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

Stefano Guerrini

A.A. 2002/03 – Canale P-Z
Versione del 22 luglio 2003

Il linguaggio

Lo scopo del progetto è quello di realizzare un interprete per il linguaggio PINO, una versione molto semplificata e ridotta del linguaggio LISP—una sorta di LISPino o, per gli amici, PINO.

Purtroppo PINO non è un sottoinsieme di LISP, visto che per semplificare l'implementazione dell'interprete si è dovuta cambiare la sintassi di LISP.

Come in LISP, il tipo di dato di base è quello delle liste. In PINO, le liste sono anche l'unico tipo di dato a disposizione—quindi, gli elementi contenuti in una lista sono a loro volta delle liste. Più precisamente, ogni valore è una lista di liste costruita a partire dalla lista vuota '$*$', ovvero un albero, o meglio un PINO, decorato con dei simboli '$*$' alle foglie.

L'unico tipo di dato: le liste

Come già detto, la lista vuota è denotata dalla costante $*$. La lista formata dalle liste $l_0, \ldots, l_k$ è denotata dalla sequenza delle suddette liste, separate da virgole e racchiuse tra parentesi quadre, ovvero da $[l_0, \ldots, l_k]$. Per chiarire la struttura delle liste, eccone alcuni esempi.

- La lista $[\,*\,]$ è una lista di lunghezza 1 che contiene come unico elemento la lista vuota.
- La lista $[[\,*\,], *\,]$ è una lista di lunghezza 2 che contiene, nell'ordine, le liste $[\,*\,]$ e $*$.
- La lista $[[\,*\,], [[\,*\,], *\,]]$ è una lista di lunghezza 2 che contiene le liste dei due esempi precedenti.

Le funzioni predefinite

Sono predefinite in PINO le funzioni **hd** (head) e **tl** (tail) che, presa una lista non vuota, ne ritornano la testa e la coda, rispettivamente—le precedenti funzioni non sono definite sulla lista vuota. Ecco alcuni esempi di applicazione di **hd** e **tl**:

- $\mathbf{hd}([\,*\,]) = *$ mentre $\mathbf{tl}([\,*\,]) = *$;
- $\mathbf{hd}([[\,*\,], [[\,*\,], *\,], *\,]) = [\,*\,]$ mentre $\mathbf{tl}([[\,*\,], [[\,*\,], *\,], *\,]) = [[[\,*\,], *\,], *\,]$.

Il linguaggio PINO fornisce anche una funzione di tre argomenti **ite** corrispondente al costrutto if-then-else. Se il primo argomento di **ite** è una lista non vuota, allora **ite** ritorna il valore del suo secondo argomento, altrimenti ritorna il valore del terzo argomento; ovvero,

$$\mathbf{ite}(l_0, l_1, l_2) = \begin{cases} l_1 & \text{se } l_0 \neq * \\ l_2 & \text{se } l_0 = * \end{cases}$$

.

Le espressioni

Partendo dai valori di tipo lista precedentemente definiti (le costanti del linguaggio) e combinando le precedenti operazioni si possono cominciare a costruire espressioni che, se valutate, danno come risultato una lista. Una sequenza di $k+1$ espressioni (con $k \geqslant 0$) può essere combinata nell'espressione

$$\langle e_0, \ldots, e_k \rangle$$

Il risultato della valutazione di questa espressione è una lista di lunghezza $k+1$, formata dalle liste corrispondenti alle espressioni racchiuse tra parentesi angolari. Più precisamente, se l_i è la lista che si ottiene dalla valutazione dell'espressione e_i, allora il risultato della valutazione di $\langle e_0, \ldots, e_k \rangle$ è la lista $[l_0, \ldots, l_k]$. Si osservi che esiste una differenza fondamentale tra un'espressione ed una lista, la prima è una sorta di programma il cui valore va ancora determinato, la seconda è una costante. Si pensi ad esempio, nel caso delle espressioni numeriche, alla differenza tra il numero 5 e l'espressione 3+2. La cosa è meno intuitiva che nel caso dei numeri dato che le costanti di PINO hanno una struttura più complicata di quella delle costanti numeriche. Comunque, come vedremo in seguito, la differenza tra espressioni e liste si rifletterà anche sulle strutture dati che verranno utilizzate per la loro rappresentazione.

In PINO esiste anche un altro costrutto che combina sequenze di espressioni per costruire una nuova lista e che estende quello precedentemente visto

$$\langle e_0, \ldots, e_k \ \mathbf{:}\ e_{k+1} \rangle$$

In questo caso, se l_i è la lista associata all'espressione e_i, la precedente espressione costruisce la lista ottenuta dalla concatenazione di $[l_0, \ldots, l_k]$ e l_{k+1}, ad esempio, $\langle l_0, l_1 \ \mathbf{:}\ [l_2, l_3] \rangle = [l_0, l_1, l_2, l_3]$. Infine, si osservi che $\langle e_0, \ldots, e_k \rangle = \langle e_0, \ldots, e_k \ \mathbf{:}\ * \rangle$.

Per chiarire meglio i precedenti concetti, ecco alcuni esempi: l'espressione $\langle * , * \rangle$ ha come risultato la lista $[*, *]$; l'espressione $\langle \mathbf{ite}(*, [*], *) \ \mathbf{:}\ \mathbf{tl}([*, *, [*]]) \rangle$ ha come risultato la lista $[*, *, [*]]$.

I comandi

Il linguaggio PINO permette all'utente di definire proprie variabili e funzioni, associando un valore (il risultato di una espressione) alle prime ed una espressione da valutare alle seconde.

Se x è un nome di variabile scelto dall'utente (un nome di variabile è una sequenza di caratteri alfanumerici che comincia con un carattere alfabetico), il comando

$$\mathbf{defvar}\, x \ = \ e$$

associa alla *variabile globale* x il valore (la lista) corrispondente all'espressione e, nella quale oltre alle espressioni base sulle liste ed alle funzioni predefinite, possono occorrere le variabili globali e le funzioni precedentemente definite dall'utente.

Se f è un nome di funzione scelto dall'utente (un nome di funzione è una sequenza di caratteri alfanumerici che comincia con un carattere alfabetico) e $x_1, \ldots, x_k$ sono nomi distinti di *variabili locali* scelti dall'utente, il comando

$$\mathbf{deffun}\, f(x_1, \ldots, x_k) \ = \ e$$

associa alla funzione di k argomenti f l'espressione e nella quale, oltre alle variabili e funzioni precedentemente definite, possono occorrere anche le variabili locali di f e chiamate ricorsive ad f.

La sintassi

La sintassi di PINO può essere definita in modo preciso per mezzo della notazione della cosiddetta *sintassi astratta*. Associamo un nome (nel nostro caso una lettera) a ciascuna delle categorie sintattiche che vogliamo definire—ad esempio, usiamo l per le liste. Per indicare gli elementi di tale tipo, usiamo la lettera che definisce la categoria e, nel caso servano più elementi di una categoria sintattica, facciamo seguire il nome della categoria da un indice—ad esempio, con $l_1, \ldots, l_k$ indicheremo una sequenza di k elementi della categoria l, ovvero k liste. La definizione (ricorsiva) di una certa categoria la si ottiene associando mediante il simbolo ::= una sequenza di possibili costruzioni sintattiche separate dal simbolo | ad ogni nome di categoria; nelle costruzioni sintattiche si

possono usare elementi di un'altra categoria sintattica o dei simboli costanti del linguaggio che stiamo descrivendo (che nel nostro caso indicheremo sempre in grassetto). Ad esempio, in Figura 1, l'equazione (1) definisce i valori di base (le liste) del linguaggio; in pratica, ogni elemento della categoria l o è la costante $*$ (che denota la lista vuota), oppure è della forma $[l_0, \ldots, l_k]$ (si noti che i simboli '[', ']' e ',' sono costanti della sintassi del linguaggio).

L'equazione (2) in Figura 1 definisce invece le espressioni di PINO. Si osservi che ogni lista è un'espressione (una costante). Ricordiamo ancora una volta la distinzione tra l'espressione $e = \langle e_0, \ldots, e_k \rangle$ e la lista $l = [l_0, \ldots, l_k]$. Nel primo caso, le e_i sono espressioni che possono contenere applicazioni delle funzioni **hd**, **tl** e **ite**, variabili o funzioni definite dall'utente; nel secondo caso, le l_i sono delle liste. Di conseguenza, il valore di e è ancora implicito e va calcolato determinando il valore di tutte le e_i, mentre, nel caso di l, abbiamo un'espressione costante il cui valore è la lista stessa.

$$
\begin{array}{lcll}
l & ::= & * \mid [l_0, \ldots, l_k] & (1) \\
e & ::= & l \mid \langle e_0, \ldots, e_k \rangle \mid \langle e_0, \ldots, e_k : e_{k+1} \rangle & (2) \\
 & & \mid \mathbf{hd}(e) \mid \mathbf{tl}(e) \mid \mathbf{ite}(e_0, e_1, e_2) & \\
 & & \mid x \mid f(e_1, \ldots, e_k) & \\
c & ::= & \mathbf{defvar}\, x \;=\; e \mid \mathbf{deffun}\, f(x_1, \ldots, x_k) \;=\; e \mid \mathbf{eval}\, e & (3) \\
p & ::= & c_1; \ldots; c_k. & (4)
\end{array}
$$

Figura 1: La sintassi di PINO

Un programma p di PINO è una sequenza di comandi separati da un '**;**' e seguiti da un '**.**' finale. Come già parzialmente visto, i comandi sono di tre tipi: la definizione di una variabile globale mediante una **defvar**; la definizione di una funzione mediante una **deffun**; la valutazione di una espressione mediante il comando "**eval** e".

Per quanto riguarda i nomi di variabile e funzione che possono occorrere in un'espressione e, questi devono rispettare i seguenti vincoli:

- le variabili globali che occorrono in e devono essere già state definite in precedenza (ad esempio, in "**defvar** $x \;=\; e$", la variabile x non può apparire in e, a meno che ad x non fosse già stato assegnato un valore con una precedente **defvar**, valore che verrà cancellato dalla nuova definizione);
- le funzioni che occorrono in e devono essere già state definite in precedenza, salvo nel caso del comando "**deffun** $f(x_1, \ldots, x_k) \;=\; e$", dove l'espressione e può contenere chiamate della funzione f che si sta definendo, nel qual caso, la funzione f è *ricorsiva*;
- in un comando "**deffun** $f(x_1, \ldots, x_k) \;=\; e$" l'espressione e può contenere occorrenze delle *variabili locali* $x_1, \ldots, x_k$ (si ricorda che i nomi delle variabili locali di una funzione devono essere distinti).

L'esecuzione dei comandi

L'esecuzione di un programma di PINO consiste nell'iterazione della seguente procedura, fino a che non si trova il comando seguito da '**.**'.

1. lettura di un comando;
2. esecuzione dell'operazione corrispondente;
3. stampa di un messaggio con il risultato dell'operazione.

Occorre osservare che L'esecuzione di una **defvar** differisce in modo sostanziale dall'esecuzione di una **deffun**.

- L'esecuzione di un comando **defvar** richiede la valutazione dell'espressione alla destra del simbolo '=' e quindi la creazione di un'associazione tra il nome della variabile che si sta definendo ed il valore dell'espressione alla destra del simbolo '='.

- L'esecuzione di un comando **deffun** non richiede la valutazione dell'espressione alla destra del simbolo '=', ma la creazione di un "programma" corrispondente a tale espressione. Programma che dovrà poi essere eseguito (valutato) quando la funzione verrà utilizzata durante la valutazione delle successive espressioni, associando a ciascuna delle variabili locali i valori passati al momento della chiamata della funzione. Ad esempio, data la seguente definizione di funzione "**deffun** cons(h ,t) = ⟨h :t⟩", vogliamo che "**eval** cons([∗], [∗, [∗]])" ritorni come risultato [[∗], ∗ , [∗]].

Struttura del progetto

Il progetto è suddiviso in moduli. Per garantire la massima indipendenza nella realizzazione dei singoli moduli, verranno chiaramente specificate le strutture dati e le funzioni di interfaccia. Rispettare le specifiche di tali interfacce è fondamentale per garantire la costruzione dell'interprete a partire dai moduli. *Non rispettare le specifiche sarà valutato come un errore.*

I moduli in cui è suddiviso il progetto sono:

1. Modulo I: Le tavole dei simboli (TS)
2. Modulo II: L'analizzatore lessicale (AL)
3. Modulo III: L'analizzatore sintattico (AS)
4. Modulo IV: Le funzioni di stampa (PR)
5. Modulo V: Lo stack (ST)
6. Modulo VI: La valutazione delle espressioni (EV)
7. Modulo VII: Il garbage collector (GC)

Per favorire la chiarezza delle specifiche verranno forniti due file `pinodefs.h` (vedi Appendice A) e `pinodefs.c` (vedi Appendice B). Il primo contiene le principali dichiarazioni di tipo e di costanti da includere in tutti i moduli che verranno sviluppati, il secondo contiene la definizione di variabili globali ed eventuali procedure e dovrà essere compilato e collegato con i moduli ed un opportuno main.

Verifica e valutazione dei moduli del progetto

Per ciascun modulo verrà fornito un main file (si veda l'appendice) e le istruzioni su come, a partire dal modulo sviluppato, ottenere un programma funzionante che verifichi il modulo. In molti casi, la verifica di un modulo dipende da alcuni dei moduli precedenti; a tale scopo, al posto dei moduli mancanti o non funzionanti si utilizzeranno dei moduli sviluppati dal docente.

Si fa presente che tutti i moduli dovranno essere compilabili con un compilatore ANSI C e dovranno utilizzare solo le librerie standard ANSI, oltre a quanto specificato nei file `pinodefs.h` e contenuto in `pinodefs.c` e nei main file per la verifica `pino-x.c`, dove `x` è il numero di modulo che si sta verificando.

Il codice sorgente di ciascuno dei moduli sarà contenuto in un unico file. Nelle istruzioni di compilazione riportate in appendice, per il file sorgente di ciascun modulo si userà il nome riportato nella seguente tabella in corrispondenza del numero (sigla) del modulo.

I (TS)	II (AL)	III (AS)	IV (ST)	V (PR)	VI (EV)	VII (GC)	totale
3	5	8	5	3	8	4	36
`tavole.c`	`lexan.c`	`syntan.c`	`stampa.c`	`stack.c`	`eval.c`	`gc.c`	

Per la composizione finale dei moduli verrà fornito un main file `pino.c` che compilato e collegato con tutti i moduli del progetto darà l'interprete di PINO.

La precedente tabella riporta anche il peso in 30esimi dei singoli moduli. Si osservi che non tutti i moduli hanno la stessa difficoltà e che per raggiungere il massimo - o la sufficienza - non è necessario sviluppare tutti i

moduli. Un voto complessivo tra 30 e 32 corrisponde a 30/30esimi; un voto superiore a 32 da diritto alla lode. Si osservi che per raggiungere la sufficienza (18/30esimi) occorre sviluppare almeno uno dei modui più difficili - III(AS) o VI(EV) - o sviluppare tutti gli altri 5 moduli.

Si fa osservare che il peso di ciascun modulo riportato in tabella è il voto massimo che si può ottenere per quel modulo. Requisiti essenziali affinchè un modulo sia preso in considerazione per la valutazione sono:

- il sorgente del modulo deve essere **compilabile** con un compilatore ANSI C (come riferimento useremo il compilatore gcc disponibile su tutte le architetture unix/linux);
- collegando il modulo con il main corrispondente e con gli altri moduli precedentemente sviluppati (o con quelli forniti dal docente), in base alla istruzioni riportate in appendice, **non si devono ottenere errori**;
- come risultato della compilazione/collegamento si deve ottenere un programma eseguibile che **gira correttamente su alcuni dei test proposti in appendice**.

Atri fattori che influiranno sulla valutazione del modulo sono:

- la documentazione del codice (i moduli devono essere esaurientemente commentati);
- lo stile di scrittura (il codice deve essere opportunamente indentato).

Si ricorda che il progetto di laboratorio è un **lavoro individuale**. Per verificare la similitudine dei moduli inviati si userà un apposito tool automatico: **moduli chiaramente copiati saranno annullati**.

1 Modulo I: Le tavole dei simboli (TS)

Durante la lettura di un programma occorre individuare se un dato identificatore corrisponde ad un nome di variabile globale, ad un nome di funzione, ad un nome di variabile locale, o ad una parola chiave e recuperare alcune informazioni associate a tale variabile o funzione (ad esempio, il numero di argomenti della funzione). Le strutture dati che permettono di gestire queste informazioni sono le *tavole dei simboli*.

Nell'implementazione di PINO noi utilizzeremo due tipi di tavole dei simboli: quelle implementate mediante vettori e quelle implementate mediante strutture dinamiche. Dove e come utilizzare tali tavole sarà chiarito nella specifica dei successivi moduli del progetto.

1.1 Tavole dei simboli implementate mediante vettori

La prima tavola di simboli o chiavi da implementare è realizzata con un vettore di stringhe—ogni chiave è una stringa. Il file `pinodefs.h` contiene le seguente definizione per una struttura che implementa una tavola con al massimo `MAX_KEYS` (anche questa costante è definita in `pinodefs.h`).

```
/* Struttura per la rappresentazione di una tavola mediante un vettore */
struct tavola {
    unsigned n; // elementi nella tavola
    char *keys[MAX_KEYS]; // vettore con le stringhe nella tavola
};
```

Dove `n` è il numero di elementi attualmente contenuti nella tavola ed il vettore `keys` contiene nelle sue prime `n` posizioni le stringhe delle chiavi memorizzate nella tavola.

Si devono scrivere le seguenti procedure:

```
void init_tavola(struct tavola *ptv);
/* inizializza la tavola *ptv */

unsigned avail_tavola(struct tavola *ptv);
/* numero di spazi ancora disponibili nella tavola *ptv */
```

```
int ins_key_tavola(struct tavola *ptv, char *key);
/* aggiunge la chiave key alla tavola se non gia' presente
 * ritorna l'indice della posizione della chiave inserita
 * o -1 nel caso in cui key e' gia' presente nella tavola o
 * non c'e' posto per inserirla
 */

int search_key_tavola(struct tavola *ptv, char *key);
/* cerca la chiave key nella tavola *ptv
 * ritorna la posizione della chiave nella tavola
 * o -1 se la chiave non \`e presente
 */
```

Il cui comportamento è descritto nei corrispondenti commenti.

1.2 Tavole dei simboli implementate mediante liste

Le tavole delle variabili e delle funzioni definite dall'utente verranno implementate mediante liste. Le strutture dati che servono per definire tali liste sono riportate qui di seguito (anche queste definizioni sono contenute in `pinodefs.h`).

```
/* Descrittore di funzione */
struct descr_fun {
    char id[MAX_ID_LEN]; // il nome della funzione
    struct tavola tv; // tavola con il nome degli argomenti della funzione
    struct expr *exp; // l'espressione associata alla funzione
};

/* Nodo lista descrittori di funzioni */
struct nodo_lista_dfun {
    struct descr_fun *pdf;
    struct nodo_lista_dfun *next;
};

/* Lista descrittori di funzioni */
typedef struct nodo_lista_dfun *lista_dfun;

/* Descrittore di variabile */
struct descr_var { // descrittore di variabile globale
    char id[MAX_ID_LEN]; // il nome della var
    struct expr *exp; // l'espressione associata alla var
};

/* Nodo lista descrittori di variabile */
struct nodo_lista_dvar {
    struct descr_var *pdv;
    struct nodo_lista_dvar *next;
};

/* Lista descrittori di variabile */
typedef struct nodo_lista_dvar *lista_dvar;
```

Ogni funzione ed ogni variabile ha un descrittore di tipo `desr_fun`, se funzione, o `descr_var`, se variabile. Il dettaglio dei dati contenuti in questi descrittori verrà analizzato in seguito (nella parte di analisi sintattica). Per il momento è sufficiente osservare che ciascun descrittore contiene un vettore di caratteri di dimensione `MAX_ID_LEN` (anche questa costante è definita in `pinodefs.h`). In tale vettore va memorizzato il nome della

variabile o funzione rappresentata dal descrittore. Si osservi che ciò implica che tutti gli identificatori validi di PINO hanno una lunghezza inferiore a `MAX_ID_LEN`.

I prototipi delle funzioni da implementare sono riportati qui di seguito. Il comportamento di ciascuna funzione è descritto dal corrispondente commento.

```
struct descr_fun *ins_key_dfun(lista_dfun *pls, char *id);
/* aggiunge un nuovo descrittore per id alla lista *pls
 * ritorna il puntatore al nuovo descrittore inserito
 * o NULL in caso di errore o nel caso in cui id e'
 * gia' presente in *pls
 */

struct descr_fun *search_key_dfun(lista_dfun ls, char *id);
/* cerca il descrittore di funzione di id in ls
 * ritorna il puntatore al descrittore trovato
 * o NULL se non presente
 */

struct descr_var *ins_key_dvar(lista_dvar *pls, char *id);
/* aggiunge un nuovo descrittore per id alla lista *pls
 * ritorna il puntatore al nuovo descrittore inserito
 * o NULL in caso di errore o nel caso in cui id e'
 * gia' presente in *pls
 */

struct descr_var *search_key_dvar(lista_dvar ls, char *id);
/* cerca il descrittore di variabile di id in ls
 * ritorna il puntatore al descrittore trovato
 * o NULL se non presente
 */
```

Le funzioni della libreria `string.h`

Per realizzare la ricerca nelle tavole si consiglia di studiare le funzioni di manipolazione di stringhe contenute nella libreria `string.h` (si veda ad esempio il D&D). Alcune funzioni di tale libreria sono sicuramente utili per scrivere le funzioni che confrontano stringhe. Alcune funzioni della `string.h` potranno risultare utili anche per altri moduli del progetto.

1.3 Verifica

Per verificare le tavole dei simboli si utilizzerà il main file `pino-I.c` riportato in Appendice C.

Se il nome del file con il codice del modulo è `tavole.c`, su di un sistema linux con compilatore gcc il comando per la compilazione del programma di test è:

```
gcc -g pino-I.c pinodefs.c tavole.c -o pino-I
```

Come risultato della compilazione si ottiene l'eseguibile `pino-I` che, dopo aver chiesto di selezionare mediante una maschera la tavola da verificare, legge una sequenza di stringhe alfanumeriche da inserire e cercare all'interno della tavola selezionata.

2 Modulo II: L'analizzatore lessicale (AL)

L'input che l'interprete di PINO riceve è una sequenza di caratteri. La prima cosa che l'interprete deve fare per poter valutare i comandi e le espressioni di PINO è individuare il tipo del comando e costruire un'opportuna

rappresentazione interna delle espressioni. In questa fase, l'interprete deve anche analizzare la correttezza sintattica del programma e per tale motivo viene denominata *analisi sintattica*. Ad esempio, in un testo in italiano, l'analisi sintattica verifica la correttezza del modo in cui le unità sintattiche sono state messe insieme per formare la frase. Ma, le unità sintattiche che compongono un testo non sono i singoli caratteri, ad esempio, nel testo in italiano le unità da cui si parte sono le parole ed i simboli di punteggiatura.

Nel caso dei programmi PINO si possono invece individuare le seguente unità sintattiche:

- i *simboli speciali* "[]⟨⟩();:,.=∗".
- le *parole chiave* **deffun**, **defvar**, **eval**, **hd**, **tl** e **ite**;
- i nomi di variabile o funzione, anche detti *identificatori*, definiti come sequenze di caratteri alfanumerici che cominciano con un carattere alfabetico.

Una sequenza di caratteri appartenente ad una delle precedenti categorie è detta un *token*. Gli eventuali spazi bianchi che separano i token non sono significativi—per spazio bianco si intende il carattere spazio, i caratteri di tabulatore e il carattere di linea nuova—ovvero, due token del programma possono essere separati da uno o più spazi bianchi. Anche se in certi casi lo spazio bianco può essere assente, come nel caso della sequenza "alfa(" composta dai due token "alfa" e "(". Si osservi che nel suddividere, una sequenza di caratteri in token si considerano sempre le sottosequenze più lunghe che rientrano in una delle precedenti categorie; per questo nell'esempio "alfa(" si ha un unico token per "alfa".

Il primo passo per poter procedere all'analisi sintattica di un programma è la sua decomposizione in token. Questa fase non richiede la conoscenza delle regole sintattiche del linguaggio, ma solo delle sue unità sintattiche, ovvero del suo lessico. Per tale motivo, la fase di suddivisione di un programma in token è detta *analisi lessicale*.

Lo scopo di questa parte del progetto è proprio quello di scrivere l'analizzatore lessicale dei programmi scritti in PINO.

2.1 La funzione `next_token`

Il cuore dell'analizzatore analizzatore è la funzione che trasforma la sequenza di caratteri di input in una sequenza di token. Si richiede pertanto di scrivere un modulo che, come interfaccia, con le altre parti del progetto fornisca la funzione:

```
void next_token(struct descr_token *pdtk);
/* Ritorna in *pdtk il prossimo token nello stream di input */
```

che alla sua prima chiamata ritorna il descrittore del primo token nello stream di input, alla sua seconda chiamata ritorna il descrittore del secondo token, poi il terzo, il quarto, e cosìvia.

Il descrittore di un token è una struttura cosìdefinita:

```
/* Descrittore di un token:
 * - Il campo tag individua il tipo di token.
 *   Contiene il valore della costante corrispondente al tipo di token
 *   o la costante ERR per segnalare un errore.
 *   Osservazione: per i simboli speciali, la costante associata
 *   a ciascuno di essi e' pari al corrispondente char
 *   eg, la costante LSQ che individua una quadra aperta e' pari al
 *   carattere '['.
 * - nel caso di un identificatore, il campo id contiene la stringa
 *   dell'identificatore. Per semplicita', supponiamo che gli
 *   identificatori non superino i MAX_ID_LEN caratteri.
 */
struct descr_token {
    int tag; // il tag che individua il token
    char id[MAX_ID_LEN]; // stringa con il nome nel caso di identificatore
};
```

Come indicato nel commento alla definizione della struttura il campo `tag` contiene una valore intero che individua il token letto. Per questo motivo nel file `pinodefs.h` è definita una costante per ciascuno dei token del linguaggio PINO, in base alla corrispondenza riportata nelle seguenti tabelle, dove nella prima riga è riportato il nome della costante e nella seconda riga il corrispondente token.

ID	HEAD	TAIL	ITE	DFUN	DVAR	EVAL
id	**hd**	**tl**	**ite**	**deffun**	**defvar**	**eval**

NIL	LSQ	RSQ	LANG	RANG	LBR	RBR	COMMA	SEMI	COL	DOT	EQ
*	[	]	⟨	⟩	(	)	,	;	:	.	=

Il valore delle costanti non è importante per la scrittura dell'analizzatore sintattico. Per semplificare la scrittura del codice si è però fatto in modo che il valore del tag corrispondente ad un simbolo speciale é il codice ASCII del simbolo speciale.

La costante `ID` indica che il token è un identificatore e che la stringa corrispondente si trova nel campo `id` del descrittore. Si osservi che tale campo è un vettore di lunghezza `MAX_ID_LEN` (si ricordi che abbiamo assunto che gli identificatori abbiano lunghezza inferiore a `MAX_ID_LEN`).

In aggiunta alle costanti nelle tabelle, in `pinodefs.h` è definita una costante `ERR` che segnala un errore lessicale nell'input (ad esempio, si è letto un carattere non alfanumerico diverso da uno dei simboli speciali, oppure si è trovato un carattere numerico come primo carattere del token).

Il problema del lookahead nella `next_token`

Una piccola difficoltà che si può incontrare nella scritture della `next_token` è legata al fatto che per riconoscere la fine di un identificatore (o di una parola chiave) si deve procedere con la lettura dell'input fino a che non trova un carattere non alfanumerico. Ciò potrebbe portare a delle difficoltà nel caso in cui la `next_token` acquisisse un carattere per volta dallo stream di input. Infatti, supponiamo di dover leggere alfa(, ci si accorgerebbe che il token dell'identificatore alfa è terminato solo dopo aver letto il carattere (, che però fa parte del secondo token. Ciò significa che alla seconda chiamata, la `next_token`, dovendo ritornare il token (, dovrebbe ricordarsi che nella precedente chiamata aveva già letto il carattere (. In pratica, in certi casi sarebbe utile poter vedere qual è il successivo carattere di input (lookahead) senza però cancellarlo dallo stream di input (in modo che una successiva operazione di lettura ritorni proprio tale carattere). Normalmente, una volta letto ed eliminato dallo stream di input, un carattere non può essere letto una seconda volta mediante una delle funzioni di input standard. Il linguaggio C fornisce però la funzione `ungetc`, che il manuale fornisce la seguente descrizione:

> `ungetc() pushes c back to stream, cast to unsigned char, where it is available for subsequent read operations. Pushed - back characters will be returned in reverse order; only one pushback is guaranteed.`

La funzione `ungetc` risolve quindi il problema del lookahead di un carattere. Infatti, quando ci si accorge che l'ultimo carattere letto non appartiene al token che si sta analizzando, ma a quello successivo, è sufficiente rispedire il carattere nello stream di input con una `ungetc`. Un altro approccio al problema è quello di utilizzare un buffer nel quale leggere un'intera riga di programma ed utilizzando un puntatore che scandisce tale buffer per sapere qual è il successivo carattere da analizzare. In questo modo, un accorto uso del puntatore di scansione permette di risolvere il problema del lookahead in modo molto semplice. A tale scopo nel file `pinodefs.h` è definita la struttura:

```
/* Struttura per la implementazione di un buffer
 * Suppongo che il contenuto del buffer sia terminato da \0
 */
struct buf_t {
    char buf[MAX_LN_LEN] ; // il buffer vero e proprio
    char *pos; // posizione nel buffer: punta il successivo ch da leggere
};
```

Volendo leggere una riga alla volta in un buffer del precedente tipo, si deve assumere che le righe siano di lunghezza inferiore a `MAX_LN_LEN` (una costante anch'essa definita in `pinodefs.h`).

Alcuni suggerimenti

Per riconoscere se un carattere è uno dei simboli speciali, in `pinodefs.c` è definita come variabile globale la stringa

```
char *spec_chars ="[]<>();:,.=*"; // stringa con i caratteri speciali di PINO
```

Per riconoscere se un carattere è un carattere speciale è pertanto sufficiente vedere se tale carattere è contenuto in `spec_chars`.

Invece, per riconoscere se una certa sequenza di caratteri alfanumerici è una parola riservata, in `pinodefs.c` è definita ed inizializzata la tavola

```
struct tavola tav_keywords = { // tavola delle parole chiave
    6, {"hd", "tl", "ite", "deffun", "defvar", "eval"}
};
```

Per riconoscere se una certa stringa è una parola chiave è pertanto sufficiente vedere se tale stringa appare nella tavola `tav_keywords`.

2.2 Verifica

Per verificare l'analizzatore lessicale si utilizzerà il main riportato in Appendice D.

Se si sta lavorando su di un sistema linux con compilatore gcc, supponendo che il nome del file che contiene il codice delle funzioni che implementano l'analizzatore lessicale è `lexan.c` (si ricorda comunque che l'unica funzione di questo modulo utilizzata dagli altri moduli del progetto è la `next_token`), il comando per la compilazione del programma di test è:

```
gcc -g pino-II.c pinodefs.c tavole.c lexan.c -o pino-II
```

Come risultato della compilazione si otterrà l'eseguibile `pino-II` che legge da input una sequenza di linee di input ed individua i token di PINO che le compongono, stampando per ciascun token riconosciuto un tag che distingue il tipo di token letto seguito, nel caso di identificatore, dalla stringa dell'identificatore.

Si osservi che per verificare l'analizzatore lessicale serve il modulo della tavola dei simboli (il file `tavole.c`). Se non si è implementato tale modulo o si è inviato un modulo non funzionante, si utilizzerà un file `tavole.c` sviluppato dal docente.

3 Modulo III: L'analizzatore sintattico (AS)

L'analizzatore lessicale (modulo II) trasforma la sequenza di input dell'interprete in una sequenza di token, ma non esegue alcun controllo sulla correttezza dell'ordine dei token. Ad esempio, un input del tipo `[ * alfa )`, essendo composto da token validi è lessicamente corretto, ma non può apparire in nessun programma PINO. Più precisamente, la precedente sequenza di token non rispetta la sintassi di PINO e pertanto non è sintatticamente corretta. Questo tipo di errori devono essere individuati durante la cosiddetta *analisi sintattica* o *parsing*[1].

Le regole della sintassi di PINO sono già state descritte nell'introduzione al progetto. Per comodità le riportiamo nuovamente in Figura 2.

Il modulo che implementa l'analisi sintattica del progetto è l'analizzatore sintattico o *parser* del linguaggio PINO. Oltre a verificare che le sequenze di token del programma sono legali rispetto alle regole in Figura 2, il parser deve costruire la rappresentazione interna delle strutture sintattiche analizzate.

[1] to *parse*: to resolve (as a sentence) into component parts of speech and describe them grammatically. (Primo significato come verbo transitivo riportato sul Merrian-Webster Collegiate Dictionary.)

$$
\begin{array}{lcll}
l & ::= & * \mid [l_0, \ldots, l_k] & (5)\\
e & ::= & l \mid \langle e_0, \ldots, e_k\rangle \mid \langle e_0, \ldots, e_k \texttt{ : } e_{k+1}\rangle & (6)\\
 & & \mid \textbf{hd}(e) \mid \textbf{tl}(e) \mid \textbf{ite}(e_0, e_1, e_2) & \\
 & & \mid x \mid f(e_1, \ldots, e_k) & \\
c & ::= & \textbf{defvar}\, x \;=\; e \mid \textbf{deffun}\, f(x_1, \ldots, x_k) \;=\; e \mid \textbf{eval}\, e & (7)\\
p & ::= & c_1\texttt{; } \ldots\texttt{; } c_k\texttt{.} & (8)
\end{array}
$$

Figura 2: La sintassi di PINO

3.1 Le liste

Partiamo dal caso più semplice, quello delle costanti di PINO, le liste. In base alla regola (5) in Figura 2, per le liste si hanno due casi:

1. la lista vuota `*`;
2. la lista $[l_0, \ldots, l_k]$ costituita dalla sequenza ordinata di $k+1$ liste $l_0, \ldots, l_k$.

Per la memorizzazione delle liste si utilizzerà la seguente struttura di tipo nodo/puntatore definita nel file `pinodefs.h`.

```
/* Struttura per la memorizzazione delle liste di PINO */
struct list {
    struct list *hd; // la testa della lista
    struct list *tl; // la coda della lista
};
```

3.2 Le espressioni

Per quanto riguarda le espressioni, in base alla regola (6) in Figura 2, possiamo individuare i seguenti casi:

1. l'espressione è una lista l;
2. l'espressione è ottenuta componendo le espressioni $e_0, \ldots, e_k$ ed e per formare le espressioni racchiuse tra parentesi angolose $\langle e_0, \ldots, e_k\rangle$ oppure $\langle e_0, \ldots, e_k \texttt{ : } e\rangle$;
3. l'espressione è ottenuta applicando una delle funzioni predefinite **hd**, **tl** o **ite** ad un opportuno numero di espressioni (gli argomenti della funzione);
4. l'espressione è una variabile x, nel qual caso occorre distinguere se
   (a) x è locale, ovvero si sta analizzando un'espressione alla destra del segno $=$ di una **deffun** e la variabile x è nella lista dei parametri della funzione f della **deffun**;
   (b) x è globale, ovvero la variabile x non è locale ma è stata precedentemente definita mediante una **defvar**;
5. l'espressione è ottenuta applicando una funzione f definita dall'utente a k espressioni $e_1, \ldots, e_k$.

In base alle precedenti considerazioni, nel file `pinodefs.h`, è definita la seguente struttura dati.

```
/* Nodo dell'albero per la memorizzazione delle espressioni */
struct expr {
    int tag; /* Il tag che individua il tipo di espressione */
    union {
        // Valore di base di tipo lista
```

```
        struct list *ls;
        // Lista di espressioni
        struct {
            /* Lista di espressioni della forma
             *   <e0, ..., ek : e>
             * il campo le contiene la lista delle espressioni e0, ..., ek
             * il campo exp contiene l'espressione e
             * Notare che
             *   <e0, ..., ek>
             * corrisponde al caso in cui exp e' NULL
             */
            struct lsexpr *le;
            struct expr *exp;
        } lexp;
        /* hd o tl: memorizzo l'argomento */
        struct expr *arg;
        /* ite: memorizzo i suoi argomenti in un vettore di tre elementi */
        struct expr **args;
        /* Variabile locale
         * Di una variabile locale si memorizza l'indice della
         * corrispondente posizione nella lista dei parametri
         * formali della funzione
         */
        int ofs;
        /* Variabile globale: memorizzo il puntatore al descrittore */
        struct descr_var *pdv;
        /* Funzione definita dall'utente:
         * memorizzo il puntatore al descrittore
         * e i suoi k argomenti in un vettore di lunghezza k
         */
        struct {
            struct descr_fun *pdf;
            struct expr **args; // argomenti della funzione
        } fun;
    } u;
};
```

Dove il tipo `struct lsexpr` è così definito (in `pinodefs.h`):

```
/* Lista di espressioni */
struct lsexpr {
    struct expr *exp; // espressione in testa alla lista
    struct lsexpr *next; // la coda della lista di espressioni
}
```

Come si può vedere, la struttura per la memorizzazione delle espressioni è un albero con nodi di diverse forme. Ogni nodo dell'albero è una struttura formata da un tag che indica la forma del nodo ed una union con i campi corrispondenti a tutte le possibili forme del nodo. Di conseguenza, dato un nodo, il valore contenuto nel suo campo tag determina quale campo della union del nodo è quello significativo.

Nel file `pinodefs.h` sono definite le seguenti costanti da utilizzare per distinguere i nodi in base alla classificazione precedentemente vista:

```
/* I tag per distinguere i nodi di una espressione */
#define LIST   0x01 // lista
#define LEXP   0x02 // lista di espressioni
#define HDFUN  0x03 // funzione predefinita hd
#define TLFUN  0x04 // funzione predefinita tl
```

```
#define ITEFUN 0x05 // funzione predefinita ite
#define GVAR   0x06 // variabile globale
#define FUN    0x07 // funzione definita dall'utente
#define LVAR   0x08 // variabile locale
```

Corrispondentemente alla precedente analisi della struttura delle espressioni, la union `u` di `struct expr` viene utilizzata nel seguente modo:

1. se l'espressione è una lista, il campo `ls` della union contiene il puntatore a tale lista;
2. se l'espressione è una lista di espressioni racchiusa tra parentesi angolose, ad esempio $\langle e_0, \ldots, e_k : e \rangle$, la struttura `lexp` contiene nel campo `le` la lista delle espressioni $e_0, \ldots, e_k$ e nel campo `exp` l'espressione e; nel caso in cui, l'espressione è della forma $\langle e_0, \ldots, e_k \rangle$ il campo `exp` della struttura `lexp` è `NULL`;
3. se l'espressione è del tipo **hd**(e) o **tl**(e), il campo `arg` contiene l'espressione e, mentre se l'espressione è della forma **ite**(e_0, e_1, e_2), il campo `args` fa riferimento ad un vettore di tre espressioni contenente nella posizione i l'argomento e_i della funzione;
4. se l'espressione è
   (a) una variabile locale, il campo `ofs` contiene il numero d'ordine della variabile nell'intestazione della funzione, cominciando a contare da 0—ad esempio, nell'espressione alla destra del simbolo $=$ di una **deffun** $f(a, b, c)$, alla variabile a corrisponde l'indice (offset) 0, alla variabile b l'indice 1, alla variabile c l'indice 2;
   (b) una variabile globale, il campo `pdv` fa riferimento al descrittore della variabile nella corrispondente tavola dei simboli;
5. se l'espressione è del tipo $f(e_1, \ldots, e_k)$, il campo `pdf` della struttura `fun` fa riferimento al descrittore della funzione f nella corrispondente tavola, mentre il campo `args` fa riferimento ad un vettore di k espressioni contenente l'argomento e_i nella posizione di indice $i - 1$.

3.3 Lettura ed analisi delle espressioni

Per costruire l'albero di un'espressione si devono scrivere le seguenti due funzioni (più le altre funzioni ausiliarie che si ritiene opportuno definire):

```
struct list *parse_list();
/* Analizza e costruisce una lista della forma [l_0, ..., l_k]
 * Il token [ e' stato letto prima della chiamata della funzione.
 * Ritorna il puntatore alla lista letta o NULL in caso di errore.
 */

struct expr *parse_expr(struct tavola *lvars);
/* Analizza e costruisce l'albero di una espresssione.
 * Se si sta analizzando l'espressione alla destra dell'= di una
 * deffun, lvars punta alla tavola delle variabili locali contenuta
 * nel descrittore della funzione definita dalla deffun sotto esame,
 * negli altri casi lvars e' NULL.
 * Usa le variabili globali ls_dfun per la lista dei descrittori di
 * funzione e ls_dvar per la lista dei descrittori di variabili globali.
 * Ritorna la radice dell'espressione letta o NULL in caso di errore.
 */
```

La funzione `parse_expr` legge un'espressione verificandone la correttezza sintattica. Si osservi che tale funzione dovrà richiamare la `next_token` dell'analizzatore lessicale ed in base al valore ottenuto riconoscere che tipo di nodo deve costruire—ovvero, in quale dei casi separati da | della definizione di espressione si trova. In pratica, la `parse_expr` o costruisce il nodo di una variabile, o richiama la funzione per la lettura delle liste, oppure

richiama ricorsivamente la `parse_expr` controllando che tra le occorrenze delle espressioni ci siano i corretti caratteri di separazione.

Vediamo un esempio in dettaglio. Se il primo token che la `parse_expr` trova è un identificatore, per prima cosa la funzione dovrà verificare se si tratta di una variabile (locale o globale) o di una funzione cercando l'identificatore nelle tavole dei simboli. A tale scopo si noti che la `parse_expr` ha come parametro di input un puntatore alla tavola in cui cercare le eventuali variabili locali, mentre per i simboli di funzione e di variabile globale si usano le due variabili

```
/* Lista dei descrittori delle funzioni */
lista_dfun ls_dfun;

/* Lista dei descrittori delle variabili globali */
lista_dvar ls_dvar;
```

contenute in `pinodefs.c` e dichiarate esterne in `pinodefs.h`. Quindi, ritornando al nostro esempio, per prima cosa occorre vedere se si tratta di una variabile locale, poi si può passare a vedere se è una variabile globale o una funzione. Supponiamo di determinare che si tratta di un simbolo di funzione, occorre eseguire i seguenti passi: (*i*) verificare che il successivo token è '`(`'; (*ii*) leggere la lista degli argomenti della funzione richiamando `parse_expr` fino a che il token successivo all'espressione letta è '`,`'; (*iii*) verificare che l'espressione è correttamente chiusa da una '`)`'. Se tutto va bene, alla fine la `parse_expr` avrà letto e costruito l'albero di un'espressione della forma $f(e_1, \ldots, e_k)$.

Si osservi che la funzione `parse_list` non considera il caso di lista vuota, ma solo quello di liste della forma $[l_0, \ldots l_k]$. Infatti, il caso di lista vuota può essere direttamente gestito dalla `parse_expr` non appena si accorge che il primo token letto è '`*`'.

L'ultimo token letto

In `pinodefs.c` è definita la variabile globale

```
/* Descrittore dell'ultimo token letto dalla next_token */
struct descr_token last_tk;
```

che può essere utilizzata per memorizzare l'ultimo token letto durante l'analisi sintattica e semplificare alcune operazioni del parser. Ad esempio, il progetto non richiede nessuna particolare gestione degli errori; l'uso della variabile globale `last_tk` combinato con alcune modifiche all'analizzatore lessicale per memorizzare nel token la sua posizione nella sequenza di input (numero di riga e colonna) permettono di implementare una prima semplice ed efficace gestione degli errori che segnala il punto dell'input in cui è stato individuato un errore.

La variabile `last_tk` è dichiarata `extern` in `pinodefs.h`.

3.4 I comandi

Oltre alle funzioni in sezione 3.3 si devono scrivere le funzioni

```
struct descr_fun *parse_deffun();
/* Analizza un comando deffun
 * Ritorna il descrittore alla funzione letta o NULL in caso di errore
 * Il token deffun e' stato letto prima della chiamata della funzione.
 */

struct descr_var *parse_defvar();
/* Analizza un comando defvar
 * Ritorna il descrittore alla variabile globale letta o NULL in caso di errore
 * Il token defvar e' stato letto prima della chiamata della funzione.
 */
```

che analizzano ed eseguono le operazioni corrispondenti ai comandi **deffun** e **defvar**. Si osservi che tali funzioni vengono chiamate dopo aver letto il token corrispondente al comando da eseguire. Pertanto il primo token che entrambe le funzioni si aspettano di leggere in input è un identificatore. Nel caso della **deffun**, l'identificatore della funzione da definire; nel caso della **defvar**, l'identificatore della variabile da definire.

La funzione `parse_defvar` dovrà creare il descrittore della variabile che si vuole definire e memorizzare l'espressione associata alla variabile nel corrispondente campo. La funzione `parse_deffun` dovrà: (*i*) creare il descrittore della funzione che si vuole definire; (*ii*) inizializzare la tavola contenuta nel descrittore della funzione ed inserirvi gli identificatori dei parametri della funzione (si noti che, in base alle convenzioni assunte, l'indice di una variabile locale è proprio la sua posizione in questa tavola); (*iii*) associare alla funzione l'espressione letta alla destra dell'uguale.

Il comando **eval** non richiede una particolare analisi sintattica e verrà implementato direttamente nel modulo di valutazione delle espressioni.

3.5 Allocazione della memoria

In previsione della successiva implementazione del garbage collector, si raccomanda che tutte le allocazioni di memoria per la costruzione dei nodi di liste ed espressioni (incluse le allocazioni di vettori per i nodi con tag `ITEFUN` e `FUN`) non siano implementate invocando direttamente la funzione `malloc`, ma bensì attraverso la funzione

```
void *gc_alloc(size_t size);
/* Funzione per l'allocazione della memoria.
 * E' un involucro che richiama la malloc di sistema ed esegue alcune
 * operazioni necessarie per l'implementazione del garbage collector.
 * 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).
 */
```

contenuta in `pinodefs.c` ed il cui prototipo è riportato in `pinodefs.h`. Per il momento tale funzione non fa altro che richiamare la `malloc`; nel modulo del garbage collector, la `gc_alloc` diverrà un involucro per una serie di operazioni necessarie all'implementazione della garbage collection.

3.6 Verifica

Per verificare l'analizzatore sintattico si utilizzerà il main riportato in Appendice E.

Questo programma di verifica non controlla la correttezza delle espressioni e delle liste costruite dalla `parse_expr` e `parse_list`, nè se la `parse_defvar` o la `parse_deffun` inseriscono correttamente nella relativa tavola dei simboli l'identificatore definito dalle corrispondente **defvar** o **deffun**. La verifica di questa parte dell'analizzatore sintattico richiede le funzioni di stampa del modulo IV e potrà essere completata solo dopo l'implementazione di tale modulo.

Se si sta lavorando su di un sistema linux con compilatore gcc, supponendo che il nome del file che contiene il codice delle funzioni che implementano l'analizzatore lessicale è `syntan.c` e che i moduli I e II sono contenuti nei file `tavole.c` e `lexan.c`, il comando per la compilazione del programma di test è:

```
gcc -g pino-III.c pinodefs.c tavole.c lexan.c syntan.c -o pino-III
```

Come risultato della compilazione si otterrà l'eseguibile `pino-III` che, attraverso una maschera, permette di eseguire dei test sulle quattro funzioni principali dell'analizzatore sintattico.

Si osservi che per verificare l'analizzatore sintattico servono i moduli della tavola dei simboli e dell'analizzatore lessicale. Se tali moduli non sono stati implementati, o sono stati inviati moduli non funzionanti, si utilizzeranno i corrispondenti moduli sviluppati dal docente.

4 Modulo IV: Le funzioni di stampa (PR)

Le principali funzioni di stampa sono quelle per la stampa delle liste e delle espressioni. Si dovranno pertanto implementare le funzioni

```
void print_list(struct list *ls);
/* Stampa la lista ls.
 */
```

che stampa una lista e

```
void print_expr(struct expr *exp, struct tavola *lvars);
/* Stampa l'espressione exp.
 * Se l'espressione e' quella assegnata ad una funzione mediante una deffun
 * allora lvars fa riferimento alla tavola delle variabili locali di tale
 * funzione, altrimenti lvars e' NULL
 */
```

che stampa un'espressione.

Entrambe le precedenti funzioni devono produrre un output corretto secondo la sintassi di PINO—cosa particolarmente importante se si vuole usare il risultato di una stampa come successivo input dell'interprete.

La `print_expr`, oltre all'espressione da stampare, riceve come parametro di input un puntatore `lvars` ad una tavola. Questo parametro serve per stampare l'espressione associata ad una funzione f dichiarata mediante una **deffun**, ed in tale caso dovrà fare riferimento alla tavola dei parametri locali contenuta nel descrittore di f. Negli altri casi invece, questo parametro dovrà essere pari a `NULL`.

Oltre alle precedenti funzioni si dovranno implementare le funzioni che stampano i descrittori di variabili e funzioni

```
void print_defvar(struct descr_var *pdv);
/* Stampa il contenuto di una variabile globale
 * sotto forma di defvar
 */

void print_deffun(struct descr_fun *pdf);
/* Stampa il contentuto di un descrittore di funzione
 * sotto forma di deffun
 */
```

Anche in questo caso l'output prodotto dovrà essere corretto rispetto alla sintassi di PINO. In particolare, nel caso del descrittore di una variabile di nome x precedentemente assegnata ad una lista `[*, [*]]`, l'output prodotto dalla `print_defvar` dovrà essere una **defvar** del tipo

```
defvar x = [*, [*]]
```

Invece, nel caso del descrittore di una funzione f di tre argomenti di nome (nell'ordine in cui appaiono nella lista dei parametri) x, y e z, precedentemente associata all'espressione $\langle x, y \texttt{ : } z\rangle$, l'output prodotto dalla `print_deffun` dovrà essere una **deffun** del tipo

```
deffun f(x, y, z) = <x, y : z>
```

Ovviamente, per implementare le quattro funzioni di stampa sopra specificate si potranno definire tutte le funzioni ausiliarie che si ritengono necessarie.

4.1 Verifica

Per verificare l'analizzatore sintattico si utilizzerà il main riportato in Appendice F.

Questo programma di verifica è molto simile a quello utilizzato per la verifica dell'analizzatore sintattico. Anzi, assumendo che le funzioni di stampa sono corrette, serve anche come verifica della correttezza delle espressioni e delle liste costruite dalla `parse_expr` e `parse_list` e per verificare che la `parse_defvar` e la `parse_deffun` inseriscono correttamente le variabili e le funzioni definite dalle **defvar** e dalle **deffun** nelle corrispondenti tabelle.

Se si sta lavorando su di un sistema linux con compilatore gcc, supponendo che il nome del file che contiene il codice delle funzioni di stampa è `stampa.c` e che i moduli I, II e III sono contenuti nei file `tavole.c`, `lexan.c` e `syntan.c`, il comando per la compilazione del programma di test è:

```
gcc -g pino-IV.c pinodefs.c tavole.c lexan.c syntan.c stampa.c -o pino-IV
```

Come risultato della compilazione si otterrà l'eseguibile `pino-IV` che, attraverso una maschera, permette di eseguire dei test sulle quattro principali funzioni di stampa.

Si osservi che per questo programma di verifica servono i moduli della tavola dei simboli, dell'analizzatore lessicale e dell'analizzatore sintattico. Se tali moduli non sono stati implementati, o sono stati inviati moduli non funzionanti, si utilizzeranno i corrispondenti moduli sviluppati dal docente.

5 Modulo V: Lo stack (ST)

Gli elementi nello stack o pila che si deve implementare sono puntatori generici, quindi di tipo `void *`. Lo stack deve essere implementato mediante un vettore di memoria pari al numero massimo di elementi memorizzabili nello stack. Per questo motivo in `pinodefs.h` è definita la seguente struttura

```
/* Struttura per l'implementazione di uno stack di puntatori
 * mediante un vettore di memoria
 */
struct stack {
    size_t size; // dimensione dello stack (num max di puntatori nello stack)
    size_t nump; // numero di puntatori memorizzati nello stack
    void **vect; // vettore di memoria per memorizzare i puntatori,
                 // viene allocato dalla init_stack
};
```

Prima di poter essere utilizzato, uno stack va inizializzato, assegnandogli una dimensione ed allocando la memoria necessaria a contenere gli elementi (il campo `vect` della struttura). L'inizializzazione dello stack è eseguita dalla funzione

```
void *init_stack(struct stack *pst, size_t n);
/* Inizializza lo stack di puntatori *pst
 * allocando il vettore di memoria dello stack per
 * contenere sino a n puntatori
 * Ritorna un puntatore alla base del vettore di memoria
 * che implementa lo stack, o NULL in caso di errore
 */
```

Oltre ai campi `vect` e `size`, la struttura che implementa lo stack contiene il campo `nump` che indica il numero di elementi nello stack.

Per inserire ed eliminare elementi dallo stack si devono implementare le funzioni

```
void *push(struct stack *pst, unsigned n);
/* Inserisce n nuovi puntatori nelo stack *pst
 * Il valore dei puntatori inseriti nello stack e' indefinito,
 * in pratica, crea solo lo spazio per scrivere n nuovi puntatori
```

```
 * in testa allo stack
 * Ritorna il puntatore all'elemento del vettore in testa allo stack
 * dopo l'inserimento, o NULL in caso di errore (stack overflow)
 */

void *pop(struct stack *pst, unsigned n);
/* Elimina n puntatori dallo stack *pst
 * Ritorna il puntatore all'elemento del vettore in testa allo stack
 * dopo la cancellazione, o NULL in caso di errore (se non ci sono n elementi
 * da cancellare)
 */
```

che, rispettivamente, creano lo spazio per n puntatori in cima alla pila e rimuovono n puntatori dalla pila. Si osservi che la push non inserisce nessun elemento in testa alla pila. Dopo l'esecuzione di una `push(pst, n)`, in testa alla pila ci sono n posti che contengono puntatori dal valore indefinito, il contenuto dovrà essere inserito dopo l'esecuzione della push per mezzo della funzione `top`.

L'ultima funzione da implementare è la

```
void *top(struct stack *pst, unsigned n);
/* Ritorna il puntatore all'elemento del vettore in posizione n-esima
 * rispetto alla testa dello stack, o NULL in caso di errore (se lo stack
 * contiene meno di n+1 elementi)
 */
```

che permette di prendere il puntatore all'elemento in posizione n rispetto alla testa della pila. Si osservi che la funzione non deve ritornare il contenuto dela posizione a distanza n dalla testa della pila (ad esempio, della testa della pila se $n = 0$), ma il puntatore alla posizione n della pila.

5.1 Verifica

Per verificare le funzioni che implementano lo stack si utilizzerà il main riportato in Appendice G.

Se si sta lavorando su di un sistema linux con compilatore gcc, supponendo che il nome del file che contiene il codice delle funzioni che implementano lo stack è `stack.c` il comando per la compilazione del programma di test è:

```
gcc -g pino-V.c pinodefs.c stack.c -o pino-V
```

Come risultato della compilazione si otterrà l'eseguibile `pino-V` che, attraverso una maschera, permette di inserire in una pila dei puntatori agli elementi di un vettore definito nel programma di test, di eliminare elementi dalla pila, di leggere il valore riferito in posizione n-esima rispetto alla testa della pila e di stampare il contenuto della pila (i valori puntati dagli elementi nella pila).

6 Modulo VI: La valutazione delle espressioni (EV)

Il risultato della valutazione di un'espressione di PINO è una lista calcolata in base alle seguenti regole:

1. se l'espressione è una lista l, il risultato della valutazione dell'espressione è la lista l;
2. se l'espressione è della forma $\langle e_0, \ldots, e_k : e\rangle$, il risultato è la lista che si ottiene appendendo la lista l alla lista $[l_0, \ldots, l_k]$, dove l_i è il risultato della valutazione di e_i, per $i = 0, \ldots, k$, ed l è il risultato della valutazione di e (nel caso particolare $\langle e_0, \ldots, e_k\rangle$, il risultato è semplicemente $[l_0, \ldots, l_k]$);
3. se l'espressione è della forma **hd**(e) oppure **tl**(e) ed l è la lista che si ottiene valutando e, se l non è vuota il risultato è l'elemento di testa di e altrimenti, se l è vuota, si verifica un errore;

4. se l'espressione è della forma **ite**(e_0, e_1, e_2) ed l_0 è la lista che si ottiene valutando e_0, se l_0 non è vuota il risultato è la lista l_1 che si ottiene valutando e_1 altrimenti, se l_0 è vuota, il risultato è la lista l_2 che si ottiene valutando e_2;
5. se l'espressione è una variabile x, allora
   (a) se x è una variabile globale, il risultato è la lista l ottenuta valutando l'espressione e associata ad x nella sua definizione (per maggiori dettagli si veda la sezione 6.2);
   (b) se x è una variabile locale, il risultato è la lista ottenuta dalla valutazione dell'espressione associata al parametro x della funzione di cui si sta valutando il corpo (si veda la valutazione delle chiamate di funzione al punto 6 e nella sezione 6.3);
6. se l'espressione è una chiamata di funzione $f(e_1, \ldots, e_k)$ con la funzione f definita mediante il comando **defvar** $f(x_0, \ldots, x_{k-1})$ ed l_{i-1} è la lista ottenuta dalla valutazione di e_i, il risultato è la lista che si ottiene valutando l'espressione e, assumendo che il valore della variabile locale di indice (offset) i è pari ad l_i (per i dettagli su come implementare il passaggio di parametri usando lo stack si veda la sezione 6.3).

Il trattamento dei casi 1, 2 e 4 non richiede particolari commenti. Gli altri punti richiedono invece alcune considerazioni aggiuntive che verranno analizzate nelle sezioni 6.2 e 6.3.

Il precedente procedimento di valutazione (ovviamente ricorsivo) deve essere implementato per mezzo della funzione

```
struct list *eval_expr(struct expr *exp);
/* Valuta un'espressione
 * In caso di errore durante la valutazione (ad esempio, una
 * hd o tl applicata ad una lista vuota o in caso di stack overflow)
 * segnala l'errore nella variabile globale error assegnandogli
 * il codice dell'errore (un valore diverso da 0)
 * Altrimenti, ritorna la lista corrispondete all'espressione valutata
 * e la variabile error viene lasciata uguale a 0.
 */
```

più le altre funzioni ausiliarie che si ritengono necessarie.

6.1 Errori durante la valutazione

Nel caso di applicazione di una funzione **hd** o **tl** ad una lista vuota si ha un errore (si veda il caso 3 delle regole di valutazione). La funzione `eval_expr` segnala le situazioni di errore assegnando un valore diverso da 0 alla variabile globale

```
/* Segnala l'eventuale errore verificatosi durante la valutazione
 * In caso di errore durante la valutazione contiene il codice
 * dell'errore (un valore diverso da 0), altrimenti contiene 0
 */
unsigned error;
```

che in caso di valutazione senza errori deve contenere invece il valore 0.

Il caso di lista vuota come argomento a **hd** o **tl** non è l'unico caso di errore. Un'altra situazione di errore si può avere quando a causa di un eccessivo numero di chiamate di funzione (probabilmente dovuto ad una ricorsione troppo grande) si esaurisce lo stack in cui vengono memorizzati i parametri locali delle chiamate di funzione (vedi sezione 6.3).

Nel file `pinodefs.h` sono definite due costanti con due codici distinti corrispondenti ai due casi di errore prima visti.

```
/* Codici degli errori che si possono verificare durante la valutazione */
#define NULL_LIST_ERR 0x01
#define STACK_OVERFLOW 0x02
```

In `pinodefs.c` (dichiarato esterno in `pinodefs.h`) viene anche definito il seguente vettore di messaggi di errore

```
/* Messaggi di errore
 * Il messaggio corrispondente al codice di errore i,
 * si trava nella posizione di indice i del vettore
 */
char *error_msg[] = {
    "",
    "hd o tl applicata a lista vuota",
    "stack overflow"
};
```

Il messaggio corrispondente all'errore di codice i si trova nella posizione i di `error_msg`.

6.2 Valutazione delle variabili globali

Durante la fase di parsing, nel descrittore di una variabile x definita mediante una **defvar** è stato memorizzata l'espressione e assegnata ad x dalla suddetta **defvar**. Disponendo ora della funzione che valuta le espressioni, subito dopo la definizione della variabile si vuole associare a questa anche la lista l corrispondente alla valutazione di e. Per questo motivo, al descrittore delle variabili è stato aggiunto un campo per la memorizzazione di tale lista.

```
/* Descrittore di variabile */
struct descr_var { // descrittore di variabile globale
    char id[MAX_ID_LEN]; // il nome della var
    struct expr *exp; // l'espressione associata alla var
    struct list *ls; // valore dell'espressione associata alla var
};
```

Durante la valutazione di una espressione, si può quindi assumere che, quando si arriva a valutare una variabile globale con puntatore `pdv` al suo descrittore, il campo `pdv->ls` contiene il valore (la lista) della variabile.

6.3 Valutazione delle chiamate di funzione

La valutazione di una chiamata di funzione $f(e_1, \ldots, e_k)$, caso 6 delle regole di valutazione, è di sicuro il caso che richiede maggior cura nell'implementazione. Per prima cosa, osserviamo che, prima di poter passare alla valutazione dell'espressione e associata alla funzione f occorre memorizzare i valori delle espressioni $e_1, \ldots, e_k$ passate come argomenti nelle chiamate di f e creare un legame tra tali valori e le variabili locali di f. Per tale scopo nel file `pinodefs.c` è definito (dichiarato esterno in `pinodefs.h`) lo stack di puntatori

```
/* Lo stack per la memorizzazione dei valori delle variabili locali
 * ad ogni chiamata di funzione
 */
struct stack lvars_stack;
```

Al momento della chiamata della funzione, le espressioni $e_1, \ldots, e_k$ sono valutate ed i loro valori (i puntatori alle liste) $l_0, \ldots, l_{k-1}$ (supponiamo che il valore di e_i è la lista l_{i-1}), sono memorizzati nelo stack, in modo che il puntatore a distanza 0 dalla testa dello stack fa riferimento alla lista l_0, che quello a distanza 1 fa riferimento alla lista l_1, etc. In questo modo, nell'espressione e associata ad f, il valore della variabile locale di indice (offset) i si trova nell'elemento a distanza i dalla testa dello stack. Al termine della valutazione dell'espressione e, i k valori assegnati ai parametri di f vanno rimossi dallo stack. Infatti, supponendo che la chiamata della funzione f si trovi all'interno dell'espresione e' associata ad una chiamata di funzione $g(e'_1, \ldots, e'_h)$, al momento della chiamata di f i valori che si trovavano in testa allo stack erano quelli assegnati ai parametri della g dalla sua chiamata; durante la valutazione di e, i valori in testa allo stack sono quelli assegnati ai parametri di f dalla chiamata di funzione $f(e_1, \ldots, e_k)$; al termine della valutazione di e, occorre ripristinare i valori dei parametri di g eliminando dallo stack queli di f.

6.4 Sulla ridefinizione di variabili e funzioni

Per semplicità, assumiamo che in un programma PINO non accada mai che un identificatore definito come variabile o funzione sia poi ridefinito mediante una nuova **defvar** o **deffun**.

Infatti, trattare correttamente la ridefinizione di funzioni porta a dover gestire situazioni abbastanza complicate. Ad esempio, supponiamo di aver inserito il comando **deffun** $f(a) = \langle a \rangle$ e di aver usato la funzione f nella successiva definizione di una funzione g—ad esempio, **deffun** $g(a) =$ **hd**$(f(a))$. Se adesso decidiamo di ridefinire la funzione f trasformandola in una funzione con due argomenti—ad esempio, **deffun** $f(a, b) = \langle a$ **:** $b \rangle$—la definizione della funzione g non è più corretta (in g, la f era usata come funzione di un argomento). Il precedente problema può essere risolto mantenendo entrambe le versioni della f: la g continuerà a far riferimento alla prima versione con un argomento; i successivi usi della f faranno invece riferimento alla seconda definizione con due argomenti. Problemi analoghi si possono verificare nella ridefinizione di una variabile.

Per evitare di dover fronteggiare troppi problemi tecnici di questo tipo, nel progetto, assumiamo l'ipotesi semplificatrice che una variabile o funzione non sia mai definita due volte.

6.5 Verifica

Per verificare la funzion `eval_expr` si utilizzerà il main riportato in Appendice H.

Se si sta lavorando su di un sistema linux con compilatore gcc, supponendo che il nome del file che contiene il codice del valutatore di espressioni è `eval.c` e che i precedenti moduli sono contenuti nei file `tavole.c`, `lexan.c`, `syntan.c`, `stampa.c` e `stack.c` il comando per la compilazione del programma di test è:

```
gcc -g pino-III.c pinodefs.c tavole.c lexan.c syntan.c stampa.c stack.c eval.c -o pino-III
```

Come risultato della compilazione si otterrà l'eseguibile `pino-VI` 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 `->`.

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

7 Modulo VII: Il garbage collector (GC)

Per capire i compiti del garbage collector, si prenda l'espressione **tl**(e) con $e = \langle l_1$ **:** $[l_2] \rangle$. 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*.[2]

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

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

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 implementabile.[3] Per questo motivo, il compito di gestire il recupero della garbage è affidato ad un opportuno modulo detto *garbage collector*.

7.1 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 *`,

[3] 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.

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`.

7.2 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 7.5).

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 7.5.

7.3 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 3). 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.

7.4 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.

7.5 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 7.3) 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).

7.6 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.

7.7 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.

7.8 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 7.9 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.

7.9 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 7.9.

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.

8 Esempi di programmi PINO

In questa sezione diamo alcuni esempi di funzioni ricorsive definibili in PINO. Questi esempi saranno utilizzati per verificare i moduli.

La prima funzione che prendiamo in considerazione è quella che verifica se due liste sono uguali. La funzione ritorna $[*]$ se le due liste di input sono uguali e $*$ se non lo sono.

deffun $eq(l_1, l_2) =$
  ite(l_1,
    ite(l_2,
      ite(eq(**hd**(l_1), **hd**(l_2)),
        ite(eq(**tl**(l_1), **tl**(l_2)), $[*]$, $*$),
        $*$),
      $*$),
    ite(l_2, $*$, $[*]$));

La funzione *append* concatena due liste, appendendo la seconda in fondo alla prima.

deffun $append(l_1, l_2)$ = **ite**(l_1, $\langle$**hd**(l_1) **:** $append$(**tl**(l_1), l_2)$\rangle$, l_2);

La prossima funzione esegue un appiattimento della lista di input. Ad esempio, data una lista in cui il simbolo $*$ appare cinque volte (a qualsiasi livello di profondità), il risultato della funzione *flat* su tale lista è la lista che si ottiene concatenando cinque volte $*$, quindi, $flat([*, [*, [*, *], *]]) = [*, *, *, *, *]$. In modo equivalente, possiamo dire che la funzione *flat* cancella tutte le parentesi quadre all'interno di una lista (ovviamente, le parentesi più esterne rimangono).

deffun $flat(l)$ = **ite**(l, $append$(**ite**(**hd**(l), $flat$(**hd**(l)), $[*]$), $flat$(**tl**(l))), $*$);

Definiamo infine una funzione utile per creare liste di prova.

deffun $dupnil(l)$ = **ite**(l, $\langle$**ite**(**hd**(l), $dupnil$(**hd**(l)), $[*, *]$) **:** $dupnil$(**tl**(l))$\rangle$, $*$);

La funzione *dupnil* sostituisce tutti i simboli $*$contenuti all'interno di una lista (a qualsiasi livello di profondità) sono rimpiazzati dalla lista $[*, *]$. Per mezzo della funzione *dupnil* si possono facilmente costruire liste molto lunghe, basta tener conto che ad ogni applicazione della *dupnil*, la dimensione del risultato è circa il doppio.

8.1 Un esempio di sessione

Riportiamo qui di seguito un esempio di sessione con l'interprete PINO senza garbage collection (modulo VI).

Dopo aver definito le funzioni descritte precedentemente

```
# deffun append(l1, l2) = ite(l1, <hd(l1) : append(tl(l1), l2)>, l2);
deffun append(l1, l2) =
  ite(l1, <hd(l1) : append(tl(l1), l2)>, l2)
# deffun eq(l1, l2) =
      ite(l1,
          ite(l2,
              ite(eq(hd(l1), hd(l2)),
                  ite(eq(tl(l1), tl(l2)), [*], *),
                  *),
              *),
```

```
          ite(l2, *, [*]));
deffun eq(l1, l2) =
  ite(l1, ite(l2, ite(eq(hd(l1), hd(l2)), ite(eq(tl(l1), tl(l2)), [*], *), *), *),
ite(l2, *, [*]))
# deffun dupnil(l) =
       ite(l, < ite(hd(l), dupnil(hd(l)), [*, *]) : dupnil(tl(l))>, *);
deffun dupnil(l) =
  ite(l, <ite(hd(l), dupnil(hd(l)), [*, *]) : dupnil(tl(l))>, *)
# deffun flat(l) =
  ite(l, append(ite(hd(l), flat(hd(l)), [*]), flat(tl(l))), *);
deffun flat(l) =
  ite(l, append(ite(hd(l), flat(hd(l)), [*]), flat(tl(l))), *)
```

engono definite alcune variabili usando la funzione *append*.

```
# defvar a = [[*], *];
defvar a =
  [[*], *]
-> [[*], *]
# defvar b = [*, [*]];
defvar b =
  [*, [*]]
-> [*, [*]]
# defvar c = append(a, b);
defvar c =
  append(a, b)
-> [[*], *, *, [*]]
# defvar d = append(b, a);
defvar d =
  append(b, a)
-> [*, [*], [*], *]
```

Si osservi che le variabili c e d contengono valori diversi, dato che sono ottenute concatenando le liste a a b con un diverso ordine. Quindi $eq(c, d)$ deve dare $*$.

```
# eval eq(c, d);
eq(c, d)
-> *
# eval eq(flat(dupnil(c)), flat(dupnil(d)));
eq(flat(dupnil(c)), flat(dupnil(d)))
-> [*]
```

In ogni caso però che c e d contengono lo stesso numero di simboli $*$ e che tale proprietà rimane vera se applichiamo *dupnil* ad entrambe le liste. Quindi, $eq(flat(dupnil(c)), flat(dupnil(d)))$ da correttamente $[*]$.

I successivi esempi sono simili a quelli precedentemente visti, ma eseguiti su liste più lunghe.

```
# defvar e = <dupnil(c), dupnil(d), dupnil(c) : dupnil(d)>;
defvar e =
  <dupnil(c), dupnil(d), dupnil(c) : dupnil(d)>
-> [[[[*, *]], [*, *], [*, *], [[*, *]]], [[*, *], [[*, *]], [[*, *]], [*, *]],
[[[*, *]], [*, *], [*, *], [[*, *]]], [*, *], [[*, *]], [[*, *]], [*, *]]
# defvar f = <dupnil(d), dupnil(c), dupnil(c) : dupnil(c)>;
defvar f =
  <dupnil(d), dupnil(c), dupnil(c) : dupnil(c)>
-> [[[*, *], [[*, *]], [[*, *]], [*, *]], [[[*, *]], [*, *], [*, *], [[*, *]]],
[[[*, *]], [*, *], [*, *], [[*, *]]], [[*, *]], [*, *], [*, *], [[*, *]]]
# eval eq(flat(e), flat(f));
eq(flat(e), flat(f))
-> [*]
```

A Il file `pinodefs.h`

```
/* -*- Mode: C -*- */
/* Time-stamp:  <pinodefs.h 03/07/04 15:58:44 guerrini@zambujero.lan.home> */

/*************************************************************
 * Progetto di Laboratorio di Programmazione
 * Stefano Guerrini - AA 2002-03
 *
 * Header con le principali definizioni.
 *************************************************************/

/*************************************************************
 * Modulo I: Le tavole dei simboli (TS)
 */

/*
 * Tavole implementate con un vettore
 */

#define MAX_KEYS    16
#define MAX_ID_LEN  32
#define MAX_LN_LEN 256

/* Struttura per la rappresentazione di una tavola mediante un vettore */
struct tavola {
    unsigned n; // elementi nella tavola
    char *keys[MAX_KEYS]; // vettore con le stringhe nella tavola
};

void init_tavola(struct tavola *ptv);
/* inizializza la tavola *ptv */

unsigned avail_tavola(struct tavola *ptv);
/* numero di spazi ancora disponibili nella tavola *ptv */

int ins_key_tavola(struct tavola *ptv, char *key);
/* aggiunge la chiave key alla tavola se non gia' presente
 * ritorna l'indice della posizione della chiave inserita
 * o -1 nel caso in cui key e' gia' presente nella tavola o
 * non c'e' posto per inserirla
 */

int search_key_tavola(struct tavola *ptv, char *key);
/* cerca la chiave key nella tavola *ptv
 * ritorna la posizione della chiave nella tavola
 * o -1 se la chiave non \`e presente
 */

/*
 * Tavole implementate con liste
 */

/* Descrittore di funzione */
struct descr_fun {
    char id[MAX_ID_LEN]; // il nome della funzione
    struct tavola tv; // tavola con il nome degli argomenti della funzione
```

```
    struct expr *exp; // l'espressione associata alla funzione
};

/* Nodo lista descrittori di funzioni */
struct nodo_lista_dfun {
    struct descr_fun *pdf;
    struct nodo_lista_dfun *next;
};

/* Lista descrittori di funzioni */
typedef struct nodo_lista_dfun *lista_dfun;

/* Descrittore di variabile */
struct descr_var { // descrittore di variabile globale
    char id[MAX_ID_LEN]; // il nome della var
    struct expr *exp; // l'espressione associata alla var
    struct list *ls; // valore dell'espressione associata alla var
};

/* Nodo lista descrittori di variabile */
struct nodo_lista_dvar {
    struct descr_var *pdv;
    struct nodo_lista_dvar *next;
};

/* Lista descrittori di variabile */
typedef struct nodo_lista_dvar *lista_dvar;

struct descr_fun *ins_key_dfun(lista_dfun *pls, char *id);
/* aggiunge un nuovo descrittore per id alla lista *pls
 * ritorna il puntatore al nuovo descrittore inserito
 * o NULL in caso di errore o nel caso in cui id e'
 * gia' presente in *pls
 */

struct descr_fun *search_key_dfun(lista_dfun ls, char *id);
/* cerca il descrittore di funzione di id in ls
 * ritorna il puntatore al descrittore trovato
 * o NULL se non presente
 */

struct descr_var *ins_key_dvar(lista_dvar *pls, char *id);
/* aggiunge un nuovo descrittore per id alla lista *pls
 * ritorna il puntatore al nuovo descrittore inserito
 * o NULL in caso di errore o nel caso in cui id e'
 * gia' presente in *pls
 */

struct descr_var *search_key_dvar(lista_dvar ls, char *id);
/* cerca il descrittore di variabile di id in ls
 * ritorna il puntatore al descrittore trovato
 * o NULL se non presente
 */


/*********************************************************
 * Modulo II: L'analizzatore lessicale (AL)
```

```
 */

/* variabile globale inizializzata con le parole chiave di PINO */
extern struct tavola tav_keywords;
/* variabile globale con i caratteri speciali di PINO */
extern char *spec_chars;

/* I tag dei token */
#define ERR   0x00

#define ID    0x01

#define HEAD  0x02
#define TAIL  0x03
#define ITE   0x04
#define DFUN  0x05
#define DVAR  0x06
#define EVAL  0x07

#define NIL   '*'
#define LSQ   '['
#define RSQ   ']'
#define LANG  '<'
#define RANG  '>'
#define LBR   '('
#define RBR   ')'
#define COMMA ','
#define SEMI  ';'
#define COL   ':'
#define DOT   '.'
#define EQ    '='

/* Descrittore di un token:
 * - Il campo tag individua il tipo di token.
 *   Contiene il valore della costante corrispondente al tipo di token
 *   o la costante ERR per segnalare un errore.
 *   Osservazione: per i simboli speciali, la costante associata
 *   a ciascuno di essi e' pari al corrispondente char
 *   eg, la costante LSQ che individua una quadra aperta e' pari al
 *   carattere '['.
 * - nel caso di un identificatore, il campo id contiene la stringa
 *   dell'identificatore. Per semplicita', supponiamo che gli
 *   identificatori non superino i MAX_ID_LEN caratteri.
 */
struct descr_token {
    int tag; // il tag che individua il token
    char id[MAX_ID_LEN]; // stringa con il nome nel caso di identificatore
};

/* Struttura per la implementazione di un buffer
 * Suppongo che il contenuto del buffer sia terminato da \0
 */
struct buf_t {
    char buf[MAX_LN_LEN] ; // il buffer vero e proprio
    char *pos; // posizione nel buffer: punta il successivo ch da leggere
};
```

```
void next_token(struct descr_token *pdtk);
/* Ritorna in *pdtk il prossimo token nello stream di input */


/*************************************************************
 * Modulo III: L'analizzatore sintattico (AS)
 */

/* I tag per distinguere i nodi di una espressione */
#define LIST   0x01 // lista
#define LEXP   0x02 // lista di espressioni
#define HDFUN  0x03 // funzione predefinita hd
#define TLFUN  0x04 // funzione predefinita tl
#define ITEFUN 0x05 // funzione predefinita ite
#define GVAR   0x06 // variabile globale
#define FUN    0x07 // funzione definita dall'utente
#define LVAR   0x08 // variabile locale

/* Struttura per la memorizzazione delle liste di PINO */
struct list {
    struct list *hd; // la testa della lista
    struct list *tl; // la coda della lista
};

/* Lista di espressioni */
struct lsexpr {
    struct expr *exp; // espressione in testa alla lista
    struct lsexpr *next; // la coda della lista di espressioni
};

/* Nodo dell'albero per la memorizzazione delle espressioni */
struct expr {
    int tag; /* Il tag che individua il tipo di espressione */
    union {
         // Valore di base di tipo lista
        struct list *ls;
        // Lista di espressioni
        struct {
            /* Lista di espressioni della forma
             *   <e0, ..., ek : e>
             * il campo le contiene la lista delle espressioni e0, ..., ek
             * il campo exp contiene l'espressione e
             * Notare che
             *   <e0, ..., ek>
             * corrisponde al caso in cui exp e' NULL
             */
            struct lsexpr *le;
            struct expr *exp;
        } lexp;
        /* hd o tl: memorizzo l'argomento */
        struct expr *arg;
        /* ite: memorizzo i suoi argomenti in un vettore di tre elementi */
        struct expr **args;
        /* Variabile locale
         * Di una variabile locale si memorizza l'indice della
         * corrispondente posizione nella lista dei parametri
         * formali della funzione
```

```
         */
        int ofs;
        /* Variabile globale: memorizzo il puntatore al descrittore */
        struct descr_var *pdv;
        /* Funzione definita dall'utente:
         * memorizzo il puntatore al descrittore
         * e i suoi k argomenti in un vettore di lunghezza k
         */
        struct {
            struct descr_fun *pdf;
            struct expr **args; // argomenti della funzione
        } fun;
    } u;
};

/*
 * Variabili globali per la memorizzazione delle tavole dei
 * descrittori di funzione e di variabile
 */

/* Lista dei descrittori delle funzioni */
extern lista_dfun ls_dfun;

/* Lista dei descrittori delle variabili globali */
extern lista_dvar ls_dvar;

/* Descrittore dell'ultimo token letto dalla next_token */
extern struct descr_token last_tk;

/*
 * Le principali funzioni implementate dall'analizzatore sintattico
 */

struct list *parse_list();
/* Analizza e costruisce una lista della forma [l_0, ..., l_k]
 * Il token [ e' stato letto prima della chiamata della funzione.
 * Ritorna il puntatore alla lista letta o NULL in caso di errore.
 */

struct expr *parse_expr(struct tavola *lvars);
/* Analizza e costruisce l'albero di una espresssione.
 * Se si sta analizzando l'espressione alla destra dell'= di una
 * deffun, lvars punta alla tavola delle variabili locali contenuta
 * nel descrittore della funzione definita dalla deffun sotto esame,
 * negli altri casi lvars e' NULL.
 * Usa le variabili globali ls_dfun per la lista dei descrittori di
 * funzione e ls_dvar per la lista dei descrittori di variabili globali.
 * Ritorna la radice dell'espressione letta o NULL in caso di errore.
 */

struct descr_fun *parse_deffun();
/* Analizza un comando deffun
 * Ritorna il descrittore alla funzione letta o NULL in caso di errore
 * Il token deffun e' stato letto prima della chiamata della funzione.
 */

struct descr_var *parse_defvar();
```

```
/* Analizza un comando defvar
 * Ritorna il descrittore alla variabile globale letta o NULL in caso di errore
 * Il token defvar e' stato letto prima della chiamata della funzione.
 */


/************************************************************
 * Modulo IV: Le funzioni di stampa (PR)
 */

void print_list(struct list *ls);
/* Stampa la lista ls.
 */

void print_expr(struct expr *exp, struct tavola *lvars);
/* Stampa l'espressione exp.
 * Se l'espressione e' quella assegnata ad una funzione mediante una deffun
 * allora lvars fa riferimento alla tavola delle variabili locali di tale
 * funzione, altrimenti lvars e' NULL
 */

void print_defvar(struct descr_var *pdv);
/* Stampa il contenuto di una variabile globale
 * sotto forma di defvar
 */

void print_deffun(struct descr_fun *pdf);
/* Stampa il contentuto di un descrittore di funzione
 * sotto forma di deffun
 */


/************************************************************
 * Modulo V: Lo stack (ST)
 */

/* Struttura per l'implementazione di uno stack di puntatori
 * mediante un vettore di memoria
 */
struct stack {
    size_t size; // dimensione dello stack (num max di puntatori nello stack)
    size_t nump; // numero di puntatori memorizzati nello stack
    void **vect; // vettore di memoria per memorizzare i puntatori,
                 // viene allocato dalla init_stack
};

void *init_stack(struct stack *pst, size_t n);
/* Inizializza lo stack di puntatori *pst
 * allocando il vettore di memoria dello stack per
 * contenere sino a n puntatori
 * Ritorna un puntatore alla base del vettore di memoria
 * che implementa lo stack, o NULL in caso di errore
 */

void *push(struct stack *pst, unsigned n);
/* Inserisce n nuovi puntatori nelo stack *pst
 * Il valore dei puntatori inseriti nello stack e' indefinito,
```

```
 * in pratica, crea solo lo spazio per scrivere n nuovi puntatori
 * in testa allo stack
 * Ritorna il puntatore all'elemento del vettore in testa allo stack
 * dopo l'inserimento, o NULL in caso di errore (stack overflow)
 */

void *pop(struct stack *pst, unsigned n);
/* Elimina n puntatori dallo stack *pst
 * Ritorna il puntatore all'elemento del vettore in testa allo stack
 * dopo la cancellazione, o NULL in caso di errore (se non ci sono n elementi
 * da cancellare)
 */

void *top(struct stack *pst, unsigned n);
/* Ritorna il puntatore all'elemento del vettore in posizione n-esima
 * rispetto alla testa dello stack, o NULL in caso di errore (se lo stack
 * contiene meno di n+1 elementi)
 */


/************************************************************
 * Modulo VI: La valutazione delle espressioni (EV)
 */

/* Segnala l'eventuale errore verificatosi durante la valutazione
 * In caso di errore durante la valutazione contiene il codice
 * dell'errore (un valore diverso da 0), altrimenti contiene 0
 */
extern unsigned error;

/* Codici degli errori che si possono verificare durante la valutazione */
#define NULL_LIST_ERR 0x01
#define STACK_OVERFLOW 0x02

/* Messaggi di errore
 * Il messaggio corrispondente al codice di errore i,
 * si trava nella posizione di indice i del vettore
 */
extern char *error_msg[];

/* Lo stack per la memorizzazione dei valori delle variabili locali
 * ad ogni chiamata di funzione
 */
extern struct stack lvars_stack;

struct list *eval_expr(struct expr *exp);
/* Valuta un'espressione
 * In caso di errore durante la valutazione (ad esempio, una
 * hd o tl applicata ad una lista vuota o in caso di stack overflow)
 * segnala l'errore nella variabile globale error assegnandogli
 * il codice dell'errore (un valore diverso da 0)
 * Altrimenti, ritorna la lista corrispondete all'espressione valutata
 * e la variabile error viene lasciata uguale a 0.
 */

/************************************************************
 * Modulo VII: Il garbage collector (GC)
```

```
 */

/* -- Nei moduli da I a VI --
 * La funzione gc_alloc non deve essere implementata.
 * Il suo codice si trova in pinodefs.c
 * -- Nel modulo VII --
 * La gc_alloc deve essere inplementata nel modulo.
 * La versione definita in pinodefs.c viene esclusa utilizzando
 * la compilazione condizionale
 */
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
 */

#ifdef PINO_GC

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

void gc(void);
/* Esegue la garbage collection mediante 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.
 */

/* 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
};

/* La lista dei nodi allocati da gc_alloc */
extern struct header_nodo *lista_nodi_mem;

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

```
 */
extern int parsing;

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

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
 */

#endif
```

B Il file `pinodefs.c`

```
/* -*- Mode: C -*- */
/* Time-stamp:  <pinodefs.c 03/07/05 01:22:17 guerrini@zambujero.lan.home> */

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

/************************************************************
 * Progetto di Laboratorio di Programmazione
 * Stefano Guerrini - AA 2002-03
 *
 * Alcune variabili globali e funzioni di utitilita'
 ************************************************************/


/************************************************************
 * Modulo II: L'analizzatore lessicale (AL)
 */

struct tavola tav_keywords = { // tavola delle parole chiave
    6, {"hd", "tl", "ite", "deffun", "defvar", "eval"}
};

char *spec_chars ="[]<>();:,.=*"; // stringa con i caratteri speciali di PINO

/************************************************************
 * Modulo III: L'analizzatore sintattico (AS)
 */

/* Lista dei descrittori delle funzioni */
lista_dfun ls_dfun = NULL;

/* Lista dei descrittori delle variabili globali */
lista_dvar ls_dvar = NULL;
```

```
/* Descrittore dell'ultimo token letto dalla next_token */
struct descr_token last_tk;

/*************************************************************
 * Modulo VI: La valutazione delle espressioni (EV)
 */

/* Segnala l'eventuale errore verificatosi durante la valutazione
 * In caso di errore durante la valutazione contiene il codice
 * dell'errore (un valore diverso da 0), altrimenti contiene 0
 */
unsigned error;

/* Messaggi di errore
 * Il messaggio corrispondente al codice di errore i,
 * si trava nella posizione di indice i del vettore
 */
char *error_msg[] = {
    "",
    "hd o tl applicata a lista vuota",
    "stack overflow"
};

/* Lo stack per la memorizzazione dei valori delle variabili locali
 * ad ogni chiamata di funzione
 */
struct stack lvars_stack;


/*************************************************************
 * Modulo VII: Il garbage collector (GC)
 */

#ifdef PINO_GC // Definizioni necessarie alla compilazione del GC

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

/* Segnala al GC se si sta eseguo 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;

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

void gc(void) {
    /* Esegue la garbage collection con un algoritmo di mark-andsweep.
     * 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);
}


#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
```

C Modulo I: Verifica delle tavole dei simboli (TS)

```
/* -*- Mode: C -*- */
/* Time-stamp:  <pino-I.c 03/06/19 22:45:50 guerrini@zambujero.lan.home> */

/*************************************************************
* Progetto di Laboratorio di Programmazione
 * Stefano Guerrini - AA 2002-03
 *
 * Main per la verifica del modulo I: le tavole dei simboli (TS)
 * Per la compilazione servono i seguenti file:
 *   pinodefs.h        header principali defs
 *   pinodefs.c        alcune var globali e funzioni di utilita
 *   pino-I.c          questo file
 *   tavole.c          tavole dei simboli - modulo I (TS)
 * Per la compilazione su linux con gcc
 *   gcc -g pino-I.c pinodefs.c tavole.c -o pino-I
 * crea l'eseguibile pino-I con le info necessarie per il debugging
 *************************************************************/

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

void get_word(char *str, unsigned lim) {
    /* legge una sequenza di caratteri alfanumerici
    * di lunghezza inferiore a lim e la memorizza in str */
    int c;
    while (--lim > 0 && (c = getchar()) != EOF && isalnum(c))
        *(str++) = c;
    *str = '\0';
    // elimina gli spazi bianchi dopo la parola fino all'eventuale newline
```

```
    while (c != '\n' && isblank(c))
        c = getchar();
}

void test_key_tavola(struct tavola *ptv) {
    /* verifica tavola implementata con vettore
     * - legge una stringa e prova ad inserirla nella tavola,
     *   in caso di inserimento verifica che indice di ritorno
     *   sia quello della stringa inserita stampandola
     * - prova nuovamante l'inserimento
     *   ci si aspetta che la risposta sia elemento gia' inserito
     *   o che la tavola e' piena
     * - verifica la ricerca
     * termina quando si immette la stringa vuota
     */
    int ind;
    char str[MAX_ID_LEN];
    printf("** Verifica funzioni tavola implementata con vettore **\n");
    init_tavola(ptv);
    printf("Dopo inizializzazione: disponibili %02d posti nella tavola.\n",
           avail_tavola(ptv));
    printf("Inserisci delle stringhe di lungh. max %02d.\n", MAX_ID_LEN);
    printf("[Stringa vuota per finire]\n");
    do {
        printf("tv> ");
        get_word(str, MAX_ID_LEN);
        if (*str == '\0')
            break; // Stringa vuota. Esci dal ciclo
        /* Prova ad inserire la stringa */
        printf("(ins  1) ");
        if ((ind = ins_key_tavola(ptv, str)) < 0)
            printf(avail_tavola(ptv) > 0 ?
                   "Gia' inserito!\n" : "Tavola piena!\n");
        else
            printf("'%s'\n", ptv->keys[ind]);
        /* Prova ad inserire la stringa una seconda volta */
        printf("(ins  2) ");
        if ((ind = ins_key_tavola(ptv, str)) < 0)
            printf(avail_tavola(ptv) > 0 ?
                   "Gia' inserito!\n" : "Tavola piena!\n");
        else
            printf("Errore: ci si aspetta -1 da ins_key_tavola!\n");
        /* Cerca la stringa */
        printf("(search) ");
        if ((ind = search_key_tavola(ptv, str)) < 0)
            printf(avail_tavola(ptv) > 0 ?
                   "Gia' inserito!\n" : "Tavola piena!\n");
        else
            printf("'%s'\n", ptv->keys[ind]);
        /* Stampa info sullo stato della tavola */
        printf("Inseriti %02d elementi\nDisponibili %02d posti\n",
               ptv->n, avail_tavola(ptv));
    } while (-1);
}

void test_tav_dfun(lista_dfun ldf) {
    /* verifica tavola descrittori funzione
```

```
     * opera sulla lista di descrittori di funzione ldf
     * che si suppone gia' inizializzata
     * - legge una stringa e prova ad inserirla nella lista
     *   in caso di inserimento verifica che il descrittore ottenuto
     *   come valore di ritorno sia quello giusto stampandone l'id
     * - prova nuovamante l'inserimento
     *   ci si aspetta che la risposta sia elemento gia' inserito
     * - verifica la ricerca dell'elemento appena inserito
     * termina quando si immette la stringa vuota
     */
    char str[MAX_ID_LEN];
    struct descr_fun *pdf;
    printf("** Verifica funzioni tavola descrittori funzione **\n");
    printf("Inserisci delle stringhe di lungh. max %02d.\n", MAX_ID_LEN);
    printf("[Stringa vuota per finire]\n");
    do {
        printf("df> ");
        get_word(str, MAX_ID_LEN);
        if (*str == '\0')
            break; // Stringa vuota. Esci dal ciclo
        /* Prova ad inserire la stringa */
        printf("(ins  1) ");
        if ((pdf = ins_key_dfun(&ldf, str)) == NULL)
            printf("Gia' inserito!\n");
        else
            printf("'%s'\n", pdf->id);
        /* Prova ad inserire la stringa una seconda volta */
        printf("(ins  2) ");
        if ((pdf = ins_key_dfun(&ldf, str)) == NULL)
            printf("Gia' inserito!\n");
        else
            printf("Errore: la stringa sarebbe dovuta essere nella lista!\n");
        /* Prova la ricerca della stringa, avendola appena inserita
         * mi aspetto sia presente nella tavola */
        printf("(search) ");
        pdf = search_key_dfun(ldf, str);
        printf("'%s'\n", pdf == NULL ?
               "Errore: la stringa sarebbe dovuta essere nella lista!\n" :
               pdf->id);
    } while (-1);
}

void test_tav_dvar(lista_dvar ldv) {
    /* verifica tavola descrittori variabile
     * opera sulla lista di descrittori di varaibile ldv
     * che si suppone gia' inizializzata
     * - legge una stringa e prova ad inserirla nella lista
     *   in caso di inserimento verifica che il descrittore ottenuto
     *   come valore di ritorno sia quello giusto stampandone l'id
     * - prova nuovamante l'inserimento
     *   ci si aspetta che la risposta sia elemento gia' inserito
     * - verifica la ricerca dell'elemento appena inserito
     * termina quando si immette la stringa vuota
     */
    char str[MAX_ID_LEN];
    struct descr_var *pdv;
    printf("** Verifica funzioni tavola descrittori variabili **\n");
```

```
    printf("Inserisci delle stringhe di lungh. max %02d.\n", MAX_ID_LEN);
    printf("[Stringa vuota per finire]\n");
    do {
        printf("dv> ");
        get_word(str, MAX_ID_LEN);
        if (*str == '\0')
            break; // Stringa vuota. Esci dal ciclo
        /* Prova ad inserire la stringa */
        printf("(ins  1) ");
        if ((pdv = ins_key_dvar(&ldv, str)) == NULL)
            printf("Gia' inserito!\n");
        else
            printf("'%s'\n", pdv->id);
        /* Prova ad inserire la stringa una seconda volta */
        printf("(ins  2) ");
        if ((pdv = ins_key_dvar(&ldv, str)) == NULL)
            printf("Gia' inserito!\n");
        else
            printf("Errore: la stringa sarebbe dovuta essere nella lista!\n");
        /* Prova la ricerca della stringa, avendola appena inserita
         * mi aspetto sia presente nella tavola */
        printf("(search) ");
        pdv = search_key_dvar(ldv, str);
        printf("'%s'\n", pdv == NULL ?
               "Errore: la stringa sarebbe dovuta essere nella lista!\n" :
               pdv->id);
    } while (-1);
}

int main(void) {
    /* Chiede, con una maschera, di selezionarte la tavola da verificare
     * Quindi, legge una sequenza di stringhe alfanumeriche di lunghezza
     * inferiore a MAX_ID_LEN e prova inserimento e ricerca nelle tavola
     * selezionata.
     */
    int c;
    struct tavola tv; // per il test della tavola implem con vettore
    lista_dfun ldf = NULL; // per il test della lista descr funzione
    lista_dvar ldv = NULL; // per il test della lista descr variabile
    do {
        printf("Digitare\n");
        printf(" 1 per il test delle tavole mediante vettori (tv)\n");
        printf(" 2 per il test delle liste descr. funzioni (df)\n");
        printf(" 3 per il test delle liste descr. variabili (tv)\n");
        printf(" 0 per terminare: ");
        /* leggi il primo carattere della riga ed ignora il resto */
        while ((c = getchar()) < '0' || c > '3');
        while (getchar() != '\n'); // elimina il resto della riga
        putchar('\n');
        switch (c) {
        case '0':
            exit(0); // termina
            break;
        case '1': /* Prova tavola mediante vettore */
            test_key_tavola(&tv);
            break;
        case '2': /* Prova tavola/lista descrittori di funzione */
```

```
            test_tav_dfun(ldf);
            break;
        case '3': /* Prova tavola/lista descrittori di funzione */
            test_tav_dvar(ldv);
            break;
        }
        putchar('\n');
    } while (-1);
}
```

D Modulo II: Verifica dell'analizzatore lessicale (AL)

```
/* -*- Mode: C -*- */
/* Time-stamp:  <pino-II.c 03/06/19 22:46:01 guerrini@zambujero.lan.home> */

/*********************************************************
* Progetto di Laboratorio di Programmazione
 * Stefano Guerrini - AA 2002-03
 *
 * Main per la verifica del modulo II: l'analizatore lessicale (AL)
 * Per la compilazione servono i seguenti file:
 *   pinodefs.h        header principali defs
 *   pinodefs.c        alcune var globali e funzioni di utilita
 *   pino-II.c         questo file
 *   tavole.c          tavole dei simboli - modulo I (TS)
 *   lexan.c           analizzatore lessicale - modulo II (AL)
 * Per la compilazione su linux con gcc
 *   gcc -g pino-II.c pinodefs.c tavole.c lexan.c -o pino-II
 * crea l'eseguibile pino-II con le info necessarie per il debugging
 *********************************************************/

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

int main(void) {
    int i = 0; // conta i DOT consecutivi
    struct descr_token dtk;
    /* legge una sequenza di token fino a che non trova
     * due '.' consecutivi
     */
    do {
        next_token(&dtk);
        switch (dtk.tag) {
        case ERR:
            printf("Errore sintattico\n");
            exit(1);
            break;
        case ID:
            printf("<ID> %s\n", dtk.id);
            break;
        case HEAD:
            printf("<HEAD>\n");
            break;
        case TAIL:
            printf("<TAIL>\n");
```

```
            break;
        case ITE:
            printf("<ITE>\n");
            break;
        case DFUN:
            printf("<DFUN>\n");
            break;
        case DVAR:
            printf("<DVAR>\n");
            break;
        case EVAL:
            printf("<EVAL>\n");
            break;
        case NIL:
        case LSQ:
        case RSQ:
        case LANG:
        case RANG:
        case LBR:
        case RBR:
        case COMMA:
        case SEMI:
        case COL:
        case DOT:
        case EQ:
            printf("'%c'\n", dtk.tag);
            break;
        default:
            printf("Errore interno\n");
            break;
        }
        if (dtk.tag == DOT) i++; // incrementa num DOT
        else i = 0; // azzera num DOT
    } while (i < 2);
}
```

E Modulo III: Verifica dell'analizzatore sintattico (AS)

```
/* -*- Mode: C -*- */
/* Time-stamp:  <pino-III.c 03/06/20 00:29:27 guerrini@zambujero.lan.home> */

/*************************************************************
* Progetto di Laboratorio di Programmazione
 * Stefano Guerrini - AA 2002-03
 *
 * Main per la verifica del modulo III: l'analizatore sintattico (AS)
 * Per la compilazione servono i seguenti file:
 *   pinodefs.h        header principali defs
 *   pinodefs.c        alcune var globali e funzioni di utilita
 *   pino-III.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)
 * Per la compilazione su linux con gcc
 *   gcc -g pino-III.c pinodefs.c tavole.c lexan.c syntan.c -o pino-III
 * crea l'eseguibile pino-III 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 situzioni 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);
}

void test_parse_list(void) {
    /* Verifica la parse_list.
     * Legge una sequenza di liste separate da ';'.
     * La sequenza e' terminata da '.'
     * Per ogni lista, risponde OK se e' sintatticamente corretta
     * (se parse_list restituisce un valore diverso da NULL ed il token
     * che segue la lista letta e' ';' o '.'), o Errore negli altri casi.
     * Non verifica la correttezza della lista costruita da parse_list.
     */
    int ok;
    printf("Verifica di parse_list\n");
    printf("\tInserire una sequenza di liste.\n");
    printf("\tScrivere ';' dopo la lista da leggere se si vogliono\n");
    printf("\teseguire altri test, oppure '.' se e' l'ultima lista.\n");
    do {
        next_token(&last_tk);
        if (last_tk.tag == LSQ) // lista non vuota, chiama la parse_list
            ok = (parse_list() != NULL);
        else // verifica se si tratta della lista vuota o di token errato
            ok = (last_tk.tag == NIL);
        if (ok)
            // ha letto una lista, ora legge il token che segue la lista
            next_token(&last_tk);
        if (ok && (last_tk.tag == SEMI || last_tk.tag == DOT))
            printf("OK\n");
        else {
            /* c'e' stato un errore nella lettura di una lista, oppure la
             * lista letta non e' seguita da un ';' o da un '.'
             */
            skiptk(&last_tk);
            printf("Errore\n");
        }
    } while (last_tk.tag == SEMI);
}

void test_parse_expr(void) {
    /* Verifica la parse_expr.
     * Legge una sequenza di espressioni separate da ';'.
```

```
     * La sequenza e' terminata da '.'
     * Per ogni espressione, risponde OK se e' sintatticamente corretta
     * (se parse_expr restituisce un valore diverso da NULL ed il token
     * che segue l'espressione letta e' ';' o '.'), o Errore negli altri casi.
     * Non verifica la correttezza dell'espressione costruita da parse_expr.
     * NB. Dato che le tavole dei simboli sono vuote, non verifica il caso
     * di espressioni con variabili o funzioni definite dall'utente.
     */
    printf("Verifica di parse_expr\n");
    printf("\tInserire una sequenza di espressioni.\n");
    printf("\tScrivere ';' dopo l'espressione da leggere se si vogliono\n");
    printf("\teseguire altri test, oppure '.' se e' l'ultima espressione.\n");
    do {
        if (parse_expr(NULL) == NULL)
            printf("Errore\n");
        else { // ha letto un'espressione corretta
            next_token(&last_tk); // legge il token che segue l'espressione
            if (last_tk.tag == SEMI || last_tk.tag == DOT)
                printf("OK\n");
            else {
                // l'espressione letta non e' seguita da ';' o '.'
                skiptk(&last_tk);
                printf("Errore\n");
            }
        }
    } while (last_tk.tag == SEMI);
}

void test_parse_defs(void) {
    /* Verifica parse_deffun e parse_defvar.
     * Legge una sequenza di comandi deffun e defvar separati da ';'.
     * La sequenza e' terminata da '.'
     * Per ogni comando, risponde OK se e' sintatticamente corretto
     * (se la corrispondente parse restituisce un valore diverso da NULL
     * ed il token che segue l'espressione letta alla destra dell'=
     * del comando e' ';' o '.'), o Errore negli altri casi.
     * Non verifica la correttezza del descrittore costruito.
     * NB. Dato che ogni def deve inserire il simbolo definito nella
     * corrispondente tavola, una variabile o una funzione definita
     * da un comando puo' essere usata nei successivi comandi
     */
    int ok;
    printf("Verifica di parse_defvar e parse_deffun\n");
    printf("\tInserire una sequenza di espressioni.\n");
    printf("\tScrivere ';' dopo l'espressione da leggere se si vogliono\n");
    printf("\teseguire altri test, oppure '.' se e' l'ultima espressione.\n");
    do {
        next_token(&last_tk); // legge il token del tipo di definizione
        if (last_tk.tag == DVAR) // si tratta di defvar
            ok = (parse_defvar() != NULL);
        else if (last_tk.tag == DFUN) // si tratta di deffun
            ok = (parse_deffun() != NULL);
        else // token non valido
            ok = 0;
        if (ok) // comando letto correttamente
            next_token(&last_tk); // ora legge il token che segue il comando
        if (ok && (last_tk.tag == SEMI || last_tk.tag == DOT))
```

```
            printf("OK\n");
        else { // il comando letto non e' seguito da ';' o '.'
            skiptk(&last_tk);
            printf("Errore\n");
        }
    } while (last_tk.tag == SEMI);
}

int main(void) {
    int c;
    do {
        printf("Digitare\n");
        printf(" 1 per verificare la parse_list\n");
        printf(" 2 per verificare la parse_expr\n");
        printf(" 3 per verificare parse_deffun e parse_defvar\n");
        printf(" 0 per terminare: ");
        /* leggi il primo carattere della riga ed ignora il resto */
        while ((c = getchar()) < '0' || c > '3');
        while (getchar() != '\n'); // elimina il resto della riga
        putchar('\n');
        switch (c) {
        case '0':
            exit(0); // termina
            break;
        case '1':
            test_parse_list();
            break;
        case '2':
            test_parse_expr();
            break;
        case '3':
            test_parse_defs();
            break;
        }
        putchar('\n');
    } while (-1);
}
```

F Modulo IV: Verifica delle funzioni di stampa (PR)

```
/* -*- Mode: C -*- */
/* Time-stamp:  <pino-IV.c 03/06/23 16:30:56 guerrini@zambujero.lan.home> */

/*************************************************************
* Progetto di Laboratorio di Programmazione
 * Stefano Guerrini - AA 2002-03
 *
 * Main per la verifica del modulo IV: le funzioni di stampa (PR)
 * Per la compilazione servono i seguenti file:
 *   pinodefs.h        header principali defs
 *   pinodefs.c        alcune var globali e funzioni di utilita
 *   pino-IV.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 di stampa - modulo IV (PR)
```

```
 * Per la compilazione su linux con gcc
 *   gcc -g pino-IV.c pinodefs.c tavole.c lexan.c syntan.c stampa.c -o pino-IV
 * crea l'eseguibile pino-IV con le info necessarie per il debugging
 ************************************************************/

#include <stdlib.h>
#include <stdio.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);
}

void test_print_list(void) {
    /* Verifica la print_list.
     * Legge una sequenza di liste separate da ';'.
     * La sequenza e' terminata da '.'
     * Per ogni lista, risponde OK se e' sintatticamente corretta
     * (se parse_list restituisce un valore diverso da NULL ed il token
     * che segue la lista letta e' ';' o '.'), o Errore negli altri casi.
     * Non verifica la correttezza della lista costruita da parse_list.
     */
    int ok;
    struct list *ls;
    printf("Verifica di print_list\n");
    printf("\tInserire una sequenza di liste.\n");
    printf("\tScrivere ';' dopo la lista da leggere e stampare se si vogliono\n");
    printf("\teseguire altri test, oppure '.' se e' l'ultima lista.\n");
    do {
        next_token(&last_tk);
        if (last_tk.tag == LSQ) // lista non vuota, chiama la parse_list
            ok = ((ls = parse_list()) != NULL);
        else if (last_tk.tag == NIL) // lista vuota
            ls = NULL, ok = -1;
        else // token errato
            ok = 0;
        if (ok) { // ha letto una lista
            print_list(ls); // stampa la lista
            printf("\n");
            next_token(&last_tk); // legge il token che segue la lista
        }
        if (ok && (last_tk.tag == SEMI || last_tk.tag == DOT))
            printf("OK\n");
        else {
            /* c'e' stato un errore nella lettura di una lista, oppure la
             * lista letta non e' seguita da un ';' o da un '.'
             */
            skiptk(&last_tk);
```

```
            printf("Errore\n");
        }
    } while (last_tk.tag == SEMI);
}

void test_print_expr(void) {
    /* Verifica la print_expr.
     * Legge una sequenza di espressioni separate da ';'.
     * La sequenza e' terminata da '.'
     * Per ogni espressione, risponde OK se e' sintatticamente corretta
     * (se parse_expr restituisce un valore diverso da NULL ed il token
     * che segue l'espressione letta e' ';' o '.'), o Errore negli altri casi.
     * Non verifica la correttezza dell'espressione costruita da parse_expr.
     * NB. Dato che le tavole dei simboli sono vuote, non verifica il caso
     * di espressioni con variabili o funzioni definite dall'utente.
     */
    struct expr *exp;
    printf("Verifica di print_expr\n");
    printf("\tInserire una sequenza di espressioni.\n");
    printf("\tScrivere ';' dopo l'espressione da leggere e stampare se si vogliono\n");
    printf("\teseguire altri test, oppure '.' se e' l'ultima espressione.\n");
    do {
        if ((exp = parse_expr(NULL)) == NULL)
            printf("Errore\n");
        else { // ha letto un'espressione corretta
            print_expr(exp, NULL); // stampa l'espressione
            printf("\n");
            next_token(&last_tk); // legge il token che segue l'espressione
            if (last_tk.tag == SEMI || last_tk.tag == DOT)
                printf("OK\n");
            else {
                // l'espressione letta non e' seguita da ';' o '.'
                skiptk(&last_tk);
                printf("Errore\n");
            }
        }
    } while (last_tk.tag == SEMI);
}

void test_print_defs(void) {
    /* Verifica print_deffun e print_defvar.
     * Legge una sequenza di comandi deffun e defvar separati da ';'.
     * La sequenza e' terminata da '.'
     * Per ogni comando, risponde OK se e' sintatticamente corretto
     * (se la corrispondente parse restituisce un valore diverso da NULL
     * ed il token che segue l'espressione letta alla destra dell'=
     * del comando e' ';' o '.'), o Errore negli altri casi.
     * Non verifica la correttezza del descrittore costruito.
     * NB. Dato che ogni def deve inserire il simbolo definito nella
     * corrispondente tavola, una variabile o una funzione definita
     * da un comando puo' essere usata nei successivi comandi
     */
    int ok;
    unsigned tag;
    struct descr_var *pdv;
    struct descr_fun *pdf;
    printf("Verifica di print_defvar e print_deffun\n");
```

```
    printf("\tInserire una sequenza di espressioni.\n");
    printf("\tScrivere ';' dopo l'espressione da leggere se si vogliono\n");
    printf("\teseguire altri test, oppure '.' se e' l'ultima espressione.\n");
    printf("\tImmettendo ';' non preceduto da nessuna espressione\n");
    printf("\tviene stampato il contenuto della tavole dei simboli.\n");
    do {
        next_token(&last_tk); // legge il token del tipo di definizione
        tag = last_tk.tag;
        if (tag == DVAR) // si tratta di defvar
            ok = ((pdv = parse_defvar()) != NULL);
        else if (tag == DFUN) // si tratta di deffun
            ok = ((pdf = parse_deffun()) != NULL);
        else if (tag == SEMI) { // si devono stampare le tavole dei simboli
            lista_dvar ldv = ls_dvar;
            lista_dfun ldf = ls_dfun;
            printf("** Variabili **\n");
            while (ldv != NULL) {
                print_defvar(ldv->pdv);
                printf("\n");
                ldv = ldv->next;
            }
            printf("** Funzioni **\n");
            while (ldf != NULL) {
                print_deffun(ldf->pdf);
                printf("\n");
                ldf = ldf->next;
            }
            printf("** ** ** ** **\n");
            ok = -1;
        } else // token non valido
            ok = 0;
        if (ok) { // comando letto correttamente
            if (tag == DVAR) {
                print_defvar(pdv);
                printf("\n");
            } else if (tag == DFUN) {
                print_deffun(pdf);
                printf("\n");
            }
            if (tag != SEMI) // comando non vuoto
                next_token(&last_tk); // legge il token che segue il comando
        }
        if (ok && (last_tk.tag == SEMI || last_tk.tag == DOT)) {
            if (tag != SEMI)
                printf("OK\n");
        } else { // il comando letto non e' seguito da ';' o '.'
            skiptk(&last_tk);
            printf("Errore\n");
        }
    } while (last_tk.tag == SEMI);
}

int main(void) {
    int c;
    do {
        printf("Digitare\n");
        printf(" 1 per verificare print_list\n");
```

```
        printf(" 2 per verificare la print_expr\n");
        printf(" 3 per verificare print_deffun e print_defvar\n");
        printf(" 0 per terminare: ");
        /* leggi il primo carattere della riga ed ignora il resto */
        while ((c = getchar()) < '0' || c > '3');
        while (getchar() != '\n'); // elimina il resto della riga
        putchar('\n');
        switch (c) {
        case '0':
            exit(0); // termina
            break;
        case '1':
            test_print_list();
            break;
        case '2':
            test_print_expr();
            break;
        case '3':
            test_print_defs();
            break;
        }
        putchar('\n');
    } while (-1);
}
```

G Modulo V: Verifica dello stack (ST)

```
 /* -*- Mode: C -*- */
/* Time-stamp:  <pino-V.c 03/06/24 15:39:17 guerrini@zambujero.lan.home> */

/*************************************************************
* Progetto di Laboratorio di Programmazione
 * Stefano Guerrini - AA 2002-03
 *
 * Main per la verifica del modulo V: lo stack (ST)
 * Per la compilazione servono i seguenti file:
 *   pinodefs.h        header principali defs
 *   pinodefs.c        alcune var globali e funzioni di utilita
 *   pino-V.c          questo file
 *   stack.c           stack - modulo V (ST)
 * Per la compilazione su linux con gcc
 *   gcc -g pino-V.c pinodefs.c stack.c -o pino-V
 * crea l'eseguibile pino-V con le info necessarie per il debugging
 *************************************************************/

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

#define DIM 32 // dimensione stack

#define N 16 // per prendere dei numeri quasi casuali
int V[N];

char *keys = ".+-=*"; // comandi del programma di test
```

```
int main(void) {
    int c, n, **p, ok = -1;
    unsigned count = 0; // contatore modulo N dei numeri inseriti
    for (n = 0; n < DIM; n++) // inizializza vettore numeri
        V[n] = n;
    init_stack(&lvars_stack, DIM); // inizializza lo stack
    printf("Lo stack ha dimensione %d\n", DIM);
    printf("Digitare\n");
    printf(" + n per inserire i puntatori a n numeri nella pila\n");
    printf(" - n per eliminare i puntatori a n numeri dalla pila\n");
    printf(" = n per stampare il numero puntato dall'n-esimo elemento in testa alla pila\n");
    printf(" * per stampare i numeri contenuti nella pila\n");
    printf(" . per terminare: \n");
    do {
        printf("$ ");
        while (! index(keys, (c = getchar())));
        switch (c) {
        case '.':
            exit(0); // termina
            break;
        case '+':
            scanf("%d", &n); // numero elementi da inserire
            if (ok = (push(&lvars_stack, n) != NULL)) {
                while (n-- > 0) // memorizza n puntatori ad elementi di V
                    *((int **)top(&lvars_stack, n)) = V+(count++ % N);
                printf("OK! Lo stack contiene %d elementi\n",
                       lvars_stack.nump);
            } else
                printf("Stack overflow\n");
            break;
        case '-':
            scanf("%d", &n); // numero elementi da eliminare
            if (ok = (pop(&lvars_stack, n) != NULL))
                printf("OK! Lo stack contiene %d elementi\n",
                       lvars_stack.nump);
            else
                printf("Errore! Nello stack ci sono solo %d elementi\n",
                       lvars_stack.nump);
            break;
        case '=':
            scanf("%d", &n); // elemento de leggere
            if (ok = ((p = top(&lvars_stack , n)) != NULL)) {
                // stampa il valore puntato dal puntatore preso dallo stack
                printf("%d: %d\n", n, **p);
            }
            else
                printf("Errore! Nello stack ci sono solo %d elementi\n",
                       lvars_stack.nump);
            break;
        case '*':
            // stampa i valori puntati dagli elementi contenuti nello stack
            for (n = 0; n < lvars_stack.nump; n++) {
                printf("%d: %d\n", n, **(int **)top(&lvars_stack, n));
            }
            break;
        }
```

```
        while (getchar() != '\n'); // elimina il resto della riga
    } while (-1);
}
```

H Modulo VI: Verifica del modulo per la valutazione delle espressioni (EV)

```
/* -*- Mode: C -*- */
/* Time-stamp:  <pino-VI.c 03/07/04 23:45:33 guerrini@zambujero.lan.home> */

/*************************************************************
* Progetto di Laboratorio di Programmazione
 * Stefano Guerrini - AA 2002-03
 *
 * Main per la verifica del modulo VI: la valutazione delle espressioni (EV)
 * Per la compilazione servono i seguenti file:
 *   pinodefs.h        header principali defs
 *   pinodefs.c        alcune var globali e funzioni di utilita
 *   pino-VI.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)
 * Per la compilazione su linux con gcc
 *   gcc -g pino-VI.c pinodefs.c tavole.c lexan.c syntan.c stampa.c stack.c eval.c -o pino-VI
 * crea l'eseguibile pino-VI con le info necessarie per il debugging
 *************************************************************/

#include <stdlib.h>
#include <stdio.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;
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 {
    printf("# "); // prompt
    next_token(&last_tk); // legge il token del tipo di comando
    switch (last_tk.tag) {
    case DVAR: // defvar
        if ((pdv = parse_defvar()) == 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
        if ((pdf = parse_deffun()) == 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
        if ((exp = parse_expr(NULL)) == 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);
}
```

I 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);
}
```