Guida agli Esami di Linguaggi

VR443470

luglio 2023

Indice

1 Esercizio 1 - Domanda di teoria su Interprete e Compilatore

1.1 Interprete

In molti esami si presenta la richiesta della definizione di interprete. Nonostante possa essere banale, viene richiesto un "alto" livello di approfondimento dato che vale ben 4 punti all'interno dell'esame. In ogni caso, è possibile affermare che questa domanda sia una delle più gettonate.

Definire intuitivamente e formalmente (mediante definizione semantica) cosa è un interprete. Descrivere la struttura di un interprete e spiegarne il funzionamento.

Risposta

La definizione intuitiva di un interprete è la seguente.

Un **interprete** è un programma $\text{int}^{L_0,L}$ che esegue, sulla macchina astratta per L_0, programmi P^L, i quali sono scritti nel linguaggio di programmazione L, su un input fissato appartenente all'insieme dei dati D (input e output).
Utilizzando parole povere, un interprete non è altro che una "macchina universale" che preso un programma e un suo input, esegue (il programma) sul quel determinato input usando soltanto le funzionalità messe a disposizione dal livello (macchina astratta) sottostante.

Un attimo, ma che cosa si intende per livello? E macchina astratta? Cerchiamo di fare chiarezza.

Dato un linguaggio di programmazione L, la macchina astratta M_L per L è un insieme di strutture dati ed algoritmi che consentono di memorizzare ed eseguire programmi scritti in L. La sua realizzazione può essere fatta in Hardware, Firmware o Software.

Si ma quindi cosa si intende per livello?

Con la scelta della categoria "realizzazione software", vengono utilizzati, per la realizzazione di strutture dati e algoritmi, linguaggi di programmazione ad alto livello poiché essi implementano una struttura suddivisa a livelli di astrazione.
Per concludere, la macchina astratta può essere dunque vista come una stratificazione di livelli dove ciascuno di essi coopera in modo sequenziale, ma allo stesso tempo è indipendente.

Per non lasciare niente al caso, con "linguaggio di programmazione ad alto livello" ci si riferisce a tutti quei linguaggi di programmazione che offrono un livello di astrazione molto alto dei dettagli del funzionamento del calcolatore.

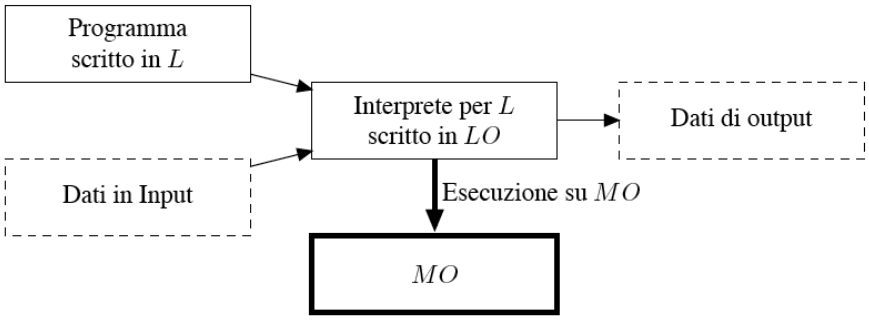
Prima della definizione formale di interprete, si illustrano due notazioni necessarie:

- Con il termine $Prog^L$ ci si riferisce all'insieme dei programmi scritti nel linguaggio di programmazione L;
- Con la dicitura:
$$[\![P^L]\!] : D \to D$$
Si indica che l'esecuzione del programma scritto nel linguaggio di programmazione L ($[\![P^L]\!]$) con input in è uguale all'output out. Ovverosia:
$$[\![P^L]\!](in) = out$$

Un **interprete formalmente** è descrivibile nel seguente modo. Si consideri un interprete da L a L_0: dato $P^L \in Prog^L$ e $in \in D$, un interprete int^{L,L_0} per L su L_0 è un programma tale che:

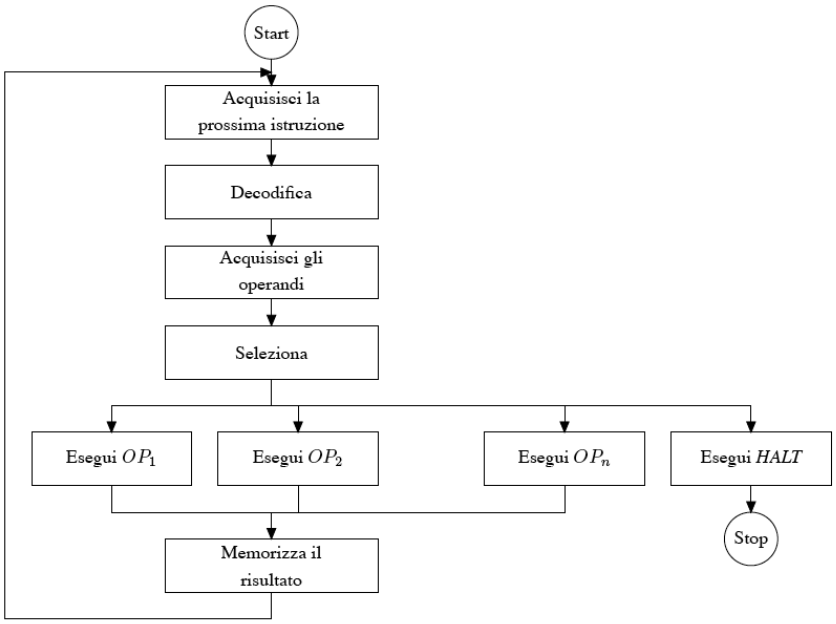
$$[\![int^{L,L_0}]\!] : \left(Prog^L\right) \to D$$

Ne consegue che:
$$[\![int^{L,L_0}]\!] : \left(P^L, in\right) = [\![P^L]\!](in)$$



Riassunto visivo di quanto appena descritto.

Qua di seguito si presenta il ciclo di esecuzione di un interprete:



1. Il ciclo di esecuzione di un interprete si apre con l'acquisizione di un'istruzione da eseguire;
2. L'operazione del passo precedente viene decodificata;
3. Nel caso in cui l'operazione abbia bisogno di operandi, anch'essi vengono acquisiti dalla memoria;
4. Data l'operazione acquisita al passo 1, viene selezionata una determinata operazione: nel caso di $OP_{...}$ si esegue un'operazione, nel caso di $HALT$ l'esecuzione di un interprete si ferma;
5. Dopo l'operazione eseguita, se vi è un risultato, esso viene salvato. In ogni caso, il ciclo inizia nuovamente la sua esecuzione.

1.2 Compilatore

Non è frequente la richiesta della definizione di compilatore, ma rimane una domanda di teoria che può essere richiesta.

Definire intuitivamente e formalmente (mediante definizione semantica) cosa è un compilatore. Descrivere e spiegare poi la struttura di un compilatore (preferibilmente come flow-chart).

Risposta

La definizione intuitiva di un compilatore è la seguente.

Un **compilatore** è un programma $comp^{L_0,L}$ che traduce, preservando semantica e funzionalità, programmi scritti nel linguaggio di programmazione L in programmi scritti in L_0, e quindi eseguibili direttamente sulla macchina astratta per L_0.

Dato un linguaggio di programmazione L, una macchina astratta M_L per L è un insieme di strutture dati e algoritmi che consentono la memorizzazione e l'esecuzione dei programmi scritti nel linguaggio di programmazione L (P^L).

Prima di dare la definizione formale di interprete, si forniscono alcune notazioni:

- $Prog^L$ è un insieme di programmi scritti nel linguaggio di programmazione L;
- D è l'insieme dei dati (input e output);
- P^L è il programma scritto nel linguaggio di programmazione L;
- $[\![P^L]\!] : D \to D$ rappresenta la semantica di P^L, ovvero l'esecuzione del programma nel linguaggio di programmazione L con input in è uguale all'output out:
$$[\![P^L]\!](in) = (out)$$

La definizione formale di un compilatore è la seguente.

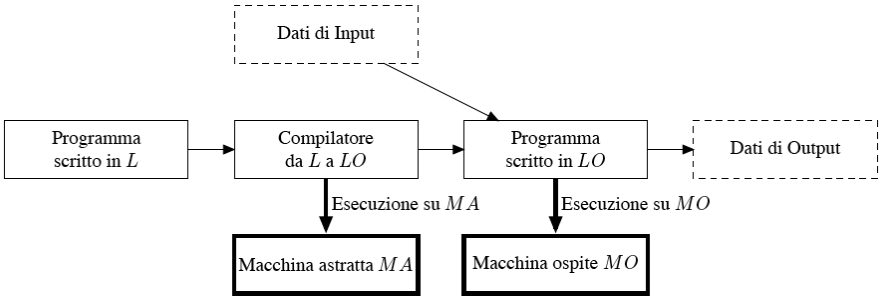
Dato $P^L \in Prog^L$, un **compilatore formalmente** $comp^{L,L_0}$ da L a L_0 è un programma che:

$$[\![comp^{L,L_0}]\!] : Prog^L \to Prog^{L_0}$$

Ovvero:

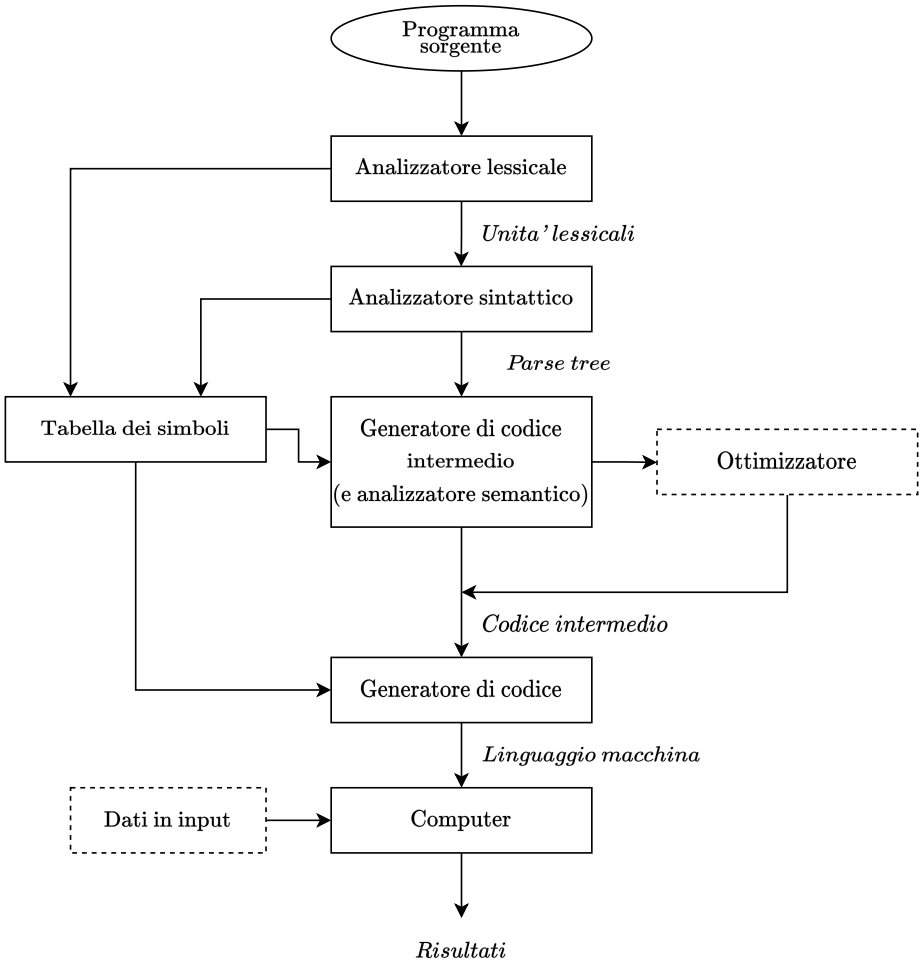
$$[\![comp^{L,L_0}]\!]\left(P^L\right) = \left(P^{L_0}\right) \quad \text{tale che} \quad \forall in \in D.[\![P^{L_0}]\!](in) = [\![P^L]\!](in)$$

In linguaggio non matematico, significa che l'esecuzione del compilatore da L a L_0 insieme (input) ad un determinato programma scritto in un linguaggio di programmazione L è uguale (output) al programma scritto nel linguaggio di programmazione P^{L_0}; tale che per ogni input in appartenente all'insieme dei dati D, l'esecuzione di un programma scritto nel linguaggio di programmazione L_0 con input in è uguale all'esecuzione del programma scritto nel linguaggio di programmazione L con input in.



Flow-chart di quanto detto precedentemente.

Si presenta qua di seguito la **struttura di un compilatore**:



1. Il programma sorgente, da compilare, passa in prima battuta da un analizzatore lessicale. Vengono **convertiti i caratteri del programma sorgente in unità lessicali**. Quest'ultime possono essere *identificatori*, *numeri*, *parole riservate* e formano i linguaggi regolari.

2. Le unità lessicali finiscono in un analizzatore sintattico, il quale crea una albero che rappresenta la sintassi del programma:

   - Le foglie (*token*) vengono lette da sinistra verso destra e costituiscono le frasi ben formate del linguaggio;
   - Dal precedente punto, ne consegue che l'impossibilità della costruzione di un albero è dovuta all'illegalità di quale frase (mal formata, errore di compilazione!);

   Quindi, l'**analizzatore sintattico trasforma unità lessicali in *parse tree* (albero di parse) che rappresentano la struttura sintattica del programma**.

3. Riceve informazioni dall'analizzatore lessicale/sintattico e **memorizza informazioni sui nomi presenti nel programma** (identificatori, chiamate di procedura, ecc.);

4. A questo punto, viene **generato un codice intermedio** che è *indipendente dall'architettura*, e vengono **rilevati eventuali errori semantici** grazie all'*analizzatore semantico*;

5. (Opzionale) Il codice può essere ottimizzato in questa fase;

6. Si conclude la struttura del compilatore con la generazione del codice macchina che, a differenza del codice intermedio, è dipendente dall'architettura.

La parte finale dello schema mostra l'esecuzione del programma compilato su un computer con eventuali dati in input.

2 Esercizio 2 - Induzione

2.1 Dimostrare $\forall n \in \mathbb{N}.n + n^2$ è un numero pari

Dimostrare per induzione che $\forall n \in \mathbb{N}.n + n^2$ ***è un numero pari.***

Risposta

L'**induzione** è una tecnica di dimostrazione che consente di dimostrare la validità di una tesi dalla verifica di due condizioni: la validità della "base induttiva" e la validità del "passo induttivo".

Caso base. Si sceglie come caso base il valore $n = 1$. Si sostituisce:

$$\begin{aligned} n + n^2 \quad &\rightarrow \quad \text{è pari? } \checkmark \\ &\downarrow \quad \text{sostituzione di } n = 1 \\ 1 + 1^2 = 2 \quad &\rightarrow \quad \text{è pari? } \checkmark \end{aligned}$$

Utilizzando un linguaggio più formale, utile per la dimostrazione, è anche possibile affermare che il modulo di $n + n^2$ diviso 2 è uguale a zero. Ovvero, che dividendo un numero pari per 2 (per definizione), si ottiene il valore zero:

$$\begin{aligned} n + n^2 \quad &= \quad 0 \pmod 2 \\ &\downarrow \quad \text{sostituzione di } n = 1 \\ 1 + 1^2 \quad &= \quad 0 \pmod 2 \end{aligned}$$

QED

Ipotesi induttiva: si assume che sia vera:

$$n + n^2 = 0 \pmod 2 \quad \forall n \in \mathbb{N}$$

Passo induttivo. Si dimostra che l'ipotesi induttiva implica la validità della proprietà per $n + 1$:

$$\begin{aligned} n + n^2 \quad &= \quad 0 \pmod 2 \\ &\downarrow \quad \text{applico passo induttivo} \\ n + 1 + (n+1)^2 \quad &= \quad 0 \pmod 2 \\ n + 1 + n^2 + 2n + 1 \quad &= \quad 0 \pmod 2 \\ n^2 + n + 2n + 2 \quad &= \quad 0 \pmod 2 \\ &\downarrow \quad \text{applico l'ipotesi induttiva} \\ n^2 + n + 2n + 2 \quad &= \quad n^2 + n \\ 2n + 2 \quad &= \quad 0 \pmod 2 \end{aligned}$$

Per definizione:

- Qualsiasi numero moltiplicato per 2 si ottiene un numero pari;
- Il risultato tra la somma di due numeri pari è ancora un numero pari;
- 2 è un numero pari.

QED

2.2 Dimostrare $\sum_{i=1}^{n} \frac{1}{i\,(i+1)} = \frac{n}{n+1}$

Dimostrare per induzione che $\sum_{i=1}^{n} \frac{1}{i\,(i+1)} = \frac{n}{n+1}$.

Risposta

L'**induzione** è una tecnica di dimostrazione che consente di dimostrare la validità di una tesi dalla verifica di due condizioni: la validità della "base induttiva" e la validità del "passo induttivo".

Caso base. Si sceglie come caso base il valore $n = 1$. Quindi, si va a sostituire:

$$\begin{aligned}
\sum_{i=1}^{n} \frac{1}{i\,(i+1)} &= \frac{n}{n+1} \\
&\downarrow \text{ sostituzione } n = 1 \\
\sum_{i=1}^{1} \frac{1}{i\,(i+1)} &= \frac{1}{1+1} \\
&\downarrow \text{ calcolo della sommatoria} \\
\sum_{i=1}^{1} \frac{1}{1\,(1+1)} &= \frac{1}{1+1} \\
\frac{1}{2} &= \frac{1}{2}
\end{aligned}$$

QED

Ipotesi induttiva: si assume che sia vera:

$$\sum_{i=1}^{n} \frac{1}{i\,(i+1)} = \frac{n}{n+1} \qquad \forall n$$

Passo induttivo. Si dimostra che l'ipotesi induttiva implica la validità della proprietà per $n + 1$:

$$\begin{aligned}
\sum_{i=1}^{n} \frac{1}{i\,(i+1)} &= \frac{n}{n+1} \\
&\downarrow \text{ applicazione passo induttivo} \\
\sum_{i=1}^{n+1} \frac{1}{i\,(i+1)} &= \frac{n+1}{(n+1)+1} \\
\sum_{i=1}^{n} \frac{1}{i\,(i+1)} + \frac{1}{(n+1)\cdot((n+1)+1)} &= \frac{n+1}{(n+1)+1}
\end{aligned}$$

$$\downarrow \quad \text{utilizzo ipotesi induttiva}$$

$$\frac{n}{n+1} + \frac{1}{(n+1)\cdot((n+1)+1)} = \frac{n+1}{(n+1)+1}$$

$$\frac{n}{n+1} + \frac{1}{(n^2+2n+1)+(n+1)} = \frac{n+1}{(n+1)+1}$$

$$\frac{n}{n+1} + \frac{1}{(n+1)(n+2)} = \frac{n+1}{(n+1)+1}$$

$$\frac{n(n+2)+1}{(n+1)(n+2)} = \frac{n+1}{(n+1)+1}$$

$$\frac{n^2+2n+1}{(n+1)(n+2)} = \frac{n+1}{(n+1)+1}$$

$$\frac{(n+1)^2}{(n+1)(n+2)} = \frac{n+1}{(n+1)+1}$$

$$\frac{n+1}{n+2} = \frac{n+1}{n+2}$$

QED

2.3 Dimostrare $\sum_{i=0}^{n} i^2 = \frac{n\,(n+1)\,(2n+1)}{6}$

Dimostrare per induzione che $\sum_{i=0}^{n} i^2 = \frac{n\,(n+1)\,(2n+1)}{6}$.

Risposta

L'**induzione** è una tecnica di dimostrazione utilizzata per dimostrare la validità di una tesi grazie alla verifica di due condizioni: la validità della "base induttiva" e la validità del "passo induttivo".

Caso base. Scelgo come caso base il valore $n = 0$. La sua applicazione:

$$\begin{aligned} \sum_{i=0}^{n} i^2 &= \frac{n\,(n+1)\,(2n+1)}{6} \\ &\downarrow \quad \text{sostituisco } n = 0 \\ \sum_{i=0}^{0} 0^2 &= \frac{0\,(0+1)\,(2\cdot 0+1)}{6} \\ 0 &= 0 \end{aligned}$$

QED

Ipotesi induttiva: assumo che per $\forall n$ sia sempre vera la seguente proprietà:

$$\sum_{i=0}^{n} i^2 = \frac{n\,(n+1)\,(2n+1)}{6}$$

Passo induttivo. Si dimostra che l'ipotesi induttiva implica la veridicità della proprietà per $n+1$:

$$\begin{aligned} \sum_{i=0}^{n} i^2 &= \frac{n\,(n+1)\,(2n+1)}{6} \\ &\downarrow \quad \text{applico il passo induttivo} \\ \sum_{i=0}^{n+1} i^2 &= \frac{(n+1)\,((n+1)+1)\,(2\,(n+1)+1)}{6} \\ \sum_{i=0}^{n} i^2 + (n+1)^2 &= \frac{(n+1)\,((n+1)+1)\,(2\,(n+1)+1)}{6} \\ &\downarrow \quad \text{applico l'ipotesi induttiva} \\ \frac{n\,(n+1)\,(2n+1)}{6} + (n+1)^2 &= \frac{(n+1)\,((n+1)+1)\,(2\,(n+1)+1)}{6} \\ \frac{n\,(n+1)\,(2n+1) + 6\,(n+1)^2}{6} &= \frac{(n+1)\,((n+1)+1)\,(2\,(n+1)+1)}{6} \end{aligned}$$

$$
\begin{aligned}
2n^3 + n^2 + 2n^2 + n + 6n^2 + 12n + 6 &= \left(n^2 + 3n + 2\right)(2n + 3) \\
2n^3 + 9n^2 + 13n + 6 &= 2n^3 + 9n^2 + 13n + 6
\end{aligned}
$$

QED

2.4 Dimostrare $\forall n \in \mathbb{N}.\, n > 2$ si ha che $n^2 > 2n + 1$

Dimostrare per induzione che $\forall n \in \mathbb{N}.\, n > 2$ ***si ha che*** $n^2 > 2n + 1$ **.**

Risposta[1]

L'**induzione** è una tecnica di dimostrazione utilizzata per dimostrare la validità di una tesi tramite la verifica di due condizioni: la validità del "passo base" e la validità del "passo induttivo".

Caso base. Si sceglie come caso base $n = 3$, rispettando la condizione $n > 2$:

$$\begin{aligned} n^2 &> 2n + 1 \\ &\downarrow \text{ sostituisco } n = 3 \\ 3^2 &> 2 \cdot 3 + 1 \\ 9 &> 7 \end{aligned}$$

QED

Ipotesi induttiva: si assume per vero che $\forall n \in \mathbb{N}. n > 2$ allora si ha $n^2 > 2n+1$ o equivalentemente, con qualche manipolazione algebrica $n^2 - 2n - 1 > 0$.

Passo induttivo. Si dimostra che l'ipotesi induttiva implica la veridicità della proprietà per $n + 1$:

$$\begin{aligned} n^2 &> 2n + 1 \\ &\downarrow \text{ applico il passo induttivo} \\ (n+1)^2 &> 2(n+1) + 1 \\ n^2 + 2n + 1 &> 2n + 2 + 1 \\ n^2 + 2n + 1 - 2n - 2 - 1 &> 0 \\ n^2 - 2n - 1 + 2n + 1 - 2 &> 0 \\ n^2 - 2n - 1 + 2n - 1 &> 0 \\ &\downarrow \text{ applico l'ipotesi induttiva} \\ n^2 - 2n - 1 &> 0 + 2n - 1 > 0 \\ &\downarrow \text{ per } n > 2 \Rightarrow 2n - 1 \text{ sarà sempre positiva e } > 0 \\ n^2 - 2n - 1 &> 0 \end{aligned}$$

QED

[1] Fonte soluzione: YouTube Link

3 Esercizio 3 - Scoping statico e dinamico

3.1 Tipologia codice 1

Si consideri il programma sulla destra. Si dica cosa viene calcolato dall'assegnamento in caso di scoping statico (mostrando come vengono calcolati i link statici e come vengono risolti riferimenti non locali) e in caso di scoping dinamico (mostrando l'evoluzione della tabella centrale dei riferimenti (CRT) e come vengono risolti riferimenti non locali).

```
{int a=2;
void pippo(){
    a = a + 1;}
void pluto(){
    a = 1;
    void minnie(int b){
        int a = 2*b;
        pippo();}
    pippo();
    minnie(3);}
a=3;
pluto();}
```

Risposta

Esistono **due tipologie** di scope:

- Scope **statico**, in cui un **nome non locale è risolto nel blocco** che testualmente lo racchiude;
- Scope **dinamico**, in cui un **nome non locale è risolto nella chiamata** attivata **più di recente** e **non ancora terminata**.

La **tabella centrale dei riferimenti** si riferisce allo scope dinamico. Quest'ultimo, a livello pratico, può essere implementano con una tecnica chiamata Shallow Access, nella quale le variabili locali sono poste in delle strutture dati centrali. Ed ecco che entra in gioco la CRT, una tabella con le seguenti **caratteristiche**:

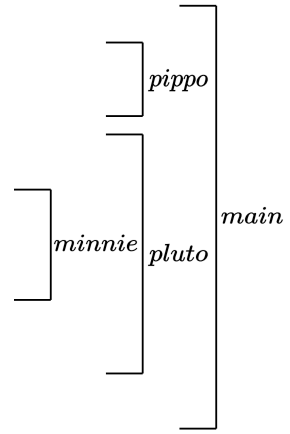
- Una **entry per ogni variabile** nel programma;
- Ogni entry ha un **puntatore ad una lista di elementi**;
- Ogni lista di elementi contiene **informazioni** necessarie per **accedere** ad un eventuale **ambiente di riferimento**.

Si inizia con l'analisi del codice e l'applicazione dello scoping statico/dinamico.

Scoping statico

Passo 1: si calcola la profondità del codice dato. Si indica con Sd la profondità statica, ovvero la profondità di annidamento della definizione della procedura corrispondente al RdA (record di attivazione, i.e. stack/pila).

```
{int a=2;
void pippo(){
    a = a + 1;}
void pluto(){
    a = 1;
    void minnie(int b){
        int a = 2*b;
        pippo();}
    pippo();
    minnie(3);}
a=3;
pluto();}
```



$$\begin{aligned} Sd\,(main) &= 0 \\ Sd\,(pippo) &= 1 \\ Sd\,(pluto) &= 1 \\ Sd\,(minnie) &= 2 \end{aligned}$$

Passo 2: si procede con l'esecuzione del codice, calcolando, ad ogni chiamata, i link statici e mostrando come vengono risolti i riferimenti non locali. Si ricorda che la formula per calcolare i link statici è:

$$k = Sd(Ch) - Sd(P) + 1$$

Un link dinamico punta sempre al chiamante, così da formare la catena dinamica, mentre un link statico dipende dalla formula soprastante.

La profondità statica (Sd, *static deep*) del chiamante Ch, meno la profondità statica del chiamato, cioè la procedura P, più uno. Inoltre, la formula per calcolare i riferimenti non locali è:

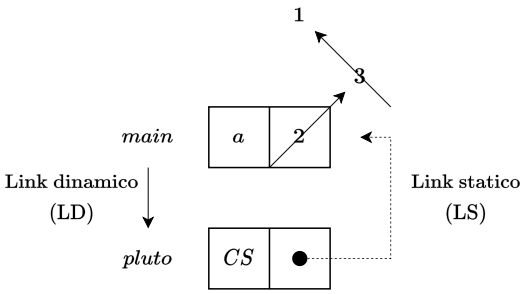
$$N = Sd(P) - Sd(D)$$

Numero di volte necessario per risalire la catena statica è dato dalla differenza tra la profondità statica della procedura in cui viene utilizzato l'identificatore ($Sd(P)$), meno la profondità statica della procedura D che definisce tale identificatore ($Sd(D)$).

Infine, con CS si indica l'abbreviazione di Catena Statica, ovvero la catena di link statici che collegano le istanze del RdA.

1. **main** invoca la funzione **pluto**.

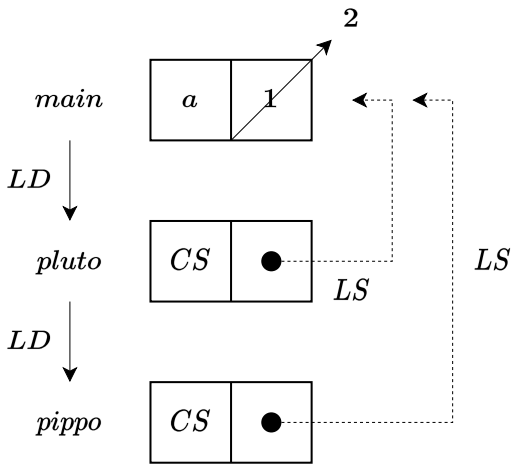
$$\begin{aligned} \text{Link statico} &\rightarrow Sd(main) - Sd(pluto) + 1 = 0 - 1 + 1 = 0 \\ &\Rightarrow CS\,(pluto) = indirizzo\,(main) \\ \text{Riferimento non locale } a &\rightarrow N = Sd(pluto) - Sd(main) = 1 - 0 = 1 \end{aligned}$$



Evoluzione della memoria.

2. **pluto** invoca la funzione **pippo**.

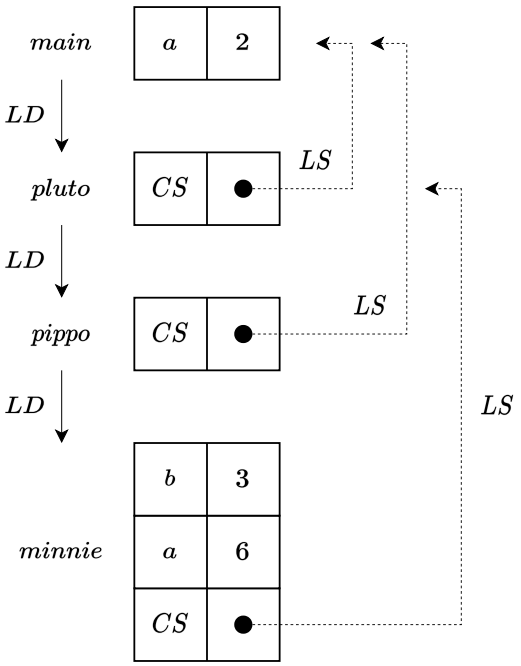
$$\begin{aligned}
\text{Link statico} \quad &\rightarrow \quad Sd(pluto) - Sd(pippo) + 1 = 1 - 1 + 1 = 1 \\
&\Rightarrow \quad CS(pippo) = indirizzo(CS(pluto)) \\
&\qquad\qquad = indrizzo(main) \\
\text{Riferimento non locale } a \quad &\rightarrow \quad Sd(pippo) - Sd(main) = 1 - 0 = 1
\end{aligned}$$



Evoluzione della memoria.

3. **pluto** invoca la funzione **minnie**.

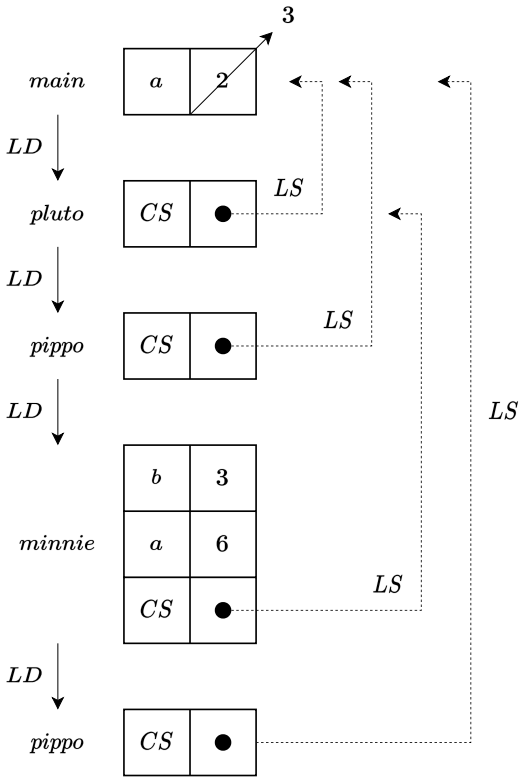
$$\begin{aligned} \text{Link statico} &\rightarrow Sd(pluto) - Sd(minnie) + 1 = 1 - 2 + 1 = 0 \\ &\Rightarrow CS(minnie) = indirizzo(pluto) \\ \text{Riferimento non locale} &\rightarrow \text{nessuno} \end{aligned}$$



Evoluzione della memoria.

4. **minnie** invoca la funzione **pippo**.

$$\begin{aligned}
\text{Link statico} \quad &\rightarrow \quad Sd(minnie) - Sd(pippo) + 1 = 2 - 1 + 1 = 2 \\
&\Rightarrow \quad CS(pippo) = indirizzo(CS(CS(minnie))) \\
&\qquad\qquad = indirizzo(CS(pluto)) \\
&\qquad\qquad = indirizzo(main) \\
\text{Riferimento non locale } a \quad &\rightarrow \quad N = Sd(pippo) - Sd(main) = 1 - 0 = 1
\end{aligned}$$

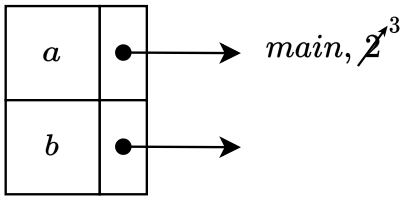


Evoluzione della memoria.

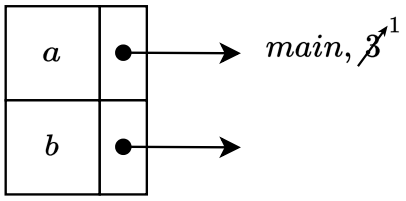
Scope dinamico

Ogni aggiornamento della tabella, ovvero ogni aggiornamento della lista di elementi per gli identificatori, viene eseguita inserendo in cima l'ultima dichiarazione eseguita.

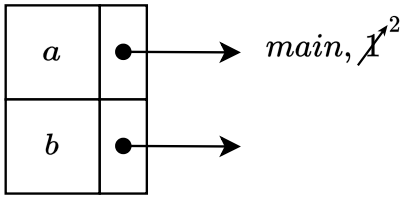
Chiamata principale del *main*. Variabile inizializzata a 2 e poi aggiornata al valore 3:



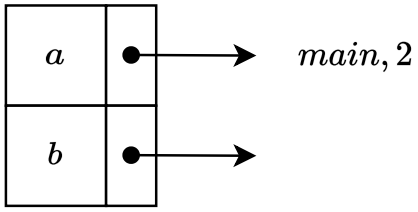
main invoca pluto. Aggiornamento del valore della variabile a con il valore 3:



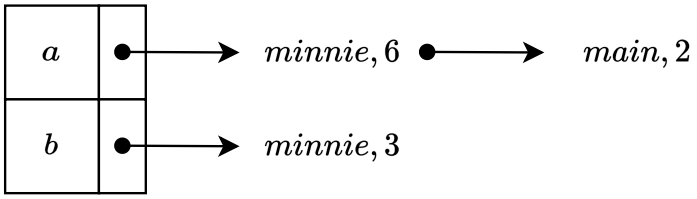
pluto invoca pippo. Incremento di 1 del valore della variabile a, quindi 1+1 = 2:



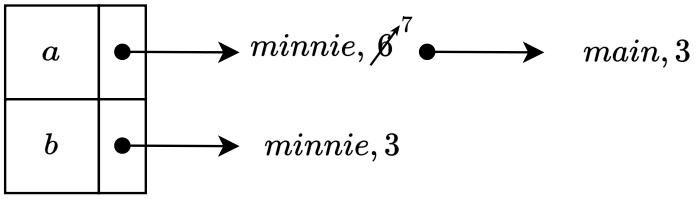
Ritorno da **pippo**:



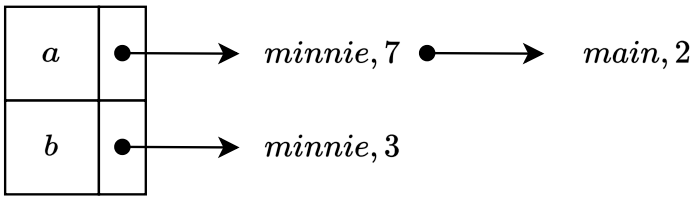
pluto invoca minnie. Il parametro b assume il valore 3, viene aggiornata la lista degli elementi per l'identificatore a inserendo la moltiplicazione tra b e 2, cioè $2 \times b = 2 \times 3 = 6$:



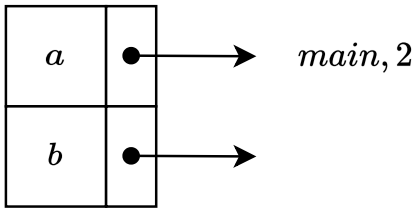
minnie invoca pippo. Ancora una volta, *pippo* incrementa la variabile a, ma questa volta si riferisce a *minnie*, a causa dello scoping dinamico (shallow binding):



Ritorno da **pippo**:



Ritorno da **minnie**. Vengono rimossi gli elementi a, b che si riferiscono a *minnie* poiché la chiamata è stata conclusa:



La CRT dopo il ritorno da **pluto** è identico alla figura soprastante.

3.2 Tipologia codice 2

3.3 Tipologia codice 3

3.4 Tipologia codice 4

3.5 Tipologia codice 5

3.6 Tipologia codice 6

3.7 Tipologia codice 7

3.8 Tipologia codice 8

3.9 Tipologia codice 9

Si consideri il programma sulla destra. Si dica cosa viene calcolato dall'assegnamento in caso di scoping statico (mostrando come vengono calcolati i link statici e come vengono risolti riferimenti non locali) e in caso di scoping dinamico (mostrando l'evoluzione della tabella centrale dei riferimenti (CRT) e come vengono risolti riferimenti non locali).

```
{int a = 3; int b = 5;
void fun(){
    int z = a * b;}
void fun2(){
    int a = 4; int c = 7;
    void fun3(){
        int a = c - b;
        fun();}
    b = a + b;
    fun3();}
fun2();}
```

Risposta

Esistono due tipologie di scoping:

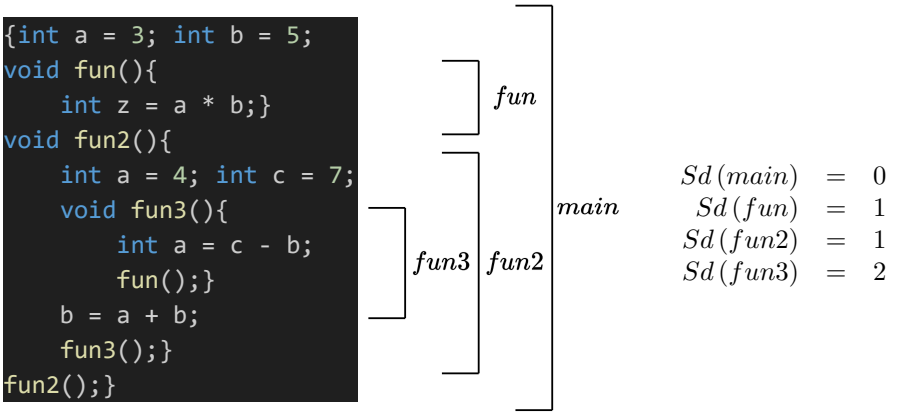
- Statico, in cui un nome non locale è risolto nel blocco che lo racchiude;
- Dinamico, in cui un nome non locale è risolto nella chiamata attivata più di recente e non ancora terminata.

La tabella centrale dei riferimenti (CRT) si riferisce allo scope dinamico. Quest'ultimo può essere implementato, a livello pratico, tramite una tecnica chiamata Shallow Access. Essa consente di avere una struttura dati centrale contenente le variabili locali. Dunque, la tabella CRT ha tre caratteristiche principali:

- Ogni entry corrisponde ad una variabile nel programma;
- Ogni entry ha un puntatore ad una lista di elementi;
- Ogni lista di elementi contiene informazioni necessarie ad accedere ad un eventuale ambiente di riferimento

Scoping statico

Passo 1: si calcola la profondità statica (Sd, Static deep) di ciascuna funzione presente all'interno del programma. La profondità statica indica la profondità di annidamento della definizione della procedura corrispondente al Record di Attivazione (RdA), i.e. Stack/Pila).



Passo 2: si inizia ad analizzare il codice. Come richiesto dall'esercizio, si espone il calcolo dei link statici:

$$k = Sd(Ch) - Sd(P) + 1$$

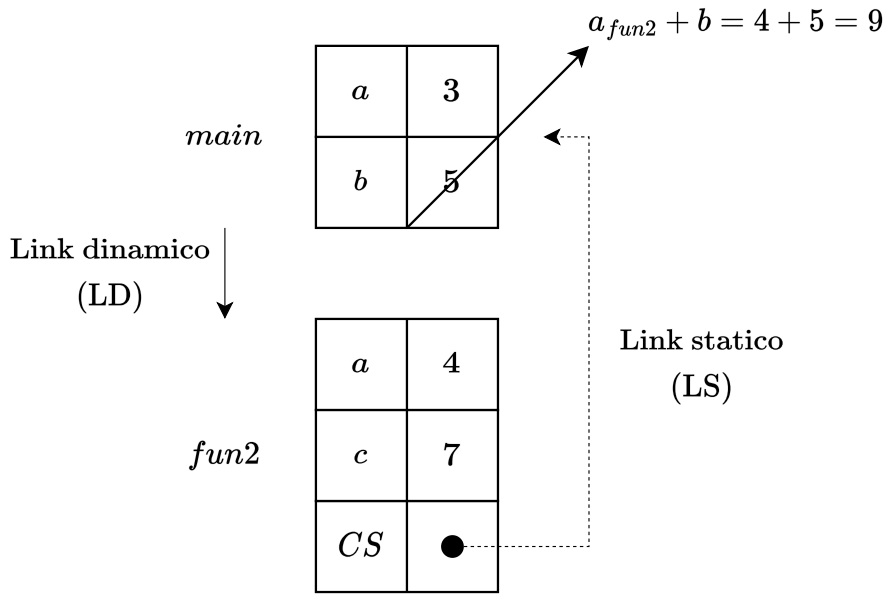
Con $Sd(Ch)$ che rappresenta la profondità statica del chiamante, $Sd(P)$ la profondità statica del programma invocato e k la profondità (def. di *static deep*). L'insieme dei link statici forma una catena statica (CS), ovvero un collegamento delle varie istanze del RdA. Al contrario, i link dinamici (DL, *dynamic link*) puntano sempre al chiamante e formano la catena dinamica;
Il calcolo di eventuali riferimenti non locali:

$$N = Sd(P) - Sd(D)$$

Con $Sd(P)$ che rappresenta la profondità statica della procedura che sta utilizzando il riferimento non locale, $Sd(D)$ che rappresenta la profondità statica della procedura in cui è stato definito il riferimento, N che rappresenta il numero di volte necessarie per risalire la catena statica.

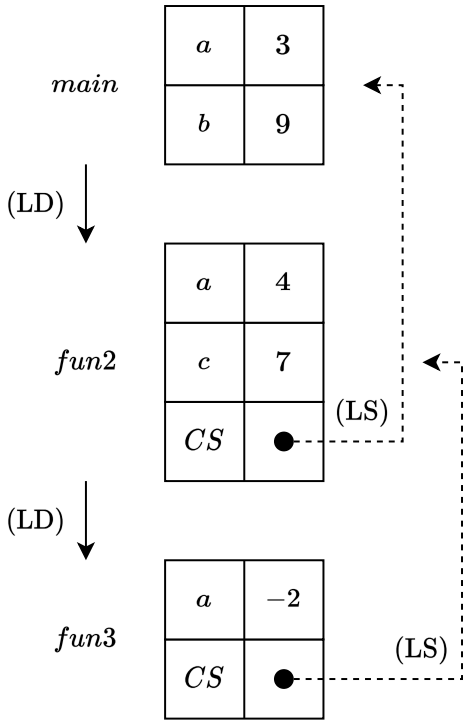
main invoca la funzione **fun2**.

$$\begin{aligned} \text{Link statico} &\rightarrow Sd(main) - Sd(fun2) + 1 = 0 - 1 + 1 = 0 \\ &\Rightarrow CS(fun2) = indirizzo(main) \\ \text{Riferimento non locale } b &\rightarrow N = Sd(fun2) - Sd(main) = 1 - 0 = 1 \end{aligned}$$



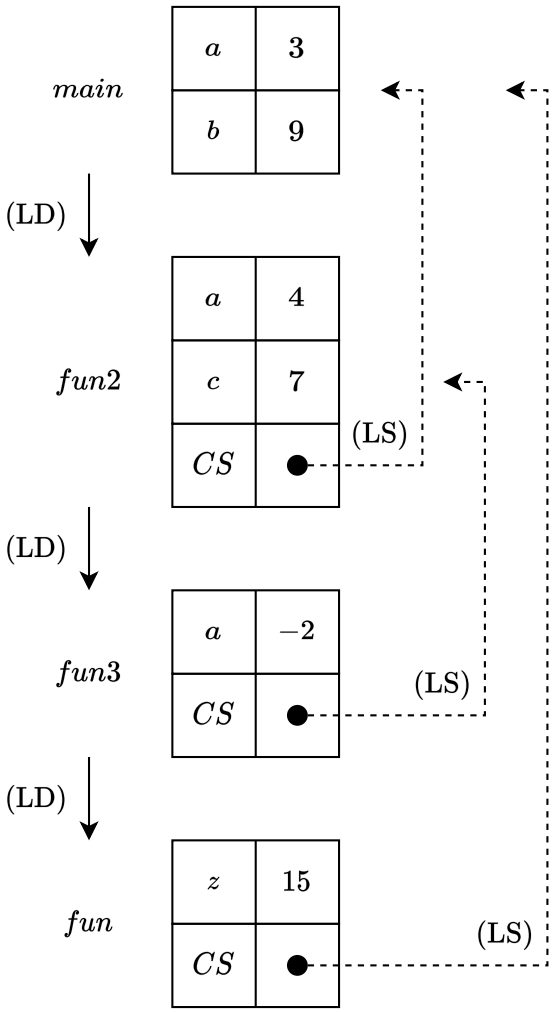
fun2 invoca la funzione **fun3**.

$$
\begin{array}{rcl}
\text{Link statico} & \rightarrow & Sd(fun2) - Sd(fun3) + 1 = 1 - 2 + 1 = 0 \\
& \Rightarrow & CS(fun3) = indirizzo(fun2) \\
\text{Riferimento non locale } c & \rightarrow & N = Sd(fun3) - Sd(fun2) = 2 - 1 = 1 \\
\text{Riferimento non locale } b & \rightarrow & N = Sd(fun3) - Sd(main) = 2 - 0 = 2
\end{array}
$$



fun3 invoca la funzione **fun**.

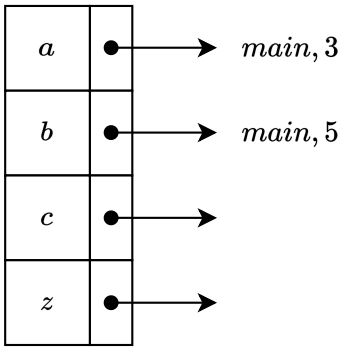
$$
\begin{aligned}
\text{Link statico} \quad &\rightarrow \quad Sd(fun3) - Sd(fun) + 1 = 2 - 1 + 1 = 2 \\
&\Rightarrow \quad CS(fun) = indirizzo(CS(CS(fun3))) \\
&\qquad\qquad = indirizzo(CS(fun2)) \\
&\qquad\qquad = indirizzo(main) \\
\text{Riferimento non locale } a \quad &\rightarrow \quad N = Sd(fun) - Sd(main) = 1 - 0 = 1 \\
\text{Riferimento non locale } b \quad &\rightarrow \quad N = Sd(fun) - Sd(main) = 1 - 0 = 1
\end{aligned}
$$



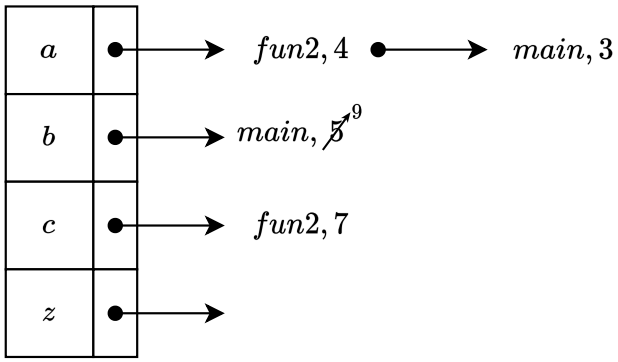
Scoping dinamico

Ogni aggiornamento della tabella CRT, ovvero ogni aggiornamento della lista di elementi degli identificatori, viene effettuata inserendo in cima l'ultima dichiarazione eseguita.

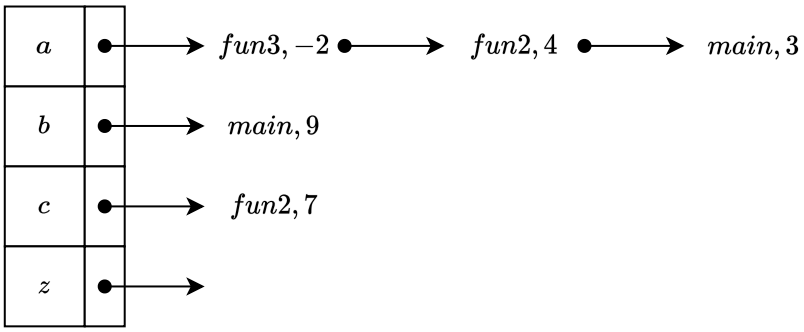
Chiamata principale del main. Inizializzazione delle variabili a, b:



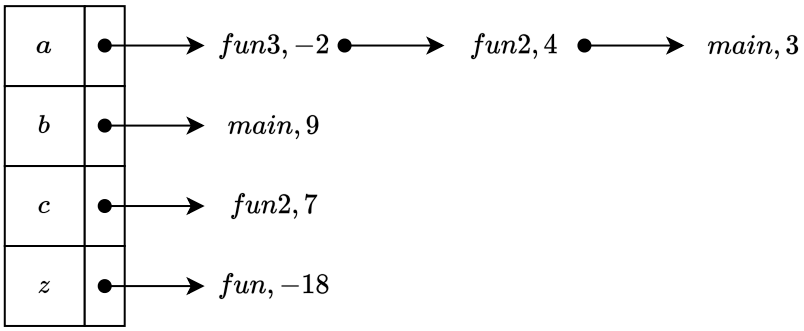
main invoca fun2. Vengono inizializzate due variabili locali, quindi si aggiungono alla lista degli elementi dei rispettivi identificatori. Inoltre, la variabile b, che è stata dichiarata nel *main*, viene aumentata di 4:



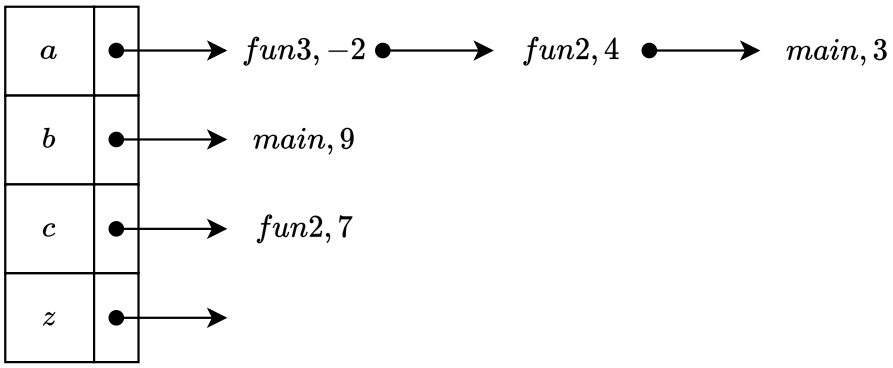
fun2 invoca fun3. Viene inizializzata una variabile locale, quindi si aggiunge alla lista degli elementi dell'identificatore a:



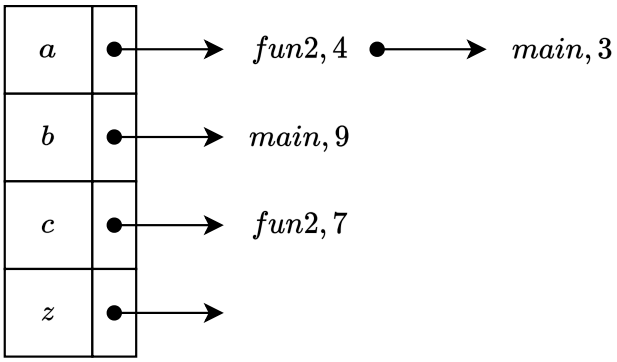
fun3 invoca fun. Viene inizializzata una variabile locale, quindi si aggiunge alla lista degli elementi dell'identificatore z:



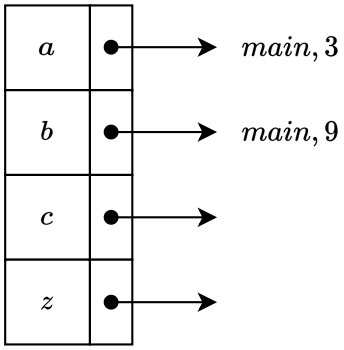
Ritorno di fun in fun3. La lista degli elementi dell'identificatore z viene aggiornata eliminando l'elemento riferito alla procedura fun (a causa della sua terminazione):



Ritorno di fun3 in fun2. La lista degli elementi dell'identificatore a viene aggiornata eliminando l'elemento riferito alla procedura $fun3$ (sempre a causa della terminazione):



Ritorno di fun2 in main. Si libera a, c:



4 Esercizio 4 - Scoping (statico/dinamico) e Binding

4.1 Regole di scoping e di binding

Spiegare dove e perché nel programma è necessario parlare di regole di scoping, e dove (e perché) anche di regole di binding. Si dica quale risulta essere il valore finale di z nelle varie situazioni, ovvero con: (1) scoping statico; (2) scoping dinamico e deep-binding; (3) shallow-binding (e scoping dinamico).

```
1  {int a = 4; int b = 3;
2  int pippo (int in){
3      return a + b - in;
4  }
5  int pluto(int f(int b)){
6      int b = 5; int a = 6;
7      return f(b) + a;
8  }
9  {int b = 2;
10 int z = pluto(pippo);}
11 }
```

Risposta

In questo caso, ma vale in generale, le regole di scope e di binding sono necessarie quando la procedura contiene un ambiente non locale. Precisamente:

- Le regole di scoping poiché è necessario risolvere riferimenti non locali;
- Le regole di binding poiché le chiamate a funzione hanno tra i parametri delle funzioni.

Ovviamente la premessa è che le procedure abbiano ambienti non locali.

La definizione di *scoping* è stata data in modo esaustivo nell'esercizio 3 (capitolo 3).

Con *binding* si intende l'associazione tra un nome e un valore. Quest'ultimo non deve per forza essere un valore numerico, ma può anche essere una stringa, una funzione, una lista, ecc.

- Deep-binding, viene utilizzato l'ambiente valido al momento della definizione del legame tra parametro attuale e formale;
- Shallow-binding, gli ambienti da tenere in considerazione sono due:
  - L'ambiente valido al momento della creazione del legame tra parametro attuale e parametro formale;
  - L'ambiente valido al momento della chiamata del parametro attuale attraverso il parametro formale.

Il parametro formale è un parametro noto nella definizione della funzione. Invece, il parametro attuale è quello che viene passato effettivamente alla funzione quando viene invocata.

Scoping statico

All'invocazione di $f\,(b)$ (riga 6), la funzione *pluto* passa come parametro la sua variabile locale b, ovvero 5. La funzione f, parametro formale, è legata al parametro attuale *pippo*. Quindi, nel corpo di *pippo* viene fatto il calcolo:

$$a + b - in$$

Dove a si riferisce alla a del *main*, ovvero la variabile a riga 1 con valore 4; dove b si riferisce alla b del *main*, ovvero la variabile a riga 1 con valore 3; dove in si riferisce al parametro formale, quindi corrisponde a 5, ovvero il valore che gli è stato passato.

$$4 + 3 - 5 = 2$$

Il valore 2 viene ritornato. Quindi, alla riga 6 viene effettuata la somma tra 2 e la variabile locale a. Il risultato che viene ritornato dalla funzione *pluto*, che corrisponde al risultato della variabile z, è $2 + 6 = 8$.

Scoping dinamico e deep-binding

All'invocazione della funzione $f(b)$ (riga 6), la funzione *pluto* passa come parametro la sua variabile locale b, ovvero 5. La funzione f, parametro formale, è legata al parametro attuale *pippo*. Quindi, nel corpo di *pippo* viene fatto il calcolo:

$$a + b - in$$

- a considera l'ambiente al momento della definizione. Quindi, si prende il valore 4 del main;
- b sempre considerando l'ambiente al momento della definizione, si prende il valore 2;
- in è sempre 5.

$$4 + 2 - 5 = 1$$

Quindi, viene tornato il valore 1 alla funzione *pluto*. Quest'ultima ritorna la somma tra 1 e la variabile locale a, cioè $1 + 6 = 7$. Quindi il valore finale di z è 7.

Scoping dinamico e shallow-binding

Il procedimento è lo stesso dei precedenti. L'unica differenza è nel calcolo che esegue la funzione *pippo*, che ovviamente si ripercuote sul valore ritornato dalla funzione *pluto*.

$$a + b - in$$

Viene preso il valore 6 per a, quello della funzione *pluto*; viene preso il valore 5 per b, quello della funzione *pluto*; viene preso il valore 5 per in, il valore passato come parametro da *pluto*.

$$6 + 5 - 5 = 6$$

Il valore di z sarà:

$$6 + 6 = 12$$

Poiché si prende a come variabile locale.

4.2 Codice da inserire in caso di scoping statico/dinamico

4.2.1 Tipologia di codice 1

4.2.2 Tipologia di codice 2

Si consideri lo schema del seguente codice, nel quale vi sono due buchi indicati rispettivamente con (*) e (**). Si dia il codice da inserire al posto di (*) e (**) in modo tale che:

(a) ***Se il linguaggio usato adotta lo scope statico, le due chiamate alla procedura pluto assegnino a x lo stesso valore;***

(b) ***Se il linguaggio usato adotta lo scope dinamico, le due chiamate alla procedura pluto assegnino a x valori diversi.***

Si descrivano dettagliatamente le scelte fatte e si descriva cosa avviene a tempo di esecuzione in entrambi i casi e perché.

```
{ int i; int x := 3;
(*)
void pippo(){
    int y;
    (**);
    x=pluto()+x; i=i+1;
    }
while(i <= 1){pippo();}}
```

Risposta

La differenza tra scoping statico e scoping dinamico è stata data nell'esercizio 3. Si procede dunque con la soluzione dei due codici nelle due casistiche.

Si riscrive il codice per avere una spaziatura migliore:

```
{
    int i; int x := 3;
    (*)
    void pippo(){
        int y;
        (**);
        x = pluto() + x;
        i = i + 1;
    }
    while (i <= 1)
    {
        pippo();
    }
}
```

- (*) il codice da inserire è:

```
i = 0;
int y = 2;
void pluto(){
  return y;
}
```

- (**) il codice da inserire è:

```
y = i;
x = 3;
```

Quindi, il codice finale diventa:

```
1  {
2      int i; int x := 3;
3      int y = 2;
4      void pluto(){
5          return y;
6      }
7      void pippo(){
8          int y;
9          y = i;
10         x = 3;
11         x = pluto() + x;
12         i = i + 1
13     }
14     while (i <= 1)
15     {
16         pippo();
17     }
18 }
```

Alla partenza del codice, viene creata una variabile y e gli viene assegnato il valore 2. Alla chiamata della funzione *pippo*, viene dichiarata un'altra variabile y e gli viene assegnato il valore di i che inizialmente è 0 e poi 1. Alla variabile x, presente nel *main* viene assegnato il valore 3. Successivamente, viene chiamata la funzione *pluto* e qua i risultati sono differenti:

- Scope statico:
  - $i = 0$: la funzione *pluto* ritorna il valore 2, ovvero quello del *main*. Quindi la x diventa $2 + 3 = 5$.
  - $i = 1$: la funzione *pluto* ritorna nuovamente il valore 2 e la somma è di nuovo 5 poiché la x con il ciclo $i = 0$ è stata aggiornata a 5, ma con il ciclo $i = 1$ è ritornata a 3 grazie alla dichiarazione a riga 10.
- Scope dinamico:
  - $i = 0$: la funzione *pluto* ritorna il valore 0, ovvero quello uguale alla variabile i. Quindi la x diventa $0 + 3 = 3$.
  - $i = 1$: la funzione *pluto* ritorna il valore 1, ovvero quello uguale alla variabile i. Quindi la x diventa $1 + 3 = 4$. Il fatto che la x non sia stata aggiornata è dovuta allo stesso motivo spiegato nello scope statico.

4.2.3 Tipologia di codice 3

5 Esercizio 5 - Ricorsione e passaggio di parametri

5.1 Ricorsione e ricorsione in coda

Definire brevemente il concetto di ricorsione e di ricorsione in coda. Si consideri il programma ricorsivo a destra e si descriva che cosa calcola mostrando anche il calcolo sulla lista $(2, 4, 6)$***. Trasformare quindi tale programma in un programma ricorsivo in coda. Mostrare poi il funzionamento dell'algoritmo fornito sulla lista*** $(2, 4, 6)$***.***

NOTA: Data una lista list, length(list) restituisce la sua lunghezza, car(list) restituisce il primo elemento, cdr(list) restituisce tutta la lista tranne il primo elemento, concat concatena due liste (se un elemento non è una lista, concat lo converte prima in lista). () è la lista vuota.

```
lista function (lista list){
    if(length(list) = 0) then return();
    else return concat(function(cdr(list)),car(list) * 3)
}
```

Risposta

Una funzione viene definita **ricorsiva** se viene definita in termini di se stessa. Le funzioni ricorsive derivano direttamente dalle definizioni di induzione, tuttavia rimangono concetti diversi:

- L'induzione è un concetto matematico ed è sempre ben fondata;
- L'induzione "informatica" può divergere. Ovvero, un programma potrebbe non terminare mai (i.e. `while(true){x=x}`).

Una chiamata di una funzione g in una funzione f viene definita "in coda", se la funzione f restituisce il valore restituito dalla funzione g senza ulteriore computazione. Se ne deduce, che una funzione ricorsiva viene chiamata "**funzione ricorsiva in coda**" quando contiene solo chiamate ricorsive in coda.

Calcolo ricorsione

Considerando la lista $(2, 4, 6)$, la funzione ricorsiva esegue le seguenti operazioni:

$$
\begin{array}{lll}
 & & = (18, 12, 6) \\
function(2, 4, 6) & \longrightarrow \quad concat\,((4, 6), 2 * 3)\,; & \uparrow 7° \\
 & 1° \downarrow & (18, 12) \\
 & concat\,((6), 4 * 3)\,; & \uparrow 6° \\
 & 2° \downarrow & \uparrow (18) \\
 & concat\,((), 6 * 3)\,; & \uparrow 5° \\
 & 3° \downarrow & \uparrow () \\
 & return(); & \uparrow 4°
\end{array}
$$

L'**obbiettivo** della funzione ricorsiva è invertire l'ordine della lista e moltiplicare ciascun valore per 3. In caso di lista vuota, il risultato è una lista vuota.

Trasformazione da ricorsione a ricorsione in coda

La trasformazione da ricorsione a ricorsione in coda può avvenire se si definisce una funzione che restituisce il valore di un'altra funzione senza altre computazioni. Quindi, si definisce la funzione $functionrc$ che esegue le stesse cose della funzione $function$ originale, ma con un parametro in più, ovvero il risultato parziale. Questo consente di non elaborare valori dopo le chiamate ricorsive. Mentre la funzione $function$ (nuova, non l'originale) che prende come parametro una lista e ritorna come risultato la computazione della ricorsione senza eseguire ulteriori operazioni:

```
lista functionrc (lista list, lista res){
    if(length(list) = 0) then return res;
    else return functionrc(cdr(list), concat(car(list) * 3), res)
}

lista function (lista list){
    return functionrc(list, ());
}
```

Considerando la lista $(2, 4, 6)$, la ricorsione in coda esegue le seguenti operazioni:

$$
\begin{array}{lll}
 & & = (18, 12, 6) \\
function((2,4,6),()) & \longrightarrow & functionrc\,((2,4,6),())\,; \quad \uparrow 9^\circ \\
 & & 1^\circ \downarrow () \qquad\qquad\qquad\quad (18, 12, 6) \\
 & & functionrc\,((4,6), 2*3)\,; \quad \uparrow 8^\circ \\
 & & 2^\circ \downarrow (6) \qquad\qquad\qquad\quad \uparrow (18, 12, 6) \\
 & & functionrc\,((6), 4*3)\,; \quad \uparrow 7^\circ \\
 & & 3^\circ \downarrow (12, 6) \qquad\qquad\quad \uparrow (18, 12, 6) \\
 & & functionrc\,((), 6*3)\,; \quad \uparrow 6^\circ \\
 & & 4^\circ \downarrow (18, 12, 6) \qquad\quad \uparrow (18, 12, 6) \\
 & & returnres; \qquad\qquad\qquad\quad \uparrow 5^\circ
\end{array}
$$