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

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

```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;

}
```

statico:
dinamico: vado indietro *nell'esecuzione* per cercare l'occerrenza ch ci interessa (e' l'ultima che e' stata introdotta) blocco attivato per ultimo (che deve essere ancora attivo)
notare che se cambio il nome di una variabile locale, con scope statico la semantica non cambia, ma con scope dinamico puo' cambiare (esempio p.24) -> collegamento con lo shadowing logica
L'ambiente e' quindi determinato da:

- Regole di scope
- Regole di visibilita'
- Regole di binding (solo quando posso passare funzioni come parametri)
- Regole per il passaggio di parametri

Chapter 2

Gestione Memoria

- sttica
- 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