Pseudocodice e problemi elementari

corso di laurea in **Matematica**

Informatica Generale, Lezione **1(b)**

Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

Pseudocodice

Descriveremo gli algoritmi usando un linguaggio **semi-formale** (detto **pseudocodice**). Userò uno `pseudo-Python`.

Questo linguaggio può descrivere algoritmi in modo che un **esecutore** (**automa**) possa eseguirli e fornire dei risultati, in modo **deterministico** (cioè senza ambiguità).

Questo linguaggio permette di descrivere algoritmi che sono **sequenze di comandi** (C).

I **comandi modificano la memoria**, e il loro effetto dipende dalla valutazione di **espressioni**.

Ci sono comandi che permettono di **fare scelte** (sulla base della valutazione di espressioni) e di **ripetere** altri comandi fino al verificarsi di una condizione.

Cominciamo con uno pseudocodice (`Tiny-Python`) che descrive le azioni di **automi semplicissimi** con un insieme limitato azioni elementari.

Espressioni

Consideriamo all'inizio due tipi espressioni: **espressioni aritmetiche** (denotate da *exp*, *e*, e_1, e_2) ed **espressioni booleane** (denotate da *bexp*, *b*, b_1, b_2).

All'inizio, considereremo esecutori **capaci solo di sommare e sottrarre 1**, per cui le **espressioni aritmetiche** possono contenere solo **nomi di variabili**, costanti numeriche, +1 e -1.

Le espressioni **valutano** a un valore, che all'inizio sarà per noi un **numero naturale arbitrariamente grande** o **True/False**.

Una **variabile** valuta al **valore contenuto** nell'area di **memoria** associata alla variabile.

Le **espressioni booleane** sono nella forma exp_1==exp_2 (test di uguaglianza) oppure nella forma exp_1!=exp_2 (test di non uguaglianza) dove exp_1 ed exp_2 sono espressioni aritmetiche.

Esempi: espressioni aritmetiche: `x+1`, `m+1+1`, `n-1`
espressioni booleane: `x!=y+1`, `y-1==x`

Comandi: assegnazione

Il comando principale è l'**assegnazione**, che permette di modificare il valore di una variabile con il valore ottenuto dalla valutazione di un'espressione:

$$x = exp$$

La cui **semantica** è:

"Valuta il valore dell'espressione exp e scrivi il valore calcolato nell'area di memoria associato alla variabile x*".*

È bene stigmatizzare che nel comando:

$$x = x + 1$$

a **sinistra** x indica **l'area di memoria** ad essa associato (le caselle del foglio di carta) detto **left value**, mentre a **destra** indica il **valore contenuto** in quell'area, detto **right value**.

Questa operazione è **tutt'altro che commutativa**! Alcuni usano ← oppure **:=** per indicarne la direzionalità.

Comandi: assegnazione parallela

Risulta spesso comodo usare **l'assegnamento parallelo**, proposto di E. W. Dijkstra negli anni '70, e implementato solo di recente nel linguaggio Python:

$$x_1,\ x_2,\ \dots,\ x_n = exp_1,\ exp_2,\ \dots,\ exp_n$$

dove tutte le espressioni vengono valutate **prima di modificare i valori delle variabili** $x_1,\ x_2,\ \dots,\ x_n$.

Osservate che $x_1,\ x_2 = exp_1,\ exp_2$ **non è equivalente** alla sequenza di assegnazioni $x_1 = exp_1$ e poi $x_2 = exp_2$ perché la modifica di x_1 nel primo assegnamento può influenzare la valutazione dell'espressione exp_2.

Gli assegnamenti vengono fatti **in ordine**, per cui se programmate in Python, ci possono essere interferenze e i seguenti due comandi **non sono equivalenti**:

```
v[i], i = x , i+1          i, v[i] = i+1, x
```

Assegnamento parallelo: scambio

Poniamoci il problema di **scambiare il contenuto** di due variabili, ***x*** ed ***y***.

Siccome l'**assegnazione** è **un'operazione distruttiva**, la sequenza di comandi:

$$x = y$$
$$y = x$$

non raggiunge l'effetto voluto. È necessario **salvare** il valore di ***x*** in **una terza variabile** prima di assegnarla (un po' come deve fare l'oste disonesto che vuole scambiare il contenuto di una bottiglia di vino buono con quella di vino meno buono):

$$h = x$$
$$x = y$$
$$y = h$$

Con l'assegnamento parallelo è sufficiente:

$$x, y = y, x$$

Comandi: sequenza, selezione e iterazione

I comandi possono venire messi in **sequenza**: C_1; C_2

La sequenza di comandi può essere alterata da:

- **Selezione**: `if b:` C_1 `else:` C_2 che ha la semantica: "*valuta l'espressione b, se il risultato è True esegui* C_1 *altrimenti esegui* C_2".
- **Ripetizione**: `while b: C` che ha la semantica: "*continua a eseguire il comando C finché l'espressione booleana b valuta a* TRUE".

`b` è detta **guardia** del ciclo e `C` è detto **corpo** del ciclo.

Esempio: Assumendo che *x* e *y* contengano i nostri valori da sommare, il seguente frammento di pseudocodice:

```
while n ≠ 0: m, n = m+1, n-1
```

descrive l'algoritmo di somma unaria tra due numeri naturali visto prima, per un esecutore capace di **sommare** e **sottrarre 1** (risultato nella variabile `m`).

Funzioni

Possiamo aumentare le capacità del nostro esecutore, definendo **funzioni**: una funzione è **l'astrazione di un'espressione** e (per ora) serve quindi a **calcolare un valore**, che necessita di una sequenza non elementare di passi.

Esempio: funzione che calcola la somma, per l'esecutore che sa eseguire solo +1:

Una funzione ha dei **parametri** che permettono di **generalizzare** il comportamento della funzione su un insieme **virtualmente illimitato** di possibili valori.

Il comando `return exp` specifica il valore tornato dalla funzione e **ne sospende l'esecuzione**!

```
def piu(m, n):
    while n!=0:
        m, n = m+1, n-1
    return m
```

Funzioni: uso

Una volta che abbiamo definito una funzione, possiamo immaginare che il nostro esecutore abbia **una nuova capacità elementare**.

Quindi, tra le espressioni, considereremo la possibilità di chiamare una funzione già definita.

Esempio: vediamo come sia possibile definire il prodotto per il nostro automa che sa solo fare +1 e -1: iteriamo la somma, usando una **variabile accumulatore** `p`.

➧ **Esercizio:** scrivere la funzione prodotto senza utilizzare la funzione `somma`. Quali sono le difficoltà?

```
def per(m, n):
    p = 0;
    while n!=0:
        p, n = piu(p, m), n-1
    return p
```

Funzioni: memoria

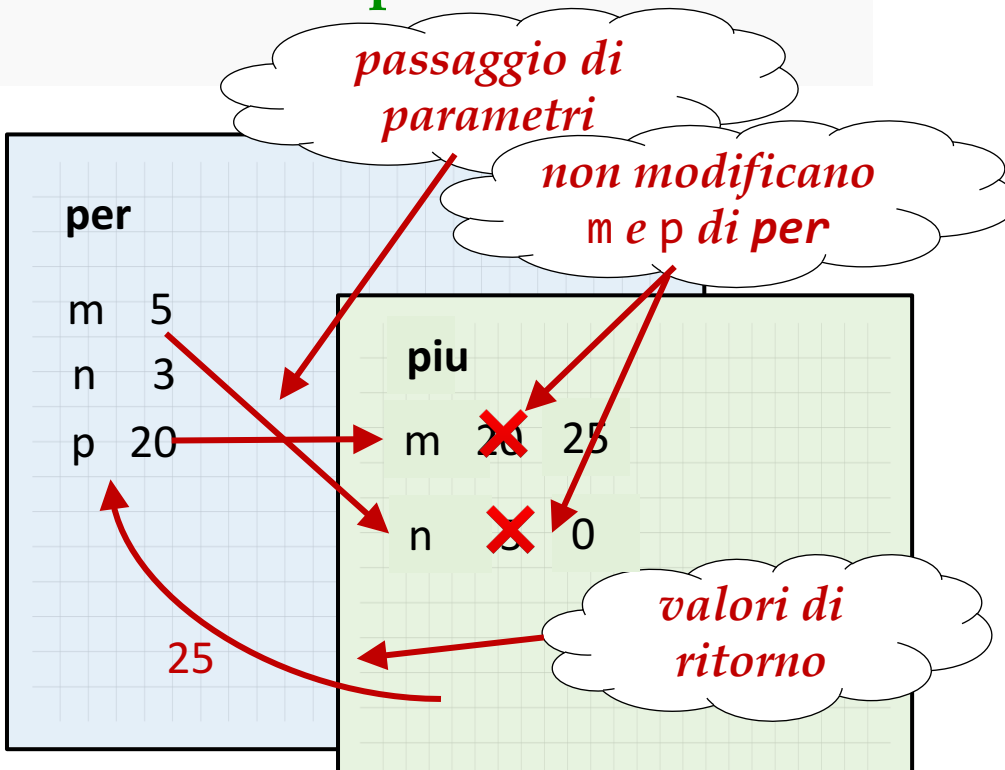
Ogni **funzione** ha la sua area di **memoria locale** contenente i parametri e le variabili locali, come ognuna avesse il **suo foglio di carta privato**.

Modifiche alle **variabili locali non influenzano** il valore di variabili in altre funzioni (magari con **lo stesso nome**).

Le funzioni **comunicano valori** attraverso i **parametri** e i **valori di ritorno**.

```
def per(m, n):
    p = 0;
    while n!=0:
        p, n = piu(p,m), n-1
    return p
```

```
def piu(m, n):
    while n!=0:
        m, n = m+1, n-1
    return m
```



Esempio: funzioni booleane

Abbiamo visto alcune funzioni numeriche, vediamo la **funzione booleana `and`**.

In questo caso, `a` e `b` sono due variabili che contengono valori booleani, mentre `True` e `False` sono le **costanti booleane**.

Vediamo anche le sorelline **`not`** e **`or`**.

```
def and(a, b):
    if a: return b
        else: return False
```

```
def not(a):
    if a: return False
        else: return True
```

```
def or(a, b):
    if a: return True
        else: return b
```

Operatori di confronto

Abbiamo visto che possiamo verificare se due espressioni sono uguali o diverse. Ma possiamo definire gli **operatori di confronto** <, >, ≤, ≥, etc? Ovviamente sì…

Idea: decrementare **finché** (**almeno**) una non **diventa 0**…

Ricordiamo che il **`return`** sospende l'esecuzione di una funzione (**non serve l'else**). Quindi anche...

- **Esercizio**: definire le altre funzioni di confronto.

```
def minUg(m, n):
    while and(m!=0, n!=0):
        m, n = m-1, n-1
    if m==0: return True
    return False
```

```
def minUg(m, n):
    while True:
        if m==0: return True
        if n==0: return False
        m, n = m-1, n-1
```

ciclo infinito. Si esce per via dei `return`

Il linguaggio visto finora è minimale?

Lo studente attento si sarà accorto che **l'operatore `!=` non è necessario** in quanto l'espressione booleana e_1`!=`e_2 è chiaramente **equivalente a** `not(`e_1`==`e_2`)`.

In realtà, anche il **-1 non è necessario**. Infatti posso scrivere una funzione che calcola -1, usando solo +1.

Idea: mantenere due variabili `i` e `j`, con `j` sempre un passo avanti a `i`. Quando `j` raggiunge il valore di `n`, i vale `n-1`.

Esercizio: Mostrare che il costrutto **`if`** `b:`C_1 **`else:`** C_2 **è "inutile"**

```
def pred(n):
    i, j = 0, 1
    while j!=n:
        i, j = i+1, j+1
    return i
```

```
def pred(n):
    j = 1
    while j+1 != n: j = j+1
    return j
```

si può fare con una sola variabile

Il futuro è passato qui…

Teorema [**BÖHM-JACOPINI**]: *Ogni funzione (**parziale**) **calcolabile** $f : \mathbb{N}^k \to \mathbb{N}$ può essere calcolata usando una macchina capace di:*

- ***sommare 1**,*
- ***verificare** se due **espressioni** siano **uguali o meno**,*
- *eseguire azioni in **sequenza**,*
- ***ripetere** azioni con il costrutto di controllo* `while B: C`.

SAPIENZA
UNIVERSITÀ DI ROMA