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

Modulo III: L'analizzatore sintattico (AS)

Stefano Guerrini

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

1 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 1.

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

Figura 1: La sintassi di PINO

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 1, il parser deve costruire la rappresentazione interna delle strutture sintattiche analizzate.

1.1 Le liste

Partiamo dal caso più semplice, quello delle costanti di PINO, le liste. In base alla regola (1) in Figura 1, 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 $l_0, \ldots, l_k$.

[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.)

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
};
```

1.2 Le espressioni

Per quanto riguarda le espressioni, in base alla regola (2) in Figura 1, 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 $\langle e_0, \ldots, e_k : e \rangle$;
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 $i - 1$.

1.3 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`.

1.4 I comandi

Oltre alle funzioni in sezione 1.3 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.

1.5 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.

1.6 Verifica

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

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.

A 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);
}
```