Specifiche e Asserzioni Logiche

corso di laurea in **Matematica**

Informatica Generale, **Ivano Salvo**

Lezione **2(a)**, **[29/9/23]**

SAPIENZA
UNIVERSITÀ DI ROMA

Terminazione

È facile scrivere programmi che **non terminano**.

La funzione ***beata*** non fa nulla per l'eternità!

La terminazione può occorrere in modi meno ovvi!

La funzione ***sisifo*** è condannata a incrementare e decrementare la variabile ***x*** per l'eternità!

corpo del ciclo vuoto

```
def beata():
    while True:
        return
```

```
def sisifo():
    x = 0
    while x==0:
        x = x + 1
        x = x - 1
    return x
```

Terminazione: predecessore

Cosa accade se effettuiamo la chiamata `pred(0)`?

La variabile `j` comincia l'iterazione con il valore 1 e crescerà ad ogni iterazione **non assumendo mai il valore di *n***, cioè 0.

L'esecuzione del ciclo **non termina mai** nel nostro mondo di **naturali illimitati** `pred` calcola una **funzione parziale**, non definita su 0.

- **Esercizio Pratico**: verificare cosa accade veramente in C.
- **Esercizio Pratico**: verificare cosa accade in Python.

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

Specifiche

Possiamo ritenere la funzione **`pred`** **corretta**?

Una funzione può essere corretta o meno solo **rispetto a una specifica**, cioè a una formula logica che ne descrive il comportamento atteso.

La funzione vista prima **è corretta rispetto alla specifica**:

$$pred(n) = \begin{cases} n-1 & \text{per } n > 0 \\ ? & \text{per } n = 0 \end{cases}$$

Anche **la seguente è corretta** rispetto a **questa specifica**.

```
def pred(int n):
    i = 0
    if n==0: return 42
    while i+1!=n:
        i = i+1
    return i
```

il valore su 0
non è specificato
Viviamo in un
mondo di
funzioni parziali

Asserzioni Logiche

Spesso scriveremo nei **commenti** allo pseudocodice **asserzioni logiche**: si tratta di formule logiche che dipendono dal valore delle variabili di programma e che possono esprimere:

- **precondizioni:** sono proprietà **richieste sui valori di ingresso** di una funzione e specificano l'insieme dei valori su cui la funzione svolge correttamente il suo compito;
- **postcondizioni:** sono proprietà che una porzione di codice o una funzione **assicura** sui **valori** delle variabili alla fine o sui **risultati che restituisce al chiamante** (ovviamente a patto che le precondizioni siano soddisfatte).

Precondizioni e postcondizioni formano le **specifiche**.

- **invarianti:** sono proprietà sempre soddisfatte durante l'esecuzione di un ciclo;
- **altre proprietà** (**terminazione**, proprietà soddisfatte in certi punti del programma).

Predecessore revisited

Le specifiche stabiliscono una sorta di **contratto** tra chi scrive la funzione e chi la usa.

Le precondizioni determinano le **assunzioni** sotto cui la funzione **garantisce risultati corretti**.

Il comportamento della funzione al di fuori delle precondizioni non è rilevante ed **è responsabilità del chiamante** verificare che i valori dei parametri rispettino le precondizioni.

```
def pred(int n):
# REQ: n > 0
# ENS: return n-1
    i = 0
    while i+1!=n:
        i = i+1
    return i
```

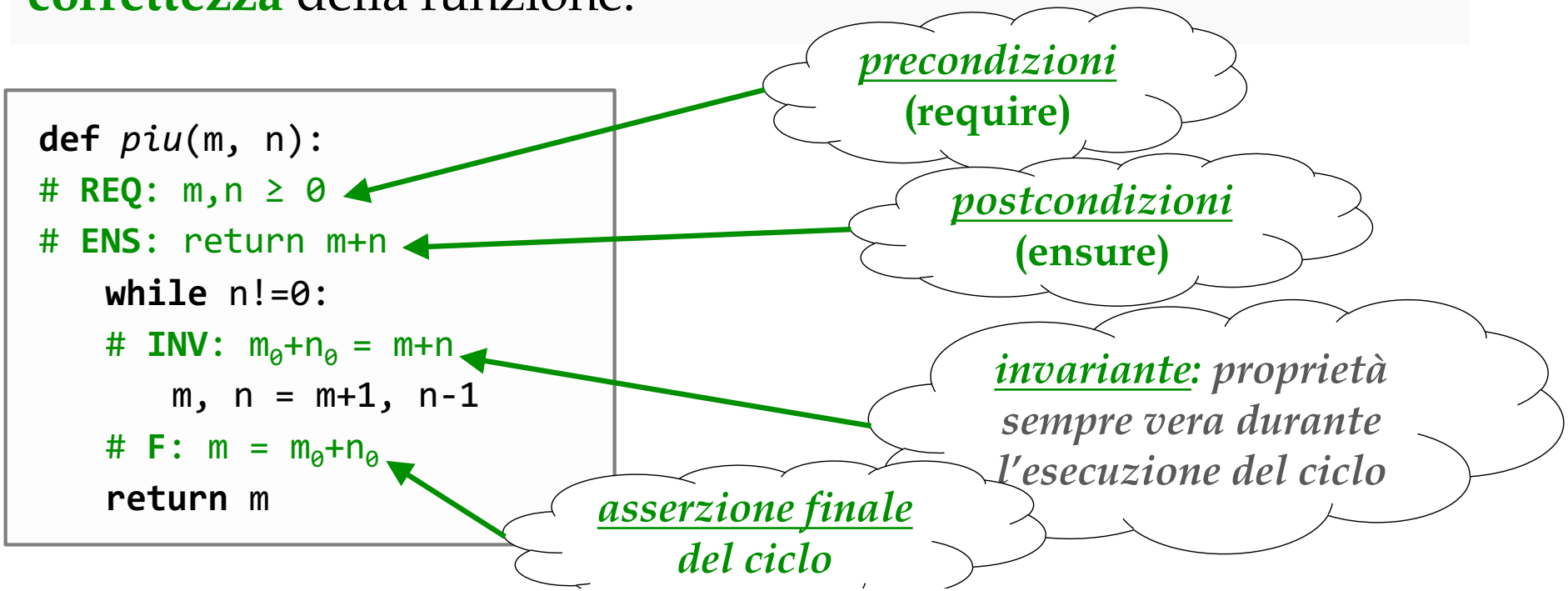
La non terminazione su 0 è "colpa" del chiamante che non rispetta il contratto.

Asserzioni Logiche: Esempio

Riscriviamo il programma della somma, scrivendo le **asserzioni logiche**. Ad esempio, cosa serve richiedere $n \geq 0$?

In generale, data una variabile x, indicheremo nelle asserzioni con x_0 il suo valore all'inizio della computazione (useremo il font x e x_0 nelle dimostrazioni per indicarne il **valore**.)

L'invariante $m_0+n_0=m+n$ e la negazione della guardia del while, $n=0$ implicano **l'asserzione finale F:** $m=m_0+n_0$ quindi la **correttezza** della funzione.



Terminazione

In generale, saremmo interessati in questo corso a funzioni la cui computazione **termina**.

Un comando iterativo potrebbe **non terminare**. Ad esempio: `while n!=0: n=n+1` entrando nel ciclo con `n` **positivo**.

Il programma della somma **non terminerebbe** se entrassimo nel ciclo con un valore di `n` **negativo**.

```
def piu(int m, int n):
# REQ: m,n ≥ 0
# ENS: return m+n
    while n!=0:
    # INV: m₀+n₀ = m+n
        m, n = m+1, n-1
    # F: m = m₀+n₀
    return m
```

Conservazione degli invarianti

Come dimostrare che una proprietà è invariante per un ciclo?

Si dimostra che `INV` **vale all'ingresso del ciclo**.

Si dimostra che **assumendo** `INV` e la **guardia** `B`, dopo l'esecuzione del corpo del ciclo **vale ancora** `INV`.

Chiamando x', y', … il valore delle variabili x, y, … dopo l'esecuzione del corpo del ciclo, facciamo vedere che

$$Inv[x, y, \ldots] \;\&\; B \text{ implica } Inv[x', y', \ldots]$$

```
def somma(int m, int n):
# REQ: m,n ≥ 0
# ENS: return m+n
    while n!=0:
    # INV: m₀+n₀ = m+n
        m, n = m+1, n-1
    # F: m = m₀+n₀
    return m
```

$m+n=m_0+n_0$ è banale per via delle (non) **inizializzazioni** ($m=m_0$ e $n=n_0$)

So che $m+n=m_0+n_0$, (**assumo che valga l'invariante per** m, n) allora anche $m'+n'=m_0+n_0$ (l'**invariante vale anche per** m', n'). Infatti:

$m'+n' = m+\cancel{1}+n-\cancel{1}$ (assegn. parall.)

$= m+n$

Dimostrare la Terminazione

Una tipica tecnica per dimostrare la terminazione di un ciclo, è definire una **funzione di terminazione**, cioè una funzione t con le seguenti proprietà:

- t dipende dal valore delle variabili di programma $x_1, .., x_n$
- l'**invariante** e la **guardia** implicano che t assume **valori positivi**, cioè vale sempre $t(x_1, .., x_n) > 0$
- t **decresce ad ogni iterazione**, cioè $t(x'_1, .., x'_n) < t(x_1, .., x_n)$

```
def somma(int m, int n):
# REQ: m,n ≥ 0
# ENS: return m+n
    while n!=0:
    # TERM: n
    # INV: m₀+n₀ = m+n
        m, n = m+1, n-1
    # F: m = m₀+n₀
    return m
```

Una possibile funzione di terminazione è semplicemente n.

La precondizione e la guardia del ciclo implicano l'invariante $n > 0$.

n decresce a ogni iterazione: siccome non ci sono **catenete infinite discendenti** nei naturali, esco dal ciclo dopo un numero finito di passi.

Teorema di Iterazione Finita

***Teorema**: Sia* `while B: C` *un comando iterativo e* φ *una asserzione logica. Allora, se:*

1. φ *vale all'**ingresso** del ciclo*
2. φ & `B` *implica che* φ *sia soddisfatta **dopo l'esecuzione** di* `C`
3. *Esiste una **funzione di terminazione** t*
4. φ & `not(B)` *implica* ψ

Allora l'asserzione ψ *è soddisfatta alla fine dell'esecuzione del ciclo* `while B: C`

Osservazione: 1. e **2**. implicano che φ sia una **proprietà invariante** per il ciclo `while B: C` mentre la proprietà **3**. implica che il ciclo **termina**.

Esempio: Predecessore

Vediamo che $i = j - 1$ e $j < n$ sono proprietà invarianti.

Ingresso del ciclo: $i = j - 1$ è banalmente vera perché $i = 0$ e $j = 1$.

$j < n$ è vera se $n \geq 1$ (**precondizione**). Infatti, siccome $j = 1$, in tal caso $j \leq n$ e si entra nel ciclo solo se $j \neq n$, quindi se $j < n$.

Conservazione: $i' = i+1 = j$ (*INV*) $= j' - 1$ perché $j' = j+1$. Se $j < n$ allora $j' = j+1 \leq n$. Ma se è soddisfatta la guardia $j' \neq n$ ciò implica $j' < n$.

Terminazione: $n - j$ descresce a ogni iterazione e sotto l'ipotesi $j < n$ è positiva. Infatti $n' - j' = n - j - 1 < n - j$. Questo implica che il numero di iterazioni è proprio $n - 1$ (**costo computazionale**).

```
def pred(n):
# REQ: n > 0
    i, j = 0, 1
    while j!=n:
        # INV: i=j-1 & j<n
        i, j = i+1, j+1
    # F: i=n-1
    return i
```

$j < n$ è necessario per provare la terminazione ed è implicato dalla precondizione $n > 0$

Sottrazione

Come la somma è iterazione di +1, anche la sottrazione può essere calcolata iterando -1.

Osservate che $m > n > 0$ implica che le **chiamate** alla funzione `pred` **soddisfano le precondizioni** di `pred` (parametro > 0).

Siccome nel nostro modesto modello di calcolo, `pred(n)` costa n operazioni, è possibile **evitare di chiamare** `pred`?

Contare in avanti come i negozianti quando danno il resto!

```
def sub(m, n):
# REQ: m ≥ n ≥ 0
# ENS: return m-n
    while n!=0:
    # INV: m₀-n₀=m-n, T=n
        m, n = pred(m), pred(n)
    # F: m = m₀-n₀
    return m
```

```
def sub(m, n):
# REQ: m ≥ n ≥ 0
# ENS: return m-n
    i = 0 # contapassi
    while n!=m:
    # INV: m₀-n₀=m-n+i, T=m-n
        n, i = n+1, i+1
    # F: m = m₀-n₀
    return i
```

positiva per le precondizioni

That's all Folks!

corso di laurea in **Matematica**

Informatica Generale, **Ivano Salvo**

Lezione **2(a)** [**29/9/23**]

SAPIENZA
UNIVERSITÀ DI ROMA