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

Descrizione del progetto

Stefano Guerrini

A.A. 2002/03 – Canale P-Z
Versione del 24 giugno 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**.

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
```