

Appunti di

PROGRAMMAZIONE 2

2018/19

Tommaso Macchioni

LEZIONE 3

Oggetto

Una struttura dati con diversi campi:

- 1) Le variabili di istanza (Stato dell'oggetto)
- 2) Alcuni metodi (capacità computazionale dell'oggetto/comportamento)

A differenza dei record le variabili di istanza sono private mentre tendenzialmente tutti i metodi sono pubblici.

Java oltre ad essere Object Oriented è Class Based: La classe è lo strumento che mi permette di dichiarare il tipo degli oggetti che appartengono a quella classe. Inoltre la classe mi darà un meccanismo per creare un oggetto che appartiene a quella classe. (E' un template).

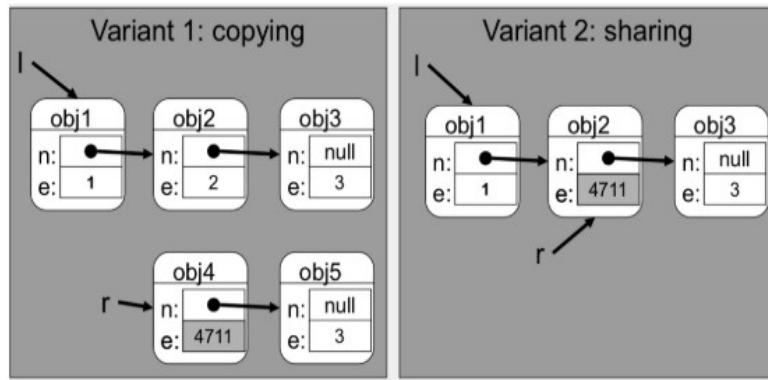
- La definizione di una classe specifica
 - Tipo e valori iniziali dello stato locale degli oggetti (le variabili di istanza)
 - Insieme delle operazioni che possono essere eseguite (metodi)
 - Costruttori (uno o più): codice che deve essere eseguito al momento della creazione di un oggetto
- Ogni oggetto è una istanza di una classe e può (opzionalmente) implementare una interfaccia

L'idea è di NON rendere visibile all'esterno l'implementazione dell'oggetto (Information Hiding).

Cosa significa programmare ad oggetti? Un programma ad oggetti è un insieme di oggetti che cooperano tra di loro (un oggetto utilizza i metodi di un altro oggetto) tramite un "messaggio": nome metodo e parametri su cui operare.

Ci sono due modalità per l'assegnamento:

`r = l.rest()`



Potremmo aver bisogno di una copia perché si vuole continuare a fare in modo che le informazioni di stato rimangano nell'individuo 'l' (sicurezza).

Interfacce

Le interfacce definiscono i metodi che un oggetto mette a disposizione.

Un oggetto che implementa un'interfaccia deve obbligatoriamente implementare i metodi dichiarati nell'interfaccia.

Le interfacce possono essere multiple.

Metodo statico

E' un metodo associato alla classe e non alle variabili d'istanza. Vale per tutte le istanze della classe senza dover conoscere il valore delle variabili d'istanza specifiche.

In una classe è semplicemente una funzione che preso un parametro restituisce un risultato.

Infatti nel main non bisogna dichiarare un oggetto della classe ma basta scrivere "NomeClasse"."nomeMetodo"("parametri");

Sono utilizzati quindi per implementare quelle funzionalità che non dipendono dallo stato dell'oggetto.

Variabili statiche

Sono variabili *static* che sono accessibili tramite il nome della classe.

NON possono essere inizializzate nel costruttore della classe dato che non possono essere associate a istanze della classe.

E' accedibile solo da metodi statici.

E' globale a tutte le istanze di quella determinata classe.

Riferimento

E' un valore che identifica un determinato oggetto.

```
public IntList(ConsCell s) {  
    start = s;  
}
```

Nella variabile *start* memorizziamo il riferimento all'oggetto *s* e NON una copia (NON è un puntatore).

In soldoni un riferimento è un puntatore all'oggetto ma mentre in C si espone la natura “indirizzo” (con l'aritmetica dei puntatori) in Java non si dice come sono implementati i riferimenti: sono solo valori che identificano univocamente oggetti.

Stringhe

Le stringhe sono oggetti IMMUTABILI (come in Ocaml).

Con “+” denotiamo l'operatore di concatenazione di stringhe.

Java ha due operatori per testare l'uguaglianza:

- 1) **Pointer equality** (*o1 == o2*) TRUE se i due oggetti denotano lo stesso riferimento.
- 2) **Deep equality** (*o1.equals(o2)*) TRUE se le variabili *o1* e *o2* denotano due oggetti identici.

Esempio:

- `String s = new String("test"); String t = new String("test")`
- `s.equals("test") --> true`
- `s == "test" --> false`
- `s == t --> false`

```
String s1 = "Java";  
  
String s2 = "Java";
```

```
s1.equals(s2) // true...  
perché?  
s1==s2 // true... perché?
```

Perché ho fatto un assegnamento con un letterale e quindi puntano allo stesso indirizzo.

```
String str1 = new String("Java");  
String str2 = new String("Java");  
  
str1.equals(str2) // true: stesso  
contenuto  
str1==str2 // false: oggetti differenti
```

Perché chiamando il costruttore si crea di fatto un nuovo oggetto in memoria e str1 e str2 punteranno a due indirizzi diversi.

LEZIONE 4

Asserzioni (Java Assert):

Permettono di testare, in un determinato punto del programma, i valori dello stato.

- assert **booleanExpression**;
- valuta l'espressione booleana
- se l'espressione viene valutata true il comando non modifica lo stato, se l'espressione restituisce false viene sollevata l'eccezione **AssertionError**

- **assert booleanExpression : expression;**
- In questo caso, se l'espressione booleana viene valutata false la seconda espressione è utilizzata all'interno del messaggio di **AssertionError**
- La seconda espressione può avere qualunque tipo con l'eccezione del tipo **void**

(Vanno abilitate)

Abstract Stack Macchine (NON è la Java Virtual Machine)

E' un modello che ci permette di comprendere cosa significa eseguire un programma Java, cosa significa avere memoria dinamica e legare le variabili a riferimenti ecc.

Ha tre componenti:

- 1) **Workspace** per la memorizzazione dei programmi in esecuzione
- 2) **Stack** ci permette di gestire i binding (quando dichiariamo una variabile in un metodo la mettiamo nello stack per avere un modo per legare il nome di una variabile al suo valore che può essere nella memoria statica o dinamica)
- 3) **Heap** per la gestione della memoria dinamica

Ereditarietà

Una super-classe generalizza una sotto-classe fornendo un comportamento che è condiviso dalle sotto-classi.

B è un sotto-tipo di A: "ogni oggetto che soddisfa l'interfaccia di B soddisfa anche l'interfaccia A"

- 1) una istanza del sotto-tipo soddisfa le proprietà del super-tipo
- 2) una istanza del sotto-tipo può avere maggiori vincoli di quella del super-tipo

Permette il "polifomismo del sottotipo": tramite l'ereditarietà una variabile può assumere tipi (di classi) differenti.

PRINCIPIO DI SOSTITUZIONE DI LISKOV:

“Se una classe B extends una classe A vorrei poter utilizzare la classe B in tutti i posti in cui poteva essere utilizzata A senza che chi utilizza A si accorga che sta utilizzando B.”

(In soldoni: Il mondo conosce A ma ora mando B e siccome B può essere utilizzato in tutti i posti in cui poteva essere utilizzato A, ha anche il tipo di A. Allora può usare la soluzione più efficiente nuova ma non ho dovuto modificare nessun riferimento).

Se A ha variabili e metodi *private* non sono visibili comunque a B.

Con *protected* invece sono visibili nella gerarchia di raffinamento (e quindi in B).

Il metodo costruttore non viene ereditato ma siccome il metodo costruttore della sotto-classe deve accedere anche alle variabili di istanza della super-classe allora utilizziamo il metodo *super* (“parametri del costruttore della classe padre”).

All'interno di un metodo o di un costruttore, la parola chiave *this* permette di riferire l'oggetto corrente ovvero l'oggetto il cui metodo o costruttore viene chiamato (è utilizzato anche come costruttore隐式的).

Upcasting & Downcasting

B sotto-classe di A

Downcasting: un oggetto di tipo A può essere legato ad una variabile di tipo B (In Java è esplicito)

Upcasting: un oggetto di tipo B può essere legato ad una variabile di tipo A (In Java è implicito)

```
class Vehicle { ... }; A
class Car extends Vehicle; // Car sotto-tipo di Vehicle B
Vehicle v = (Vehicle) new Car(); // upcasting
Car c = (Car) new Vehicle(); // downcasting
```

Tipo statito e dinamico

Statico

E' il tipo di una classe o interfaccia che può essere determinato dal compilatore.

E' l'espressione che descrive il valore calcolato in base alla struttura statica del programma.

Dinamico

E' il tipo della classe di cui l'oggetto è istanza.

In un upcasting nell'esempio di prima il tipo statico è Vehicle mentre il tipo dinamico è Car.

```
class A {  
    void f() {  
        System.out.println("A f");  
    }  
}  
  
class B1 extends A {  
    void f() {  
        System.out.println("B1 f");  
    }  
}  
  
B1 b = new B1();  
b.f(); B1 f  
A a = b;  
a.f(); B1 f  
a = new A();  
a.f(); A f
```

PR2

LEZIONE 5

Static dispatch (C++)

L'operazione di individuazione del metodo da invocare è fatta staticamente (a tempo di compilazione).

Dynamic dispatch (Java)

L'informazione di quale metodo devo andare a invocare quando sono in esecuzione dipende dal tipo corrente associato all'oggetto (quindi dal tipo dinamico e non statico).

Tabella dei metodi (versione intuitiva)

Ogni oggetto nello heap ha un puntatore alla tabella dei metodi del suo tipo dinamico.

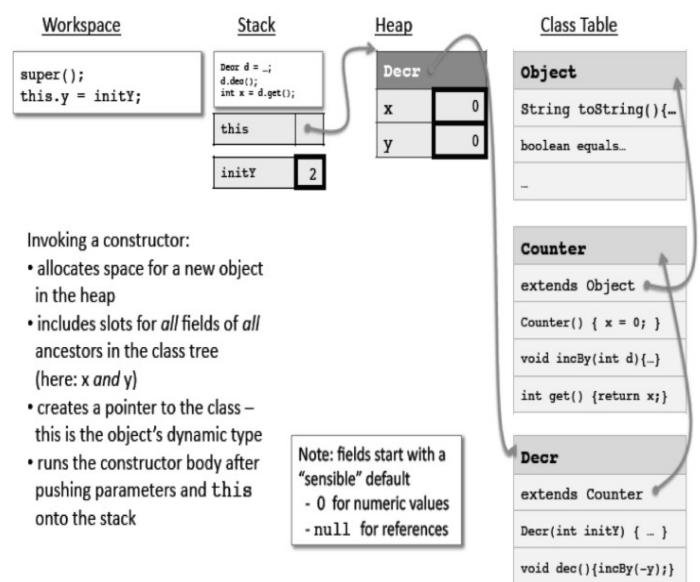
Estendiamo la ASM (Abstract Stack Machine) con la tabella dei metodi.

Contiene il codice dei metodi definiti nella classe e tutte le componenti statiche definite nella classe stessa.

Contiene un puntatore alla classe padre in modo da utilizzare i metodi della classe padre.

I metodi stanno in memoria dinamica perché sono condivisi tra tutte le istanze dell'oggetto (stesso codice). E' impensabile che per 50 istanze di una classe abbia in memoria il solito codice di un metodo moltiplicato 50.

Se non è già presente, l'invocazione del metodo costruttore determina l'allocazione sullo heap della tabella dei metodi associata alla classe dell'oggetto creato.



Dispatch

esempio:

L'invocazione del metodo “*o.m()*” utilizza il puntatore alla tabella dei metodi per effettuare l'operazione di dispatch: ricerca sulla gerarchia dell'oggetto a partire dalla tabella dei metodi associata al tipo dinamico dell'oggetto *o*.

Eccezioni

E' un modo per gestire gli errori a tempo di esecuzione che io programmatore mi aspetto possano presentarsi.

Le eccezioni sono dunque quella componente linguistica all'interno del linguaggio di programmazione che mi permette di fare due cose:

- 1) *Sollevare (throwing)* un'eccezione: determinare quali sono i punti di uscita di emergenza di un programma dicendo “se arriva a questo punto io genero un'eccezione”. Dunque il programma standard termina e viene mandata in esecuzione l'eccezione.
- 2) *Catturare (catching)* un'eccezione: significa programmare le azioni da eseguire per gestire il comportamento anomalo.

Sono utili perché

- 1) il compilatore in generale non è in grado di determinare tutti i possibili errori che possono succedere
- 2) *Separation Of Concerns*: permettono di identificare chiaramente qual è il codice globale da quello di errore

Throw (Sollevare)

Il linguaggio prevede questa primitiva specifica per dichiarare e programmare il modo in cui le eccezioni sono sollevate

“ if (myObj.equals(null)) **throw new NullPointerException()** ”

Throw richiede come argomento un oggetto che abbia come tipo un qualunque sotto-tipo di Throwable (classe che contiene tutti i tipi di errore e di eccezioni).

Try-Catch (Catturare)

```
try {  
    // codice che può sollevare diverse eccezioni  
}  
catch (IOException e) {  
    // gestione IOException  
}  
catch (ClassNotFoundException e) {  
    // gestione ClassNotFoundException  
}  
finally {  
    // codice di clean-up che viene sempre eseguito  
}
```

Formato dei messaggi

[exception class]:
[additional description of exception] at
[class].[method]([file]: [line number])

- java.lang.ArrayIndexOutOfBoundsException: 3 at
ArrayExceptionExample.main(ArrayExceptionExample.java:6)
- **Exception Class?**
 - java.lang.ArrayIndexOutOfBoundsException
- **Quale indice dell' array (additional information)?**
 - 3
- **Quale metodo solleva l'eccezione?**
 - ArrayExceptionExample.main
- **Quale file contiene il metodo?**
 - ArrayExceptionExample.java
- **Quale linea del file solleva l'eccezione?**
 - 6

Eccezioni Checked e Unchecked

Unchecked

Sono quelle eccezioni che sono tipicamente di sistema che vengono sollevate quando c'è un comportamento anomalo e comportano la terminazione prematura del programma.

Sono Unchecked perché non ho un meccanismo di recovery dell'eccezione ovvero il programma le segnala quando vengono e l'effetto netto di questa operazione è che il programma si blocca dandomi tutta l'informazione relativa all'errore (Formato dei messaggi sopra).

Checked

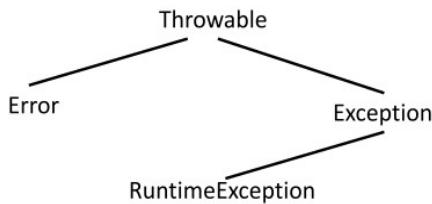
Se un metodo può contenere del codice che può generare un'eccezione allora si deve esplicitare nella dichiarazione del metodo:

“public void myMethod **throws Exception { ... }** “

Queste sono eccezioni Checked: rappresentano eccezioni che sono considerate “non fatali” all'esecuzione del programma.

Richiede obbligatoriamente la gestione dell'eccezione tramite clausa try-catch.

La gerarchia di tipi per le eccezioni



- Se un nuovo tipo di eccezione estende la classe `Exception`, l'eccezione è checked
- Se un nuovo tipo di eccezione estende la classe `RuntimeException`, l'eccezione è unchecked

Le eccezioni non sono necessariamente errori sono delle situazioni anomali che si possono presentare. I comportamenti che sono errore ad un certo livello potrebbero non esserlo ad un livello successivo.

Dobbiamo ridurre al minimo la struttura del programma in modo da avere delle informazioni sui comportamenti cioè usare le eccezioni come meccanismo di strutturazione del programma per riuscire ad individuare possibili potenziali errori durante l'esecuzione del programma stesso.

Checked vs Unchecked

Checked

- Controllo maggiore: perché devono essere monitorate obbligatoriamente (il compilatore è dunque coinvolto).

Unchecked

- Difficili da gestire: non so da dove viene l'eccezione

LEZIONE 6

Java Generics

Si vorrebbe all'interno di un linguaggio di programmazione la possibilità di definire delle caratteristiche comuni per dei dati parametrizzando rispetto al tipo.

```
interface ListOfNumbers {
    boolean add(Number elt);
    Number get(int index);
}
interface ListOfIntegers {
    boolean add(Integer elt);
    Integer get(int index);
}
... e ListOfStrings e ...
// Indispensabile astrarre sui tipi
interface List<E> {
    boolean add(E n);
    E get(int index);
}
```

Usiamo i tipi!!!

`List<Integer>`
`List<Number>`
`List<String>`
`List<List<String>>`
...

PH2 2018-19

Variabili di tipo

```
class NewSet<T> implements Set<T> {
    // non-null, contains no duplicates
    // ...
    List<T> theRep;
    T lastItemInserted;
    ...
}
```

Dichiarazione

Utilizzo

Dichiarare generici

```
class Name<TypeVar1, ..., TypeVarN> {...}

interface Name<TypeVar1, ..., TypeVarN> {...}



- convenzioni standard
  - T per Type, E per Element,
  - K per Key, V per Value, ...

```

Istanziare una classe generica significa fornire un valore di tipo

`Name<Type1, ..., TypeN>`

Posso anche dare un limite superiore degli oggetti che posso avere per gli oggetti concreti che poi mi andranno a sostituire il parametro (E, T, K o V):

```
interface List1<E extends Object> {...}
interface List2<E extends Number> {...}

List1<Date> // OK, Date è un sottotipo di Object
List2<Date> // compile-time error, Date non è
// sottotipo di Number
```

Vincoli di tipo

1) Upper Bound

<TypeVar extends SuperType>

Va bene il SuperType come istanziazione della variabile di tipo o uno dei sotto-tipi.

2) Multiple upper bounds

<TypeVar extends ClassA & InterfB & InterfC & ... >

Si mette un limite un limite superiore ai vincoli di tipo. Come limite superiore abbiamo l'oggetto concreto A ma che implementi le interfacce B,C ecc.

3) Lower Bound

<TypeVar super SubType>

Va bene il sotto-tipo o uno qualunque dei super-tipi.

NB: Covarianza, Controvarianza e Invarianza

Un operatore F è *covariante* se $F<T'> \leq F<T>$ con $T' \leq T$

Un operatore F è *controvariante* se $F<T> \leq F<T'>$ con $T' \leq T$

Un operatore è *invariante* se non è né covariante né controvariante.

Regole di Java

Se Type2 è sotto-tipo di Type 3 allora Type1<Type2> NON è sotto-tipo di Type1<Type3>

Formalmente: la nozione di sotto-tipo usata in Java è invariante per le classi generiche.

Esempi

Integer è un sotto-tipo di Number

ArrayList<E> è sotto-tipo di List<E> che a sua volta è sotto-tipo di Collection<E>

List<Integer> non è un sotto-tipo di List<Number>

Limiti

Tutti i tipi generici sono trasformati in *Object* nel processo di compilazione per un motivo di backward-compatibility con il codice vecchio e quindi a runtime tutte le istanziazioni generiche hanno lo stesso tipo:

```
List<String> lst1 = new ArrayList<String>();  
List<String> lst2 = new ArrayList<String>();  
lst1.getClass() == lst2.getClass() // TRUE
```

Wildcard ?

La wildcard è una variabile di tipo anonima. Si utilizzano quando so che ho dei vincoli di sotto o super-tipo e che voglio usare un tipo generico ma non ne so il nome e lo voglio usare soltanto una volta.

L'unica cosa che la wildcard mi garantisce è l'unicità del tipo.

Esempio:

```
interface Set<E> {  
    // Aggiunge a this tutti gli elementi di c  
    // (che non appartengono a this)  
    void addAll(??? c);  
}
```

Quale è il miglior tipo per il parametro formale?

- Il più ampio possibile ...
- ... che permette di avere implementazioni corrette

```
interface Set<E> {  
    void addAll(Collection<? extends E> c);  
}
```

- maggiormente flessibile rispetto a
`void addAll(Collection<E> c);`
- espressiva come
`<T extends E> void addAll(Collection<T> c);`

Quando si usa?

C'è una regola del pollice che si chiama *Producer Extends, Consumer Super* :

- Si usa “ ? extends T “ nei casi in cui si vogliono ottenere dei valori da un *produttore di valori* (addAll precedente ad esempio).
- Si usa “ ? super T “ nei casi in cui si vogliono inserire valori in un *consumatore*.
- NON si usa (basta T) quando si ottengono e si producono valori.

Java Array

In Java se Type1 è sotto-tipo di Type2 allora Type1[] è sotto-tipo di Type2[]

LEZIONE 6

La progettazione di un Software segue varie fasi:

- *Specifica*: cosa si deve realizzare
- *Designa*: come si decompone il sistema in moduli in modo da ridurre la complessità.
- *Implementazione*: scrivere il codice che soddisfa le specifiche.
- *Testing*: Determinare sistematicamente eventuali errori.
- *Debugging*: Risolvere in modo sistematico gli errori
- *Manutenzione*: Come far evolvere il sistema nel tempo
- *Documentazione*: Descrivere tutte le scelte fatte e le loro motivazioni.

Specificia

Ci sono due modi distinti di osservare il comportamento di un programma:

- 1) La visione di chi **implementa**
- 2) La visione di chi lo **usa**

Le scelte di implementazione siano completamente nascoste per evitare che chi lo usa faccia delle assunzioni esplicite sui moduli che avete realizzato e che queste assunzioni possano poi provare dei problemi quando vengono messe insieme le parti. Perché se qualcuno cambia una delle due parti non si sa più se il problema è chi lo usa o chi l'ha implementato.

La specifica è un *contratto d'uso* ovvero è l'insieme dei vincoli che definiscono le mutue aspettative tra chi implementa e chi lo deve utilizzare.

Bisogna dunque identificare i vincoli del cliente senza considerare dettagli d'implementazione, evitare che ci siano dei vincoli nascosti (i vincoli che pone il cliente devono essere esplicitate), separare l'uso dall'implementazione.

L'interfaccia definisce chiaramente il contratto col cliente: i vincoli di sistema che devono essere seguiti da tutti coloro che usano l'interfaccia vista.

Interface = syntax & types.

Il difetto dell'interfaccia non permette di osservare i comportamenti e il loro effetto, e quindi è un'informazione limitata verso i clienti.

Potremmo optare per fornire al cliente il codice ma:

- numero elevato di dettagli non utili al cliente
- deve fare uno sforzo non banale: comprendere l'API
- il cliente è interessato solo a *cosa* il code fa e non a *come*.

Optiamo allora per commentare il codice ma ovviamente deve essere esplicativo, accurato e non troppo complicato.

La specifica dovrà essere una guida da una parte per l'utente per evitare che fornisca dati non corretti oppure per evitare che faccia assunzioni sulla caratteristica del servizio fornito e da un parte come guida per chi implementa per far in modo che realizzi opportunamente quello che l'utente chiedeva.

Potremmo utilizzare Javadoc, strumento nativo di Java per la scrittura delle specifiche:

- Segnatura dei metodi
- Descrizione testuale del comportamento
- `@param`: description of what gets passed in
- `@return`: description of what gets returned
- `@throws`: exceptions that may occur

Noi utilizzeremo una cosa simile ma sarà più preciso:

- **Precondition:** vincolo sulle proprietà che devono valere prima dell'invocazione del metodo.

`@requires`: descrive le condizioni di validità dei parametri e dello stato che devono essere soddisfatti e che quindi il cliente deve garantire.

- **Postcondition:** vincolo sulle proprietà che devono valere dopo l'invocazione del metodo.

`@modifies`: elenca gli oggetti che sono modificati durante l'esecuzione del metodo, gli oggetti non presenti nell'elenco NON sono modificati.

`@throws`: elenca le possibili eccezioni

`@effects`: descrive le proprietà dello stato finale

`@return`: descrive i valori restituiti dal metodo

I vantaggi:

- *astrazione sulle strutture di implementazione*: il cliente deve solo comprendere come è fatto il pattern di interazione senza comprendere i dettagli del codice (per il cliente il codice potrebbe anche non esistere).
- *È un metodo di lavoro*: scrivere prima di tutto le specifiche permette di utilizzarle come guida all'implementazione.

Programmazione difensiva

Chi implementa non delega il controllo ma si mette in condizione di controllare comunque se i dati trasmessi dal cliente sono corretti: meglio fallire che fornire un risultato non corretto.

Rafforzare una specifica

- Promettere molto di più: effects, returns, minor numero di oggetti modificati, minor numero di eccezioni.
- Chiedere di meno al cliente: rilassare la requires

(Indebolire una specifica è l'opposto)

Barriera di astrazione

Un meccanismo linguistico che ci permette di descrivere una struttura dati senza vedere il perché questa struttura dati è implementata in un certo modo.

Struttura dati mutable e immutable

Una struttura mutable significa che è modificabile, immutable altrimenti.

Specifiche delle struttura dati ADT (Abstract data type)

Immutable	Mutable
1. overview	1. overview
2. abstract state	2. abstract state
3. creators	3. creators
4. observers	4. observers
5. producers	5. producers (rari)
6. mutators	6. mutators
<ul style="list-style-type: none">• Creators: restituiscono un nuovo oggetto del ADT (Java constructors)• Producers: Operazioni (dell'ADT) che restituiscono valori• Mutators: Modificano lo stato concreto ADT• Observers: Restituiscono infotmazioni sullo stato ADT	

Clausola Overview:

- Definisce se il nuovo ADT è mutable o immutable
- Definisce il modello astratto da utilizzare nella specifica del tipo di dato astratto
 - Difficile! ... ma essenziale
 - Bene usare notazioni matematiche non ambigue

Metodi Creatori, Osservatori, Produttori, Modificatori

Creatori: Sono i metodi che creano la struttura dati. (clausola @effects e/o @throws)

Osservatori: Sono metodi che mi permettono di restituire come risultati delle proprietà della struttura dati. (clausola @returns e/o @throws)

Produttori: Sono i metodi che producono un risultato. (clausola @return)

Modificatori: Sono i metodi che modificano la struttura dati e che di solito non producono valori (clausola @modifies e/o @effects)

LEZIONE 8

Quello che vogliamo ora studiare è come passare dalla specifica all'implementazione.

Dobbiamo vedere come affrontiamo, da un punto di vista metodologico, la creazione dell'implementazione di una particolare API garantendo le proprietà che abbiamo descritto nella specifica.

Gli strumenti che utilizzeremo per comprendere *che cosa fa e se fa la cosa giusta* sono l'**invariante di rappresentazione** e la **funzione di astrazione**.

Invariante di rappresentazione

Concettualmente bisogna immaginarla come le strutture dati dell'implementazione ovvero quello che occorre per implementare la struttura abbiammo in mente.

Dal nostro punto di vista, astrattamente, possiamo vederla come una funzione che prendere le variabili d'istanza che vogliamo implementare e ci restituisce true/false.

Ci dice dunque se la rappresentazione che abbiamo scelto è ben formata ovvero se rispetta i vincoli che volevamo dall'astrazione (e.g. Il vincolo procedurale sulla struttura dati "Insieme" è che non ci siano due occorrenze dello stesso elemento).

Possiamo vederla come una guida per chi deve implementare la struttura dati a partire dalla specifica data.

Funzione di astrazione

E' il meccanismo che deve portare la struttura di implementazione nella visione astratta.

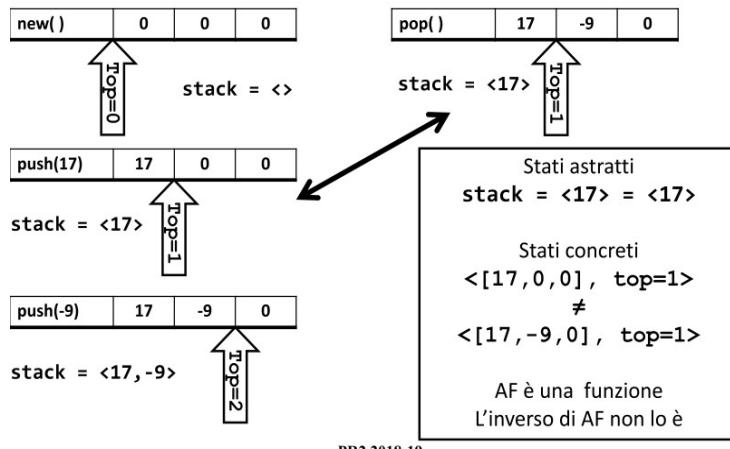
La clausola *Overview* ci diceva come potevamo astrattamente rappresentare la struttura dati che abbiammo in mente.

Quindi la funzione di astrazione è quel meccanismo che ci porta l'implementazione concreta (le strutture dati che vengono usate nell'implementazione) nella rappresentazione astratta che abbiammo in mente della struttura data.

Ci dice dunque com'è l'interfaccia di comportamento tra gli strumenti sulle strutture cheabbiamo implementato e la visione esterna e quindi ci dice se il sistema fa ciò che ci chiedeva la specifica.

Esempio: AF per Stack

75.11.734



Representation Invariant: Object → boolean

- Stabilisce se una istanza è *ben formata*
- Stabilisce l'insieme concreto dei valori dell'astrazione (ovvero quelli che sono una implementazione dei valori astratti)
- **Guida per chi implementa/modifica/verifica l'implementazione delle astrazioni: nessun oggetto deve violare rep invariant**

Abstraction Function: Object → abstract value

- Stabilisce come interpretare la struttura dati concreta della implementazione
- È definita solamente sui valori che rispettano l'invariante di rappresentazione
- **Guida per chi implementa/modifica l'astrazione:** ogni operazione deve fare "la cosa giusta" con la rappresentazione concreta

defensive programming

- Assunzione: programmare è un processo di tipo "trial and error"
- Progettare del codice in modo tale che
 - alla chiamata dei metodi
 - ✓ verifica rep invariant
 - ✓ verifica pre-condizioni
 - all'uscita del metodo
 - ✓ verifica rep invariant
 - ✓ verifica post-condizioni
- **Verificare rep invariant = verificare la presenza di errori**
- **Ragionare sul rep invariant = evitare di fare errori**

Rischio di esporre la rappresentazione

C'è il rischio di dare in mano al cliente l'accesso diretto alla struttura dati.

L'uso del *private* potrebbe non bastare ed avere un aliasing di struttura mutabili all'interno e all'esterno dell'astrazione.

Per evitare l'esposizione della rappresentazione una prima tecnica è quella di fare copie dei dati che oltrepassano la barriera dell'astrazione:

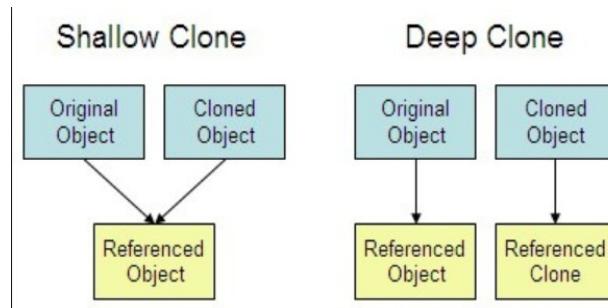
- *copy in*: parametri che diventano valori della rappresentazione

```
class Line {  
    private Point s, e;  
    public Line(Point s, Point e) {  
        this.s = new Point(s.x,s.y);  
        this.e = new Point(e.x,e.y);  
    } ...}
```

- *copy out*: risultati che sono parte dell'implementazione

```
public Point getStart( ) {  
    return new Point(this.s.x,this.s.y);  
}
```

Una shallow copy non è sufficiente a causa dell'aliasing si opta per una deep copy



Tutto questo dipende anche da che tipo di struttura dati abbiamo perché se abbiamo ad esempio String, Character, Integer ecc. sono classi immutabili e quindi sono solo leggibili e non modificabili.

Vantaggi:

- l'aliasing non è un problema
- non è necessario fare copie
- rep invariant non può essere “rotto”

Svantaggi:

- posso comunque derivare informazioni su elementi della struttura che avrei potuto non avere (e.g. Per capire se c'è un elemento o meno in una struttura dati immutabile, prendo una copia di questa, inserisco un dato nell'originale e confronto le due).

LEZIONE 12

Iteratori

E' un meccanismo che mi permette di scorrere gli elementi di una collezione "uno alla volta" senza espornere la rappresentazione.

Sono oggetti di classi che implementano la seguente interfaccia:

```
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. */  
}
```

PR2 2018-19

Con *next()* si visitano tutti gli elementi della struttura dati esattamente una volta.

E' la struttura che determina con quale metodo visitare gli elementi ovvero l'ordine nel vengono restituiti gli elementi dipende dall'implementazione dell'iteratore (e.g. Le strutture lineari come come *List* l'iteratore di default rispetta l'ordine).

Si possono attivare più iteratori simultaneamente su una collezione.

Se invoco la *remove()* senza aver chiamato prima *next()* si lancia una *IllegalStateException()*.

Se la collezione viene modificata durante l'iterazione di solito viene invocata una *ConcurrentModificationException()*.

Specifica di iteratori

Immaginiamo di dover implementare un proprio iteratore.

Principalmente dovrà avere un meccanismo che mi genera gli elementi della struttura dati (una sola volta, senza ripetizioni) e poi un meccanismo che mi permette di determinare se sono arrivato in fondo.

Esempio:

Specifiche di iteratore per IntSet

```
public class IntSet implements Iterable<Integer> {  
    // specifica standard +  
    public Iterator<Integer> iterator( );  
    // REQUIRES: this non deve essere modificato  
    // finche' il generatore e' in uso  
    // EFFECTS: ritorna un iteratore che produrrà tutti  
    // gli elementi di this (come Integers) ciascuno una  
    // sola volta, in ordine arbitrario  
}
```

- La clausola REQUIRES impone condizioni sul codice che utilizza il generatore
 - tipica degli iteratori su tipi di dati modificabili
- Dato che la classe implementa **Iterable<E>** si può usare **for-each**

Specifiche di generatore stand alone

-s. § 7.2.

```
public class Primes implements Iterator<Integer> {  
    public Iterator<Integer> iterator( )  
    // EFFECTS: ritorna un generatore che produrrà tutti  
    // i numeri primi (come Integers), ciascuno una  
    // sola volta, in ordine crescente  
}
```

- Un tipo di dato può avere anche più iteratori, quello restituito dal metodo **iterator()** è il “default”
- In questo caso il limite al numero di iterazioni deve essere imposto dall'esterno
 - il generatore può produrre infiniti elementi

Implementazione di iteratori

Possiamo dichiarare una classe all'interno di un'altra classe (inner class).

L'idea è che la inner class è il meccanismo di generazione dell'iteratore per la classe esterna.

Implementazione iteratori: Primes

```
public class Primes implements Iterable<Integer> {  
    public Iterator<Integer> iterator( ) { return new PrimeGen( ); }  
    // EFFECTS: ritorna un generatore che produrrà tutti i numeri primi  
    // (come Integers) ciascuno una sola volta, in ordine crescente  
    private static class PrimeGen implements Iterator<Integer> {  
        // class interna statica  
        private List<Integer> ps; // primi già dati  
        private int p; // prossimo candidato alla generazione  
        PrimeGen( ) { p = 2; ps = new ArrayList<Integer>( ); } // costruttore  
        public boolean hasNext( ) { return true; }  
        public Integer next( ) {  
            if (p == 2) { p = 3; ps.add(2); return new Integer(2); }  
            for (int n = p; true; n = n + 2)  
                for (int i = 0; i < ps.size( ); i++) {  
                    int el = ps.get(i);  
                    if (n%el == 0) break; // non è primo  
                    if (el*el > n) { ps.add(n); p = n + 2; return n; }  
                }  
        }  
        public void remove( ) { throw new UnsupportedOperationException( ); }  
    }  
}
```

PR2 2018-19

39

Essendo classi (le inner class) devono avere un' IR (invariante di rapp.), una FA (Funzione di astrazione) e tutti i vincoli che abbiamo visto nella nostra metodologia.

Ad esempio una FA di un iteratore è la sequenza degli elementi che devono ancora essere generati.

LEZIONE 13

Gerarchie di tipi

Immaginiamo di avere:

```
Casse A {  
    metodo m (A a) { ... }  
}
```

```
Classe B extends A {
```

```
}
```

Assunto che “ *B b = new B()* ” possiamo chiamare “ *m(b)* ” a prescindere ?

No perché se *A* fosse un insieme infinito e *B* un insieme finito, e il metodo *m* fosse una metoda di *insert*, è chiaro che potremmo saturare la classe *B* e quindi avere un comportamento anomalo.

Implementazioni multiple



- Il tipo superiore della gerarchia definisce una famiglia di tipi tale per cui
 - tutti i membri hanno esattamente gli stessi metodi e la stessa semantica che forniscono l'implementazione di tutti i metodi astratti, in accordo con le specifiche del super-tipo
 - gli oggetti dei sotto-tipi vengono visti dall'esterno come oggetti dell'unico super-tipo
 - dall'esterno si vedono solo i costruttori dei sotto-tipi

Possiamo modificare le implementazioni dei metodi.

Il dynamic dispatch assicura che raggiungeremo il metodo corretto quando chiamato.

Subtyping vs. subclassing



Sottotipi (Subtype) — una nozione semantica

- B è un sottotipo di A se e solo un oggetto di tipo B può stare al posto di (mascherarsi come) A in un qualunque contesto
- I comportamenti osservabili di B sono un sottoinsieme di quelli di A

Ereditarietà (subclass) — è una nozione di implementazione

- Permette la fattorizzazione e il riuso del codice
- Una nuova classe è creata per differenza dalla classe padre

Java mette assieme le due nozioni:

- In Java ogni sottoclasse diventa anche un sottotipo
- Ma non è un true subtype

Principio di sostituzione

Quello che noi vogliamo avere è che un oggetto del sotto-tipo possa sostituire un oggetto del super-tipo senza influenzare comportamenti di altri programmi che lo utilizzano.

Detto in altre parole: se B è un sotto-tipo di A, allora B può sempre sostituire A (ogni istanza di B può stare in tutti i contesti dove può stare una istanza di A).

Quindi i sotto-tipi devono fornire i soliti comportamenti del super-tipo.

Il principio di sostituzione ci permette di dire quando una classe B è un sottotipo della classe A e rispetta questa proprietà di composizionalità, ovvero di dire quando posso sostituire un sotto-tipo B in tutti i posti in cui stava il super-tipo A.

Non è automatico perché il compilatore non lo controlla.

B può rafforzare le proprietà o introdurne delle nuove (nuovi metodi ad esempio che preservano la RapInv di A) e/o B NON può indebolire una specifica (non è possibile rimuovere metodi, non può avere un numero maggiori di comportamenti che aveva A).

Principio di sostituzione



- Devono essere supportate
 - la regola della segnatura
 - ✓ gli oggetti del sotto-tipo devono avere tutti i metodi del super-tipo
 - ✓ le segnature dei metodi del sotto-tipo devono essere compatibili con le segnature dei corrispondenti metodi del super-tipo
 - la regola dei metodi
 - ✓ le chiamate dei metodi del sotto-tipo devono comportarsi come le chiamate dei corrispondenti metodi del super-tipo
 - la regola delle proprietà
 - ✓ il sotto-tipo deve preservare tutte le proprietà che possono essere provate sugli oggetti del super-tipo
- NB: le regole riguardano la semantica!

NB: la *extends* in Java è una struttura sintattica.

La nozione di sotto-classe in Java è solo una buona approssimazione di sotto-tipo.

Regola della segnatura

- Il sotto-tipo B deve implementare tutti i metodi del super-tipo A.
- Metodi: parametri in input: il tipo degli argomenti dei metodi in A può essere sostituito da super-tipi di quegli argomenti in B.
esempio: in A ho $m(Integer i)$ e in B ho $m(Number i)$
perché sto indebolendo le pre-condizioni.

Ho un comportamento di *controvariante* perché sto scendendo nella gerarchia ma il metodo prende cose più ampie (quindi va contro il verso di discesa).

In Java è *invariante*: l'argomento deve essere dello stesso tipo.

- Metodi: risultati in output: il tipo del risultato dei metodi in A può essere un super-tipo del tipo del risultato dei metodi in B (*covariante*).

Esempio: in A ho *Number m(...)* e in B ho *Integer m(...)*

In Java mantiene la *covarianza*.

- Non posso aggiungere eccezioni altrimenti uno si accorge che sta usando il sotto-tipo.
- Le eccezioni esistenti in metodi di A possono essere sostituite da sotto-tipi

Regola dei metodi

Pre-condizione: proprietà sul tipo degli argomenti (input) @requires

Post-condizione: proprietà sul tipo del risultato (output) @effects

Si confrontano i due metodi: quello nel super-tipo e quello riscritto nel sotto-tipo (per ogni metodo)

- **Regola della pre-condizione:**

$\text{pre}_{\text{super}} \Rightarrow \text{pre}_{\text{sub}}$

La pre-condizione viene indebolita (da super a sub): significa che gli argomenti che soddisfano la clausola *requires* del metodo del super-tipo sono un sotto-insieme degli argomenti che soddisfano la clausola *requires* del sotto-tipo.

Esempio: un metodo in IntSet

```
public void addZero( )
    // REQUIRES: this non e' vuoto
    // EFFECTS: aggiunge 0 a this
```

potrebbe essere ridefinito in un sotto-tipo

```
public void addZero( )
    // EFFECTS: aggiunge 0 a this
```

- **Regola della post-condizione:**

$\text{pre}_{\text{super}} \&& \text{post}_{\text{sub}} \Rightarrow \text{post}_{\text{super}}$

La post-condizione viene rafforzata : gli stati risultati dall'esecuzione di metodo *m* in B sono un sotto-insieme degli stati risultati eseguendo *m* soltanto in A.

In termini di comportamento significa che se fornisco un parametro che soddisfa la proprietà della pre-condizione (*requires*) di un metodo del super-tipo, quando restituisce il controllo al chiamante, il risultato verifica la proprietà della post-condizione (*effects*) del sotto-tipo allora deve valere anche la proprietà della post-condizione (*effects*) del super-tipo.

Sto restringendo i valori del risultato.

Esempio: un metodo in IntSet

```
public void addZero( )
    // REQUIRES: this non e' vuoto
    // EFFECTS: aggiunge 0 a this
```

potrebbe essere ridefinito in un sotto-tipo

```
public void addZero( )
    // EFFECTS: se this non e' vuoto aggiunge 0 a this
    // altrimenti aggiunge 1 a this
```

(this non è vuoto) && [(this non è vuoto aggiunge 0 → aggiunge 0 a this)
OR (aggiunge 1 a this)] => aggiunge 0 a this ??? SI!!!

Regola delle proprietà

Le proprietà di oggetti basati sulla specifica del super-tipo è ancora valida in oggetti del sotto-tipo.

Le proprietà sono dichiarate nella *overview* della specifica del super-tipo.

Alcuni tipi proprietà:

- **Proprietà invarianti:** proprietà sempre vere per gli oggetti di quel tipo.

Esempio:

Un tipo *FatSet* con oggetti mai vuoti.

//OVERVIEW: A *FatSet* è un insieme mutabile di interi di grandezza almeno 1

con un metodo:

public void removeNonEmpty (int x)

//EFFECTS: se *x* è in *this* e *this* contiene altri elementi allora rimuovi *x* da *this*

Se adesso considerassimo un tipo *ThisSet* (sotto-tipo di *FatSet*) con il seguente metodo:

public void remove (int x)

//MODIFIES: *this*

//EFFECTS: rimuove *x* da *this*

ThinSet non può essere un sotto-tipo “legale” di *FatSet* perché i metodi extra possono

causare che un oggetto diventi vuoto, perciò non preserva l'invariante (*overview*) del super-tipo.

- **Proprietà di evoluzione:** abbiamo visto solo quella della *immutability* (*immutabilità*).

Esempio:

Un tipo *SimpleSet* che ha solo due metodi *insert* e *isIn*.

//OVERVIEW: Un *SimpleSet* è un insieme mutabile di interi.

// Gli oggetti *SimpleSet* possono solo crescere nel tempo e non restringersi.

Un tipo *IntSet* non può essere un suo sotto-tipo perché il metodo *remove* causerebbe un restringimento.

LEZIONE 17

Programmazione funzionale

Nella programmazione imperativa (C) abbiamo uno stato iniziale globale che è la memoria (legami tra nomi identificatori e valori) e opero per assegnamenti e quando il programma termina vado a vedere la memoria e scopro cosa fa un programma.

Ovvero un programma è definito in termini dello stato del programma stato (le associazioni presenti in memoria).

Questo vale anche per la programmazione ad oggetti.

Nella programmazione funzionale invece, il risultato osservabile è un valore che uno si aspetta di avere in base al problema che deve risolvere.

Quindi da un certo punto di vista un programma OCaml è un'espressione, un'espressione in Ocaml denota un valore e il passo di calcolo è come si riduce questa espressione in modo tale da ottenere un valore.

Sinistassi

Espressioni

Le regole di buon comportamento hanno due aspetti:

1. Type checking : il sistema di Ocaml garantisce che un'espressione che è tipata correttamente non genererà errori di tipo a runtime quando viene eseguito. Significa che il sistema garantisce a chi programma che le entità usate nel programma sono usate correttamente rispetto la loro definizione di tipo altrimenti genera errore a tempo di compilazione. (strongly type)
2. Regole di esecuzione : ovvero nel meccanismo della macchina astratta che è in grado di eseguire Ocaml come se fosse un linguaggio macchina ci dicono quali sono le regole di controllo che guidano l'esecuzione di un programma.

Valori

Un valore è un'espressione che non deve essere valutata ulteriormente ovvero qualcosa di già valutato. Esempi:

34 è un valore di tipo int

34+17 NON è un valore ma è un'espressione che deve ancora essere valutata

- La notazione $\langle \text{exp} \rangle \Rightarrow \langle \text{val} \rangle$ indica che la espressione $\langle \text{exp} \rangle$ quando valutata calcola il valore $\langle \text{val} \rangle$

$3 \Rightarrow 3$ (valori di base)

$3+4 \Rightarrow 7$

$2*(4+5) \Rightarrow 18$

$\text{eval}(e) = v$ metanotazione per $e \Rightarrow v$

“ eval(e) = v ” significa se applico l'interprete delle espressioni ad e , mi restituisce il valore v.

Let

Let è la nozione di blocco in Ocaml.

Esempio: *let x = e1 in let y = e2 in e3*

```
{  
  x = e1;  
  {  
    y = e2;  
    e3;  
  }  
}
```

“let x = e1 in e2 “ significa che bisogna coerentemente il legame tra x ed e1 nella valutazione di e2.

In realtà è più corretto dire che sintatticamente si associa un'espressione ma in realtà quando uno va a fare il calcolo quello che ci interessa è il valore associato.

- **let x = e1 in e2**
- **Regola di valutazione**
 - **eval(e1) = v1**
 - sostituire il valore v1 per tutte le occorrenze di x in e2 ottenendo l'espressione e2'
 - **subst(e2, x, v1) = e2'**
 - **eval(e2') = v**
 - **eval(let x = e1 in e2) = v**

Possiamo vederlo anche così:

$$\frac{\text{eval}(e1) = v1 \ subst(e2, x, v1) = e2' \ eval(e2') = v}{\text{eval}(\text{let } x = e1 \text{ in } e2) = v}$$

Ovviamente sostituisco solo quelle libere perché ad esempio un'espressione:

```
# let x = "aaa" in let x = 3 in x+x;;  
- : int = 6
```

Esempio:

- **eval(let x = 1 + 4 in x * 3)**
 - **eval(1 + 4) = 5**
- **eval(let x = 5 in x*3)**
 - **subst(x * 3, x, 5) = 5 * 3**
- **eval(5 * 3) = 15**
- **eval(let x = 1 + 4 in x * 3) = 15**

```
let f (x : int) : int =  
    let y = x * 10 in  
        y * y;;
```

```
f 5;;  
- : int = 2500
```

Infatti:

*eval(let x = 5 in let y = *10 in y*y)*

eval(5) = 5

*subst(let y = x*10 in y*y,x,5) = let y = 5*10 in y*y*

*eval(let y = 5*10 in y*y)*

*eval(5*10)=50*

*subst(y*y,y,50) = 50*50*

*eval(50*50) = 2500*

Dichiarazione funzione

La valutazione di una dichiarazione di una funzione è la funzione stessa => le funzioni sono valori.

Questo è il **meccanismo di ordine superiore** perché se le funzioni sono valori allora posso utilizzare una funzione in tutti quei posti in cui io mi aspetto un valore (ovviamente di quel tipo): come parametro attuale di un'altra funzione o come risultato di una funzione.

In C o in Java non avviene perché dovremmo avere un funzione/metodo che prende come parametro un'altra funzione/metodo.

Al limite in Java posso passare come parametro una classe che a sua volta ha un metodo.

Applicazione

- **eval(e0 e1 ... en) = v' se**
 - **eval(e0) = let f x1 ... xn = e**
 - **eval(e1) = v1 ... eval(en)= vn**
 - **subst(e, x1, ..., xn, v1, ..., vn) = e'**
 - **eval(e') = v'**

1. Meccanismo di passaggio per parametri = per valore (il parametro attuale deve essere un valore prima della chiamata della funzione).
2. Ogni funzione deve produrre una funzione
3. Posso scrivere una espressione che mi restituisce una funzione ma questa potrebbe essere anche una funzione anonima ovvero potrei scrivere una funzione a cui non ho dato un nome nell'ambiente (Scrivere una funzione che da come risultato un'altra funzione).

Esempi:

- let twice ((f : int -> int), (x : int)) : int = f (f x)
- let quad (x : int) : int = twice (double, x)
- let fourth (x : int) : int = twice (square, x)
- *twice*
 - *higher-order function*: una funzione da funzioni ad altri valori

List

- [] la lista vuota (nil derivato dal LISP)
- **e1::e2** inserisce l'elemento **e1** in testa alla lista **e2**
 - (:: = LISP cons)
- **[e1; e2; ...; en]** notazione sintattica per la lista
e1::e2::...::en::[]

Esempi:

- **let rec sum xs = match xs with**
 - | [] -> 0
 - | h::t -> h + sum t
- **let rec concat ss = match ss with**
 - | [] -> ""
 - | s::ss' -> s ^ (concat ss')
- **let rec append lst1 lst2 = match lst1 with**
 - | [] -> lst2
 - | h::t -> h::(append t lst2)

Pattern Matching

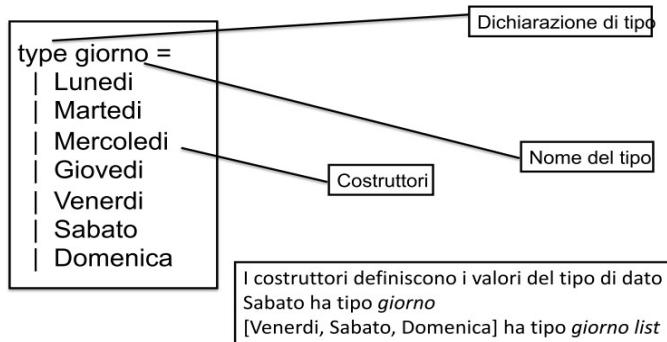
- **match e with**
 - | p1 -> e1
 - | p2 -> e2
 - | ...
 - | pn -> en
- Le espressioni **pi** si chiamano *pattern*
- Il pattern **h::t** “match” una qualsiasi lista con almeno un elemento, e inoltre ha l’effetto di legare quell’elemento alla variabile **h** e la lista rimanente alla variabile **t**
- **match [1; 2; 3] with**
 - | [] -> 0
 - | h::t -> h (* restituisce il valore 1 *)
- **match [1; 2; 3] with**
 - | [] -> []
 - | h::t -> t (* restituisce il valore [2; 3] *)
- Il pattern **a::[]** “match” tutte le liste con esattamente un elemento
- Il pattern **a::b** “match” tutte le liste con almeno un elemento
- Il pattern **a::b::[]** “match” tutte le liste con esattamente due elementi
- Il pattern **a::b::c::d** “match” tutte le liste con almeno tre elementi

Iteratori

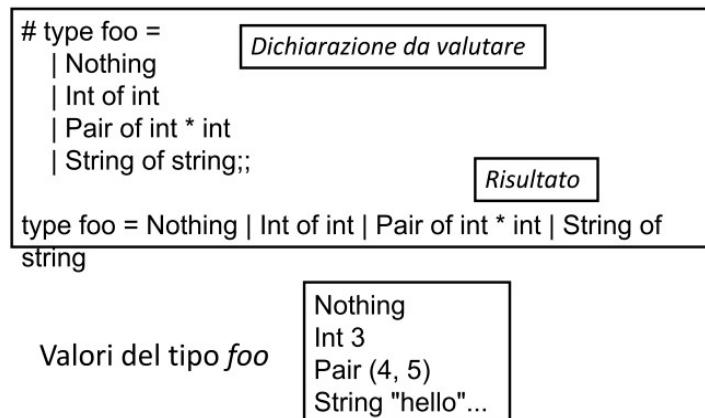
- **let rec map f = function**
 - | [] -> []
 - | x::xs -> (f x)::(map f xs)
- **map : ('a -> 'b) -> 'a list -> 'b list**

Definire nuovi tipi

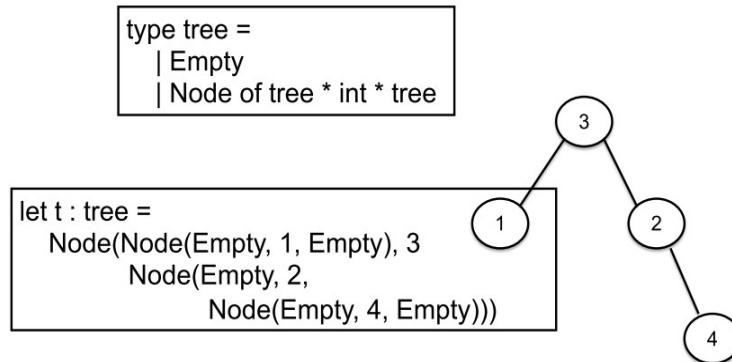
Ocaml permette al programmatore di definire *nuovi* tipi di dato



I costruttori possono trasportare “valori”



Alberi binari in OCaml



Ricerca in un albero



```
let rec contains (t : tree) (n : int) : bool =
begin match t with
| Empty -> false
| Node(lt, x, rt) -> x = n ||
(contains lt n) ||
(contains rt n)
end
```

La funzione contains effettua una ricerca del valore n sull’albero t
Caso peggiore: deve visitare tutto l’albero

Generici

```
let rec zip (l1 : 'a list) (l2 : 'b list) : ('a * 'b) list =
begin match (l1,l2) with
| (h1::t1, h2::t2) -> (h1, h2)::(zip t1 t2)
| _ -> []
end
```

La funzione opera su tipi generici multipli
(da 'a list e 'b list verso ('a * 'b) list)

```
zip [1;2;3] ["a";"b";"c"] = [(1,"a");(2,"b");(3,"c")] : (int * string) list
```

Interfacce in Ocaml : moduli

OCaml: Set Interface



```
module type Set = sig
  type 'a set
  val empty : 'a set
  val add : 'a -> 'a set -> 'a set
  val remove : 'a -> 'a set -> 'a set
  val list_to_set : 'a list -> 'a set
  val member : 'a -> 'a set -> bool
  val elements : 'a set -> 'a list
end
```

Idea (solita): fornire diverse funzionalità nascondendo la loro implementazione

Module type (in un file .mli) per dichiarare un TdA
sig ... end racchiudono una segnatura, che definisce il TdA e le operazioni
val: nome dei valori che devono essere definiti e dei loro tipi

Per implementarli:

Moduli in OCaml



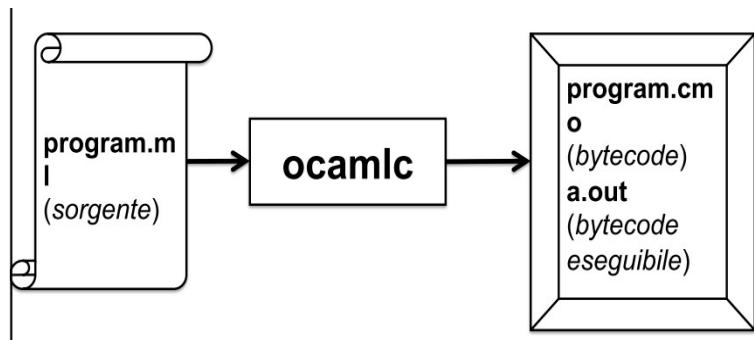
Nome del modulo

Signature che deve essere implementata

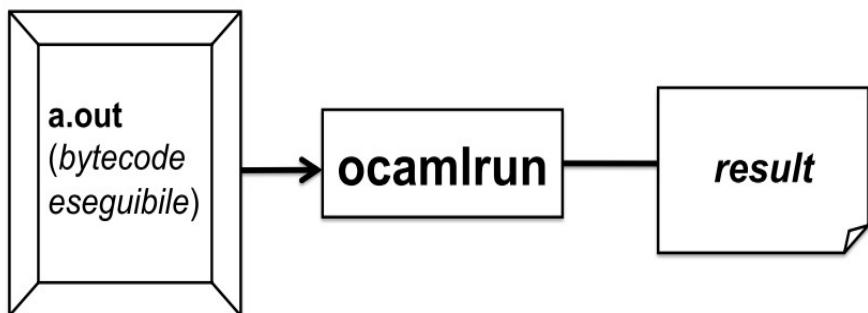
```
module Myset : Set = struct ...
(* implementations of all the operations *)
:
end
```

Compilare programmi Ocaml

Ocaml ha un'implementazione mista perchè abbiamo una prima parte di compilazione (→ codice intermedio) poi dobbiamo usare l'interprete dell'eseguibile bytecode per poter poi mandare il programma in esecuzione.



Eseguire bytecode Ocaml



LEZIONE 18

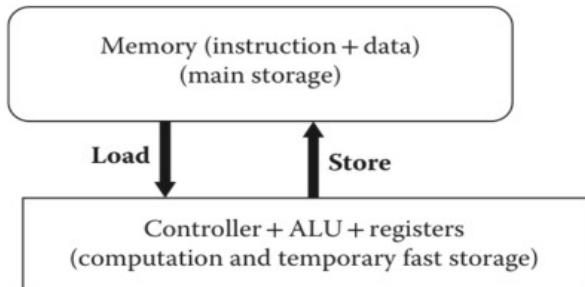
Modello di Von Neumann

Il modello di Von Neumann è alla base della struttura dei computer attuali.

Ha due componenti principali:

- **Memoria:** dove sono memorizzati i dati e i programmi (visti sempre come dati).
- **Unità centrale di elaborazione (CPU):** ha il compito di eseguire i programmi memorizzati in memoria.

Una macchina specializzata sia digitale che elettromeccanica fa solo una sola cosa e dunque NON programmabile. Si contrappone quindi alla macchina basata sul modello di Von Neumann.



Il Program Counter è un registro che contiene l'indirizzo della prossima istruzione da prelevare in memoria e da far eseguire al processore.

- **Architetture “tradizionali”:** il ciclo viene eseguito in modo sequenziale, ogni istruzione viene elaborata prima dell'avvio dell'istruzione successiva.
- **Architetture “moderne”:** l'istruzione successiva inizia ad essere elaborata prima che l'istruzione precedente sia terminata (architetture pipeline).

I linguaggi di programmazione devono essere realizzati su di una macchina hardware e quindi realizzati tenendo conto di questa.

I primi linguaggi vengono realizzati proprio sullo strato della macchina hardware che veniva fornita.

Il mondo però si è evoluto e si parla di livelli di astrazione.

Questo permette di caratterizzare un aspetto (livello) specifico.

Facendo un altro balzo, quello che vogliamo è che vorremmo far girare

un'applicazione Java su diversi sistemi operativi (Windows, Linux, Macintosh).

Se però il sistema operativo è differente ho bisogno di qualcosa che mi permetta di i diversi servizi.

Grazie alla macchina virtuale ho dei meccanismi d'interfaccia definiti sulla macchina hardware che mi permettono di vedere quello che sta sotto in base ad una opportuna interfaccia d'uso.

Vedo quindi uno strato di astrazione che mi virtualizza, in termini del software, le caratteristiche della macchina sottostante.

L'interfaccia d'uso verso il livello applicativo quindi è sempre lo stesso su sistemi operativi differenti.

Java Virtual Machine

Un'applicazione Java tramite il Javac (Java Compiler) viene compilata in bytecode.

Quindi a livello di astrazione delle interfacce, Javac conosce un insieme di istruzioni (quelle del bytecode) e quindi riesce a compilare un'applicazione.

La JVM è la macchina virtuale per l'esecuzione di programmi Java che virtualizza l'hardware sotto e mette a disposizione quelle primitive di bytecode che servono per compilare il sorgente Java.

Il passaggio moderno della realizzazione dei linguaggi di programmazione non è quello di andare sull'hardware ma sulla macchina virtuale che dipende dalle caratteristiche del sistema.

La JVM è stata implementata sotto Windows, Linux, Macintosh ecc. quindi possiamo compilare un programma su Linux e mandarlo in esecuzione (il bytecode ottentuno) su Windows senza doverlo ricompilare perchè su Windows ci sarà l'implementazione della JVM.

La JVM è scritta in C++ e compilata nell'istruzione set della macchina su cui girerà qualunque esso sia.

Come si implementano i linguaggi di programmazione

Abbiamo 3 soluzioni:

(Se siamo negli anni '60 il codice macchina sarà esattamente il codice assembler, se siamo negli anni 2010 sarà qualcosa di misto, ora è una macchina virtuale)

Compilazione

(il codice sorgente viene compilato/tradotto nel codice della macchina virtuale)

Interpretazione

(non lo compila ma lo esegue direttamente istruzione per istruzione)

Esiste dunque un linguaggio di implementazione dell'interprete.

Misto

3 aspetti significativi

All'interno di questo corso, nei linguaggi di programmazione, sono importanti 3 cose:

- Astrazione sui dati
- Astrazione sul controllo
- Astrazione di modularità

Quello che ci interessa è presentare quali sono le primitive linguistiche che i linguaggi di programmazione mettono a disposizione per affrontare questi 3 aspetti in modo da comprendere le scelte di progetto che uno ha di fronte quando deve realizzare un linguaggio di programmazione oppure per un programmatore per comprendere al meglio le primitive che il linguaggio mette a disposizione.

Astrazione sui dati

Una struttura dati è una struttura che mi permette di organizzare in modo coerente i dati che ho a disposizione.

Voglio un linguaggio di programmazione con coppie o record come tipi primitivi? Delle liste oppure di strutture di tipo ricorsive à la Ocaml? Puntatori esplicativi (in C) oppure i riferimenti (in Java)?

L'astrazione sui dati è quel meccanismo che mi permette di nascondere le scelte di implementazione delle strutture dati.

Astrazione sul controllo

Il controllo è quel meccanismo che a livello del linguaggio di programmazione si usa per controllare l'ordine delle istruzioni (for, while, iteratori, switch, exceptions, pattern matching).

Poi c'è anche il meccanismo, mascherando l'implementazione, che mi rompe il flusso normale di esecuzione in presenza di anomale (interruzioni).

Astrazione di modularità

Programmare in larga scala richiede di considerare molte dimensioni di design trasversali: eager vs lazy evaluation, immutable vs mutable, static vs dynamic typing.

E di decidere le caratteristiche di modularità: inserire nel path del progetto le librerie importanti, classi, ereditarietà di interfacce, information hiding ecc.

Macchine astratte

Una macchina astratta è un sistema virtuale che rappresenta il comportamento di una macchina fisica individuando precisamente l'insieme delle risorse necessarie per l'esecuzione di programmi.

Un linguaggio di programmazione di fatto è il linguaggio macchina della macchina che vogliamo realizzare.

Una macchina astratta è una collezione di strutture dati e algoritmi in grado di memorizzare ed eseguire programmi.

Le componenti principali di una macchina astratta sono:

- un interprete;
- una memoria, destinata a contenere il programma che deve essere eseguito e i dati su cui si sta operando;
- un insieme di operazioni primitive (cioè funzionalità che si assume la macchina sia in grado di fornire) utili all'elaborazione dei dati primitivi (ovvero dati sui quali la macchina astratta sa lavorare);
- un insieme di operazioni e strutture dati che gestiscono il flusso di controllo, ovvero che governano l'ordine secondo il quale le operazioni, descritte dal programma, vengono eseguite;
- un insieme di operazioni e strutture dati per il controllo del trasferimento dei dati, che si occupa di recuperare gli operandi e memorizzare i risultati delle varie istruzioni;
- un insieme di operazioni e strutture dati per la gestione della memoria.

Interprete

Dà la capacità alla macchina astratta di eseguire programmi (esegue ciclo fetch/execute)

Esempio di macchina astratta nella vita quotidiana:

La nozione di macchina astratta è ben più generica di quanto si possa credere. Infatti anche un ristorante, in un certo senso si può vedere come una macchina

astratta. I programmi "eseguiti" da tale macchina sono composti da sequenze di ordinazioni di piatti (Es.: antipasto alla marinara; linguine al pesto; pepata di cozze; macedonia; limoncello). Supponiamo di avere un ristorante in cui ci sia un cuoco specializzato per ogni piatto ed un insieme di inservienti preposti a portare a tali cuochi gli ingredienti per i loro piatti. L'insieme dei cuochi può esser visto come l'inseme delle "operazioni" della macchina. La "memoria" della nostra macchina sarà il taccuino del cameriere. L'"interprete" del ristorante può essere il cameriere stesso che, letta la prima "istruzione" la "decodifica", dicendo agli inservienti quali ingredienti portare a quale cuoco. Ovviamente anche la dispensa dovrà esser vista come parte della memoria del ristorante (la parte che memorizza gli "argomenti" delle istruzioni. Un altro cameriere provvederà a "memorizzare" sul nostro tavolo il risultato dell'esecuzione dell'istruzione.

Ovviamente certi esempi di macchine astratte necessitano di un minimo di elasticità mentale per poter esser ricondotti al nostro schema formale. Teniamo presente inoltre che in realtà quella che abbiamo fornito e' una descrizione della "realizzazione" della macchina astratta Ristorante.

Macchine Intermedie e Struttura a livelli dei computer moderni

Chiamiamo:

- **M** macchina astratta
- **L_M** linguaggio macchina di **M** : è il linguaggio che ha come stringhe legali tutti i programmi interpretabili dall'interprete di **M**

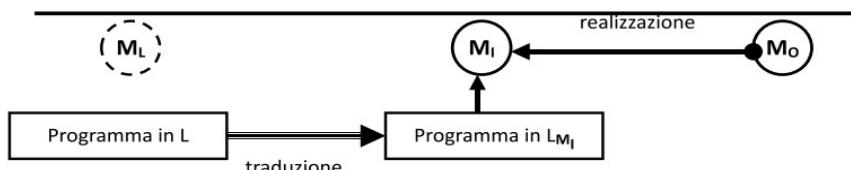
La differenza di potenza espressiva fra una macchina astratta che vogliamo realizzare e la macchina che abbiamo a disposizione (**macchina ospite M_o**) è detto **semantic gap**.

Spesso la semantic gap tra la macchina da realizzare e la macchina ospite è talmente grande che è opportuno introdurre una o più **macchine intermedie M_I**.

Tipicamente, nell'implementare un linguaggio di programmazione (cioè la sua corrispondente macchina astratta) non si procedere mai per pura interpretazione o compilazione.

La pura interpretazione potrebbe non essere soddisfacente a causa della scarsa efficienza della macchina realizzata emulando via software strutture dati,, Algoritmi e soprattutto l'interprete.

La compilazione, a causa di una notevole semantic gap, potrebbe portare a produrre programmi per la macchina ospite di dimensioni eccessive e lenti.



Per colmare il divario fra la macchina astratta e quella ospite si progetta un'apposita **macchina intermedia** che viene realizzata sulla macchina ospite.

A questo punto la traduzione dei programmi in \mathbf{L} avverrà in termini del linguaggio di $\mathbf{M_I}$ ed il programma così ottenuto verrà eseguito tramite l'interpretazione sulla macchina ospite (implementazione mista).

Se la differenza fra la macchina ospite e la macchina astratta è molto limitata allora potremmo scegliere altri tipi di implementazioni:

- Se $\mathbf{M_L} = \mathbf{M_I} \rightarrow$ interpretazione pura.
- Se $\mathbf{M_O} = \mathbf{M_I} \rightarrow$ traduzione pura.

Runtime Support

La differenza tra la macchina intermedia e la macchina ospite è il **supporto a tempo di esecuzione (RTS)**: collezione di strutture dati e sottoprogrammi che devono essere caricati sulla macchina ospite per permettere l'esecuzione del codice prodotto dal traduttore (compilatore).

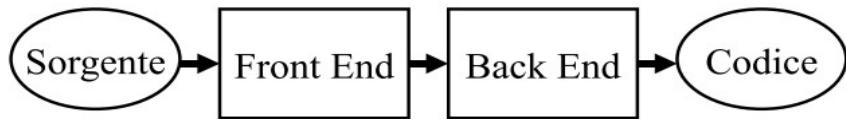
$$\mathbf{M_I} = \mathbf{M_O} + \mathbf{RTS}$$

Il runtime support fornisce i servizi necessari all'esecuzione di un programma.

Esempio:

- gestione dello stack
- libreria software per la gestione della memoria
- il codice che gestisce il caricamento dinamico e il linking
- codice di gestione dei thread

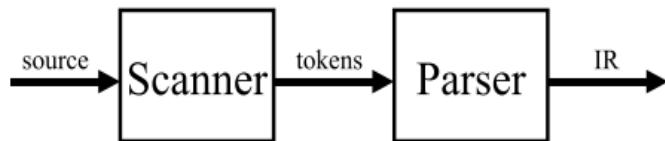
Compilatore



Normalmente viene diviso in 2 componenti:

- **Front End: (fasi di analisi)** Legge il programma sorgente e determina la sua struttura sintattica e semantica. Qui il compilatore riconosce se il programma sorgente è scritto usando le regole della grammatica del linguaggio, se non lo fosse o se l'inferenza dei tipi non è come si aspettava segnala errore.
- **Back End: (sintesi)** Genera il codice nel linguaggio macchina e lo ottimizza.

Front End



Due fasi principali:

- **scanner (analisi lessicale):** trasforma il programma sorgente (composto da operatori, punteggiatura, parole chiavi if, while,..., costanti int, float,...) nel lessico (token).

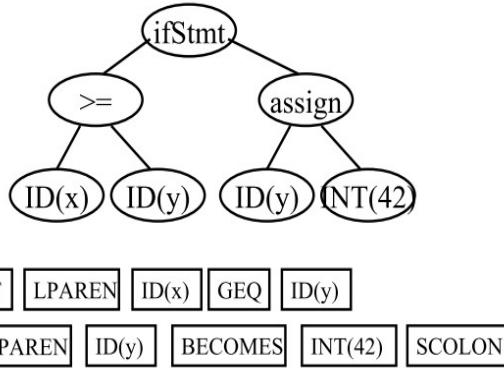
Scanner: un esempio

- Input
// codice stupido
if (x >= y) y = 42;
- Token

IF	LPAREN	ID(x)	GEQ	ID(y)
RPAREN	ID(y)	BECOMES	INT(42)	SCOLON

- **parser (analisi sintattica):** legge una sequenza di tokens e controlla che sia legale, ovvero rispetta le regole sintattiche che il linguaggio richiede per scrivere un programma. Se lo è genera il codice intermedio (IR) che lo vediamo come un abstract syntax tree (AST).
Il codice intermedio rappresenta la struttura sintattica del programma in termini di una struttura alberiforme.

Dall'esempio di prima:



Gli alberi di sintassi astratta sono particolarmente rilevanti perché mostrano la struttura significativa dei programmi. Noi non considereremo gli aspetti quali la precedenza degli operatori, ambiguità ecc.

La sintassi astratta di un linguaggio è espressa facilmente coi tipi di dato algebrici di Ocaml: ogni categoria sintattica diventa un tipo di dato algebrico di Ocaml.

Esempio:

```

Algebraic Data Type
Type BoolExp =
| True
| False
| Not of BoolExp
| And of BoolExp * BoolExp

```

Back End

Il Back End prende l'albero di sintassi astratta (codice intermedio) e lo traduce nel linguaggio della macchina ospite.

Deve conoscere le caratteristiche della macchina ospite ed utilizzare dei meccanismi di ottimizzazione per evitare di generare del codice ridondante.

Esempio:

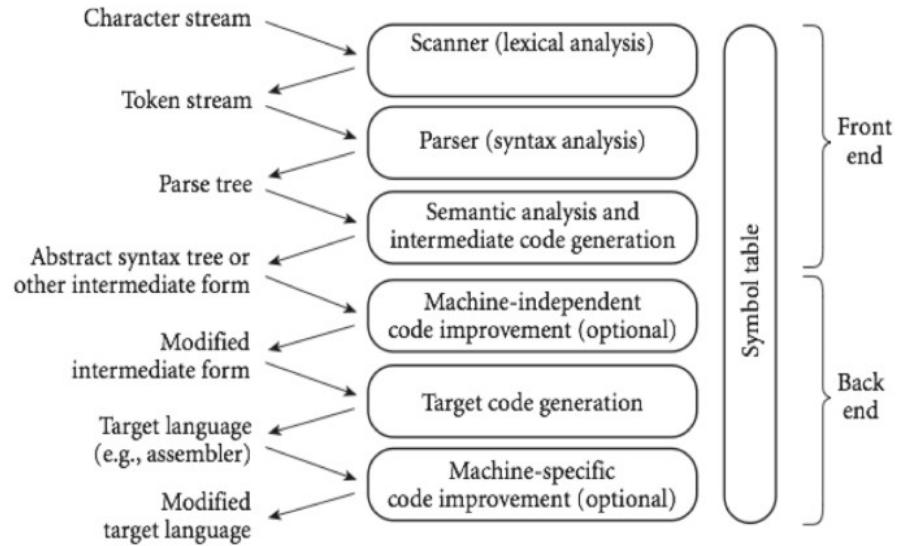
• Input	• Output
<pre> if (x >= y) y = 42; </pre>	<pre> mov eax,[ebp+16] cmp eax,[ebp-8] jl L17 mov [ebp-8],42 L17: </pre>

```

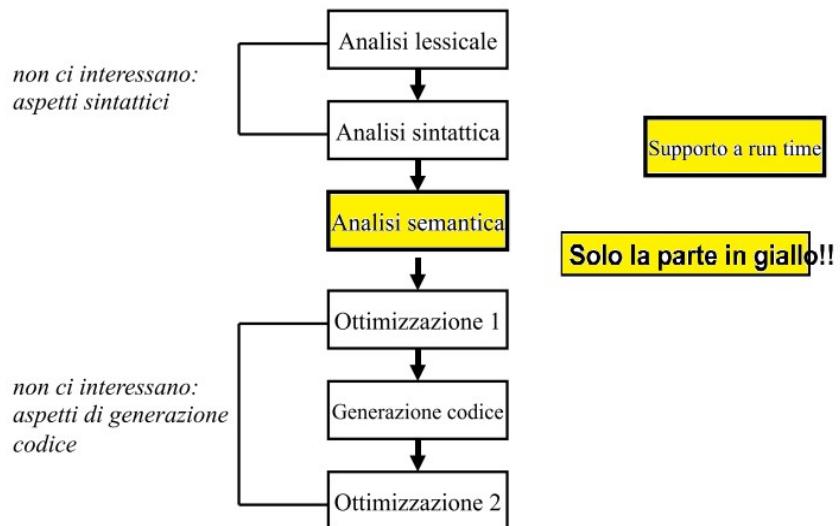
graph TD
    ifStmt((ifStmt)) --> opGtE((>=))
    ifStmt --> assign((assign))
    opGtE --> idX((ID(x)))
    opGtE --> idY1((ID(y)))
    assign --> idY2((ID(y)))
    assign --> int42(INT(42))

```

Riassunto: struttura di un compilatore



Cosa ci interessa?



LEZIONE 19

Realizzare un interprete in Ocaml

Considerando un semplice linguaggio per scrivere espressioni aritmetiche

$Exp \ e := e1 + e2 \mid e1 * e2 \mid n \in \text{Int}$ (le costanti intere sono valori che non devono essere valutati ulteriormente)

$v \in \text{Values} := n \in \text{Int}$

Per valutare un'espressione e utilizziamo $\text{eval}(e)$ t.c. :

- $e = n$ allora $\text{eval}(e) = n$
- $e = e1 + e2$, se $\text{eval}(e1) = v1$ & $\text{eval}(e2) = v2$ allora $\text{eval}(e1 + e2) = v1 + v2$
- $e = e1 * e2$, se $\text{eval}(e1) = v1$ & $\text{eval}(e2) = v2$ allora $\text{eval}(e1 * e2) = v1 * v2$

Graficamente:

In Ocaml (per rappresentare AST):

- type op = Plus | Times
- type exp = Int_e of int
| Op_e of exp * op * exp

- AST -- 1 + (2 * 3)
- Op_e(Int_e(1), Plus,
Op_e(Int_e(2),
Times,
Int_e(3)))

Ora serve, però, decodificare/interpretare un'espressione in termini delle operazioni primitive sulla macchina virtuale che stiamo utilizzando (in questo caso quella di Ocaml):

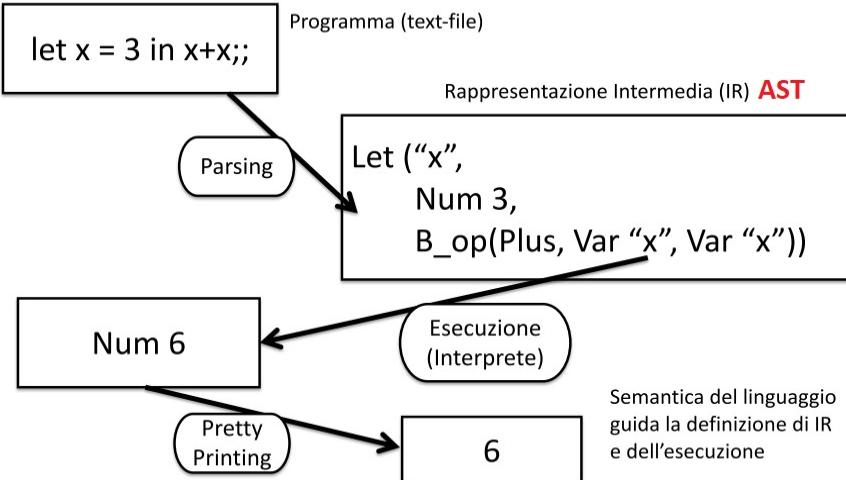
- **let eval_op (v1:exp) (op:operand) (v2:exp) : exp =**
match v1, op, v2 with
| Int_e i, Plus, Int_e j -> Int_e (i + j)
| Int_e i, Times, Int_e j -> Int_e (i * j)
| _, (Plus | Times), _ ->
 if is_value v1 && is_value v2
 then raise failwith "Type_Error"
 else raise failwith "Not_Value"
- **let is_value (e : exp) : bool =** match e with
| Int_e _ -> true
| Op_e _ -> false;;
- **let rec eval (e : exp) : exp =**
match e with
| Int_e i -> Int_e i
| Op_e(e1,op,e2) -> let v1 = eval e1 in
 let v2 = eval e2 in
 eval_op v1 op v2

(qui si capisce che le valutazioni vanno a sx verso dx e che ha una valutazione eager: prima si valuta e1 poi e2 e infine si opera con i valori ottenuti)

esempio:

```
# let e = Op_e(Int_e(5), Times, Int_e(10));;
val e : exp = Op_e (Int_e 5, Times, Int_e 10)
# eval e;;
- : exp = Int_e 50
#
```

In generale:



Regole di valutazione

Sia \exp un'espressione e v un valore, diciamo che v è il valore di \exp se valutando \exp otteniamo v come risultato.

$$\boxed{\exp \Rightarrow v}$$

Valutazione di valori

$$\boxed{v \Rightarrow v}$$

Valutazione di operatori

$$\frac{\exp_1 \Rightarrow v_1, \exp_2 \Rightarrow v_2}{\exp_1 * \exp_2 \Rightarrow v_1 * v_2}$$

$$\frac{\exp_1 \Rightarrow v_1, \exp_2 \Rightarrow v_2}{\exp_1 + \exp_2 \Rightarrow v_1 + v_2}$$

Tra queste e le regole in Ocaml non c'è alcuna differenza: la regola di valutazione matematica corrispondente esattamente al meccanismo di calcolo scritto all'interno di un linguaggio di programmazione.

Lo facciamo perché così possiamo dimostrare una proprietà: la correttezza di un interprete.

Teorema

Sia E una espressione. $E \Rightarrow v$ se e solo se $\text{eval}(E) = v$

(Si dimostra per induzione strutturale)

Passiamo a linguaggi più articolati

```
type variable = string

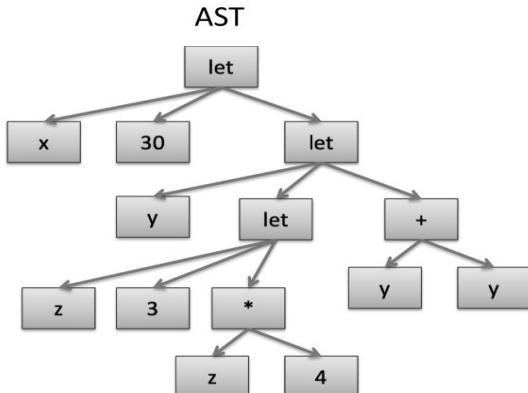
type op = Plus | Minus | Times | ...

type exp =
| Int_e of int
| Op_e of exp * op * exp
| Var_e of variable
| Let_e of variable * exp * exp

type value = exp
```

Esempio:

Programma OCaml	Exp
<pre>let x = 30 in let y = (let z = 3 in z*4) in y+y;;</pre>	<pre>Let_e("x", Int_e 30, Let_e("y", Let_e("z", Int_e 3, Op_e(Var_e "z", Times, Int_e 4)), Op_e(Var_e "y", Plus, Var_e "y")))</pre>



Interprete 1.0

(Runtime Support)

```

let is_value (e : exp) : bool =
  match e with
  | Int_e _ -> true
  | Op_e _
  | Let_e _
  | Var_e _ ) -> false;;

```

```

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, Minus, Int_e j -> Int_e (i - j)
  | Int_e i, Times, Int_e j -> Int_e (i * j)
  | _, (Plus | Minus | Times), _ ->
    if is_value v1 && is_value v2
    then raise TypeError
    else raise NotValue

```

```

let substitute (v:value) (x:variable) (e:exp) : exp =
  let rec subst (e:exp) : exp =
    match e with
    | Int_e _ -> e
    | Op_e(e1,op,e2) -> Op_e(subst e1,op,subst e2)
    | Var_e y -> if x = y then v else e
    | Let_e (y,e1,e2) -> Let_e (y,subst e1,
                                   if x = y then e2 else subst e2)

```

(Interprete)

```

let rec eval (e : exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) -> eval_op eval e1 op eval e2
  | Let_e(x,e1,e2) -> let v1 = eval e1 in
    let e2' = substitute v1 x e2 in
    eval e2'
  | Var_e x -> raise (UnboundVariable x)

```

Tali eccezioni fanno parte del RTS

Interprete 2.0

Aggiungiamo adesso le Funzioni.

Aggiorniamo!

Sintassi

```
type exp = Int_e of int | Op_e of exp * op * exp  
| Var_e of variable | Let_e of variable * exp * exp  
| Fun_e of variable * exp | FunCall_e of exp * exp
```

La sintassi OCaml `fun x e` viene rappresentata come `Fun_e(x, e)`

La chiamata `fact 3` viene rappresentata come
`FunCall_e (Var_e "fact", Int_e 3)`

Estendiamo il RTS

```
let is_value (e : exp) : bool =  
  match e with  
    | Int_e _ -> true  
    | Fun_e (_, _) -> true  
    | (Op_e (_, _, _))  
    | Let_e (_, _, _)  
    | Var_e _  
    | FunCall_e (_, _) ) -> false
```

Le funzioni sono valori!!

Interprete ++

```
is_value : exp -> bool  
eval_op : value -> op -> value -> value  
substitute : value -> variable -> exp -> exp
```

```
let rec eval (e : exp) : exp =  
  match e with  
    | :  
    | Var_e x -> raise (UnboundVariable x)  
    | Fun_e (x, e) -> Fun_e (x, e)  
    | FunCall_e (e1, e2) ->  
      match eval e1, eval e2 with  
        | Fun_e (x, e), v2 -> eval (substitute v2 x e)  
        | _ -> raise (TypeError)
```

Interprete 2.5

Funzioni ricorsive

```
type exp = Int_e of int | Op_e of exp * op * exp  
| Var_e of variable | Let_e of variable * exp * exp  
| Fun_e of variable * exp | FunCall_e of exp * exp  
| Rec_e of variable * variable * exp
```

Esempio:

Let g = rec f x → f(x+1) in g 3

tradotto:

Let_e ("g", Rec_e ("f", "x", FunCall_e (Var_e "f", Op_e (Var_e "x", Plus, Int_e 1)), FunCall (Var_e "g", Int_e 3)))

Morale

Abbiamo imparato che Ocaml può essere utilizzato come linguaggio per simulare gli interpreti e quindi simulare la semantica operazionale dei linguaggi di programmazione, incluso se stesso.

Di fatto abbiamo fatto un interprete di Ocaml in Ocaml.

Il vantaggio è quello di vedere la simulazione dell'implementazione.

Lo svantaggio è che è complicato: avere un interprete che fa tutte queste operazioni è poco efficiente infatti negli interpreti ideali non abbiamo mai una nozione di sostituzione (→ il runtime è più complicato).

Interpreti, compilatori e semantica operazionale

Collegiamo l'uso delle tecniche di natura formale basata sulla semantica operazionale per la definizione di "interprete" del linguaggio.

Linguaggi di programmazione

Per studiare un linguaggio di programmazione per noi, si ottiene dalla **semantica** del linguaggio.

La **semantica** sarà una guida per la progettazione, l'implementazione e l'uso di un linguaggio di programmazione.

Un linguaggio di programmazione possiede:

- una **sintassi** che definisce le "formule ben formate" del linguaggio. Ovvero i programmi sintatticamente corretti, tipicamente generati da una grammatica.
- Una **semantica** che fornisce sia un'interpretazione dei "token" in termini di entità (matematiche) note che un significato ai programmi sintatticamente corretti.

Sintassi

La **sintassi concreta** di un linguaggio di programmazione è definita di solito da una **grammatica libera da contesto**.

La **sintassi astratta** è una rappresentazione lineare dell'albero sintattico dove gli operatori sono nodi dell'albero e gli operandi sono rappresentati dai sottoalberi (abbiamo sia una notazione lineare che una grafica).

Semantica

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 (quella che utilizzeremo per comprendere com'è il runtime dei linguaggi di programmazione).
- **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.

Semantica operazionale

La semantica operazionale descrive come vengono modificati gli stati della macchina astratta di esecuzione del linguaggio. Dunque bisogno avere un modo per definire in maniera formale gli stati legali dell'esecuzione, delle regole che ci dicono che da uno stato si passa ad un altro e gli stati finali.

Utilizziamo un sistema di transizioni il quale è costituito da:

- un insieme **Config** di configurazioni (stati)
- una relazione di transizione $\rightarrow \subseteq \text{Config} \times \text{Config}$

Notazione: $c \rightarrow d$ significa che c e d sono nella relazione \rightarrow (cambiamento di stato)

Ovvero $c \rightarrow d$ significa che lo stato c evolve nello stato d

Una semantica operazionale “small step” la relazione di transizione descrive un passo del processo di calcolo.

Noi vedremo quella “big step” in cui la relazione di transizione descrive la valutazione completa di un programma/espressione e scriveremo $e \Rightarrow v$ se l'esecuzione del programma/espressione e produce il valore v .

Regole di derivazione

- Le regole di valutazione costituiscono un *proof system* (sistema di dimostrazione)

$\text{premessa}_1 \dots \text{premessa}_k$

conclusione

- Tipicamente le regole sono definite per induzione strutturale sulla sintassi del linguaggio
- Le “formule” che ci interessa dimostrare sono transizioni del tipo $e \Rightarrow v$
- Componiamo le regole in base alla struttura sintattica di e ottenendo un *proof tree*

Quello che vedremo è come ogni regola del proof sysyem corrisponde operazionalmente ad un programma Ocaml, permettendoci di derivare in modo sistematico, a partire dalla semantica operazionale, le regole di base dell'interprete Ocaml.

LEZIONE 20

Dati

I dati servono a varie cose:

- **Livello di progetto:** i dati servono per organizzare in modo preciso l'informazione (visto con la nozione di tipo astratto e con la costruzione di API). Infatti abbiamo visto un meccanismo esplicito previsto da Java per definire nuovi tipi di dati utilizzando le istruzioni linguistiche che il linguaggio mi mette a disposizione.
- **Livello di programma:** identificano e prevengono errori (3+ "pippo" deve essere sbagliata).
- **Livello di implementazione:** permettono alcune ottimizzazioni (bool richiede meno bit di int).

Possiamo classificarli:

- **Denotabili:** se posso associarli ad un nome.
- **Esprimibili:** se possono essere il risultato di una valutazione di una espressione complessa.
- **Memorizzabili:** se possono essere memorizzati in una variabile.

In una macchina astratta si possono vedere due classi di tipi di dato:

- **di sistema:** definiscono lo stato e le strutture dati utilizzate nella simulazione di costrutti di controllo.
- **di programma:** dati corrispondenti ai tipi primitivi del linguaggio e ai tipi che l'utente può definire (se il linguaggio lo consente)

Un **tipi di dato** è una collezione di valori rappresentati da opportune strutture dati e un insieme di operazioni per manipolarli.

Il **descrittore del dato** serve per avere un meccanismo che a livello dell'implementazione del linguaggio di programmazione nel run-time serve per decodificare la struttura che ho.

```
type exp =
  (* AST *)
  | Eint of int
  | Ebool of bool
```

```
type evT =
  (*Valori run-time*)
  |Int of int
  |Bool of bool
```

I descrittori dei tipi di dato sono espressi tramite i costruttori Int e Bool

I descrittori Int e Bool (di evT) è quel meccanismo che dice che una determinata sequenza di bit va interpretata come un intero o come booleano e questo mi dice anche quant'è la memoria occupata.

I descrittori, a run-time, possono servire per tutti i linguaggi che non hanno un controllo statico dei tipi perchè usano i descrittori dei tipi per fare un controllo dinamico.

Ad esempio se abbiamo un'operazione di somma: $x+y$. La somma è sensata se x e y sono valori interi, se non ho un controllo di tipo statico devo andarlo a controllare dinamicamente a run-time tramite un'operazione di type-checking.

Uso dei descrittori: run time type checking



```
let plus(x, y) =
  if typecheck("int", x) & typecheck("int", y)
  then
    (match (x, y) with
     | (Int(u), Int(w)) -> Int(u + w))
  else failwith ("type error")
```

```
let typecheck (x, y) = match x with
  | "int" ->
    (match y with
     | Int(u) -> true
     | _ -> false)
```

Riassunto:

Tipi a tempo di compilazione e a tempo di esecuzione

1. Se l'informazione sui tipi è conosciuta completamente "a tempo di compilazione" (OCaml)
 1. si possono eliminare i descrittori di dato a run-time
 2. il type checking è effettuato totalmente dal compilatore (type checking statico)
2. Se l'informazione sui tipi è nota solo "a tempo di esecuzione" (JavaScript)
 1. sono necessari i descrittori per tutti i tipi di dato
 2. il type checking è effettuato totalmente a tempo di esecuzione (type checking dinamico)
3. Se l'informazione sui tipi è conosciuta solo parzialmente "a tempo di compilazione" (Java)
 1. i descrittori di dato contengono solo l'informazione "dinamica"
 2. il type checking è effettuato in parte dal compilatore e in parte dal supporto a tempo di esecuzione

LEZIONE 21

Nomi

Un nome è una sequenza di caratteri usata per denotare simbolicamente un oggetto.

L'uso dei nomi realizza un primo meccanismo elementare di **astrazione**:

- **Astrazione sui dati:** la dichiarazione di una variabile permette di introdurre un nome simbolico per una locazione di memoria, astraendo sui dettagli della gestione della memoria.
- **Astrazione sul controllo:** la dichiarazione del nome di una funzione (procedura) permette di associare un nome simbolico a una sequenza di comandi. Questa possibilità è essenziale nel processo di astrazione procedurale.

Le **entità denotabili** sono gli elementi di un linguaggio di programmazione a cui posso assegnare un nome:

- entità i cui nomi sono definiti dal linguaggio di programmazione (tipi primitivi, operazioni primitive, ...)
- entità i cui nomi sono definiti dall'utente (parametri, procedure, tipi, costanti simboliche, classi, oggetti, ...)

Binding

Un **binding** è un'associazione tra un nome e un'entità del linguaggio di programmazione.

Le **regole di visibilità (scope)** sono quelle regole che mi definiscono quelle porzioni del programma in cui un binding particolare è attiva, ovvero dove riesco a riferire quel nome e tramite il nome riesco ad accedere a quell'entità.

Il **binding time** definisce il momento temporale nel quale viene definita una associazione e nel quale vengono prese le decisioni relative alla gestione delle associazioni (ovvero delle scelte di progetto che caratterizzano quel legame).

Esempi di binding time

- **Language design time** – progettazione del linguaggio
 - binding delle operazioni primitive, tipi, costanti
- **Program writing time** – tempo di scrittura del programma
 - binding dei sottoprogrammi, delle classi, ...
- **Compile time** – tempo di compilazione
 - associazioni per le variabili globali
- **Run time** – tempo di esecuzione
 - associazioni tra variabili e locazioni, associazioni per le entità dinamiche

Si parla di **static (dynamic) binding** è solitamente utilizzato per fare riferimento a una associazione attivata *prima (dopo)* aver mandato il programma in esecuzione.

I linguaggi “compilati” cercano di risolvere il binding staticamente.

I linguaggi “interpretati” cercano di risolvere il binding dinamicamente.

Ambiente

E' la struttura a run-time che ci permette di risolvere i binding.

L'ambiente è quella struttura dati a tempo di esecuzione che mi permette di risolvere le associazioni tra un nome e l'entità associata a questo. In ogni specifico punto del programma l'ambiente mi dice cosa è collegato ad un determinato nome.

Questo significa che deve essere presente nella macchina astratta.

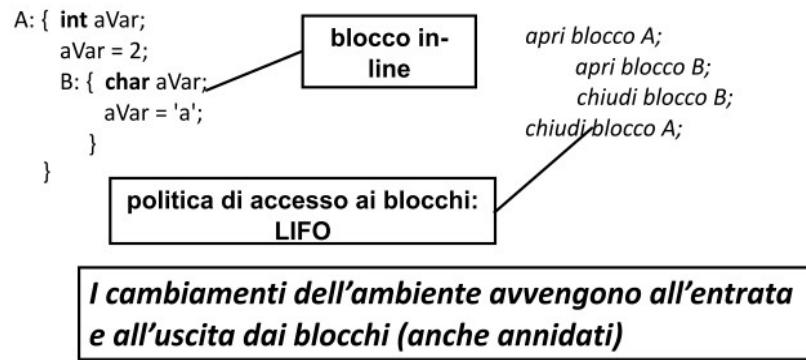
L'ambiente viene creato al momento dell'esecuzione di un programma.

Le dichiarazioni sono quel costrutto linguistico che permette di dire che sto introducendo nuove associazioni nell'ambiente.

Quando introduco una variabile tramite una dichiarazione sto dicendo anche la portata (scope) della dichiarazione.

I costrutti linguistici che operano nell'ambiente, ovvero che permettono di introdurre nuove dichiarazioni, sono i **blocchi**.

Un **blocco** è quella regione testuale all'interno del programma in cui è possibile fare delle dichiarazioni. (In C e Java sono le {} mentre in Ocaml è il Let).



Se ho parentesi { creo un nuovo ambiente, ogni volta che ho } chiudo l'ambiente associato.

3 tipi di ambiente:

- **Ambiente locale:** l'insieme delle associazioni dichiarate localmente e nel caso di metodi e procedure sono comprese le eventuali associazioni relative ai parametri.
- **Ambiente non locale:** associazioni dei nomi che sono visibili all'interno del blocco ma non dichiarati nel blocco stesso.
- **Ambiente globale:** associazioni per i nomi usabili da tutte le componenti che costituiscono il programma.

Tipi di ambiente: esempio in C



```
#include <stdio.h>

int main( ){
    A:{ int a = 1 ;
        B:{ int b = 2;
            int c = 2;
            C:{ int c = 3;
                int d;
                d = a + b + c;
                printf("%d\n", d);
            }
            D:{ int e;
                e = a + b + c;
                printf("%d\n", e);
            }
        }
    }
}
```

Ambiente locale di C

associazioni per **c** e **d**

Ambiente non locale per C

associazione per **b** ereditata da **B**

associazione globale per **a**

Ambiente Globale

associazione per **a**

Cosa stampa?

Tipi di ambiente: esempio in Java



```
public class Prova {
    public static void main(String[ ] args) {
        A:{ int a =1 ;
            B:{ int b = 2;
                int c = 2;
                C:{ int c = 3;
                    int d;
                    d = a + b + c;
                    System.out.println(d);
                }
                D:{ int e;
                    e = a + b + c;
                    System.out.println(e);
                }
            } } }
```

NB. in Java non è possibile ri-dichiarare una variabile già dichiarata in un blocco più esterno

```
$ javac Prova.java
Prova.java:7: c is already defined in main(java.lang.String[])
      C:{ int c = 3;
           ^
```

Cambiamenti dell'ambiente



- L'ambiente può cambiare a **run time**, ma i cambiamenti avvengono di norma in precisi momenti
 - **entrando in un blocco**
 - creazione delle associazioni fra i nomi locali al blocco e gli oggetti denotati
 - disattivazione delle associazioni per i nomi ridefiniti
 - **uscendo dal blocco**
 - distruzione delle associazioni fra i nomi locali al blocco e gli oggetti denotati
 - riattivazione delle associazioni per i nomi che erano stati ridefiniti

Operazioni su ambienti



- **Naming:** creazione di associazione fra nome e oggetto denotato (dichiarazione locale al blocco o parametro)
- **Referencing:** riferimento a un oggetto denotato mediante il suo nome (uso del nome per accedere all'oggetto denotato)
- **Disattivazione** di associazione fra nome e oggetto denotato (la nuova associazione per un nome maschera la vecchia associazione, che rimane disattivata fino all'uscita dal blocco)
- **Riattivazione** di associazione fra nome e oggetto denotato (una vecchia associazione che era mascherata è riattivata all'uscita da un blocco)
- **Unnaming:** distruzione di associazione fra nome e oggetto denotato (esempio: ambiente locale all'uscita di un blocco)
- **Il tempo di vita degli oggetti denotati non è necessariamente uguale al tempo di vita di un'associazione**

Implementazione dell'ambiente

Un ambiente possiamo vederlo come una collezione di binding (legami).

Esempio $env = \{ x \rightarrow 25, y \rightarrow 6 \}$

Astrattamente possiamo vederlo come una funzione di tipo:

$$Id \rightarrow Value + Unbound$$

L'uso della costante *Unbound* permette di rendere la funzione totale.

- Dato un ambiente **env**: **Id → 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 $env = \{x \rightarrow 25, y \rightarrow 7\}$ allora
 $env[5/x] = \{x \rightarrow 5, y \rightarrow 7\}$

Implementazione semplice

```
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
```

Implementazione più estesa

Ambiente: interfaccia

```
# module type ENV =
sig
  type 't env
  val emptyenv : 't -> 't env
  val bind : 't env * string * 't -> 't env
  val bindlist : 't env * (string list) * ('t list)
    -> 't env
  val applyenv : 't env * string -> 't
  exception WrongBindlist
end
```

Ambiente: semantica

```
# module Funenv: ENV =
struct
  type 't env = string -> 't
  exception WrongBindlist
  let emptyenv(x) = function (y: string) -> x
    (* x: valore default *)
  let applyenv(x, y) = x y
  let bind(r, l, e) =
    function lu -> if lu = l then e else applyenv(r, lu)
  let rec bindlist(r, il, el) = match (il, el) with
  | ([], []) -> r
  | (i::ill, e::ell) -> bindlist (bind(r, i, e), ill, ell)
  | _ -> raise WrongBindlist
end
```

Scope

Lo **scope** di un binding definisce quella parte del programma nella quale il binding è attivo.

2 tipidi regole di visibilità:

- **scope statico:** è determinato dalla struttura sintattica del programma.
- **scope dinamico:** è determinato dalla struttura a tempo di esecuzione.

Una dichiarazione locale in un blocco è visibile: in quel blocco e in tutti i blocchi in esso annidati (salvo ri-dichiarazioni dello stesso nome e si parla di *shadowing*).

```

int main( ) {
    int x = 0;
    void proc(int n) { x = n+1; }
    proc(2);
    printf("%d\n", x);           _____| Stampa 3
{ int x = 0;
  proc(3);
  printf("%d\n", x);           _____| Stampa 0
}
printf("%d\n", x);           _____| Stampa 4
}

```

Cosa stampa?

```

let x = 5;;
let f z = let w = x + z in w +1 ;;
let x = 10;;
f 25;

```

Quale è l'ambiente locale dell'invocazione di f?

La dichiarazione di w con x=5 e z=25

Quale è l'ambiente non locale dell'invocazione di f?

La dichiarazione di x=5

- Lo scope statico permette di determinare tutti gli ambienti di un programma staticamente, osservando la struttura sintattica del programma
 - controlli di correttezza a compile time
 - ottimizzazione del codice a compile time
 - possibile il controllo statico dei tipi
- Gestione a run time articolata
 - gli ambienti non locali di funzioni e procedure evolvono diversamente dal flusso di attivazione e disattivazione dei blocchi

Scope dinamico

- L'associazione valida per un nome **x**, in un punto **P** di un programma, è la più recente associazione creata (in senso temporale) per **x** che sia ancora attiva quando il flusso di esecuzione arriva a **P**
- Come vedremo, lo scope dinamico ha una gestione a run time semplice
 - vantaggi: flessibilità nei programmi
 - svantaggi: difficile comprensione delle chiamate delle procedure e controllo statico dei tipi non possibile

LEZIONE 22

Un linguaggio di programmazione contiene una sintassi ovvero un modo non ambiguo di descrivere programmi che sono sintatticamente corretti ovvero quelli generati dalla grammatica che definisce la struttura del linguaggio.

Questo serve a noi e al compilatore per controllare che il programma che abbiamo scritto segue i vincoli sintattici della grammatica.

Poi abbiamo una nozione semantica perchè l'idea è che uno abbia anche un modello di linguaggio di programmazione in modo da comprendere quali sono le caratteristiche da un punto di vista computazionale del linguaggio (cosa calcola, come lo calcola, in che ordine...).

La teoria degli automi serve per l'analisi sintattica (lo scanner), la teoria delle grammatiche serve per l'analisi del parser ovvero per vedere se un linguaggio è scritto secondo le regole della grammatica.

Per definire un modello ci sono varie tecniche: semantica denotazionale, operazione e assiomatica.

Noi utilizzeremo la semantica operazionale ovvero quel meccanismo che permette di definire formalmente com'è fatto l'interprete.

Espresioni logiche

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

Definizione della sintassi astratta tramite i tipi
algebrici di OCaml

$$\frac{e \Rightarrow v}{\text{not } e \Rightarrow \neg v} \quad \frac{e_1 \Rightarrow v_1, e_2 \Rightarrow v_2}{e_1 \text{ and } e_2 \Rightarrow v_1 \wedge v_2}$$

Tabella: $true \wedge true = true$

Diventa

```
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
```

Ma potremmo utilizzare un regola di corto circuito in cui $e1 \&& e2$ restituisce *false* se $e1$ viene valutata *false* altrimenti si valuta $e2$.

Anche in $e1 || e2$ restituisce *true* se $e1$ viene valutato *true* altrimenti si valuta $e2$.

REGOLA LOGICA	$\frac{e_1 \Rightarrow \text{false}}{e_1 \&\& e_2 \Rightarrow \text{false}}$	$\frac{e_1 \Rightarrow \text{true}, e_2 \Rightarrow v_2}{e_1 \&\& e_2 \Rightarrow v_2}$
INTERPRETE		<pre>SAnd(exp0,exp1) -> match eval exp0 with False -> False True -> eval exp1</pre>

Espressioni a valori interi

<pre>type expr = ... CstI of int // costanti intere Sum of expr * expr // operatori binary Times of expr *expr</pre>
--

Regole di valutazione e Interpretazione

$$n \Rightarrow n \quad \frac{e_1 \Rightarrow v_1, e_2 \Rightarrow v_2}{Sum(e_1, e_2) \Rightarrow v_1 + v_2}$$

eval CstI(n) -> n	eval Sum(e1, e2) -> eval e1 + eval e2
-------------------	--

Dichiarazioni

<pre>type expr = Var of string Let of string * expr * expr</pre>

Esempio <pre>Let("z", CstI 17, Sum(Var "z", Var "z"))</pre> <p>In sintassi concreta</p> <pre>let z = 17 in z + z</pre>
--

Dobbiamo prima introdurre l'**ambiente**.

Ambiente

Per definire l'interprete dobbiamo introdurre una **struttura di implementazione (run-time structure)** che permette di recuperare i valori associati agli identificatori

Un'implementazione ingenua potrebbe essere la seguente:

```
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
```

Regole di valutazione e interprete

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

eval (Var x) env -> lookup env x

$$\frac{env \triangleright e_1 \Rightarrow v_1, env \triangleright e_2 \Rightarrow v_2}{env \triangleright Sum(e_1, e_2) \Rightarrow v_1 + v_2}$$

eval Sum(e1, e2) env ->
eval e1 env + eval e2 env

Dichiarazioni

$$env \triangleright erhs \Rightarrow xval \quad env[xval/x] \triangleright ebody \Rightarrow v$$

$$env \triangleright Let\ x = erhs\ in\ ebody \Rightarrow v$$

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**

```
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
  | SUM(e1, e2) -> eval e1 env + eval e2 env
  | TIMES(e1, e2) -> eval e1 env * eval e2 env
  | MINUS e1, e2) -> eval e1 env - eval e2 env
  | _                -> failwith "unknown primitive"
```

IF THEN ELSE

- Sintassi concreta
 - $e ::= \dots e1 == e2 \mid \text{if } e \text{ then } e1 \text{ else } e2$
- Sintassi astratta
 - Type exp =
 - | Eq of exp * exp
 - | Cond of exp * exp * exp

Valutazione EQ

REGOLA LOGICA

$$\frac{e_1 \Rightarrow v, e_2 \Rightarrow v}{Eq(e_1, e_2) \Rightarrow True}$$

INTERPRETE

```
Eq(exp0,exp1) ->
  match (eval exp0 eval exp1) with
    (n, n) -> True
    (-,-) -> False
```

Valutazione Condizionale

REGOLA LOGICA

$$\frac{\begin{array}{l} env \triangleright e \Rightarrow \text{true}, e_1 \Rightarrow v \\ env \triangleright \text{Cond}(e, e_1, e_2) \Rightarrow v \end{array}}{\begin{array}{l} env \triangleright e \Rightarrow \text{false}, e_2 \Rightarrow v' \\ env \triangleright \text{Cond}(e, e_1, e_2) \Rightarrow v' \end{array}}$$

INTERPRETE

```
eval Cond(g,exp0,exp1) env ->
  match eval g env with
    True -> eval exp0 env
    False -> eval exp1 env
    _ -> failwith("non Boolean guard")
```

INTRODURRE LA NOZIONE DI TIPI

Il problema è che non viene effettuato un controllo di tipi e potremmo scrivere cose strane tipo

$e1 = 1 + (2 == 3)$ oppure $e2 = \text{if } 1 \text{ then } 2 \text{ else } 3$

Una prima soluzione è un run.time type checking.

Introduciamo la nozione di tipi esprimibili:

- Introdurre la nozione di tipi esprimibili
 - type evT = Int of int | Bool of bool
- Modificare le regole dell'interprete di conseguenza
- let rec eval (e : exp) : evT =


```
match e with
        | CstTrue -> Bool(true)
        | CstFalse -> Bool(false)
        | CstI n -> Int(n)
```

E poi una funzione ausiliaria:

Run Time Type Checking

```
let typecheck (x, y) = match x with
  | "int" -> (match y with
    | Int(u) -> true
    | _ -> false)
  | "bool" -> (match y with
    | Bool(u) -> true
    | _ -> false)
  | _ -> failwith ("not a valid type");;

val typecheck : string * evT -> bool = <fun>
```

A questo punto dobbiamo modificare l'interprete con alcune funzioni ausiliarie.
Esempio per la somma:

```
let int_plus(x, y) =
  match(typecheck("int",x), typecheck("int",y), x, y) with
  | (true, true, Int(v), Int(w)) -> Int(v + w)
  | (_, _) -> failwith("run-time error ");;
```

$$\frac{env \triangleright e_1 \Rightarrow v_1, env \triangleright e_2 \Rightarrow v_2, v_1: int, v_2: int}{env \triangleright Sum(e_1, e_2) \Rightarrow v_1 + v_2 : int}$$

```
let rec eval e env = match e with
  | CstInt(n) -> Int(n)
  | CstTrue -> Bool(true)
  | CstFalse -> Bool(false)
  | Sum(e1, e2) -> int_plus((eval e1 env), (eval e2 env))
```

Condizionale

REGOLA LOGICA

$$\frac{\begin{array}{c} env \triangleright e \Rightarrow true, env \triangleright e_1 \Rightarrow v \\ env \triangleright Cond(e, e_1, e_2) \Rightarrow v \end{array}}{env \triangleright e \Rightarrow false, env \triangleright e_2 \Rightarrow v'}$$
$$\frac{\begin{array}{c} env \triangleright e \Rightarrow false, env \triangleright e_2 \Rightarrow v' \\ env \triangleright Cond(e, e_1, e_2) \Rightarrow v' \end{array}}{env \triangleright Cond(e, e_1, e_2) \Rightarrow v'}$$

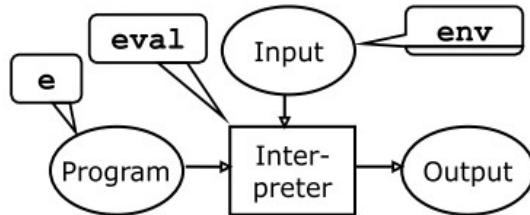
INTERPRETE

```
eval Cond(g,e0,e1) env ->
  match eval g env with
    (match (typecheck("bool", g), g) with
      |(true, Bool(true)) -> eval e0 env
      |(true, Bool(false)) -> eval e1 env
      |(_, _) -> failwith ("nonboolean guard")
    )
```

Occorrenze legate e libere

- La nozione di variabile libera o legata si applica anche al caso del costrutto **let**
- Infatti il costrutto **let** si comporta come un quantificatore per la variabile che introduce
- Un identificatore **x** si dice “legato” se appare nel **ebody** dell’espressione **let x = ehrs in ebody**, altrimenti si dice libero
- Esempi
 - **let z = x in z + x** (* z legata, x libera *)
 - **let z = 3 in let y = z + 1 in x + y**
(* z, y legate, x libera *)

Al momento questo è il nostro assetto:



Se volessimo avvicinarci verso la compilazione:

Indici per variabili

Idea: indice di una variabile =
numero dei **let** che si attraversano per raggiungerla

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



```
Let(CstI 17,  
    Let(CstI 25,  
        Sum(Var 1, Var 0)))
```

LEZIONE 23

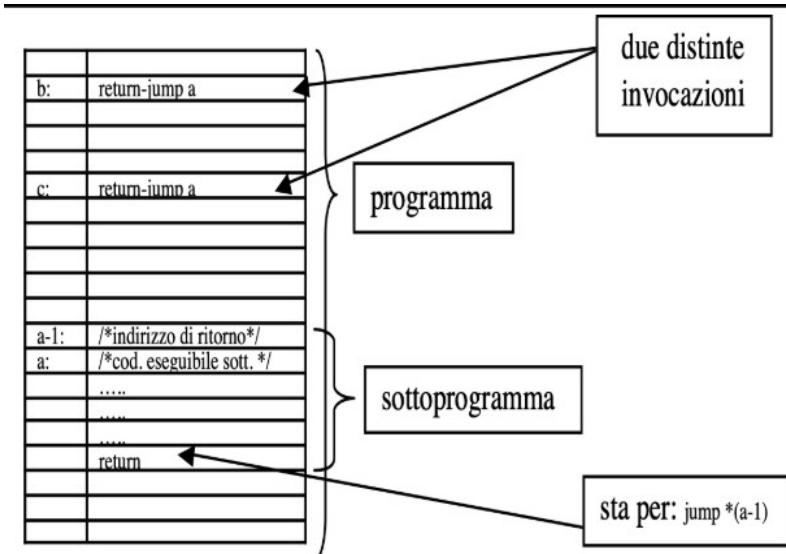
Funzioni/Procedure

Le chiamate ai sottoprogrammi:

- Astraggono da una sequenza di istruzioni
- Astraggono sul controllo
- Astraggono via parametrizzazione

L'hardware, per le chiamate ai sottoprogrammi, fornisce:

- Primitiva di **return jump** con opportune strutture ausiliarie
- Viene eseguita (nel programma chiamante) l'istruzione **return jump a** memorizzata nella cella **b**
 - il controllo viene trasferito alla cella **a** (entry point della subroutine)
 - l'indirizzo dell'istruzione successiva del chiamante (**b + 1**) viene memorizzato in qualche posto noto, per esempio nella cella (**a - 1**) (**punto di ritorno**)
- quando nella subroutine si esegue una operazione di return
 - il controllo ritorna all'istruzione (del programma chiamante) memorizzata nel punto di ritorno



Ogni volta che invoco un sottoprogramma devo istanziare una incarnazione di quel sottoprogramma e non a quella precedente (come avveniva in FORTRAN).

Ogni chiamata di sottoprogramma è rimpiazzata a tempo di esecuzione da una copia del codice.

Quindi ogni volta che invochiamo una procedura dobbiamo creare un nuovo ambiente locale che tiene conto dei valori delle variabili dichiarate localmente ma per quella specifica attivazione e non quella precedente. (in FORTRAN questo non avveniva e quindi era un problema per le procedure ricorsive).

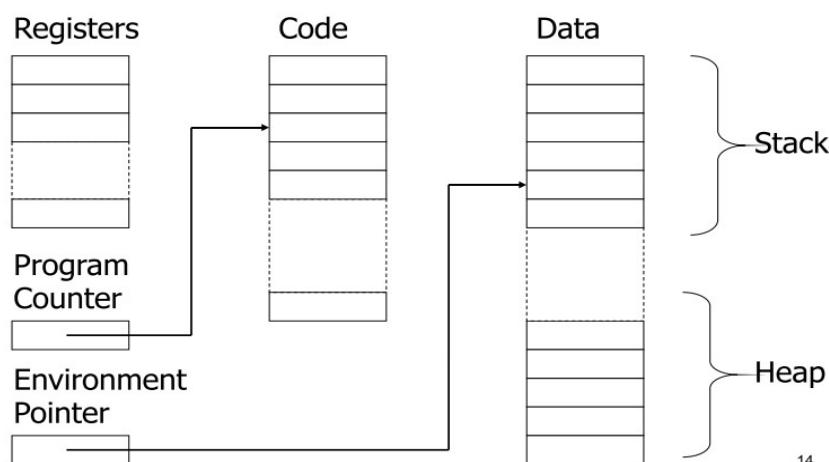
Le strutture di implementazione



Invece delle informazioni **staticamente** associate al codice compilato di FORTRAN

- punto di ritorno, parametri, ambiente e memoria locale
 - si usano i **record di attivazione (frame)**
 - contenenti le stesse informazioni
- ma associati dinamicamente alle varie chiamate di sottoprogrammi
- Dato che l'accesso ai sottoprogrammi segue una politica LIFO
- l'ultima attivazione creata nel tempo è la prima che ritorna
- possiamo organizzare i record di attivazione in una pila

Modello di macchina hw



14

I dati in memoria sono divisi in due parti:

- **Heap:** per i dati che sono strutture dinamiche.
- **Stack:** per i dati che corrispondono ai meccanismi di gestione dei sottoprogrammi. Ovvero abbiamo un modo per definire l'ambiente locale del sottoprogramma il quale verrà messo nello Stack perché il meccanismo di chiamata di ritorno del sottoprogramma è LIFO.

L'*Environment Pointer* punta sempre all'ambiente locale del sottoprogramma che in quel momento è in esecuzione. Punta sempre al **record di attivazione (l'ambiente locale del sottoprogramma)** del sottoprogramma in esecuzione nello Stack.

Meccanismo call/return di sottoprogramma

- Chiamante
 - crea una istanza del record di attivazione
 - salva lo stato dell'unità corrente di esecuzione
 - effettua il passaggio dei parametri
 - inserisce il punto di ritorno
 - trasferisce il controllo al chiamato
- Chiamato (prologo)
 - salva il valore corrente di Environment Pointer (EP) e lo memorizza nel link dinamico
 - definisce il nuovo valore di EP
 - alloca le variabili locali
- Chiamato (epilogo)
 - eventuale passaggio di valori (dipende dalla modalità di passaggio dei parametri - lo vedremo dopo)
 - il valore calcolato dalla funzione viene trasferito al chiamante
 - ripristina le informazioni di controllo (il vecchio valore di EP salvato come link dinamico)
 - ripristina lo stato di esecuzione del chiamante
 - trasferisce il controllo al chiamante

(La valutazione dei parametri attuali avviene nell'ambiente del *chiamante*)

Record di attivazione per in-line block

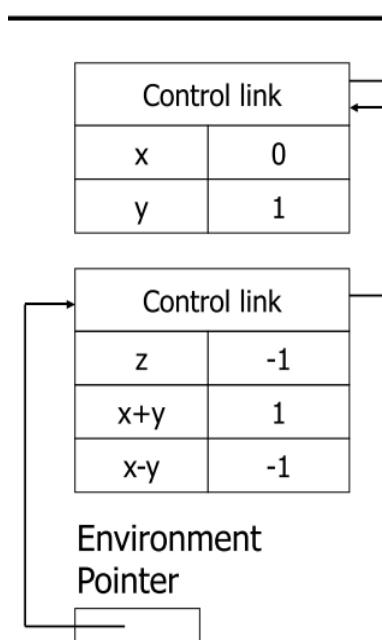
Vediamo ora come si realizza sui blocchi vedendoli come delle procedure senza nome e senza parametri.

- Esempio

```
{ int x = 0;
    int y = x+1;
    { int z = (x+y)*(x-y);
    };
};
```

Occorre prevedere spazio per memorizzare i risultati intermedi

Push AR con spazio per x, y
Assegna i valori a x, y
Push AR per blocco interno
con spazio per z
Assegna valore a z
Pop AR per blocco interno
Pop AR per blocco esterno



E le regole di scope?



- Variabili e ambiente

- x, y locali al blocco esterno
- z locale al blocco interno
- x, y non locali per il blocco interno

```
{ int x = 0;  
  int y = x+1;  
  { int z=(x+y)*(x-y);  
  };  
};
```

- **Static scope**

- riferimenti non locali si risolvono nel più vicino blocco esterno

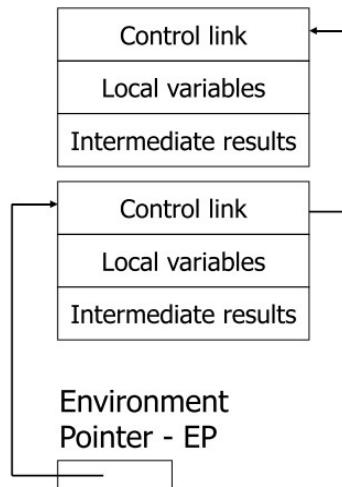
- **Dynamic scope**

- riferimenti non locali si risolvono nell'AR precedente sullo stack

Nel caso di in-line block le due nozioni coincidono

In generale:

Record di attivazione per in-line block



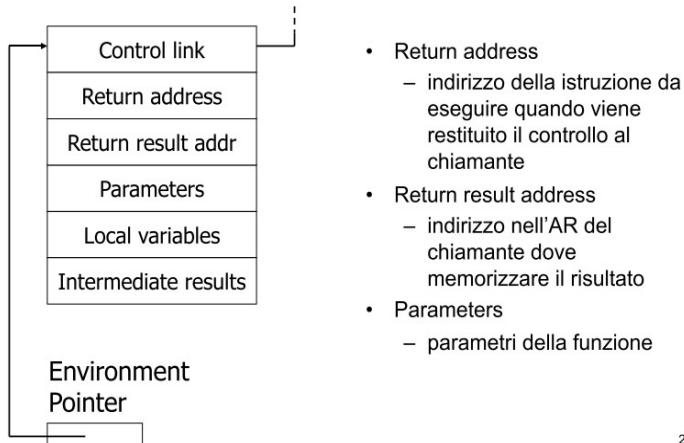
- Control link
 - puntatore (indirizzo base) a AR precedente nello stack
- Push AR
 - il valore di EP diviene il valore del control link del nuovo AR
 - modifica EP a puntare al nuovo AR
- Pop record off stack
 - il valore del nuovo EP viene ottenuto seguendo il control link

2'

Il Control Link viene chiamato anche Puntatore di catena dinamica

Record di attivazione per funzioni e procedure

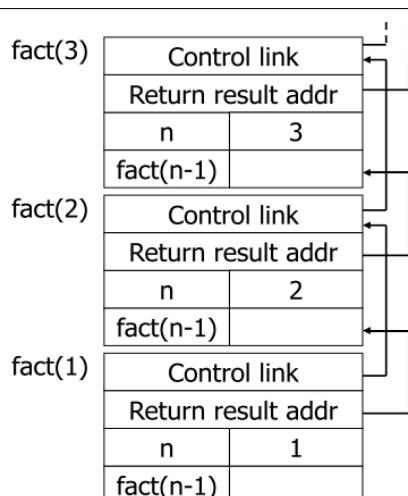
Funzioni: struttura AR



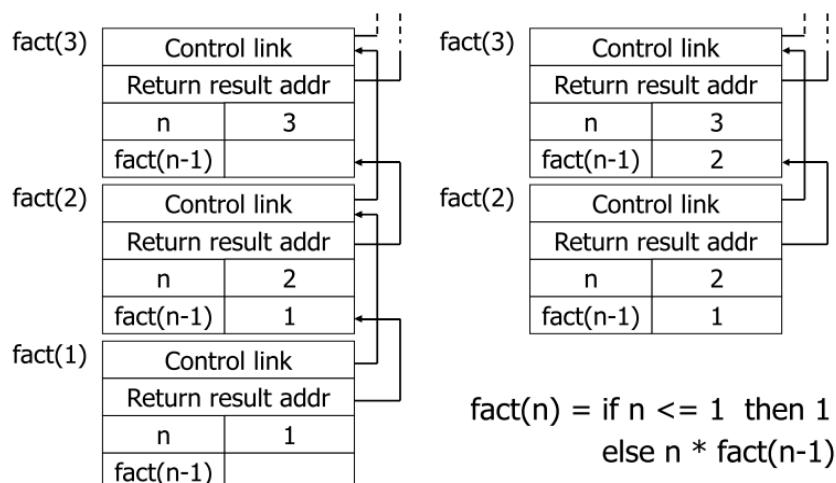
27

Esempio fact(k):

Chiamate...

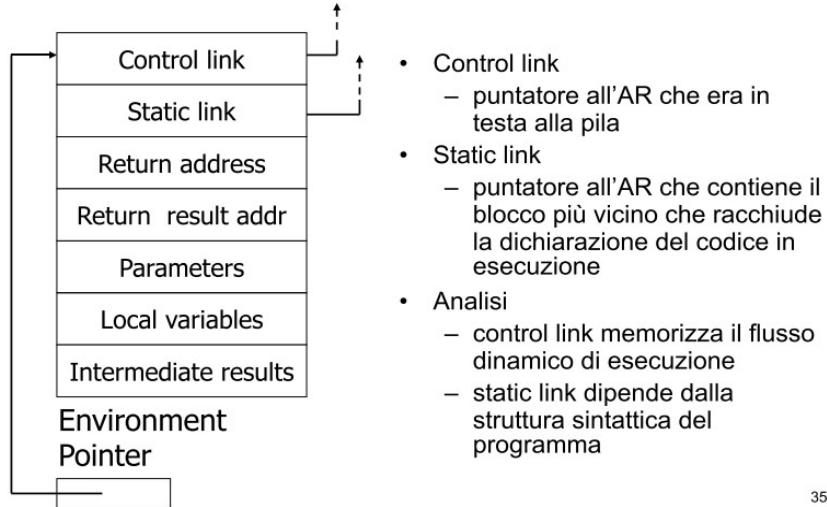


Ritorni..



Record di attivazione per funzioni e procedure con SCOPING STATICO

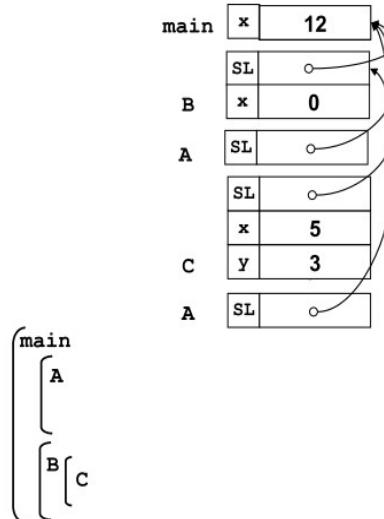
Scoping statico



35

Esempio

```
{ 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( );
}
```



E' il chiamante a determinare il link statico del chiamato

Static depth

- **Static depth (SD)** = profondità statica della dichiarazione
- SD può essere determinato staticamente: dipende solo dalla struttura sintattica del programma

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

Chiamato esterno al chiamante



- Le regole dello scoping statico assicurano che affinché il chiamato sia visibile si deve trovare in un blocco esterno che includa 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
- Assumiamo 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

(Quando si dichiara una funzione si attiva un record!!!)

Funzioni come valori

Nei linguaggi funzionali le funzioni sono valori esprimibili (possono essere risultato della valutazione di espressioni).

Ci sono due casi:

- una funzione è passata come parametro attuale.
- funzione restituita come risultato di un'altra funzione.

Funzioni come parametro attuale

Il valore di una funzione quando viene dichiarata è una coppia denominata **chiusura**:

< puntatore ambiente della dichiarazione, codice_funzione >

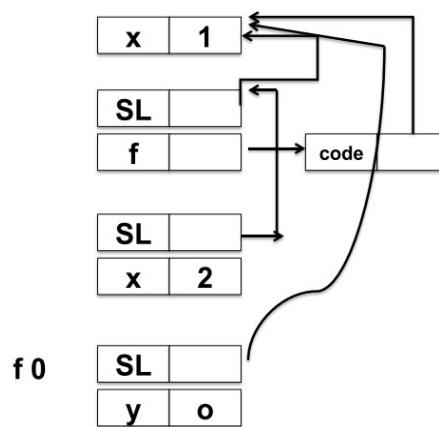
Quando una funzione è un parametro formale di un'altra funzione e viene invocata allora si alloca sullo stack l'AR della funzione e si mette come valore dello *Static Link* il “puntatore ambiente della dichiarazione”.

Lo *Static Link* di una chiamata di una funzione punta sempre a quello a cui punta il puntatore della chiusura di quella funzione.

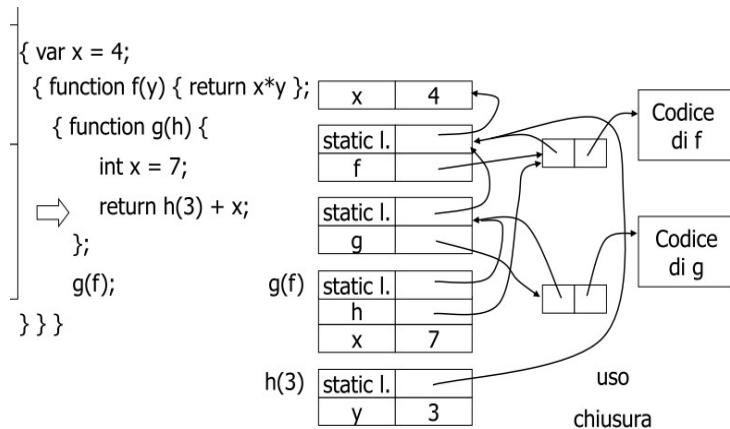
Se la funzione NON è ricorsiva allora il puntatore della chiusura punta all'AR precedente

OCAML: funzioni e chiusure

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



Se la funzione è ricorsiva oppure il linguaggio di programmazione non esplicita la ricorsione come fa Ocaml allora il puntatore della chisura punta all'AR della dichiarazione stessa.

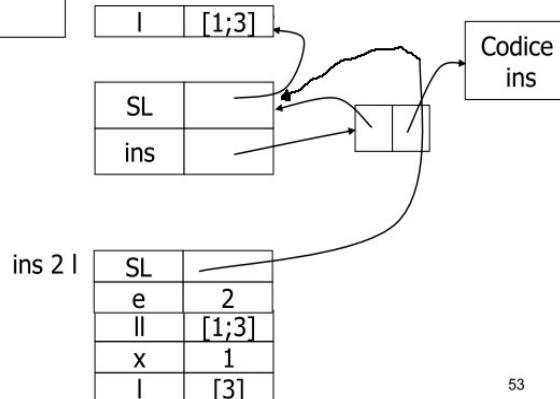


OCAML: Ricorsione



```

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



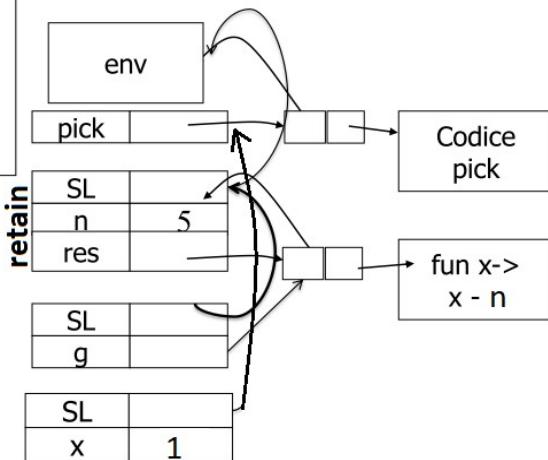
Funzioni come risultato

Esempio



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

Risultato 1



La chiusura di *res*, ovvero della funzione $\text{fun } x \rightarrow x - n$, ha un puntatore all'ambiente da dove poter prelevare le informazioni della funzione $\text{fun } x \rightarrow x - n$.

Meccanismo di retention: l'AR della funzione che produce come risultato un'altra funzione non è tolto dallo stack ma ha un flag che dice "retain", ovvero deve rimanere nello stack perché ha tutte le informazioni per l'ambiente locale della funzione restituita.

LEZIONE 24

Linguaggio funzionale didattico

```
type ide = string
type exp = Eint of int
| Ebool of bool
| Den of ide
| Prod of exp * exp
| Sum of exp * exp
| Diff of exp * exp
| Eq of exp * exp
| Minus of exp
| Iszero of exp
| Or of exp * exp
| And of exp * exp
| Not of exp
| Ifthenelse of exp * exp * exp
| Let of ide * exp * exp (* Dichiaraione di ide: modifica ambiente *)
| Fun of ide list * exp (* Astrazione di funzione *)
| Appl of exp * exp list (* Applicazione di funzione *)
```

Valori esprimibili e ambiente

- **Valori esprimibili**

```
type evT = Int of int
| Bool of bool
| Unbound
```

- **Ambiente**

evT env

I legami nell'ambiente sono *nomi ide*, valore della valutazione dell'*exp*.

I valori esprimibili sono i possibili risultati finali di una valutazione di una espressione, *Unbound* se non c'è un legami di ambiente e che genererà un'eccezione a tempo di esecuzione.

L'ambiente è una struttura che prende il tipo effettivo dei valori esprimibili e quindi è una funzione da identificatori a valori esprimibili.

Il blocco (Let)

E' la primitiva linguistica che mi permette di cambiare l'ambiente a tempo di esecuzione perchè introduce un nuovo legame nell'ambiente e mi dice qual è la portata del legame di ambiente introdotto (ovvero il body del Let).

Il blocco dato che ha una politica LIFO, viene gestito con uno Stack a tempo di esecuzione dove l'ambiente locale del blocco, ovvero le variabili dichiarate all'interno del blocco, corrispondono ai legami che metto sul record di attivazione di quell'ambiente.

$$\begin{array}{c}
 \boxed{\text{run-time stack}} \qquad \boxed{\text{push RA}} \\
 \frac{\text{env} \triangleright e1 \Rightarrow v1 \quad \text{env}[v1/x] \triangleright e2 \Rightarrow v2}{\text{env} \triangleright \text{Let}(x, e1, e2) \Rightarrow v2} \\
 \boxed{\begin{array}{l} \text{Usciamo dal blocco} \\ \text{Effettuando pop} \end{array}}
 \end{array}$$

- 1) Si valuta l'espressione $e1$ nell'ambiente esterno env e otteniamo un valore $v1$
- 2) Estendiamo l'ambiente di prima env con un nuovo legame $\text{env}[v1/x]$
- 3) Si valuta l'espressione $e2$ in questo nuovo ambiente e otteniamo un valore $v2$
- 4) Allora $\text{Let}(x, e1, e2) \rightarrow v2$

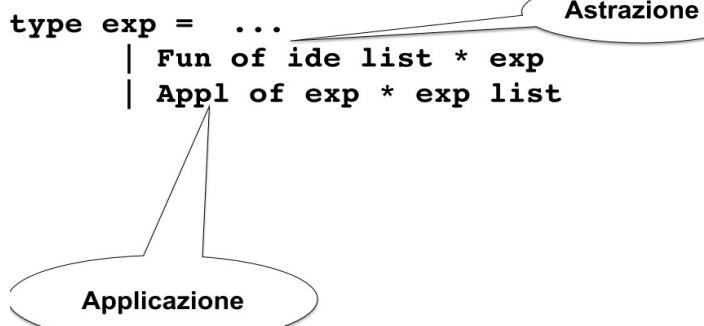
```

let rec eval((e: exp), (r: evT env)) =
  match e with
  | Eint(n) -> Int(n)
  | Ebool(b) -> Bool(b)
  | Den(i) -> applyenv(r, i)
  | Iszero(a) -> iszero(eval(a, r))
  | Eq(a, b) -> equ(eval(a, r), eval(b, r))
  | Prod(a, b) -> mult(eval(a, r), eval(b, r))
  | Sum(a, b) -> plus(eval(a, r), eval(b, r))
  | Diff(a, b) -> diff(eval(a, r), eval(b, r))
  | Minus(a) -> minus(eval(a, r))
  | And(a, b) -> et(eval(a, r), eval(b, r))
  | Or(a, b) -> vel(eval(a, r), eval(b, r))
  | Not(a) -> non(eval(a, r))
  | Ifthenelse(a, b, c) -> let g = eval(a, r) in
    if typecheck("bool", g) then
      (if g = Bool(true) then eval(b, r) else eval(c, r))
    else failwith ("nonboolean guard")
  | Let(i, e1, e2) ->
    eval(e2, bind(r, i, eval(e1, r)))
  
```

- L'espressione **e2** (corpo del blocco) è valutata nell'ambiente "esterno" esteso con l'associazione tra il nome **i** e il valore di **e1**

Funzioni non ricorsive

Sintassi



Identifieri (**parametri formali**) nel costrutto di **astrazione**: *Fun of ide list * exp*

Espressioni (**parametri attuali**) nel costrutto di **applicazione**: *Appl of exp * exp list*

Ovviamente bisogna estendere i tipi esprimibili **evT**.

Assumendo **scoping statico**:

$$\begin{aligned}
\text{type evT} &= \mid \text{Int of int} \mid \text{Bool of bool} \mid \text{Unbound} \mid \text{Funval of efun} \\
\text{and efun} &= \text{ide}^* \text{exp}^* \text{evT env}
\end{aligned}$$

La definizione di **efun** mostra che una astrazione funzionale è una **chiusura** che comprende:

- nome del parametro formale della funzione
- codice della funzione dichiarata
- ambiente al momento della dichiarazione

Semantica applicazione di funzione:

$$\frac{\begin{array}{c} \text{env} \triangleright e1 \Rightarrow v1 \quad v1 = \text{Funval}(x, e, \text{env1}) \\ \text{env} \triangleright e2 \Rightarrow v2 \quad \text{env1}[v2 / x] \triangleright e \Rightarrow v \end{array}}{\text{env} \triangleright \text{Apply}(e1, e2) \Rightarrow v}$$

Semantica dichiarazione di funzione:

$$\frac{\begin{array}{c} \text{env} \triangleright \text{Fun}(x, r) \Rightarrow \text{Funval}(x, e, \text{env}), \\ \text{env}[f = \text{Funval}(x, e, \text{env})] \triangleright \text{body} \Rightarrow v \end{array}}{\text{env} \triangleright \text{Let}(f, \text{Fun}(x, e), \text{body}) \Rightarrow v}$$

Semantica eseguibile:

```
let rec eval((e: exp), (r: evT env)) =
  match e with
  | ...
  | Fun(i, a) -> Funval(i, a, r)
  | Apply(e1, e2) -> match eval(e1, r) with
    | Funval(i, a, r1) ->
        eval(a, bind(r1, i, eval(e2, r)))
    | _ -> failwith("no funct in apply")
```

- Il corpo della funzione viene valutato nell'ambiente ottenuto legando i parametri formali ai valori dei parametri attuali nell'ambiente **r1**, nel quale era stata valutata l'astrazione

Funzioni ricorsive

C'è un problema:

```

let rec eval((e: exp), (r: evT env)) =
  match e with
  | Let(i, e1, e2) ->
    eval(e2, bind(r, i, eval(e1, r)))
  | Fun(i, a) -> Funval(i, a, r)
  | Appl(a, b) ->
    match eval(a, r) with
    | Funval(i, a, r1) ->
      eval(a, bind(r1, x, eval(b, r)))

```

- Il corpo **a** (che include **Den "fact"**) è valutato in un ambiente che è quello (**r1**) nel quale si valutano sia l'espressione **Let** che l'espressione **Fun**, esteso con una associazione per il parametro formale **x**. Tale ambiente non contiene il nome "**fact**" pertanto **Den "fact"** restituisce **Unbound!!!**

Quindi abbiamo bisogno che il corpo della funzione venga valutato in un ambiente nel quale è già stata inserita l'associazione tra il nome e la funzione.

Dunque bisogna avere un nuovo costrutto per dichiarare funzioni ricorsive oppure un diverso costrutto di astrazione per le funzioni ricorsive.

Estendiamo la sintassi astratta del linguaggio didattico con un opportuno costruttore:

```

type exp =
  :
  | Letrec of ide * ide * exp * exp

```

Estendiamo i tipi esprimibili:

```
type evT = ... | RecFunVal of ide * efun and efun = ide * exp * evT env
```

Estendiamo l'interprete e modifichiamo l'*Apply* (sotto denominato *FunCall*) :

```

FunCall(f, eArg) ->
  let fClosure = (eval f r) in
  (match fClosure with
   | FunVal(arg, fBody, fDecEnv) ->
     eval fBody (bind fDecEnv arg (eval eArg r)) |
   | RecFunVal(g, (arg, fBody, fDecEnv)) ->
     let aVal = (eval eArg r) in
     let rEnv = (bind fDecEnv g fClosure) in
     let aEnv = (bind rEnv arg aVal) in
     eval fBody aEnv |
   _ -> failwith("non functional value")) |
Letrec(f, funDef, letBody) ->
  (match funDef with
   | Fun(i, fBody) -> let r1 = (bind r f (RecFunVal(f, (i, fBody, r)))) in
     eval letBody r1 |
   _ -> failwith("non functional def"));;

```


LEZIONE 25

Passaggio dei parametri

Il passaggio dei parametri è il meccanismo che viene utilizzato per avere condivisioni di legami formali tra entità che dipendono dalle diverse attivazioni che dipendono dal flusso di esecuzione del programma (procedure..).

Il passaggio dei parametri non è altro che una dichiarazione dinamica di una variabile (il parametro formale) con associato il valore del parametro attuale.

Ci sono varie tecniche:

- **Passaggio per riferimento (Call by reference).**
- **Passaggio per costante.**
- **Passaggio per valore (Call by value).**
- **Passaggio per valore-risultato.**
- **Passaggio per nome.**

Passaggio per riferimento

Il parametro formale ha come tipo una locazione di memoria ed è quindi modificabile.

Il parametro attuale e formale condividono la stessa struttura (aliasing): tutte le modifiche fatte col parametro formale si ripercuoto sul dato anche se il cammino d'accesso è quello del parametro attuale.

Passaggio per costante

Meccanismo di valutazione eager: il parametro viene valutato nell'ambiente del chiamante. Presente nell'interprete didattico Ocaml.

Il parametro formale ha come tipo un valore non modificabile.

Il parametro attuale e formale devono avere il solito tipo.

Passaggio per valore

(C opera solo per valore perché anche con le locazioni si passa un indirizzo).

Meccanismo di valutazione eager.

Valuto il parametro attuale e lo associo al valore del parametro formale.

E' un'entità modificabile e quindi posso modificarlo usando il cammino di accesso nome-parametro formale ma non si ripercuote fuori.

Passaggio per valore-risultato

Valuto il parametro attuale e lo associo al valore del parametro formale.

Nella procedure modifco il valore e alla fine quando la procedura restituisce il controllo al chiamante vado a vedere qual è l'ultimo valore associato al nome-parametro formale e lo vado a mettere nella locazione associata al nome-parametro attuale.

E' simile al passaggio per riferimento solo che è molto più controllato il meccanismo di condivisione.

Passaggio per nome

Meccanismo di valutazione lazy: il parametro viene valutato nel chiamato.

Quindi l'espressione del parametro attuale viene valutata solo se all'interno del metodo/procedura effettivamente mi serve il valore che l'espressione si porta dietro.

Quindi lego il parametro formale ad una espressione e all'ambiente del chiamante (è una chiusura perché effettivamente l'espressione può essere vista come una funzione senza nome e senza parametri).

Esercizio

```
int z = 1;  
void p (int x) < body >;  
p(z)  
print(z)
```

```
x=x+1 ; z = z+2
```

Scrivere il corpo di p in modo tale che la chiamata si comporti differentemente in caso di
pass by value -- 3
pass by reference -- 4
pass by value-result – 2

```
int n;  
void p(int k); {n = n+1; k = k+4; print(n);}  
main{ n = 0; p(n); print(n);}
```

Risultato
call by value: 1 1
call by value-result: 1 4
call by reference: 5 5

```

int a[1..4];
int n;
void p(int b){
print(b); n = n+1; print(b); b = b+5;}
main{
a[1] = 10;
a[2] = 20;
a[3] = 30;
a[4] = 40;
n := 1;
p(a[n+2]);
print(a);
}
call by reference: 30 30 10 20 35 40
call by name: 30 40 10 20 30 45

```

a[1]	I1
a[2]	I2
a[3]	I3
a[4]	I4
n	I

b	I3
---	----

I1	10
I2	20
I3	35
I4	40
I	2

Call by reference

a[1]	I1
a[2]	I2
a[3]	I3
a[4]	I4
n	I

b	I3
---	----

I1	10
I2	20
I3	30
I4	45
I	2

Call by name

LEZIONE 27

Controllo dei tipi

Per ora abbiamo sempre considerato il controllo dei tipi in modo dinamico (a tempo di esecuzione) tramite una funzione primitiva *typechecker* che andava a controllare la correttezza del tipo che si aspettava.

Sarebbe però più comodo avere un controllo di tipi statico per evitare un overhead a run-time.

Vedremo che tenere una visione molto legata al sistema dei tipi aiuta anche nella realizzazione di un linguaggio di programmazione.

Sistema dei tipi

Un **sistema dei tipi** è un meccanismo che associa i *tipi* ai valori calcolati. (nel paradigma funzionale).

Il sistema dei tipi si propone di dimostrare che se un programma è tipato correttamente non ci sono errori di tipo dovuti ad un uso scorretto dei tipi a tempo di esecuzione.

In un'espressione come: *let x = e1 in e2*

Il tipo di e2 dipende dal tipo di e1.

Siccome x potrebbe essere ridichiarata all'interno di e2 bisogna introdurre un **ambiente di tipi** che tiene traccia dei tipi associati ad ogni identificatore nel programma (è un ambiente come quello a tempo di esecuzione ma statico).

L'ambiente dei tipi è proprio quello che nei compilatori viene chiamata **tabella dei simboli**, dove ad ogni simbolo presente nel programma viene associato il suo tipo.

Notazione:

- Noi scriveremo:

$$\Gamma = x_1 : \tau_1, x_2 : \tau_2 \dots x_k : \tau_k$$

- Per indicare la funzione

$$\Gamma(x_i) = \tau_i$$

- che associa il tipo τ_i al valore x_i

- Useremo

$$\Gamma, x : \tau$$

- per indicare l'estensione della funzione Γ con l'associazione $x : \tau$

- $(\Gamma, x : \tau)(x) = \tau$ e $(\Gamma, x : \tau)(y) = \Gamma(y)$ per $y \neq x$

Supponiamo che Γ sia un ambiente di tipo.

$$\boxed{\Gamma \vdash e : \tau}$$

Per indicare che l'espressione e ha tipo τ nell'ambiente di tipo Γ :

Definiamo il sistema per il controllo dei tipi (type checker) per il linguaggio funzionale didattico (senza funzioni)

$$\boxed{\Gamma \vdash n : int}$$

$$\boxed{\Gamma \vdash b : bool}$$

$$\boxed{\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}}$$

Il typechecker altro non è che un esecuzione simbolica del programma che calcola i tipi associati alle espressioni presenti nel programma.

Let

Blocco let

$$\boxed{\frac{\Gamma \vdash e_1 : \tau_1, \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2}}$$

Calcolo il tipo di e_1 e ottengo τ_1 poi estendo l'ambiente dei tipi col legame $x : \tau_1$.

Vado a vedere nell'ambiente esteso se e_2 mi da tipi τ_2 .

Allora l'espressione *let...* mi da tipo τ_2 .

Operatori binari e condizionali

Operatori binari e condizionale

$$\boxed{\frac{\Gamma \vdash e_1 : \tau_1, \Gamma \vdash e_2 : \tau_2,}{\oplus : \tau_1 \times \tau_2 \rightarrow \tau}} \\ \boxed{\frac{\Gamma \vdash e : \text{bool}, \Gamma \vdash e_1 : \tau, \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau}}$$

Funzioni

Una funzione ha tipo $\tau_1 \rightarrow \tau_2$ che prendere un argomento di tipo τ_1 restituisce un risultato di tipo τ_2 .

DEFINIZIONE

$$\boxed{\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{fun}(x : \tau_1) = e : \tau_1 \rightarrow \tau_2}}$$

INVOCAZIONE

$$\boxed{\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2, \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash \text{App}(e_1, e_2) : \tau_2}}$$

Programmiamo un type-checker in Ocaml

- La struttura dei tipi

```
type tval = TInt | TBool | FunT of tval * tval
```

- La definizione dell'ambiente dei tipi

```
type tenv = ide -> tval;;
let bind (r : tenv) (i : ide) (v : tval) =
  function x -> if x = i then v else r x;;
let tenv0 = fun (x: ide) -> raise EmptyEnv;;
```

E' uguale all'ambiente polimorfo sui valori dove al posto di *tval* avevamo '*t*' ($\text{ev}T$)

La sintassi delle espressioni con tipi

```
type texp = EInt of int
| EBool of bool
| Den of ide
| Add of texp * texp
| Sub of texp * texp
| Not of texp
| And of texp * texp
| Ifz of texp * texp * texp
| Eq of texp * texp
| Let of ide * texp * texp
| Fun of ide * tval * texp
| App of texp * texp
```

Ora vediamo come dalle regole passiamo all'interprete type-checker

let rec teval e tenv =
match e with

$$\boxed{\Gamma \vdash n : \text{int}}$$

EInt i -> TInt

$$\boxed{\Gamma \vdash b : \text{bool}}$$

| EBool b -> TBool

$$\boxed{\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}}$$

| Den s -> tenv s

Il blocco let

$$\boxed{\frac{\Gamma \vdash e_1 : \tau_1, \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2}}$$

Let(i, e1, e2) -> let t = teval e1 tenv in
 teval e2 (bind tenv i t)

Operatori binari

$$\boxed{\frac{\Gamma \vdash e_1 : \tau_1, \Gamma \vdash e_2 : \tau_2, \oplus : \tau_1 \times \tau_2 \rightarrow \tau}{\Gamma \vdash e_1 \oplus e_2 : \tau}}$$

Add (e1,e2) -> let t1 = teval e1 tenv in
 let t2 = teval e2 tenv in
 (match (t1, t2) with
 (TInt, TInt) -> TInt
 | (_, _) -> raise WrongType)

Eq (e1,e2) ->
 if ((teval e1 tenv) = (teval e2 tenv))
 then TBool
 else raise WrongType

Condizionale

$$\frac{\Gamma \vdash e : \text{bool}, \Gamma \vdash e_1 : \tau, \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau}$$

```
Ifz (e1,e2,e3) -> let t = teval e1 tenv in
  ( match t with
    TBool -> let t1 = teval e1 tenv in
      let t2 = teval e2 tenv in
        ( match (t1, t2) with
          | (TInt, TInt) -> TInt
          | (TBool, TBool) -> TBool
          | (_,_) -> raise WrongType )
          | _ -> raise WrongType )
```

Funzioni

$$\frac{\Gamma, x: \tau_1 \vdash e: \tau_2}{\Gamma \vdash \text{fun}(x: \tau_1) = e: \tau_1 \rightarrow \tau_2}$$

Fun(i, t1, e) -> let tenv1 = bind tenv i t1 in
let t2 = teval e tenv1 in FunT(t1,t2)

$$\frac{\Gamma \vdash e_1: \tau_1 \rightarrow \tau_2, \Gamma \vdash e_2: \tau_1}{\Gamma \vdash \text{App}(e_1, e_2): \tau_2}$$

App(e1, e2) ->
let f = teval e1 tenv in
(match f with
 FunT(t1,t2) -> if ((teval e2 tenv) = t1)
 then t2 else raise WrongType
 | _ -> raise WrongType)

LEZIONE 29

Il paradigma ad oggetti

Fino ad ora abbiamo visto come si implementa l'astrazione procedurale nei linguaggi di programmazione imperativo e funzionale facendo vedere che il meccanismo di implementazione della chiamata al ritorno del sottoprogramma richiede uno stack con i record di attivazioni con tutte le informazioni per la gestione dei legami tra i nomi delle variabili dichiarate nell'ambiente, i parametri, i valori di ritorno ecc.

Nel caso dei linguaggi funzionali abbiamo un caso particolare della gestione dei record di attivazione ovvero quello in cui una funzione restituisce un risultato che è una funzione.

In questo caso il classico meccanismo di togliere il record di attivazione dallo stack quando la funzione restituisce il controllo al chiamante, non va più bene perchè il valore restituito è una chiusura che tiene come informazione di ambiente il puntatore al record di attivazione della funzione che ha generato la funzione risultato e quindi non deve essere tolto ma deve essere retain in modo tale che quando la funzione restituita come valore verrà utilizzata, avrà a disposizione l'ambiente in cui era stata generata.

Ora vediamo l'implementazione del linguaggio ad oggetti *Class Based*.

L'istanziazione della classe avviene attraverso la chiama del costruttore, ad esempio:

new Classe(parametri_attuali)

E' simile alla nozione di attivazione perchè come nel paradigma imperativo/funzionale le funzioni si scrivono una volta e si possono attivare (lanciare) più volte, anche la classe si scrive una volta e si può attivare (istanziare) più volte.

Inoltre quando si crea un oggetto di istanza di una classe usiamo un metodo costruttore, il quale definisce i valori iniziali delle variabili d'istanza, e quindi è come si stesse creando un ambiente particolare che corrisponde ai legami tra i nomi delle variabili d'istanza dell'oggetto e i valori. Questo per ogni creazione di un oggetto di una classe.

E' esattamente la stessa operazione di quando chiamiamo un sottoprogramma perchè si crea un record di attivazione che crea un nuovo ambiente per la chiamata della procedura.

La differenza rispetto alla chiamata di ritorno di un sottoprogramma è che l'oggetto (e quindi il suo l'ambiente) persiste nella memoria.

In termini di struttura a run-time è un **ambiente locale statico** che permane in memoria fin che non muore (Garbage Collector).

Quindi l'invocazione del metodo costruttore deve creare l'ambiente locale statico e restituire a chi ha invocato il metodo costruttore, la maniglia di accesso all'oggetto creato.

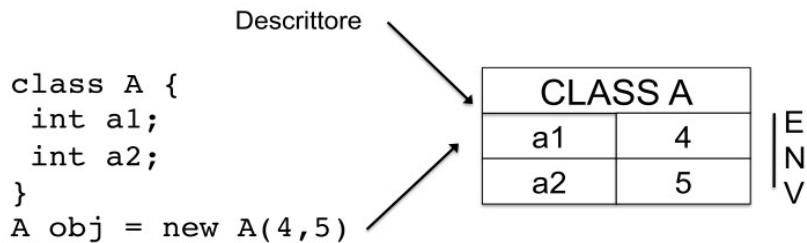
La definizione di una classe però comprende anche di metodo, oltre alle

variabili d'istanza.

I metodi sono differenti dalle variabili d'istanza perché le istanze di un oggetto di una classe hanno differenti valori per le variabili d'istanza ma condividono gli stessi metodi.

Consideriamo per il momento definizioni di classi senza metodi e modificatori:

- Soluzione: ambiente locale statico che contiene i binding delle variabili di istanza
 - con associato il descrittore di tipo



```
class A {  
    int a1;  
    int a2;  
}  
  
class B extends A {  
    int a3  
}
```

CLASS B	
Ereditati	a1
Ereditati	a2
	a3

Soluzione: i campi ereditati dall'oggetto vengono inseriti all'inizio nell'ambiente locale dell'oggetto creato

11

L'utilizzo di un ambiente locale statico permette di implementare facilmente la persistenza dei valori.

- La gestione dell'ereditarietà (singola) è immediata.
- La gestione dello *shadowing* (variabili d'istanza con lo stesso nome usata nella sottoclassa) è immediata.

Se il linguaggio prevede meccanismo di controllo statico si può facilmente implementare un accesso diretto: *indirizzo di base* (dato dal nome del puntatore) + *offset* (campo).

Vediamo ora un problema:

Implementazioni multiple



```

interface IntSet {
    public IntSet insert(int i);
    public boolean has(int i);
    public int size();
}

class IntSet1 implements IntSet {
    private List<Integer> rep;
    public IntSet1() {
        rep = new LinkedList<Integer>();
    }
    public IntSet1 insert(int i) {
        rep.add(new Integer(i));
        return this;
    }
    public boolean has(int i) {
        return rep.contains(new Integer(i));
    }
    public int size() {return rep.size();}
}

class IntSet2 implements IntSet {
    private Tree rep;
    private int size;
    public IntSet2() {
        rep = new Leaf(); size = 0;
    }
    public IntSet2 insert(int i) {
        Tree nrep = rep.insert(i);
        if (nrep != rep) {
            rep = nrep; size += 1;
        }
        return this;
    }
    public boolean has(int i) {
        return rep.find(i);
    }
    public int size() {return size;}
}

```

13

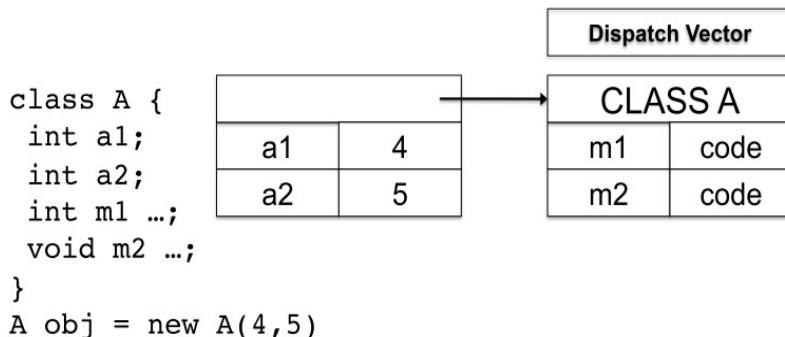
Il cliente non ha informazioni sufficienti per risolvere quale metodo viene invocato tra IntSet1.size() e IntSet2.size().

Quindi il metodo dipende dall'oggetto specifico.

Dunque, avendo una dichiarazione di una classe, quando devo implementare la creazione di un'istanza di questa classe devo tenere di conto la gestione dei metodi. Ovvero l'invocazione di un metodo deve "passare" dagli oggetti.

Soluzione: associare un puntatore alla *tabella dei metodi* (sinonimi: *vtable*, *dispatch table*) che contiene il binding dei metodi e il descritto con altre informazione associate alla classe.

(E' un altro ambiente ma che non è associato all'oggetto che viene creato ma alla classe)



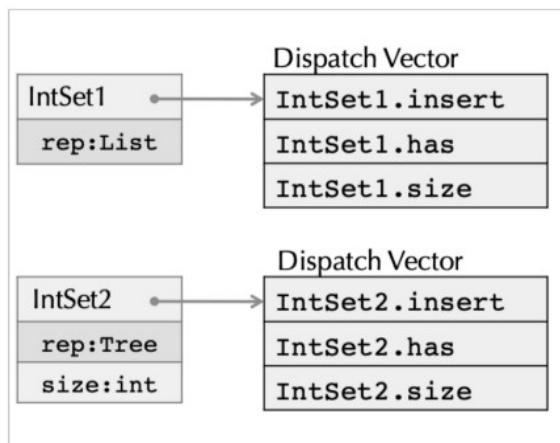
Quando faccio il metodo costruttore devo creare l'ambiente locale statico per i

valori delle variabili d'istanza e nel descrittore dell'oggetto che viene creato devo mettere il puntatore ad una struttura di implementazione che tiene traccia di tutti i metodi che sono dichiarati per quella classe.

Se la vtable associata ad una classe è la stessa per tutte le istanze di un oggetto e viene invocato un metodo come fa a sapere quale variabili di istanza deve prendere?

Nella struttura di implementazione di un metodo (il codice del metodo) ha un parametro che mette il compilatore riferito al parametro dell'oggetto corrente. In modo tale che quando un metodo viene invocato gli si passa anche il riferimento all'oggetto (this) che l'ha chiamato, in modo tale da "pescare" le variabili di istanza corrette.

Per i metodi statici non gli viene passato l'oggetto corrente perché fa riferimento direttamente alla classe e quindi anche le variabili statiche sono all'intero della vtable.



Un metodo è eseguito come una funzione (implementazione standard: AR sullo stack con variabili locali, parametri ecc.).

Altro problema:

Ereditarietà

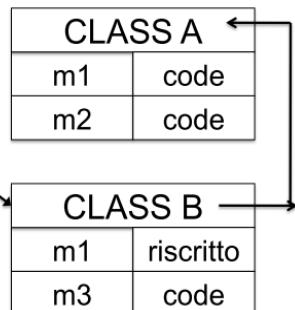


• Soluzione 1 (Smalltalk)

- o lista di tabelle

```
class A {  
    int a1, a2;  
    void m1 ...;  
    void m2 ...;  
}
```

a1	
a2	
a3	



```
class B extends A  
    int a3;  
    void m1 ...;  
    void m3 ...;  
}  
new B(...)
```

PRO: meno memoria occupata
CONTRO: overhead per la ricerca di un metodo

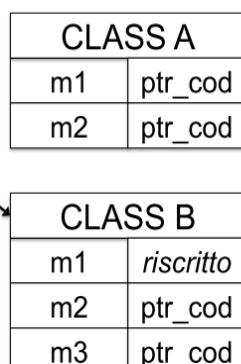
18

• Soluzione 2 (C++ e Java)

- o sharing strutturale

```
class A {  
    int a1, a2;  
    void m1 ...;  
    void m2 ...;  
}
```

a1	
a2	
a3	

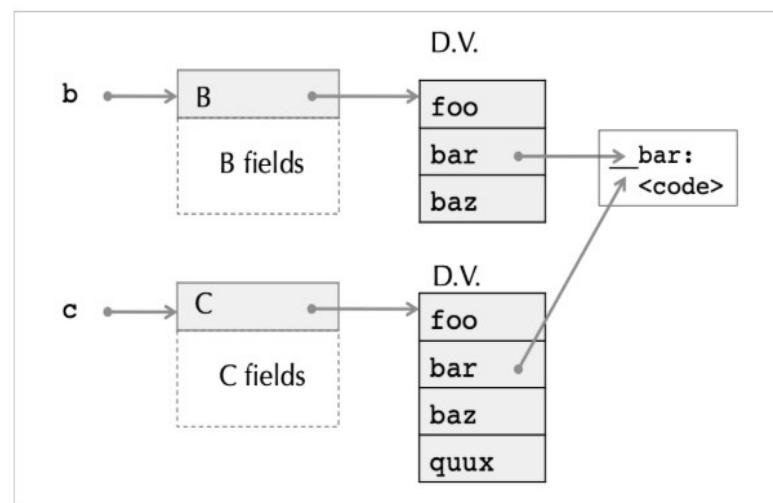


```
class B extends A  
    int a3;  
    void m1 ...;  
    void m3 ...;  
}  
new B(...)
```

PRO: trovo subito il metodo ricercato
CONTRO: più memoria occupata

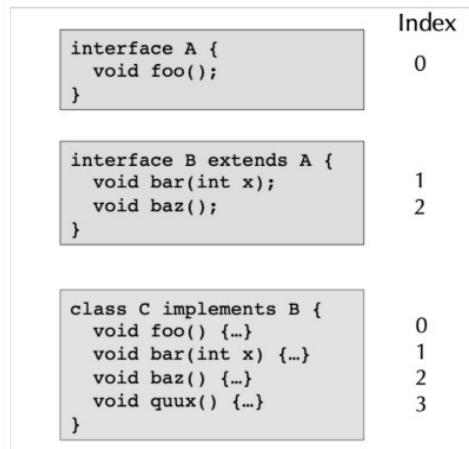
19

Sharing strutturale



Dispatching in dettaglio

Idea: ogni metodo è caratterizzato da un indice unico
Indice permette di identificare il metodo nella vtable



Così il compilatore ha un modo per generare del codice efficiente in modo tale che tutte le operazioni di accesso ai metodi vengano date in base alla struttura degli indici creati in fase di analisi statica (compilazione).

Problema d'ereditarietà multipla:

```
Class A { int m(); }  
Class B { int m(); }  
Class C extends A,B { ... }
```

Quale metodo *m* viene invocato da *C*?

La soluzione di Java è questa:

- Una classe può estendere una sola classe
- Una classe può implementare più classi e se le interfacce contengono lo stesso metodo, la classe avrà una sola implementazione.

Ma per quanto riguarda gli indici?

```

interface Shape {                                D.V.Index
    void setCorner(int w, Point p);             0
}

interface Color {
    float get(int rgb);                      0
    void set(int rgb, float value);           1
}

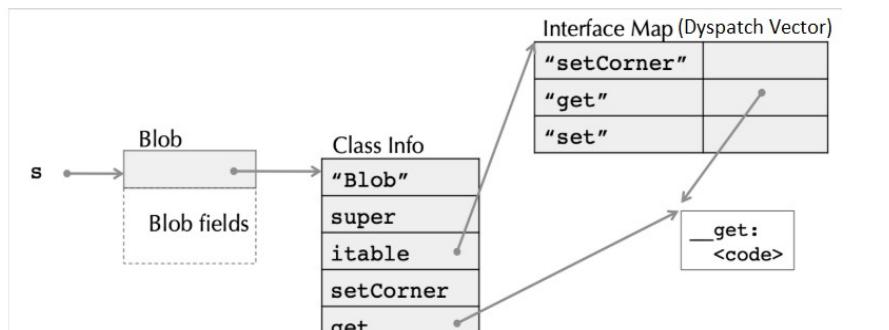
class Blob implements Shape, Color {
    void setCorner(int w, Point p) {...}      0?
    float get(int rgb) {...}                  0?
    void set(int rgb, float value) {...}       1?
}

```

Soluzione 1



Tabella di supporto Interface Table per associare
il codice ai metodi



Soluzione 2: hashing per indici



```

interface Shape {                                D.V.Index
    void setCorner(int w, Point p);      hash("setCorner") = 11
}

interface Color {
    float get(int rgb);                 hash("get") = 4
    void set(int rgb, float value);     hash("set") = 7
}

class Blob implements Shape, Color {
    void setCorner(int w, Point p) {...}   11
    float get(int rgb) {...}              4
    void set(int rgb, float value) {...}  7
}

```

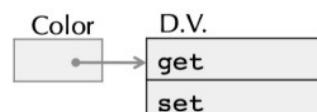

Soluzione 3: duplicare i dispatch vector



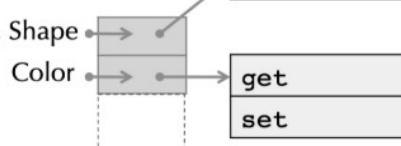
```
interface Shape {  
    void setCorner(int w, Point p);  
}  
D.V. Index 0
```



```
interface Color {  
    float get(int rgb);  
    void set(int rgb, float value);  
}  
D.V. Index 0  
1
```



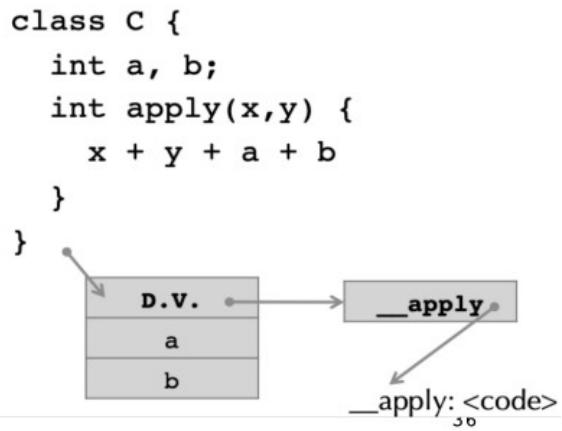
```
class Blob implements Shape, Color {  
    void setCorner(int w, Point p) {...}  
    float get(int rgb) {...}  
    void set(int rgb, float value) {...}  
}  
Blob, Shape D.V. Index 0
```



In questa l'unico problema è che se le due interfacce hanno un metodo col solito nome e *Blob* lo implementa allora il compilatore dovrà decidere in quale *Dispatch Vector* inserirlo.

Osservazione sulle chiusure

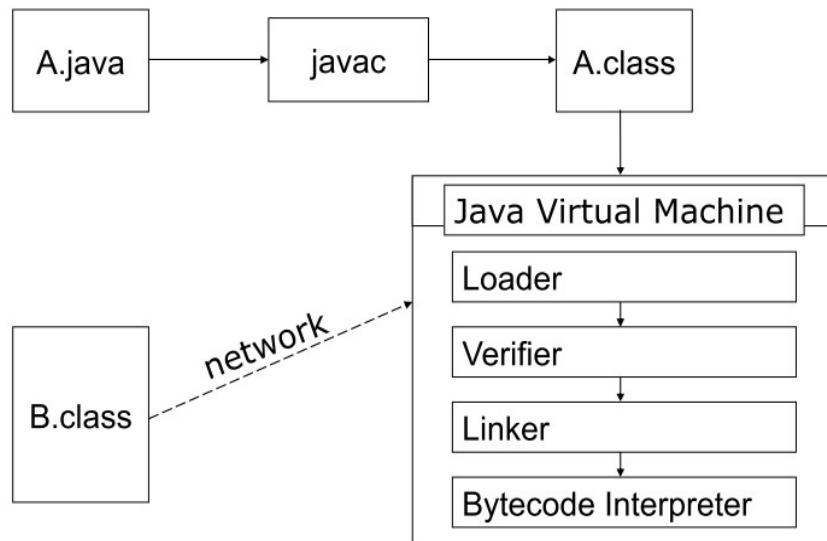
L'invocazione di un metodo è esattamente una chiusura, perchè ha il codice del metodo e il puntatore all'ambiente locale dell'oggetto dove ci sono i binding per le variabili d'istanza.



Java Virtual Machine

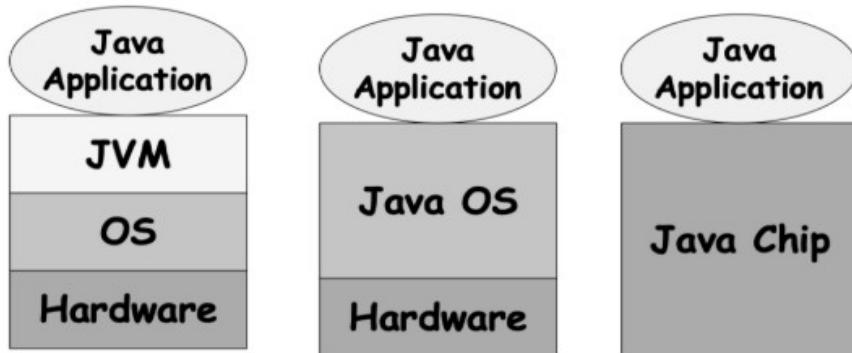
E' la macchina astratta di Java e descrive com'è fatto l'interprete del bytecode, quali sono le istruzioni, qual è il flusso di esecuzione e quale è il formato dei file.

JVM: visione di insieme



- **Loader:** carica come prima classe quella contenente il metodo *main*.
- **Verifier:** verifica che il bytecode rispetti certe proprietà.
- **Linker:** collega ciò che prima era un nome simbolico al suo valore effettivo.
- **Bytecode Interpreter:** interprete.

La specifica della JVM non dice come deve essere implementata ma dice solo quali sono i vincoli, com'è fatta la struttura, come sono fatti i class file e le istruzioni ecc.

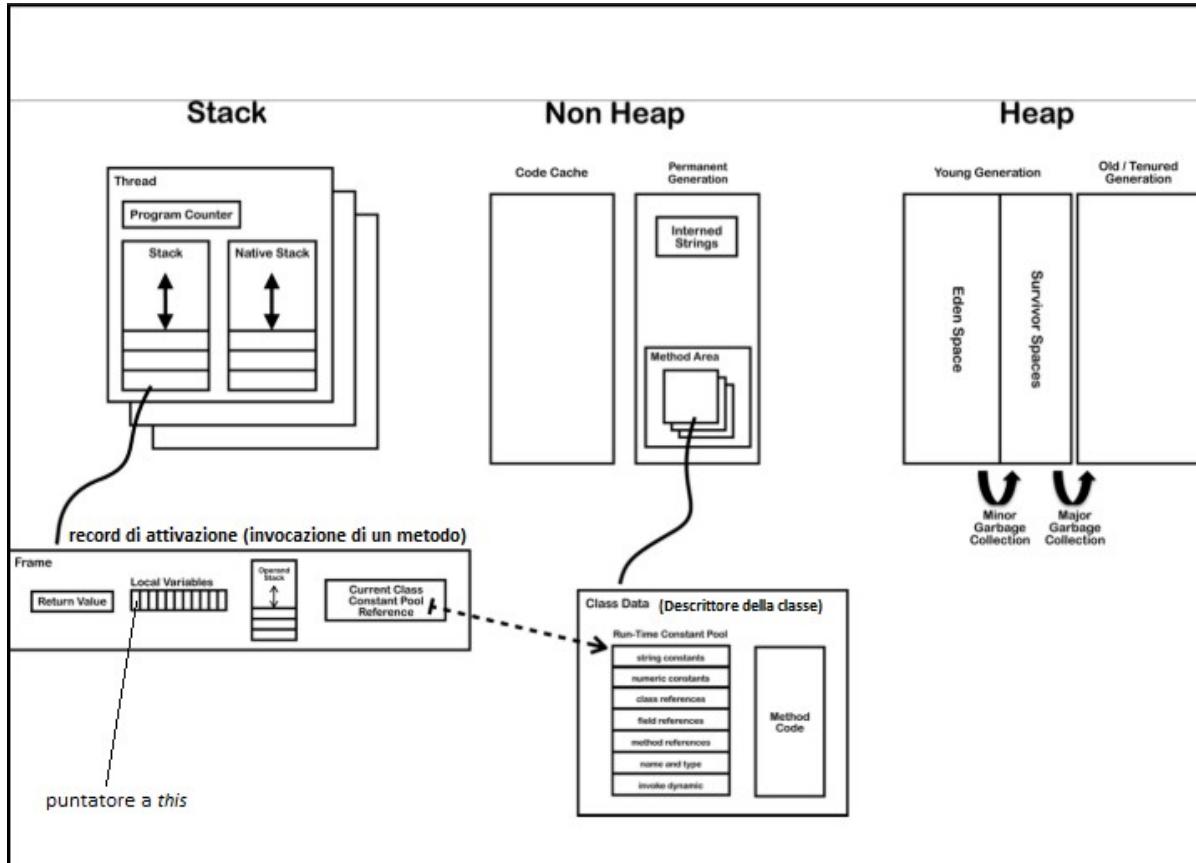


La JVM è una macchina a stack e quindi l'insieme delle istruzioni del bytecode assumono una pila, in esecuzione.

Gli argomenti di una istruzione si trovano in testa allo stack, il risultato viene messo in testa allo stack.

Lo stack degli operandi viene utilizzato per:

- trasmissione dei parametri ai metodi
- restituzione del risultato dell'invocazione di un metodo
- memorizzazione dei risultati intermedi
- memorizzazione delle variabili locali



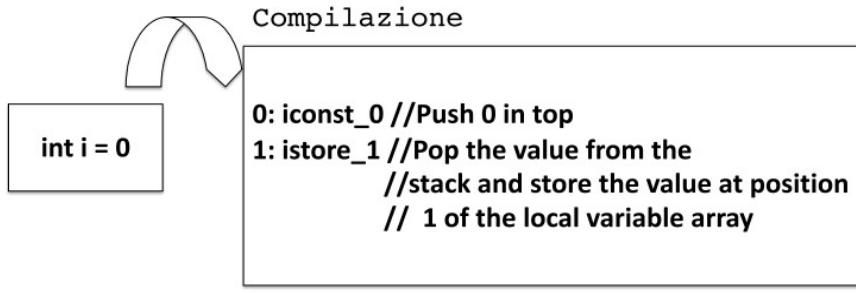
Local Variables

Per i metodi NON statici la posizione 0 è utilizzata per la gestione di *this*.

Per i metodi statici le variabili locali partono dall'indice 0.

Operand Stack

Lo stack degli operandi è utilizzato dalle istruzioni del bytecode per la gestione degli argomenti e risultati delle operazioni



- Spinge 0 nella pila locale del record di attivazione.
- Tira fuori il valore (in questo caso 0) e lo mette alla posizione 1 dell'array delle variabili locali (lo mette nell'1 perchè nello 0 c'è *this*).

(Intermezzo) Stack Machine

Instruction	Stack Before	Stack Later
<i>STCstInt(n)</i>	<i>s</i>	$\rightarrow s, n$
<i>STAdd</i>	<i>s, n₁, n₂</i>	$\rightarrow s, (n_1 + n_2)$
<i>STSub</i>	<i>s, n₁, n₂</i>	$\rightarrow s, (n_1 - n_2)$
<i>STMul</i>	<i>s n₁, n₂</i>	$\rightarrow s, (n_1 * n_2)$
<i>STDup</i>	<i>s, n</i>	$\rightarrow s, n, n$
<i>STSwap</i>	<i>s, n₁, n₂</i>	$\rightarrow s, n_2, n_1$

Istruzioni operano sullo stack
 Stack = controllo trasferimento dati

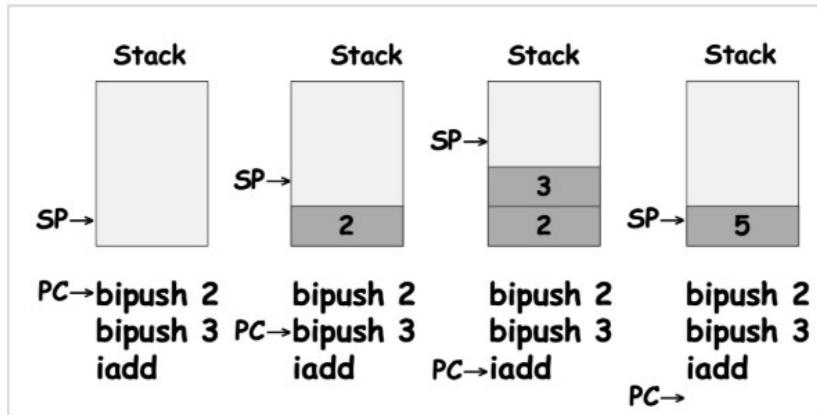
Vediamolo in Ocaml:

Sintassi astratta

```
type stkinstr =
| STCstInt of int
| STAdd
| STSub
| STMul
| STDup
| STSwap
```

```
let rec stkEval (control: stkinstr list)
               (stack: int list): int =
  match (control, stack) with
  | ([], v::_) -> v
  | ([], []) -> failwith("no result on the stack")
  | (STCstInt(n)::cs, stack)-> stkEval cs (n::stack)
  | (STAdd::cs, n1::n2::ss) -> stkEval cs ((n1+n2)::ss)
  | (STSub::cs, n1::n2::ss) -> stkEval cs ((n1-n2)::ss)
  | (STMul::cs, n1::n2::ss) -> stkEval cs ((n1*n2)::ss)
  | (STDup::cs, n::ss) -> stkEval cs (n::n::ss)
  | (STSwap::cs, n1::n2::ss) -> stkEval cs (n2::n1::ss)
  | (_::_, []) -> failwith("too few operands")
```

Esempio: bytecode



Linking

I riferimenti simbolici presenti nel codice devono essere sostituiti con indirizzi assoluti reali.

Linking Dinamico

Ogni frame (ogni attivazione di un metodo) contiene un puntatore a una struttura chiamata: **Constant Pool**.

La Constant Pool è il descrittore di tipo associato alla classe dove è definito il metodo in esecuzione.

Quando compilo una classe metto tutti i riferimenti simbolici (tutti i riferimenti a variabili e metodi presenti nel codice della classe) nella Constant Pool.

Poi ad ogni riferimento simbolico bisogna legarci un indirizzo fisico e lo possiamo fare in due modi diversi:

- a **loading time**: eager resolution (C++)
- **on demand**: il legame viene risolto la prima volta che si incontra un link simbolico durante l'esecuzione (lazy resolution) (JVM)

Ogni riferimento risolto diviene un offset rispetto alla struttura di memorizzazione a runtime.

Class Area

E' quella porzione di memoria dove sono memorizzate le classi.

Per ogni classe abbiamo:

- constant pool
- i campi dell'oggetto
- i dati dei metodi
- i codici dei metodi

In modo tale da avere tutto a disposizione quando carico, nello Heap, un'istanza di una classe.

Il bytecode generato dal compilatore Java viene memorizzato in una *class file* (.class) contenente:

- bytecode dei metodi della classe
- constant pool: una sorta di tavella dei simboli che descrive le costanti e altre informazioni presenti nel codice della classe.

ClassFile {	
u4 magic;	0xCAFEBABE
u2 minor_version;	Java Language Version
u2 major_version;	
u2 constant_pool_count;	Constant Pool
cp_info contant_pool[constant_pool_count-1];	
u2 access_flags;	access modifiers and other info
u2 this_class;	References to Class and Superclass
u2 super_class;	
u2 interfaces_count;	References to Direct Interfaces
u2 interfaces[interfaces_count];	
u2 fields_count;	Static and Instance Variables
field_info fields[fields_count];	
u2 methods_count;	Methods
method_info methods[methods_count];	
u2 attributes_count;	Other Info on the Class
attribute_info attributes[attributes_count];	
}	

All'interno di un file .class


```

Compiled from "SimpleClass.java"
public class SimpleClass {
    minor version: 0
    major version: 52
    flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
#1 = Methodref      #6.#14      // java/lang/Object."<init>":()V
#2 = Fieldref       #15.#16     // java/lang/System.out:Ljava/io/PrintStream;
#3 = String          #17          // Hello
#4 = Methodref       #18.#19     // java/io/PrintStream.println:(Ljava/lang/String;)V
#5 = Class           #20          // SimpleClass
#6 = Class           #21          // java/lang/Object
#7 = Utf8            <init>
#8 = Utf8            ()V
#9 = Utf8            Code
#10 = Utf8           LineNumberTable
#11 = Utf8           sayHello
#12 = Utf8           SourceFile
#13 = Utf8           SimpleClass.java
#14 = NameAndType    #7:#8       // "<init>":()V
#15 = Class           #22          // java/lang/System
#16 = NameAndType    #23:#24     // out:Ljava/io/PrintStream;
#17 = Utf8            Hello
#18 = Class           #25          // java/io/PrintStream
#19 = NameAndType    #26:#27     // println:(Ljava/lang/String;)V
#20 = Utf8           SimpleClass
#21 = Utf8           java/lang/Object
#22 = Utf8           java/lang/System
#23 = Utf8           out
#24 = Utf8           Ljava/io/PrintStream;
#25 = Utf8           java/io/PrintStream
#26 = Utf8           println
#27 = Utf8           (Ljava/lang/String;)V

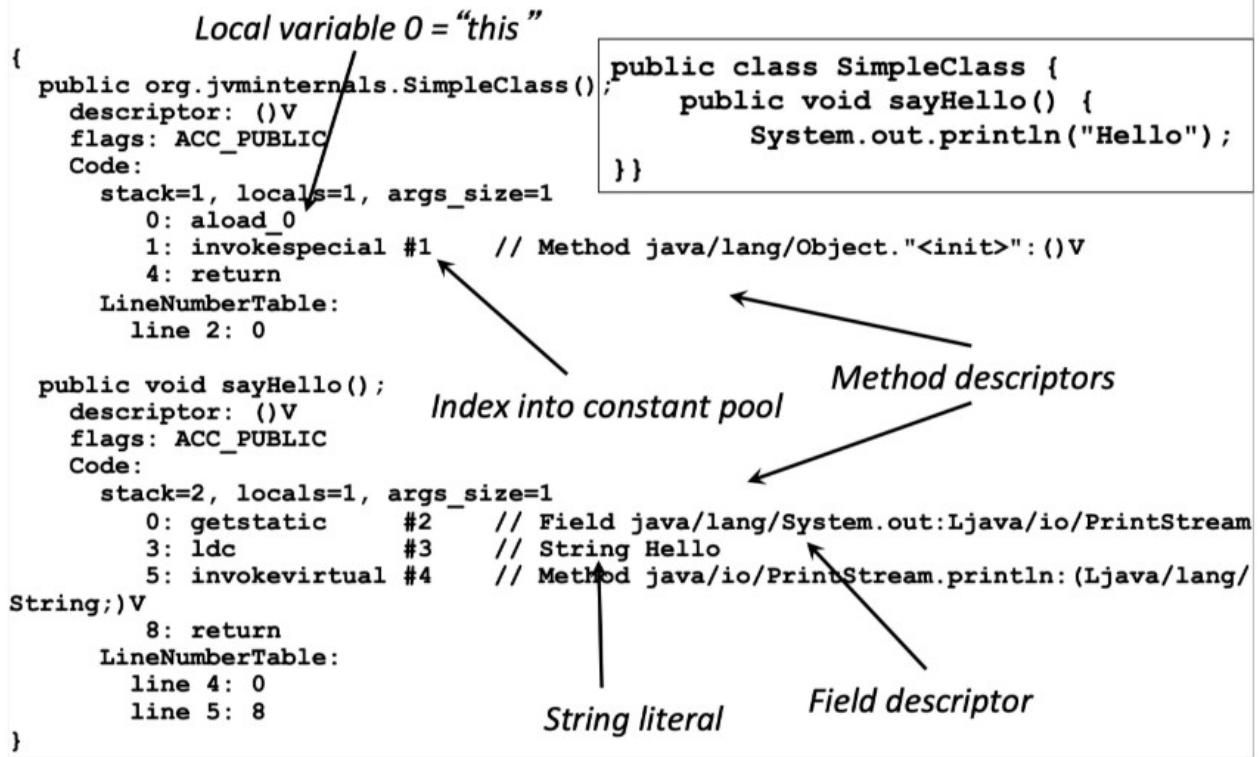
```

<pre> public class SimpleClass { public void sayHello() { System.out.println("Hello"); } } </pre>	<pre> public void sayHello(); descriptor: ()V Code: stack=2, locals=1, args_size=1 0: getstatic #2 3: ldc #3 5: invokevirtual #4 8: return </pre>
---	--

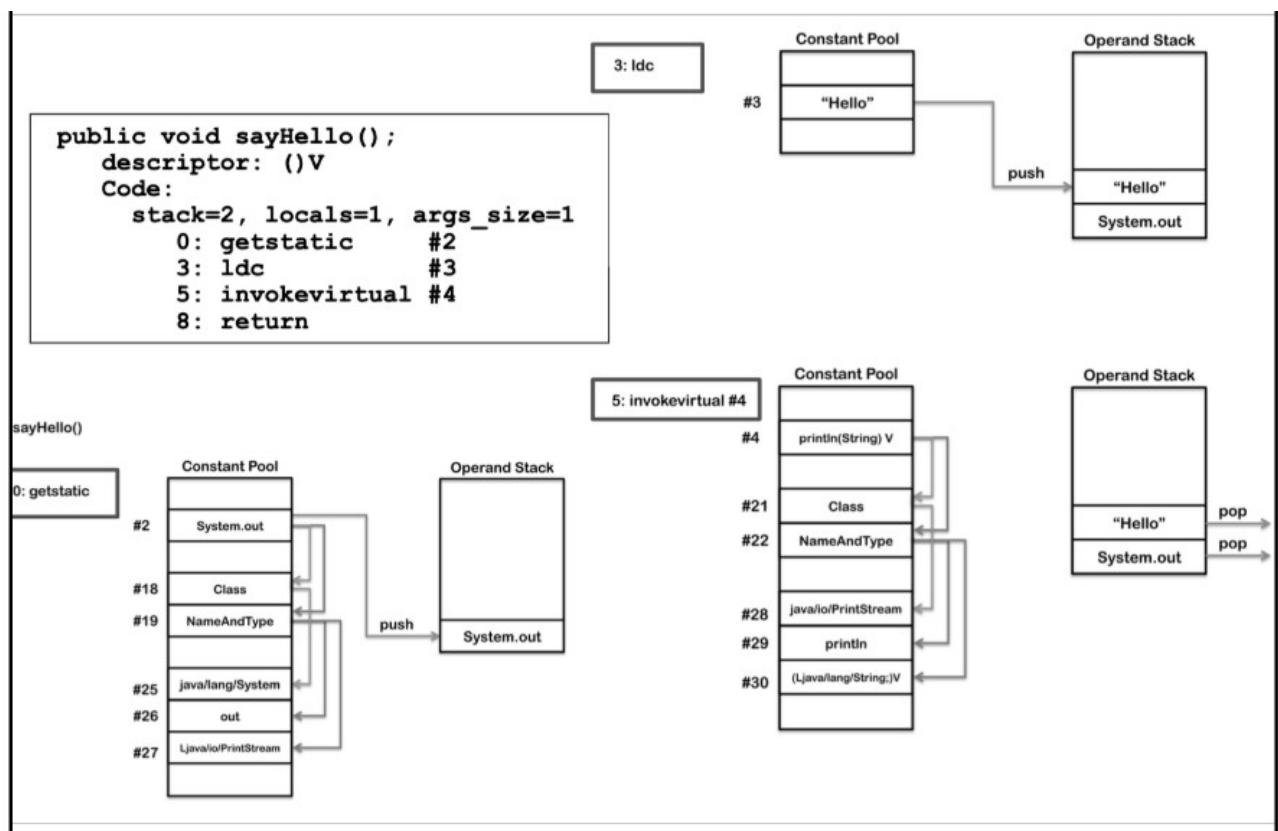
14

Ad esempio in #1 : #6 → #21 → E' una classe Object ; #14 → #7 e #8 → <init> (è un costruttore) che restituisce ()V (void)

Inoltre ci sono altri riferimenti ai singoli metodi:



Nel dettaglio:



Quindi la **Constant Pool** è sostanzialmente l'ambiente simbolico dei nomi dove per ogni nome c'è scritto il suo riferimento "simbolico" ed è l'informazione

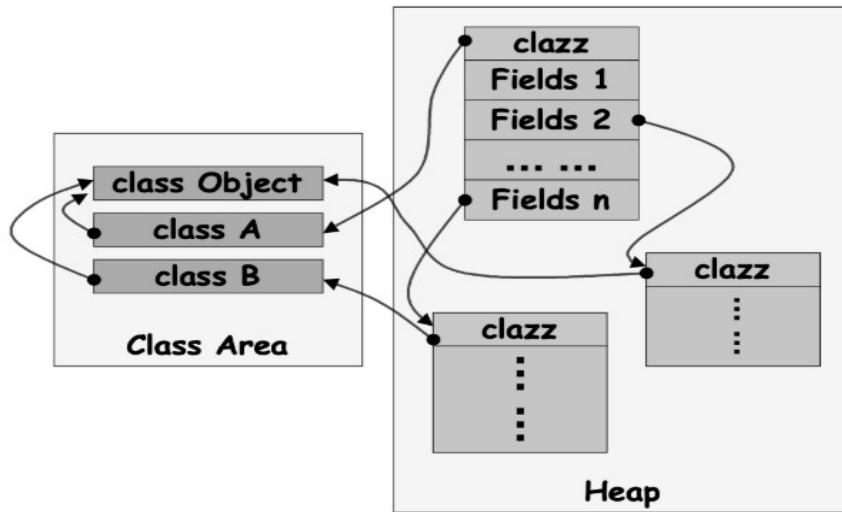
che ho bisogno in fase di compilazione nella tabella dei simboli per risolvere il problema del tipo.

Serve nel *Class Loading* per il processo di risoluzione dei nomi: quando carico la classe, vado a vedere la constant pool, vedo quali sono le classi padre e carico anche quelle e poi ho tutte le informazioni che mi permettono di risolvere i nomi simbolici che ho presente lì dentro.

L'**invokevirtual** è la virtualizzazione dell'invocazione del metodo ma che ha un nome simbolico, quindi bisogna trovare un modo affinché si acceda alla classe in cui è presente il metodo, alla tabella dei metodi della classe e poi eseguire il codice.

La prima volta che dobbiamo invocare un metodo, risolviamo la traduzione simbolico->codice metodo ma le altre volte operiamo con indirizzo di base+offeset.

Quindi ogni classe che viene caricata risiede in una struttura della JVM che si chiama *Class Area* ed è quindi dove opera il *Class Loader*.



Procedura di accesso ai metodi

Esempio

- Codice di un metodo

```
void add2(Incrementable x) { x.inc(); x.inc(); }
```
- Ricerca del metodo
 - trovare la classe dove il metodo è definito
 - trovare la vtable della classe
 - trovare il metodo nella vtable
- Chiamata del metodo
 - creazione del record di attivazione, ...

Una volta che dal riferimento simbolico risalgo all'indirizzo+offset di un metodo tramite la **invokevirtual**, la JVM si tiene l'informazione e dopo se viene invocato di nuovo il metodo farà prima.

Method invocation:

invokevirtual: usual instruction for calling a method on an object

invokeinterface: same as invokevirtual, but used when the called method is declared in an interface (requires different kind of method lookup)

invokespecial: for calling things such as constructors. These are not dynamically dispatched (this instruction is also known as invokenonvirtual)

invokestatic: for calling methods that have the "static" modifier (these methods "belong" to a class, rather an object)

Returning from methods:

return, ireturn, lreturn, areturn, freturn, ...

LEZIONE 30

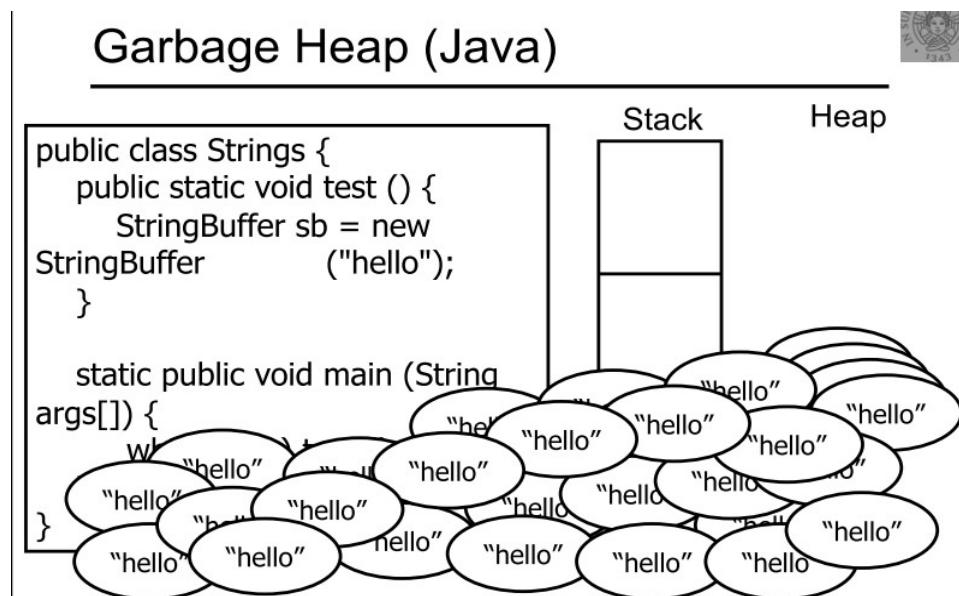
Gestione della memoria

Nelle tecniche di implementazione di un linguaggio abbiamo visto che grosso modo quando la macchina virtuale di esecuzione del linguaggio chiede al gestore della memoria dello spazio per allocare un programma, ha bisogno di un'area statica per tutti quei contenuti di dimensione fissa che sono determinati staticamente dal compilatore (e.g. Sorgenti del programma, codice, strutture statiche della macchina intermedia di esecuzione del linguaggio) poi ha bisogno di 2 aree di memoria dinamica: lo stack per i record di attivazione per la gestione dei sottoprogrammi e lo heap per la gestione delle strutture dinamica (e.g. Istanziazione degli oggetti).

Soltanmente sono allocati staticamente: variabili globali, variabili locali sottoprogrammi (senza ricorsione), costanti determinabili a tempo di compilazione, tabelle usate dal supporto a run-time (per type checking, garbage collection, ecc.).

Per quanto riguarda lo Stack, la dimensione di un record di attivazione non è fissata perchè dipende dalla dimensione del sottoprogramma ma sicuramente ha una struttura fissa.

Nello heap può essere gestito con blocchi di dimensione fissa o variabile, che dipende dalle caratteristiche del linguaggio. Lo heap è necessario quando il linguaggio permette: allocazione esplicita di memoria a run-time, oggetti di dimensioni variabili, oggetti con vita non LIFO. La sua gestione non è banale perchè va gestito efficientemente lo spazio (problema di frammentazione) e deve avere un accesso veloce.

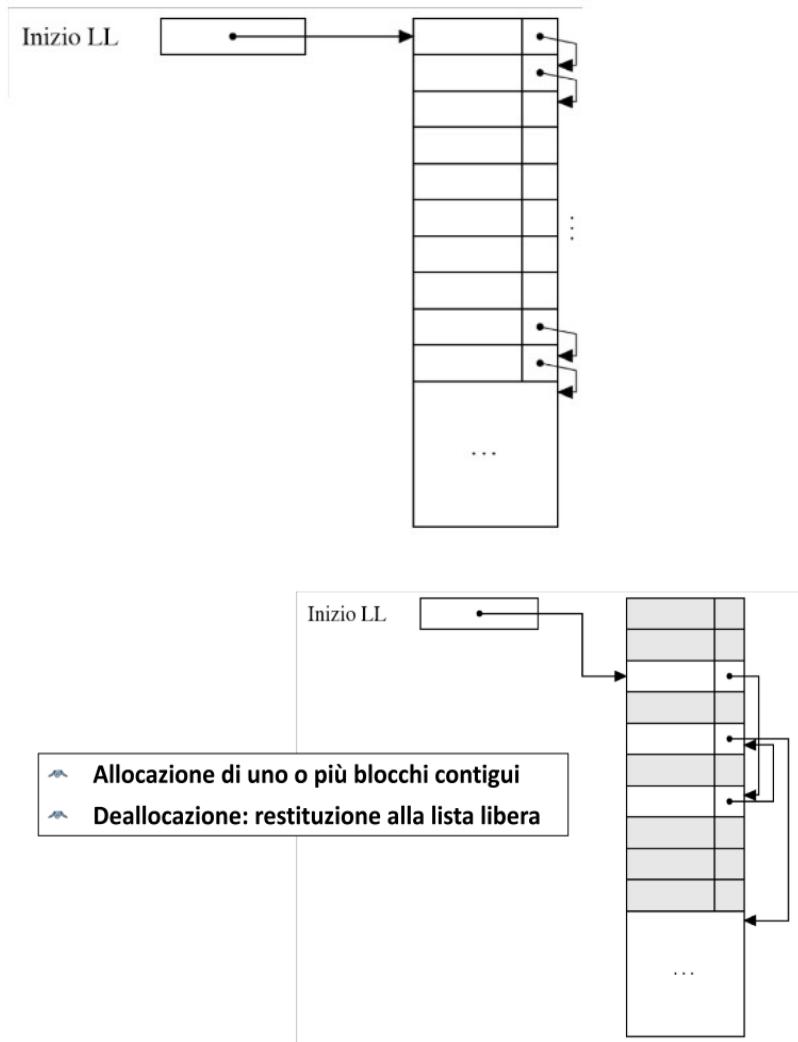


Nello stack al massimo abbiamo *main* e *test* ma ogni volta che faccio *new* creo nuove istanze nello heap

Heap dimensione fissa

E' un blocco di memoria divisivo in altri piccoli blocchi di memoria di dimensione fissa che sono collegati tra di loro tramite dei puntatori. Si parla di lista libera (LL) all'inizio ogni blocco è vuoto e si chiama così il puntatore alla testa della lista.

Per richiedere memoria si chiede alla lista libera di darmi un blocco e questo viene marcato occupato e il puntatore LL punterà a quello successivo.



Quando alloco chiedo lo spazio necessario per allocare la struttura dinamica che voglio memorizzare e in più anche il descrittore.

Heap dimensione variabile

Inizialmente lo heap viene visto come un unico blocco.

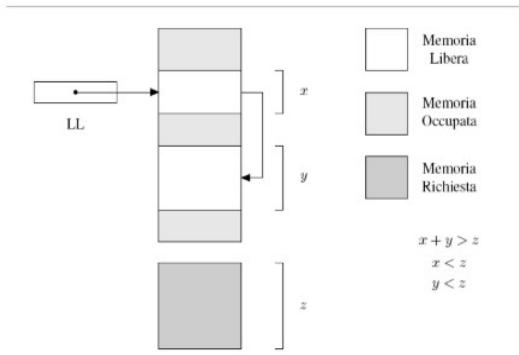
Quando alloco cerco un blocco libero della dimensione opportuna.

Quando dealloco restituisco alla lista libera.

Dopo un po' che alloco e dealloco ho dei problemi:

- **Frammentazione interna**

- lo spazio richiesto è X
- viene allocato un blocco di dimensione $Y > X$
- lo spazio $Y-X$ è sprecato



- **Frammentazione esterna**

- ci sarebbe lo spazio necessario ma è inusabile perché suddiviso in "pezzi" troppo piccoli

Quandoalloco ho due strategie possibili:

- **Best fit:** alloco il blocco di memoria migliore ovvero che spreca meno memoria. (lento)
- **First fit:** scorro la lista, trovo il primo blocco di memoria che mi permette di allocare quanto necessario e alloco. (veloce)

Quando dealloco un blocco devo effettuare un meccanismo di fusione in modo da avere blocchi adiacenti più grandi possibili.

Lasciare al programmatore la possibilità di gestire la memoria dinamica può causare errori (dangling reference: la memoria viene liberata prima che tutti i cammini di accesso a quella struttura di memoria sia stati eliminati) mentre se non diamo questo possibilità richiedere un opportuno modello della memoria per definire "raggiungibilità" (se un blocco di memoria è ancora raggiungibile allora non lo dealloco).

Modello a grafo della memoria (foresta di alberi)

Si individua sulle strutture di implementazione della JVM una specie di origine del grafo, **root set**.

Il root-set è l'insieme delle radici degli alberi della foresta, dove ogni radice è una struttura attiva che punta allo heap.

Lo stack contiene i record di attivazione che sono attivi i quali contengono le variabili locali attive che punteranno ad oggetti nello heap, queste variabili sono le radici della foresta.

Nomenclatura:

- **cella:** blocco di memoria nello Heap.
- cella **live:** se il suo indirizzo è memorizzato in una radice o in altra cella live (quindi una cella è live se e solo se appartiene ai *reachable active data*, ovvero raggiungibile da un elemento del root-set).
- Cella **garbage:** se non è live.

Garbage Collector

E' quel meccanismo che ha un modello della memoria in termini di un grafo, che individua le radici della foresta del grafo, fa un'operazione di raggiungibilità a partire dalle radici e restituisce, alla lista libera (allo Heap non utilizzato), tutte le celle che NON sono live (ovvero garbage).

Il garbage collector perfetto

- Nessun impatto visibile sull'esecuzione dei programmi
- Opera su ogni tipo di programma e su ogni tipo di struttura dati dinamica (esempio: strutture cicliche)
- Individua il garbage (e solamente il garbage) in modo efficiente e veloce
- Nessun overhead sulla gestione della memoria complessiva (caching e paginazione)
- Gestione heap efficiente (nessun problema di frammentazione)

Tecniche di GC

- **Reference counting – Contatori di riferimento**
 - gestione diretta delle celle live
 - la gestione è associata alla fase di allocazione della memoria dinamica
 - non ha bisogno di determinare la memoria garbage
- **Tracing:** identifica le celle che sono diventate garbage
 - **mark-sweep**
 - **copy collection**
 - **generational GC**

Reference counting

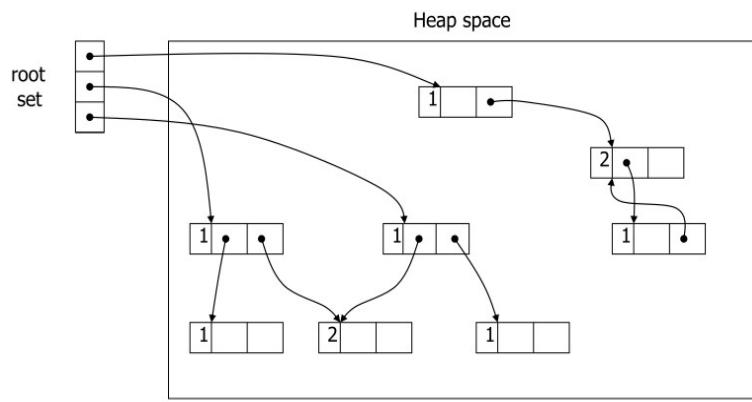
Ogni volta che alloco e dealloco celle nello Heap, il sistema si preoccupa di gestire se la cella è live o NON live.

Ad ogni cella dello Heap viene messo un contatore che dice quanti sono i cammini di accesso attivi a quella cella. La cella diventa garbage quando il valore del contatore di riferimento è arrivato a 0.

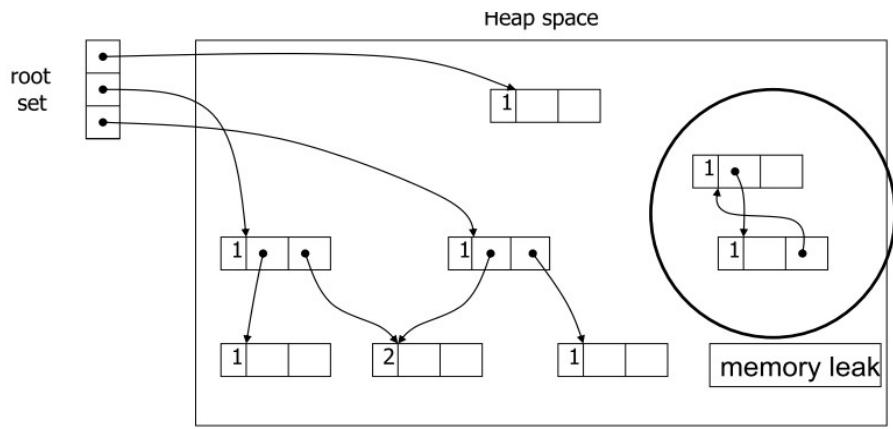
Nella macchina virtuale di esecuzione del linguaggio tutte le operazioni di memoria dinamica richiedono:

- un overhead di memoria: perchè richiedono spazio per il contatore
- un overhead di esecuzione: perchè tutte le operazioni che fanno aumentare o diminuire i contatori di accesso ad una cella devono operare sul valore del contatore di riferimento.

```
Integer i = new Integer(10);
    ○ RC (i) = 1.
j = k; (j, k riferiscono a oggetti)
    ○ RC(j) --.
    ○ RC(k) ++.
```



Problemi:



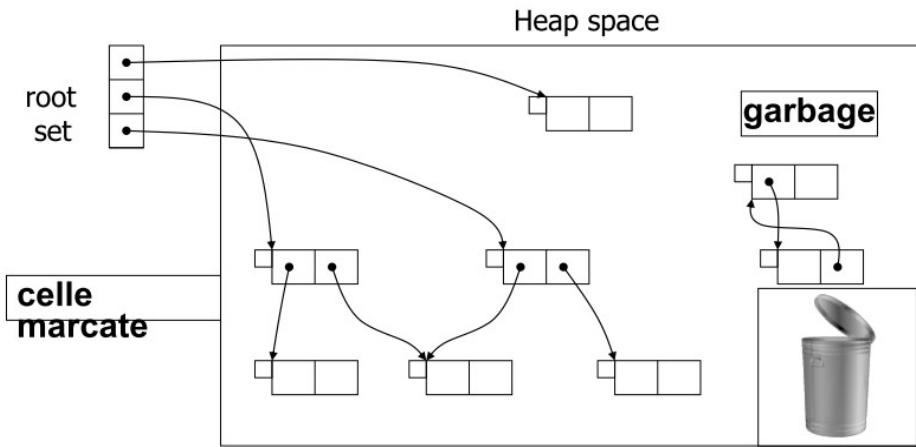
Mark-sweep

Ogni cella prevede un **bit di marcatura**.

E' un bit che viene attivato a 1 quando viene eseguita l'operazione di **tracing**: parto dal root-set e raggiungo una cella (**mark**).

Quando ho eseguito la tracciatura le celle garbage saranno tutte quelle che hanno il bit di marcatura = 0.

Ovviamente se restituisco qualcosa alla lista libera (**sweep**), devo rifare una tracciatura perchè le cose potrebbero esser cambiate.



- Opera correttamente sulle strutture circolari (+)
- Nessun overhead di spazio (+)
- Sospensione dell'esecuzione (-)
- Non interviene sulla frammentazione dello heap (-)

Copy-collection

E' basato sull'**algoritmo di Cheney**.

Si divide lo Heap in due parti: **front-space** e **to-space**.

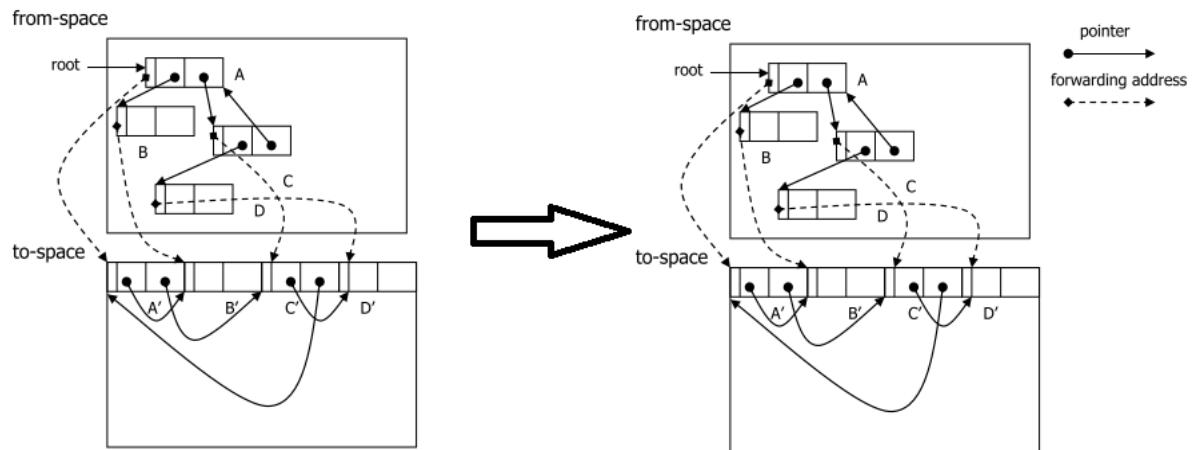
Ad ogni istante dell'esecuzione del programma solo una delle parti è attiva.

Viene eseguito il **mark-sweep**, le celle che sono live vengono copiate in porzioni contigue dello Heap nella parte che non era attiva (Evito la frammentazione esterna).

Alla fine dell'operazione di copia, i ruoli tra le due parti dello Heap vengono scambiati (la parte non attiva diventa attiva e viceversa).

Le celle nella parte non attiva vengono restituite alla lista libera in un unico blocco evitando problemi di frammentazione.

Esempio



- E' efficace nella allocazione di porzioni di spazio di dimensioni differenti e evita problemi di frammentazione (+)
- Duplicazione dello heap (-)

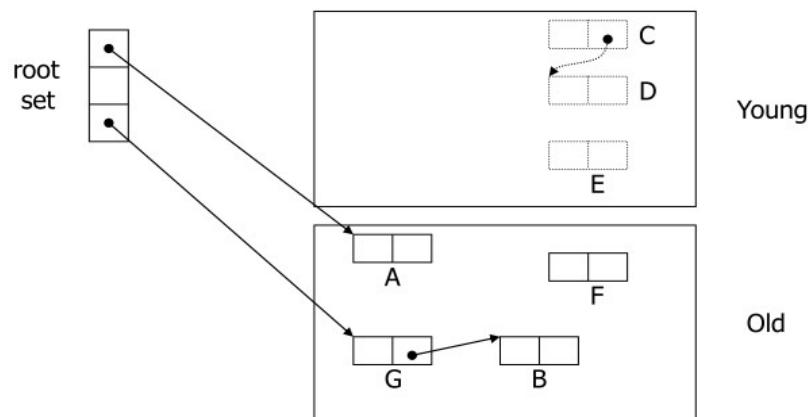
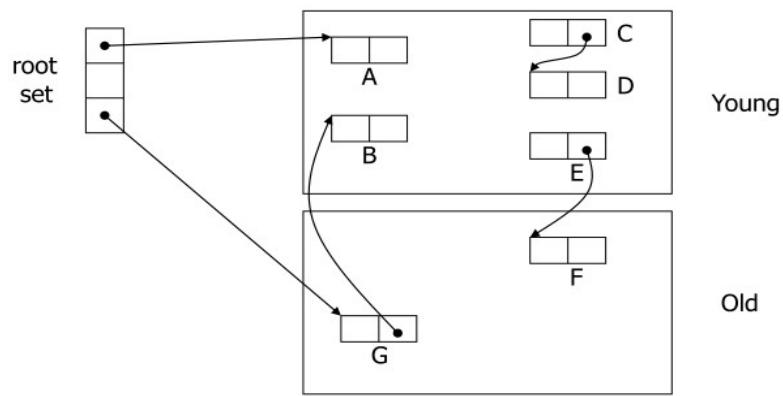
Generation GC

Solitamente le celle più “fresche” diventano subito garbage.

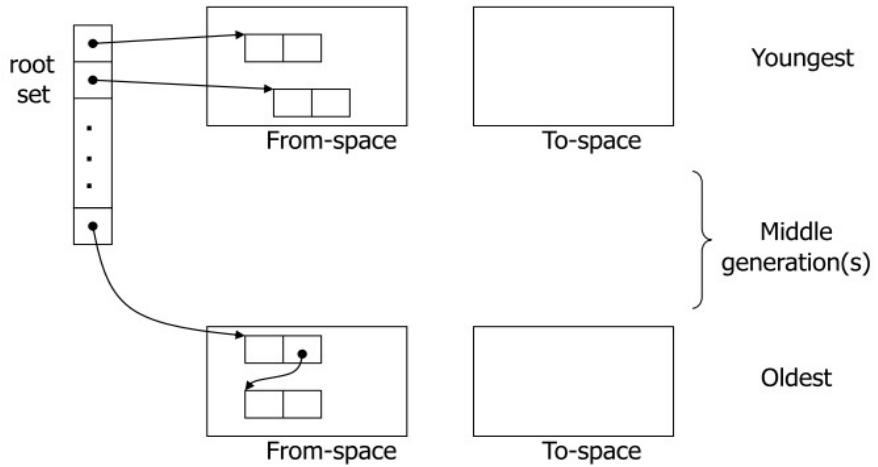
Quindi l'idea è di dividere la memoria in **generazioni**.

Il GC opera sulle generazioni più giovani, quando una cella è da un po' di tempo attiva viene promossa ad una generazione.

Esempio 2 generazioni:



Per evitare la frammentazione, **copying + 2 generazioni**:



Così ho tutta l'efficienza del GC che individua velocemente quelli che muoiono prima ed evita la frammentazione, generazione per generazione.

Nella pratica JVM:

- GC con 3 generazioni (0, 1, 2)
- Gen. 1 fa copy-collection
- Gen. 2 fa mark-sweep esteso con meccanismi per evitare la frammentazione.