Lezione 3-6

22-27 febbraio 2012

- Naming, Binding, Valori, V. Denotabili
- Env, Store, AR e Blocchi: Motivazioni
- Blocchi: Scoping Statico e Dinamico
- AR: Implementazione
- Unità di Programmazione di un L.P.
- Alising, chiusure e Lambda Lifting
- Env: Formalizzazione, Implementazione
- Store: Formalizzazione, Implementazione

Naming

- Naming = uso di Nomi per Definizioni
    - In Linguaggio di Programmazione:
        - Nomi = Identificatori
        - Definizioni = Valori Denotabili (Den)
            - Val = Dominio Valori del linguaggio
            - Den $\subseteq$ Val

- Esempi
    - Primitive del linguaggio: codice per la loro implementazione nel linguaggio oggetto
    - Var x: int -- un (valore) variabile di tipo int e di nome x
    - Const int y -- un (valore) costante di nome y
    - *int z[] -- un v. variabile di tipo array di pointer int e di nome z
    - label a -- un v. etichetta di programma di nome a
    - void p(…){…} -- un v. procedura di C di nome p (e tipo?…C/Java: una lunga storia)
    - class A {…} -- un v. classe di Java di nome A ed anche un v. tipo "descrittore di A"
    - struct S{…} -- un v. struct di C di nome S
    - type B = … -- un v. tipo di Caml (et al…) di nome B (e di tipo?…: una diversa l. storia)
    - …

Den

- Den è una caratteristica fondamentale del linguaggio
  - Semplice: ha solo variabili
  - C: …
  - Java: …
  - Caml: …
  - Prolog: …

- Esempi
  - I valori variabile (o la variabile, comunemente detta) è:
    - un Valore Denotabile in Semplice perchè Semplice prevede che possa essere riferita con un identificatore.
    - un valore modificabile (in tutti i linguaggi), da qui il nome
    - operazioni sono previste nei linguaggi per la sua modifica: assegn., assegn. con indice, …
    - queste operazioni talora sono comandi, altre operatori di espressioni
    - un valore modificabile è implementato con Locazioni di Memoria (**Store**)

Binding

- Binding = Legame di Identificatore al Suo Valore Denotabile
  - Creato Staticamente
    - front-end compilatore/interprete: Symbol Table
  - Gestiti attraverso gli AMBIENTI (Env)

- Esempi
  - Nel caso di una variabile il Valore Denotabile o Denotazione è una Locazione di Memoria
  - Var Ide :Int, ha come significato un binding tra Ide e una Locazione (Dominio Loc), ovvero:
    - Ide x Loc

Env, Store, AR e Blocchi: Motivazioni

- Env = usato per permettere il naming nei programmi
  - un L.P. di basso livello (macchina):
    - non usa il naming nei programmi
    - non ha bisogno di Env
- Store = usato per valori modificabili ma anche per valori che non saranno modificati (Mem=Dominio valori memorizzabili)
  - un L.P. ha sempre uno Store
  - Linguaggi Funzionali Puri (Haskell) non hanno valori modificabili ma hanno comunque Env e Store (dinamica)
- AR = Activation Record
  - usato per supportare più sezioni del programma ognuna in grado di definire i propri naming
    - blocchi, procedure/funzioni, (moduli,) astrazioni,…

Blocchi: S. Statico - S. Dinamico/1

• Blocchi = usati per creare sezioni di programma *parzialmente autonome e parzialmente dotate di funzionalità specifica* per l'algoritmo implementato dal programma

- hanno un proprio spazio di nomi
- hanno un codice che fa riferimento a tali valori denotabili
- sono di due tipi:
  - in-line:
    - anonimi;
    - sintatticamente delimitati ({..},let..in)
    - eseguiti in accordo alle regole di composizione del linguaggio per blocchi in-line:sequenza o annidamento

Blocchi: S. Statico - S. Dinamico/2

- Blocchi = usati per creare sezioni di programma *parzialmente*
  - sono di due tipi:
    - in-line:
    - procedure/funzioni:
      - Hanno nome, parametri e (spesso) tipo
      - I parametri sono namings del blocco sebbene il binding è differito e creato ad ogni invocazione
      - eseguiti utilizzando meccanismi di trasferimento di controllo di tipo *call-return*
- Scope di un binding definito da un naming **I** in un blocco **A** =
  = sezione **Z** di codice che accede tale binding riferendo I
  - Dipende dal Linguaggio
  - Due classi di Linguaggi:
    - Linguaggi con Scope (degli identificatori) Statico
    - Linguaggi con Scope (degli identificatori) Dinamico

Blocchi: S. Statico - S. Dinamico/3

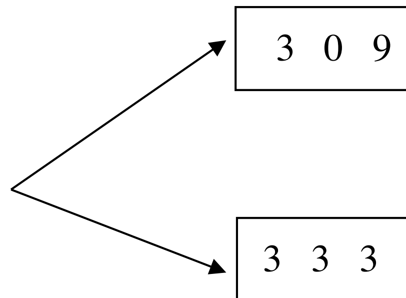
- Scope di un binding definito da un naming **I** in un blocco **A** = …
  = sezione **Z** di codice che accede tale binding riferendo I
  - Due classi di Linguaggi:
    - Linguaggi con Scope (degli identificatori) Statico
      - Z contiene il codice di A che non sia un blocco
      - Z contiene il codice di ogni blocco B, **interno ad A**, e tale che B non abbia un proprio naming per I:
        - I è un identificatore e un binding non locale di B.
    - Linguaggi con Scope Dinamico
      - Z contiene il codice di A che non sia un blocco
      - Z contiene il codice di ogni blocco B, **eseguito, durante l'esecuzione del codice di A**, e tale che B non abbia un proprio naming per I:
        - I è un identificatore e un binding non locale di B.

Blocchi: S. Statico - S. Dinamico/4

- Scope di un binding definito da un naming **I** in un blocco **A** = …
  = sezione **Z** di codice che accede tale binding riferendo I
  - Due classi di Linguaggi:
    - Linguaggi con Scope (degli identificatori) Statico …
    - Linguaggi con Scope Dinamico …
    - Differiscono solo sulle non locali di procedure/funzioni

- Esempi

```
{int x = 0;
  void pippo(int n){ x=n+x;}
  pippo(3); print(x);
    {int x = 0;
      pippo(3); print(x);}
  print(x);}
```



Blocchi: S. Statico - S. Dinamico/5

- Scope di un binding definito da un naming **I** in un blocco **A** = …
  - Due classi di Linguaggi:
    - Linguaggi con Scope (degli identificatori) Statico
      - Motivazione: I non locali utilizzatti dalla procedura/ funzione sono quelli “visibili” e “localizzati” nella sezione di programma in cui è definita la procedura/ funzione
    - Linguaggi con Scope Dinamico:
      - Motivazione: Più semplice da implementare.

- Esempi

```
{int x = 0;
 void pippo(int n){ x=n+x;}
 pippo(3); print(x);
  {int x = 0;
   pippo(3); print(x);}
 print(x);}
```

Blocchi: S. Statico - S. Dinamico/6

- Scope di un binding definito da un naming **I** in un blocco **A** = …
  - Due classi di Linguaggi:
    - Linguaggi con Scope (degli identificatori) Statico
    - Linguaggi con Scope Dinamico:
  - Implementazione = come troviamo il binding di I?
    - Associamo ad ogni blocco un AR contenente l'Env del blocco
    - Uno Stack di AR contiene gli AR di blocchi non completamente attraversati:
      - blocchi in-line nested o
      - procedure/funzioni invocate ma non completate
    - La ricerca del binding:
      - di un locale: trovato nell'Env dell'AR corrente
      - di un non-locale: richiede ricerca nello stack

C: Delimitatori di blocco

In alcuni linguaggi (incluso C) i blocchi non sono sempre sintatticamente delimitati

- Esempi

```
{int x=5;
      …
       {int y = 0;
        x+1;
        ….
        int x=10;
        …
        y = x+1;
       }
    …
  }
```

- Quandi blocchi sono contenuti in questo blocco C ?
- Per ogni blocco si mostri il corrispondente ambiente
  - apparentemente abbiamo:
    - 2 blocchi
    - 3 ambienti
  - si trovi un criterio per delimitare i blocchi in C.

Ricordiamoci sempre che un blocco non può avere più di un binding per un identificatore

AR: Implementazione/1

- Activation Record ha una Struttura che dipende da
  - Blocco in-line contiene
    - Env (frame)
    - Program Counter (pc)
    - Unità di Memoria per Valori Risultati Intermedi (ri)
    - Puntatore Catena Dinamica (cd)

- Esempi

```
{int x = 0;
 void p(int n){ x=n+x;}
 p(3); print(x);
  {int x = 0;
    p(3); print(x);}
 print(x);}
```

x	lx_1	ri	pc
p	dp		cd

x	lx_2	ri	pc
			cd

AR: Implementazione/2

- Activation Record ha una Struttura che dipende da
  - Blocco in-line contiene
  - Blocco procedura contiene
    - Env (frame)
    - Program Counter (pc)
    - Unità di Memoria per Valori Risultati Intermedi (ri)
    - Puntatore Catena Dinamica (cd)
    - Puntatore Catena Statica (cs) - *se scoping statico*
    - Indirizzo di Ritorno (ret)
    - Indirizzo del Risultato (val)

- Esempi

```
{int x = 0;
 void p(int n){ x=n+x;}
 p(3); print(x);
  {int x = 0;
   p(3); print(x);}
 print(x);}
```

n ln	ri	pc	ret	val
		cd	cs	

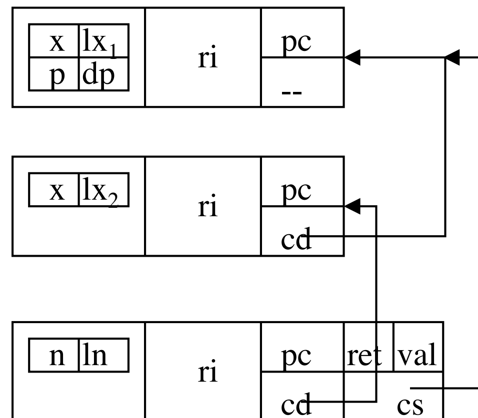
n ln	ri	pc	ret
		cd	val

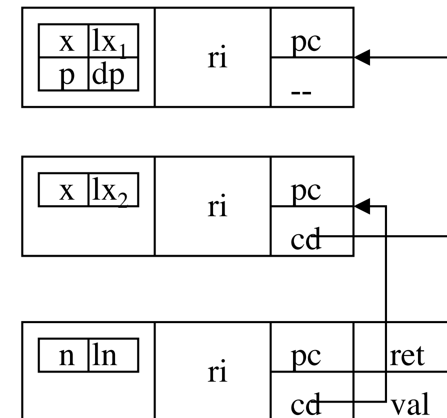
AR: Implementazione/3

- Scope di un binding definito da un naming **I** in un blocco **A** = …
  - Implementazione = come troviamo il binding di I?
    - Uno Stack di AR contiene gli AR di blocchi non completamente attraversati:
    - La ricerca del binding:

- Esempi

```
{int x = 0;
 void p(int n){ x=n+x;}
 p(3); print(x);
  {int x = 0;
    p(3); print(x);}
 print(x);}
```





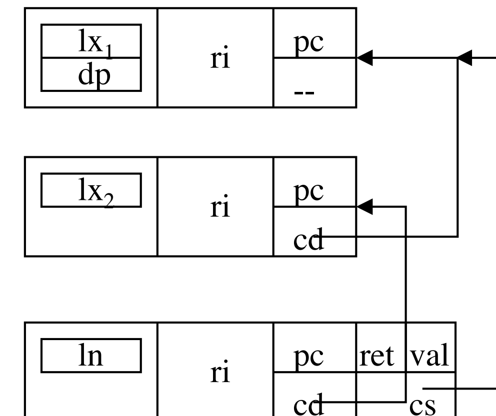
AR: Implementazione/4

- Scope di un binding definito da un naming **I** in un blocco **A** = …
  - Implementazione = come troviamo il binding di I?
    - scope statico:
      - più efficiente: accesso diretto (senza ricerca)
    - Algoritmo di Le Blank - Cook (1983):
      - ogni identificatore I usato in un blocco B è individuato da una coppia [l,p]:
        - l = *link catena statica* indica il livello di annidamento di B rispetto al blocco A in cui è il naming di I
        - p = posizione del naming di I nel blocco A

- Esempi

```
{int x = 0;
 void p(int n){ x=n+x;}
 p(3); print(x);
   {int x = 0;
     p(3); print(x);}
 print(x);}
```

```
{int x = 0;
 void p(int n){ [1,0]=[0,0]+[1,0];}
 [0,1](3); print([1,0]);
   {int x = 0;
     [1,1](3); print([0,0]);}
 print([0,0]);}
```



Unità di Programmazione di un L.P.

- Unità (di programazione di un L.P.) = Ogni struttura del linguaggio utilizzabile in un programma per descrivere completamente una (o più) funzionalità dell'algoritmo realizzato dal programma.
  - *procedure*, *funzioni*, *blocchi*, *moduli*, *astrazioni*, *classi*,... possono essere unità:
    - dipende da come possono essere usati gli identificatori in tali costrutti.

- Esempi

  A = algoritmo che:
  - legge una sequenza di schede studenti (utilizzando, ad esempio un algoritmo N per leggere i dati anagrafici ed un algoritmo M per quelli accademici);
  - ordina in accordo ad un algoritmo B (utilizzando un algoritmo N' per... M per...)
  - ...

Qui le procedure non sono unità

```
{...
 void leggiN(...){...}
 void leggiM(...){...}
 void leggi (void){
       leggiN (); leggiM ();}
 void ordinaB(...){...}// il codice di N' e M' non si
distinguono
...}
```

Qui le procedure sono unità

```
{...
  void leggi (void){
       void leggiN ();
       void leggiM ();
       {leggiN (); leggiM ();}
       }
  void ordinaB(...){
      void ordina N'(...);
     ...}
...}
```

Scope in C: A Case Study/1

- Esempi

questa sezione di programa
sembra codice C ma non lo è

```
{int x = 0;
 void p(int n){ x=n+x;}
 p(3); print(x);
  {int x = 0;
   p(3); print(x);}
 print(x);}
```

	C	Pascal	Java	Caml
non locali	solo globali	qualunque livello	qualunque livello	qualunque livello
scope	statico	statico	ereditarietà inner	statico
unità	modulo	procedura modulo	classi	funzione modulo

- Le procedure del C possono contenere solo identificatori globali
- Come è ottenuto questo vincolo in C?
- Cosa comporta questa limitazione?
    - uso (metodologie di programmazione: procedura non forma unità
    - implementazione: semplifica la gestione dello scoping statico

Scope in C: A Case Study/2

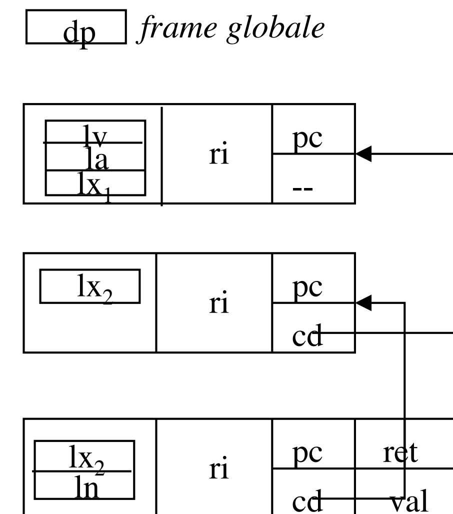
- Le procedure del C possono contenere come non locali solo identificatori globali (definiti nel modulo o in moduli importati)
- Come è ottenuto questo vincolo in C?
    - i blocchi (in-line e procedura) non possono introdurre naming per nuove procedure

- Cosa comporta questa limitazione?
    - uso (metodologie di programmazione: procedura non forma unità
    - implementazione: semplifica la gestione dello scoping statico
        - le procedure hanno AR con *cs* sempre uguale al *globale*
        - **Quindi:** la catena statica coincide con quella dinamica o ha cs uguale al *globale*
        - **Quindi:** la catena statica non si usa e utilizziamo il valore di link l=-1 come tag per idetificatori nel *globale*

- Esempi

```
#include <stdio.h>
#include <stdlib.h>
void p(int *x, int n);
void main(int v, char *a[]){
      int x = 0;
      p(&x,3);
      printf(”%s=%d\n", a[0],x);
      {int x = 0;
        p(&x,3);
        printf(”%s=%d\n", a[1],x);
       }
}
void pippo(int *x, int n){
      *x=n+*x;
}
```

```
#include <stdio.h>
#include <stdlib.h>
void p(int *x, int n);
void main(int v, char *a[]){
       int x = 0;
      [-1,0](&[0,2],3);
      printf(”%s=%d\n", [0,1][0],[0,2]);
      {int x = 0;
        [-1,0](&[0,0],3);
        printf(”%s=%d\n", [1,1][1],[0,0]);
       }
}
void pippo(int *x, int n){
      *[0,0]=[0,1]+*[0,0];
}
```



Ambiente: Aliasing

- Env = permette il naming
- Naming = permette di:
  - usare nomi come riferimenti a valori denotabili
  - condividere valori denotabili:
    - aliasing = nomi diversi per uno stesso v. den.
      - quali costrutti generano aliasing?
        - + trasmissione by reference
        - + alias: x = alias y

- Esempi

```
void XSwap(int A[], int B[]){
    A[0] ^= B[0];B[0] ^= A[0];A[0] ^= B[0];
};
int main(void){
    int A[] = {5};int B[] = {3};int C[] = {3};
    printf("A=%d,B=%d,C=%d\n",A[0],B[0],C[0]);
    XSwap(A,B); printf("A=%d,B=%d; ",A[0],B[0]);
    XSwap(A,C);
    printf("A=%d,C=%d\n",A[0],C[0]);
    XSwap(A,A);
    printf("A=%d\n",A[0]);return 0;
}
```

Pericolosi quando nascosti
in scelte implementative

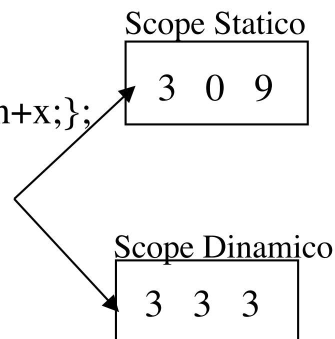
Il programma C a sinistra stampa:
A=5,B=3,C=3
A=3,B=5; A=3,C=3
A=0
- perchè?
- cosa calcola XSwap?

Lambda Lifting

- Scoping dinamico = Può essere evitato con LL
  - LL = Tecnica di eliminazione delle variabili libere [i.e. parametri non locali] nei L.P. (funzionali) con scope dinamico.
    + le variabili libere diventano parametri aggiuntivi della proc.
    + le variabili di LP procedurali sono:
      - trasmesse per reference
      - oppure, trasformate in variabili di valori puntatori

- Esempi

```
{int x = 0;
  void pippo(int n){ x=n+x;};
  pippo(3); print(x);
    {int x = 0;
     pippo(3); print(x);}
  print(x);}
```



Il compilatore C non riconosce questo prog.

```
#include <stdio.h>
#include <stdlib.h>
void pippo(int n);
void main(void){
    int x = 0;
     pippo(3);
     printf("valore di x=%d\n", x);
          {
          int x = 0;
           pippo(3);
          printf("valore di x=%d\n", x);
          }
    printf("valore di x=%d\n", x);
}
void pippo(int n){
     *x=n+*x;
}
```

Questo è un programma C: cosa stampa.

```
#include <stdio.h>
#include <stdlib.h>
void pippo(int *x, int n);
void main(void){
    int x = 0;
     pippo(&x,3);
     printf("valore di x=%d\n", x);
          {
           int x = 0;
           pippo(&x,3);
           printf("valore di x=%d\n", x);
          }
    printf("valore di x=%d\n", x);
}
void pippo(int *x, int n){
     *x=n+*x;
}
```

Chiusure

- Chiusure = codice i cui identificatori sono tutti legati
  - chiusure a livello sintattico: codice con sole locali
    - Lambda Lifting (LL) genera chiusure
    - Può essere condotto a compile time ad es. dal compilatore stesso
    - L’esecutore del codice generato non richiede accessi di catena statica
  - Chiusure a livello semantico:
    - coppia <codice, ambiente dei legami non locali>
    - originata da vari meccanismi:
      - trasmissione per nome
      - trasmissione di funzioni (procedure)
      - invocazioni che calcolano valori funzione (higher-order)

- Esempi

```
let rec fun_C f (x:tuple) =
     match x with
        C y -> f(y.uno,y.due)
       | T y -> let n = fun_C f (C y.c) in f(n,y.tre);;
```

Un codice Caml con un parametro funzione f

- Esempi

```
let isNin na =
   let getN(C(n,_)) = n
   in let f = fun x -> eqN(getN(x),na)
      in let h = fun g -> List.fold_right(||)(List.map f g)false
         in h;;
```

Un codice Caml che definisce una funzione
che calcola una funzione h

Env: Formalizzazione, Implementazione

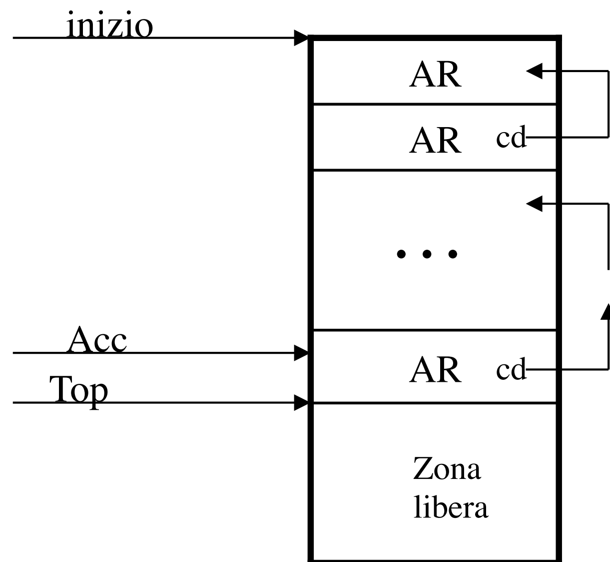
- Formalizzazione. Env = Struttura con le seguenti operazioni
  - bind: Ide x Den -> Env
    - astratta: bind(i,d,e) = λ j. if (i=j) then d else e(j)
    - più concreta: bind(i,d,e) = (i,d)::e   -- una lista di coppie
  - find: Ide x Env -> Den + Ide
    - astratta: find(i,e) = e(i)
    - più concreta: find(i,e) = | d   if (e=(j,d)::e' & i=j)
| find(i,e')   if (e=(j,d)::e' & i≠j)
| i   if (e=empty())
  - empty: () -> Env
    - astratta: empty() = λ j. j
    - più concreta: empty() = []   -- una lista vuota di coppie
- Implementazione. Un frame come visto negli AR

Store: Formalizzazione, Implementazione

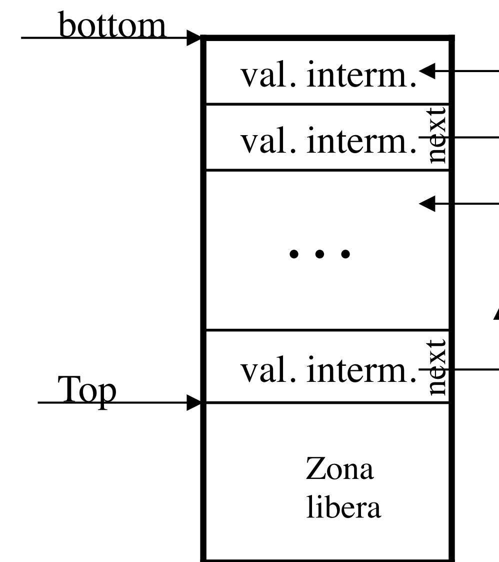
- Formalizzazione. STORE = Struttura con le seguenti operazioni
  - operazioni di allocazione: new -- più d'una
  - upd: Loc x Mem x Store -> Store
    - astratta: …
  - look: Loc x Store -> Mem
    - astratta: …
- Implementazione.
  - Tre dipi di memoria:
    - Statica -- tipica dei linguaggi macchina
    - Stack -- presente in alcuni linguaggi macchina (risultati intermedi) e in tutti i linguaggi con procedure/funzioni ricorsive
    - Dinamica -- presente in molti L.P. ad alto livello

Stack: Implementazione/1

•Implementazione.
- utilizziamo una sezione di memoria statica
- un puntatore *Inizio* alla prima parola della sezione utilizzata
- un puntatore *Acc* per l'accesso dell'AR top
- un puntatore all'ultima parola allocata per l'AR top



Stack per AR



Stack per valori risultati intermedi

Stack: Implementazione/2
(valori risultati intermedi)

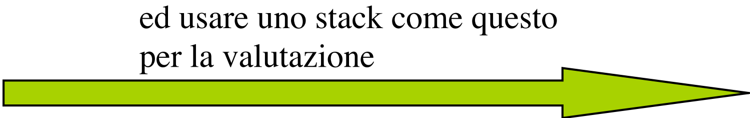
- Esempi

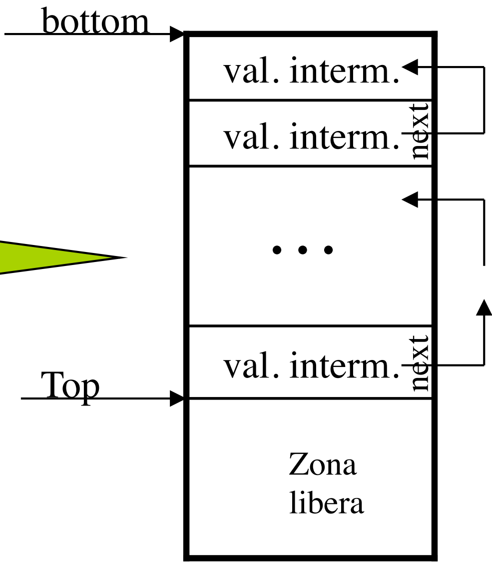
questo frammento

```
x or y ? x : z
```

può avere questa forma in JVM (Oolong/Jasmine)

```
  iload_0
  iload_1
  iload_0
  iload_2
  dup_x2
  pop
  dup_x2
  pop
  ifne false
  pop
  goto next1
false: swap
  pop
next1: ior
```





Stack per valori risultati intermedi

:::::::::::: SYMBOL TABLE :::::::::::::

index	lessema	valore	locazione	tipo
0	x	UNDEF	_0	UNDEF
1	y	UNDEF	_1	UNDEF
2	z	UNDEF	_2	UNDEF

:::::::::: SYMBOL TABLE: end :::::::::::

Heap: Implementazione/1

- Implementazione.
  - Utilizziamo sempre una sezione di memoria statica
  - Due tipi di Heap:
    - Omogeneo: alloca/de- blocchi di dimensione fissa
    - Variabile: alloca/de- blocchi di dimensione variabile

- Esempi

```
{…
 char *x;…
 x = malloc(sizeof(char));
 ….
 char *y[];
 y = malloc((length1(p)+1)*sizeof(char));
…}
```

Le strutture allocate potrebbero essere trattate com omogenee

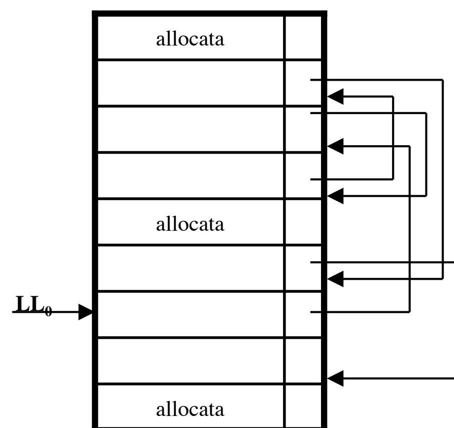
- Esempi

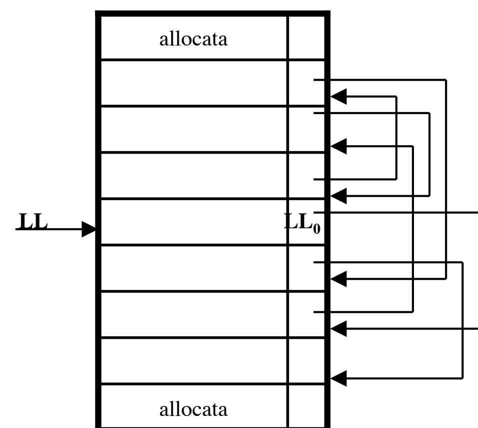
```
{…
 typedef struct quadrup {
        char *opt;
        struct opd *opd1;
        struct opd *opd2;
        struct opd *opd3;
        }quadrup;
 y = malloc((length1(p)+1)*sizeof(char));
…
 q = malloc(sizeof(quadrup));
 …}
```

Qui è impossibile allocare in modo omogeneo

Heap: Implementazione/2

- Implementazione.
  - Utilizziamo sempre una sezione di memoria statica
  - Due tipi di Heap:
    - Omogeneo: alloca blocchi di dimensione fissa K
      - organizziamo la memoria in una lista di blocchi di dimensione K (ognuno contiene una parola in più per il link al blocco libero successivo)
      - un puntatore LL contiene l'indirizzo del primo blocco allocabile.



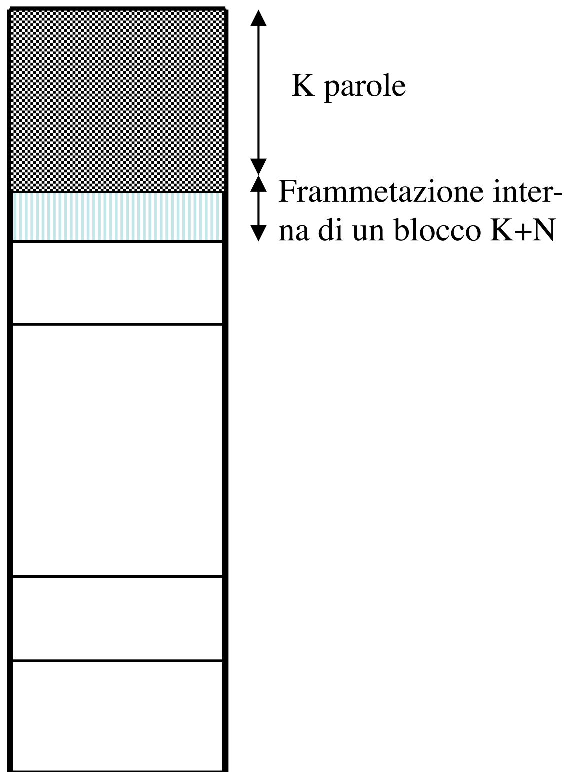


Heap: Implementazione/3

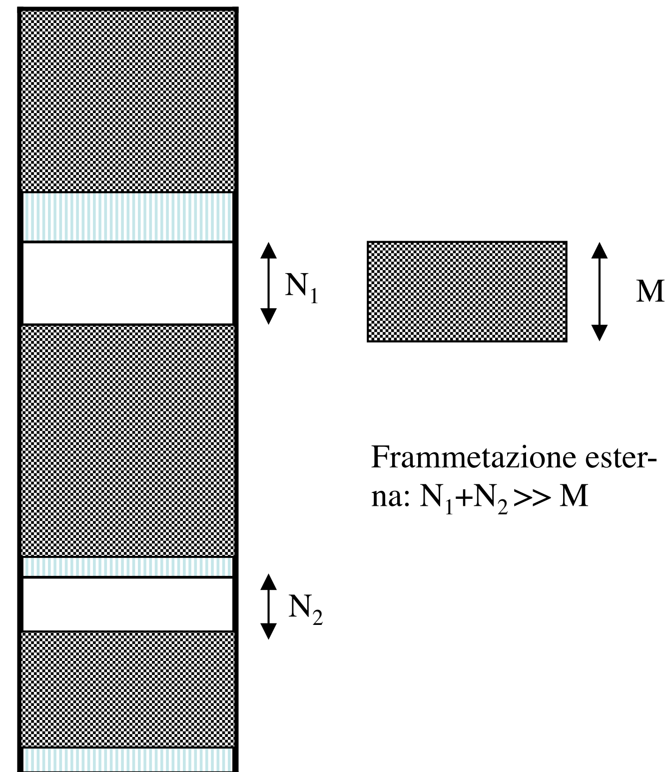
- Implementazione.
  - Utilizziamo sempre una sezione di memoria statica
  - Due tipi di Heap:
    - variabile: alloca blocchi di dimensione variabile
      - frammentazione interna: inevitabile ma accettabile
      - frammentazione esterna: pericolosa, costosa, da contenere con due implementazioni:
        - **lista unica** - best fit - compattamento
        - **lista multipla** - buddy/Fibonacci allocation

Heap: Implementazione/4

- frammentazione interna: inevitabile ma accettabile
- frammentazione esterna: pericolosa, costosa, da contenere



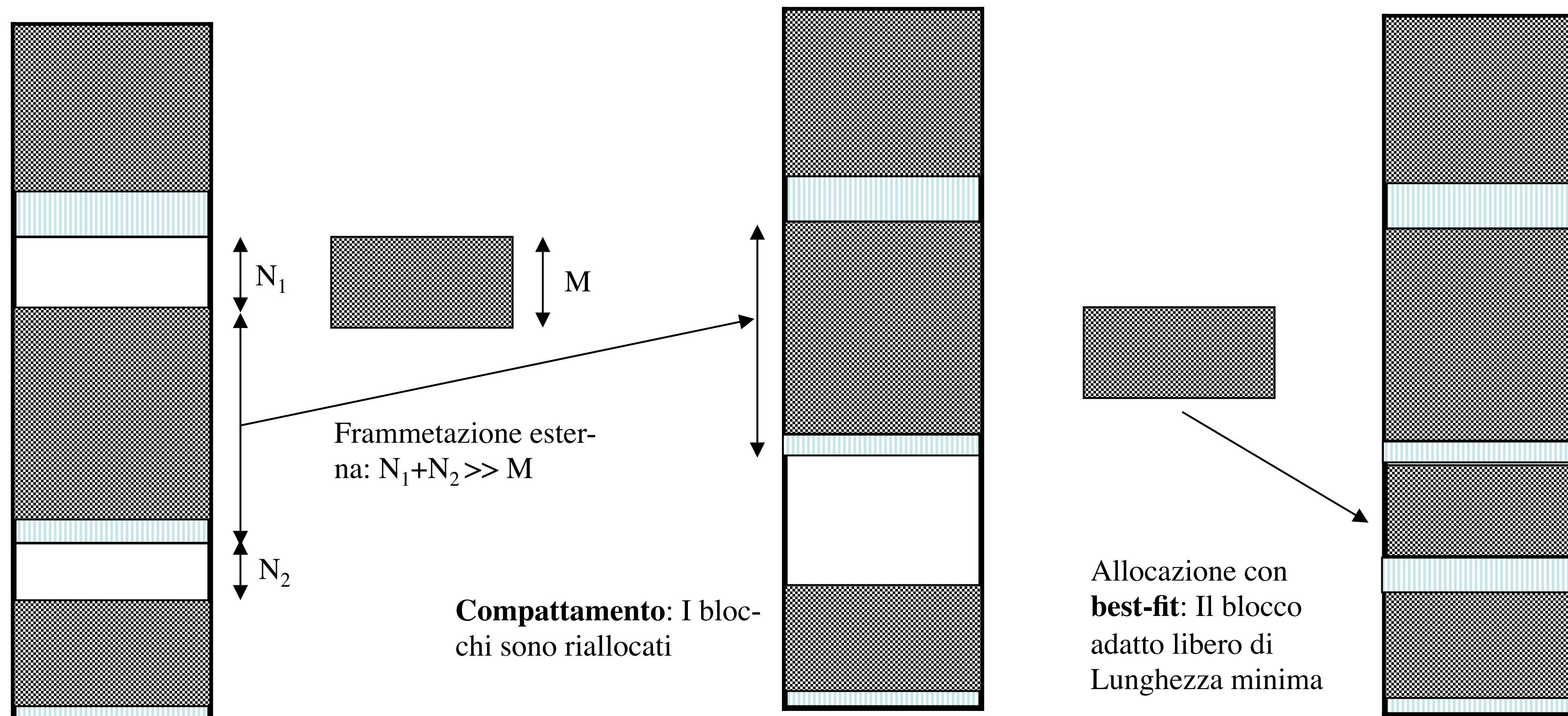
Dobbiamo allocare un blocco di K parole, ma i blocchi dispobili hanno più di k o meno di k parole



Dobbiamo allocare un blocco di M parole, ma i blocchi dispobili hanno ciascuno meno di M parole ma tutti insieme più di M

Heap: Implementazione/5

- Frammentazione esterna: pericolosa, costosa, da contenere con due implementazioni:
  - **lista unica** - best fit - compattamento
    - limiti:compattamento
      - time expensive
      - blocchi non rilocabili



Heap: Implementazione/5

- **lista unica** - best fit - compattamento
- limiti:compattamento
  - time expensive
  - blocchi non rilocabili

- Esempi

```
{…
  char *x;…
  x = &y
  ….
  while(*w){… (*x++ = *w++) …}
 …}
```

Cosa può accadere rilocando quando è in esecuzione un frammento come questo?

Cosa significa l'espressione occorrente nel corpo del while?

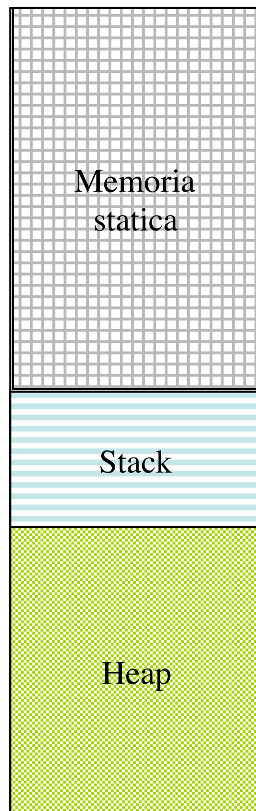
Heap: Implementazione/4

- Frammentazione esterna: pericolosa, costosa, da contenere con due implementazioni:
  - lista multipla - Buddy/Fibonacci allocation
    + $[k_1],[k_2],\ldots,[k_n]$ = n liste omogenee di dim. $k_1<k_2<\ldots<k_n$
    + allochiamo k nella lista $[k_i]$: $k_{i-1}<k\leq k_i$
      + direttamente se $[k_i]$ ha un blocco libero
      + con prestito da $[k_{i+j}]$ altrimenti
        * $[k_{i+j}]$ deve avere blocchi sommatoria di multipli di $k_i,\ldots, k_{i+j-1}$
        * il prestito è restituito quando $[k_i],\ldots, [k_{i+j-1}]$ hanno almeno tanti blocchi liberi quanti quelli ricevuti in prestito
  - Buddy: $k_1<k_2<\ldots<k_n$ sono potenze (contigue) di 2
  - Fibonacci: $k_1<k_2<\ldots<k_n$ sono numeri di fibonacci
- limiti: critica la scelta della dimensione del blocco
  - piccolo => tante liste
  - grande => tanta frammentazione interna

Heap: Implementazione/5

- Buddy: m=2 ($k_1<k_2<\ldots<k_n$ sono potenze di 2)
- Fibonacci: $k_1<k_2<\ldots<k_n$ sono numeri di fibonacci

Abbiamo un bloco di parole: Decidiamo di diverlo in tre parti



Organizzaimo la memoria dinamica in tre pari utilizzando

Buddy System = 1,2,4

Fib. System = 1,2,3

Come sono fatti i blocchi della Lista1, Lista2 e Lista3?

Quanti LL avremo: Si disegni ogni lista allo stato iniziale.

Si mostri il comportamento dei due sistemi allorchè:
Dobbiamo allocare un blocco di lunghezza 1 ma solo la Lista3 ha me-
Moria libera.

Ambiente e Memoria in:Fortran, C, Java, Caml, Java, Prolog

	Fortran	**C**	**Pascal**	**Java**	**Caml**	**Prolog**
Ambiente	SI	SI	SI	SI	SI	SI
Statica	SI	SI	SI	SI	SI	SI
Stack	NO	SI	SI	SI	SI	SI
Heap	NO	SI	SI	SI	SI	SI

Cose da Fare

- STUDIARE e approfondire il contenuto dei lucidi utilizzando cap. 6-7 del testo, indicazioni bibliografiche contenuti nei capitoli, e i testi di programmazione utilizzati negli anni passati
- PROGRAMMARE e controllare l'istallazione degli strumenti per l'uso dei linguaggi: C, Java, Caml, Prolog [usare i primi 3 con un programma per il calcolo del fattoriale]
- ESERCIZI:
  - verificare, completare, estendere tutti gli esempi e/o esercizi inclusi nei lucidi utilizzando tutti gli strumenti di cui disponete (programmi, linguaggi,)
  - svolgere tutti gli esercizi riportati nel capitoli 6-7 del testo.