Dipartimento
di Matematica
e Informatica

Implementazione dei linguaggi

Leggere sezioni 6.1, 6.2, 6.3, 7.1, 7.2, 7.3, 7.4, 8.3, 8.4, 8.5, 10.2, 10.11, 12.2, 12.3 di “Linguaggi di programmazione - Principi e paradigmi”

Fabrizio Riguzzi

Università di Ferrara

Nomi ed Ambienti

- Un **nome** è una sequenza di caratteri (possibilmente con un significato) utilizzata per rappresentare un'entità
- I nomi permettono di rendere astratti sia i dettagli dei dati, per esempio utilizzando un nome per indicare una cella di memoria, sia dettagli relativi al controllo, per esempio indicando una serie di comandi con un nome
- La corretta gestione dei nomi richiede regole semantiche precise e adeguati meccanismi di implementazione

Dipartimento di Matematica e Informatica

Oggetti Denotabili

Gli oggetti ai quali viene attribuito un nome sono detti **oggetti denotabili**. Una lista non esaustiva di questi è

- Oggetti i cui nomi sono definiti dall'utente: variabili, parametri formali, procedure (in senso ampio), tipi di dato user defined, etichette, moduli, costanti definite, eccezioni, ...
- Oggetti i cui nomi sono definiti dal linguaggio di programmazione: tipi primitivi, operazioni primitive, costanti predefinite

L'associazione (o legame, binding) tra un nome e l'oggetto al quale si riferisce possono essere creati in tempi diversi: alcuni nomi sono associati ad oggetti durante la creazione del linguaggio, altri sono associati quando il programma viene eseguito

Dipartimento di Matematica e Informatica

Ambiente

- Non necessariamente tutte le associazioni tra nomi ed oggetti denotabili sono immutabili
- Molte possono variare durante l'esecuzione

Definizione di Ambiente: L'insieme di associazioni tra nomi ed oggetti denotabili che esiste a tempo di esecuzione in uno specifico punto del programma ed ad uno specifico momento dell'esecuzione è chiamato **ambiente** (**referencing environment**, **namespace** in Python)

Dipartimento di Matematica e Informatica

Visibilità

Le regole di visibilità specificano come e quando una dichiarazione può essere visibile
Definizione: Una dichiarazione locale ad un blocco è visibile in quel blocco e in tutti i blocchi in esso annidati, a meno che non ci sia in tali blocchi una nuova dichiarazione dello stesso nome. In tal caso, nel blocco in cui compare la ridefinizione, la nuova dichiarazione nasconde (o maschera) la precedente.

Dipartimento di Matematica e Informatica

Regole di Scope I

- All'ingresso e all'uscita di un blocco, l'ambiente può cambiare a seguito delle operazioni che attivano o disattivano le associazioni
- Questi cambiamenti sono abbastanza intuitivi per ambienti locali, ma lo sono meno per ambienti non-locali. La regola di visibilità elencata sopra permette due interpretazioni

Dipartimento di Matematica e Informatica

Regole di Scope II

Considera il seguente frammento di programma:

```
A:{int x = 0;
  void fie(){
    x = 1; }
  B:{int x;
    fie();}
  write(x); }
```

Che valore viene stampato? Possiamo pensare che venga stampato 1, visto che la procedura `fie` è definita all'interno del blocco `A` e quindi la variabile `x` nel corpo della procedura potrebbe essere quella definita nella prima linea di `A`. Tuttavia, `x` viene ridefinita nel blocco `B`, quindi la chiamata a `fie` potrebbe modificare questa variable, invece che quella definita in `A`, quindi `write(x)` potrebbe stampare 0. Notare che le funzioni innestate non esistono nel linguaggio C.

Dipartimento di Matematica e Informatica

Regole di Scope III

Il risultato del precedente frammento di programma dipende dalla regola di scope che viene utilizzata. La regola precedentemente introdotta non specifica se il concetto di annidamento deve essere considerato statico (basato sul testo del programma) o dinamico, basato sul flusso di esecuzione

- Scope **statico o lessicale** (nel caso precedente viene stampato 1) (Ada, Pascal, C, Java, Python)
- Scope **dinamico** (nel caso precedente viene stampato 0)b (LISP, Snobol, Perl)

Dipartimento di Matematica e Informatica

Gestione della Memoria

- La gestione della memoria è un componente fondamentale dell'interprete della macchina astratta
- Semplice in una macchina fisica
- Complicato in una macchina astratta per un linguaggio di alto livello
- Questa funzionalità gestisce l'allocazione della memoria per programmi e per i dati. Per esempio, determina come questi devono essere organizzati in memoria, quanto tempo rimangono e quali strutture ausiliarie sono necessarie per ottenere informazioni dalla memoria
- Nel caso di una macchina astratta di basso livello, per esempio a livello hardware, la gestione della memoria è semplice e può essere completamente statica
- Prima dell'inizio del programma, il linguaggio macchina e i dati associati ad esso sono caricati in un'opportuna area di memoria, e lì rimangono fino al termine dell'esecuzione
- Nel caso di un linguaggio ad alto livello, se il linguaggio permette la ricorsione, l'allocazione statica non è più sufficiente
- Infatti, nelle procedure ricorsive, il numero di chiamate a procedure dipende dai parametri della procedura stessa

Esempio

```
int fib (int n) {
  if (n == 0) return 1;
  else if (n == 1) return 1;
    else return fib(n-1) + fib(n-2);
  }
```

- Se chiamata con un argomento n, calcola (in maniera molto inefficiente) il valore dell'n-esimo numero di Fibonacci
- I numeri di Fibonacci sono i termini della sequenza definita in maniera induttiva come segue: $Fib(0) = Fib(1) = 1$, $Fib(n) = Fib(n-1) + Fib(n-2)$ per $n > 1$
- Il numero di chiamate attive a `fib` dipende, oltre che dal punto di esecuzione, dal valore dell'argomento `n`
- Utilizzando una semplice relazione di ricorrenza si può verificare che il numero di chiamate $C(n)$ necessarie per calcolare il valore di $Fib(n)$ è uguale a questo valore
- Infatti, $C(n) = 1$ per $n = 0$ ed $n = 1$, $C(n) = C(n-1) + C(n-2)$ per $n > 1$
- In numero di chiamate a `fib` è dell'ordine di $O(2^n)$

Dipartimento di Matematica e Informatica

Memoria Dinamica

- L'allocazione statica non è sufficiente: è necessaria l'allocazione e la deallocazione (durante l'esecuzione del programma) dinamica della memoria
- Stack (pila) per le attivazioni di procedure (o blocchi in-line) poiché seguono una policy LIFO (Last In First Out): l'ultima procedura chiamata (o l'ultimo blocco nel quale siamo entrati) è la prima dalla quale usciremo
- In alcuni casi, lo stack non è sufficiente
- Esistono specifiche operazioni per l'allocazione e la deallocazione esplicita della memoria, come accade per il C con le funzioni `malloc` e `free`
- Poiché allocazione e deallocazione possono essere alternate con un qualsiasi ordine, non è possibile utilizzare uno stack per gestire la memoria: viene utilizzata una struttura chiamata **heap**

Dipartimento di Matematica e Informatica

Gestione Statica della Memoria

- La gestione statica della memoria viene effettuata dal compilatore prima dell'inizio dell'esecuzione
- Gli oggetti per i quali la memoria è allocata staticamente risiedono in una zona di memoria fissata (determinata dal compilatore) e lì rimangono fino al termine del programma. Per esempio: variabili globali (visibili in tutto il programma), istruzioni del codice oggetto (solitamente non cambiano durante l'esecuzione), costanti (se il loro valore non dipende da altri valori sconosciuti a tempo di compilazione), tabelle generate dal compilatore per il supporto a runtime del linguaggio (per esempio, per gestire nomi, per type checking, per garbage collection, . . .)
- Se il linguaggio non supporta la ricorsione è possibile associare in maniera statica un'area di memoria per salvare le informazioni locali ad una procedura, per ciascuna procedura (variabili locali, eventuali parametri della procedura, indirizzo di ritorno, valori temporanei utilizzati per calcoli complicati e diverse informazioni di "bookkeeping", come i valori dei registri salvati, le informazioni per il debugging, ecc)
- Chiamate successive alla stessa procedura condividono la stessa area di memoria
- In assenza di ricorsione, non ci possono essere due chiamate attive alla stessa procedura contemporaneamente

Dipartimento di Matematica e Informatica

Gestione Dinamica della Memoria Tramite Stack I

- La maggior parte dei moderni linguaggi di programmazione permette una strutturazione a blocchi del programma
- I blocchi, sia in-line o associati a procedure, sono aperti e chiusi utilizzando uno schema LIFO: quando si entra in un blocco A e poi in un blocco B, prima di lasciare A, è necessario lasciare B
- Viene naturale gestire lo spazio di memoria utilizzato per salvare le informazioni locali utilizzando uno stack (pila)

```
A:{
  int a = 1;
  int b = 0;
  B:{
     int c = 3;
     int b = 3;}
  b=a+1;}
```

Dipartimento di Matematica e Informatica

Gestione Dinamica della Memoria Tramite Stack II

- Lo spazio di memoria allocato sullo stack e dedicato ad un blocco è chiamato **record di attivazione** o **frame**
- Un record di attivazione è associato ad una specifica attivazione di una procedura (è creato quando la procedura viene chiamata) e non alla dichiarazione della procedura: i valori che devono essere salvati in un record di attivazione (variabili locali, variabili temporanee, ecc) sono differenti per differenti chiamate alla stessa procedura
- Lo stack sul quale vengono salvati i record di attivazione è chiamato **runtime stack** (o stack di sistema)

Dipartimento di Matematica e Informatica

Record di Attivazione per Blocchi In-line I

Il record di attivazione contiene le seguenti informazioni:

- **Risultati intermedi**: può essere necessario salvare risultati intermedi, anche se il programmatore non associa nomi espliciti a questi valori. Per esempio, in

```
{int a =3;
b= (a+x)/ (x+y);}
```

  i valori intermedi a+x e x+y vengono esplicitamente salvati prima di effettuare la divisione. La necessità di salvare risultati intermedi sullo stack dipende anche dal compilatore che viene utilizzato e dall'architettura sulla quale viene compilato il programma. In molte architetture, questi valori possono essere salvati nei registri

- **Variabili Locali**: devono essere salvate in uno spazio di memoria la cui dimensione dipende dal numero e dal tipo delle variabili. In alcuni casi, ci possono essere delle dichiarazioni che dipendono da valori determinati a runtime (per esempio, array dinamici, presenti in alcuni linguaggi). In altri casi, il record di attivazione contiene anche una parte di dimensione variabile

Dipartimento di Matematica e Informatica

Record di Attivazione per Blocchi In-line II

- **Puntatore di Catena Dinamica**: questo campo salva un puntatore al record di attivazione precedente nello stack. Questa informazione è necessaria perché i record di attivazione possono avere dimensioni differenti. Viene chiamato anche **link dinamico** o **link di controllo**. L'insieme dei link implementati da questi puntatori è chiamato **catena dinamica**

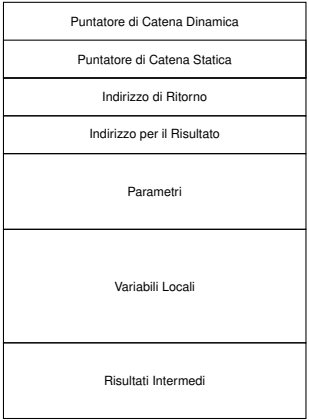
Dipartimento di Matematica e Informatica

Record di Attivazione per le Procedure I

Il caso per le procedure e le funzioni è analogo a quello per i blocchi in-line, con però alcune complicazioni per la gestione del flusso di controllo. I campi di un record di attivazione sono:

- Valori intermedi, variabili locali e puntatore di catena dinamica, come per i blocchi in-line
- Puntatore di catena statica: informazioni necessarie per implementare regole di scope statico
- Indirizzo di ritorno: l'indirizzo della prima istruzione da eseguire dopo che la chiamata alla procedura/funzione corrente ha terminato l'esecuzione
- Indirizzo per il risultato (presente solo nelle funzioni): indirizzo della locazione di memoria dove il sottoprogramma salva il valore che deve essere restituito dalla funzione. Questa locazione di memoria si trova all'interno del record di attivazione del chiamante
- Parametri: il valore dei parametri attuali utilizzati per chiamare la procedura o la funzione

Record di Attivazione per le Procedure II



- L'organizzazione dei record di attivazione varia da implementazione ad implementazione

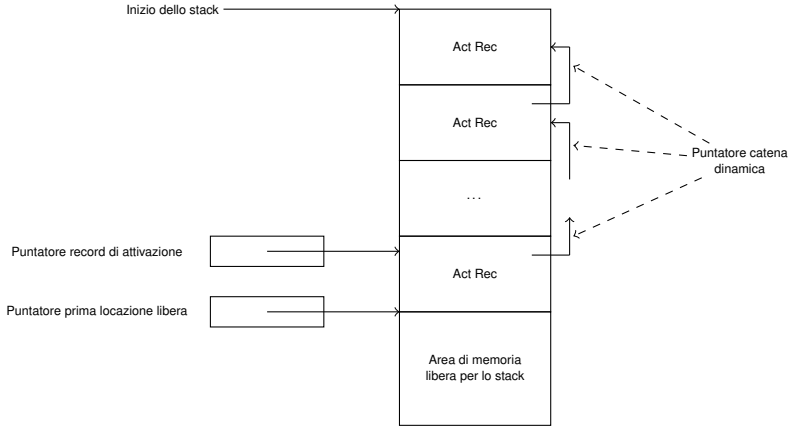
Dipartimento di Matematica e Informatica

Record di Attivazione per le Procedure III

- Il puntatore a catena dinamica e, in generale, ogni puntatore ad un record di attivazione, punta ad una zona fissa (solitamente centrale) del record di attivazione
- Gli indirizzi dei diversi campi sono ottenuti aggiungendo un offset negativo o positivo al valore del puntatore
- I nomi delle variabili solitamente non vengono memorizzati nei record di attivazione: il compilatore sostituisce i riferimenti a variabili locali con indirizzi relativi rispetto ad una posizione fissa del record di attivazione
- Nel caso di riferimenti a variabili non locali è possibile utilizzare un meccanismo per evitare il salvataggio dei nomi e quindi evitare una ricerca a runtime del nome nello stack
- I compilatori moderni spesso ottimizzano il codice che producono e salvano alcune informazioni sui registri invece che nel record di attivazione
- Per semplicità, negli esempi successivi, supponiamo che i nomi delle variabili siano salvati nei record di attivazione

Gestione dello Stack I

Assumiamo che lo stack di sistema cresca verso il basso (la direzione di crescita dipende dall'implementazione)



Dipartimento di Matematica e Informatica

Gestione dello Stack II

- Un puntatore esterno allo stack punta all'ultimo record di attivazione sullo stack: **puntatore al record di attivazione** chiamato anche frame o puntatore all'ambiente corrente
- Puntatore alla prima locazione libera: può essere omesso se il puntatore al record di attivazione punta sempre ad una posizione che è ad una distanza predefinita dall'inizio dell'area libera sullo stack
- I record di attivazione sono inseriti e rimossi dallo stack a tempo di esecuzione
- Quando si entra in un blocco o quando viene chiamata una procedura il record di attivazione associato viene inserito nello stack: quando la procedura termina o quando si esce dal blocco, questo viene rimosso
- La gestione a runtime dello stack di sistema è implementata tramite alcuni frammenti di codice che il compilatore (o l'interprete) inserisce immediatamente prima e dopo la chiamata di una procedura o prima dell'inizio e dopo la fine di un blocco
- Terminologia: **chiamante** si riferisce al programma o alla procedura che effettua la chiamata, **chiamato** si riferisce alla procedura chiamata

Dipartimento di Matematica e Informatica

Gestione dello Stack III

- Una frazione di una parte di codice detta **sequenza di chiamata** è inserita all'interno del chiamate e viene eseguita subito prima della chiamata a procedura, la parte rimanente viene inserita ed eseguita subito dopo la chiamata
- Due parti di codice sono inserite nel chiamato: un prologo che deve eseguito immediatamente dopo la chiamata ed un epilogo che viene eseguito quando la procedura termina l'esecuzione
- L'effettiva suddivisione su ciò che il chiamante ed il chiamato fanno dipende dal compilatore
- Per ottimizzare la dimensione di codice generata, è preferibile assegnare gran parte delle attività al chiamato, poiché il codice viene inserito una sola volta e non molte volte

Dipartimento di Matematica e Informatica

Attività alla Chiamata I

- Modifica del valore contatore del programma: necessario per passare il controllo alla procedura chiamata. Il vecchio valore (incrementato) deve essere salvato per mantenere l'indirizzo di ritorno
- Allocazione dello spazio nello stack: lo spazio per il nuovo record di attivazione deve essere allocato e quindi il puntatore alla prima locazione di memoria libera nello stack deve essere aggiornato di conseguenza
- Modifica del puntatore al record di attivazione: il puntatore deve puntare al nuovo record di attivazione per la procedura chiamata
- Passaggio di parametri: questa attività viene solitamente effettuata dal chiamante, visto che diverse chiamate alla stessa procedura possono avere parametri diversi
- Salvataggio dei registri: i valori per la gestione del controllo, solitamente contenuti in registri, devono essere salvati, per esempio nel caso del vecchio puntatore al record di attivazione
- Esecuzione di codice per l'inizializzazione: alcuni linguaggi richiedono costrutti espliciti per inizializzare alcuni elementi contenuti nel nuovo record di attivazione

Attività al Ritorno del Controllo al Chiamante I

Quando il controllo ritorna al programma chiamante, l'epilogo (nel chiamato) e la sequenza di chiamata (nel chiamante) devono eseguire le seguenti operazioni

- Aggiornamento del valore del contatore del programma: necessario per restituire il controllo al chiamante
- Restituzione del valore di ritorno: i valori dei parametri che passano informazioni dal chiamato al chiamante o i valori calcolati dalla funzione devono essere salvati in opportune locazioni che solitamente si trovano nel record di attivazione del chiamante e sono accessibili a partire dal record di attivazione dalla procedura chiamata
- Ripristino dei registri: i valori dei registri precedentemente salvati deve essere ripristinato. In particolare, il vecchio valore del puntatore al record di attivazione deve essere ripristinato
- Esecuzione del codice per la finalizzazione: alcuni linguaggi richiedono l'esecuzione di codice per la finalizzazione prima che gli oggetti locali possano essere distrutti
- Deallocazione dello spazio sullo stack: il record di attivazione della procedura che ha terminato la sua esecuzione deve essere rimosso dallo stack. Il puntatore alla prima posizione libera dello stack deve essere modificato di conseguenza

Gestione Dinamica Tramite Heap I

Nel caso in cui il liguaggio includa specifici comandi per l'allocazione di memoria, come per esempio C o Pascal, la gestione tramite stack non è sufficiente

```
int *p, *q; /* p,q NULL pointers to integers */
p = malloc(sizeof (int));
/* allocates the memory pointed to by p */
q = malloc(sizeof (int));
/* allocates the memory pointed to by q */
*p = 0; /* dereferences and assigns */
*q = 1; /* dereferences and assigns */
free(p); /* deallocates the memory pointed to by p */
free(q); /* deallocates the memory pointed to by q */
```

- Poiché le operazioni di deallocazione della memoria sono effettuate nello stesso ordine delle allocazioni (prima `p` poi `q`), la memoria non può essere gestita in maniera LIFO

Dipartimento di Matematica e Informatica

Gestione Dinamica Tramite Heap II

- Per gestire l'allocazione di memoria esplicita viene usata una particolare zona di memoria detta **heap**
- Notare che questo temine viene solitamente utilizzato anche per indicare una particolare struttura dati che può essere rappresentata come un albero binario o come vettore, utilizzata per implementare efficientemente code con priorità
- Nel gergo dei linguaggi di programmazione, un heap è semplicemente un'area di memoria nella quale i blocchi di memoria possono essere allocati e deallocati in maniera relativamente libera (non ha quindi niente a che fare con la struttura dati)
- I metodi per la gestione dell'heap si dividono in due categorie a seconda del fatto che i blocchi vengano considerati di dimensioni fisse o variabili

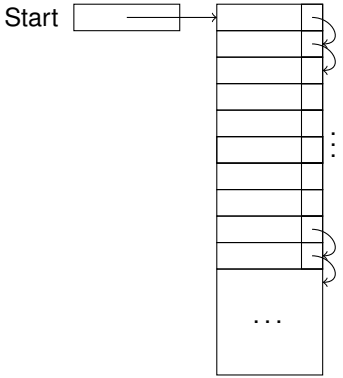
Dipartimento di Matematica e Informatica

Blocchi di Lunghezza Fissa I

- L'heap è diviso in un certo numero di elementi, o **blocchi**, di dimensione fissa ed abbastanza limitata, collegati in una struttura a lista chiamata **lista libera**
- A runtime, quando un'operazione richiede l'allocazione di un blocco di memoria nell'heap (per esempio utilizzando il comando malloc), il primo elemento della lista libera è rimosso dalla lista, il puntatore a questo elemento viene restituito e il puntatore alla lista libera è aggiornato per puntare all'elemento successivo

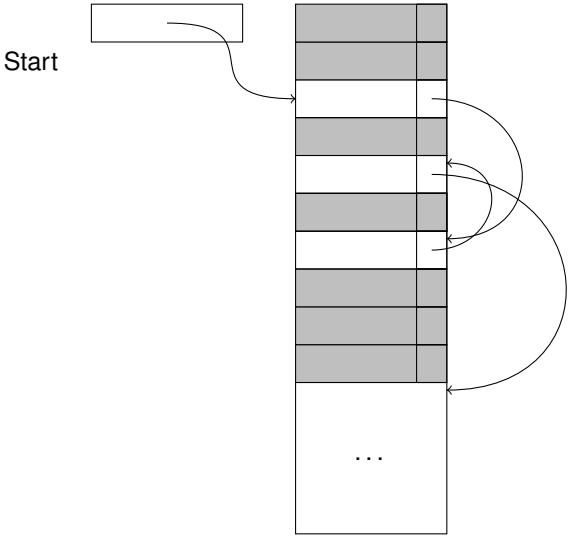
Dipartimento di Matematica e Informatica

Blocchi di Lunghezza Fissa II



- Quando la memoria viene liberata o dellocata (per esempio utilizzando free), il blocco liberato è nuovamente collegato alla testa della lista libera

Blocchi di Lunghezza Fissa III



Dipartimento di Matematica e Informatica

Blocchi di Lunghezza Variabile

- Se il linguaggio permette l'allocazione a runtime di memoria di lunghezza variabile, per esempio array di dimensione variabile, i blocchi a dimensione fissa non sono più adeguati
- La memoria che deve essere allocata può avere una dimensione maggiore della dimensione di un blocco fisso, ed il salvataggio di un array richiede una regione contigua di memoria che non può essere allocata come una serie di blocchi
- Esistono diverse tecniche con lo scopo di aumentare l'occupazione di memoria ed la velocità di esecuzione per la gestione delle operazioni nell'heap: buone implementazioni tendono ad scegliere un compromesso

Dipartimento di Matematica e Informatica

Problemi

- Frammentazione della memoria
- Frammentazione interna, quando si alloca un blocco di dimensione maggiore: la porzione di memoria interna al blocco non utilizzata viene sprecata finché il blocco non è restituito alla lista libera
- Frammentazione esterna: quando la lista libera è composta da blocchi di piccole dimensioni e non si riesce ad utilizzare effettivamente la memoria libera, anche se la somma della memoria libera totale è sufficiente, perché i blocchi non sono contigui
- Soluzione: ricompattare la memoria libera, muovendo blocchi liberi ed allocati
- Tecniche di allocazione di memoria per blocchi di lunghezza variabile: **unica lista libera** o **liste libere multiple**

Unica Lista Libera

- Un'unica lista libera, inizialmente composta da un unico blocco di memoria che contiene l'intero heap
- Risulta conveniente mantenere i blocchi della massima dimensione possibile
- Quando viene richiesta l'allocazione di un blocco di *n* parole (word) di memoria, le prime *n* parole vengono allocate e il puntatore all'inizio dell'heap viene incrementato di *n*. Richieste successive vengono gestite in maniera simile
- I blocchi deallocati vengono collegati in una lista libera
- Quando termina lo spazio dedicato all'heap bisogna riutilizzare la memoria deallocata: **utilizzo diretto della lista libera** oppure **compattazione della memoria libera**

Dipartimento di Matematica e Informatica

Utilizzo Diretto della Lista Libera I

- Quando viene richiesta l'allocazione di un blocco di memoria di n parole si cerca attraverso la lista libera un blocco di dimensione $k \geq n$ parole
- La memoria viene allocata all'interno di questo blocco
- La parte inutilizzata del blocco (di dimensione $k - n$), se più grande di una soglia prefissata, viene utilizzata per creare un nuovo blocco che viene inserito nella lista libera
- La ricerca di un blocco di dimensioni sufficienti può essere svolta in due modi: **first fit** o **best fit**
- Utilizzando first fit si cerca il primo blocco di dimensione sufficiente, privilegiando il tempo di gestione
- Utilizzando best fit, si cerca il blocco di dimensione minima tra tutti i blocchi di dimensione sufficiente, privilegiando l'occupazione di memoria
- Per entrambi, il costo dell'allocazione è lineare rispetto al numero di blocchi sulla lista libera
- Se i blocchi vengono mantenuti in ordine crescente di dimensione, le due tecniche sono uguali ed il costo di inserimento di un blocco nella lista libera passa da costante a lineare

Dipartimento di Matematica e Informatica

Utilizzo Diretto della Lista Libera II

- Quando un blocco deallocato torna nella lista libera, per ridurre la frammentazione esterna, viene effettuato un controllo per determinare se i blocchi fisicamente adiacenti sono liberi. Se sì, questi vengono compattati in un unico blocco. Compattazione **parziale** perché si compattano solo i blocchi adiacenti

Dipartimento di Matematica e Informatica

Compattazione della Memoria Libera

- Quanto termina lo spazio inizialmente allocato per l'heap, tutti i blocchi che sono ancora attivi sono spostati alla fine, lasciando tutta la memoria libera in un singolo blocco contiguo
- Il puntatore all'heap è aggiornato per puntare all'inizio dell'unico blocco di memoria libera, e l'allocazione riparte
- I blocchi di memoria allocata devono essere spostabili, proprietà che non è sempre garantita (per esempio, in caso di blocchi i cui indirizzi sono salvati in puntatori sullo stack)

Liste Libere Multiple

- Per ridurre il costo di allocazione dei blocchi, alcune tecniche di gestione dell'heap utilizzano più liste libere per blocchi di dimensione diversa
- Quando viene richiesto un blocco di dimensione n, viene scelto un blocco dalla lista che contiene blocchi di dimensione uguale o maggiore ad n (con possibile frammentazione interna se la dimensione del blocco è maggiore di n)

Dipartimento di Matematica e Informatica

Linguaggi Imperativi e Dichiarativi

- In maniera estremamente sintetica, possiamo classificare come **imperativi** i linguaggi che hanno sia ambiente che stato che possono variare, mentre i linguaggi **dichiarativi** hanno solamente l'ambiente che può cambiare
- I linguaggi imperativi sono ispirati alla struttura fisica del computer
- Il concetto di memoria (o stato) è interpretato come l'insieme delle associazioni tra locazioni di memoria e valori salvati in queste locazioni
- Un programma, secondo questo paradigma, è un insieme di comandi imperativi e calcoli che consistono in una sequenza di passi che modifica lo stato, utilizzando come comando elementare l'assegnamento

Dipartimento di Matematica e Informatica

Linguaggi Imperativi

- La terminologia “imperativo” deriva dal linguaggio naturale: così come diciamo con una frase imperativa “prendi quella mela” per esprimere un comando, analogamente diciamo “assegna ad x il valore 1”
- Molti dei linguaggi di programmazione solitamente usati sono imperativi (Fortran, Algol, Pascal, C, Java, ecc)

Dipartimento di Matematica e Informatica

Linguaggi Dichiarativi

- I linguaggi dichiarativi sono stati introdotti con lo scopo di offrire un paradigma di programmazione ad alto livello, vicino alla notazione matematica e logica, astraendo dalle caratteristiche della macchina fisica sulla quale il programma viene eseguito
- Nei linguaggi dichiarativi (o almeno nelle versioni “pure”) non esistono comandi per modificare lo stato, poiché non ci sono né variabili che possono essere modificate né la funzione semantica della memoria
- I programmi sono insiemi di dichiarazioni (dalla quale deriva il paradigma) di funzioni o relazioni che definiscono nuovi valori
- Secondo il meccanismo elementare utilizzato per definire le caratteristiche di un risultato, i linguaggi dichiarativi sono divisi in due classi: **linguaggi di programmazione funzionali** e **logici**

Dipartimento di Matematica e Informatica

Linguaggi di Programmazione Funzionali e Logici

- Neil linguaggi funzionali (LISP, ML, Haskell), la computazione consiste nella valutazione di funzioni definite dal programmatore utilizzando regole di tipo matematico (sostanzialmente composizione ed applicazione)
- Nei linguaggi logici (Prolog) la computazione è basata sulla deduzione della logica del prim'ordine
- In verità, esistono linguaggi funzionali e logici "non puri" che hanno anche caratteristiche imperative (in particolare, permettono l'assegnamento)

Comandi

- L'assegnamento è il comando base nei linguaggi imperativi (e nei linguaggi dichiarativi "non puri")
- I comandi rimanenti servono a definire il controllo di sequenza
- Tre categorie:
  - **Comandi per il controllo di sequenza esplicito**: comando sequenziale e `goto`
  - **Comandi condizionali o di selezione**: per la specifica di percorsi alternativi per la computazione che dipendono dal soddisfacimento di alcune condizioni
  - **Comandi iterativi**: ripetizione di un comando per un certo numero di volte, o finché una condizione è verificata

Dipartimento di Matematica e Informatica

Comando Sequenziale

- Il comando sequenziale, indicato in molti linguaggi con `;`, specifica direttamente l'esecuzione sequenziale di due comandi
- In `C1 ; C2` l'esecuzione di `C2` inizia subito dopo il termine di `C1`
- Nei linguaggi in cui la valutazione di un comando resituisce anche un valore, il valore restituito dalla valutazione del comando sequenziale è quello del secondo argomento
- Sequenze di comandi sono possibili come `C1 ; C2 ; ... ; Cn` con l'assunzione implicita che `;` associ a sinistra

Dipartimento di Matematica e Informatica

Comando Composto

- Nei moderni linguaggi imperativi è possibile raggruppare una sequenza di comandi in un comando composto utilizzando specifici delimitatori. In Algol `begin ... end`, in C e Java `{ ... }`
- Un tale comando composto, chiamato anche blocco, può essere utilizzato in una qualsiasi situazione in cui ci si aspetta un comando semplice, quindi può essere a sua volta utilizzato all'interno di un comando composto, per creare strutture annidate di complessità arbitraria

Dipartimento di Matematica e Informatica

Goto I

- Presente già nelle primissime versioni dei linguaggi di programmazione
- Di due diversi tipi: condizionale o diretto
- Ispirato alle istruzioni jump dei linguaggi assembly
- Il comando `goto A` trasferisce il controllo al punto del programma dove è definita la label `A` (i vari linguaggi differiscono sul significato di label)
- Questo comando è stato al centro di molti dibattiti a partire dagli anni '70 (vedi anche l'articolo di Dijkstra `https://it.wikipedia.org/wiki/Considered_harmful`) e, dopo circa 50 anni possiamo dire che i detrattori hanno vinto sui sostenitori
- Il goto non è essenziale per l'espressività del linguaggio
- Un teorema dimostrato da Böhm e Jacopini asserisce che ogni programma che usa il goto può essere trasformato in un altro programma equivalente senza goto
- Utilizzando goto è facile scrivere codice incomprensibile (spaghetti code)

Dipartimento di Matematica e Informatica

Goto II

- Pensiamo per esempio ad un programma di grandi dimensioni dove abbiamo inserito salti tra punti che sono distanti migliaia di linee di codice, o ad un sottoprogramma dal quale si esce tramite goto, secondo alcune condizioni
- I casi precedenti (ma anche molti altri) rendono il codice molto difficile da capire, e quindi difficile da modificare, correggere e mantenere, con evidenti conseguenze negative in termini di costi
- Il goto non si accorda bene con altri meccanismi presenti in linguaggi di alto livello: cosa succede se saltiamo all'interno di un blocco? Quando e come bisogna inizializzare il record di attivazione per questo blocco per far funzionare il tutto?
- Se il goto fosse utilizzato in maniera controllata, localmente in piccole regioni di codice, gran parte di questi svantaggi sparirebbero
- Le situazioni in cui questo comando può essere utile, come l'uscita da loop, ritorno da sottoprogrammi e gestione delle eccezioni possono essere gestite da costrutti più appropriati nei moderni linguaggi di programmazione
- Nei moderni linguaggi di alto livello il goto sta scomparendo. Java è il primo linguaggio commerciale ad aver completamente rimosso il goto

Altri Comandi di Controllo di Sequenza

- Se il goto è pericoloso nel caso generale, ci sono alcuni casi limitati in cui può risultare utile
- Molti linguaggi mettono a disposizione forme limitate di salti per alcune situazioni:
  - **break**: per terminare l'esecuzione di un loop, di un caso o, in alcuni linguaggi, di un blocco
  - **continue**: per terminare l'iterazione corrente in un comando iterativo e passare alla prossima iterazione
  - **return**: per terminare la valutazione della funzione, restituire il controllo al chiamante, e qualche volta restituire un valore
  - **eccezioni**

Comandi Condizionali I

- I **comandi condizionali** o di **selezione** esprimono un'alternativa tra due possibili continuazioni della computazione a seconda del valore di un'espressione logica
- Due gruppi: `if` e `case`
- Il comando `if`, introdotto originariamente nel linguaggio ALGOL60, è presente in quasi tutti i linguaggi imperativi ed anche in alcuni linguaggi dichiarativi
- Utilizzo: `if Boolexp then C1 else C2` dove `Boolexp` è un'espressione booleana e `C1` e `C2` sono comandi. Si può utilizzare anche senza la parte `else`
- La presenza di if annidati come in `if Boolexp1 if Boolexp2 then C1 else C2` causa problemi di ambiguità, che possono essere risolti utilizzando una grammatica appropriata che descrive formalmente le regole adottate nel linguaggio (per esempio, il ramo else appartiene all'if più interno, come è adottato il Java e nella maggior parte dei linguaggi)
- Per evitare problemi di ambiguità, alcuni linguaggi utilizzano un terminatore per indicare il termine del comando condizionale `if Boolexp then C1 else C2 endif`
- In alcuni casi, possono esserci più rami

Dipartimento di Matematica e Informatica

Comandi Condizionali II

```
if Boolexp1 then C1
elseif Boolexp2 then C2
...
elseif Boolexpn then Cn
else Cn+1
endif
```

- L'implementazione del comando condizionale non causa nessun problema ed utilizza le istruzioni di test e jump presenti nella macchina fisica sottostante

Dipartimento di Matematica e Informatica

Case I

Specializzazione del comando `if` ma con più rami. Nella sua forma più semplice:

```
case Exp of
label1: C1;
label2: C2;
...
labeln: Cn;
else Cn+1
```

dove `Exp` è un'espressione il cui valore è di un tipo compatibile con quello delle etichette (label) `label1, ..., labeln`, mentre `C1, ..., Cn+1` sono comandi

- Ogni etichetta è rappresentata da una o più costanti e le costanti utilizzate nelle varie etichette sono tutte diverse
- I tipi permessi dalle etichette, così come la loro forma, variano da linguaggio a linguaggio
- Nella maggior parte dei casi, è permesso l'utilizzo di tipi discreti
- Comando analogo all'`if` con più rami

Dipartimento di Matematica e Informatica

Case II

- Una volta che viene valutata l'espressione `Exp`, il comando che appare nell'unico ramo la cui etichetta include il valore che risulta dalla valutazione di `Exp` viene eseguito
- Il ramo else è eseguito ogni volta che non ci sono altri rami la cui etichetta soddisfa la condizione calcolata prima
- Il costrutto `case` può essere espresso utilizzando una serie di `if` innsestati: tuttavia, molti linguaggi mantengono comuque il `case` sia per migliorare la leggibilità del codice sia perché è più efficiente compilare il `case` rispetto ad una serie di `if` innestati
- Infatti, il case è implementato in linguaggio assembly utilizzando un vettore di celle contigue chiamate **jump table**, in cui ciascun elemento della tabella contiene l'indirizzo della prima istruzione del comando corrispondente al ramo
- L'espressione che appare come argomento al case viene valutata: il valore ottenuto è usato come offset (indice) che determina la posizione nella jump table dell'indirizzo della prima istruzione del comando corrispondente al ramo

Dipartimento di Matematica e Informatica

Comandi Iterativi

- I comandi visti finora, a differenza del goto, permettono di esprimere solamente computazioni finite, la cui lunghezza massima è determinata staticamente dalla lunghezza del codice scritto
- Un linguaggio che ha solamente questa possibilità è molto limitato e sicuramente non è Turing completo
- Sono necessarie istruzioni di jump che permettono la ripetizione di un gruppo di istruzioni
- Nei linguaggi di alto livello, esistono due meccanismi fondamentali: **iterazione strutturata** e **ricorsione**

Dipartimento di Matematica e Informatica

Iterazione e Ricorsione

- **Iterazione determinata** ed **iterazione indeterminata**
- Nell'iterazione determinata, la ripetizione è implementata attraverso un costrutto che permette l'iterazione un numero prefissato di volte
- L'iterazione indeterminata è realizzata da costrutti che permettono di iterare finché una condizione non diventa vera
- La ricorsione invece permette di esprimere dei cicli in modo implicito, permettendo alla funzione di chiamare se stessa e ripetendo quindi il corpo un numero arbitrario di volte
- La ricorsione è più comune nei linguaggi dichiarativi

Dipartimento di Matematica e Informatica

Iterazione Indeterminata I

- Implementata da costrutti linguistici composti da due parti: una **condizione** o guardia del ciclo e un **corpo** costituito da un comando (o un comando composto)
- Il corpo viene ripetuto finché la guardia non risulta falsa (o vera, a seconda del costrutto usato)
- La forma più comune è il `while`, introdotto originariamente in ALGOL: `while (Boolexp) do C`
- Significato: (1) l'espressione booleana `Boolexp` viene valutata: (2) se risulta vera viene eseguito il comando `C` e si ritorna al passo (1) altrimenti il `while` termina
- In alcuni linguaggi ci sono anche comandi per testare la condizione dopo l'esecuzione del comando (che viene quindi sempre eseguito almeno una volta). Per esempio, in Pascal: `repeat C until Boolexp` che è un'abbreviazione per `C; while not Boolexp do C`. In C: `do C while (Boolexp)` che corrisponde a `C; while (Boolexp) do C`
- Il costrutto while corrisponde direttamente ad un loop che viene implementato nella macchina fisica tramite un'istruzione di jump con condizione
- Potenza di questo costrutto: aggiungendolo ad un linguaggio di programmazione che contiene solo assegnamento e comandi condizionali lo rende Turing completo

Iterazione Determinata I

- L'iterazione determinata (alcune volte chiamata iterazione controllata numericamente) è implementata usando una variante del comando `for`: `for I = start to end by step do body` dove `I` è una variabile, chiamata indice (o contatore o variabile di controllo), `start` ed `end` sono due espressioni (per semplicità assumiamo che siano di tipo intero e, in generale, devono essere discrete), `step` è una costante intera diversa da 0 e `body` è il comando che viene ripetuto
- Un importante vincolo semantico impone che la variabile di controllo non possa essere modficata (né implicitamente né esplicitamente) durante l'esecuzione del corpo (body)
- Semantica:
  1. Le espressioni `start` ed `end` vengono valutate. I valori vengono congelati e salvati in variabili dedicate (che non possono essere modificate dal programmatore). Chiamiamole `start_save` ed `end_save`
  2. `I` viene inizializzata con il valore di `start_save`
  3. Se il valore di `I` è strettamente maggiore del valore di `end_save`, l'esecuzione del comando `for` termina
  4. Viene eseguito il corpo (`body`) e viene incrementata `I` del valore `step`
  5. Ritorna al punto 3

Dipartimento di Matematica e Informatica

Espressività dell'Iterazione Determinata

- Utilizzando l'iterazione determinata si può esprimere la ripetizione di un comando n volte, dove n è un valore arbitrario non noto quando il programma viene scritto, ma è fissato quando l'iterazione inizia
- Questa situazione non può essere espressa utilizzando solamente comandi condizionali ed assegnamenti
- Tuttavia, da sola l'iterazione determinata non è sufficiente per rendere il linguaggio Turing completo

Programmazione Strutturata I

- Le critiche al costrutti goto negli anni ’70 non furono un fenomeno isolato ma bensì un problema di un dibattito più ampio che ha portato all’affermazione della **programmazione strutturata**, l’antenato della moderna metodologia di programmazione
- Consiste in una serie di prescrizioni con l’obiettivo di permettere lo sviluppo di software strutturato sia nel codice che nel flusso di controllo: sia precisi metodi di sviluppo che opportune tipologie di comandi da utilizzare (in sostanza, tutti quelli che abbiamo visto finora ad eccezione del goto)
- **Progettazione top-down** del programma o **gerarchica**: il programma è sviluppato attraverso raffinamenti successivi, iniziando da una prima (e molto astratta) specifica ed aggiungendo ulteriori dettagli ad ogni passo
- **Modularizzazione del codice**: raggruppamento dei comandi che rappresentano specifiche funzioni nell’algoritmo (comandi composti, costrutti per l’astrazione come procedure, funzioni e anche moduli, se il linguaggio lo permette)
- **Utilizzo di nomi significativi**: l’utilizzo di nomi significativi per le variabili, così come per le procedure ecc, semplifica enormemente il processo di comprensione del codice, rendendo più semplici i cambiamenti. Anche se può sembrare ovvio, in pratica solitamente viene ignorato

Programmazione Strutturata II

- **Utilizzo estensivo di commenti**: i commenti sono essenziali per capire, testare, verificare, correggere e modificare il codice. Un programma senza commenti diventa rapidamente incomprensibile, una volta che ha raggiunto una certa lunghezza
- **Utilizzo di tipi di dato strutturati**: l'utilizzo di tipi di dato appropriati, per esempio i record, per raggruppare e strutturare le informazioni, anche se di tipi eterogenei, facilita sia la progettazione che il mantenimento del codice. Per esempio, se possiamo utilizzare una sola variabile, di tipo record studente, per salvare le informazioni riguardanti il cognome, il numero di registrazione e l'anno di immatricolazione per uno studente, la struttura del programma risulterà molto più chiara rispetto al caso in cui vengono utilizzate tre variabili diverse per mantenere le informazioni riguardanti uno studente
- **Utilizzo di costrutti strutturati per il controllo**: un aspetto essenziale. Per aderire alla programmazione strutturata è necessario utilizzare costrutti di controllo strutturati, o costrutti che solitamente hanno un singolo punto d'entrata ed un singolo punto di uscita (tutti tranne il goto)

Dipartimento di Matematica e Informatica

Ricorsione I

- Meccansimo alternativo all'iterazione per ottenere linguaggi di programmazione Turing equivalenti
- In maniera informale, una funzione (o procedura) ricorsiva è una procedura nella quale il corpo contiene una chiamata a sè stessa
- Possibilità di avere **ricorsione indiretta** o **mutua ricorsione** dove una procedura P chiama una procedura Q che a sua volta chiama P
- Fibonacci:

```
int fib (int n) {
  if (n == 0) return 1;
  else if (n == 1) return 1;
    else return fib(n-1) + fib(n-2);
}
```

- Definizioni ricorsive, dette anche definizioni induttive, sono comuni in matematica: si descrive il risultato dell'applicazione di una funzione *f* ad un argomento X in termini di applicazione della stessa f ad argomenti più "piccoli" di X

Dipartimento di Matematica e Informatica

Ricorsione II

- Il dominio sul quale f è definita deve essere tale da non ammettere infinite catene di elementi sempre più piccoli: in questo modo, dopo l'applicazione della funzione f un certo numero di volte, si arriverà ad un caso terminale
- Per esempio, il fattoriale di un numero n è dato dal prodotto $n \cdot (n-1) \cdot \cdots \cdot 2 \cdot 1$
- Possiamo definire induttivamente il calcolo del fattoriale come segue: fattoriale(0) = 1, fattoriale(n+1) = $(n+1)\cdot$ fattoriale(n)

Dipartimento di Matematica e Informatica

Ricorsione in Coda (Tail) I

- La ricorsione richiede la gestione dinamica della memoria
- In alcuni casi si può evitare l'allocazione di nuovi record di attivazione per chiamate successive ad una singola funzione
- Concentriamoci su due funzioni ricorsive per il calcolo del fattoriale di un numero. La prima è una versione base:

```
int fact (int n){
  if (n <= 1) return 1;
  else
    return n * fact(n-1);
}
```

Dipartimento di Matematica e Informatica

Ricorsione in Coda (Tail) II

- Record di attivazione per la chiamata `fact(n)`

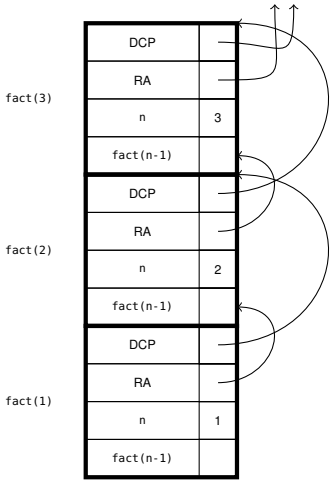
Dipartimento di Matematica e Informatica

Ricorsione in Coda (Tail) III

- Il campo Indirizzo Risultato contiene l'indirizzo dell'area di memoria nella quale deve essere restituito il risultato (cioè l'indirizzo del campo Risultato Intermedio del chiamante per chiamate successive alla prima)
- Il valore del campo Risultato Intermedio può essere calcolato solamente quando le chiamate ricorsive a `fact(n-1)` terminano, e viene utilizzato per calcolare `n` $\cdot$ `fact(n-1)` per ottenere il valore di `fact(n)`

Dipartimento di Matematica e Informatica

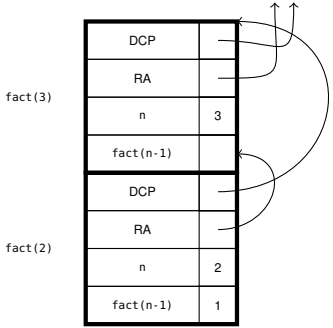
fact(3) I



Dipartimento di Matematica e Informatica

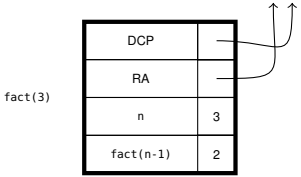
fact(3) II

- Quando viene raggiunto il caso terminale, la chiamata `fact(1)` termina immediatamente e restituisce il valore 1, che a sua volta viene restituito nel campo Risultato Intermedio del record di attivazione per `fact(2)`



fact(3) III

- A questo punto la chiamata a `fact(2)` può terminare, restituendo il valore $2 \cdot$ `fact(1)` $=$ `2` alla chiamata `fact(3)`



fact(3) IV

- Infine, la chiamata `fact(3)` termina restituendo al programma chiamante il valore $3 \cdot$ `fact(2)` $= 3 \cdot 2 = 6$

Dipartimento di Matematica e Informatica

Ricorsione Tail I

- Considera ora una seconda funzione

```
int factrc (int n, int res){
  if (n <= 1)
    return res;
  else
    return factrc(n-1, n * res)
}
```

- Se chiamata come `factrc(n,1)` restituisce n fattoriale
- La chiamata iniziale a `factrc(n,1)` produce n-1 chiamate ricorsive

```
factrc(n-1,n*1),
factrc(n-2,(n-1)*n*1),
. . . ,
factrc(1,2*...*(n-1)*n*1).
```

Dipartimento di Matematica e Informatica

Ricorsione Tail II

- Per $n > 1$ il valore calcolato dalla funzione generica `factrc(n,res)` è lo stesso valore calcolato da `factrc(n-1,n*res)`, senza nessuna computazione aggiuntiva
- Il valore finale restituito dalla chiamata iniziale `factrc(n,1)` è quindi lo stesso valore restituito dall'ultima chiamata ricorsiva `factrc(1,2*...*(n-1)*n*1)`, che è quindi `1*2*...*(n-1)*n`, senza bisogno di risalire la catena di chiamate utilizzando risultati intermedi per poter calcolare il valore finale
- Una volta che `factrc(n,res)` ha chiamato ricorsivamente `factrc(n-1,n*res)` non c'è più bisogno di mantenere le informazioni presenti nel record di attivazione per la chiamata `factrc(n,res)`, dato che tutte le informazioni necessarie per effettuare il calcolo del valore finale sono passate a `factrc(n-1,n*res)`
- Il record di attivazione per la chiamata ricorsiva `factrc(n-1,n*res)` può riutilizzare la memoria allocata per il record per `factrc(n,res)`
- Questa considerazione è valida anche per le chiamate successive, quindi, in breve, la funzione `factrc` necessita di una singola area di memoria per allocare un solo record di attivazione, indipendentemente dal numero di chiamate ricorsive che devono essere fatte

Ricorsione Tail III

- Abbiamo quindi ottenuto una funzione ricorsiva per la quale la memoria può essere allocata staticamente
- Questo tipo di ricorsione è detta **ricorsione in coda** (o **tail**). **Definizione**: sia f una funzione nella quale, nel suo corpo, c'è una chiamata ad una funzione g, diversa o uguale ad f. La chiamata a g è detta una *chiamata in coda (tail call) se la funzione f restituisce il valore calcolato da g senza dover effettuare altre computazioni. Diciamo che la funzione f è **ricorsiva in coda** se tutte le chiamate riscorsive presenti in f sono chiamate in coda
- Alcuni compilatori ottimizzano il codice tail ricorsivo
- Risulta sempre possibile trasformare una funzione non tail ricorsiva in una tail ricorsiva equivalente, rendendola leggermente più complicata
- L'idea è quella di eseguire tutte le operazioni che dovrebbero essere fatte dopo la chiamata, prima della chiamata stessa
- La parte di computazione che non può essere svolta prima della chiamata ricorsiva (perché, per esempio, utilizza il risultato) può essere passata, utilizzando opportuni parametri aggiuntivi, alla chiamata ricorsiva stessa, come accade per la funzione `factrc` vista prima

Ricorsione Tail IV

- Il calcolo del fattoriale viene eseguito incrementalmente da chiamate ricorsive successive, in modo analogo a come viene calcolato da una funzione iterativa come:

```
int fact_it (int n, int res){
  res=1;
  for (i=n; i>=1; i--)
    res = res*i;
}
```

- Possiamo anche trasformare la funzione `fib` in una funzione tail ricorsiva `fibrc` aggiungendo due parametri aggiuntivi

Dipartimento di Matematica e Informatica

Ricorsione Tail V

```
int fibrc (int n, int res1, int res2){
  if (n == 0)
    return res2;
  else
    if (n == 1)
      return res2;
  else
    return fibrc(n-1,res2,res1+res2);
}
```

- La chiamata `fibrc(n,1,1)` calcola l'n-esimo valore della successione di Fibonacci
- Se vogliamo rendere i parametri aggiuntivi invisibili, possiamo incapsulare la funzione all'interno di un'altra con un unico parametro `n`, per esempio

Dipartimento di Matematica e Informatica

Ricorsione Tail VI

```
int fibrctrue (int n){
  return fibrc(n,1,1);
}
```

- La trasformazione di una funzione in un'altra equivalente tail ricorsiva può essere effettuata automaticamente utilizzando una tecnica nota con il nome di **continuation passing style** che consiste nel rappresentare, in un determinato punto del programma, la parte rimanente del programma tramite una funzione detta **continuazione**
- Nel caso in cui vogliamo trasformare una funzione ricorsiva in una tail ricorsiva basta utilizzare una continuazione per rappresentare tutti i calcoli rimanenti e poi passarla alla chiamata ricorsiva

Dipartimento di Matematica e Informatica

Ricorsone ed Iterazione I

- Ricorsione ed iterazione sono due metodi equivalenti per raggiungere la stessa potenza espressiva
- L'utilizzo di una o dell'altra dipende spesso dalla predisposizione del programmatore invece che dalla natura del problema
- Per l'elaborazione di dati utilizzando strutture rigide (matrici, tabelle, ecc) è spesso più semplice utilizzare costrutti iterativi
- Quando invece vengono utilizzate strutture dati di natura simbolica, facilmente rappresentabili in maniera ricorsiva come liste o alberi, l'utilizzo della ricorsione risulta più naturale
- La ricorsione solitamente viene considerata meno efficiente rispetto all'iterazione, quindi i linguaggi dichiarativi sono ritenuti meno efficienti di quelli imperativi
- Le considerazioni precedenti sulla ricorsione tail ci fanno capire che non necessariamente la ricorsione è meno efficiente dell'iterazione, sia in termini di spazio che di tempo di esecuzione

Ricorsone ed Iterazione II

- Implementazioni dirette di funzioni ricorsive, come quelle ottenute dalla traduzione diretta delle definizioni induttive, possono risultare molto inefficenti (per esempio, `fib(n)` ha sia tempo di esecuzione che utilizzo di memoria che sono esponenziali in `n`)
- La funzione `fibrc` utilizza uno spazio costante ed esegue in tempo lineare in `n`

Dipartimento di Matematica e Informatica

Funzioni di Prima Classe I

- Si dice che un linguaggio di programmazione ha **funzioni di prima classe** (**first-class functions**) quando tratta le funzioni come cittadini di prima classe
- Il linguaggio deve supportare il passaggio di funzioni come argomenti ad altre funzioni, restituirle come valori da altre funzioni e assegnarle a variabili o salvarle in strutture dati
- Alcuni teorici dei linguaggi di programmazione richiedono anche il supporto di **funzioni anonime** (function literals), come accade in Java, Python o JavaScript
- In linguaggi con funzioni di prima classe, il nome delle funzioni non ha uno status speciale, viene trattato come un valore ordinario con un tipo funzione
- Il nome fu coniato da Christopher Strachey nel contesto di "'funzioni come cittadini di prima classe", a metà degli anni '60 (`https://en.wikipedia.org/wiki/First-class_function`)
- Le funzioni di prima classe sono necessarie per il paradigma di programmazione funzionale, nel quale le funzioni di ordine superiore sono normalmente utilizzate
- Esempio di funzione di ordine superiore: `map`, che prende come argomenti una funzione e una lista, e restituisce la lista formata dall'applicazione della funzione a ciascun elemento della lista

Funzioni di Prima Classe II

- Affinché un linguaggio supporti la funzione map, deve supportare il passaggio di una funzione come argomento
- Ci sono difficoltà implementative nel passaggio di funzioni come argomenti e nel restituirle come risultato, specialmente in presenza di variabili non locali introdotte in funzioni innestate ed anonime
- **Funarg problem** (Function Argument Problem)
- Nei primi linguaggi imperativi, questi problemi erano evitati o non supportando le funzioni come tipi per un risultato (come in ALGOL 60 e Pascal) oppure escludendo funzioni innestate e quindi variabili non locali (come in C)
- Uno tra i primi linguaggi funzionali, Lisp, scelse l'approccio dello scope dinamico, dove le variabili non locali si riferivano alla definizione più vicina della variabile stessa rispetto al punto di esecuzione, invece che rispetto a dove veniva definita
- Un supporto adeguato per funzioni di prima classe fu introdotto in Scheme e richiede la gestione dei riferimenti alle funzioni come chiusure (**closure**) invece che come puntatori a funzione, rendendo quindi necessaria la garbage collection

Dipartimento di Matematica e Informatica

Chiusure (closure)

- Le **chiusure** (closure, lexical closure o function closure) sono una tecnica necessaria nei linguaggi con funzioni di prima classe
- Una chiusura è un record che salva l'associazione funzione-ambiente: un mapping che associa ad ogni variabile libera della funzione (variabili che sono utilizzate localmente, ma definite in uno scope che la racchiude) un valore o una locazione in memoria alla quale il nome viene legato quando la chiusura è creata
- Una chiusura, a differenza di una funzione normale, permette alla funzione di accedere alle variabili non locali, anche quando la funzione è invocata al di fuori dello scope
- `https://en.wikipedia.org/wiki/Closure_(computer_programming)`

Dipartimento di Matematica e Informatica

Esempio

Applichiamo la funzione map con la funzione `f(int x){return a*x+b}` in C

```
typedef struct {
    int (*f)(int, int, int);
    int *a;
    int *b;
} closure_t;

void map(closure_t *closure, int x[], size_t n) {
    for (int i = 0; i < n; ++i)
        x[i] = (*closure->f)(*closure->a, *closure->b, x[i]);
}

int f(int a, int b, int x) {
    return a * x + b;
}

void main() {
    int l[] = {1, 2, 3, 4, 5};
    int a = 3;
    int b = 1;
    closure_t closure = {f, &a, &b};
    map(&closure, l, 5);
}
```

Dipartimento di Matematica e Informatica

Esempio

Applichiamo la funzione map con la funzione `f(int x){return a*x+b}` in Python

```
def map(f,l):
  return [f(x) for x in l]

def closure(a,b):
  def f(x):
    return a*x+b
  return f

l=[1, 2, 3, 4, 5]
f=closure(3,1)
print(map(f,l))
```

Dipartimento di Matematica e Informatica

Garbage Collection I

- Nei linguaggi senza deallocazione esplicita della memoria è necessario equipaggiare la macchina astratta con un meccanismo che possa automaticamente reclamare la memoria allocata nell'heap e non più utilizzata
- La precedente operazione viene svolta da un **garbage collector**, introdotto per la prima volta in LISP intorno al 1960 e poi introdotto in molti linguaggi, inizialmente funzionali e poi anche imperativi
- Java ha un garbage collector molto efficiente
- Un garbage collector consiste in due fasi (non sempre temporalmente separate): 1) distinguere gli oggetti ancora attivi da quelli non più attivi (garbage detection) e 2) raccogliere quelli non più attivi, cosicché il programma possa utilizzarli nuovamente
- Il concetto di "non più in uso" in un garbage collector solitamente è un'approssimazione conservativa: per motivi di efficienza, non tutti gli oggetti che non saranno più usati sono considerati tali dal garbage collector

Garbage Collection II

- Possiamo classificare i garbage collector classici a seconda di come determinano se un oggetto è o no in uso: basati su **reference counting** (contatore dei riferimenti) oppure su **mark and sweep** (marca e copia)

Dipartimento di Matematica e Informatica

Reference Counting I

- Un oggetto non è in uso se non ci sono puntatori ad esso
- Quando un oggetto viene creato nell'heap, allo stesso tempo viene allocato un intero: un contatore di riferimenti per l'oggetto, inaccessibile al programmatore
- Per ogni oggetto, la macchina astratta deve assicurarsi che il contatore contenga il numero di puntatori ancora attivi verso questo oggetto
- Quando un oggetto viene creato (gli viene assegnato per la prima volta un puntatore), il suo contatore è inizializzato a 1
- Quando troviamo un assegnamento tra puntatori `p = q`, il contatore che appartiene all'oggetto puntato da `q` viene incrementato di 1 mentre quello di `p` decrementato di 1
- Quando si esce da un ambiente locale, i contatori di tutti gli oggetti puntati da un puntatore locale a quell'ambiente vengono decrementati di 1
- Quando un contatore raggiunge il valore 0, il corrispondente oggetto può essere deallocato e restituito alla lista libera

Dipartimento di Matematica e Informatica

Reference Counting II

- Tuttavia, tale oggetto potrebbe contenere puntatori al suo interno, quindi, prima di restituire la memoria occupata da un oggetto alla lista libera, la macchina astratta segue i puntatori all'interno dell'oggetto e decrementa di 1 tutti i puntatori degli oggetti puntati, e recupera tutti gli oggetti il cui contatore è 0
- Vantaggio: incrementale perché il controllo e la raccolta sono alternate alle operazioni normali del programma
- Svantaggio: non è possibile deallocare strutture circolari
- Inoltre, reference counting è abbastanza inefficiente: il suo costo è proporzionale al lavoro complessivo svolto dal programma (e non alla dimensione dell'heap o alla percentuale di esso in uso o meno)

Dipartimento di Matematica e Informatica

Mark and Sweep I

- Il metodo mark and sweep prende il nome da come sono implementate le due fasi astratte
- **Mark**: per riconoscere qualcosa che è inutilizzato, vengono analizzati tutti gli oggetti nell'heap e vegnono marcati tutti come "non in uso". Successivamente, iniziando dai puntatori che sono attivi e presenti sullo stack (root set), si attraversano ricorsivamente tutte le strutture presenti nell'heap (ricerca solitamente depth-first) ed ogni oggetto incontrato viene marcato come "in uso"
- **Sweep**: l'heap viene scansionato e tutti i blocchi "in uso" rimangono immutati mentre quelli "non in uso" sono restituiti alla lista libera
- Questo collector non è incrementale
- Viene invocato solamente quando la memoria libera disponibile nell'heap è quasi terminata
- L'utilizzatore del programma può notare una notevole degradazione del tempo di risposta, aspettando che il garbage collector finisca le operazioni
- Tre principali difetti

Mark and Sweep II

- Come per reference counting, è asintoticamente causa di frammentazione esterna: oggetti vivi e non più in uso sono mischiati arbitrariamente nell'heap, situazione che può rendere difficile l'allocazione di grandi oggetti
- Efficienza: richiede un tempo proporzionale alla lunghezza totale dell'heap, indipendentemente dalla percentuale utilizzata e dallo spazio libero
- Località dei riferimenti: esistono oggetti contigui di età differenti, il che diminuisce drasticamente la località dei riferimenti, con degrado delle prestazioni in sistemi con memoria gerarchica

Dipartimento di Matematica e Informatica

Implementazione degli Oggetti I

- Un'operazione è eseguita inviando un messaggio all'oggetto che consiste nel nome del metodo da eseguire assieme ai parametri
- Terminologia Smalltalk. C++ esprime la stessa cosa tramite: chiamata del campo funzione di un oggetto
- Invocare un metodo coinvolge (o può coinvolgere) la selezione dinamica del metodo da eseguire, mentre l'accesso ai campi dato è statico
- Il livello di opacità dell'oggetto è definito alla creazione dell'oggetto
- I dati possono essere più o meno accessibili dall'esterno, alcune operazioni possono essere visibili ovunque, altre visibili solo ad alcuni oggetti, mentre altre completamente private, visibili solamente all'interno dell'oggetto stesso
- I linguaggi Object Oriented mettono a disposizione meccanismi per organizzare gli oggetti, che permettono di raggruppare quelli con la stessa struttura o simile
- Anche se concettualmente è permesso immaginare che ogni oggetto contenga tutti i suoi dati e metodi, questo sarebbe un enorme spreco

Dipartimento di Matematica e Informatica

Implementazione degli Oggetti II

- In un programma vengono utilizzati molti oggetti che si differenziano solo per il valore dei dati, non per la struttura né per i metodi
- Il codice di ogni metodo dovrebbe essere memorizzato una sola volta, invece di essere copiato all'interno di ogni oggetto con struttura simile
- Il principio di organizzazione più noto è quello delle classi, sebbene esista un'intera famiglia di linguaggi orientati agli oggetti che non dispongono di classi

Dipartimento di Matematica e Informatica

Classi I

- Possiamo assumere che il codice dei metodi sia salvato una volta solamente nella classe
- Quando gli oggetti devono eseguire un metodo specifico, il codice deve essere recuperato dalla classe della quale l'oggetto è istanza: per fare ciò, il codice del metodo deve accedere correttamente alle variabili di istanza che sono differenti per ogni oggetto e non sono quindi salvate insieme alla classe ma all'interno dell'istanza
- Quando un oggetto riceve un messaggio che richiede l'esecuzione di un metodo, l'indirizzo dell'oggetto stesso è un parametro implicito del metodo
- Quando nel corpo del metodo si fa riferimento alle variabili di istanza, c'è un implicito riferimento all'oggetto che sta attualmente eseguendo il metodo
- Ogni variabile di istanza è accessibile attraverso il suo offset dall'inizio dell'area di memoria che contiene l'oggetto
- L'oggetto corrente è solitamente indicato con un nome particolare, tipicamente `self` o `this`
- Alcuni linguaggi permettono di associare variabili e metodi alle classi (e non alle loro istanze): questi sono detti variabili o metodi di classe, o **statici**

Classi II

- Le variabili statiche sono memorizzate con la classe e i metodi statici non possono fare riferimento a `this` nel corpo perché non c'è un oggetto corrente

Oggetti nell'Heap o nello Stack

- Ogni linguaggio orientato agli oggetti permette la creazione dinamica di oggetti. Dove questi oggetti vengono creati dipende dal linguaggio
- La soluzione più comune consiste nell'allocare oggetti nell'heap e accedere a questi tramite riferimenti (che saranno puntatori se il linguaggio lo permette, o saranno variabili se il linguaggio ha scelto un modello a riferimento)
- Alcuni linguaggi permettono l'allocazione e la deallocazione esplicita di oggetti nell'heap (come il C++)
- Altri, probabilmente la maggior parte, sfruttano un garbage collector
- La possibilità di creare oggetti sullo stack come variabili ordinarie non è molto comune (C++ lo permette)

Selezione Dinamica dei Metodi I

- Un metodo definito per un oggetto può essere **ridefinito** (**overridden**) in oggetti che appartengono a sottotipi dell'oggetto originale
- Quindi quando un metodo `m` viene invocato su un oggetto `o` ci possono essere molteplici versioni di `m` per `o`
- La selezione di quale implementazione per `m` utilizzare in un'invocazione `o.m(parametri);` è una funzione del tipo dell'oggetto che riceve il messaggio, non del tipo del riferimento (o nome) di quell'oggetto (che è un'informazione statica)
- Un compilatore non riesce ad intuire il tipo dell'oggetto a cui il metodo è inviato, da cui la dinamicità di questo meccanismo
- Analogie tra overloading e selezione dinamica dei metodi: il problema è lo stesso, risolvere una situazione ambigua in cui un singolo nome può avere più significati
- Nell'overloading, l'ambiguità è risolta staticamente, in base al tipo dei nomi coinvolti
- Nella selezione dei metodi la soluzione al problema avviene a tempo di esecuzione sfruttando il tipo dinamico dell'oggetto e non il suo nome

Selezione Dinamica dei Metodi II

- La selezione dei metodi può essere considerata come un'operazione di overloading a runtime nel quale l'oggetto che riceve il messaggio è considerato il primo argomento (implicito) del metodo di cui deve essere risolto il nome

Implementazione degli Oggetti

- Un oggetto può essere rappresentato come un record, con tanti campi quante sono le variabili nella classe della quale è istanza, in aggiunta a quelle delle superclassi
- La rappresentazione contiene anche un puntatore al descrittore della classe di cui è istanza
- Un linguaggio ha un sistema di tipi statico se il controllo dei vincoli sui tipi può essere effettuato a tempo di compilazione direttamente sul codice. In caso contrario, ha un sistema di tipi dinamico (se il controllo avviene a tempo di esecuzione)
- Per un linguaggio con un sistema di tipi statico, la rappresentazione di un oggetto come record permette una semplice implementazione della compatiblità di sottotipo (nel caso di ereditarietà singola)
- Per ogni variabile di istanza viene registrato l'offset dall'inizio del record
- Se si accede ad un'istanza di un oggetto tramite riferimento (statico) ad una superclasse, il controllo statico dei tipi assicura che l'accesso possa essere effettuato solamente al campo della superclasse, che è allocato nella parte iniziale del record

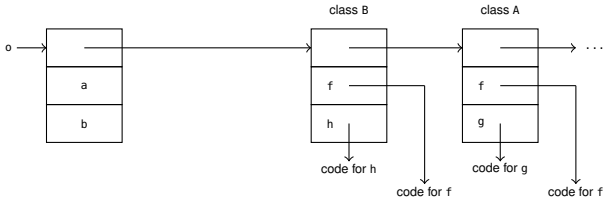
Rappresentazione delle Classi

- La più semplice ed intuitiva implementazione è quella che rappresenta la gerarchia delle classi utilizzando una lista collegata
- Ogni elemento rappresenta una classe e contiene (i puntatori alla) implementazione dei metodi che sono definiti o ridefiniti nella classe
- Gli elementi sono collegati utilizzando un puntatore che parte dalla sottoclasse e va alla superclasse immediata
- Quando viene invocato un metodo `m` su un oggetto `o` istanza di una classe `C`, il puntatore salvato in `o` è utilizzato per accedere al descrittore di `C` e per determinare se contenga un'implementazione di `m`. Se non la contiene, il puntatore viene impostato alla superclasse e la procedura continua

Dipartimento di Matematica e Informatica

Implementazione dell'Ereditarietà

```
class A{
  int a;
  void f(){...}
  void g(){...}}
class B extending A{
  int b;
  void f(){...}  //redefined
  void h(){...}}
B o = new B();
```



Esecuzione di un Metodo

- Un metodo viene eseguito in maniera simile ad una funzione
- Viene inserito nello stack un record di attivazione per le variabili locali del metodo, per i parametri e per tutte le altre informazioni
- A differenza delle funzioni, un metodo deve anche accedere alle variabili di istanza dell'oggetto sul quale è invocato, che non è noto a tempo di compilazione
- Tuttavia, la struttura dell'oggetto è nota (dipende dalla classe) e quindi l'offset di ogni variabile di istanza all'interno della rappresentazione dell'oggetto è staticamente noto (ciò dipende anche dal linguaggio adottato)
- Quando viene invocato un metodo viene anche passato come parametro un puntatore all'oggetto sul quale è stato invocato il metodo
- Durante l'esecuzione del corpo del metodo, questo puntatore è il `this` del metodo
- Durante l'esecuzione del metodo, ogni accesso ad una variabile d'istanza utilizza l'offset da questo puntatore
- La macchina astratta solitamente mantiene il valore corrente di `this` in un registro

Ereditarietà Singola

- Se assumiamo che il linguaggio abbia un sistema di tipi statico, l'implementazione utilizzando liste collegata può essere sostituita da un'altra soluzione più efficiente nella quale la selezione di un metodo richiede tempo costante
- Se i tipi sono statici, l'insieme dei metodi che ogni oggetto può invocare è noto a tempo di compilazione
- La lista di questi metodi è mantenuto nel descrittore della classe
- La lista contiene, oltre ai metodi che sono esplicitamente definiti o ridefiniti nella classe, anche tutti i metodi ereditati dalle superclassi
- Seguendo la terminologia C++, utilizziamo il temine **vtable** (virtual function table) per riferirci a questa struttura

Dipartimento di Matematica e Informatica

Vtable I

- Ogni definizione di una classe ha una vtable e tutte le istanze della stessa classe condividono la stessa vtable
- Quando viene definita una sottoclasse `B` della classe `A`, la vtable per `B` viene generata copiando quella per `A` e sostituendo tutti i metodi ridefiniti in `B` ed aggiungendo i nuovi metodi definiti da `B` in coda
- Se `B` è una sottoclasse di `A`, la vtable di `B` contiene una copia di quella di `A` come parte iniziale, con opportune modifiche ai metodi ridefiniti
- In questo modo, l'invocazione dei metodi richiede solamente due accessi indiretti poiché l'offset di ogni metodo nella vtable è noto in maniera statica
- Risulta possibile accedere ad un oggetto con un riferimento che appartiene staticamente ad una superclasse
- Quando un metodo `f` viene invocato, il compilatore calcola un offset che rimane lo stesso indipendentemente dal fatto che `f` sia invocato su un oggetto di una classe o una delle sue superclassi
- Implementazioni diverse di `f` definite in due classi si trovano allo stesso offset nelle vtable delle due classi

Vtable per Ereditarietà Singola

```
class A{
  int a;
  char c;
  void g(){...}
  void f(){...}}
class B extending A{
  int a;
  int b;
  void h(){...}
  void f(){...}}//redefined
B pb = new B();
A pa = new A();
A aa = pb;
aa.f();
```