Progetto per il Laboratorio di Programmazione
Un interprete per un piccolo C

Modulo II: L'analizzatore lessicale (AL)

Stefano Guerrini

A.A. 2003/04 – Canale P-Z
Versione del 27 maggio 2004

1 Modulo II: L'analizzatore lessicale (AL)

L'input che l'interprete di CCINO riceve è una sequenza di caratteri. È però evidente che un programma corretto non è composto da sequenze di caratteri arbitrarie, ma da sequenze organizzate in opportune unità lessicali. Per comprendere meglio il concetto, si prenda il caso di un testo, le unità lessicali sono le parole della lingua in cui è scritto il testo ed i simboli di punteggiatura.

Nei programmi CCINO si possono avere le seguenti unità lessicali o *token*:

- i *simboli speciali* ,, ;, (,), {, }, +, −, ∗, /, %, &&, ||, !, ==, ! =, <, >, <=, >=, =, /∗ e ∗/.
- le *parole chiave* **int**, **if**, **else**, **while**, **read**, **write**, **wsp**, **wnl** e **return**;
- i nomi di variabile o funzione, anche detti *identificatori*, definiti come sequenze di caratteri alfanumerici il cui primo carattere è alfabetico;
- i numerali interi con o senza segno.

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. In certi casi lo spazio bianco può essere assente, come la 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 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 lessicali, 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 quello di scrivere l'analizzatore lessicale dei programmi scritti in CCINO.

1.1 La funzione `next_token`

Il cuore dell'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, fornisce 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 così via.

Il descrittore di un token è una struttura così definita:

```
/* 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.
 * - 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.
 * - nel caso di un numero, il campo n contiene il suo valore
 */
struct descr_token {
    int tag; // il tag che individua il token
    int n; // valore del token nel caso di numero intero
    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 `ccinodefs.h` è definita una costante per ciascuno dei token del linguaggio CCINO, in base alla corrispondenza riportata nelle seguenti tabelle, dove nella prima riga è riportato il nome della costante e nella seconda il corrispondente token.

CC_ERR	CC_ID	CC_NUM
errore	*id*	*n*

CC_INT	CC_IF	CC_ELSE	CC_WHILE	CC_READ	CC_WRITE	CC_WSP	CC_WNL	CC_RET
int	**if**	**else**	**while**	**read**	**write**	**wsp**	**wnl**	**return**

CC_COMMA	CC_SEMI	CC_LBR	CC_LBR	CC_LCURLY	CC_RCURLY	CC_PLUS	CC_MINUS
,	;	(	)	{	}	+	−
CC_MULT	CC_DIV	CC_MOD	CC_AND	CC_OR	CC_NOT	CC_EQ	CC_NEQ
*	/	%	&&	||	!	==	!=
CC_LT	CC_GT	CC_LEQ	CC_GEQ	CC_ASS	CC_LCOMM	CC_RCOMM	
<	>	<=	>=	=	/*	*/	

Il valore esatto delle precedenti costanti non è importante per la scrittura dell'analizzatore lessicale.

La costante `CC_ID` indica che il token è un identificatore e che la stringa corrispondente si trova nel campo `id` del descrittore. Si osservi che il campo `id` è un vettore di lunghezza `MAX_ID_LEN`; di conseguenza, gli identificatori devono avere lunghezza inferiore a `MAX_ID_LEN`. Anche la costante `MAX_ID_LEN` è definita in `ccinodefs.h`.

La costante `CC_NUM` indica che il token è un numero con segno il cui valore si trova nel campo `n` del descrittore.

La costante `CC_ERR` segnala un errore lessicale nell'input (ad esempio, si è letto un carattere non alfanumerico diverso da uno dei caratteri contentuti dei simboli speciali).

Il problema del lookahead nella `next_token`

Una piccola difficoltà che si può incontrare nella scrittura 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 si 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 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`, per la quale il manuale il manuale in linea contiene le seguenti informazioni:

```
SYNOPSIS
       #include <stdio.h>

       ...
       int ungetc(int c, FILE *stream);
       ...

DESCRIPTION
       ...
       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 il problema del lookahead di un carattere. Infatti, quando ci si accorge che l'ultimo carattere letto appartiene al token che segue il token che si sta analizzando, è sufficiente rispedire il carattere nello stream di input per mezzo di una `ungetc`.

Un altro approccio che risolve il problema del lookahead è quello di utilizzare un buffer nel quale mantenere la riga di programma che si sta utilizzando. Un accorto uso del puntatore di scansione di questo buffer permette di risolvere il problema molto semplicemente, visto che, se si legge un carattere di troppo, per tornare indietro è sufficiente spostare il puntatore indietro di una posizione (si noti che il precedente inconveniente non può mai capire all'inizio di una riga e quindi non c'è mai il rischio che per tornare indietro si debba ritornare alla precedente riga).

L'utilizzo di un buffer per la lettura delle righe di input ha anche dei vantaggi per quanto riguarda la gestione degli eventuali errori sintattici nel programma di input. Anche se, per semplicità, nel progetto trascureremo il problema degli errori, si suggerisce di adottare la soluzione con buffer di riga. A tale scopo nel file `ccinodefs.h` viene 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
};
```

Si noti che che si sta assumendo che le righe siano di lunghezza inferiore a `MAX_LN_LEN` (una costante anch'essa definita in `ccinodefs.h`).

Alcuni suggerimenti

Per riconoscere se una certa sequenza di caratteri alfanumerici è una parola riservata, in `ccinodefs.c` è definita ed inizializzata la tavola

```
struct tavola tav_keywords = { // tavola delle parole chiave
    9, {"int", "if", "else", "while", "read", "write", "wsp", "wnl", "return"}
};
```

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 ccino-II.c ccinodefs.c tavole.c lexan.c -o ccino-II
```

Come risultato della compilazione si otterrà l'eseguibile `ccino-II` che legge da input una sequenza di linee di input ed individua i token di CCINO 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:  <ccino-II.c 04/05/28 01:09:14 stefano@sgport> */

/*************************************************************
* 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:
 *   ccinodefs.h       header principali defs
 *   ccinodefs.c       alcune var globali e funzioni di utilita
 *   ccino-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 ccino-II.c ccinodefs.c tavole.c lexan.c -o ccino-II
 * crea l'eseguibile ccino-II con le info necessarie per il debugging
 *************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "ccinodefs.h"

int main(void) {
    struct descr_token dtk;
    /* legge una sequenza di token
     * termina in caso di token errato (errore lessicale)
     * o se riceve il comando di terminazione ^C
     */
    do {
        next_token(&dtk);
        switch (dtk.tag) {
        case CC_ERR:
            printf("Errore lessicale\n");
            exit(1);
            break;
        case CC_ID:
```

```
        printf("<ID> %s\n", dtk.id);
        break;
    case CC_NUM:
        printf("<NUM> %d\n", dtk.n);
        break;
    case CC_INT:
        printf("<INT>\n");
        break;
    case CC_IF:
        printf("<IF>\n");
        break;
    case CC_ELSE:
        printf("<ELSE>\n");
        break;
    case CC_WHILE:
        printf("<WHILE>\n");
        break;
    case CC_READ:
        printf("<READ>\n");
        break;
    case CC_WRITE:
        printf("<WRITE>\n");
        break;
    case CC_WSP:
        printf("<WSP>\n");
        break;
    case CC_WNL:
        printf("<WNL>\n");
        break;
    case CC_RET:
        printf("<RET>\n");
        break;
    case CC_COMMA:
        printf("<COMMA>\n");
        break;
    case CC_SEMI:
        printf("<SEMI>\n");
        break;
    case CC_LBR:
        printf("<LBR>\n");
        break;
    case CC_RBR:
        printf("<RBR>\n");
        break;
    case CC_LCURLY:
        printf("<LCURLY>\n");
        break;
    case CC_RCURLY:
        printf("<RCURLY>\n");
        break;
    case CC_PLUS:
        printf("<PLUS>\n");
        break;
    case CC_MINUS:
        printf("<MINUS>\n");
        break;
    case CC_MULT:
```

```
            printf("<MULT>\n");
            break;
        case CC_DIV:
            printf("<DIV>\n");
            break;
        case CC_MOD:
            printf("<MOD>\n");
            break;
        case CC_AND:
            printf("<AND>\n");
            break;
        case CC_OR:
            printf("<OR>\n");
            break;
        case CC_NOT:
            printf("<NOT>\n");
            break;
        case CC_EQ:
            printf("<EQ>\n");
            break;
        case CC_NEQ:
            printf("<NEQ>\n");
            break;
        case CC_LT:
            printf("<LT>\n");
            break;
        case CC_GT:
            printf("<GT>\n");
            break;
        case CC_LEQ:
            printf("<LEQ>\n");
            break;
        case CC_GEQ:
            printf("<GEQ>\n");
            break;
        case CC_ASS:
            printf("<ASS>\n");
            break;
        case CC_LCOMM:
            printf("<LCOMM>\n");
            break;
        case CC_RCOMM:
            printf("<RCOMM>\n");
            break;
        default:
            printf("Errore interno\n");
            break;
        }
    } while (-1);
}
```