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

Modulo II: L'analizzatore lessicale (AL)

Stefano Guerrini

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

1 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.

1.1 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`.

1.2 Verifica

Per verificare l'analizzatore lessicale 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 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.

A 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);
}
```