nomi_scopi_binding

Introduzione

I **linguaggi imperativi** sono astrazioni dell'architettura di Von Neumann, i cui componenti principali sono:

- Memoria, che immagazzina sia le istruzioni che i dati
- Processore, che si occupa di fornire operazioni per la modifica dei contenuti nella memoria
  Altrettanto importanti sono le **variabili**, che vedremo essere astrazioni delle celle di memoria della macchina.
  Una variabile e' caratterizzata da una collezione di attributi, il cui piu' importante e' il **tipo**.

> **Note**
>
> Progettare un tipo e' complesso e ci richiede di considerare diversi aspetti come
>
> - scope
> - ciclo di vita
> - controllo dei tipi
> - inizializzazione
> - compatibilita' dei tipo

Nomi

Il **nome** di una variabile, e' sicuramente uno degli attributi piu' importanti che possiede, ma ci sono alcune complicazioni da considerare:

- I nomi sono case sensitive?
- Le parole chiave del linguaggio sono anche riservate oppure no?

Forme dei nomi

> **Definition**
>
> Un nome non e' altro che una sequenza di caratteri che identifica una variabile

> **Caution**

nella maggior parte dei linguaggi, specialmente i linguaggi C-based, i nomi delle variabili sono **case-sensitive**

Caratteristica fondamentale di un nome, oltre al nome stesso, la sua *lunghezza*: di base, se un nome e' troppo corto non puo' essere evocativo, non ci aiuta a capire a che cosa ci stiamo riferendo.
Vediamo come alcuni linguaggi trattano i nomi delle variaibili:

- C99 e Fortran 90 -> non hanno limiti, ma solo i primi **63** caratteri sono significativi, inoltre i nomim esterni, quelli gestiti dal linker, hanno una lunghezza limite di 31 caratteri
- C# e Java -> non hanno limiti
- C++ -> di per se' non ha limiti, ma gli implementatori possono imporli

Alcuni linguaggi riservano anche dei caratteri speciali per le variabili:

- PHP -> tutti i nomi devono iniziare con il $
- Perl -> tutti i nomi devono iniziare con dei caratteri speciali che indicano anche il tipo della variaibile

Problemi della case-sensitivity

Ci permette di avere a disposizione piu' nomi per le variaibili ma rischiamo di avere incompresioni

Example

input e InPuT possono sembrare la stessa cosa ma per il programma sono due cose completamente diverse, potenzialmente anche con nessuna correlazione fra loro

Quindi come gestiscono questa cosa i vari linguaggi?
Dipende, C lo fa imponendo come convenzione le sole: lettere maiuscole (si guardi le call di posix) ma questa convenzione non e' adottata da tutti i linguaggi C-based (C#, Java), che addirittura usano nomi misti (IndexOutOfBoundException. ParseToInt,...)

Nomi e Parole Speciali

Le parole speciali ci servono per rendere i programmi piu' leggibili dando appunto un nome a certe azioni.
In molti linguaggi vengono classificate come **parole riservate** pertanto non utilizzabili come nomi di variabili e ne tantomeno possono essere ridefinite.

Definizione: parola chiave

Una parola chiave e' una parola che e' speciale solo in certi contesti

Definizione: parola riservata

E' una parola speciale che non puo' essere usata come nome

C'e' un problema con linguaggi che presentano un gran numero di parole riservate: se ce ne sono troppe, l'utente potrebbe avere difficolta' nell'ideare nomi di variabili significativi senza che vadano in collisione con le parole riservate.

Variabili e loro attributi

Una variabile e' **un'astrazione** di una cella di memoria (o di una collezione di celle) della macchina. L'utilizzo di NOMI con l'avvento di assembly fu proprio un grande passo, perche' ha permetto di rimpiazzare riferimenti di memoria assoluti con qualcosa che noi umani siamo in grado di capire meglio, rendendo i programmi *molto piu' leggibili*

Una variabile puo' essere descritta come una sestupla di attributi:

- Nome (ne abbiamo gia' parlato)
- Indirizzo
- Valore
- Tipo
- Lifetime
- Scope

Indirizzo

l'**indirizzo** di una variabile e' l'indirizzo di memoria della macchina a cui la variabile e' associata.
Per quanto la definizione sia banale, l'associazione in realta' non e' cosi' semplice

L'indirizzo di una variabile e' anche chiamato come il suo **l-value**

Example

In molti linguaggi e' possibile che una variabile venga associata a diversi indirizzi in diversi momenti durante l'esecuzione del programma o diversi punti del programma

E' possibile avere diverse variaibile che hanno lo stesso indirizzo, in tal caso vengono chiamati **alias**

> ⚠ **Caution**
>
> Gli alias sono spesso dannosi per la leggibilita' del codice, perche' permette ad una variabile di cambiare valore con un'operazione di assegnamento NON su di lei, ma su uno degli alias

Tipo

Determina l'intervallo di valori delle variabili e l'insieme di operazioni definite per i valori di quel tipo

Valore

> 🔥 **Important**
>
> E' importante pensare a "celle", come celle astratte e non fisiche, in quanto una cella di memoria fisica contiene 1 byte ed e' sicuramente troppo piccolo per la maggior parte dei tipi di variabili nel linguaggi moderni

Il valore di una variabile e' spesso chiamata come il suo **r-value**

Concetto di Binding

> 🖊 **Definition**
>
> Il processo di binding e' un'associazione tra un'entita' e un attributo come:
>
> - Tra una variabile e il suo tipo
> - Tra una variabile e il suo valore
>   Il **momento logico** dove avviene questa operazione di binding e' detto **binding time**.

Diversi tipi di tempo di binding

I possibili binding time sono:

- *Fase di progettazione del linguaggio*

  > **Example**
  > associare i simboli degli operatori alle loro corrispettive operazioni

- *Fase di implementazione del linguaggio*

  > **Example**
  > Associare un certo tipo alla sua corrispettiva rappresentazione (x.y per i numeri a virgola mobile)

- *Fase di compilazione o linking*

  > **Example**
  > associare una certa variabile a un tipo in C o java oppure il binding di una chiamata di funzione al suo codice corrispondente

- *Fase di caricamento*

  > associare una variabile statica ad una cella di memoria

- *Fase di esecuzione*

  > **Example**
  > associare una variabile locale non statica ad una cella di memoria
  > [!CAUTION]
  > Fa parte del binding in fase di esecuzione il binding di chiamate di funzioni virtual al loro codice corrispondente (nei linguaggi ad oggetti che li supportano)
  > Chiariamo il tutto con degli esempi di codice:

```
int status; //binding in fase di loading
int fact(n){
    //TODO
}
int main(){
    fact(3) //binding in fase di linking
}
```

```
public class Father{
    //virtual
    public void display(){}
    public class Son: extends Father{}
}
int main(){
    Father *v;
    v-> display(); //binding di una funzione virtual a run time
}
```

Tipi di binding

Ne esitono di 2 tipi: **STATICO** e **DINAMICO**

> **Definition: statico**
>
> Viene detto statico se avviene per la prima volta PRIMA del tempo di esecuzione e non cambia piu' durante l'esecuzione di un programma

Binding di tipo Statico

In questo caso parliamo di **dichiarazioni**

> **Definition: dichiarazione esplicita**
>
> Una dichiarazione esplicita in un programma e' un istruzione che elenca il nome di una variabile specificando che 'e di un certo tipo

> **Defintion: dichiarazione implicita**
>
> E' un modo diverso di associare semplicemente le variaibili ai loro tipi tramite delle convezioni standard, invece di usare delle vere e proprie dichiarazioni. L'assegnazione tramite standard, e' fatta puramente mediante la forma sintattica del nome della variabile

Il caso piu' eclatante di un linguaggio che utilizza una dichirazione implicita e' sicuramete Perl

```
@ciao = 2 //questo e' un valore scalare
```

Un altro tipo di **dichiarazione implicita** e' quella che usa il contesto della variabile, ed e' detta anche **inferenza di tipo**. In questi casi il contesto non e' altro che il tipo del valore che viene assegnato alla variabile
Da Rust user posso fare questo esempio, in quanto Rust supporta entrambe le dichiarazioni

```
fn main(){
    let s = "Ciao"; //dichiarazione implicita di una stringa tramite inferenza
    let s: String = "Ciao"; //dichiarazione esplicita di una stringa
}
```

Ovviamente questo porta ad una ottima flessibilita' del linguaggio

Definition: dinamico

Viene detto dinamico se avviene per la prima volta DURANTE il tempo di esecuzione del programma OPPURE puo' cambiare durante il corso del programma

Example

- In C il tipo di una variabile e ben specificato durante la sua dichiarazione e non puo' piu' variare, rimane immutabile
- In altri linguaggi come python, il tipo viene *inferito* e inoltre puo' cambiare diverse volte durante il programma

BINDING di tipo Dinamico

Quando viene eseguita un' istruzione di assegnemento, la variabile a cui viene assegnato il valore, viene anche legata al tipo del r-value. In questo modo qualsiasi variabile puo' assumere qualsiasi tipo (non come in Perl) ed e' importante capire anche che il tipo puo' essere temporaneo e cambiare da un momento all'altro.

Binding Statico vs Binding Dinamico

Per decidere tra i due, bisogna pensare al trade-off tra efficienza e flessibilita'

- Statico: il linguaggio fa molto col costo minore possibile, in quanto il piu' possibile viene fatto durante la fase di compilazione

- Dinamico: Si ha un linguaggio piu' flessibile ma ora la maggior parte del lavoro e' fatta in fase di esecuzione

In generale, gli svantaggi del binding dinamico sono:

- Programmi meno affidabili: le capacita' di error-finding del compilatore sono inibite
- Costo maggiore in fase di compilazione a cause di strutture dati aggiuntive
- Costo maggiore in fase di esecuzione per fare type-checking e sono generalmente piu' lenti a causa dell'uso di un interprete

Storage Binding e Lifetime

Definition: storage binding

Lo storage binding non e' altro che **la fase di allocazione in memoria** di una variabile che riguarda la scelta della cella di memoria da "dare" alla variabile a partire da una pool di celle disponibili per l'uso. Questo processo e' appunto detta **allocazione**
La **deallocazione** e' il processo opposto, cioe' piazzare una cella di memoria che non e' piu' utilizzata da una variabile di nuovo dentro la pool di celle libere

Strettamente collegato allo *Storage Binding* abbiamo il **lifetime** delle variabili:

Definition: lifetime

Il lifetime di una variabile e' il tempo durante il quale tale variabile e' effettivamente legata da una cella in memoria

In base ai lifetime delle variaibile, si possono suddividere le varie categorie di storage binding.

Variabili Statiche

Definition: variaibili statiche

E' detta statica se viene legata ad una cella di memora PRIMA che l'esecuzione incominci e tale legame rimane lo stesso fino a quando il programma non termina

Quale potrebbe una possibile applicazione?

Example

Un esempio e' sicuramente quello delle variabili globali. Siccome devono essere disponibili sempre e allo stesso modo e necessario che rimangano sempre legati alla stessa posizione in memoria

Vantaggi e Svantaggi delle variabili statiche

- Vantaggi:
  - Efficienza, perche' e' possibile fare indirizzamento diretto sulle variabili statiche
  - Nessun overhead per allocazione e deallocazione
  - *Permette l'esecuzione di sottoprogrammi sensibili alla cronologia: history-sensitive subprograms*

Question

Cosa sono i sottoprogrammi sensibili alla cronologia?
Per quanto il nome possa essere complesso e fuorviante, in realta' il concetto e' piuttosto semplice.
Succede quando un sottoprogramma utilizza delle variabili statiche che quindi **mantengono il loro valore anche tra chiamate successive**. Guardiamo un esempio banale:

```
int contatore(){
    static int x = 0;
    x = x + 1;
    return x;
}
```

Ad ogni chiamata a *contatore()*, x verra' incrementato perche' allocato staticamente pertanto sempre collegato alla stessa cella di memoria

- Svantaggi:
  - Flessibilita' ridotta
  - Nessun supporto per la ricorsione

Question

Perche' la ricorsione non puo' essere supportata?
Sappiamo che per ogni chiamata ricorsiva che facciamo, viene creata una copia indipendente delle variabili locali e avremo quindi diverse istanze delle stesse variabile durante la ricorsione. Il problema e' che nel caso di variabili

statiche, tutte le "copie" in realta' sono ancora legate alla stessa cella di memoria andando a sovrascrivere di volta in volta il valore della variabile, rendendo le funzioni ricorsive imprevedibili

Variabili Stack-Dynamic

Rappresentano quelle variabili dove lo storage binding viene effettuato quando le loro istruzioni di dichiarazione vengono effettivamente elaborate, quindi quando il codice eseguibile di tale istruzione viene eseguito.
Sono il tipo di variabili piu' comune che noi usiamo

Example

Quando chiamiamo una funzione, se quella funzione ha delle variabili locali NON STATICHE, esse vengono allocate grazie alle loro dichiarazioni quando il metodo viene chiamato e poi verranno deallocate quando il metodo termina la sua esecuzione

Per essere piu' precisi, in realta' i compilatori sono spesso intelligenti, e in linguaggi come C++ e Java, se in qualsiasi parte di un metodo viene fatta una dichiarazione di qualche variaible, tale dichiarazione viene in realta' fatta all'inizio della chiamata di tale funzione e non dove viene posta l'istruzione di dichiarazione. Questo e' fatto principalmente per una questione di **ottimizzazione**

Caution

Anche se viene fatta all'inizio del metodo comunque la variabile rimane utilizzabile solo a partire dalla sua istruzione di dichiarazione per motivi di coerenza

Vantaggi e Svantaggi delle variabili Stack Dynamic

Vantaggi

- Permette la ricorsione
- Permette un'ottimizzazione della memoria: anche se un programma non utilizza ricorsione, alla fine tutti i sottoprogrammi di un certo codice condividerannola stessa parte di memoria. Siccome (almeno nella maggior parte dei casi) non verranno mai eseguite tutte le funzioni di un programma contemporaneamente, cio' permette a piu' sottoprogrammi di condividere la stessa area di memoria alternativamente
  Svantaggi:

- Overhead minimo di allocazione e deallocazione
- Indirizzamento possibilmente piu' lento in quanto indiretto
- Programmi con SOLO Stack-Dynamic variables non possono piu' avere sottoprogrammi sensibili alla cronologia

Variabili Dinamiche esplicitamente Heap Dynamic

Vengono allocate e deallocate nello Heap tramite **direttive esplicite** specificate dal programmatore durante l'esecuzione. Una volta allocate in tal modo possono essere referenziate tramite puntatori oppure variabile riferimento

> **Example**
>
> Alcuni esempi di direttive sono *new* in C++, oppure *malloc()* in C

Allo stesso modo (solitamente) i linguaggi predispongono anche direttive specifiche per la loro deallocazione

> **Example**
>
> Come la *delete* in C++ o la *free()* in C
> o altri sistemi piu' complessi senza direttive esplicite come quelli di garbage collecting fatti da linguaggi come Java

Variabili di questo tipo sono spesso utilizzate per strutture dati la cui dimensione non e' ben nota a compile time come Alberi, Vector, Liste,....

Vantaggi e Svantaggi di Vaiabili Dinamiche esplicitamente Heap Dynamic

Vantaggi:

- Gestione dinamica della memorizzazione (Strutture dati complesse)
  Svantaggi:
- Difficolta' nell'uso di puntatori e riferimenti e una maggiore attenzione nell'uso corretto
- Allocazione e Deallocazione piu' onerosa (overhead)

Variabili Dinamiche Implictamente Heap-Dynamic

Il metodo di allocazione e' sempre lo stesso con la differenze che non esistono istruzioni esplicite, bensi' viene fatto implicitamente da alcuni linguaggi

> **Example**

Un esempio e' proprio Javascript

```
vals = [23,75,21,55,12];
```

Stiamo creando un array (in realta' lista in JS) senza direttive specifiche ma nonostante cio' tale variabile, a nostra insaputa, verra' messa sullo heap

Vantaggi e Svantaggi di Variabile Dinamiche Implicitamente Heap Dynamic

Vantaggi

- Flessibilita' piu' alta disponibile
  Svantaggi
- Inefficiente, tanto overhead di allocazione e deallocazione se tutte le variaibili anche le piu' semplici, sono Heap-Dynamic
- Perdita di rilevazione di errori da parte dei compilatori

Scope

Definition

Lo scope di una variabile e' l'intervallo di istruzioni in cui quella variabile e' visibile.

Una variabile e' **visibile** se' possibile farci riferimento oppure le si puo' assegnare un valore.
Le regole di scope di un linguaggio determinano come le varie occorrenze dei nomi in un programma sono associati alle variabili e in particolar modo determinano come i riferimenti di variabili dichiarate al di fuori del sottoprogramma corrente sono associati con le dichiarazioni originali

Esistono 3 tipi di scope:

- *Variabili Locali* : Variabili dichiarate in una certa unita' di un programma
- _Variabili non locali: sono quelle visibili in una certa unita' di un programma ma che non sono state dichiarate in essa
- *Variabili Globali*: Sono un categoria speciale per le variabili non locali

Scope statico

Viene chiamato cosi' perche' in questo caso lo scope di una variabile si puo' determinare staticamente, quindi prima che il programma venga messo in esecuzione.
Esistono in realta' due categorie di linguaggi statically-scoped, ma noi considereremo solo quella che consente sottoprogrammi innestati.

Question

Come funziona?
Quando viene individuata (dal lettore/compilatore) un riferimento ad una certa variabile, l'obiettivo e' appunto trovarne la sua dichiarazione. Viene cercata quindi nello scope corrente/locale, poi nello scope "superiore" dove e' definito il sottoprogramma corrente (detto **static ancestor**), e cosi' via fino a quando non viene trovata, altrimenti viene sollevato un errore.

Example

```
function big(){
   function sub1(){
   var x = 7;
   sub2();
   }
   function sub2(){
       var y = x;
   }
   var x = 3;
   sub1();
}
```

- In sub2, la definizione di X viene trovata nel suo Ancestor, quindi nella funzione big()
- Sub1 invece la definizione nell'ancestor viene nascosta da quella locale in sub1

Scope dinamico

In caso di scope dinamico, lo scope si basa sulla sequenze delle chiamate a funzioni invece che alla loro relazione spaziale come abbiamo visto con lo scope statico. In questo caso quindi, l'unico modo per determinare lo scope di una variabile e' solamente a **run time**

> Inoltre, una dichiarazione resta valida fino a quando viene trovata una dichiarazione per lo stesso identificatore

Il funziomento e' simile, solamente che la dichiarazione di tale variabile viene cercata nella funzione chiamante e non nello scope dell'ancestor. Relazione di tipo "Calling History".
Vediamo un esempio:

```
function big(){
    function sub1() {
        var x = 7;
    }
    function sub2(){
        var y = x;
        var z = 3;
    }
    var x = 3;
}
```

Notiamo che: il significato del riferimento di x, nella funzione *sub2()* e' solamente comprensibile a run time perche':

- Se chiamiamo sub2() a partire da big, a quel punto il valore di x e' quello dichiarato in big
- Se chiamiamo sub2() a partire da sub1() a quel punto il valore di x e' quello dichiarato in sub1()

Al posto che cercare gli "Static Ancestor" ora cerchiamo le funzioni chiamanti in ordine inverso, chiamate **dynamic parent**

Lo svantaggio principale dei linguaggi che utilizzano scope dinamici riguarda la leggibilita' del codice ma inoltre i programmi sono piu' lenti perche' il controllo di tipo deve essere per forza a run time ed e' pertanto poco efficiente

Scope e lifetime

Molte volte, lo scope e il lifetime di una variabile sono strettamente correlati ma sono ovviamente due concetti diversi.
L'esempio piu' chiaro che si puo' avere e' tramite la keyword *static* in C/C++

> **Example**
>
> Se una variabile e' dichiarata in una funzione con la keyword *static* e' legata staticamente allo scope di quella funzione (quindi esiste solo li) ed

e' anche legata staticamente alla memoria. Quindi il suo scope e' statico e locale alla funzione ma il suo lifetime si estende per la durata intera dell'esecuzione del programma di cui fa parte

Altro esempio:

Example

```
void printheader() {
 . . .
} /* end of printheader */
void compute() {
 int sum;
 . . .
 printheader();
} /* end of compute */
```

- Lo scope della variabile sum è completamente contenuto all'interno della funzione compute. Non si estende al corpo della funzione printheader, anche se printheader viene eseguita all'interno di compute
- Il lifetime di sum si estende per tutto il tempo durante il quale printheader viene eseguita. Qualunque sia la locazione di memoria a cui sum è legata prima della chiamata a printheader, quel legame continuerà durante e dopo l'esecuzione di printheader

Referencing Environment

E' l'insieme di tutti gli identificatori che sono visibili in un certo punto del programma. In un linguaggio a scope statico e' facile capire quale e' il referencing environmente in quanto non e' altro che l'insieme di tutte le variabili dichiarate nello scope locale + la collezione di tutte le variabili del suo ancestor visibili

Invece, in un linguaggio a scoping dinamico, e' costituito da tutte le variabili definite localmente piu' tutte quelle delle funzioni attive non ancora terminate.

Referencing Environment in python

```
g = 3; # A global
def sub1():
  a = 5; # Creates a local
```

```
  b = 7; # Creates another local
. . .               1
1
  def sub2():
    global g; # Global g is now assignable here

    c = 9; # Creates a new local
    . . .            2
2
    def sub3():
      nonlocal c: # Makes nonlocal c visible here
      g = 11; # Creates a new local
      . . .           3
```

In questo esempio python:

1. si vede le variabili locali a e b (da sub1), global g in sola lettura
2. Variabile locale c (da sub2), global g utilizzabile sia per reference che per assegnamento grazie alla keyword *global*. Se volessi usare a in sub2(), non basterebbe fare *a = 2*, ma dovrei specificare che voglio usare una variabile non dallo scope locale con la keyword *nonlocal a*, altrimenti avrei un altra instanza di
3. variabile non locale c (da sub2), locale g(da sub3)
   Ricoirdiamo che anche se sub1() e' piu' in alto di livello (nello scope) comunque in python sub1 non e' uno static ancestor di sub3 quindi sub3 non puo' accedere alle variabili in sub1

Referencing Environment in Linguaggi a scope dinamico

Il referencing environment in linguaggi a scope dinamico e' dato da tutte le variabili locali piu' tutte le variabili visibili in TUTTI i sottoprogrammi attivi.
Ovviamente, alcune variabili possono essere nascoste per colpa del referencing envorinment
Vediamo un esempio:

Example

```
void sub1() {
  int a, b;
  . . .                                                    1

} /* end of sub1 */
```

```
void sub2() {
  int b, c;
  .. . .                                                    2

  sub1();
} /* end of sub2 */
void main() {

  int c, d;
  . . .                                                     3
  sub2();
} /* end of main */
```

I cui referencing environemnt sono:

1. variabli a e b di sub1, c di sub2 d di main (c di main e b di sub2 rimangono nascoste)
2. b e c di sub2), d di main (c di main è nascosta)
3. c e d di main

Costanti

Una costante e' una variabile che e' legata ad un valore una sola volta, cioe' al momento dell'associazione a una cella di memoria. Vengono utilizzate spesso per la parametrizzazione dei programmi