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

Descrizione del progetto

Stefano Guerrini

A.A. 2003/04 – Canale P-Z
Versione del 26 maggio 2004

Il linguaggio

Lo scopo del progetto è quello di realizzare un interprete per il linguaggio CCINO, una versione molto semplificata del linguaggio C.

A parte le funzioni per l'input/output, CCINO è un sottoinsieme del C; in particolare, ogni programma in CCINO è un programma C corretto e, aggiungendo le definizioni delle funzioni di input/output, può essere compilato ed eseguito come un qualsiasi programma C.

Sintassi astratta

La sintassi di CCINO verrà definita per mezzo della cosiddetta *sintassi astratta*. In particolare, assoceremo un nome a ciascuna delle categorie sintattiche che vogliamo definire—ad esempio, useremo e per denotare le espressioni, f per denotare i prototipi o le definizioni di funzione, ecc. Per indicare gli elementi di una categoria sintattica useremo la lettera che definisce la categoria eventualmente seguita da un indice—ad esempio, con $e_1, \ldots, e_k$ indicheremo una sequenza di k elementi della categoria e, ovvero k espressioni.

La definizione (ricorsiva) di una certa categoria verrà data associando mediante il simbolo ::= una sequenza di possibili costruzioni sintattiche separate dal simbolo | ad ogni nome di categoria; nelle costruzioni sintattiche si potranno usare elementi di un'altra categoria sintattica o dei simboli appartenenti all'alfabeto di CCINO (che nel seguito indicheremo sempre in grassetto).

Numeri ed identificatori

Dato che l'unico tipo di dato sono gli interi, le uniche costanti che possono apparire in un programma in CCINO sono i numeri interi con o senza segno, nel seguito indicati con n. Un numero intero è una sequenza di cifre decimali eventualemente precedute dal segno $-$.

Gli identificatori sono delle sequenze di caratteri alfanumerici in cui il primo carattere è alfabetico. Nel seguito distingueremo gli identificatori per i nomi di variabile, che indicheremo con x, e quelli per i nomi di funzione, che indicheremo con g.

Le espressioni

Tutte le espressioni di CCINO sono di tipo intero. Il caso più semplice di espressione è un numero intero n. Un altro esempio di espressione atomica è l'espressione composta da una variabile x. Espressioni composte possono essere ottenute per mezzo di operatori binari o unari. Gli operatori si suddividono in

- operatori *aritmetici* $+$, $-$, $*$, $/$ e $\%$;
- operatori *booleani* **!**, **&&** and ||;
- operatori *di confronto* $==$, $!=$, $<$, $>$, $<=$ and $>=$.

Tutti i precedenti operatori sono binari, ad eccezione dell'operazione **!** che è unario.

Siccome tutte le funzioni definite in un programma CCINO devono ritornare un valore intero, una chiamata di funzione $g(e_1, \ldots, e_k)$, dove $e_1, \ldots, e_k$ sono delle espressioni e g è il nome di una funzione k-aria definita nel programma, è un caso particolare di espressione.

Allo scopo di indicare il corretto ordine di applicazione degli operatori in una espressione forzeremo che ogni espressione ottenuta per applicazione di un operatore binario sia racchiusa da una coppia di parentesi tonde e che l'espressione a cui si applica l'operatore unario **!** sia anch'essa racchiusa tra parentesi.

Riassumendo, per le espressioni e si ha la seguente sintassi astratta:

$$n \mid x \mid (e_1 \circ e_2) \mid \mathbf{!}(e_1) \mid g(e_1, \ldots, e_k)$$

dove $\circ$ denota un generico operatore binario, n denota un numerale intero con o senza segno, x un nome di variabile, g un nome di funzione.

I comandi

Il linguaggio CCINO contiene il seguente sottoinsieme di costrutti del linguaggio C:

- una forma ristretta di *assegnazione*

  $$x = e$$

  nella quale a destra del simbolo di assegnazione $=$ può apparire una generica espressione e, mentre a sinistra deve necessariamente apparire una variabile x;

- l'istruzione condizionale

  $$\textbf{if } (e)\ c_1\ [\textbf{else } c_2]$$

  nella quale le parentesi quadre intorno alla parte **else** indicano che questa parte può essere omessa;

- il ciclo

  $$\textbf{while } (e)\ c$$

- le istruzioni per la stampa e lettura di valori interi e per la stampa di una sequenza di spazi bianchi o di caratteri newline

  $$\textbf{read}(x) \mid \textbf{write}(e) \mid \textbf{wsp}(e) \mid \textbf{wnl}(e)$$

  dove x è una variabile;

- il comando

  $$\textbf{return } e$$

  che, quando eseguita, fa terminare l'esecuzione della funzione in cui appare, facendo ritornando il valore della espressione e.

Un *comando semplice* di CCINO è una delle precedenti istruzioni seguita dal carattere **;**. Un *blocco* è un comando composto da una sequenza di comandi racchiusa tra parentesi graffe

$$\{c_1\ \ldots\ c_k\}$$

Infine, il *comando vuoto*, è una stinga vuota o formata solo da spazi seguita dal carattere **;**.

Dichiarazioni di variabili

Una dichiarazione di variabili è una sequenza di nomi di variabili preceduta dalla parola chiave **int** e seguita dal carattere **;**. I nomi delle variabili in una dichiarazione sono separati dal carattere **,**.

La sintassi astratta di una dichiarazione di variabili è

$$\textbf{int } x_0\textbf{, } \ldots \textbf{ , } x_k\textbf{;}$$

Dichiarazioni e prototipi di funzioni

Le dichiarazioni e i prototipi delle funzioni di CCINO seguono le normali regole del linguaggio C, con alcuni vincoli:

- ogni funzione deve sempre ritornare un intero;
- nel prototipo di una funzione non si possono omettere i nomi delle variabili, inoltre, i nomi devono coincidere con quelli che si useranno nella successiva dichiarazione della funzione;
- nella dichiarazione di una funzione, le eventuali dichiarazioni di variabili locali devono apparire prima dei comandi del corpo della funzione.

In termini di sintassi astratta si ha quindi

$$\textbf{int } g\textbf{(int } x_1\textbf{, } \ldots \textbf{ , int } x_k\textbf{);}$$

per i prototipi e

$$\textbf{int } g\textbf{(int } x_1\textbf{, } \ldots\textbf{, int } x_k\textbf{) \{}d_1 \ \ldots \ d_m \ c_1 \ \ldots \ c_n\textbf{\}}$$

per le dichiarazioni di funzioni, dove g è un identificatore per il nome della funzione e $x_1, \ldots, x_k$ sono gli identificatori per i nomi dei parametri della funzione g; in particolare, se $k = 0$, la funzione è senza parametri (per le funzioni prive di parametri non è necessario specificare la parola chiave **void** normalmente rischiesta nei programmi C). Se $m = 0$, la funzione non ha variabili locali; se $n = 0$ la funzione è banale e non fa nulla.

In forma compatta, la sintassi per i prototipi e le dichiarazioni di funzioni può essere scritta

$$\textbf{int } g\textbf{(int } x_1\textbf{, } \ldots \textbf{ , int } x_k\textbf{) (;} \mid \textbf{\{}d_1 \ \ldots \ d_m \ c_1 \ \ldots \ c_n\textbf{\})}$$

dove le parentesi tonde non in grassetto servono a delimitare il campo di azione dell'operatore | che racchiudono; in pratica, la precedente espressione indica che la parte di intestazione della funzione può essere seguita dal carattere **;** (nel qual caso si ha un prototipo) oppure dal corpo della funzione.

I programmi

Un programma CCINO si compone di una sequenza di dichiarazioni di variabili globali seguite da una sequenza di prototipi e definizioni di funzioni. Per semplificare l'analisi di un programma CCINO le due parti devono essere separate da un commento vuoto. Quindi, la sintassi astratta di un programma è

$$d_1 \ \ldots \ d_m \ \texttt{/* */} \ f_1 \ \ldots \ f_n$$

Inoltre, la lista delle definizioni di funzioni deve contenere una funzione

$$\textbf{int } \mathsf{main}\textbf{()}$$

La sintassi di CCINO

La sintassi completa di CCINO è riportata in Figura 1. Ovviamente, in aggiunta ai vincoli precedentemente detti, valgono i soliti vincoli del C. In particolare, nella lista dei parametri di una funzione e nella lista delle variabili locali non ci possono essere due variabili con lo stesso nome.

$$
\begin{array}{lcl}
e & ::= & n \\
 & & \mid x \\
 & & \mid (e_1 \circ e_2) \\
 & & \mid \mathbf{!}(e) \\
 & & \mid g(e_1, \ldots, e_k) \\
c & ::= & x = e\mathbf{;} \\
 & & \mid \mathbf{if}\ (e)\ c_1\ [\mathbf{else}\ c_2]\mathbf{;} \\
 & & \mid \mathbf{while}\ (e)\ c\mathbf{;} \\
 & & \mid \mathbf{read}(x)\mathbf{;} \\
 & & \mid \mathbf{write}(e)\mathbf{;} \\
 & & \mid \mathbf{wsp}(e)\mathbf{;} \\
 & & \mid \mathbf{wnl}(e)\mathbf{;} \\
 & & \mid \mathbf{return}\ e \\
 & & \mid \mathbf{\{}c\mathbf{\}} \\
 & & \mid \mathbf{;} \\
d & ::= & \mathbf{int}\ x_0, \ \ldots\ , x_k\mathbf{;} \\
f & ::= & \mathbf{int}\ g(\mathbf{int}\ x_1, \ \ldots \mathbf{int}\ x_k)\ (\mathbf{;} \mid \mathbf{\{}d_1\ \ldots\ d_m\ c_1\ \ldots\ c_n\mathbf{\}}) \\
p & ::= & d_1\ \ldots\ d_m\ \texttt{/* */}\ f_1\ \ldots\ f_n
\end{array}
$$

Figura 1: La sintassi di CCINO

L'esecuzione di un programma CCINO

L'esecuzione di un programma CCINO consiste nell'esecuzione del programma secondo le normali regole di esecuzione dei programmi C.

Struttura del progetto

Il progetto è suddiviso in moduli. Per garantire la massima indipendenza nella realizzazione dei singoli moduli, verranno chiaramente specificate le strutture dati e le funzioni di interfaccia. Rispettare le specifiche di tali interfacce è fondamentale per garantire la costruzione dell'interprete a partire dai moduli. *Non rispettare le specifiche sarà valutato come un errore.*

I moduli in cui è suddiviso il progetto sono:

1. Modulo I: Le tavole dei simboli (TS)
2. Modulo II: L'analizzatore lessicale (AL)
3. Modulo III: L'analizzatore sintattico (AS)
4. Modulo IV: L'esecutore (EX)

Per favorire la chiarezza delle specifiche verranno forniti due file `ccinodefs.h` (vedi Appendice A) e `ccinodefs.c` (vedi Appendice B). Il primo contiene le principali dichiarazioni di tipo e di costanti da includere in tutti i moduli che verranno sviluppati, il secondo contiene la definizione di variabili globali ed eventuali procedure e dovrà essere compilato e collegato con i moduli ed un opportuno main.

Verifica e valutazione dei moduli del progetto

Per ciascun modulo verrà fornito un main file (si veda l'appendice) e le istruzioni su come, a partire dal modulo sviluppato, ottenere un programma funzionante che verifichi il modulo. In molti casi, la verifica di un modulo dipende da alcuni dei moduli precedenti; a tale scopo, al posto dei moduli mancanti o non funzionanti si utilizzeranno dei moduli sviluppati dal docente.

Si fa presente che tutti i moduli dovranno essere compilabili con un compilatore ANSI C e dovranno utilizzare solo le librerie standard ANSI, oltre a quanto specificato nei file `ccinodefs.h` e contenuto in `ccinodefs.c` e nei main file per la verifica `ccino-x.c`, dove `x` è il numero di modulo che si sta verificando.

Il codice sorgente di ciascuno dei moduli sarà contenuto in un unico file. Nelle istruzioni di compilazione riportate in appendice, per il file sorgente di ciascun modulo si userà il nome riportato nella seguente tabella in corrispondenza del numero (sigla) del modulo.

I (TS)	II (AL)	III (AS)	IV (EX)	totale
6	8	10	10	34
`tavole.c`	`lexan.c`	`syntan.c`	`execute.c`	

Per la composizione finale dei moduli verrà fornito un main file `ccino.c` che compilato e collegato con tutti i moduli del progetto darà l'interprete di CCINO.

La precedente tabella riporta anche il peso in 30esimi dei singoli moduli. Per superare l'esame si dovranno **consegnare almeno tre moduli**. Un voto complessivo di 30 o 31 corrisponde a 30/30esimi; un voto superiore a 31 porta alla lode.

I pesi riportati in tabella sono il voto massimo per quel modulo. Requisiti essenziali affinchè un modulo sia preso in considerazione per la valutazione sono:

- il sorgente del modulo deve essere **compilabile** con un compilatore ANSI C (come riferimento useremo il compilatore gcc disponibile su tutte le architetture unix/linux);
- collegando il modulo con il main corrispondente e con gli altri moduli precedentemente sviluppati (o con quelli forniti dal docente), in base alla istruzioni riportate in appendice, **non si devono ottenere errori**;
- come risultato della compilazione/collegamento si deve ottenere un programma eseguibile che **gira correttamente su alcuni dei test proposti in appendice**.

Altri fattori che influiranno sulla valutazione del modulo sono:

- la documentazione del codice (i moduli devono essere esaurientemente commentati);
- lo stile di scrittura (il codice deve essere opportunamente indentato).

Si ricorda che il progetto di laboratorio è un **lavoro individuale**. Per verificare la similitudine dei moduli inviati si userà un apposito tool automatico: **moduli chiaramente copiati saranno annullati**.

A Il file `ccinodefs.h`

```
/* -*- Mode: C -*- */
/* Time-stamp:  <ccinodefs-II.h 04/05/28 00:48:39 stefano@sgport> */

/************************************************************
 * Progetto di Laboratorio di Programmazione
 * Stefano Guerrini - AA 2003-04
 *
 * Header con le principali definizioni.
 ************************************************************/

/************************************************************
 * Modulo I: Le tavole dei simboli (TS)
 */

/*
 * Tavole implementate con un vettore
 */
```

```
#define MAX_KEYS    32
#define MAX_ID_LEN  32
#define MAX_LN_LEN 256

/* Struttura per la rappresentazione di una tavola mediante un vettore */
struct tavola {
    unsigned n; /* elementi nella tavola */
    char *keys[MAX_KEYS]; /* vettore con le stringhe nella tavola */
};

void init_tavola(struct tavola *ptv);
/* inizializza la tavola *ptv */

unsigned avail_tavola(struct tavola *ptv);
/* numero di spazi ancora disponibili nella tavola *ptv */

int ins_key_tavola(struct tavola *ptv, char *key);
/* aggiunge la chiave key alla tavola se non gia' presente
 * ritorna l'indice della posizione della chiave inserita
 * o -1 nel caso in cui key e' gia' presente nella tavola o
 * non c'e' posto per inserirla
 */

int search_key_tavola(struct tavola *ptv, char *key);
/* cerca la chiave key nella tavola *ptv
 * ritorna la posizione della chiave nella tavola
 * o -1 se la chiave non \`e presente
 */

/*
 * Tavole implementate con liste
 */

/* Descrittore di funzione */
struct descr_fun {
    char id[MAX_ID_LEN]; /* il nome della funzione */
    struct tavola tv_params; /* tavola con i nomi dei parametri */
    struct tavola tv_vars; /* tavola con i nomi delle var locali */
    struct ls_comm *block; /* il blocco delle istruzioni della funzione */
};

/* Nodo lista descrittori di funzioni */
struct nodo_lista_dfun {
    struct descr_fun *pdf;
    struct nodo_lista_dfun *next;
};

/* Lista descrittori di funzioni */
typedef struct nodo_lista_dfun *lista_dfun;

/* 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 */
};
```

```
/* Nodo lista descrittori di variabile */
struct nodo_lista_dvar {
    struct descr_var *pdv;
    struct nodo_lista_dvar *next;
};

/* Lista descrittori di variabile */
typedef struct nodo_lista_dvar *lista_dvar;

struct descr_fun *ins_key_dfun(lista_dfun *pls, char *id);
/* aggiunge un nuovo descrittore per id alla lista *pls
 * ritorna il puntatore al nuovo descrittore inserito
 * o NULL in caso di errore o nel caso in cui id e'
 * gia' presente in *pls
 */

struct descr_fun *search_key_dfun(lista_dfun ls, char *id);
/* cerca il descrittore di funzione di id in ls
 * ritorna il puntatore al descrittore trovato
 * o NULL se non presente
 */

struct descr_var *ins_key_dvar(lista_dvar *pls, char *id);
/* aggiunge un nuovo descrittore per id alla lista *pls
 * ritorna il puntatore al nuovo descrittore inserito
 * o NULL in caso di errore o nel caso in cui id e'
 * gia' presente in *pls
 */

struct descr_var *search_key_dvar(lista_dvar ls, char *id);
/* cerca il descrittore di variabile di id in ls
 * ritorna il puntatore al descrittore trovato
 * o NULL se non presente
 */


/*************************************************************
 * Modulo II: L'analizzatore lessicale (AL)
 */

/* variabile globale inizializzata con le parole chiave di PINO */
extern struct tavola tav_keywords;

/* I tag dei token */
#define CC_ERR     0x0000

#define CC_ID      0x0011
#define CC_NUM     0x0012

#define CC_INT     0x0021
#define CC_IF      0x0022
#define CC_ELSE    0x0023
#define CC_WHILE   0x0024
#define CC_READ    0x0025
#define CC_WRITE   0x0026
#define CC_WSP     0x0027
#define CC_WNL     0x0028
```

```
#define CC_RET     0x0029

#define CC_COMMA   0x0041
#define CC_SEMI    0x0042
#define CC_LBR     0x0043
#define CC_RBR     0x0044
#define CC_LCURLY  0x0045
#define CC_RCURLY  0x0046
#define CC_PLUS    0x0181
#define CC_MINUS   0x0182
#define CC_MULT    0x0183
#define CC_DIV     0x0184
#define CC_MOD     0x0185
#define CC_AND     0x0281
#define CC_OR      0x0282
#define CC_NOT     0x0283
#define CC_EQ      0x0481
#define CC_NEQ     0x0482
#define CC_LT      0x0483
#define CC_GT      0x0484
#define CC_LEQ     0x0485
#define CC_GEQ     0x0486
#define CC_ASS     0x0801
#define CC_LCOMM   0x1001
#define CC_RCOMM   0x1002

/* 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 */
};

/* 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 */
};

void next_token(struct descr_token *pdtk);
/* Ritorna in *dtk il prossimo token nello stream di input */


/*********************************************************
 * Modulo III: L'analizzatore sintattico (AS)
 */
```

```
/* ...omissis... */


/*************************************************************
 * Modulo IV: Le funzioni di stampa (PR)
 */

/* ...omissis... */


/*************************************************************
 * Modulo V: Lo stack (ST)
 */

/* ...omissis... */


/*************************************************************
 * Modulo VI: La valutazione delle espressioni (EV)
 */

/* ...omissis... */


/*************************************************************
 * Modulo VII: Il garbage collector (GC)
 */

/* ...omissis... */
```

B Il file `ccinodefs.c`

```
/* -*- Mode: C -*- */
/* Time-stamp:  <ccinodefs-II.c 04/05/28 02:09:08 stefano@sgport> */

#include <stdlib.h>
#include "pinodefs.h"

/*************************************************************
 * Progetto di Laboratorio di Programmazione
 * Stefano Guerrini - AA 2002-03
 *
 * Alcune variabili globali e funzioni di utitilita'
 *************************************************************/


/*************************************************************
 * Modulo I: Le tavole dei simboli (TS)
 */


/*************************************************************
 * Modulo II: L'analizzatore lessicale (AL)
 */

struct tavola tav_keywords = { // tavola delle parole chiave
```

```
    9, {"int", "if", "else", "while", "read", "write", "wsp", "wnl", "return"}
};


/*************************************************************
 * Modulo III: L'analizzatore sintattico (AS)
 */

/* ...omissis... */


/*************************************************************
 * Modulo IV: L'esecutore (EX)
 */

/* ...omissis... */
```