Concetti di Base sui Linguaggi di Programmazione

Luca Abeni

October 27, 2020

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 e Chiusure

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

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

  return f;
}
```

Figure 1: Esempio di funzione che ritorna una chiusura.

Definizione 11 (Funzione) *Si definisce funzione un'entità denotabile composta da un blocco di codice a coi viene associato un nome che può essere utilizzato 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.*

Il fatto che una funzione sia definita come un'entità denotabile chiarisce subito che al blocco di codice è possibile 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...).

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.

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 sono veramente funzioni, ma *chiusure*.

Definizione 12 (Chiusura) *Si definisce chiusura una coppia composta da una funzione e dal suo ambiente non locale.*

Sostanzialmente, una chiusura serve per sapere a che entità denotabili associare gli identificatori per cui non esiste un legame nell'ambiente locale della funzione. Si consideri per esempio la funzione `void->int contatore(void)` descritta in Figura 1, 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`")

Sebbene questo esempio sembri mostrare che una chiusura possa servire semplicemente ad "associare uno stato" ad una funzione (mostrando quindi la relazione fra chiusure ed oggetti), 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.