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

Modulo IV: Le funzioni di stampa (PR)

Stefano Guerrini

A.A. 2002/03 – Canale P-Z
Versione del 20 giugno 2003

1 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.

1.1 Verifica

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

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.

A 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);
}
```