

Lettura 2

Modello imperativo: è formato da un insieme di contenuti chiavi (variabili) avendo un indirizzo.

Queste variabili arrivano con valore.

conceptualmente, la memoria ne dà uno spazio di locazioni ad uno spazio di valori.

mem: Indirizzo: Valori / mem (loc) = "valore contenuto nella locazione loc"

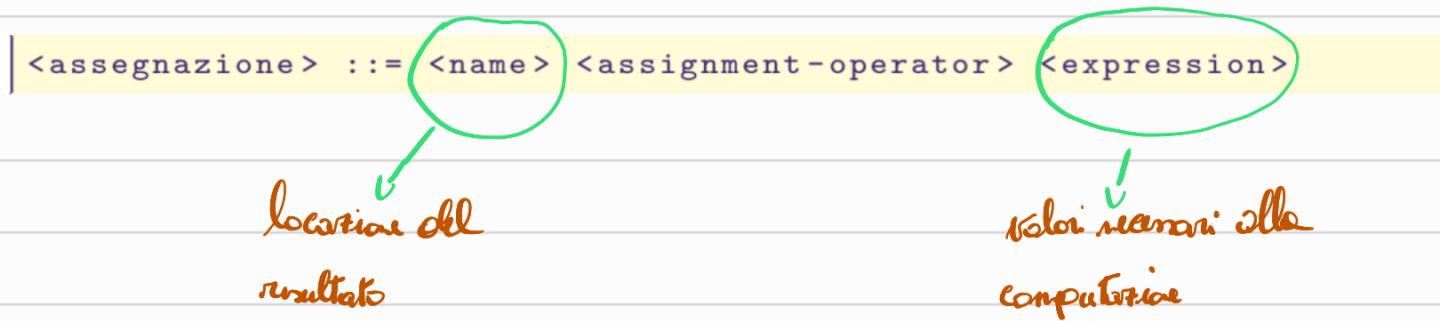
Accenniamo alle grammatiche in forma Backus-Naur \rightarrow Ricordando come avviene la derivazione per le CFG, la BNF è un formalismo usato per descrivere le sintassi dei LDP

Esempio:

CFG \Rightarrow $exp \rightarrow exp + exp$ probabile derivation

LDP \Rightarrow $exp = exp + exp$

Assegnazioni, definizione grammaticale (nei linguaggi imperativi)



Modello imperativo → molto pensano alle CPU moderne
↳ uso di nomi come adresses di indirizzi

I programmi non capiscono memoria dell'elaboratore ed ogni volta d'esecuzione ha a fare:

- 1 ottienimento locazione di operandi e risultato
- 2 " dati di operandi dalla locazione operandi
- 3 valuta il risultato
- 4 memorizzare risultato in locazione risultato

Un arretramento

Ambiente (di esecuzione) → È un set di nomi (var., funz...) associati a qualcosa da cui se può conoscere un valore.

Concretamente, l'ambiente è un funz. che sa da un set di nomi e corrisponde a ???
 $\text{env}(\text{id}) = \text{???}$
↳ dipende dal paradigma

Esempio di ambiente: Imperativo

"scrivo" associa gli id a loc. di memoria, a loro volta associate al contenuto in memoria (funzione mem)

int: id → address

↳ $\text{env}(x) = \text{indirizzo della variabile di nome } x$

↳ immutabile finché x esiste

Esempio di assegnazione: *funzione*

Il valore associato ad un nome non contiene nel tempo, pertanto non considerare gli indirizzi.

env : id \rightarrow value

Nome della variabile $x = \text{env}(x)$

Non si può mettere le memorie in questo paradosso

Esempio: *assegnazione*

$x = x + 1$

// la x di $x = x + 1$ ha le locazioni associate al nome, $\text{env}(x)$



// la x di $\text{env}(x)$ è il valore delle variabili, $\text{mem}(\text{env}(x))$

In $\text{env}(x)$ si trova $\text{mem}(\text{env}(x)) + 1$

Esercizi fatti in classe \rightarrow Seguendo le seguenti espressioni in C

int a[3] = {4, 5, 6};

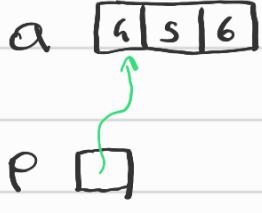
Traduci i seguenti assegnamenti in mem ed env

int *p, *q;

In "indirizzo" STORE "value"

① $a[2] = a[2] + 1 \Rightarrow \text{In env}(a) + 2 \text{ STORE mem}(\text{env}(a) + 2) + 1$

② $p = a$ In questo tempo uso il memory layout



In $\text{env}(p)$ STORE $\text{env}(a)$
oppure solo gli indirizzi

Converso al contrario

① In $\text{mem}(\text{env}(p))$ STORE 0 $\Rightarrow \text{int } *p = 0$

③ In $\text{mem}(\text{env}(p)) + 1$ STORE 0 $\Rightarrow *(p+1) = 0$ oppure $p[1] = 0$

Esercizi fatti a casa

③ $a[0] = a[0] + p[1] \Rightarrow \text{in env}(a) + 0 \text{ STORE } \text{mem}(\text{env}(a) + 0) + \text{mem}(\text{env}(p) + 1)$

④ $q = p \Rightarrow \text{in env}(q) \text{ STORE env}(p)$

⑤ $q = &a[1] \Rightarrow \text{in env}(q) \text{ STORE env}(a) + 1$

⑥ $q = p + 2 \Rightarrow \text{in env}(q) \text{ STORE env}(p) + 2$

⑦ In $\text{env}(a) + 1$ store $\text{mem}(\text{env}(a)) \Rightarrow a[1] = a[0]$

④ In $\text{mem}(\text{env}(p)) + \text{mem}(\text{env}(a))$ store $\text{mem}(\text{mem}(\text{env}(p)))$

$$\Rightarrow {}^*(p+a) = {}^{**}p$$

Data object e legami: ogni variabile, parametro, oggetto... può essere rappresentato con un data object

↳ quadrupla (L, N, V, T) :

- Locazione;
- Nome;
- Valore;
- Tipo;

Legame → (binding) è la determinazione di una delle componenti
(locazione, nome, valore, tipo)

Nel paradigma imperativo, dato un data object (L, N, V, T) :

- legame di locazione vale $L = \text{env}(N)$
- " " valore vale $V = \text{mem}(\text{env}(N))$

Riabilitazione di legami:

- ① Compile Time
- ② Load Time
- ③ run-Time

- Legame di locazione → durante il caricamento in memoria o a run-time
- Legame di nome → durante la compilazione, quando il compilatore incontra una dichiarazione
- Legame di tipo → durante la compilazione, quando il compilatore incontra una dichiarazione di tipo

Legame di tipo :

- È collegato al legame di valore (tipo di variabile e tipo del suo valore devono corrispondere)
- Quando il legame di valore viene modificato, occorre effettuare il type checking
- Un LSP è **dinamicamente tipizzato** se il legame (e le sue variazioni) è controllato durante l'esecuzione, avviengono durante la esecuzione
- Un LSP è **staticamente tipizzato** se il legame durante la compilazione e il controllo di coerenza avviene durante la compilazione e l'esecuzione

Type checking → controllo della coerenza dei legami valore-tipo .

Avvio e chiusura :

(Un LSP è **strettamente tipizzato** ⇒ il type checking)

- avviare sempre (come in Java)
- a) durante la compilazione;
 - b) durante l'esecuzione;
 - c) più non avviene

Un CSP è delsolmente tipizzato se il type checking può non avvenire.

E' delsolmente tipizzato:

- ① Supporta il casting, facendo l'interpretazione di un valore in un altro tipo;
- ② Supporta conversioni implicite, senza cost;
- ③ Supporta le anonimie, riassegna le locazioni di variabili di tipo
altrimenti, evitando il controllo fra valore e tipo

Linguaggi perfetti → Linguaggi in cui il type checking avviene durante la compilazione, e il compilatore genera meno errori del necessario.

(P sarebbe strettamente e delamente tipizzato) → Non può esistere
↳ il più forte in assoluto

Esempio → Se (P esistesse), allora il compilatore è capace di scoprire le caratteristiche della funzione:
estensione:

int x;

P; → P programma genico, e non termina, non mi accorgo dell'errore di x

$x = "pippo";$ andando contro l'ipotesi

Alcuni controlli sono impossibili a tempo di compilazione, e la strategia migliore da applicare per linguaggi **fortemente e staticamente tipati** è il controllo continuo e compile-time e il resto a run-time.

Blochi di istruzione: Scopo / Ambito di validità di un legame

Susseguenti enunciati del programma in cui quel legame è valido. Un legame è delimitato dai **blocchi di codice**.

I blocchi servono per capire dove finisce un pezzo di codice

```
public class Conto { // blocco
    private int x;

    public void preleva(int importo) { // blocco
        if (x >= importo) { // blocco
            x -= importo;
        } else { // blocco
            throw new IllegalArgumentException();
        }
    }
}
```

Prologo - linguaggio di blocchi di codice → Un blocco contiene due parti

- Sezione dichiarazione nomi
- Vincitori per la validità dei legami di nome

```
...
BLOCK A;
DECLARE I;
BEGIN A
...
END A;
...
{I DEL BLOCCO A}
```

Ambito di validità di legami

- Scopo statico o **lexicale**, si verifica lo scopo partendo da un blocco annidato e, eventualmente, si punta al globale
- Scopo dinamico, da capire meglio

Overloading → avviene quando un variabile viene mappata in un blocco interno con lo stesso nome. **Le nuove maschere le ricchie**

Legame di locazione e scopo :

- Stabilire un legame di locazione = **Allocare memoria**
- Una variabile in scopo ha un legame di locazione. **Vivente**, un legame di loc. può essere cattivo se la variabile è mascherata

Scopo di una var. = Sottinsieme dello scopo del suo legame di loc.

Allocazione della memoria :

- Statica → indirizzo fisso a load-time e valido per l'intera esecuzione

- Dinamica :

Su stack → " " a run-time e dura quanto un blocco invia fino alla sua terminazione (Record di attivazione - RDA)
↳ ha le info necessarie per l'esecuzione del blocco e post-block

Su heap → " " a run-time e viene rilasciato su richiesta o automaticamente (garbage collection)

RDA di un blocco anonimo o in-line :

Contiene :

- Puntatore di entità dinamica ; con un blocco più complesso, anche l'RDA diventa più complesso
- Ambiente locale .

Stack di basezione → contiene gli RDA "attivi" :

① Il top contiene l'RDA in esecuzione ;

② Al cambio di blocco, si crea un RDA e viene pushato nello stack ;

③ Quando il blocco termina, si fa cero pgo

Contenuto dello
stack di attivazione

```
PROGRAM P;  
    DECLARE I,J;  
BEGIN P  
    BLOCK A;  
        DECLARE I,K;  
    BEGIN A  
        BLOCK B;  
            DECLARE I,L;  
        BEGIN B  
            ...           {I e L da B, K da A, J da P}  
        END B;  
            ...           {I e K da A, J da P}  
    END A;  
    BLOCK C;  
        DECLARE I,N;  
    BEGIN C  
        ...           {I e N da C, J da P}  
    END C;  
    ...           {I e J da P}  
END P;
```

I, J
null

P

