Tipi di Dato IV: Strutture Dati Generiche

Ivano Salvo
Sapienza Università di Roma

Anno Accademico 2011-12

1 Tipi di Dato Generici

Alcuni moderni linguaggi di programmazione permettono di scrivere programmi che sono parametrici rispetto ad un tipo di dato. Funzioni come `addHead`, `length` o `append` sulle liste non dipendono in nessun modo dal fatto che le liste considerate siano di interi. Sarebbe interessante poter scrivere il codice di queste funzioni e poterlo utilizzare su liste che contengono elementi di qualsiasi tipo: funzioni con queste caratteristiche vengono dette *polimorfe*. Il linguaggio C non supporta direttamente il polimorfismo, ma permette ugualmente di scrivere funzioni polimorfe usando uno stratagemma, la già discussa compatibilità del tipo `void *` con qualsiasi altro tipo puntatore.

D'altra parte, osserviamo, che esistono molte funzioni che sfruttano il tipo degli elementi: ad esempio, nelle liste ordinate utilizziamo l'operatore di minore, definito sugli interi (e su quasi tutti i tipi base), ma non necessariamente definito per un tipo definito dall'utente. Volendo scrivere le funzioni che definiscono il tipo di dato lista ordinata polimorfa (o generica), dovremo parametrizzare le funzioni rispetto a una relazione d'ordine. Non discuteremo in dettaglio questo caso, ma sappiate che è possibile anche in C passare una *funzione come parametro*.

Un'altra difficoltà da superare, operando con variabili definite `void *`, è che le variabili di tipo `void *` non si possono *dereferenziare*: infatti il compilatore, non conoscendo quanta memoria sia necessaria per memorizzare gli oggetti in questione, non è in grado di ricostruire il valore puntato, e quindi non è mai possibile applicare l'operatore di dereferenziazione `*` a variabili di tipo `void *`.

1.1 Pile generiche

In questa sezione vedremo, come definire un tipo di dato *pila* o *stack*. Una pila è una struttura dati sequenziale su cui si impone il vincolo che inserimenti, cancellazioni e letture possano avvenire *solo* in testa, o per meglio dire in *cima alla pila*. Il nome è chiaramente reminiscente di una pila di fogli o libri, di cui possiamo vedere o prendere solo il foglio in cima alla pila: la pila segue in sostanza una disciplina LIFO (*Last*

In, First Out), tipica tra l'altro di molti fenomeni legati alla sintassi dei linguaggi di programmazione e all'esecuzione dei programmi: per esempio, la prima parentesi chiusa si riferisce all'ultima parentesi aperta, oppure la prima attivazione ricorsiva di una funzione sarà l'ultima da cui si rientrerà. Vedremo infatti, come applicazione delle pile, l'*implementazione della ricorsione*, ossia una tecnica per trasformare un qualsiasi programma ricorsivo in iterativo (sez. 2).

Le pile possono essere viste come una sequenza, esattamente come le liste: ciò che le caratterizza però sono le *funzioni* che possono essere usate per manipolarle. La specifica del tipo di dato pila, infatti, non consiste solo di in una struttura dati adatta a memorizzare pile, quanto piuttosto in un *comportamento* definito dalle operazioni che permettono di accedere e manipolare una pila. Le operazioni seguono la disciplina di accesso fissata evitando accessi incoerenti alla struttura dati. Le operazioni che caraterrizzano il tipo di dato pila possono essere specificate come segue:

$$\mathsf{empty}() = \langle\,\rangle \qquad \mathsf{empty}_?(\langle\,\rangle) = \mathsf{true} \qquad \mathsf{empty}_?(a \cdot s) = \mathsf{false}$$
$$\mathsf{pop}(\langle\,\rangle) = \mathsf{error} \qquad \mathsf{pop}(a \cdot s) = s \qquad \mathsf{push}(a, s) = a \cdot s$$
$$\mathsf{top}(\langle\,\rangle) = \mathsf{error} \qquad \mathsf{top}(a \cdot s) = a$$

Abbiamo specificato la pila in modo che l'operazione pop si limiti a rimuovere l'elemento in cima alla pila, senza restituirne il valore. Per vedene il valore, è necessario preventivamente usare la funzione top. In letteratura, spesso viene considerata una funzione pop' che contemporaneamente modifica la pila e restituisce il valore che si trova in cima. Tuttavia la nostra specifica è comunque *completa*, cioè permette di eseguire qualunque sequenza di inserimenti/estrazioni/letture su una pila possibili con l'utilizzo di pop'.

Dato che una pila è essenzialmente una sequenza (di dimensione imprevedibile), l'implementazione più adatta fa uso di una lista. Memorizzeremo nella pila elementi di tipo `void *`. Quindi, se vorremmo memorizzare interi in una pila dovremmo passare puntatori a interi. Grazie alla compatibilità del tipo `void *` con ogni altro puntatore, potremmo memorizzare dati di qualsiasi tipo (e dimensione), semplicemente passando un puntatore a questi dati, e ciò permetterà di usare il tipo pila senza dover riscrivere il codice a seconda del tipo di dati che vogliamo mettere nella pila. Del resto, le operazioni di estrazione, lettura, cancellazione e inserimento non dipendono in nessun modo dal tipo di dato che memorizzo nella pila.

Rappresenteremo una pila come una lista concatenata, accessoriata di un nodo speciale che mantiene un puntatore alla cima della pila ed altri dati, come il numero di elementi (opzionale). Negli esercizi verrà chiesto al lettore di fornire implementazioni che usano array. Un significativo esercizio sperimentale è quello di verificare che se l'interfaccia delle operazioni rimane immutata, i programmi che usano le pile continuano a funzionare correttamente, indipendentemente da quale sia l'implementazione sottostante (quantomeno finchè non si supera la dimensione massima dell'array). Vediamo ora l'implementazione della struttura dati, mentre

```
stack createEmptyStack(){
   stack S;
   S = malloc(sizeof(stackDescriptor));
       /* la cima della pila viene inizializzata a NULL  */
   S->top = NULL;
       /* numero di elementi a 0  */
   S->numElem = 0;
   return S;
}

int isEmpty(stack S){
       /* equivalente a return S->top==NULL */
   return S->numElem==0;
   }

void pop(stack S){
   Snode *tmp;
       /* si mantiene l'invariante del tipo di dato */
   S->numElem--;
       /* salva il putatore al primo elemento per la free */
   tmp = S->top;
       /* modifica il puntatore al top della pila */
    S->top = (S->top)->next;
    free(tmp);
}

void* top(stack S){
   return (S->top)->elem;
}

void push(stack S, void* el){
   Snode *tmp;
      /* alloca memoria per inserire un nuovo nodo */
   tmp = malloc(sizeof(Qnode));
       /* il link del nuovo nodo punta al vecchio top */
   tmp->next = S->top;
   tmp->elem = el;
   S->top = tmp;
      /* si mantiene l'invariante del tipo di dato */
   S->numElem++;
}

int howManyStack(stack S){
   return S->numElem;
}
```

Figura 1: Definizione del tipo di dato Pile Generiche

l'implementazione delle operazioni è mostrato in Fig. 1. La cosa più rimarchevole da far notare al lettore è che la funzione `pop` *dealloca il nodo della coda*, ma **non dealloca** la memoria puntata da `val`, e del resto non potrebbe neanche farla, visto che `val` ha tipo `void *`: tale deallocazione, se desiderata, dovrà essere fatta da chi sta usando lo stack.

```
typedef struct S {
    void * val;
    struct S * next;
} Snode;

typedef struct {
    Snode* top;
    int numElem;
} stackDescriptor;

typedef stackDescriptor *stack;
```

1.2 Code generiche

Modificando la disciplina di ingresso/uscita e seguendo una politica FIFO (*first-in-first-out*) otteniamo un altro tipo di dato interessante: le *code*. Procederemo in modo del tutto analogo alle pile, prima dando la specifica funzionale e poi mostrando una possibile implementazione in linguaggio C basata su liste. Una tipica applicazione delle code è la *visita per livelli* di un albero. Le code si sarebbero potute convenientemente applicaree all'interno del programma che calcola i cammini minimi del cavallo (Homework 3), inserendo nella coda tutte le nuove caselle raggiunte dal cavallo, e andando poi a calcolare le successive partendo dalle caselle estratte dalla coda.

$$\mathsf{empty}() = \langle\,\rangle \quad \mathsf{empty}_?(\langle\,\rangle) = \mathsf{true} \qquad \mathsf{empty}_?(a \cdot s) = \mathsf{false}$$
$$\mathsf{enqueue}(a, a' \cdot s') = a' \cdot \mathsf{enqueue}(a, s') \qquad \mathsf{enqueue}(a, \langle\,\rangle) = \langle a \rangle$$
$$\mathsf{dequeue}(\langle\,\rangle) = \mathsf{error} \qquad \mathsf{dequeue}(a \cdot s) = s$$
$$\mathsf{first}(\langle\,\rangle) = \mathsf{error} \qquad \mathsf{first}(a \cdot s) = a$$

La specifica funzionale evidenzia che lunica vera differenza (ma che produce un comportamento totalmente diverso) è che l'inserimento di un nuovo elemento (enqueue) avviene in fondo alla lista che rappresenta la coda. Siccome tale inserimento non ha un costo costante, ma proporzionale alla lunghezza della lista, nell'implementazione conviene investire 4 byte per memorizzare un puntatore alla fine della lista e ottenere quindi una implementazione in tempo costante dell'operazione enqueue, al prezzo di un po' di attenzione nel maneggiare il puntatore di fine coda. Ecco quindi il tipo di dato necessario per memorizzare una coda, mentre in Fig. 2 è presentata limplementazione completa:

```
stack createEmptyQueue(){
   queue Q;
   Q = malloc(sizeof(queueDescriptor));
       /* il primo eleemnto della coda punta inizialmente a NULL  */
   Q->first = NULL;
   Q->last = NULL;
       /* numero di elementi a 0  */
   Q->numElem = 0;
   return Q;
}

int isEmpty(stack Q){
       /* equivalente a return Q->first==NULL
        * o a Q->last==NULL */
   return Q->numElem==0;
   }

void dequeue(queue Q){
   Qnode *tmp;
       /* si mantiene l'invariante del tipo di dato */
   Q->numElem--;
       /* salva il putatore al primo elemento per la free */
   tmp = Q->first;
       /* modifica il puntatore al first della pila */
    Q->first = (Q->first)->next;
    free(tmp);
}
void* first(queue Q){
   return (Q->first)->elem;
}

void enqueue(queue Q, void* el){
   Qnode *tmp;
      /* alloca memoria per inserire un nuovo nodo */
   tmp = malloc(sizeof(Qnode));
       /* il link del nuovo nodo punta al vecchio top */
   tmp->next = NULL;
   tmp->elem = el;
   if (Q->last) Q->last->next=tmp;
   Q->last=tmp;
   if (!Q->first) Q->first=inizio;
      /* si mantiene l'invariante del tipo di dato */
   Q->numElem++;
}
int howManyQueue(queue Q){
   return Q->numElem;
}
```

Figura 2: Definizione del tipo di dato Code Generiche

```
typedef struct Q {
    void *elem;
    struct Q *next;
} Qnode;

typedef struct {
   Qnode* first;
   Qnode* last;
   int numElem;
} queueDescriptor;

typedef queueDescriptor *queue;
```

2 ★ Implementazione della Ricorsione

Nella dispensa *Iterare è umano, ricorrere è divino* è presentato il problema della torre di Hanoi. In questa sezione, vedremo che quel problema *inerentemente ricorsivo*, ammette una soluzione iterativa basata sulla gestione *esplicita* da parte del programmatore della pila delle attivazioni ricorsive della procedure.

La tecnica che vedremo è assolutamente generale e può essere applicato per trasformare una qualsiasi procedura ricorsiva in iterativa. Di fatto, tale trasformazione genera un programma la cui esecuzione è del tutto analoga a quella del corrispondente programma ricorsivo: l'unico reale vantaggio nel fare questo sforzo di traduzione, oltre a capire a fondo il meccanismo computazionale sottostante all'esecuzione di un programma ricorsivo e a far vedere un'importante applicazione delle pile, è che potreste scrivere programmi ricorsivi che hanno una eccessiva profondità nella ricorsione e quindi producono facilmente errori di *stack overflow* o *segmentation fault*, perchè la quota di memoria destinata alla pila delle attivazioni ricorsive dal sistema è troppo piccola.

Inoltre, per far vedere l'utilità di una struttura dati generica, mostreremo anche una variante del problema della torre di hanoi, che mantiene anche lo *stato* del gioco (le tre pile di dischi appunto) ed ad ogni mossa lo stampa. Cominciamo con questa semplice modifica che richiede solo:

1. la definizione del tipo di dato `disco`;
2. la definizione e inizializzazione di un array di 3 pile di dischi;
3. la definizione di una funzione di stampa di una pila di dischi;
4. la modifica della funzione `move`, dimodochè piuttosto che stampare la mossa, aggiorni lo stato del gioco.

Un disco può essere rappresentato semplicemente con un numero intero. Se devo risolvere il problema della torre di Hanoi con n dischi, rappresento ciascun disco con un numero intero nell'insieme $[n]$ dei primi n numeri naturali strettamente positivi.

Cominciamo quindi con il vedere l'inizializzazione delle tre torri: ho un'array di stack, dove il primo conterrà gli n dischi (in ordine decrescente) e gli altri due sono vuoti. Osservate che gli interi da inserire nello stack *vanno allocati*. La semplice chiamata `push(torri[0], &i)` sarebbe scorretto perchè riempirebbe tutto lo stack con *lo stesso puntatore*, e quindi ogni modifica alla variabile `i`, modificherebbe, erroneamente i valori impilati nello stack.

```
void inizializzaTorri(stack torri[], int n){
    int *x;
    int i;

    for (i=0; i<3; i++)
        torri[i]=createEmptyStack();
    for (i=n; i>0; i--){
          x = malloc(sizeof(int));
          *x=i;
          push(torri[0], x);
        }
}
```

A questo punto, occorre scrivere una procedura per stampare lo stato del gioco. Ancora una volta decomponiamo il problema, scrivendo prima una procedura per stampare un singolo stack di interi, poi una procedura per stampare un'unica torre, e infine una procedura per stampare le 3 torri.

```
void printIntStack(Snode* N){
    int *x;

    if (N) {
       printIntStack(N->next);
       x = (N->elem);
       printf("%3d", *x);
       }
}

void printTorre(stack Q){
    printIntStack(top(Q));
    printf("\n");
}
void printTorri(stack torri[]){
   int i;

   for (i=0; i<3; i++){
       printf("%d : ",i);
       if (isEmptyStack(torri[i])) printf("-\n");
         else printTorre(torri[i]);}
       printf("\n");
}
```

Osserviamo che la funzione `printIntStack` dovrebbe, a rigore, essere a carico del programmatore che progetta la struttura dati `stack`. Tale procedura stampa la lista contenuta nello stack rovesciata (al ritorno dalle chiamate). Tuttavia è bene ricordare che per stampare le informazioni contenute nello stack dovete conoscerne il tipo.

La funzione `move` va semplicemente modificata in modo che tolga un disco dalla torre che rappresenta il piolo sorgente e lo infili nel piolo destinazione. Osservate che rispetto alla vecchia `move`, devo passare un ulteriore parametro che riferisce allo stato del gioco (`torri`).

```
void move(int sorg, int dest, stack torri[]){
    push(torri[dest], top(torri[sorg]));
    pop(torri[sorg]);
    printTorri(torri);
}
```

Veniamo ora allo scopo principale della sezione. Per prima cosa, dobbiamo definire un tipo di dato adatto a contenere lo stato locale di una attivazione della funzione `hanoi`. Vi ricordo che questo comprende *parametri*, *variabili locali* e il *punto di ritorno*. Rivediamo il codice della funzione:

```
void hanoi(int sorg, int aux, int dest, int n, stack torri[]){
/* inizio */
   if (n==1) move(sorg, dest, torri);
      else {
         hanoi(sorg, dest, aux, n-1, torri);
/* mezzo */
         move(sorg, dest, torri);
         hanoi(aux, sorg, dest, n-1, torri);
/* fine */
      }
 }
```

Siccome dovremmo scrivere codice che "mima" l'esecuzione della funzione `hanoi`, abbiamo bisogno di *gestire esplicitamente* i punti di controllo del programma. Dal nostro punto di vista, i punti interessanti sono i possibili punti di ritorno dopo le due chiamate ricorsive (riportati nei commenti al codice con `mezzo` e `fine`), il punto di ritorno della prima chiamata (`princ`) e il punto di inizio di esecuzione della funzione (anche questo denotato nei commenti al codice con `inizio`).

Allo scopo di gestire i punti di controllo del programma, per amore di eleganza, definiamo il seguente tipo enumerato `controlPoint`. Per gestire gli activation records della funzione, definiamo un apposito tipo, attraverso una `struct` adeguata a memorizzare tutte le informazioni necessarie: parametri, variabili locali (qui non presenti) e punto di ritorno (la situazione del gioco (`torri`) può non essere inserita nel record di attivazione perchè non cambia mai valore, o meglio, cambia stato per side effects e viene di fatto trattata come una variabile *globale* a tutte le chiamate).

Infine, sempre per semplificare il codice finale, scriveremo una procedura per allocare un nuovo activation record.

```
typedef enum {inizio, mezzo, fine, princ} controlPoint;

typedef struct{
     int sorg, aux, dest, nd;
     controlPoint ret;
   } hanoiAR;

hanoiAR* createAR(int s, a, d, n, controlPoint r){
    hanoiAR* H;

    if (H = malloc(sizeof(hanoiAR))){
          H->sorg = s;
          H->aux = a;
          H->dest = d;
          H->nd = n;
          H->ret = r;
          return(H);
      } else {
          printf("memoria esaurita\n");
          return NULL;
    }
}
```

Abbiamo tutto il macchinario necessario per scrivere finalmente la funzione iterativa `hanoiIter` (in Fig. 3) che implementa lo stesso algoritmo ricorsivo che risolve il problema della Torre di Hanoi, gestendo però in modo esplicito le chiamate ricorsive attraverso uno stack.

Nella funzione, ogni chiamata ricorsiva corrisponde a infilare un nuovo activation record sulla pila e ricominciare ad eseguire la funzione da `inizio`. Il ritorno (nel caso `fine`) causa la deallocazione dell'activation record e riprendere l'esecuzione dal punto contenuto nel campo `ret` dell'activation record.

Il programma in Fig. 3 sostanzialmente esemplifica una procedura generale per trasformare una qualsiasi funzione ricorsiva in iterativa. Si possono, ovviamente scrivere varianti semplificate *ad hoc* cucite su misura per il problema in esame (od anche studiare ottimizzazioni generali). Ad esempio, è possibile evitare di gestire il controllo del programma e affidarsi completamente allo stack. Il ramo `else` nella procedura `hanoi` può semplicemente essere tradotto con tre `push` su `controlStack`: l'esecuzione della `move` viene simulata da un activation record *finto* con la variabile `nd` caricata con il valore 1. Il risultato è mostrato in Fig. 4.

A questo punto, l'appetito vien mangiando, e si può scrivere un'ulteriore versione "ottimizzata" che evita di inserire sullo stack gli activation record relativi a chiamate che possono essere eseguite subito. Il risultato è in Fig. 5.

```
void hanoiIter(stack torri[]){
    hanoiAR *newAR, *AR;
    stack controlStack;
    controlPoint cp = inizio;

    controlStack = createEmptyStack();
    newAR = createAR(0, 1, 2, howManyStack(torri[0]), princ);
    push(controlStack, newAR);

    while (!isEmpty(controlStack)){
       switch (cp) {
           case inizio:
               AR = top(controlStack);
               if (AR->nd == 1){
                   move(AR->sorg, AR->dest, torri);
                   cp = AR->ret;
                   pop(controlStack);
               } else {
                   newAR = createAR(AR->sorg, AR->dest, AR->aux, AR->nd - 1, mezzo);
                   push(controlStack, newAR);
                   cp = inizio;
               }
               break;
           case mezzo:
               AR = top(controlStack);
               move(AR->sorg, AR->dest, torri);
               newAR = createAR(AR->aux, AR->sorg, AR->dest, AR->nd - 1, fine);
               push(controlStack, newAR);
               cp = inizio;
               break;
           case fine:
               AR = top(controlStack);
               cp = AR->ret;
               pop(controlStack);
               break;
           default:
               break;
           }
   }
}
```

Figura 3: Funzione Iterativa che risolve il problema della Torre di Hanoi

```
void hanoiIterNaif(stack torri[]){
    hanoiAR2 *newAR, *AR;
    stack controlStack;
    controlStack = createStack();
    newAR = createAR2(0, 1, 2, howManyStack(torri[0]));
    push(controlStack, newAR);

    while (!isEmpty(controlStack)){
       AR = top(controlStack);
       pop(controlStack);
       if (AR->nd == 1) move(AR->sorg, AR->dest, torri);
       else {
          newAR = createAR2(AR->aux, AR->sorg, AR->dest, AR->nd - 1);
          push(controlStack, newAR);
          newAR = createAR2(AR->sorg, AR->aux, AR->dest, 1);
          push(controlStack, newAR);
          newAR = createAR2(AR->sorg, AR->dest, AR->aux, AR->nd - 1);
          push(controlStack, newAR);
       }
    }
}
```

Figura 4: Funzione Iterativa per Hanoi senza gestione esplicita del controllo

```
void hanoiIterOpt(stack torri[]){
    hanoiAR2 *newAR, *AR;
    stack controlStack;
    int aux, sorg, dest, nd, w;
    controlStack = createStack(2048);
    sorg=0; aux=1; dest=2; nd=howManyStack(torri[0]);

    while (1) {
          if (nd == 1) {
             move(sorg, dest, torri);
             if (!isEmpty(controlStack)) {
                 AR = top(controlStack);
                 pop(controlStack);
                 sorg = AR->sorg; dest = AR->dest;
                 aux = AR->aux; nd = AR->nd;
             } else break;
           } else {
             newAR = createAR2(aux, sorg, dest, nd - 1);
             push(controlStack, newAR);
             newAR = createAR2(sorg, aux, dest, 1);
             push(controlStack, newAR);
             nd--; w=aux; aux=dest; dest=w;
           } /* end else */
     }/* end while */
}
```

Figura 5: Funzione Iterativa per Hanoi ottimizzata

2.1 Esercizi e Spunti di Riflessione

1. Implementare pile e code usando gli array come struttura dati sottostante. Valutare pregi e difetti delle due implementazioni. Scrivere un programma che usa le code e verificare che sostituendo il modulo che implementa le code, il programma continua a funzionare correttamente (ovviamente dovete aver mantenuto le stesse operazioni con gli stessi nomi, ossia la stessa *interfaccia*).
2. Usare uno stack per scrivere un programma che legge un programma C e risponde 1 se le parentesi sono ben bilanciate, e 0 altrimenti. (Ogni parentesi aperta genera l'allocazione di un elemento sulla pila. Tenete conto dei diversi tipi di parentesi).
3. Provare a generalizzare il tipo di dato insieme, usando la tecnica del `void *`. È possibile mantenere la rappresentazione a liste ordinate? Volendo assolutamente usarle, come devo arricchire l'interfaccia (= insieme di parametri di ingresso) per usare operazioni come l'uguaglianza ed il confronto? (NOTA: se il lettore vuole cimentarsi su questo problema, dovrà studiare su un manuale di C, il passaggio di funzioni come parametri).
4. ★ Avendo definito gli insiemi generici, diventa possibile definire a basso prezzo gli *insiemi di insiemi*. Provare a scrivere una procedura che preso un insieme restituisce l'*insieme delle parti*. Risolvere prima l'esercizio nella dispensa sulla ricorsione che chiedeva di dare una definizione induttiva dell'insieme potenza.
5. Scrivere le versioni iterative delle funzioni che implementano *quickSort* e/o *mergeSort*.
6. Forse un po' meno standard è scrivere la versione iterativa del problema degli anagrammi o della determinazione della semiperfezione di un numero.
7. ★ ▶ Considerate la seguente funzione:
$$\varphi(a,b,n) = \begin{cases} a+b & \text{se } n=0 \\ 1 & \text{se } b=0 \\ \varphi(a,\varphi(a,b-1,n),n) & \text{altrimenti} \end{cases}$$
Scrivere un programma C ricorsivo che calcola φ. Sperimentata che probabilmente già invocazioni del tipo $\varphi(5,5,5)$ generano errori dovute a un numero eccessivo di chiamate ricorsive. Verificate se, usando esplicitamente uno stack, riuscite a calcolare valori più grandi.
8. Oltre a generare un numero enorme di chiamate ricorsive, il calcolo di φ genera rapidamente valori molto grandi. Provare ad usare i naturali a lunghezza illimitata.
9. ★ ♣ Ma che funzione è φ? Chiaramente $\varphi(a,b,0)$ è la somma. E $\varphi(a,b,1)$? $\varphi(a,b,2)$? Se avete capito questo, provate a generalizzare[1].

[1] è interessante ricordare un importante risultato di informatica teorica, che riparafrasato nel nostro piccolo mondo, asserisce che *non esiste nessun programma* che usa solo cicli `for` (con numero di iterazioni fissate, cioè del tipo `for(i=a; i<b; i++)` senza modifica di `i` e `b` nel corpo del ciclo) che calcola φ.