# Progetto per il Laboratorio di Programmazione
# Un interprete per il linguaggio PINO

---

# Modulo VII: Il garbage collector (GC)

---

Stefano Guerrini

A.A. 2002/03 – Canale P-Z
Versione del 4 luglio 2003

## 1 Modulo VII: Il garbage collector (GC)

Per capire i compiti del garbage collector, si prenda l'espressione **tl**$(e)$ con $e = \langle l_1 \texttt{ : } [l_2]\rangle$. La valutazione di questa espressione richiede per prima cosa la valutazione di $e$ e poi l'applicazione della funzione **tl** alla lista $l = [l_1, l_2]$ ottenuta come risultato. Siccome la lista $l$ non esisteva prima della valutazione di $e$, il nodo di tipo `struct list` in testa ad $l$ (quello contenente il riferimento ad $l_1$ nel campo `hd` ed il riferimento a $l_2$ nel campo `tl`) è stato sicuramente allocato durante la valutazione di $e$. Ora, l'applicazione di **tl** ad $l$ fornisce come risultato finale della valutazione la lista $l_2$, facendo così perdere ogni riferimento al nodo di testa di $l$. Infatti, non essendoci più nessun puntatore che fa riferimento ad esso, tale nodo è inaccessibile e non potrà più essere utilizzato dall'interprete nei successivi calcoli. Il nodo così prodotto è quindi uno scarto del calcolo, della memoria inutile di cui ci si può liberare, ovvero, il nodo è *garbage*.[1]

Un'altra situazione tipica che porta alla creazione di garbage è la chiamata di una funzione. Si prenda infatti la seguente dichiarazione **deffun** $f(x, y) =$ **ite**$(x, y, x)$. Al momento della chiamata di $f(e_1, e_2)$, le due espressioni $e_1$ ed $e_2$ sono valutate ed i loro valori sono associati (memorizzati nello stack) ai parametri locali $x$ ed $y$. Quindi, se la lista associata ad $x$ è vuota, il risultato della valutazione dell'espressione di $f$ è la lista vuota, il valore di $y$ viene ignorato e la corrispondente lista diviene garbage; vice versa, se $x$ non è vuota, il risultato è il valore di $y$ ed la lista associata ad $x$ diviene garbage.

In definitiva, la valutazione di un'espressione può portare all'allocazione di nodi per la memorizzazione di liste da usare nei calcoli intermedi che non compaiono nel risultato finale. Le locazioni di memoria di tali nodi, pur essendo allocate e quindi assegnate dal sistema all'interprete (impedendo così il riutilizzo di tale memoria), non saranno più utilizzate dall'interprete, visto che non appaiono in nessuna espressione o lista che potrà entrare a far parte della computazione in corso o di quelle successive.

Un'altra classica situazione in cui viene prodotta garbage si ha quando l'analisi sintattica di un'espressione viene interrotta a causa di un errore sintattico o lessicale. Normalmente, l'errore si verifica dopo che il parser ha creato parte dell'espressione letta. Quindi, se il parser termina la sua computazione segnalando l'errore, senza però liberare i nodi allocati, anche tali nodi diventano garbage.

In molte applicazioni, ogni operazione del programma che può portare alla creazione di garbage deve preoccuparsi della restituzione della memoria che diviene garbage, evitando il formarsi di memoria assegnata al programma, ma non utilizzata (o utilizzabile). Nel nostro caso, visto che il momento in cui un nodo non è più utilizzabile dall'interprete non è immediatamente detrminabile, questo approccio non è immediatamente

[1] La traduzione letterale di *garbage* è *spazzatura* ed il termine *garbage collection* indica la *raccolta della spazzatura*.

implementabile.[2] Per questo motivo, il compito di gestire il recupero della garbage è affidato ad un opportuno modulo detto *garbage collector*.

## 1.1 Mark-and-sweep

La tecnica di garbage collection che prenderemo in esame è detta *mark-and-sweep* e si svolge in due fasi:

**(mark)** si visitano tutte le espressioni e le liste utilizzate dall'interprete marcando con un opportuno tag i nodi raggiunti durante la visita;

**(sweep)** si scandiscono tutti i nodi attualmente utilizzati dall'interprete e (*i*) si cancellano i nodi non marcati nella fase di mark; (*ii*) si elimina il tag dai nodi non cancellati.

Per garantire il recupero di tutta la memoria non utilizzata, è fondamentale che prima della operazione di mark tutti i nodi allocati e non ancora restituiti non siamo marcati. Per questo, durante la fase di sweep viene eliminato il tag dei nodi non cancellati ed occorre garantire che

- al momento della sua creazione, un nodo non è marcato.

L'implementazione del precedente procedimento richiede che

1. l'interprete mantenga una lista di tutti i nodi allocati e non ancora restituiti;
2. ogni nodo disponga di un opportuno campo per memorizzare il tag del garbage collector, ad esempio, un valore intero (booleano) pari a 0 (falso) se il tag è assente, o diverso da 0 (vero) se il tag è presente.

Non volendo cambiare le strutture definite nei precedentemenenti moduli per la memorizzazione dei nodi, definiamo la seguente struttura

```
/* Header dei nodi allocati da gc_alloc
 * Se p e' l'indirizzo dell'header, il nodo si trova all'indirizzo
 * ((struct header *)p)+1
 */
struct header_nodo {
    struct header_nodo *next; // header successivo
    long tag; // tag per la GC
};
```

da aggiungere in testa ad ogni nodo. Più precisamente, al momento dell'allocazione di un nodo di dimensione `size`, anzichè richiedere `size` byte di memoria, se ne richiedono `size+sizeof(struct header_nodo)`. In questo modo si può assumere che il blocco ottenuto sia composto da un header contenente la struttura sopra definita seguito dalle locazioni di memoria del nodo vero e proprio. Quindi, se `p` è il puntatore al blocco di memoria di dimensione `size+sizeof(struct header_nodo)`, dichiarando `p` di tipo `struct header_nodo *`, i campi memorizzati nell'header del nodo possono essere acceduti applicando l'usuale operatore `->` seguito dal campo da leggere o scrivere. Il nodo vero e proprio segue invece l'header ed il suo indirizzo può essere calcolato utilizzando l'aritmetica dei puntatori; infatti, tenendo conto che aggiungendo 1 ad un puntatore ad una struttura si ha l'indirizzo di memoria della locazione di memoria che segue la struttura, l'indirizzo del nodo è `p+1`, nel caso che `p` abbia tipo `struct header *`; oppure, se `p` ha tipo `void *` (o tipo non noto), l'indirizzo del nodo può essere ottenuto con `((struct header *)p)+1`.

[2] Dato che durante la valutazione le liste non vengono copiate, è facile vedere che una stessa lista può essere puntata da più di un nodo. Ad esempio, se la variabile $x$ ha come valore la lista $l$, dalla valutazione di $\langle x : x \rangle$ si ottiene la lista che ha come testa e come coda a lista $l$, che è semplicemente ottenuta creando un nodo di tipo `struct list` che contiene in entrambi i campi un puntatore ad $l$.

## 1.2 Le funzioni da implementare

**La funzione `gc_alloc`**

Nei precedenti moduli, si è suggerito di usare la `gc_alloc` per allocare tutta la memoria necessaria alla memorizzazione di espressioni e liste, fornendo una versione della `gc_alloc` che allocava il nodo senza eseguire nessuna delle operazioni necessarie per l'implementazione del garbage collector. Tale versione della `gc_alloc` dovrà ora essere rimpiazzata da quella implementata in questo modulo.

Il prototipo della funzione rimane ovviamente invariato

```
void *gc_alloc(size_t size);
/* Funzione per l'allocazione della memoria.
 * La gc_alloc deve essere usata per allocare tutti i nodi
 * delle espressioni e delle liste (inclusi i vettori per gli argomenti
 * allocati nei nodi ite e funzione).
 * -- Moduli da I a VI -- :
 * E' un involucro che richiama la malloc di sistema per allocare
 * un nodo di dimensione size.
 * Da implementare per gestire la garbage collection nel modulo VII
 * -- Modulo VII --
 * Alloca un nodo di dimensione size ed il suo header.
 * Ritorna il puntatore p al nodo allocato;
 * l'header del nodo si trova all'indirizzo ((struct header *)p)-1
 * La funzione puo' richiamare il garbage collector se il numero di
 * nodi allocati dall'ultima GC supera MAX_NUM_NODI
 */
```

La funzione `gc_alloc` da implementare dovrà

1. allocare il nodo di memoria ed il suo header;
2. aggiungere il nodo di memoria alla lista dei nodi allocati e non ancora restituiti;
3. inizializzare a 0 il campo tag dell'header del nodo.
4. attivare il processo di garbage collection se si verifcano determinate condizioni (vedi sezione 1.5).

Per mantenere la lista dei nodi allocati e non ancora restituiti, nel file `pinodefs.c` è definita la variabile (dichiarata extern in `pinodefs.h`)

```
/* La lista dei nodi allocati da gc_alloc */
struct header_nodo *lista_nodi_mem = NULL;
```

**Le funzioni `mark` e `sweep`**

Le altre due funzioni da implementare sono

```
int mark(void);
/* Esegue la marcatura dei nodi utilizzati dall'interprete
 * Ritorna il numero di nodi marcati
 */

int sweep(void);
/* Recupera i nodi non marcati conenuti nella lista lista_nodi_mem
 * Ritorna il numero di nodi recuperati
 */
```

che eseguono le operazioni di mark e sweep del garbage collector. Per poter avere un'idea dell'andamento della garbage collection, la funzione che esegue il mark-and-sweep dovrà stampare il numero dei nodi marcati ed il numero di nodi recuperati (questi valori sono ritornati dalle corrispondenti funzioni). Per rendere uniforme l'output dei programmi, la seguente funzione che richiama `mark` e `sweep` stampando il loro output è contenuta nel file `pinodefs.c` (ed il suo prototipo è dichiarato in `pinodefs.h`).

```
void gc(void) {
    /* Esegue la garbage collection con un algoritmo di mark-and-sweep.
     * Stampa un messaggio che segnala l'esecuzione della garbage
     * collection. Stampa anche
     * - il numero di nodi marcati (pari al numero di nodi che rimarranno
     * dopo il completamento della gargabe collection)
     * - il numero di nodi recuperati durante la fase di sweep.
     */
    int m, r;
    printf("[GC - ");
    m = mark(); // numero dei nodi marcati
    r = sweep(); // numero dei nodi recuperati
    printf("Nodi marcati: %d; nodi recuperati: %d]\n", m, r);
    num_nodi = 0; // azzerra il numero di nodi allocati dall'ultima GC
}
```

Lo scopo del comando `num_nodi = 0;` verrà chiarito nella sezione 1.5.

## 1.3 Mark

I nodi da cancellare sono quelli che non potranno più essere raggiunti durante la valutazione o stampa di una espressione o lista. L'operazione di mark individua tali nodi per differenza, marcando cioè quei nodi che non posso essere cancellati visto che appaiono in espressioni o liste che l'interprete potrebbe visitare durante una successiva operazione. Ma quali sono le espressioni e le liste che l'interprete può utilizzare?

Nel caso in cui la garbage collection viene eseguita subito dopo il completamento di un comando, prima di cominciare a leggere il successivo, i nodi che non possono essere cancellati sono solo quelli delle espressioni e delle liste associate alle funzioni ed alle variabili globali precedentemente definite. Queste espressioni e liste possono essere facilmente individuate scandendo le tavole dei descrittori di variabili e funzioni.

Invece, se la garbage collection viene eseguita nel corso di una computazione, occorre tener conto anche delle liste associate alle variabili locali delle funzioni il cui calcolo non è ancora terminato. Per fortuna, queste liste sono tutte e sole contenute nelle stack, quindi, possono essere facilmente individuate dal garbage collector.

Infine, supponiamo di voler eseguire la garbage collection durante il parsing di un'espressione. In questo caso, a seconda di come si è implementato il parser, non è detto che si possano conoscere tutti i nodi allocati e non ancora combinati per formare l'espressione. Per poter gestire questa situazione, supponiamo di avere una variabile globale (definita in `pinodefs.c` e dichiarata extern in `pinodefs.h`)

```
/* Segnala al GC se si sta eseguendo il parsing di un'espressione.
 * Il suo valore e' diverso da 0 se si sta eseguendo il parsing,
 * e' uguale a 0 altrimenti
 */
int parsing = 0;
```

che contiene un valore diverso da 0 se si sta eseguendo il parsing di un'espressione o 0 altrimenti.

Sfruttando la precedente variabile, si possono adottare due soluzioni:

1. la garbage collection non può essere eseguita durante il parsing di un comando, espressione o lista;
2. si modifica la `gc_alloc` in modo che i nodi allocati dal parser non siano immediatamente inseriti nella lista utilizzata nella fase di sweep (la lista `lista_nodi_mem`), ma mantenuti in una lista separata che verrà spostata in testa a `lista_nodi_mem` solo alla fine del parsing. In questo modo, se si sta eseguendo il parsing

di un'espressione, i nodi allocati per memorizzare la parte di espressione letta potranno essere visitati dalla funzione di sweep (e quindi considerati per la loro eliminazione) solo dopo la corretta costruzione dell'espressione che si sta analizzando.

Per semplicità, supponiamo di adottare la prima soluzione, impedendo di eseguire la garbage collection se `parsing` contiene un valore diverso da 0. Il main di prova che verrà fornito si basa su questa ipotesi.

#### Nodi con riferimenti multipli e riferimenti ciclici

I nodi di una lista possono essere riferiti da più nodi distinti. Ad esempio, la valutazione dell'espressione $\langle x, x\rangle$, crea una lista $l'$ con due riferimenti alla lista $l$ assegnata alla variabile $x$ (si veda anche l'esempio nella nota 2). Ora, se si esegue il mark di $l'$ senza particolare attenzione, la lista $l$ verrà scandita e marcata due volte. Per evitare questa doppia scansione, si osservi che, se ad un certo punto della fase di marking si trova un nodo $r$ già marcato, allora la funzione di marking ha già raggiunto $r$ in precedenza marcando, oltre ad $r$, anche tutti i nodi dell'albero con radice $r$. Per questo motivo, la funzione ricorsiva che implementa l'algortimo di marking dell'albero di un'espressione o lista può arrestarsi immediatamente se la radice dell'albero da marcare ha un tag diverso da 0.

La precedente tecnica garantisce che durante la visita dei nodi l'algoritmo di marking non marca mai due volte lo stesso nodo. Questa ottimizzazione diviene fondamentale per garantire la terminazione del marking nel caso in cui i nodi formano strutture circolari. Per fortuna, l'unica forma di circolarità negli alberi che memorizzano le espressioni di PINO è quella legata alle chiamate ricorsive di funzione. Questa circolarità non crea però problemi indipendentemente dal controllo sul tag dei nodi; infatti, arrivati ad una chiamata di una funzione $f$, dato che l'espressione di $f$ o è già stata marcata o verrà marcata quando si arriverà a scandire il descrittore di $f$, si può evitare di visitare tale espressione.

Per completezza, facciamo però osservare che l'algoritmo di mark-and-sweep funziona correttamente anche in presenza di memoria con riferimenti circolari.

### 1.4 Sweep

In base alle scelte precedentemente fatte, l'implementazione dello sweep non presenta particolari problemi. Si tratta infatti di una semplice scansione della lista `lista_nodi_mem` alla ricerca dei nodi con tag uguale a 0. Si ricorda inoltre che durante questa fase occorrerà azzerare il tag dei nodi che non vengono cancellati.

### 1.5 Quando eseguire la garbage collection

Uno dei problemi più critici dei sistemi in cui la memoria è gestita per mezzo di un garbage collector è determinare quando richiamare la procedura di garbage collection, dove con quando non si intende in quale stato dell'interprete (questo problema è già stato analizzato nella sezione 1.3) ma con quale frequenza ed in quale situazione di memoria. Infatti, il principale vantaggio di gestire la memoria dinamica mediante garbage collection è che durante lo svolgimento dei compiti principali del programma non si deve perdere tempo per determinare e restituire la memoria da deallocare; questo vantaggio è bilanciato dal fatto l'esecuzione della garbage collection può essere molto costoso, visto che richiede la scansione di tutta la memoria allocata. Per questo motivo, se la garbage collection viene eseguita troppo spesso, l'uso del garbage collector può divenire controproducente; vice versa, se il garbage collector non viene mai chiamato, la memoria allocata ma inutilizzata può divenire ecessiva, sino ad esaurire la memoria disponibile. Per questo è essenziale che il garbage collector sia richiamato il minor numero possibile di volte e quando strettamente necessario.

Lo studio dettagliato delle condizioni che devono portare all'attivazione del garbage collector apre però problemi che non si vogliono affrontare nell'implementazione di questo progetto. Per questo motivo, nel progetto, adotteremo due regole molto semplici per l'esecuzione della garbage collection:

1. la funzione `gc` deve essere richiamata dalla `gc_alloc` se il numero di nodi allocati dall'ultima esecuzione di una garbage collection supera un valore prefissato;

2. la funzione `gc` viene invocata dopo l'esecuzione di ogni comando.

Facendo rimanere valido il fatto che `gc` non può essere chiamata durante la fase di parsing.

In particolare, per implementare il primo punto è necessario memorizzare il numero dei nodi allocati a partire dall'esecuzione dell'ultima garbage collection. Per tale motivo, viene definita in `pinodefs.c` (dichiarata extern in `pinodefs.h`) la variabile

```
/* Numero dei nodi allocati dalla gc_alloc dall'ultima esecuzione
 * del garbage collector
 */
int num_nodi = 0;
```

Ovviamente, tale variabile dovrà essere aggiornata dalla `gc_alloc` ed azzerata dopo l'esecuzione di ogni garbage collection (si veda il codice della funzione `gc`). Inoltre, in `pinodefs.h` viene definita la costante

```
/* Numero massimo di allocazioni di nodi tra due chiamate del GC
 */
#define MAX_NUM_NODI 256
```

il cui valore viene mantenuto volutamente basso per poter verificare con piccoli esempi la chiamata della garbage collection da parte di `gc_alloc` (se l'obiettivo è l'efficienza dell'interprete, probabilmente tale valore è troppo basso).

## 1.6 La memoria allocata dall'analizzatore lessicale

La garbage collection riguarda la memoria allocata dal parser e dal valutatore, ma non quella allocata dall'analizzatore lessicale per memorizzare gli identificatori ed i descrittori di variabili e funzioni. Tale memoria può diventare garbage solo nel caso di cancellazione di una variabile o funzione; quindi, siccome nella nostra versione semplificata dell'interprete ciò non è possibile, non è necessario rivedere l'analizzatore lessicale in modo che la memoria dinamica allocata in questa fase possa essere sottoposta a garbage collection.

## 1.7 Espressioni associate a funzioni e variabili

Il ragionamento fatto nella precedente sezione per quanto riguarda la memoria allocata dall'analizzatore lessicale potrebbe essere esteso ai nodi delle espressioni e delle liste assegnate alle variabili ed alle funzioni globali. Infatti, supponendo di non poter mai ridefinire tali variabili e funzioni, è ovvio che le espressioni e le liste ad esse associate non potranno mai divenire garbage. In ogni caso, pensando a possibili estensioni del sistema in cui sia possibile una ridefinizione dei variabili e funzioni (almeno in forma limitata), si preferisce applicare la garbage collection anche a tali nodi.

## 1.8 Compilazione del modulo del garbage collector

Per poter utilizzare la funzione `gc_alloc` definita nel modulo di garbage collection senza dover ricompilare i moduli precedentemente implementati, nella versione finale del file `pinodefs.c` che verrà fornita con questo modulo si utilizzerà la tecnica della compilazione condizionale per poter eliminare dal programma contenente il garbage collector la versione di `gc_alloc` definita in `pinodefs.c`.

Il preprocessore C fornisce una primitiva `#ifdef` *nome_macro* che a seconda se la macro *nome_macro* è definita o meno permette di scegliere se compilare o meno una parte di codice. Ad esempio, nel file `pinodefs.c` è contenuto un frammento di codice di questo tipo

```
#ifdef PINO_GC // Definizioni necessarie alla compilazione del GC

  ...                                       ...
  ... definizioni per l'interprete che usa GC ...
  ...                                       ...

#else // Per la compilazione dei moduli prima del GC
```

```
void *gc_alloc(size_t size) {
    /* Funzione per l'allocazione della memoria.
     * La gc_alloc deve essere usata per allocare tutti i nodi
     * delle espressioni e delle liste (inclusi i vettori per gli argomenti
     * allocati nei nodi ite e funzione).
     * E' un involucro che richiama la malloc di sistema per allocare
     * un nodo di dimensione size.
     * Da implementare per gestire la garbage collection nel modulo VII
     */
    return malloc(size);
}

#endif
```

Se la macro `PINO_GC` è definita, durante la compilazione del file `pinodefs.c`, il compilatore leggerà e compilerà il codice contenuto tra il comando `#ifdef` ed il successivo `#else`, nel quale sono contenute le definizioni per la versione dell'interprete che usa la garbage collection, mentre non compilerà il codice contenuto tra `#else` e `#endif`, ovvero non compilerà la versione di `gc_alloc` contenuta in `pinodefs.c`. Vice versa, se `PINO_GC` non è definita, il compilatore ometterà le definizioni che servono per la versione con il garbage collector e compilerà il codice di `gc_alloc` contenuto in `pinodefs.c`. Quindi, per garantire che non ci sono conflitti tra la `gc_alloc` definita nel modulo del garbage collector e quella definita in `pinodefs.c` occorre compilare il modulo del garbage collector e la versione di `pinodefs.c` da usare con esso garantendo che la macro `PINO_GC` sia definita. La maniera più semplice per farlo è passare al compilatore l'opzione `-D PINO_GC`. Ad esempio, per creare il file oggetto `pinodefs-gc.o` del modulo `pinodefs.c` da collegare con il garbage collector (aggiungiamo `-gc` al nome per non confonderlo con quello da collegare con gli altri moduli) si deve usare il comando

```
gcc -g -c -D PINO_GC pinodefs.c -o pinodefs-gc.o
```

Si osservi che siccome anche `pinodefs.h` contiene alcune parti specifiche del garbage collector condizionate alla definizione di `PINO_GC`, anche per la compilazione del file `gc.c` del garbage collector occorre definire la macro `PINO_GC`, ad esempio, mediante il comando

```
gcc -g -c -D PINO_GC gc.c
```

Nella sezione 1.9 vedremo anche qual è il comando che, partendo dai sorgenti di tutti i moduli, permette di cotruire direttamente l'eseguibile senza dover compilare separatamente i moduli.

## 1.9 Verifica

Per verificare il garbage collector il main che viene utilizzato è in una possibile versione finale dell'interprete PINO ed è riportato in Appendice 1.9.

Se si sta lavorando su di un sistema linux con compilatore gcc, supponendo che il nome del file che contiene il codice del garbage collector è `gc.c` e che i precedenti moduli sono contenuti nei file `tavole.c`, `lexan.c`, `syntan.c`, `stampa.c`, `stack.c` ed `eval.c` il comando per la compilazione dell'interprete è:

```
gcc -g -D PINO_GC pino-VII.c pinodefs.c tavole.c lexan.c syntan.c \
                  stampa.c stack.c eval.c gc.c -o pino-VII
```

Come risultato della compilazione si otterrà l'eseguibile `pino-VII` che legge un programma PINO e risponde ad ogni comando stampando il comando letto e, nel caso di **defvar** o **eval**, stampa il risultato dell'espressione letta preceduto da `->`. Il programma richiama il garbage collector tra la lettura di due comandi successivi.

Si osservi che per verificare quetso modulo servono tutti i moduli precedenti. Se alcuni moduli non sono stati implementati, o sono stati inviati moduli non funzionanti, si utilizzeranno i corrispondenti moduli sviluppati dal docente.

# A Modulo VII: Verifica del garbage collector (GC)

```
/* -*- Mode: C -*- */
/* Time-stamp:  <pino-VII.c 03/07/04 23:45:19 guerrini@zambujero.lan.home> */

/* ********************************************************
 * Progetto di Laboratorio di Programmazione
 * Stefano Guerrini - AA 2002-03
 *
 * Main per la verifica del modulo VII:
 * Per la compilazione servono i seguenti file: il garbage collector (GC)
 *   pinodefs.h        header principali defs
 *   pinodefs.c        alcune var globali e funzioni di utilita
 *   pino-VII.c        questo file
 *   tavole.c          tavole dei simboli - modulo I (TS)
 *   lexan.c           analizzatore lessicale - modulo II (AL)
 *   syntan.c          analizzatore sintattico - modulo III (AS)
 *   stampa.c          funzioni si stampa - modulo IV (PR)
 *   stack.c           stack - modulo V (ST)
 *   eval.c            valutatore - modulo VI (EV)
 *   gc.c              garbage collector - modulo VII (GC)
 * Per la compilazione su linux con gcc
 *   gcc -g -D PINO_GC pino-VII.c pinodefs.c tavole.c lexan.c syntan.c \
 *                     stampa.c stack.c eval.c gc.c -o pino-VII
 * crea l'eseguibile pino-VII con le info necessarie per il debugging
 * ********************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "pinodefs.h"

void skiptk(struct descr_token *pdtk) {
    /* Una funzione ausiliaria che, se il token ottenuto non e'
     * un ';' o '.' legge ed ignora i successivi token fino ad ottenere
     * un ';' o '.'.
     * Serve a gestire situazioni in cui si e' trovato un errore
     * prima di arrivare ad un ';' o ad un '.', oppure si e'
     * letta un'espressione corretta, ma questa e' seguita da un
     * token diverso da ';' o '.'
     */
    while (pdtk->tag != SEMI && pdtk->tag != DOT)
        next_token(pdtk);
}

int main(void) {
    /* Verifica eval_expr
     * Legge un programma pino
     * Legge e stampa ogni comando e nel caso della defvar e della eval
     * stampa anche il risultato della valutazione dell'espressione
     * inserita (nella defvar, la lista risultato viene anche assegnata
     * alla variabile).
     * In caso di errore durante la valutazione stampa il corrispondente
     * messaggio di errore.
     */
    struct descr_var *pdv;
    struct descr_fun *pdf;
    struct expr *exp;
```

```
struct list *ls;
int no_first = 0; // vero se non e' il primo comando
printf("Verifica di eval_expr\n");
printf("\tInserire un programma PINO\n");
init_stack(&lvars_stack, 1024*1024); // inizializza lo stack delle var locali
do {
    if (no_first) // esegue gc se non e' il primo comando
        gc();
    else
        no_first = -1; // il primo comando e' stato eseguito
    printf("# "); // prompt
    next_token(&last_tk); // legge il token del tipo di comando
    switch (last_tk.tag) {
    case DVAR: // defvar
        parsing = -1; // segnala che si sta eseguendo un parsing
        pdv = parse_defvar();
        parsing = 0; // fine parsing
        if (pdv == NULL) {
            printf("Errore lessicale/sintattico\n");
            skiptk(&last_tk);
        } else {
            print_defvar(pdv); // stampa la defvar letta
            printf("\n");
            next_token(&last_tk); // legge il token che segue il comando
            if (last_tk.tag != SEMI && last_tk.tag != DOT) { // Errore
                printf("Errore lessicale/sintattico\n");
                skiptk(&last_tk);
            } else { // valuta l'espressione
                pdv->ls = eval_expr(pdv->exp);
                printf("-> ");
                if(error) // errore
                    printf("Errore durante la valutazione: %s\n",
                           error_msg[error]);
                else { // stampa il risultato
                    print_list(pdv->ls);
                    printf("\n");
                }
            }
        }
        break;
    case DFUN: // deffun
        parsing = -1; // segnala che si sta eseguendo un parsing
        pdf = parse_deffun();
        parsing = 0; // fine parsing
        if (pdf == NULL) {
            printf("Errore lessicale/sintattico\n");
            skiptk(&last_tk);
        } else {
            print_deffun(pdf);  // stampa la defvar letta
            printf("\n");
            next_token(&last_tk);  // legge il token che segue il comando
            if (last_tk.tag != SEMI && last_tk.tag != DOT) { //errore
                printf("Errore lessicale/sintattico\n");
                skiptk(&last_tk);
            }
        }
        break;
```

```
        case EVAL: // eval
            parsing = -1; // segnala che si sta eseguendo un parsing
            exp = parse_expr(NULL);
            parsing = 0; // fine parsing
            if (exp == NULL)  {
                printf("Errore lessicale/sintattico\n");
                skiptk(&last_tk);
            } else {
                print_expr(exp, NULL); // stampa l'espressione letta
                printf("\n");
                next_token(&last_tk);  // legge il token che segue il comando
                if (last_tk.tag != SEMI && last_tk.tag != DOT) { // errore
                    printf("Errore lessicale/sintattico\n");
                    skiptk(&last_tk);
                } else { // valuta l'espressione
                    ls = eval_expr(exp);
                    printf("-> ");
                    if(error) // errore
                        printf("Errore durante la valutazione: %s\n",
                               error_msg[error]);
                    else { // stampa il risultato
                        print_list(ls);
                        printf("\n");
                    }
                }
            }
            break;
        default:
            printf("Errore lessicale/sintattico\n");
            skiptk(&last_tk);
            break;
        }
    } while (last_tk.tag == SEMI);
}
```