Concetti di Base sui Linguaggi di Programmazione

Luca Abeni

November 29, 2021

1 Identificatori, Legami ed Ambiente

Una caratteristica di quasi tutti i linguaggi di programmazione (da quelli di basso livello come l'Assembly a quelli di più alto livello) è la possibilità di associare nomi alle "entità" che compongono i programmi (siano esse semplicemente valori, locazioni di memoria, variabili, funzioni o altro...).

Più formalmente, ogni linguaggio di programmazione è composto da alcune *entità denotabili*, che è possibile identificare tramite nomi (identificatori) più facilmente gestibili dal programmatore. Esiste quindi una funzione, chiamata *ambiente*, che ha come dominio l'insieme degli identificatori e come codominio l'insieme delle entità denotabili del linguaggio. Tale funzione cambia nel tempo (è possibile creare o distruggere dinamicamente dei legami fra identificatori ed entità denotabili durante l'esecuzione del programma) e nei vari punti del programma (alcuni legami possono essere validi in alcune zone del codice ma non in altre)[1]. Più formalmente, si può quindi dire che:

Definizione 1 (Legame) *Si definisce legame (binding) fra un identificatore I ed un'entità denotabile E una coppia (I, E) che associa l'entità al nome.*

Definizione 2 (Ambiente) *Si definisce ambiente l'insieme di legami esistenti in uno specifico momento dell'esecuzione di un programma (mentre si esegue codice in uno specifico punto del programma).*

(questa definizione di ambiente non deve sorprendere, ricordando che dal punto di vista matematico una funzione è un insieme di coppie - un sottoinsieme del prodotto cartesiano di dominio e codominio).

Come detto, l'ambiente contiene i legami fra identificatori simbolici e generiche "entità denotabili", che dipendono dal linguaggio. Per esempio,

- nel linguaggio Assembly gli identificatori sono associabili solo ad indirizzi di memoria
- in un linguaggio imperativo di più alto livello, gli identificatori possono essere associati a valori, variabili, funzioni o tipi
- nel λ-calcolo, gli identificatori sono associabili a funzioni
- in un linguaggio funzionale, gli identificatori sono associabili solo a valori (in un linguaggio funzionale, una funzione è in effetti un valore)

In generale, quando un linguaggio prevede il concetto di blocco di codice è possibile distinguere i concetti di ambiente locale, ambiente non locale ed ambiente globale.

Definizione 3 (Ambiente Locale) *Si definisce ambiente locale di un blocco di codice il sottoinsieme dell'ambiente composto da legami che sono stati creati nel blocco di codice in questione (e che sono quindi validi solo all'interno di tale blocco di codice).*

Definizione 4 (Ambiente non Locale) *Si definisce ambiente non locale di un blocco di codice il sottoinsieme dell'ambiente che non è composto dall'ambiente locale del blocco in questione (contiene quindi legami che sono stati creati fuori dal blocco e rimarranno validi quando l'esecuzione uscirà dal blocco di codice).*

Definizione 5 (Ambiente Globale) *Si definisce ambiente globale il sottoinsieme dell'ambiente composto da legami che non sono stati creati all'interno di alcun blocco di codice (o, sono stati creati nel blocco di codice più esterno, che contiene tutti gli altri blocchi).*

[1] L'ambiente è modificato, per esempio, da una dichiarazione, o dal primo utilizzo di un'entità denotabile (per linguaggi in cui non è necessario dichiarare un'entità prima di utilizzarla).

2 Variabili Modificabili

Nei linguaggi imperativi la computazione procede attraverso la modifica di valori memorizzati in locazioni di memoria, che nei linguaggi di alto livello sono identificate da variabili.

Definizione 6 (Variabile Modificabile) *Una variabile modificabile è un tipo di entità denotabile che rappresenta una zona di memoria che può contenere entità memorizzabili.*

Nella precedente definizione si utilizza il concetto di "entità memorizzabile", il cui significato dipende ancora una volta dal linguaggio, ma che sta ad indicare le entità che possono essere conteneute in una variabile modificabile (per esempio, in un linguaggio funzionale una funzione è un'entità memorizzabile, mentre in alcuni linguaggi imperativi non lo è).

Dal punto di vista concettuale, la presenza di variabili modificabili introduce una nuova funzione, detta "store", che associa ad ogni variabile l'entità memorizzabile in essa contenuta.

Definizione 7 (store) *Si definisce store l'insieme di coppie (V, E), dove V è una variabile ed E è un'entità memorizzabile che associano ad ogni variabile il suo contenuto.*

Lo store è quindi una funzione (che rappresenta la memoria utilizzabile dal nostro programma per i dati) avente come dominio l'insieme delle variabili e come codominio l'insieme delle entità memorizzabili.

Quando in un linguaggio imperativo si utilizza il valore contenuto nella variabile "`x`", si va in realtà ad applicare la funzione store al valore ritornato dalla funzione ambiente applicata all'identificatore "`x`": $store(env(\texttt{x}))$ (dove "env" è la funzione ambiente).

3 Entità Denotabili, Esprimibili e Memorizzabili

Si è visto come un programma sia composto (fra le altre cose) da generiche "entità" (la cui definizione dipende dal linguaggio di programmazione, ma che in generale possono essere tipi, valori, variabili, funzioni, etc...). Tali entità possono essere in generale *denotabili*, *memorizzabili* ed `esprimibili`.

Definizione 8 (Entità Denotabile) *Si definisce entità denotabile un'entità che può essere associata ad un nome / identificatore.*

Definizione 9 (Entità Memorizzabile) *Si definisce entità memorizzabile un'entità che può essere contenuta in una variabile (usando un linguaggio di programmazione imperativo)*

Definizione 10 (Entità Esprimibile) *Si definisce entità esprimibile un'entità che può essere generata come risultato di un'espressione*

Un'entità denotabile è quindi ogni entità che si può indicare con un nome (definito dall'utente o predefinito dal linguaggio) e l'insieme delle entità denotabili è il codominio della funzione ambiente. Un'entità esprimibile è invece qualsiasi entità che può "essere calcolata/allocata" in qualche modo usando i costrutti del linguaggio. Per finire, le entità memorizzabili (che sono una caratteristica dei linguaggi imperativi) costituiscono il codominio della funzione store.

In un linguaggio funzionale, tutte le entità sono denotabili ed esprimibili (non ha senso parlare di entità memorizzabili, in quanto non esiste la funzione store), mentre in un linguaggio di programmazione imperativo va fatta la distinzione fra varie tipologie di entità (possono esistere per esempio entità denotabili ma non esprimibili o memorizzabili, come le funzioni in alcuni linguaggi imperativi).

Si noti che in letteratura le definizioni di denotabile, esprimibile e memorizzabile si associano talvolta solo ai valori.

4 Funzioni

Una delle caratteristiche fondamentali dei linguaggi di programmazione di alto livello è quella di permettere di modularizzare il codice, scomponendo il programma in una serie di componenti (sottoprogrammi, subroutine) ognuno dei quali implementa una specifica funzionalità, secondo un'interfaccia ben definita.

Ognuno di questi sottprogrammi è quindi un'entità che rappresenta una parte auto-contenuta del codice che può essere invocata passando dei parametri e ricevento eventualmente in ritorno dei valori. Generalmente, un sottoprogramma che può ritornare un valore viene definito funzione (anche se è una cosa molto differente da una funzione matematica - può avere effetti collaterali!!!) e nella nomenclatura informatica moderna si tende a parlare di funzione anche per sottoprogrammi che non hanno valori di ritorno (perché dal punto di vista concettuale è come se ritornassero un valore di tipo `void`, o `unit`).

```
void wrong_swap(int a, int b)
{
  int tmp;

  tmp = a; a = b; b = tmp;
}

void correct_swap(int *a, int *b)
{
  int tmp;

  tmp = *a; *a = *b; *b = tmp;
}
```

Figure 1: Esempio di funzione che scambia il contenuto di due variabili, implementata in C. Visto che i parametri vengono passati per valore, la funzione "`wrong_swap()`" non sortirà nessun effetto; la funzione "`correct_swap()`", invece, ricevendo come parametri i *puntatori* alle variabili da scambiare, potà funzionare correttamente.

Definizione 11 (Funzione) *Si definisce funzione un'entità denotabile composta da un blocco di codice a cui può essere associato un nome utilizzabile per invocarne l'esecuzione. Nel momento in cui viene invocata l'esecuzione di una funzione il codice chiamante può scambiare dei dati con la funzione tramite i suoi parametri, il valore di ritorno o lo stato globale del programma.*

Definizione 12 (Parametro Formale) *Un parametro formale di una funzione, definito nella definizione della funzione, identifica una variabile che può essere usata dal codice chiamante per passare dei dati ad una funzione quando la invoca.*

Definizione 13 (Parametro Attuale) *Un parametro attuale è un'espressione, specificata nel momento in cui si invoca una funzione, che verrà associata al corrispondente parametro formale durante l'esecuzione della funzione.*

Il fatto che una funzione sia definita come un'entità denotabile chiarisce subito che al blocco di codice è possibile (anche se non sempre obbligatorio: esistono anche funzioni anonime!) associare un nome. Inoltre, il fatto che la funzione sia un blocco di codice chiarisce che la funzione è caratterizzata da un ambiente locale (legami fra variabili locali alla funzione e loro nomi, legami fra parametri formali e parametri attuali, etc...).

Il modo in cui i parametri attuali vengono associati ai parametri attuali dipende dal meccanismo di invocazione della funzione e dalla modalità di passaggio dei parametri utilizzata. Per esempio, come detto un parametro formale identifica una variabile; tale variabile può essere creata nel momento in cui si invoca la funzione e distrutta quando l'esecuzione della funzione termina (avendo quindi un tempo di vita che coincide col tempo di esecuzione della funzione), oppure può essere pre-esistente all'esecuzione della funzione (rimanendo in vita quando l'esecuzione della funzione è terminata). Esistono quindi diverse modalità di passaggio dei parametri (dipendenti dal particolare meccanismo di applicazione di funzione che viene utilizzato), le più importanti dei quali sono il passaggio di parametri per valore, il passaggio di parametri per riferimento ed il passaggio di parametri per nome.

Nel caso di passaggio parametri *per valore*, una variabile locale viene allocata quando la funzione è invocata e deallocata quando l'esecuzione della funzione termina (questo è facilmente fattibile allocando la variabile sullo stack, nel record di attivazione dell'invocazione di funzione). L'ambiente locale della funzione contiene quindi un binding fra il nome del parametro formale e questa variabile, il cui contenuto viene inizializzato col valore ottenuto valutando l'espressione passata come parametro attuale al momento dell'invocazione (da qui il nome "passaggio di parametri per valore"). Un'importante proprietà del passaggio di parametri per valore è che eventuali modifiche al valore del parametro effettuate durante l'esecuzione della funzione vengono "perse" quando l'esecuzione termina (in altre parole, il passaggio di parametri per valore permette di passare dati dal chiamante alla funzione, ma non viceversa). Questo è l'unico tipo di passaggio di parametri supportato dal linguaggio C; si noti infatti che quando una funzione C deve modificare il valore di una variabile passata come parametro attuale, il "vero parametro" non può essere la variabile, ma deve essere un puntatore ad essa. In altre parole, la funzione "`wrong_swap()`"

```
void reference_swap(int &a, int &b)
{
  int tmp;

  tmp = a; a = b; b = tmp;
}
```

Figure 2: Esempio di funzione che scambia il contenuto di due variabili, implementata in C++ tramite passaggio di parametri per riferimento.

```
int f(int v)
{
  int a = 666;

  return a + v;
}
```

Figure 3: Esempio di funzione per il passaggio di parametri per nome.

di figura 1 non può realmente scambiare il contenuto delle variabili passate come parametri attuali (la funzione "`correct_swap()`", invece, si comporterà in modo corretto).

Nel caso di passaggio di parametri *per riferimento*, invece, all'invocazione della funzione non viene creata alcuna nuova variabile per i parametri formali. L'ambiente locale della funzione è semplicemente modificato aggiungendo un binding fra il nome del parametro formale ed il parametro attuale. Questo significa che i parametri attuali passati per riferimento non possono essere generiche espressioni, ma devono essere variabili, o comunque entità denotabili che possano stare a sinistra dell'operatore di assegnamento (L-value, usando la nomenclatura C/C++ — la "*L*" sta per "*Left*", a *sinistra* dell'operatore). Questo tipo di passaggio di parametri non è (come già accennato) supportato dal linguaggio C, ma è supportato dal linguaggio C++; si veda per esempio la funzione `reference_swap()` di Figura 2 (confrontando questo codice col codice mostrato in Figura 1 si possono capire meglio le differenze fra il passaggio di parametri per valore ed il passaggio di parametri per riferimento).

Nel caso di passaggio di parametri *per nome*, infine, all'invocazione della funzione il parametro formale viene sostituito (tramite rimpiazzamento testuale) dall'espressione utilizzata come parametro attuale. Questo è il meccanismo tipicamente utilizzato nella valutazione di programmi funzionali. Sebbene questo meccanismo di passaggio dei parametri possa a prima vista sembrare semplice da implementare, può presentare alcuni subdoli problemi. Per esempio, si consideri la funzione "`f()`" di Figura 3, che (a parte l'opinabile utilizzo della variabile locale "`a`") ha l'evidente obiettivo di sommare 666 al valore ricevuto in ingresso. In effetti, l'invocazione "`f(n)`" viene valutata per nome come "{ **int** a = 666; **return** a + n;}" che diventa "{ **return** 666 + n;}" ed il valore di ritorno è quindi "`666 + n`". Ma come deve essere valutata l'invocazione "`f(a)`", dove "`a`" è una variabile non locale di "`f()`" (per esempio, una variabile globale)? Sostituendo semplicemente "`v`" con "`a`" nel corpo della funzione, si otterrebbe "{ **int** a = 666; **return** a + a;}" che ritorna "`666 + 666`" (e quindi il valore `1332`). Questo non è ovviamente il risultato atteso. Il problema è ovviamente dovuto al fatto che sostituendo semplicemente "`v`" con "`a`" non si è più in grado di distinguere le due diverse variabili (una locale alla funzione ed una non locale) che hanno lo stesso nome "`a`". Questo fenomeno è talvolta noto come "cattura di una variabile libera".

Questo problema viene generalmente risolto nel campo della programmazione funzionale tramite un semplice cambio di nome delle variabili: se la funzione "`f()`" viene invocata usando come parametro attuale un'espressione che contiene una variabile chiamata "`a`", allora la variabile locale "`a`" deve essere rinominata (per esempio in "`a1`") in modo che la sostituzione risulti in "{ **int** a1 = 666; **return** a1 + a;}" che ritorna correttamente "`666 + a`". Dal punto di vista implementativo, invece, il passaggio di parametri per nome può essere realizzato passando un "*thunk*" (vale a dire una coppia ambiente, espressione) al posto del parametro. Quindi, una funzione che riceve parametri passati per nome è implementabile come una funzione che riceve come parametri delle coppie composte da espressioni (i parametri attuali) e dagli ambienti in cui tali espressioni vanno valutate.

Come detto, una funzione che non ha valori di ritorno è modellabile come una funzione per cui il tipo del valore di ritorno ha un unico valore possibile (tipo `unit`, o `void`). Una considerazione analoga vale

anche per i parametri.

5 Chiusure

```
void->int contatore(void)
{
  int n = 0;

  int f(void) {
    return n++;
  }

  return f;
}
```

Figure 4: Esempio di funzione che ritorna una chiusura.

In alcuni linguaggi le funzioni sono considerate entità memorizzabili (esistono variabili che possono memorizzare funzioni) o esprimibili (esistono funzioni che possono ritornare funzioni come valore di ritorno) o possono essere usate come parametri per altre funzioni. In questo caso i valori memorizzati, ritornati o passati come parametri non possono essere semplicemente funzioni, ma devono essere *chiusure*.

Definizione 14 (Chiusura) *Si definisce chiusura una coppia composta da una funzione e dal suo ambiente non locale.*

Sostanzialmente, una chiusura serve per sapere a quali entità denotabili sono associati gli identificatori per cui non esiste un legame nell'ambiente locale della funzione.

```
#include <stdio.h>

int (*counter(void))(void)
{
  int n = 0;

  int f(void) {
    return n++;
  }

  return f;
}

int main()
{
  int (*c1)(void) = counter();
  int (*c2)(void) = counter();
  int (*c3)(void) = counter();

  printf("   %d %d\n", c1(), c1());
  printf("%d %d %d\n", c2(), c2(), c2());
  printf("   %d %d\n", c3(), c3());

  return 0;
}
```

Figure 5: Esempio che mostra che i puntatori a funzione del C non implementa una chiusura (l'esempio utilizza un'estensione non-standard del compilatore `gcc`).

Si consideri per esempio la funzione `void->int contatore(void)` descritta in Figura 4, scritta in uno pseudo-linguaggio simile al C in cui "`void->int`" indica il tipo delle funzioni che non hanno argomenti e ritornano un valore di tipo `int`. La funzione `contatore()` non riceve argomenti e ritorna una funzione che ritorna progressivamente tutti i numeri interi da 0 in poi. Come si può vedere, "`n`" è una variabile locale della funzione `contatore()` e non una variabe o argomento della funzione "`f()`" che viene ritornata. Quindi, nel momento in cui viene invocata (per esempio) "`prossimo = contatore()`" nell'ambiente locale di `contatore()` esiste un legame fra l'identificatore "`n`" ed una variabile che contiene inizialmente il valore "`0`" (e viene incrementata da `f()` ogni volta che la si invoca). Ma tale legame non è nell'ambiente locale di `f()` (è nel suo ambiente non locale). Quando quindi `contatore()` termina ed il suo ambiente locale viene distrutto, non esiste più tale legame e non è chiaro come l'identificatore "`n`" debba essere risolto se invoco "`prossimo()`". Peggio, anche la variabile legata ad "`n`" viene distrutta quando `contatore()` termina. Per rendere quindi utilizzabile questo codice, devono essere fatte 2 cose distinte:

1. La variabile legata all'identificatore "`n`", che contiene valore iniziale "`0`" non deve essere distrutta quando `contatore()` termina. Questo significa che la variabile non deve essere allocata sullo stack, ma nello heap
2. Il legame fra "`n`" e tale variabile non deve essere creato solo nell'ambiente locale di `contatore()`, ma va copiato in un apposito ambiente che farà parte della chiusura ritornata da `contatore()` (e memorizzata in "`prossimo`")

Per capire meglio la differenza fra una chiusura ed una funzione (o un puntatore a funzione, come usato dal linguaggio C), si consideri il programma di Figura 5, che utilizza un'estensione supportata dal compilatore `gcc` per implementare in C la funzione della Figura 4. In questo caso la variabile "`int n`" viene distrutta quando la funzione "`contatore()`" ritorna, quindi il comportamento del programma è indefinito (come facilmente verificabile compilando il programma con `gcc`, eventualmente con vari livelli di ottimizzazione.

6 Chiusure e Classi

```
#include <functional>
#include <iostream>

auto counter(void)
{
  int n = 0;

  return [n](void) mutable {
    return n++;
  };
}

int main()
{
  auto c1 = counter();
  auto c2 = counter();
  auto c3 = counter();

  std::cout << c1() << " " << c1() << std::endl;
  std::cout << c2() << " " << c2() << " " << c2() << std::endl;
  std::cout << c3() << " " << c3() << std::endl;

  return 0;
}
```

Figure 6: Esempio che mostra come le lambda expression del C++ implementano il concetto di chiusura.

```
#include <iostream>

class Counter {
  private:
    int n;
  public:
    Counter(void) : n(0) {
    }
    int operator() (void) {
      return n++;
    }
};

int main()
{
  auto c1 = Counter();
  auto c2 = Counter();
  auto c3 = Counter();

  std::cout << c1() << " " << c1() << std::endl;
  std::cout << c2() << " " << c2() << " " << c2() << std::endl;
  std::cout << c3() << " " << c3() << std::endl;

  return 0;
}
```

Figure 7: Esempio di contatore implementato come classe C++ (da confrontarsi con l'implementazione basata su chiusure).

Il concetto di chiusura, che permette di associare un ambiente non locale ad una funzione, è stato originariamente introdotto per supportare funzioni di ordine superiore (funzioni che ritornano altre funzioni come risultato, o che le accettano come argomento). Ma le chiusure sono costrutti che vanno oltre a questo e permettendo di associare dati (lo stato contenuto nell'ambiente non locale) a codice (il codice che implementa il corpo della funzione) presentano interessanti similitudini e legami col concetto di oggetto.

Per capire meglio la relazione fra chiusure e classi, si può considerare l'implementazione in C++ di un contatore come chiusura e confrontarla con un'implementazione "più tradizionale" di contatore come classe. La Figura 6 mostra come utilizzare il costrutto delle lambda expression C++ (che sono sostanzialmente delle funzioni anonime associate ad un ambiente non locale, quindi implementano il concetto di chiusura) per realizzare un contatore, mentre la Figura 7 rappresenta un contatore implementato come classe (qui l'unica cosa "particolare" è la ridefinizione dell'operatore "`()`"). Come si può vedere confrontando le figure, la variabile intera "`n`" in un caso è una variabile locale della funzione "`contatore()`", che viene poi incapsulata nell'ambiente non locale della funzione ritornata, mentre nell'altro caso è un membro dati privato della classe.

Sostanzialmente quindi sia chiusure che classi permettono di affrontare il problema dell'accesso incontrollato ai dati, anche se lo fanno in modo diverso: le classi permettono di far si che solo il codice "autorizzato" (i metodi della classe) possa accedere allo stato incapsulato nella classe, mentre le chiusure permettono di far si che lo stato accessibile da una funzione stia nell'ambiente contenuto nella sua chiusura e sia quindi inaccessible da altre funzioni perché non referenziabile in esse.

Sebbene questo esempio (progettato per evidenziare la relazione fra chiusure e classi/oggetti) sembri mostrare che una chiusura possa servire semplicemente ad "associare uno stato" ad una funzione, esistono esempi forse più interessanti riguardo all'utilità delle chiusure nella programmazione funzionale: si pensi alla versione curryificata di una funzione che somma due numeri, o ad una funzione che riceve come argomento una funzione che accede a variabili non locali.