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

Modulo VI: La valutazione delle espressioni (EV)

Stefano Guerrini

A.A. 2002/03 – Canale P-Z
Versione del 24 giugno 2003

1 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 $\langle e_0, \ldots, e_k \texttt{ : } e\rangle$, 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 1.2);
   (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 1.3);
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 1.3).

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 1.2 e 1.3.

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.

1.1 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 1.3).

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`.

1.2 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.

1.3 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.

1.4 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.

1.5 Verifica

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

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.

A 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);
}
```