Linguaggi di programmazione
Appunti

Giovanni Palma e Alex Basta

Contents

Chapter 1

Nomi e Ambiente

Nell'evoluzione dei linguaggi di programmazione, i *nomi* hanno avuto un ruolo fondamentale nella sempre maggiore astrazione rispetto al linguaggio macchina.

Definition 1.0.1: Nome

I nomi sono solo una sequenza (significativa o meno) di caratteri che sono usati per rappresentare un oggetto, che puo' essere uno spazio di memoria se vogliamo etichettare dei dati, o un insieme di comandi nel caso di una funzione.

1.1 Nomi e Oggetti denotabili

Spesso, i nomi sono *identificatori*, ovvero token alfanumerici, ma possono essere usati anche simboli (+,-,...). E' importante ricordare che il nome e l'oggetto denotato non sono la stessa cosa, infatti un oggetto puo' avere diversi nomi (*aliasing*) e lo stesso nome puo' essere attribuito a diversi oggetti in momenti diversi (*attivazione* e *deattivazione*).

Definition 1.1.1: Oggetti denotabili

Sono gli oggetti a cui e' possibile attribuire un nome.

Note:

Non centra con la programmazione ad oggetti

Possono essere:

- Predefiniti: tipi e operazioni primitivi, ...
- Definibili dall'utente: variabili, procedure, ...

Quindi il legame fra nome e oggetto (chiamato **binding**) puo' avvenire in momenti diversi:

- Statico: prima dell'esecuzione del programma
- Dinamico: durante l'esecuzoine del programma

1.2 Ambienti e Blocchi

Non tutti i legami fra nomi e oggetti vengono creati all'inizio del programma restando immutati fino alla fine. Per capire come i binding si comportano, occorre introdurre il concetto di *ambiente*:

Definition 1.2.1: Ambiente

Insieme di associazioni nome/oggetto denotabile che esistono a runtime in un punto specifico del programma ad un momento specifico durante l'esecuzione.

Solitamente nell'ambiente non vengono considerati i legami predefiniti dal linguaggio, ma solo quelli creati dal programmatore utilizzando le *dichiarazioni*, costrutti che permettono di aggiungere un nuovo binding nell'ambiente corrente.
Notare che e' possibile che nomi diversi possano denotare lo stesso oggetto. Questo fenomeno e' detto *aliasing* e succede spesso quando si lavora con puntatori.

1.2.1 Blocchi

Tutti i linguaggi di programmazione importanti al giorno d'oggi utilizzano i *blocchi*, strutture introdotte da ALGOL 60 che servono per strutturare e organizzare l'ambiente:

Definition 1.2.2: Blocco

Pezzo contiguo del programma delimitato da un inizio e una fine che puo' contenere dichiarazioni **locali** a quella regione.

Puo' essere:

- In-line (o anonimo): puo' apparire in generale in qualunque punto nel programma e non corrisponde a una procedura.
- Associato a una procedura

Permettono di strutturare e riutilizzare il codice, oltre a ottimizzare l'occupazione di memoria e rendere possibile la ricorsione.

1.2.2 Tipi di Ambiente

Un'altro meccanismo importante che forniscono i blocchi e' il loro *annidameno*, ovvero l'inclusione di un blocco all'interno di un altro (non la sovrapposizione parziale). In questo caso, se i nomi locali del blocco esterno sono presenti nell'ambiente del blocco interno, si dice che i nomi sono *visibili*. Le regole che determinano se un nome e' visibile o meno a un blocco si chiamano *regole di visibilita'* e sono in generale:

- Un nome locale di un blocco e' visibile a esso e a tutti i blocchi annidati.
- Se in un blocco annidato viene creata una nuova dichiarazione con lo stesso nome, questa ridefinizione *nasconde* quella precedente.

Definition 1.2.3: Ambiente associato a un blocco

L'ambiente di un blocco e' diviso in:

- **locale**: associazioni create all'ingresso nel blocco:
  - variabili locali
  - parametri formali (nel caso di un blocco associato a una procedura)
- **non locale**: associazioni ereditate da altri blocchi (senza considerare il blocco globale), che quindi non sono state dichiarate nel blocco corrente
- **globale**: associazioni definite nel blocco globale (visibile a tutti gli altri blocchi)

1.2.3 Operazioni sull'ambiente

- Creazione: dichiarazione locale, in cui introduco nell'ambiente locale una nuova associazione
- Riferimento: uso di un nome di un oggetto denotato
- Disattivazione/Riattivazione: quando viene ridefinito un certo nome, all'interno del blocco viene disattivato. Quando esco dal blocco riattivo la definizione originale
- Distruzione: le associazioni locali del blocco dal quale si esce vengono distrutte

Note:

Creazione e distruzione di un *oggetto denotato* non coincide necessariamente con la creazione o distruzione sull'associazione tra il nome e l'oggetto stesso, per essere più precisi nemmeno la vita dell'oggetto e del legame è la stessa. Verrà quindi mostrato nel dettaglio

1.2.4 Vita di un oggetto

Definition 1.2.4: Vita

Si definisce **tempo di vita** o **lifetime** di un oggetto o legame il tempo che interre tra la sua creazione e la sua distruzione

Per comprendere meglio questo concetto, i seguenti notino gli *eventi fondamentali*

Creazione di un oggetto

Creazione di un legame per l'oggetto

Riferimento all'oggetto, tramite il legame

Disattivazione di un legame

Riattivazione di un legame

Distruzione di un legame

Distruzione di un oggetto

Dal punto 1 e 7 è *la vita dell'oggetto*, mentre dall'evento 2 al 6 è *la vita dell'associazione*

Note:

È pertanto vero, quindi, che la vita di un oggetto non coincide con la vita dei legami per quell'oggetto

Esistono 2 modi per categorizzare il tempo di vita di un legame/associazione:

- Vita dell'oggetto più **lunga** di quella del legame

  Si consideri questo codice

```
1    program ExampleCode;
2
3    procedure P(var X: integer); begin {...} end;
4    {...}
5    var A:integer;
6    {...}
7
8    P(A); {chiamata a P con A}
```

  Nel codice dato, inizialmente il nome A viene associato a un oggetto (un valore intero). Quando si chiama la procedura P(A), l'argomento A viene passato per riferimento, il che significa che all'interno della procedura non viene creato un nuovo oggetto, ma semplicemente un nuovo nome per lo stesso oggetto: X.

  Durante l'esecuzione della procedura, X e A sono quindi due nomi che fanno riferimento allo stesso valore in memoria. Qualsiasi modifica apportata a X all'interno della procedura si riflette direttamente su A.

  Una volta terminata l'esecuzione della procedura, il legame tra X e l'oggetto viene distrutto, mentre A continua a riferirsi allo stesso valore, eventualmente modificato dalla procedura. Questo è un classico esempio in cui la durata del legame tra un nome (X) e un oggetto è più breve della vita dell'oggetto stesso

- Vita dell'oggetto più **breve** di quella del legame

  Si consideri questo codice, piuttosto nasty in C:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *X, *Y;
    X = (int *) malloc(sizeof(int));
    Y = X;
    free(X);
    X = NULL;
    return 0;

}
```

Nel codice illustrato vengono creati due puntatori. L'oggetto puntato da `X`, attraverso il comando `malloc`, punta a un'area di memoria allocata dinamicamente. Di conseguenza, assegnando `Y = X`, anche `Y` farà riferimento allo stesso oggetto puntato da `X`.

Col comando `free(X)`, l'oggetto alla fine della catena viene deallocato, ovvero la memoria precedentemente allocata viene liberata. Successivamente, l'istruzione `X = NULL` imposta `X` a `NULL`, indicando che non punta più a un'area valida di memoria.

Tuttavia, il puntatore `Y` continua a riferirsi all'oggetto che è stato deallocato. Questo crea un *dangling pointer* (puntatore pendente), poiché il legame tra `Y` e l'oggetto non esiste più in modo sicuro. Accedere a `Y` dopo la deallocazione può portare a comportamenti indefiniti e DA EVITARE CAZZO

1.3 Regole di scope

Innanzi tutto fornirò la definizione di scope

Definition 1.3.1: Scope

Lo **scope** (o **ambito**) è un concetto semantico che determina in quali porzioni di un programma una variabile o un nome è visibile e utilizzabile. Le regole di scope stabiliscono come i riferimenti ai nomi vengono risolti all'interno di un determinato contesto di esecuzione, garantendo che l'uso delle variabili sia coerente e prevedibile.

Come detto in precedenza una dichiarazione locale ad un blocco è visibile in quel blocco e in tutti i blocchi in esso annidati, a meno ché intervenga in tali blocchi una nuova dichiarazione dello stesso nome che nasconderà quello precedente (shadowing)
Occorre tuttavia determinare come interpretare le regole di visibilità di una variabile in presenza di porzioni di blocchi eseguiti in posizioni diverse dalle loro definizioni e in presenza di ambienti non locali... nasty vero?
Vi sono due filosofie principali

- **Statico**: Basato sul testo del programma
- **Dinamico**: Basato sul flow di esecuzione

Prima di andare avanti, si noti la seguente annotazione

Note:

Entrambe gli appricci differiscono solo in presenza congiunta di ambiente non locale e non globale e procedura

Vabbuò, è normale non capirci un catso solo a parole, si consideri il seguente testo:

```
A:{int x = 0;
  void pippo(int n){
    x = n+1;
  }
  pippo(3);
  write(x);
  {
```

```
8         int x = 0;
9         pippo(3);
10        write(x);
11      }
12      write(x);
13    }
```

Che cosa caspita scriveremo a riga 10? Ebbene dipenderà dal tipo di regola di scope, o statica o dinamica
Di seguito sono riportati nel dettaglio

1.3.1 Scope statico

Definition 1.3.2: Scope statico

La regola dello **scope statico** (o **regola dello scope annidato più vicino**) si basa sui seguenti principi:

1. **Ambiente locale di un blocco**: Le dichiarazioni all'interno di un blocco definiscono il suo ambiente locale. Questo include solo le dichiarazioni presenti direttamente nel blocco stesso e non quelle eventualmente presenti nei blocchi annidati al suo interno.
2. **Ricerca delle associazioni di un nome**: Se un nome viene utilizzato all'interno di un blocco, si segue questa gerarchia per determinare quale dichiarazione è valida:
   - Se esiste una dichiarazione del nome nel **blocco locale**, questa è quella valida.
   - Se il nome non è dichiarato nel blocco locale, si cerca nel **blocco immediatamente contenitore**.
   - Se il nome non è ancora trovato, si continua a risalire nei blocchi contenitori fino al più esterno.
   - Se il nome non è dichiarato nemmeno nel blocco più esterno, si cerca nell'**ambiente predefinito del linguaggio**.
   - Se il nome non è presente neanche nell'ambiente predefinito, si genera un errore.
3. **Blocchi con nome**: Un blocco può essere **assegnato a un nome**, e in questo caso tale nome diventa parte dell'ambiente locale del **blocco immediatamente contenitore**. Questo vale anche per i blocchi associati a procedure o funzioni.

Molto più semplicemente si può dire che

Note:

Un nome non locale è risolto nel blocco che *testualmente* lo racchiude

Pertanto nel codice d'esempio nel primo `write(x)` verrà stampato 4, nel secondo 0 e nel terzo 4, in quanto la `x` che la funzione `pippo(3)` modifica è quella dichiarata all'interno del blocco che lo racchiude, in questo caso A. Nel blocco B non verrà modificata la `x` racchiusa nello stesso quindi si stamperà, quindi 0.
Si ha quindi una forte indipendenza dalla posizione della posizione da parte dei nomi. Ad esempio se si dichiara una funzione all'interno di un blocco, il corpo della procedura si riferirà sempre alle regole di scope medesime del blocco in cui è stata dichiarata, pertanto dovunque la funzione verrà chiamata lo scope a cui riferisce sarà sempre lo stesso
Tra i vantaggi dello scope statico troviamo una maggiore comprensione per il programmatore, ogni nome può essere collegato alla sua dichiarazione semplicemente analizzando la struttura del codice, senza dover simulare l'esecuzione e la facilità di analisi del programma da parte del compilatore che può determinare tutte le occorrenze di un nome e fare controlli di correttezza sui tipi di dati ed eseguire ottimizzazioni del codice prima dell'esecuzione

1.3.2 Scope dinamico

Definition 1.3.3: Regole di scope dinamico

Secondo le regole di scope dinamico, l'ssociazione valida per un nome `x` ad un punto `P` del programma è la più recente (in senso temporale) associazione creata per `x` ancora attiva appena il controllo di flusso arriva a `P`

In pratica occorre andare indietro *nell'esececuzione* per cercare l'occerrenza d'interresse (è l'ultima che è stata introdotta) blocco attivato per ultimo (che deve essere ancora attivo), come riassunto in questa nota quindi:

Note:

Un nome non locale è risolto nel blocco attivato più di recente e non ancora disattivato

Quando un nome non è dichiarato localmente in un blocco, viene cercato nel blocco attivato più recentemente che lo contiene. Questo significa che la risoluzione dei nomi segue una logica *LIFO (Last In First Out)*, cioè una gestione a stack basata sull'ordine di chiamata delle funzioni. Questo approccio è più semplice da gestire a runtime perché si basa solo sulla pila di attivazione, senza necessità di altre strutture dati

1.4 determinare l'ambiente

L'ambiente è quindi determinato da:

- Regole di scope (statico o dinamico)
- Regole di visibilità
- Regole di binding (intervengono quando una procedura P è passata come parametro ad un'altra procedura mediante il formale X)
- Regole per il passaggio di parametri

Chapter 2

Gestione Memoria

Prima di discutere su i diversi modi in cui la memoria (RAM e HD solo se necessario) puo' essere gestita dal compilatore di un linguaggio, e' importante identificare cosa esattamente deve essere contenuto all'interno di essa. Sicuramente, ogni *dato* che deve essere salvato durante l'esecuzione del programma dovra' avere un posto nella memoria, come ad esempio le variabili, ma ci sono anche informazioni per il *controllo dell'esecuzione* che necessitano di essere memorizzati.
La vita di un oggetto corrisponde con tre meccanismi di allocazione di memoria:

- **Allocazione statica**: l'oggetto viene allocato una volta sola, prima dell'inizio dell'esecuzione del programma, e deallocato alla fine dell'esecuzione,

  pertanto è una memoria allocata a tempo di compilazione

- **Allocazione automatica**: l'oggetto viene allocato all'entrata di un blocco (tipicamente una funzione) e deallocato all'uscita del blocco.

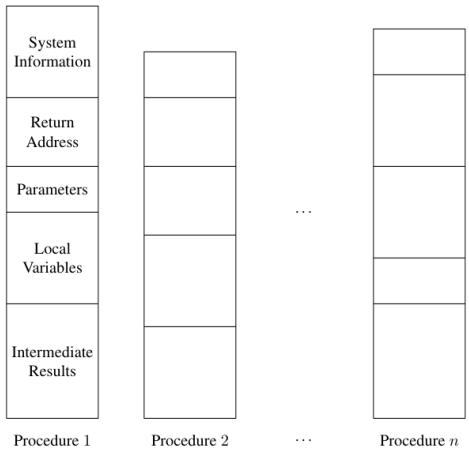
- **Allocazione dinamica**: l'oggetto viene allocato e deallocato esplicitamente dal programmatore tramite chiamate a funzioni di allocazione e deallocazione (ad esempio, `malloc` e `free` in C). *pertanto è una memoria allocata a tempo di esecuzione*

  Questo tipo di allocazione di serve di due aree di memoria:

  - pila (stack): gli oggetti sono allocati con una politica LIFO, utilizzato per le variabili locali e i parametri formali delle funzioni
  - heap: gli oggetti sono allocati e deallocati in qualsiasi ordine, utilizzato per gli oggetti dinamici (puntatori)

2.1 Allocazione Statica

Non sarebbe possibile utilizzare solo una memoria con allocazione statica? Sicuramente per le variabili *globali*, *costanti* che non dipendono da altri valori non noti inizialmente e anche per le *tabelle* che utilizza il compiler, l'allocazione statica funziona benissimo.
Per quanto riguarda i sottoprogrammi, si puo' pensare di allocare per ognuna di esse tutto lo spazio necessario per i parametri e le variabili locali (e altre informazioni necessarie), dato che possiamo determinare tutto cio' prima di eseguire il programma:



Questo metodo funziona solo se siamo sicuri che, se una procedura e' attiva, allora non puo' chiamare la stessa procedura ricorsivamente. Questo e' perche' esiste solo un'istanza per ogni procedura allocata in memoria e non e' possibile sapere quante volte una funzione verra' chiamata ricorsivamente. Quindi, se vogliamo implementare la ricorsione, serve l'allocazione *dinamica* (stack di chiamate):

2.2 Allocazione Dinamica

2.2.1 Allocazione Dinamica con Pila

Ogni istanza di sottoprogramma viene memorizzata con un *frame* (o *record di attivazione*) che contiene tutte le informazioni necessarie. Quando un'istanza viene attivata, il relativo frame viene messo in cima a una **pila**, la struttura dati naturale in quanto se una procedura A chiama una procedura B, allora siamo sicuri che B deve terminare prima che A possa continuare l'esecuzione e terminare anch'esso.

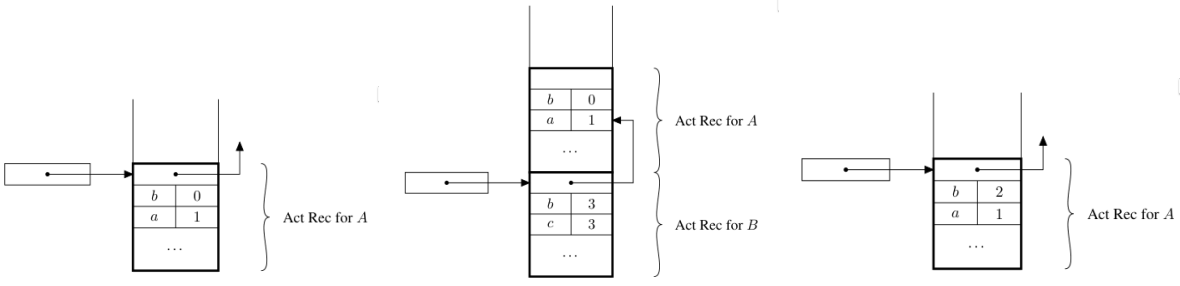
Note:

L'allocazione dinamica e' utile anche quando non c'e' ricorsione come meccanismo per risparmiare memoria.

Vediamo un esempio:

```
A:{
  int a = 1;
  int b = 0;

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



Vediamo piu' in dettaglio cosa viene memorizzato nei record di attivazione:

Record di attivazione

Per un semplice blocco anonimo, il corrispondente frame ha tale forma:

Dynamic chain pointer
Local variables
Intermediate results

Note:

Nella realta' la maggior parte dei linguaggi usa l'allocazione statica per blocchi anonimi per maggiore efficenza di calcolo (sacrificando pero' l'efficenza di memoria).

Mentre per le procedure e' un po' piu' complesso:

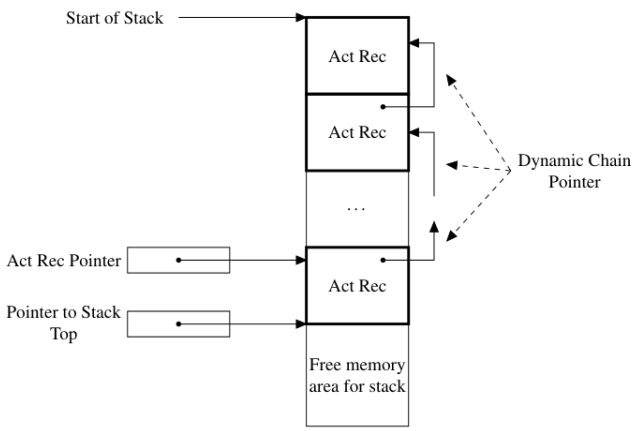
Dynamic Chain Pointer
Static Chain Pointer
Return Address
Address for Result
Parameters
Local Variables
Intermediate Results

Vediamo in dettaglio cosa sono tutti sti dati:

- **Intermediate Results**: serve per memorizzare risultati intermedi di equazioni complicate e per risultati di chiamate ricorsive.
- **Local Variables** e **Parameters**: per le variabili locali e parametri.
- **Dynamic Chain Pointer**: puntatore all'ultimo RdA creato sulla stack, l'insieme di tutti i puntatori dinamici e' chiamata *catena dinamica*.
- **Static Chain Pointer**: informazione necessaria per implementare lo scope statico, vedremo piu' avanti.
- **Return Address**: indirizzo di memoria della prima istruzione da eseguire quando termina la procedura corrente.
- **Address for Result**: indirizzo dove puo' essere salvato il valore di ritorno della funzione (sara' un indirizzo interno al frame della funzione chiamante).

Gestione della Pila

Vediamo la struttura di un sistema a pila (discendente):



Come puo' notare dall'immagine, ci sono due puntatori esterni che puntano a posti specifici sull'ultimo record di attivazione:

- **Pointer al RdA**: e' un puntatore ad un luogo predeterminato all'interno del frame usato come base da cui si puo' calcolare l'offset per accedere alle variabili locali. Questo offset e' determinabile staticamente dal compilatore (ad eccezzione del caso di variaili di dimensione variabile).
- **Stack Top Pointer**: come si puo' indovinare, e' il puntatore al primo indirizzo libero di memoria dopo l'ultimo RdA. Puo' essere omesso se e' possibile calcolare lo stesso indirizzo partendo dal pointer al RdA.

Il funzionamento corretto della pila di RdA e' dato dalla collaborazione fra il chiamante e il blocco chiamato, che eseguono dei blocchi di codice inseriti dal compilatore (o interprete) prima e dopo chiamate a procedure e blocchi anonimi:

- **Sequenza di Chiamata**: eseguito dal chiamante subito prima della chiamata
- **Prologo**: eseguito immediatamente all'inizio del blocco chiamato
- **Epilogo**: eseguito alla fine del blocco
- **Sequenza di Ritorno**: eseguito dal chiamante immediatamente dopo la chiamata

Al momento della chiamata, la Sequenza di Chiamata e il Prologo devono:

- **Modificare PC**
- **Allocare spazio sulla pila**
- **Modifica pointer RdA**
- **Passaggio parametri**
- **Memorizzare registri**

Quado il blocco o processo chiamato termina, l'Epilogo e la Sequenza di Ritorno devono:

- **Modificare PC**
- **Ritornare Valori**
- **Recuperare Registri**
- **Deallocare spazio sulla pila**

Note:

Sono stati omessi meccanismi per l'implementazione delle regole di scope. Vedremo queste piu' avanti.

2.2.2 Allocazione Dinamica con Heap

Nel caso in cui vogliamo dare la possibilita' a chi usa il linguaggio di allocare esplicitamente memoria a run-time o di usare oggetti di dimensioni variabili sorge il seguente problema: la vita degli oggetti non e' per forza LIFO, ovvero un oggetto creato prima di un altro puo' essere rimosso dalla memoria prima di un'altro oggetto creato dopo, come nel seguente blocco di codice:

```
1  int *p, *q;
2  p = malloc(sizeof(int));
3  q = malloc(sizeof(int));
4  *p = 0;
5  *q = 1;
6  free(p);
7  free(q);
```

Dobbiamo usare quindi la *heap*, ovvero una regione di memoria i cui blocchi possono essere allocati/deallocati in momenti arbitrari.

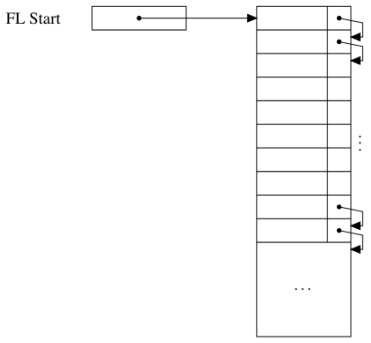
Note:

La heap che abbiamo appena definito non centra niente con la struttura dati usata per la "heap sort".

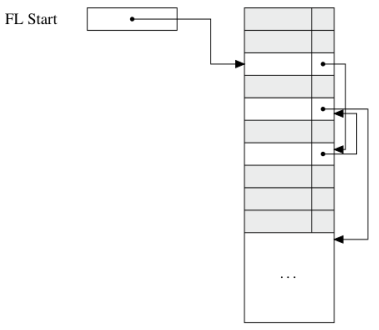
Esistono due categorie principali di metodi di gestione della heap a seconda della lunghezza dei blocchi che memorizza, che possono essere di *dimensione fissa* o *variabile*.

Dimensione fissa

La heap viene suddivisa in blocchi di dimensione fissa abbastanza limitata che sono inizialmente collegati tutti assieme nella *lista libera*:



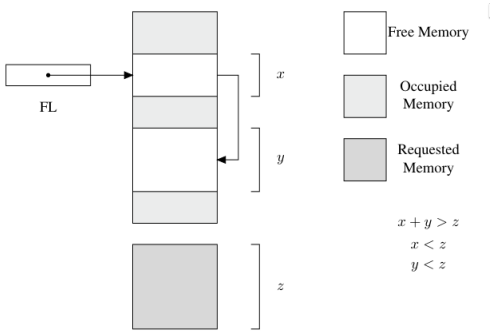
A run-time, quando viene richiesto un blocco di memoria, il primo elemento della lista libera viene rimosso e restituito al processo che ha richiesto la memoria, mentre la testa della lista libera si sposta al prossimo elemento della lista. Vediamo un esempio di heap a dimensione fissa dopo alcune operazioni di allocazione/deallocazione (i blocchi grigi sono in uso):



Dimensione variabile

Nel caso volessimo poter allocare array la cui dimensione e' determinata solo a runtime, la soluzione a dimensione fissa non sarebbe adeguata, dato che l'array puo' essere di dimensioni maggiori rispetto ai blocchi fissi e non si possono allocare piu' blocchi perche' la memoria deve essere per forza contigua.
In questi casi e' necessario poter richiedere un blocco di dimensione arbitraria. In un sistema di questo tipo, bisogna prestare attenzione ad usare *operazioni efficenti* e a limitare lo *spreco di memoria* che puo' avvenire in due situazioni:

- **Frammentazione interna**: viene richiesto un blocco di dimensione n ma ne viene restituito uno di dimensione $k > n$, quindi $k - n$ parole si perdono.
- **Frammentazione esterna**: lo spazio totale nella memoria sarebbe abbastanza per soddisfare una richista di dimensione n, ma i blocchi liberi sono separati o non contigui.



Esistono due meccanismi diversi per gestire blocchi di dimensione variabile:

- **Unica lista**:

  Inizialmente, la lista libera e' composta da un solo blocco che occupa tutta la heap. Quando viene fatta la richiesta per un blocco di dimensione n, le prime n parole dell'heap vengono allocate e l'inizio della lista si sposta di n. Si va avanti cosi' finche' la memoria rimasta dall'inizio della lista libera alla fine della heap non e' abbastanza per soddisfare la richiesta. In queslto caso dobbiamo riutilizzare memoria deallocata, che puo' essere fatto in due modi:

  - **Uso diretto della lista libera**: si scorre lungo la lista finche' si trova un blocco di dimensioni $k > n$. Se la differenza fra la grandezza del blocco e della memoria effettivamente usata e' maggiore di una tolleranza, allora il blocco viene diviso ed il blocco di dimensione $k - n$ viene reinserito nella lista. Possono essere usate due politiche di ricerca diverse per scegliere quale blocco prendere: *first fit* e *best fit*. Quando un blocco viene deallocato, si guarda se blocchi adiacenti sono anch'essi liberi e in caso affermativo si fondono per formare un blocco piu' grande (*compattazione parziale*).
  - **Compattazione della memoria libera**: tutti i blocchi attivi vengono spostati alla fine della heap, molto efficente ma funziona solo quando possiamo spostare la memoria.

- **Liste libere Multiple**:

  Per ridurre il costo operativo di cercare un blocco di dimensione arbitraria, e' possibile utilizzare diverse liste per diverse dimensioni di blocchi. Quando viene richiesto un blocco di dimensione n, si scorrono le liste finche' una che contiene blocchi di dimensioni adeguate non e' vuota. Anche in questo caso e' possibile riudurre la dimensione dei blocchi per ridurre la frammentazione interna, esistono due metodi:

  - **Buddy system**: la dimensione dei blocchi aumenta per potenze di 2. Si calcola l'intero minore k tale che $2^k \geqslant n$ e si controlla se la relativa lista ha blocchi liberi. Altrimenti, si va a cercare nella lista $k+1$ e si divide il blocco in due blocchi da 2^k (la dimensione che volevamo), uno viene allocato e l'altro viene spostato nella lista corretta. Quando viene deallocato, la meta' cerca il suo compagno nella lista libera e se lo trova si uniscono e tornano nella lista originale.
  - **Fibonacci**: funzionamento equivalente ma le dimensioni dei blocchi seguono la sequenza di Fibonacci, che sale piu' lentamente quindi porta a meno frammentazione ma piu' tempo.

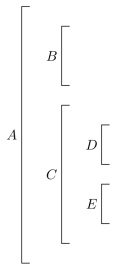
TODO: disegni esplicativi

2.3 Implementazione delle Regole di Scope

Ora che abbiamo capito come viene gestita la memoria di un programma dal compilatore, soprattuto per quanto riguarda i RdA, vediamo come possiamo implementare le regole di scope quando un blocco deve accedere a variabili non-locali.

2.3.1 Scope Statico

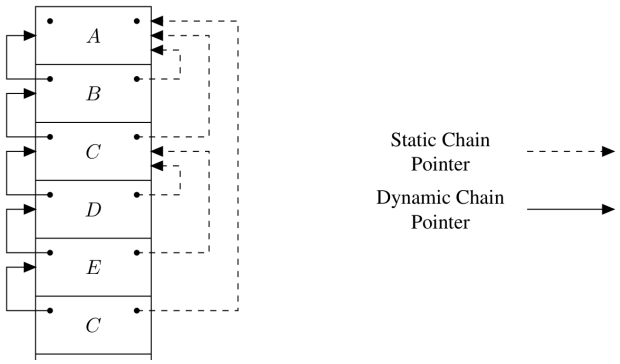
Se vengono implementate le regole di scope statico, allora l'ordine da seguire quando stiamo cercando il valore di una variabile non-locale non e' necessariamente quello dettato dalla catena dinamica, ovvero l'ordine delle chiamate. Infatti, l'RdA corretto e' determinato dalla struttura statica (testuale) del programma, seguendo quindi l'ordine di annidamento dei blocchi. Vediamo un esempio:



Data la stuttura soprastante, immaginiamo di chiamare in ordine A, B, C, D, E, C e che rimangano tutte attive:

- Aggiungiamo inizialmente l'RdA di A allo stack di sistema, aggiornando lo SP e l'RdA pointer. Dato che e' il primo sulla pila, non ha nessun link.
- Man mano che aggiungo i RdA di $B, C, ...$ aggiorno sempre SP e RdA pointer, ma e' anche necessario determinare il link dinamico e statico.

Quindi con tutte le chiamate aperte la situazione e' questa:



Come al solito, il puntatore di catena dinamica punta all'RdA *temporalmente precedente* (quello appena sotto nella pila), mentre il puntatore di catena **statica** indica l'RdA del blocco che *contiene strutturalmente* quello in cui siamo.

Note:

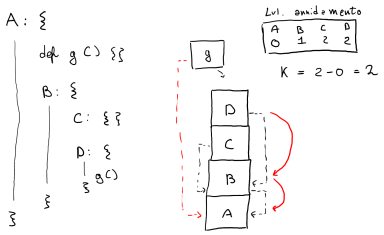
Se un sottoprogramma e' annidato a livello k, allora la catena statica sara' lunga k.

Supponiamo di essere in E e di voler accedere alla variabile x in modo statico dichiarata in A:

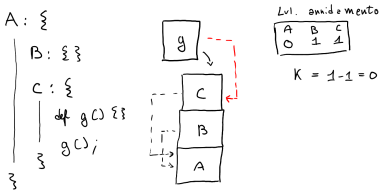
- Seguendo la catena statica, controllo se x e' dichiarata in C.
- Non c'e', quindi continuo a seguire i link statici e arrivo in A, dove trovo il valore cercato.

Il supporto a run-time della catena statica e' compito della sequenza di chiamata, prologo e l'epilogo che abbiamo visto prima per le chiamate. L'approccio piu' comune e' quello dove il chiamante calcola il puntatore a catena statica che passa poi al chiamato, e' abbastanza semplice e puo' essere diviso in due casi:

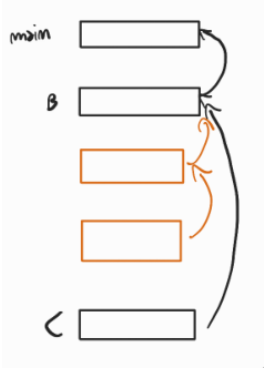
- **Il chiamato e' esterno al chiamante**: secondo le regole di visibilita', questa situazione e' possibile solo se il chiamante e' annidato internamente rispetto al blocco del chiamato. Cio' significa che tale blocco e' ancora attivo sulla pila, come tutti i blocchi annidati fino a quello del chiamante. Dato che sappiamo il livello di annidamento di ciascun blocco (puo' essere calcolato staticamente), basta trovare la differenza fra il livello del chiamante meno il chiamato e fare questo numero di salti sulla catena statica del chiamante (che e' in cima alla stack). In questo modo ci troviamo nel RdA del blocco contenente la definizione del chiamato, e ci basta solo passare l'indirizzo di tale RdA al chiamato che lo imposta come link statico.



- **Il chiamato e' interno al chiamante**: tale situazione puo' succedere solo se il chiamato e' definito nello stesso blocco dove viene chiamato. Siamo quindi in un caso speciale del punto precedente dove la distanza di annidamento e' 0, quindi basta passare come link statico al chiamato il puntatore RdA del record contenente la chiamata (che sara' quello in cima).



Il numero di "salti" da fare per raggiungere il RdA corretto e' calcolabile staticamente dal compiler usando la *tabella dei simboli*, che memorizza anche il livello di annidamento al quale e' stata dichiarata la variabile cercata. Pero' non e' possibile stabilire staticamente la locazione esatta di memoria dove si trovera' tale variabile non-locale, dato che, pur sapendo il numero di "salti", non sappiamo quanti e quali RdA si trovano fra ogni salto a run-time. Per questo motivo ci serve la catena statica.



Il Display

E' un po' uno schifo dover seguire sta catena statica, dato che se siamo a un livello k di annidamento, dobbiamo per forza eseguire k accessi a memoria per arrivare finalmente al frame giusto. Anche se nella pratica generalmente non e' un problema, possiamo ottimizzarlo quindi lo facciamo.
TODO: Fai Display, CRT e A-List

Chapter 3

Strutturare il controllo - espressioni, comandi, iterazione, ricorsione

3.1 Espressioni

Come al solito prima fornisco la definizione

Definition 3.1.1: Espressione

si definizisce **espressione** un'entità sintattica la cui valutazione produce un valore oppure non termina, nel qual caso l'espressione è indefinita

3.1.1 sintassi delle espressioni

In generale, un'espressione è composta da una singola entità (costante, variabile, ecc...) o da da un operatore (`+`, `cons`, ecc...) applicato ad una serie di argomenti anch'essi espressioni. Si tenga presenta che la sintassi di un'espressione può essere descritta da una grammatica libera e può essere rappresentata da un albero di derivazione che ne esprime anche la semantica. Con Gabrielli (e non Morbidelli) verranno utilizzate le notazioni lineari per scrivere le espressioni quindi non dovrò stare qui a disegnare alberi o automi su latex grazie a dio. Queste notazioni differiscono tra loro per il modo in cui rappresentano l'applicazione (quindi la semantica) di un operatore ai suoi operanti. Possiamo distinguere tre tipi principali di notazione:

- **infissa**: il simbolo dell'operatore è posto in mezzo a due espressioni, es. `a+b`.

  Sintassi ambigua, e richiede l'utilizzo di parentesi e regole di precedenza per la disambiguazione. Quasi tutti i linguaggi di programmazione insitono sulla notazione inflissa, ma spesso questa è solo uno *zucchero sintattico* per rendere il codice più digeribile a chi lo legge. In C++ l'espressione `a+b` è un'abbreviazione di `a.operator+(b)`

- **prefissa (polaccaa**[1]**)**: il simbolo dell'operatore è posto prima a due espressioni, es. `+ab`

  Intuitivamente è la sintassi delle funzioni (`f(a,b)` o `+(a,b)`) e non richede parantesi o regole di precedenza in quanto l'arietà di ogni operatore è conosciuta. Inoltre non c'è ambiguita su quale operatore applicare ad ogni operando peché è sempre quello che precede immediatamente gli operandi. es. `* + a b + c d`

- **postfissa (polacca inversa)**: l'operatore è posto dopo le espressioni, es. `ab+`

Un vantaggio delle due notazioni polacche rispetto a quella infix è che possono essere utilizzate in modo uniforme per rappresentare operatori con qualsiasi numero di operandi. Nella notazione infix, invece, rappresentare operatori con più di due operandi significa dover introdurre operatori ausiliari. Un secondo vantaggio, come abbiamo già detto, è che possiamo omettere completamente le parentesi. Un ultimo vantaggio della notazione polacca è che rende particolarmente semplice la valutazione di un'espressione, che adesso vediamo

[1] In onore di W. Łukasiewicz, il brodo che l'ha utilizzato estensivamente

3.1.2 Semantica delle espressioni

Adesso verrà esposta la semantica delle tre notazioni lineari sintattiche prima presentate

notazione infissa

Quando utilizziamo la notazione infissa paghiamo per la semplicità di lettura con una maggiore complicazione nel meccanismo di valutazione. Ecco qui presentati i vari problemucci:

- **Precedenza** fra gli operatori:

  Se le parentesi non sono utilizzate sistematicamente (o altri tipi di *zucchero sintattico*) è necessario specificare la precedenza di ogni operatore. I linguaggi di programmazione, quindi, impiegano delle *regole di precedenza* per definire una gerarchia tra l'ordine di valutazione e i vari operatori

  Esempio:

```
a+ b * c ** d ** e / f \\??
if A < B and C < D then \\??
```

  Che fare prima?

- **Associatività**:

  Un altro problema che nella valutazione di espressioni che concerne gli operatori, se infatti noi scriviamo `15-5-3` potremmo intendere sia `(15-5)-3` o anche `15-(5-3)`. In questo caso la normale convenzione matematica impone che la prima espressione sia quella corretta e che l'operatore `-` associ da `destra a sinistra`. Non è sempre ovvio, in APL `15-5-3` è interpretato come `15-(5-3)`!!!! CAZZO

Si può quindi concludere che valutazione di un'espressione infissa non è semplice, andiamo dai polacchi vah, che è mejo

notazione postfissa

La valutazione è molto più seplice di quella inffissa:

- non servono regole di precedenza
- non servono regole di associatività
- non servono le parentesi

Infatti questa notazione prevede una semplice strategia di valutazione che percorre l'espressione da sinistra a destra usando una pila per memorizzare i vari componenti. Si può quindi desceivere l'algoritmo di valutazione in questo modo:

1. Leggi il prossimo simbolo nell'espressione e puschalo nella pila
2. Se il simbolo appena letto è un operatore applicalo agli operandi immediamente prima nello stack, memorizza il risulato in R, e fai il `pop` degli operatori e opeeandi dalla pila e pusha il valore in `R`
3. Se la sequenza da leggere non è vuota torna a (1)
4. Se il simbolo letto un operando torna a (1).

Valutazione prefissa

Un po' più complessa di quella postfissa, qui mostrato l'algoritmo: L'algoritmo di valutazione è descritto dai seguenti passaggi, dove utilizziamo uno stack ordinario (con le operazioni push e pop) e un contatore C per memorizzare il numero di operandi richiesti dall'ultimo operatore letto:

1. Leggi un simbolo dall'espressione e inseriscilo nello stack;
2. Se il simbolo appena letto è un operatore, inizializza il contatore C con il numero di argomenti dell'operatore e vai al passaggio 1;

3. Se il simbolo appena letto è un operando, decrementa C;
4. Se $C \neq 0$, vai a 1;
5. Se $C = 0$, esegui le seguenti operazioni:
   (a) Applica l'ultimo operatore memorizzato nello stack agli operandi appena inseriti nello stack, memorizzando i risultati in un registro R, elimina operatore e operandi dallo stack e memorizza il valore di R nello stack;
   (b) Se non ci sono simboli di operatore nello stack, vai a 6;
   (c) Inizializza il contatore C a $n - m$, dove n è il numero di argomenti dell'operatore in cima allo stack e m è il numero di operandi presenti nello stack sopra questo operatore;
   (d) Vai a 4;
6. Se la sequenza rimanente da leggere non è vuota, vai a 1;

Chapter 4

Astrazione sul controllo: sottoprogrammi ed eccezioni

A cosa servono gli array? Il linguaggio assembly non ce li ha, ma riesce comunque a svolgere tutto quello che possono fare. Sono quindi un'*astrazione* che rende la vita dei programmatori piu' facile. Questo e' l'obbiettivo dei linguaggi di programmazione di alto livello, che astrae sul controllo e sui dati.
L'astrazione, quindi, consiste nell'indentificare proprietà importanti di cosa si vuole descrivere, concentrarsi su quelle e ignorare le altre. Che cosa ignorare e cosa no dipende dallo scopo del progetto
I linguaggi di programmazione (oltre al fatto che essi stessi sono astrazioni più sono ad alto livello) forniscono strumenti per implementare astrazioni e modelli astratti, questi sono chiamati, appunto, **astrazioni sul controllo e sui dati**. Adesso diamo una definizionzioncina

Definition 4.0.1: astrazione sul controllo

Si definisce **astrazione del controllo** una serie di istruzioni per svolgere un compito a prescindere dal contesto in cui questo opera, specificandone modalità e fine

Queste astrazioni sono ad esempio funzioni o blocchi. Tra le proprietà più importanti di questi costrutti è che ogni componente fornisce servizi al suo ambiente, nascondendo i dettagli interni necessari a produrlo
Un meccanismo fondamentale con cui si può definire come ogni sottoprogramma (funzione) comunica con il col programma principale `main` è attraverso i parametri o un'ambiente globale (da preferire i primi perché quest'ultimo rende nulla l'astrazione)

4.1 parametri

Introduciamo due definizioni terminologiche:

Definition 4.1.1: Parametro formale

Un parametro formale è una variabile utilizzata nella definizione di un sottoprogramma, che viene sostituita dal valore o riferimento del parametro attuale quando il sottoprogramma viene chiamato.

Pertanto si trova nella dichiarazione/definizione: `int f (int n)return n+1;`

Definition 4.1.2: Parametro attuale

Un parametro attuale è il valore o riferimento passato a un sottoprogramma quando viene chiamato. Questo valore o riferimento sostituisce il parametro formale nella definizione del sottoprogramma.

Ad esempio, nell'espressione `f(5)`, il valore `5` è il parametro attuale che sostituisce il parametro formale `n` nella definizione del sottoprogramma `f`.

Definition 4.1.3: Pragramtica

La pragramtica rappresenta il flusso di informazioni tra chiamante e chaiamto

Rappresentiamo il chiamate con `main` e il chiamato `proc`, questi sono i possibili di informazioni tra i comuniscanti:

- `main`→`proc`, es. `x=f(y+1)`
- `proc`→`main`, es. `procedure Uno (var y:integer); begin y:=1 end;`
- `proc`↔`main`, es. `procedure Succ (var y:integer); begin y:=y+1 end;`

Example 4.1.1 (Una funzione comunica col chiamante)

Valore restituito

```
int f()return 1;
```

4.1.1 Modalità di passaggio dei parametri

Vi sono due modi principali per passare i parametri ai sottoprogrammi:

- **Passaggio per valore**: In questo caso, il valore del parametro attuale viene copiato nel parametro formale del sottoprogramma. Le modifiche apportate al parametro formale all'interno del sottoprogramma non influenzano il parametro attuale.

  La pragramtica è `main` → `proc`

  Example 4.1.2 (Esempio di passaggio per valore)

  ```
  void increment(int n) { n = n + 1; }
  int main() { int x = 5; increment(x); }
  ```

  In questo esempio, il valore di `x` non cambia dopo la chiamata a `increment`.

- **Passaggio per riferimento**: In questo caso, il parametro formale del sottoprogramma diventa un riferimento al parametro attuale. Le modifiche apportate al parametro formale all'interno del sottoprogramma influenzano direttamente il parametro attuale

  La pragramtica è `main` ↔ `proc`

  Example 4.1.3 (Esempio di passaggio per riferimento)

  ```
  void increment(int &n) { n = n + 1; }
  int main() { int x = 5; increment(x); }
  ```

  In questo esempio, il valore di `x` cambia a 6 dopo la chiamata a `increment`.

Passaggio per valore

Si prenda come esempio il seguente codice

```
1  void foo (int x) { x = x+1; }
2  {...}
3  y = 1;
4  foo(y+1);
```

In questo caso il parametro formale `x`, mentre quello attuale è `y`. Alla chiamata di `foo`, viene valutato `y+1` e assegnato al formalex. Ovviamente non vi sarà nessun legame tra `x` e `y` e alla fine della computazione `x` verrà deallocata e distrutta.
Tuttavia si ha che nel record di attivazione di `foo`, appena viene eseguita la funzione, vine ecreata una copia di `y` per `x`. La cosa è influente dato che `x` è un intero, ma potrebbe essere estremamente costoso per parametri attuali di grandi dimensioni, si pensi ad un array di 1000 elementi

Passaggio per riferimento

Si prenda come esempio il seguente codice

```
    void foo (reference int x){ x = x+1;}
    ...
    y = 1;
    foo(y);
```

In questo caso, nella funzione viene passato un riferimento, ovvero un puntatore a un intero. Pertanto, qualsiasi modifica apportata al parametro formale `x` all'interno della funzione `foo` influenzerà direttamente il parametro attuale `y`. Tra la variabile `x` e `y` si verifica il cosidetto *aliasing* alla stessa cella
Questo metodo è efficiente in termini di memoria, poiché non viene creata una copia del parametro attuale. Tuttavia, bisogna fare attenzione alle modifiche non intenzionali ai parametri attuali, poiché queste possono portare a effetti collaterali indesiderati

Passaggio per costante

Il passaggio per costante è una variante del passaggio per valore, tuttavia viene garantito che nella procedura non è permessa la modifica del parametro formale

```
void foo (final int x){
  // qui x non puo' essere modificato
}
```

Se l'oggetto passato e' di grandi dimensioni, il compiler puo' evitare di fare la copia usando il passaggio a riferimento, mantenendo sempre la semantica del passaggio per valore.

Passaggio per Risultato

```
    void foo (result int x) {x = 8;}
    ...
    y = 1;
    foo(y);
```

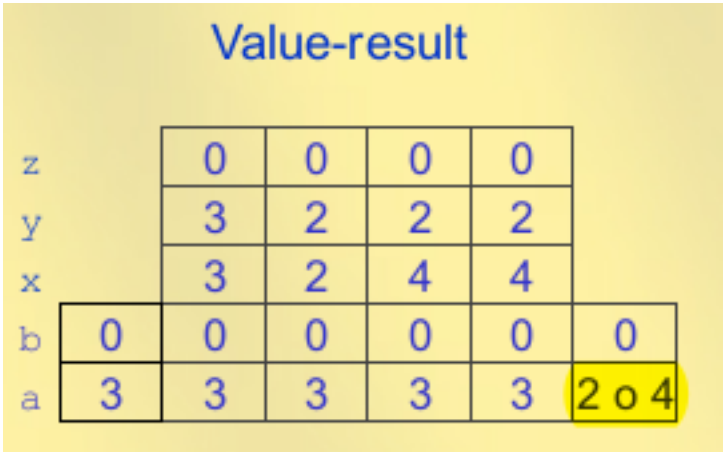
Il passaggio per risultato è la tecnica *complementare* al passaggio per valore. In questo caso, il parametro formale viene utilizzato per restituire un valore al termine dell'esecuzione del sottoprogramma
Al momento della chiamata e della computazione (all'interno del corpo di `foo`) non vi sarà alcun legame tra `x` e `y`, ma al ritorno di `foo` verrà fatta una copia di del formale sull'attuale `y=x` La pragramtica sarà dunque: `proc` $\rightarrow$ `main`

Passaggio per valore risultato

È un mix tra il passaggio per risultato e per valore, pertanto verrà fatta una copia dall'attuale a formale all'inizio e una copia dal formale all'attuale alla fine e, dato che non vi è alcun riferimento, non vi è alcun legame tra il formale e l'attuale durante la computazione dei dati nel corpo della funzione
Pragmaticamente: `main`$\leftrightarrow$`proc`
Si noti che il passaggio valore-risultato $\neq$ riferimento, ad esempio in questo codice:

```
    void foo (int x, int y, int z) {
        x = 2;
        y = 2;
        x = 4;
        if (x == y) z = 1;
    }
    int a = 3;
    int b = 0;
    foo(a,a,b);
```

Per valore risultato il valore delle variabili istruzione per istruzione è



Mentre per riferimento

Morale

Il passaggio per valore ha come pro:

- Semantica semplice: il corpo non ha necessità di conoscere come la procedura verrà chiamatà
- trasparenza referenziale: ovvero la proprietà di un'espressione che garantisce che verrà restituito sempre lo stesso risultato ogni qualvalto il gli verrà fornito lo stesso imput indipendentemente dal contesto
- implementazione abbastanza semplice

e come contro un costo potenzialmente alto per la copia del parametro attuale.
Invece il passaggio per riferimento ha come pro:

- implementazione semplice
- efficienza nel passaggio da parametro attuale a formale

E come contro una semantica complessa a causa dell'aliasing

Passaggio per nome

Il passaggio per nome è una modalità di passaggio parametri introdotta da Algol negli anni 60 che vede la chiamata come una *macro espansione* ovvero un meccanismo dove la semantica di una chiamata di funzione è definita in modo **prescrittivo** e consiste nell'esecuzione del corpo come se fosse testualmente presente nel chiamante, anche la gestione dei parametri segue lo stesso principio, ovvero il corpo della procedura viene eseguito dopo che i parametri attuali sono stati sostituiti testualmente al posto dei parametri formali, quindi quest'ultimi vengono valutati dopo questo passaggio.

Più in particolare il passaggio per nome segue la cosiddetta *regola della copia*: una chiamata alla procedura P è la stessa cosa che eseguire il corpo di P dopo aver sostituito i parametri attuali al posto dei parametri formali.
Si noti il seguente codice:

```
1  int y;
2  void foo (name int x) {x= x + 1; }
3  ...
4  y = 1;
5  foo(y);
```

In questo caso dopo la chiamata `foo(y)` viene inserito il corpo della funzione nel `main` cambiando il parametro formale con quello attuale, quindi `y=y+1`. Tuttavia questa regola per come è definita nasconde una criticità, infatti se nel corpo della funzione è presente un nome `y` potrebbe provocare un conflitto, si noti questo codice:

```
1  int y;
2  void fie (name int x){
3  int y;
4  x = x + 1; y = 0;
5  }
6  ...
7  y = 1;
8  fie(y);
```

Dopo la della chiamata verrà eseguita la macro espansione sostituendo il parametro formale con quello attuale:

```
1  int y;
2  y = y+1;
3  y =0;
```

Tuttavia all'interno di `fie` vi erano due variabili erano diverse che abbiamo reso uguali perché il nome di `y` all'interno di `fie` è uguale al parametro attuale, provocando così un conflitto se si esegue il programma con scope statico.
Per ovviare a questo problema viene passato, ai vari paramatri, una coppia `<exp, amb>` detta *closure*, dove:

- `exp`: è il parametro attuale (testo, non valutato)
- `amb`: è l'ambiente di valutazione (in scoping statico)

Si prenda come esempio il seguente codice:

```
1  int y;
2  void fie (int x ){
3      int y;
4      x = x + 1; y = 0;
5  }
6  ...
7  y = 1;
8  fie(y);
```

Quando viene eseguita la macro-espansione si avra che `x→<y, int y;>` dove l'ambiente non è altro che la dichiarazione della variabile in riga 1, gg. Si ha, quindi, che *ogni volta che il formale è usato, exp è valutata in amb*
La pragmatica in questo è `main↔proc`, inoltre si tratta di una pratica costosa infatti

- **vi è il passaggio di una strutta complessa** (soprattutto l'ambiente)
- **è prevista una rivalutazione ripetuta del parametro**, infatti differenza del passaggio per valore, in cui il parametro viene valutato una sola volta prima di entrare nella funzione, nel passaggio per nome il parametro viene rivalutato ogni volta che viene usato nel corpo della funzione

Per implementare il passaggio per nome, la coppia closure è formata, dal lato pratico, da:

- un puntatore al testo di `exp`
- un puntatore di catena statica (sullo stack) al record di attivazione del blocco di chiamata

4.2 Funzioni di ordine superiore

Alcuni linguaggi di programmazione consenstono di passare passare **funzioni come argomenti di altre funzioni** o **restituire funzioni come risultato di una funzione**:

Definition 4.2.1: Funzione di ordine superiore

Funzione che accetta una funzione come parametro o che restituisce una funzione come risultato.

è quindi lecito chiedersi come viene gestito lo scope in questi casi

4.2.1 Funzione come argomento

Vediamo un esempio:

```
A:{
  int x = 1;
  int f (int y){
    return x+y; // Quale "x"?
  }
  void g(int h(int b)){
    int x = 2;
    return h(3) + x;
  }
  ...
  B:{
    int x = 4;
    int z = g(f)
  }
}
```

Notare che f utilizza una variabile non-locale x, che e' stata dichiarata piu' volte: nel blocco A, B e nella funzione g. Dobbiamo quindi capire in quale ambiente risolverla, proviamo ad applicare le regole di scope che abbiamo visto fin'ora:

- se lo scope è statico: si usa `x` definita nel punto in cui `f` è stata dichiarata `(x=1)`
- Se lo scope è dinamico: si potrebbe usare sia la `x` della chiamata `x=4`, sia la `x` di `g` (`x=2`)

L'incertezza su quale ambiente non-locale scegliere e' dovuta dal fatto che e' possibile immaginare due implementazioni diverse della chiamata di una funzione passata come parametro (f) tramite il parametro formale (h):

- La chiamata della funzione passata avviene nel blocco della funzione di ordine superiore (la chiamata h(3) chiama la funzione f all'interno di g)
- La chiamata della funzione passata avviane nel blocco dove viene creato il legame fra parametro formale e attuale (la chiamata h(3) chiama la funzione f all'interno del blocco B)

Quindi notiamo che le regole di scope non sono abbastanza specifiche per determinare l'ambiente da utilizzare, dato che non e' chiaro da quale blocco si vuole che avvenga la chiamata di f (in questo caso lo scope statico ha solo un'opzione, ma vedremo che non e' sempre cosi'). Ci servono quindi delle ulteriori regole specifiche alle funzioni parametro, dette regole di *binding*:

Definition 4.2.2: Binding

Data una funzione di ordine superiore g che ammette una funzione parametro f, in base al blocco dal quale vogliamo che f venga chiamato quando chiamiamo il parametro formale associato h, abbiamo due casi:

- *Deep binding*, se la chiamata avviene nel blocco attivo nel momento in cui e' stato instaurato il legame fra parametro formale e attuale.
- *Shallow binding*, se la chiamata avviene nel blocco attivo nel momento in qui viene chiamato il parametro formale.

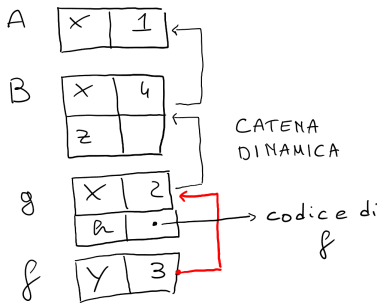
Quindi, applicando cio' all'esempio di prima:

- Usando deep binding, si simula la chiamata di f dal blocco B, quindi con scope statico x = 1 e con scope dinamico x = 4.
- Usando shallow binding, f viene chiamata dalla funzione g, quindi se si usa scope statico abbiamo comunque x = 1, ma con scope dinamico x = 2.

Vediamo in specifico l'implementazione delle due regole di binding applicate in linguaggi con scope dinamico e statico:

Binding con scope dinamico

Consideriamo prima il caso di un linguaggio con scope dinamico e shallow binding. Per definizione di shallow binding, vogliamo chiamare la funzione parametro f all'interno del corpo della funzione g. Per le regole dello scope dinamico, dobbiamo risolvere le variabili non-locali nell'ambiente immediatamente precedente alla chiamata, che quindi corrisponde all'ultimo record di attivazione sulla pila (quello di g). In questo caso quindi basta la normale implementazione di scope dinamico.

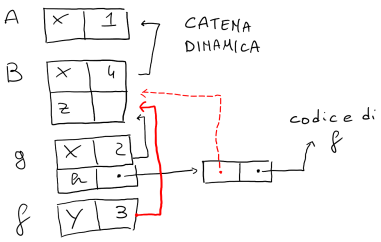


Note:

Ovviamemte puo' anche essere implementato lo scope dinamico con A-list o CRT.

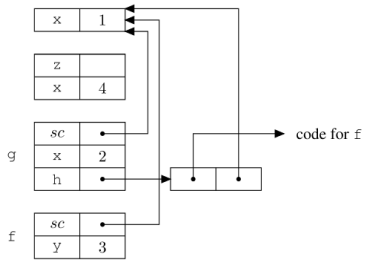
Nel caso di deep binding con scope dinamico, ci serve l'ambiente del blocco attivo al momento del legame fra la funzione e il parametro formale. Ma in questo caso, l'RdA di questo blocco (che nell' esempio e' B) non sara' direttamente sotto a quello della funzione f quando viene chiamata, dato che ci sara' sicuramente almeno il frame della funzione di ordine superiore. Dobbiamo quindi passare un puntatore a questo ambiente insieme alla funzione. Tale struttura viene chiamata *closure*, ovvero l'accoppiata ⟨`f, amb`⟩, dove:

- `f` è il puntatore alla funzione che vogliamo passare
- `amb` è il puntatore all'ambiente non-locale in cui è da valutare il corpo della funzione f



Binding con scope statico

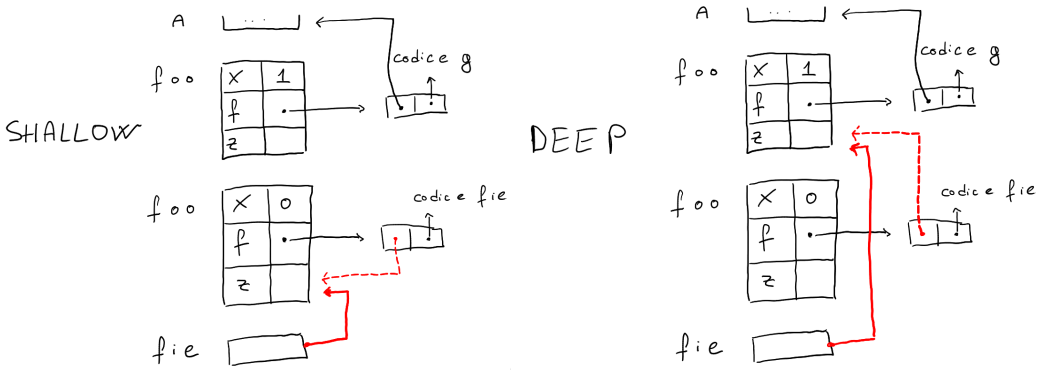
Anche nel caso generale dello scope statico e' necessario usare una closure per poter passare alla funzione di ordine superiore anche il puntatore di catena statica, che puo' essere calcolata al momento del legame fra funzione e parametro formale, come abbiamo visto studiando l'implementazione dello scope statico (2.3.1).



Puo' sembrare che le regole di scope statico siano rigide e che non servano le regole di binding per distinguere vari casi. In effetti per la maggior parte delle situazioni e' cosi, ma ci possono essere discussioni per quanto riguarda un caso specifico con funzioni ricorsive. Vediamo un esempio:

```
A:{
  void foo(int f(), int x){
    int fie(){
      return x;
    }
    int z;
    if (x==0) z = f();
    else foo(fie, 0);
  }
  int g(){
    return 1;
  }
  foo(g,1);
}
```

Il problema in questo caso e' che la funzione fie e' definita in due istanze diverse della funzione foo, alle quali sono associati due ambienti diversi: nel primo x = 1, nel secondo x = 0. Dato che possono essere considerate valide entrambi per le regole di scope, bisogna utilizzare le regole di binding. Usando shallow binding, consideriamo la definizione piu' vicina temporalmente; usando deep binding, scegliamo l'ambiente attivo al momento della prima definizione di fie.



Chapter 5

Esercitazioni

5.1 Scope

5.1.1 Esercizio 1

Testo

```
{
    int x =2;
    int func ( int y){
        x = x+y;
        write(x);
    }
    {
        int x = 5;
        func(x);
        write(x);
    }
    write(x);
}
```

- Si descriva il comportamento del programma assumendo uno scope statico.
- Si descriva il comportamento del programma assumendo uno scope dinamico.

Soluzione

- **Scope statico**: il programma stampa 7, 5, 7
- **Scope dinamico**: il programma stampa 10, 10, 2

5.1.2 Esercizio 2

Testo

```
{
    int x = 0;
    int next () {
        x = x+1;
        write(x);
    }
    int exec(){
        int x = 3;
        next();
        write(x);
```

```
        }
        exec();
        write(x);
    }
```

- Si descriva il comportamento del programma assumendo uno scope statico
- Si descriva il comportamento del programma assumendo uno scope dinamico

Soluzione

- **Scope statico**: il programma stampa 1, 3, 1
- **Scope dinamico**: il programma stampa 4 infatti `next` prenderà l'istanza di x dell'unltimo ambiente non disattivato, ovvero exec, modifico poi il valore di `x` nell'ambiente exec, 4 dato che l'amiente di exec è stato modificato, 0 dato che `exec` è stato disattivato si ha che `x = 0`

5.1.3 Esercizio 3

Testo

```
{
    int x = 0;
    void pippo(value int y, rif int z){
        z= x+y+z;
    }
    {
        int x = 1;
        int y = 100;
        int z = 30;
        pippo(x++,x); // ricordarsi che il x++ prima passa il valore di x nudo e
            crudo poi come side-effect modica x staticamente in questo caso,
            pertanto il secondo parametro sara' 2: pippo(1, 2) -> 3 = 0+ 1 +2
        pippo(x++, x); // pippo(3, 4) -> 7 = 0 + 3 + 4
        write(x); // 7
    }
    write(x); // 0 perche' il blocco e' finito, terminato e riprende la variabile
        x dichiarata all'inizio
}
```

Si descriva il comportamento del programma assumendo uno scope statico

Soluzione

Il programma stampa 7, 0

5.1.4 Esercizio 4

Testo

```
{
    void pippo(value int y, value int z){
        x=y+z;
    }
    int x = 100;
    pippo(x++, x); // pippo(100, 101) -> 201, quindi x = 201
    pippo(x++, x); // pippo(201, 202) -> 403, quindi x = 403
    write(x); // 403
}
```

Si descriva il comportamento del programma assumendo uno scope dinamico

Soluzione

Il programma stampa 403

5.1.5 Esercizio 5

Testo

```
{
    int f(value int k){ //1. 2, 1. 1
        int g (value int n){
            return n+y //
        };
        int x=10;
        int y=10;
        if k=1 return g(x) + g(y); // 40
        else {
            int x = 30;
            int y= 30;
            return f(k-1);
        }
    }
    int x =50;
    int y=50;
    x= f(2); // x = 40
    write(x); // 40
}
```

si risolva utilizzando uno scope dinamico

Soluzione

Il programma stampa 40

5.1.6 Esercizio 6

Testo

```
{
    int x =10;
    int v =5;
    void B(){
        write(x);
    }
    void A(z){
        int x = z * v;
        B();
    }
    A(??) // si ha che qualsiasi valore di al posto di ?? si scrivera' 10
}
```

Fornire una chiamata alla funzione A di modo che il prgramma, usando scope statico stami il valore 10

```
    {
        int x =10;
        int v =5;
        void B(){
            write(x);
        }
        void A(z){
            int x = z * v;
            B();
```

```
        }
        A(??) // si scrive 2 cazzo
    }
```

fornire una chiamata per stampare il valore 10, ma usando uno scope dinamico

Soluzione

PEr la prima parte si stampera sempre 10, per la seconda parte si passa il parametro 2

5.1.7 Esercizio 7

Testo

```
{
    int x =1;
    int y=2;
    void A(){
        int x = 2;
        int k = 3;
    }
    void B(){
        int x = 5; // 2
        A();
        // ** 1**
        x=2;
        C():
    }
    void C(){
        int z =5;
        // ** 2 **
    }
    B();
}
```

SI descriva qual è il contenuto delle variabili attive al punto **1** e **2**, utilizzando scope statico e dinamico

Soluzione

- **Scope statico**:
  - **Punto 1**: `x = 5`, `y = 2`
  - **Punto 2**: `x = 1`, `y = 2`, `z = 5`
- **Scope dinamico**:
  - **Punto 1**: `x = 5`: nel momento in cui si attiva `A()` lo shadowing impone che che x in memoria è 2, ma appena A termina si "disattiva" e si riprende x=5 definito nel punto B, `y = 2`
  - **Punto 2**: `x = 2`, `y = 2`, `z = 5`

5.2 sottoprogrammi

5.2.1 Es. 1

Testo

Considerare un linguaggio che ammette il passaggio per nome. Dire cosa stampra il codice:

```
int k = 2;
int A[5];
A = {1, 2, 4, 7, 3};
void swap(int name x, int name y) {
    int temp = x; // temp = 2
    x = y; // x = A[k] -> x = A[2] -> x = 4
    y = temp; //A[4] = 2
    write(A); // A = {1, 2, 4, 7, 4};
}
swap(k, A[k]);
```

svolgimento

è probabile che scriverà `A = 1, 2, 4, 7, 4;`

5.2.2 Es. 2

Testo

L'esecuzione del seguente frammento di codice su una certa implementazione risulta nella stampa dei valori 4 e 10.

```
int W[10];
int x = 4;
for (int i=0; i<10; i++) W[i]=i; // quindi, [0,1,2,3,4,5,6,7,8,9]

void foo(int x; int y){
    x = x+1;
    y=10;
}

foo (x, W[x])
write (W[4]) // 4
write (W[5]) // 10
```

Si fornisca una possibile spiegazione.

svolgimento

Una possibile soluzione è che entrambi i paramenti siano passati per nome, infatti se andiamo a valutare i parametri dopo la macro espansione tutto torna, provare per credere

5.2.3 Es. 3

Testo

La definizione di un certo linguaggio di programmazione specifica che la valutazione procede da sinistra a destra, ma nel valutare un'espressione complessa eventuali sottoespressioni che vi compaiono più di una volta devono essere valutate una sola volta, usando il valore così calcolato anche per le altre occorrenze della stessa sottoespressione. Un assegnamento è una particolare forma di espressione complessa e il passaggio dei parametri avviene per nome. Si consideri il seguente frammento di codice:

```

int x = 1;
int A[5];
for (int i=0; i<5; i++) A[i] = i; // A = {0,1,2,3,4}

void f(int name x, int name y){
    x= y + x + x;
}

f(A[x++],x) // dopo la chiamata avremo:
```

```
    // A[2] = A[2] + 1 +1 (dato che avremo modificato x nello scope del main)
    //Quindi A[2] = 1 + 2 +2
```

Qual'è lo stato del vettore A dopo la chiamata della funzione f?

Soluzione

A = 0,5,2,3,4

5.2.4 Es. 4

Testo

È dato il seguente frammento di codice in uno pseudolinguaggio con goto, scope dinamico e blocchi annidati etichettati (indicati con A :...):

```
A: {
    int x = 5;
    int y = 4;
    goto B;
    B: {
        int x = 4;
        int z = 3;
        goto C;
    }
    C: {int x = 3;
        D: {int x = 2;}
        goto E;
    }
    E: {
        int x = 1; // (***)

    }
}
```

Lo scope dinamico è gestito mediante CRT. Si illustri graficamente la situazione della CRT nel momento in cui l'esecuzione raggiunge il punto segnato con il commento (***)

5.2.5 Es. 5

Testo

Si consideri il seguente schema di codice scritto in uno pseudolinguaggio che usa scope statico e passaggio per riferimento

```
{
    int x = 0;
    int A(reference int y) {
        int x =2;
        y=y+1;
        return B(y)+x;
    }
    int B(reference int y){
        int C(reference int y){
            int x = 3;
            return A(y)+x+y;
        }
        if (y==1) return C(x)+y;
        else return x+y;
    }
write (A(x));
}
```

Supponiamo che lo scope statico sia implementato mediante display. Si dia graficamente la situazione del display e della pila dei record di attivazione al momento in cui il controllo entra per la seconda volta nella funzione A. Per ogni record di attivazione si dia solo il valore del campo destinato a salvare il valore precedente del display

zio pera