e

| let - e (y, e_1, e_2) \rightarrow let - e (y , subst e_1 ,

if $x = y$ then e_2 else subst e_2)

Interprete \rightarrow is_value : exp \rightarrow bool

RTS : eval - op : Value \rightarrow Variable \rightarrow exp \rightarrow exp

(Runtime support) \rightarrow subst : value \rightarrow variable \rightarrow exp \rightarrow exp

} utilizzati dall'
interprete per fare
la valutazione delle
espressioni

\rightarrow valutazione delle espressioni (interprete)

let rec eval (e : exp) : exp =

match e with

| Int - e $i \rightarrow$ Int - e i

| OP - e (e_1, op, e_2) \rightarrow eval - op eval e_1 op eval e_2

| let - e (x, e_1, e_2) \rightarrow let $v_1 =$ eval e_1 in

let $e_2' =$ substitute $V_1 x e_2$ in
eval e_2'

| Var - e $x \rightarrow$ var (x) (Unbound Variable x)

↳ eccezione fa parte del RTS

Se e' un valore lo restituisce senz'altro l'espressione tramite
l'operazione in modo ricorsivo fin quando non diventa un valore.

Se e' un lett. valore l'espressione 1, sostituisce il valore in e2' e poi valore e2'
e ottengo il valore finale.

Se e' una var. lancia un ecc perche' non la dovrei avere.

Regole di valutazione

1) $\text{exp} \rightarrow V$ V e' il valore di exp se valutando
 exp si ottiene V

2) $V \rightarrow V$ Valutazione valori

3) $\frac{\text{exp} 1 \Rightarrow V_1, \text{exp} 2 \Rightarrow V_2}{\text{exp} 1 * \text{exp} 2 \Rightarrow V_1 * V_2}$ Valutazione op. times

$\frac{\text{exp} 1 \Rightarrow V_1, \text{exp} \Rightarrow V_2}{\text{exp} 1 + \text{exp} 2 \Rightarrow V_1 + V_2}$ Valutazione op. somma

Dati

Classificazione:

1) Denotabili: Se possono essere associati ad un nome

2) Esprimibili: Se possono essere result della valutazione di un'espressione completa

3) Memorizzabili: Se possono essere mem. in una variabile

Un OCAML sono: Denotabili e esprimibili, ma non memorizzabili

Tipi di dato : Collezione di valori rappresentati con strutture dati e un insieme di op per manipolarli

1) **Dato di sistema:** Definiscono strutture dati e ruolo nella simulazione di costituti di controllo

2) **Dato di programma:** Domini corrispondenti ai tipi primi del linguaggio e ai tipi che l'utente puo' definire (se il linguaggio lo consente)

Descrittori (Valori ritrovati dalla valutazione di un'espressione)

`type evT =` Fa un eval sul tipo

| `Unit` of `int`

| `Bool` of `bool`

typecheck: Verifica se i tipi dei due operandi x e y corrispondono

`let typecheck (x,y) = match x with`

| "int" → match y with

| `Int (v)` → true

| _ → false

Valutazione dei tipi:

1) **Informazione tipi completamente conosciuta a tempo di compilazione (OCAML):**

- Si possono eliminare i descrittori di dato

- typechecking effettuato dal compilatore completamente (type checking statico)

2) Informazione tipi conosciuta completamente a tempo di esecuzione (JAVASCRIPT)

- Necesari descrittori per tutti i tipi di dato
- typechecking effettuato completamente a tempo di esecuzione (type checking dinamico)

3) Informazione tipi conosciuta parzialmente a tempo di compilazione (JAVA)

- Descrittori contengono solo l'informazione dinamica
- type checking effettuato in parte dal compilatore e in parte dall'CTS

tipi scalari:

	Boolean	Caratteri	int	Reali
Val:	true, false	A,B,C... ã, ã, ...	0,1,-1,...,maxint	valori razionali in un intervallo
OP:	or, and, not, condizionali	uguaglianza, codice/decodice, ecc...	+, -, *, mod, div	+, -, *, /, ...
rep:	un byte	1 byte (ASCII), 2 byte (UNICODE)	2 byte o 4 byte	4 byte
note:	C non ha tipo bool		Stringhe e interi lunghi anche 8 byte	reali e reali lunghi 8 byte

tipi composti:

- 1) Record (Struct): collezione di campi ciascuno di un tipo, ogni campo è selezionato con il suo nome
- 2) Record varianti:
- 3) Array: Funzione da un tipo ruolice ad un altro tipo
- 4) Unione: Sottoinsieme di un tipo base
- 5) Puntatore: Riferimento ad un oggetto

Record

```
Struct MixedData {
```

```
Struct MixedData {
```

1 byte ← chan Data1;
2 byte ← Short Data2;
4 byte ← int Data3;
1 byte ← chan Data4;

}

12 byte occupati usando
il padding

⇒
cambiando
ordine
↓
Viene applicato
in automatico
dal compilatore

1 byte ← chan Data1;
1 byte ← chan Data4,
2 byte ← Short Data2;
4 byte ← int Data3;;
}
8 byte occupati
(Nessun Padding)

Implementazione record

type label = lab of String

type expr =

| Record of (label * expr) list

⇒ lista di record

| Select of expr * label

⇒

Funzione di valutazione

let rec lookupRecordBody (Lab l) =

match body with

| [] → raise FieldNotFound

| (Lab l', v) :: t →

if l = l' then v else lookupRecord t (Lab l)

Interprete

evalRecord body = match body with

| [] → []

| (Lab l, e) :: t → (Lab l, eval e) :: evalRecord t

let rec eval e = match e with

| Record (body) → Record (evalRecord body)

| select (e, l) \rightarrow match eval e with

| Record (body) \rightarrow lookupRecord body l

| _ \rightarrow raise typeErrorMatch

Array : collezione di dati omogenei accedibili tramite indice

Formule di accesso:

$$v[i] = \text{base} + D^* i$$

OFFSET



D = Dimensione in byte del tipo della base

Esempio: 32 bit = 4 byte $\Rightarrow D = 4 \text{ byte}$

64 bit = 8 byte $\Rightarrow D = 8 \text{ byte}$

v[0]
v[1]
v[2]
⋮
⋮

D [

C non controlla la correttezza degli indici a run-time

utilizza i puntatori

Puntatori

1) Valori:

Riferimenti

costante null

2) Operazioni:

- creazione: funzioni di libreria che alloca e restituisce un puntatore

- Dereveriazione: Accesso al dato puntato: $*P$

- test di uguaglianza

Puntatori e array sono intercomparabili in C: Dinamica dei puntatori

Tipi di dato di sistema

Pila

Pila non modificabile: interfaccia

```
# module type PILA =
  sig
    type 'a stack
    val emptystack : int * 'a -> 'a stack
    val push : 'a * 'a stack -> 'a stack
    val pop : 'a stack -> 'a stack
    val top : 'a stack -> 'a
    val empty : 'a stack -> bool
    val lungh : 'a stack -> int
    exception Emptystack
    exception Fullstack
  end
```

Pila non modificabile: semantica

```
# module SemPila: PILA =
  struct
    type 'a stack = Empty of int | Push of 'a stack * 'a ("tipo algebrico ")
    exception Emptystack
    exception Fullstack
    let emptystack (n, x) = Empty(n)
    let rec max = function
      | Empty n -> n
      | Push(p,a) -> max p
    let rec lungh = function
      | Empty n -> 0
      | Push(p,a) -> 1 + lungh(p)
    let push (a, p) = if lungh(p) = max(p) then raise Fullstack else Push(p,a)
    let pop = function
      | Push(p,a) -> p
      | Empty n -> raise Emptystack
    let top = function
      | Push(p,a) -> a
      | Empty n -> raise Emptystack
    let empty = function
      | Push(p,a) -> false
      | Empty n -> true
  end
```

Pila non modificabile: implementazione

```
# module ImpPila: PILA =
  struct
    type 'a stack = Pila of ('a array) * int
    exception Emptystack
    exception Fullstack
    let emptystack (nm,x) = Pila(Array.create nm x, -1)
    let push(x, Pila(s,n)) = if n = (Array.length(s) - 1) then
      raise Fullstack else
      (Array.set s (n + 1) x;
       Pila(s, n + 1))
    let top(Pila(s,n)) = if n = -1 then raise Emptystack
      else Array.get s n
    let pop(Pila(s,n)) = if n = -1 then raise Emptystack
      else Pila(s, n -1)
    let empty(Pila(s,n)) = if n = -1 then true else false
    let lungh(Pila(s,n)) = n
  end
```

listा

Lista (non polimorfa): interfaccia

```
# module type LISTAINT =
  sig
    type intlist
    val emptylist : intlist
    val cons : int * intlist -> intlist
    val tail : intlist -> intlist
    val head : intlist -> int
    val empty : intlist -> bool
    val length : intlist -> int
    exception Emptylist
  end

# module ImpListaint: LISTAINT =
  struct
    type intlist = int
    let heapsize = 100
    let heads = Array.create heapsize 0
    let tails = Array.create heapsize 0
    let next = ref(0)
    let emptyheap =
      let index = ref(0) in
      while !index < heapsize do
        Array.set tails !index (!index + 1); index := !index + 1
      done;
      Array.set tails (heapsize - 1) (-1); next := 0
    exception Fullheap
    exception Emptylist
    let emptylist = -1
    let empty l = if l = -1 then true else false
    let cons (n, l) = if !next = -1 then raise Fullheap else
      (let newpoint = !next in next := Array.get tails !next;
       Array.set heads newpoint n; Array.set tails newpoint l; newpoint)
    let tail l = if empty l then raise Emptylist else Array.get tails l
    let head l = if empty l then raise Emptylist else Array.get heads l
    let rec length l = if l = -1 then 0 else 1 + length (tail l)
  end
```

Pila Modificabile

Pila modificabile: interfaccia

```
# module type MPILA =
sig
  type 'a stack
    val emptystack : int * 'a -> 'a stack
    val push : 'a * 'a stack -> unit
    val pop : 'a stack -> unit
    val top : 'a stack -> 'a
    val empty : 'a stack -> bool
    val lungh : 'a stack -> int
  val svuota : 'a stack -> unit
  val access : 'a stack * int -> 'a
exception Emptystack
exception Fullstack
exception Wrongaccess
end
```

Pila modificabile: semantica

```
# module SemMPila: MPILA =
struct
  type 'a stack = ('a SemPila.stack) ref
  exception Emptystack
  exception Fullstack
  exception Wrongaccess
  let emptystack (n, a) = ref(SemPila.emptystack(n, a))
  let lungh x = SemPila.lungh(!x)
  let push (a, p) = p := SemPila.push(a, !p)
  let pop x = x := SemPila.pop(!x)
  let top x = SemPila.top(!x)
  let empty x = SemPila.empty !x
  let rec svuota x = if empty(x) then () else (pop x; svuota x)
  let rec faccess (x, n) =
    if n = 0 then SemPila.top(x) else faccess(SemPila.pop(x), n-1)
  let access (x, n) = let nofops = lungh(x) - 1 - n in
    if nofops < 0 then raise Wrongaccess else faccess(!x, nofops)
end
```

Pila modificabile: implementazione

```
module ImpMPila: MPILA =
struct
  type 'x stack = ('x array) * int ref
  exception Emptystack
  exception Fullstack
  exception Wrongaccess
  let emptystack(nm, (x: 'a)) = ((Array.create nm x, ref(-1)): 'a stack)
  let push(x, ((s, n): 'x stack)) = if !n = (Array.length(s) - 1) then
    raise Fullstack else (Array.set s (!n + 1) x; n := !n + 1)
  let top(((s, n): 'x stack)) = if !n = -1 then raise Emptystack
    else Array.get s !n
  let pop(((s, n): 'x stack)) = if !n = -1 then raise Emptystack
    else n := !n - 1
  let empty(((s, n): 'x stack)) = if !n = -1 then true else false
  let lungh( (s, n): 'x stack) = !n
  let svuota (((s, n): 'x stack)) = n := -1
  let access (((s, n): 'x stack), k) =
    (*      if not(k > !n) then      *)
      Array.get s k
    (*      else raise Wrongaccess  *)
end
```

53

Metaprogramma

Programma che opera su altri programmi

- Utile per definire:
- Strumenti di supporto allo sviluppo
 - Estensioni del linguaggio

Metaprogrammazione

- Reflection : Abilità del programma di manipolare come

un dato qualcosa

- **Introspection**: Abilità di un programma di conoscere il suo stesso dato
- **Intercession**: Abilità di un programma di modificare il suo stesso stato in esecuzione o di alterare la sua interpretazione/significato

Java

```
import java.io.*  
import java.lang.reflect.*;  
Class c = Class.forName("java.lang.System");  
// Fetch System class  
Field f = c.getField("out");  
// Get static field  
Object p = f.get(null);  
// Extract output stream  
Class cc = p.getClass();  
// Get its class  
Class types[] = new Class[] { String.class };  
// Identify argument types  
Method m = cc.getMethod("println", types);  
// Get desired method  
Object a[] = new Object[] { "Hello, world" };  
// Build argument array  
m.invoke(p, a);  
// Invoke method
```

Javascript: eval

```
eval("3+4"); // Returns 7  
  
a= "5-"; b="2";  
eval(a+b); // Returns 3, result of 5-2  
  
eval(b+a); // Runtime syntax error  
  
a= "5-"; b = "1"; c = "a+a+b";  
eval(c); // Returns the string "5-5-1"  
  
eval(eval(c)) // Returns the number -1
```

Meta Ocaml

```
# let x = .<4 + 2>. ;;  
val x: int code = .<4 + 2>.  
  
# let y = .<.-x + .~x>. ;;  
val y int code = <(4 + 2) +  
(4 + 2)>.  
  
#let z = .! y;;  
val z : int = 12
```

Quote .<e>.
Antiquote .~
Execute .!

```
let rec metapower (n,x) =  
  if n = 0  
    then <@ 1 @>  
    else <@ _ _ @> (lift x) (metapower(n-1,x))  
  
val metapower : int int -> Expr<int>  
  
> let metapower4 = fun x -> metapower (4,x) ;;  
val metapower4 : int -> Expr<int>
```

Linguaggio di programmazione portuale:

- **Sintassi**: Definisce se un programma è sintatticamente corretto
- **Semantica**: Fornisce un'interpretazione ai tokeni in termini di realtà

e un significato ai programmi sintatticamente corretti

Teoria dei linguaggi formali: Fornisce formalismi di specifica e tecniche di analisi per trattare aspetti sintattici

Sintassi concreta: Grammatica libera da contesto (es: formule booleane)

Sintassi Antistetica: Rappresentazione lineare dell'albero sintattico (AST)

Albero sintattico di un exp morta come
exp può essere generata dalla grammatica
Gli operatori sono i nodi dell'albero e
gli operandi sono napp. dai sottialberi.

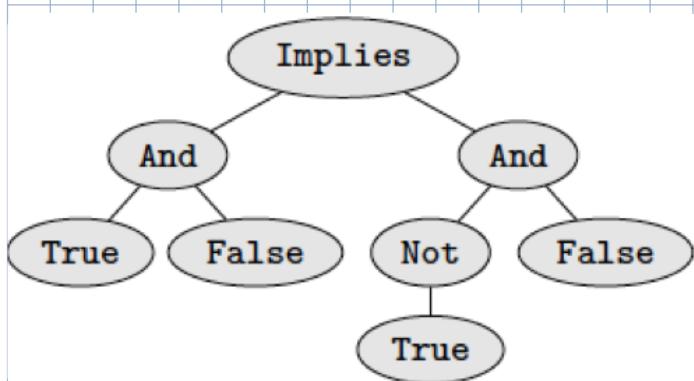
Esempio :

(True And False) Implies
((Not True) And False)

Sintassi concreta

Implies(And(True,False),
And(Not(True),False))

Sintassi Antistetica



Albero Sintattico

Semantica dei linguaggi

Tre metodi principali di analisi semantica

- **Semantica operazionale:** descrive il significato di un programma in termini dell'evoluzione (cambiamenti di stato) di una macchina astratta
- **Semantica denotazionale:** il significato di un programma è una funzione matematica definita su opportuni domini
- **Semantica assiomatica:** descrive il significato di un programma in base alle proprietà che sono soddisfatte prima e dopo la sua esecuzione

Ci dice come evolve lo stato delle macchine astratte per eseg. una istruzione / programma ecc..

Semantica operazionale di un linguaggio

Definire in modo formale una M_L in grado di eseguire i prog. scritti in L

Sistema di transizioni: Costituito da:

- Unione Config di stati
 - Relazione di transizione
- $\subseteq \text{Config} \times \text{Config}$

Notazione: $c \rightarrow d$ significa che c ed d sono nelle relazioni →

Intuizione: $c \rightarrow d$ lo stato c evolve nello stato d

Semantica operazionale "Small step"

- Relazione di transizione descrive un passo del processo di calcolo
- Abbiamo la transizione $e \rightarrow l$ se partendo da e l'esecuzione di un passo di calcolo ci porta nell'espressione l.

Quindi una valutazione completa di e sarà: $e \rightarrow l_1 \rightarrow \dots \rightarrow l_n$

- Nella small-step la valutazione di un prog. procede attraverso le configurazioni intermedie che può assumere il programma

Semantica operazionale "Big step" (USATA IN PRZ)

- La relazione di transizione descrive la valutazione completa di un prog./ espressione

- Scriviamo $e \Rightarrow v$ se l'esecuzione del programma/espressione e produce il valore v
- La valutazione completa di e è ottenuta unendo la val. completa di 2 sottoespressioni

Semantica OP. "big step" di espressioni logiche

$$\begin{array}{ll} \text{true} \Rightarrow \text{true} & \text{VALORI} \\ \text{false} \Rightarrow \text{false} & \frac{e \Rightarrow v}{\text{not } e \Rightarrow \neg v} \quad (\text{not}) \end{array}$$

$$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 \text{ and } e_2 \Rightarrow v_1 \wedge v_2} \quad (\text{and})$$

Regole di valutazione analoghe per OR e IMPLIES
Usiamo OPERATORI LOGICI sul dominio dei valori con tabelle di verità

Regole di derivazione : costituiscono un proof system

premessa₁ ... premessa_k
conclusione

- Definite per induzione strutturale sulle sintassi del linguaggio
- le "formule" che ci interessa dimostrare sono trasizioni del tipo $e \Rightarrow v$
- Componiamo le regole in base alla struttura sintattica di e e ottengono un proof tree

<u>True \Rightarrow True</u>	<u>False \Rightarrow False</u>	<u>True \Rightarrow True</u>
		<u>False And True \Rightarrow False</u>
	True And (False And True) \Rightarrow False	

- Ogni regola del proof system corrisponde a un pg. OCaml

Parm:

Passo 1: sintassi astratta

```
type BoolExp =  
| True  
| False  
| Not of BoolExp  
| And of BoolExp * BoolExp
```

Definizione della sintassi astratta tramite i tipi algebrici di OCaml

Passo 2: dalle regole di valutazione all'interprete OCaml

Passo 2: dalle regole di valutazione all'interprete OCaml

- Obiettivo: definire una funzione **eval** tale che **eval(e) = v** se e solo se $e \Rightarrow v$

- Esempio: dalla regola

$$\frac{e \Rightarrow v}{\text{not } e \Rightarrow \neg v}$$

- otteniamo il seguente codice OCaml

```
eval Not(exp0) -> match eval exp0 with  
    True -> False  
    | False -> True
```

Passo 3: Interprete di espressioni logiche (True, False, And, Not)

```
let rec eval exp =  
  match exp with  
    True -> True  
    | False -> False  
    | Not(exp0) -> match eval exp0 with  
        True -> False  
        | False -> True  
    | And(exp0,exp1) ->  
        match (eval exp0, eval exp1) with  
          (True,True) -> True  
          | (_,False) -> False  
          | (False,_) -> False
```

Espressioni a valori interi

Sintassi OCaml (astratta)



```
type expr =  
| Cstl of int // costanti intere  
| Var of string // variabili  
| Prim of string * expr * expr // operatori binari
```

→ Non basta, mi serve UNA struttura di implementazione (num - time structure)

che permetta di recuperare i valori associati agli identificatori (Bindina)

Ambiente e Memoria

Mi serve un Ambiente

- La macchina astratta di un linguaggio di programmazione include due componenti astratte particolarmente importanti
 - **Ambiente** (environment)
 - **Memoria** (store)
- L'ambiente è una collezione di **binding**
 - Binding: associazione tra nomi e valori
 - Valori: valori di una variabile, locazione in memoria, il codice di un sottoprogramma, l'indirizzo dove viene memorizzato un oggetto,
- **Memoria** è una collezione di **binding** tra locazioni di memoria e **valori memorizzabili**.
 - I binding della memoria sono modificati dalle operazioni di assegnamento

Esempio Binding:

- esempio in ML

```
let x = 2 + 1 in
    let y = x + x in
        x * y
```
- il binding di x è 3, il binding di y è 6, il valore calcolato dal programma è 18

Ambiente :

Un ambiente env è una collezione di binding

Esempio : env = { x → 25 , y → 6 }

- l'associazione tra l'identificatore x e il valore 25
- l'associazione tra l'identificatore y e il valore 6
- l'identificatore z non è legato nell'ambiente
- Astrattamente un ambiente è una funzione di tipo
Ide → Value + Unbound
- L'uso della costante **Unbound** permette di rendere la funzione totale

Unbound = Valore per quelle associazioni che non sono nell'ambiente

Nell'esempio sopre ad esempio : z → Unbound

Notazione

- Dato un ambiente env: Ide → Value + Unbound
- env(x) denota il valore v associato a x nell'ambiente oppure il valore speciale **Unbound**
- env[v/x] indica l'ambiente così definito
 - env[v/x](y) = v se y = x
 - env[v/x](y) = env(y) se y != x
- Esempio: se env = {x -> 25, y -> 7} allora
env[5/x] = {x -> 5, y -> 7}

IMPLEMENTAZIONE AMBIENTE

```
let emptyenv = []
(* the empty environment *)

let rec lookup env x =
  match env with
  | []           -> failwith ("not found")
  | (y, v)::r -> if x = y then v else lookup r x
```

Tema 2 : Regole di valutazione

$$\frac{env(x) = v}{env \triangleright Var\ x \Rightarrow v}$$

eval (Var x) env -> lookup env x

$$\frac{env \triangleright e1 \Rightarrow v1 \quad env \triangleright e2 \Rightarrow v2}{env \triangleright \text{Prim}(+, e1, e2) \Rightarrow v1 + v2}$$

Valutare e1 in env

eval Prim("+", e1, e2) env ->
eval e1 env + eval e2 env

Tema 3 :

Interprete per semplici
espressioni intere

(* la valutazione è parametrica rispetto a env *)

```
let rec eval e (env : (string * int) list) : int =
  match e with
  | CstI i           -> i
  | Var x            -> lookup env x
  | Prim("+", e1, e2) -> eval e1 env + eval e2 env
  | Prim("*", e1, e2) -> eval e1 env * eval e2 env
  | Prim("-", e1, e2) -> eval e1 env - eval e2 env
  | Prim _             -> failwith "unknown primitive"
```

Il problema è che questo interprete non considera la creazione di nuovi bindings

Esecuzione con dichiarazioni

Passo 1: Sintassi Assesta

```
type expr =
| CstI of int
| Var of string
| Let of string * expr * expr
| Prim of string * expr * expr
```

Esempio

```
Let("z", CstI 17, Prim("+", Var "z", Var "z"))
```

Passo 2

Regola del Let

$$\frac{env \triangleright erhs \Rightarrow xval \quad env[xval/x] \triangleright ebody \Rightarrow v}{env \triangleright \text{Let } x = erhs \text{ in } ebody \Rightarrow v}$$

```
eval (Let(x, erhs, ebody)) env ->
let xval = eval erhs env in
let env1 = (x, xval) :: env in
eval ebody env1
```

- Si valuta **erhs** nell'ambiente corrente ottenendo **xval**
- Si valuta **ebody** nell'ambiente esteso con il legame tra **x** e **xval** ottenendo il valore **v**]→
- La valutazione del "let" nell'ambiente corrente produce il valore **v**

valuto ebody
con xval
sostituito
a x

Passo 3

Interprete per espressioni con dichiarazioni

```
let rec eval e (env : (string * int) list) : int =
match e with
| CstI i           -> i
| Var x            -> lookup env x
| Let(x, erhs, ebody) ->
let xval = eval erhs env in
let env1 = (x, xval) :: env in
eval ebody env1
| Prim("+", e1, e2) -> eval e1 env + eval e2 env
| Prim("*", e1, e2) -> eval e1 env * eval e2 env
| Prim("-", e1, e2) -> eval e1 env - eval e2 env
| Prim _              -> failwith "unknown primitive"
```

OSS: l'interprete ogni volta che valuta un binding deve fare un'operazione di lookup sull'ambiente.
Questo ha costo $O(n)$

IDEA DI OTTIMIZZAZIONE:

trasformare la ricerca in $O(1)$

Per fare ciò utilizzo un piccolo compilatore che associa ad ogni binding dell'ambiente un indice di accesso.
Questo viene applicato tramite le seguenti idee:
Conto il numero di let che mi servono per raggiungere la variabile cercata.

INDICE VAR = numero let
che si attraversano
per raggiungerla

Esempio:

```
Let("z", CstI 17,  
    Let("y", CstI 25,  
        Prim("+", Var "z", Var "y")))
```



COMPILAZIONE

```
Let(CstI 17,  
    Let(CstI 25,  
        Prim("+", Var 1, Var 0)))
```

yindice

$z = 1$



vener raggiunto
dopo un solo let

yindice $y = 0$

↓
vener raggiunta dopo
∅ let

OTTIMIZZAZIONE:

OSS: Ambiente a run-time è una int list

Linguaggio in Intermedio (compiuto)

TARGET EXPRESSION

```
type texpr = (* target expression *)
| TCstI of int
| TVar of int (* indice a run time *)
| TLet of texpr * texpr (* erhs e ebody *)
| TPrim of string * texpr * texpr
```

Compilazione in codice intermedio

```
(* Compila Expr in Texpr. Usa lista di identificatori *)
let rec tcomp e (cenv : string list) : texpr =
  match e with
  | CstI i -> TCstI i
  | Var x -> TVar (getindex cenv x)
  | Let(x, erhs, ebody) ->
    let cenvl = x :: cenv in
    TLet(tcomp erhs cenv, tcomp ebody cenvl)
  | Prim(ope, e1, e2) ->
    TPrim(ope, tcomp e1 cenv, tcomp e2 cenv)

let rec getindex cenv x =
  match cenv with
  | [] -> failwith("Variable not found")
  | y::yr -> if x=y then 0 else 1 + getindex yr x
```

Interpretazione in codice intermedio

```
(* Ambiente a run time e' una lista di interi *)
open list

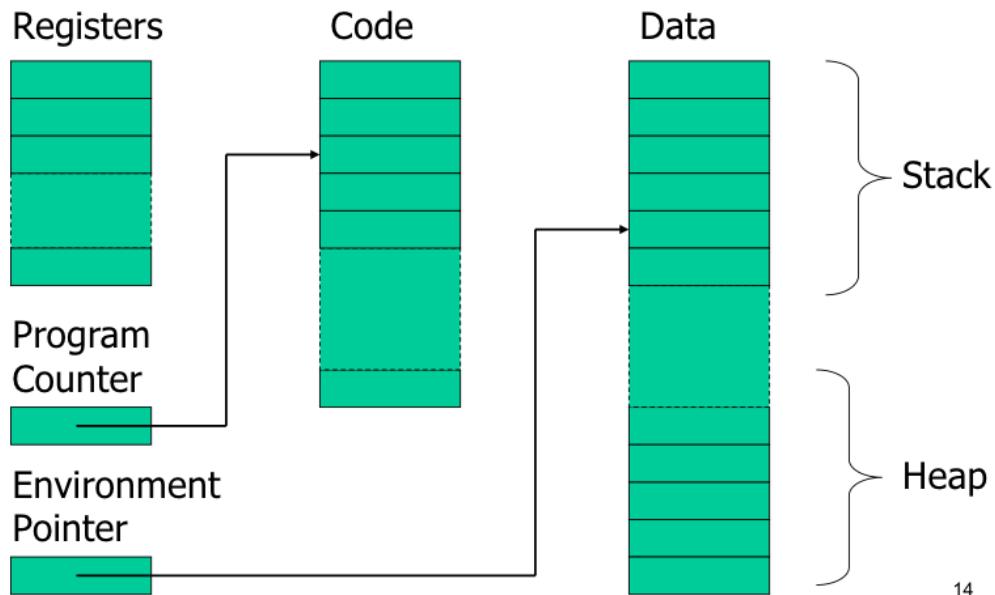
let rec teval (e : texpr) (renv : int list) : int =
  match e with
  | TCstI i -> i
  | TVar n -> nth renv n
  | TLet(erhs, ebody) ->
    let xval = teval erhs renv in
    let renv1 = xval :: renv in
    teval ebody renv1
  | TPrim("+", e1, e2) -> teval e1 renv + teval e2 renv
  | TPrim("*", e1, e2) -> teval e1 renv * teval e2 renv
  | TPrim("-", e1, e2) -> teval e1 renv - teval e2 renv
  | TPrim _ -> failwith("unknown primitive")
```

48

Codice Intermedio

Codice intermedio

- Rappresentare il programma sorgente in un codice intermedio è una tecnica che permette di dominare la complessità della implementazione di un linguaggio di programmazione
- La rappresentazione in codice intermedio permette di effettuare numerose ottimizzazioni sul codice (nel nostro caso, l'eliminazione dei nomi a run-time)
- Esempi
 - Java bytecode: codice intermedio della JVM
 - Microsoft Common Intermediate Language: codice intermedio .NET



14

Subroutine (procedura / Funzione): Perito di codice compilato al quale sono associati:

- Una cella destinata a contenere i punti di ritorno relativi alle chiamate
- Alcune celle destinate a contenere i valori di eventuali parametri
- L'ambiente locale è statico

Cosa è un vero sottoprogramma

- **Astrazione procedurale** (operazioni)
 - astrazione di una sequenza di istruzioni
 - astrazione via parametrizzazione
- **Luogo di controllo** per la gestione dell'ambiente e della memoria
 - Aspetto interessante dei linguaggi, intorno al quale ruotano decisioni semantiche importanti
 - binding: statico o dinamico

Mechanisms Call di un sottoprogramma

- Chiamante
 - 1) o crea una istanza del record di attivazione
 - 2) o salva lo stato dell'unità corrente di esecuzione (*salva lo stato nello stack*)
 - 3) o effettua il passaggio dei parametri
 - 4) o inserisce il punto di ritorno (*indirizzo dove ritornare dopo il chiamata*)
 - 5) o trasferisce il controllo al chiamato

- Chiamato (prologo)
 - 6) o salva il valore corrente di Environment Pointer (EP) e lo memorizza nel link dinamico
 - 7) o definisce il nuovo valore di EP
 - 8) o alloca le variabili locali

*→ Salva l'indirizzo dell'EP per poterlo ripristinare
(indirizzo base dello stack prima della chiamata)*

Mechanism Return di un sotto programma

- Chiamato (epilogo)
 - o eventuale passaggio di valori (dipende dalla modalità di passaggio dei parametri - lo vedremo dopo)
 - o il valore calcolato dalla funzione viene trasferito al chiamante
 - o ripristina le informazioni di controllo (il vecchio valore di EP salvato come link dinamico) *Ripristina lo stack*
 - o ripristina lo stato di esecuzione del chiamante *Ripristina PC*
 - o trasferisce il controllo al chiamante

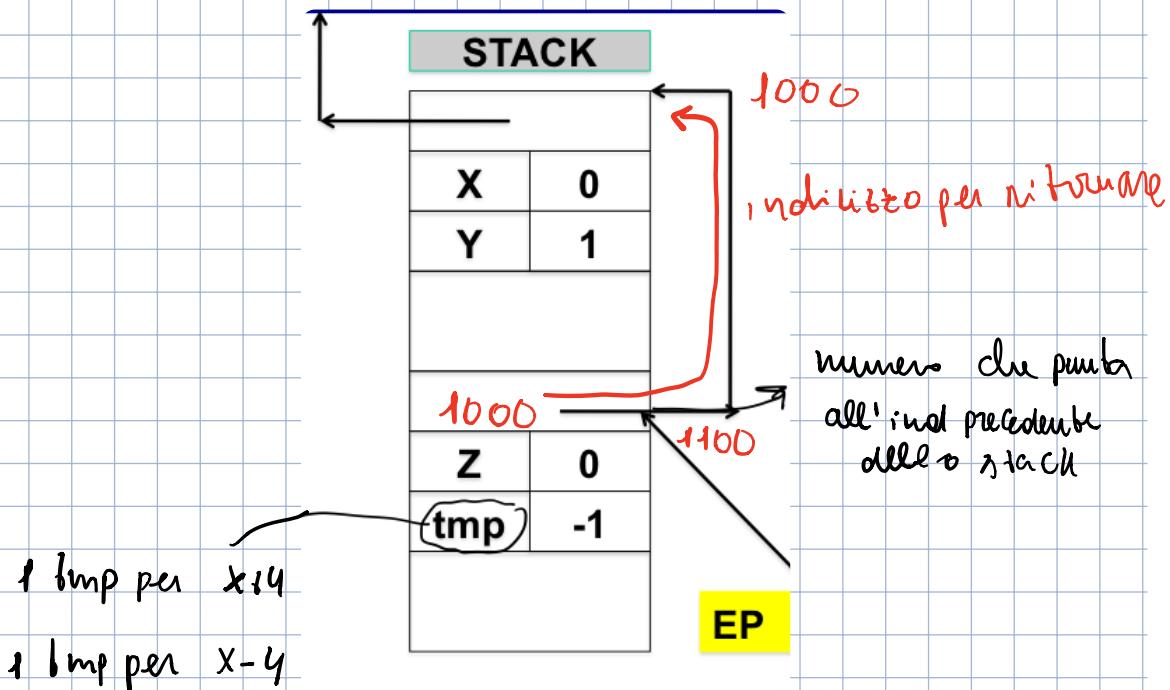
1) **Blocchi:** procedure senza nome e senza parametri:

Per ogni blocco viene memorizzato un record di attivazione (AR) sullo stack, il quale punta all'indirizzo precedente sullo stack e contiene le var con i valori create nel blocco.
contiene lmp per i calcoli che richiedono la val di une espressione

Esempio:

{ int $x = 0$
 int $y = x + 1$ } viene creato AR per x e y , e messo nello stack
 { int $z = (x+y) * (x-y);$ } viene creato AR per z e messo
 sullo stack
 } ;

}



Quindi ogni AR è composto così:

Control link
Puntatore di catena dinamica
Variabili locali
Risultati intermedi

- yn Generale
 - **Control link**
 - puntatore (indirizzo base) a AR precedente nello stack
 - **Push AR**
 - 1) il valore di EP diviene il valore del control link del nuovo AR
 - 2) modifica EP a puntare al nuovo AR
 - **Pop record off stack**
 - il valore del nuovo EP viene ottenuto seguendo il control link
- Quando entra in un blocco
- Quando esco da un blocco

Scoping: Numero di variabili visibili ad un certo istante del codice

- **Static Scope:** Riferimenti non locali si risolvono nel più vicino blocco esterno
- **Dynamic Scope:** Riferimenti non locali si risolvono nell'AR precedente sullo stack

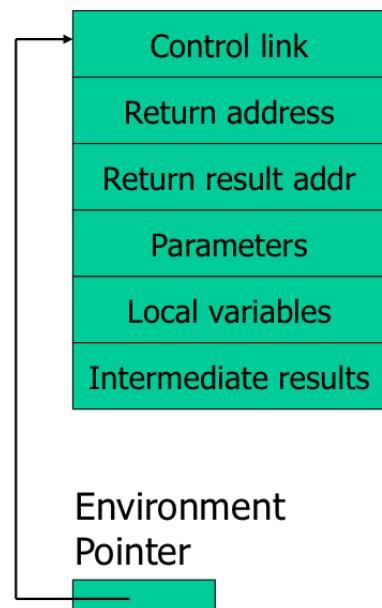
Nei blocchi soho le stessa cosa

2) Funzioni e procedure

- parametri

Record di Attivazione contiene:

- Indirizzo di ritorno
- Variabili locali / risultati intermedi
- Valore restituito
- Indirizzo per il valore restituito al momento del ritorno



→ Indirizzo intuizione da seguire dopo il return

→ Dove memorizzare il risultato

→ Parametri della funzione

per valore (valvo
contenuto)

Copiere il valore del parametro
attuale nello spazio previsto nel record
di attivazione

no aliasing

per riferimento (salvo
riferimenti)

Copiare il valore dell'
indirizzi nello Record
di attivazione

↳
Aliasing

Scopri o ↳

```

function f (x) =
{ x = x+1; return x; }
var y = 0;
println (f(y)+y);

```

Cosa stampa nei due casi?

y	0
f(y)	
Control link	
Return result addr	

y	0
f(y)	
Control link	
Return result addr	

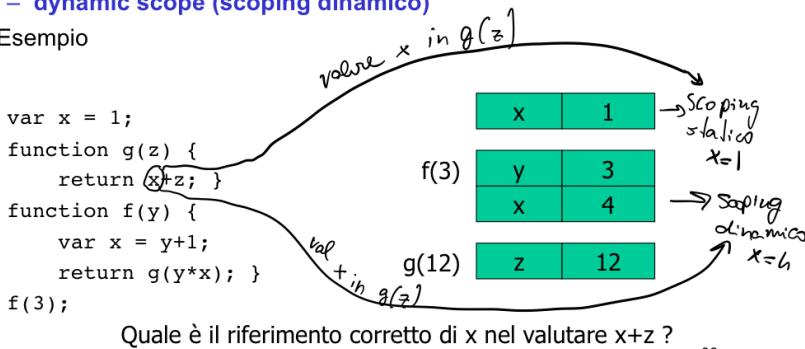
x	0
---	---

35

Variabili non local:

- Due alternative
 - static scope (scoping statico)**
 - dynamic scope (scoping dinamico)**

Esempio



36

- Ambito statico dei blocchi (rel. dal compilatore)
- AR

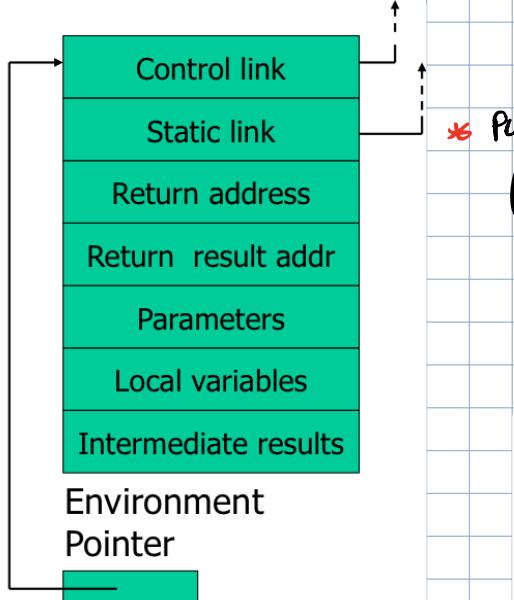
INFO
A
eseguzione

Scoping statico:

Si estende AR con static link

viene determinato dal chiamante
il link statico del chiamante

* Puntatore al AR del blocco dove la funzione è stata dichiarata
(riferimento non locale => trovare istanza AR dove il riferimento non locale è stato dichiarato)

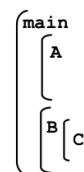


```

int x;
void A() {
    x = x+1;
}
void B() {
    int x;
    void C(int y) {
        int x;
        x = y+2; A();
    }
    x = 0; A(); C(3);
}
x = 10;
B();

```

main	x	12
B	SL	0
A	SL	
C	SL	
A	SL	



C punta a
B perché è
stato creato
in B

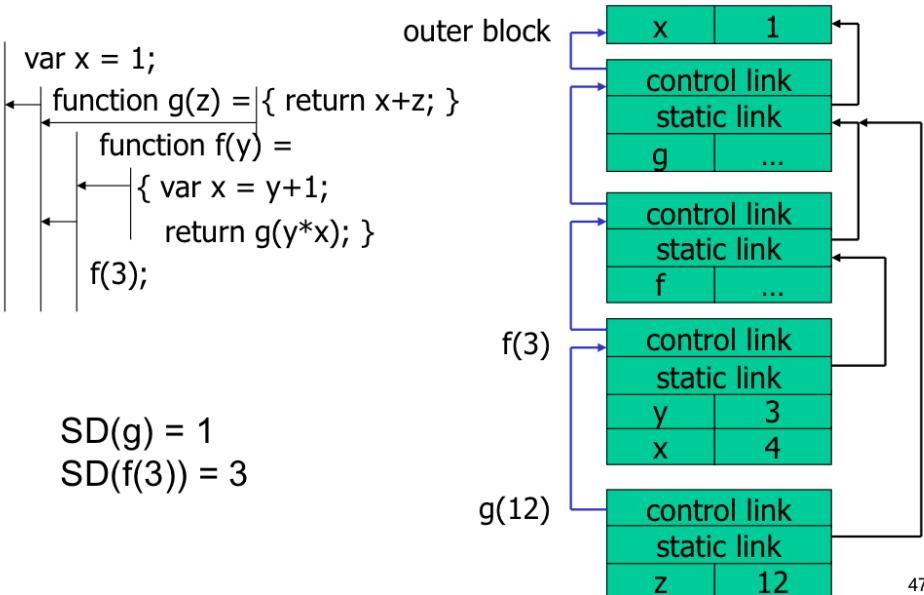
Static Depth = Profondità statica delle dichiarazioni

```

Main {           -- SD = 0
  A {           -- SD = 1
    B {           -- SD = 2
      } B
    } A
  C {           -- SD = 1
    } C
} Main
  
```

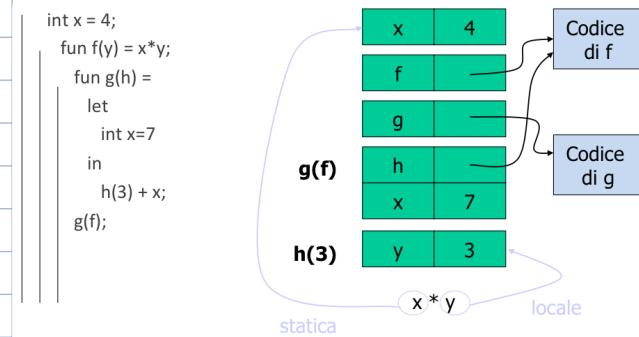
Proprietà:

- Le regole dello scoping statico assicurano che affinché il chiamato sia visibile si deve trovare in un blocco esterno che include il blocco del chiamante: *il chiamato deve essere dichiarato prima del chiamante.*
- Questo implica che l'AR che contiene la dichiarazione del chiamato è già presente sullo stack
- Assumano che
 - SD(Chiamante) = n**
 - SD(Chiamato) = m**
 - distanza statica tra chiamante e chiamato **n-m**
 - il chiamante deve fare **n-m** passi lungo la sua catena statica per definire il valore del puntatore della catena statica del chiamato



Parametri Funzionali : Nei linguaggi funzionali le funzioni tipicamente sono

Valori esprimibili



Come si determina?

51

Chiusure :

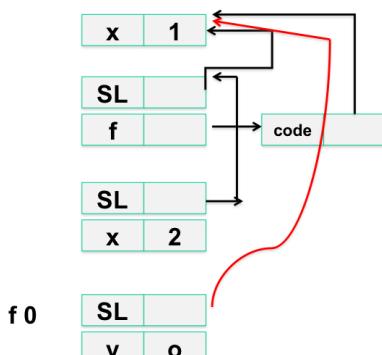
- Il valore di una funzione trasmessa come parametro è una coppia denominata **chiusura**
 - $\text{closure} = \langle \text{env_dichiarazione}, \text{codice_funzione} \rangle$
- Quando il parametro formale (funzionale) viene invocato
 - si alloca sullo stack l'AR della funzione
 - si mette come valore del **puntatore di catena statica** il puntatore a **env_dichiarazione**

Ocaml

Funzioni e chiusure

```

let x = 1;;
let f y = y + x;;
let x = 2;;
let z = f 0;;
  
```

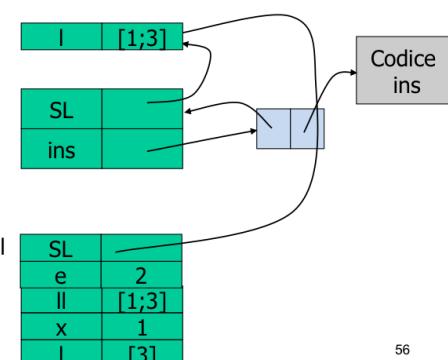


55

```

let l = [1;3];
let rec ins e ll = match ll with
| [] -> [e]
| x :: l -> if e < x then e :: x :: l
              else x :: ins e l;;
let l1 = ins 2 l
  
```

Ricorsione



56

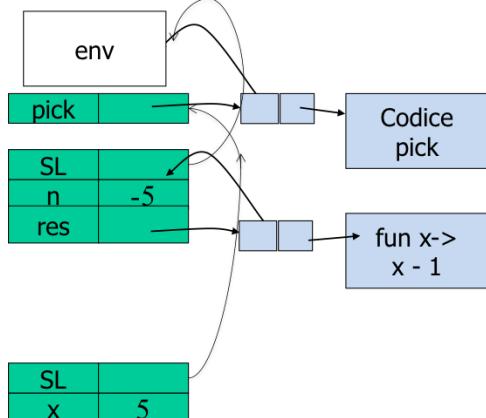
Funzioni come risultato:

esempio

- Funzione che restituisce come valore una nuova funzione
 - bisogna congelare l'ambiente dove la funzione è "dichiarata"
- Esempio


```
function compose(f, g)
  { return function(x) { return g(f(x)) } };
```
- Funzione "dichiarata" dinamicamente
 - la funzione può avere variabili non locali
 - valore restituito è una chiusura `(env, code)`
 - **attenzione:** l'AR cui punta `env` non può essere distrutto finché la funzione può essere usata (**retention**)

```
::
let pick n =
if n > 0 then (fun x -> x + 1)
else (fun x -> x - 1)
let g = (pick -5);
g 6;;
```



Risultato 5

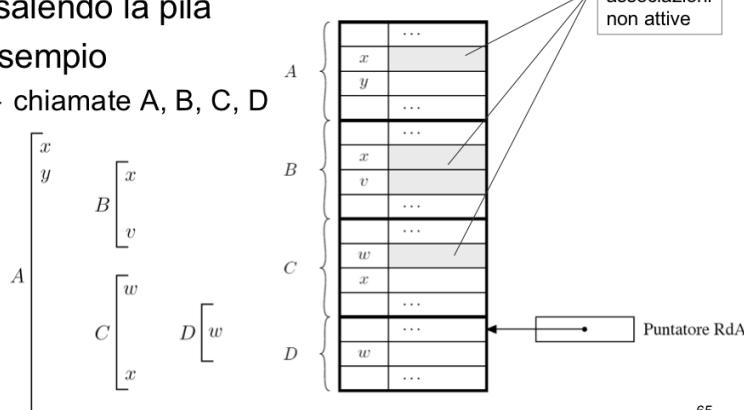
Scope dinamico: l'associazione nomi - oggetti dipende:

- Dal flusso di controllo a run-time
- Ordine con il quale i sotto programmi sono chiamati



Implementazione ovvia

- Ricerca per nome risalendo la pila
- Esempio
 - chiamate A, B, C, D



IMPLEMENTATIONE AR :

- La strutturazione dei vari campi del record di attivazione cambia a seconda del linguaggio e dell'implementazione
- Gli identificatori generalmente non vengono memorizzati nell'AR (se il linguaggio ha controllo statico dei tipi) ma sono sostituiti dal compilatore con un indirizzo relativo (offset) rispetto a una posizione fissa dell'AR

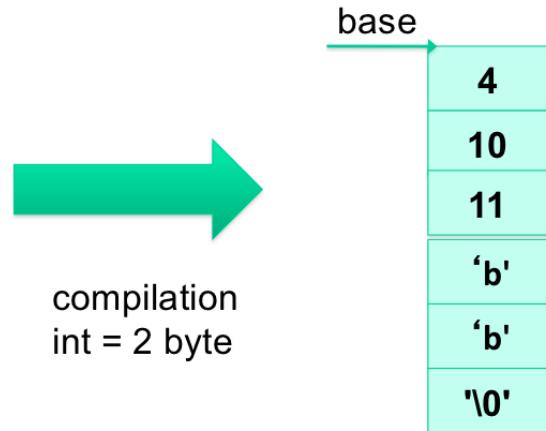
Esempio:



Calcolo offset

```
{ int x;
  int arr[2];
  char * s;
  x = 4;
  arr[0] = 10;
  arr[1] = 11;
  s = "bb";
}
```

x	4
arr[0]	10
arr[1]	11
s[0]	'b'
s[1]	'b'
s[2]	'\0'



$$\text{access}(x) = \text{base}$$

$$\text{access}(arr[1]) = \text{base} + 4\text{byte}$$

69

Allocazione di array :

Dimensione fissa

– calcolo offset immediato a tempo di compilazione

```
void foo() {
  int a;
  int b[10];
  int c;
}
```

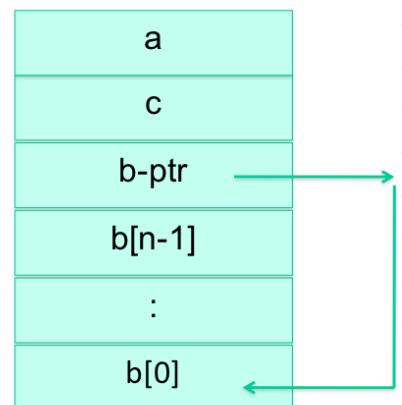
a
b[9]
:
b[0]
c

Dimensione Variabile

– calcolo offset a tempo di compilazione

```
void foo(int n) {
  int a;
  int b[n];
  int c;
}
```

ponendo una dimensione limite



In brevissimo

- Ambiente non locale con scope statico
 - il numero di passi che a tempo di esecuzione vanno fatti lungo la catena statica per trovare l'associazione (non locale) per l'identificatore "x" è uguale alla differenza fra le profondità di annidamento del blocco nel quale "x" è dichiarato e quello in cui è usato
- Ogni riferimento a un identificatore *ide* nel codice può essere staticamente tradotto in una coppia (m,n) di numeri interi
 - m è la differenza fra le profondità di nesting dei blocchi (0 se *ide* si trova nell'ambiente locale)
 - n è la posizione relativa – offset – (partendo da 0) della dichiarazione di *ide* fra quelle contenute nel blocco

74



Valutazione

- Efficienza nella rappresentazione
 - l'accesso diventa efficiente (non c'è più ricerca per nome)
- Si può economizzare nella rappresentazione degli ambienti locali che non necessitano più di memorizzare i nomi

Linguaggio funzionale didattico

```

type ide = string
type exp =
  | CstInt of int
  | CstTrue
  | CstFalse
  | Times of exp * exp
  | Sum of exp * exp
  | Sub of exp * exp
  | Eq of exp * exp
  | Iszero of exp
  | Or of exp * exp
  | And of exp * exp
  | Not of exp
  | Den of ide
  | Ifthenelse of exp * exp * exp
  | Let of ide * exp * exp (* Dichiarazione di ide: modifica ambiente *)
  | Fun of ide list * exp (* Astrazione di funzione *)
  | Appl of exp * exp list (* Applicazione di funzione *)

```