

Linguaggi di programmazione

Appunti

Giovanni Palma e Alex Basta

CONTENTS

CHAPTER	NOMI E AMBIENTE	PAGE
1.1	Nomi e Oggetti denotabili	
1.2	Ambienti e Blocchi Blocchi — • Tipi di Ambiente — • Operazioni sull'ambiente — • Vita di un oggetto —	
1.3	Regole di scope Scope statico — • Scope dinamico —	
1.4	determinare l'ambiente	

CHAPTER	GESTIONE MEMORIA	PAGE
---------	------------------	------

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 può 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 può avere diversi nomi (*aliasing*) e lo stesso nome può essere attribuito a diversi oggetti in momenti diversi (*attivazione* e *deattivazione*).

Definition 1.1.1: Oggetti denotabili

Sono gli oggetti a cui è 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**) può avvenire in momenti diversi:

- Statico: prima dell'esecuzione del programma
- Dinamico: durante l'esecuzione 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 *annidamento*, 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 intercorre 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:

```

1      #include <stdio.h>
2      #include <stdlib.h>
3
4      int main() {
5          int *X, *Y;
6          X = (int *) malloc(sizeof(int));
7          Y = X;
8          free(X);
9          X = NULL;
10         return 0;
11     }
12

```

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

Innanzitutto 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 che 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 approcci differiscono solo in presenza congiunta di ambiente non locale e non globale e procedura

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

```

1      A:{int x = 0;
2          void pippo(int n){
3              x = n+1;
4          }
5          pippo(3);
6          write(x);
7      }

```

```

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'associazione 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'esecuzione* per cercare l'occorrenza d'interesse (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

- statica
- dinamica
 - pila
 - heap

A cosa serve la memoria? variabili locali, parametri formali,

Se ho la ricorsione serve della memoria dinamica (stack di chiamate):

se non ammetto la ricorsione, non avro' mai piu' di un frame attivo per ogni funzione, quindi basta allocare staticamente quella memoria (se ci sono 10 funzioni, alloco lo spazio per 10 funzioni)

se ammettiamo ricorsione, questa prima condizione non e' vera, ed il numero di chiamate attive contemporanee e' illimitata. Possiamo quindi usare una allocazione dinamica a pila dato che la funzione chiamante non potra' terminare prima che termini la funzione chiamata.

zio pera