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

Modulo III: L'analizzatore sintattico (AS)

Stefano Guerrini

A.A. 2003/04 – Canale P-Z
Versione del 12 agosto 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, la stringa "-32 **int** $+$) 12", essendo composta da token validi, è lessicamente corretta, ma non può apparire in nessun programma CCINO. Più precisamente, la precedente sequenza di token non rispetta la sintassi di CCINO e pertanto non è sintatticamente corretta. Questo tipo di errori devono essere individuati durante la cosiddetta *analisi sintattica* o *parsing*[1].

1.1 La sintassi

Le regole della sintassi di CCINO sono già state descritte nell'introduzione al progetto. Per comodità le riportiamo nuovamente in Figura 1.

Il modulo che implementa l'analisi sintattica del progetto è l'analizzatore sintattico o *parser* del linguaggio CCINO. 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.

1.1.1 Costanti e variabili

Per memorizzare le costanti di CCINO, visto che abbiamo solo costanti intere, è sufficiente utilizzare una variabile di tipo intero.

Per le variabili si deve invece distinguere se la variabile è globale o locale. In entrambi i casi, ciò che individua univocamente la variabile è la sua posizione nella corrispondente sequenza di definizioni. Per le variabili globali, ciò equivale all'indice della variabile nella corrispondente tavola (vedi 1.2). Per le variabili locali, supponendo di indicizzare le variabili in base all'ordine in cui sono definite, cominciando con i parametri e a seguire con le variabili definite nel corpo della funzione; anche in questo caso, l'indice della variabile può essere facilmente determinato partendo dalla sua posizione nella corrispondente tavola (per i dettagli si veda 1.1.5).

Quindi, per la rappresentazione delle variabili è sufficiente memorizzare un indice intero ed un tag per distinguere le variabili globali da quelle locali. Per questo motivo, nel file `ccinodefs.h` è definita la seguente struttura realizzata con campi di bit nella quale un valore del bit `isglobal` pari a 1 denota una variabile globale, mentre un valore pari a 0 denota una variabile locale.

[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.)

$$
\begin{array}{lcll}
e & ::= & n & (1)\\
 & & \mid x & \\
 & & \mid (e_1 \circ e_2) & \\
 & & \mid \mathbf{!}(e) & \\
 & & \mid g(e_1, \ldots, e_k) & \\
c & ::= & x = e\mathbf{;} & (2)\\
 & & \mid \mathbf{if}\ (e)\ c_1\ [\mathbf{else}\ c_2] & \\
 & & \mid \mathbf{while}\ (e)\ c & \\
 & & \mid \mathbf{read}(x)\mathbf{;} & \\
 & & \mid \mathbf{write}(e)\mathbf{;} & \\
 & & \mid \mathbf{wsp}(e)\mathbf{;} & \\
 & & \mid \mathbf{wnl}(e)\mathbf{;} & \\
 & & \mid \mathbf{return}\ e\mathbf{;} & \\
 & & \mid \mathbf{\{}c_1\ \ldots\ c_n\mathbf{\}} & \\
 & & \mid \mathbf{;} & \\
d & ::= & \mathbf{int}\ x_0\mathbf{,}\ \ldots\ \mathbf{,}\ x_k\mathbf{;} & (3)\\
f & ::= & \mathbf{int}\ g(\mathbf{int}\ x_1\mathbf{,}\ \ldots\mathbf{,}\ \mathbf{int}\ x_k)\ (\mathbf{;} \mid \mathbf{\{}d_1\ \ldots\ d_m\ c_1\ \ldots\ c_n\mathbf{\}}) & (4)\\
p & ::= & d_1\ \ldots\ d_m\ \texttt{/* */}\ f_1\ \ldots\ f_n & (5)
\end{array}
$$

Figura 1: La sintassi di CCINO

```
/* Tipo per la memorizzazione di una variabile
 * Ogni variabile e' rapresentata tramite un intero che indica la sua
 * posizione nella sequenza delle definizioni di variabili locali o globali
 * per il quale riserviamo un campo di 7 bit (pari a 128 variabili)
 * Il bit isglobal indica se si tratta di variabile globale (se pari a 1)
 * o locale (se pari a 0)
 */
struct variabile {
    unsigned isglobal : 1;
    unsigned ofs : 7;
};
```

1.1.2 Le espressioni

Per quanto riguarda le espressioni, in base alla regola (1) in Figura 1, possiamo individuare i seguenti casi:

1. l'espressione è un intero n;
2. l'espressione è una variabile x. Nel qual caso occorre distinguere se
   (a) x è locale alla funzione che si sta analizzando, in particolare, x è un parametro della funzione oppure è stata definita con una dichiarazione locale;
   (b) x è stata definita con una dichiarazione globale.
3. l'espressione è ottenuta componendo le espressioni e_1 ed e_2 mediante un operatore binario;
4. l'espressione è ottenuta applicando l'operatore unario **!** all'espressione e;
5. l'espressione è ottenuta applicando una funzione g definita dall'utente a k espressioni $e_1, \ldots, e_k$.

In base alle precedenti considerazioni, nel file `ccinodefs.h`, sono definite le seguenti strutture dati:

```
/* Tipi di espressione */
enum exp_tags { EXP_NUM, /* numero */
```

```
                EXP_VAR, /* variabile */
                /* operatori binari */
                EXP_PLUS, EXP_MINUS, EXP_MULT, EXP_DIV, EXP_MOD, /* aritmetici */
                EXP_AND, EXP_OR, /* booleani */
                EXP_EQ, EXP_NEQ, EXP_LT, EXP_GT, EXP_LEQ, EXP_GEQ, /* confronto */
                /* operatori unari */
                EXP_NOT, /* negazione */
                EXP_FUNCALL /* chiamata di funzione */
              };

/* Nodo dell'albero per la memorizzazione di espressioni */
struct exp {
    enum exp_tags tag; /* Il tag che individua il tipo di espressione */
    union {
        int n; /* costante intera */
        struct variabile var; /* variabile */
        struct { /* argomenti di un operatore binario */
            struct exp *left, *right;
        } bexp;
        struct exp *arg; /* argomento di un operatore unario */
        struct { /* chiamata di una funzione definita nel programma */
            struct descr_fun *fun; /* descrittore di funzione */
            struct exp *args; /* vettore con gli argomenti della funzione */
        } funcall;
    } u;
};
```

La struttura per la memorizzazione delle espressioni è una strutura che contiene puntatori ad altre strutture e che finisce per formare un albero eterogeneo con nodi di diverse forme. Ogni nodo dell'albero di una espressione ha un tag che contiene un valore del tipo `enum exp_tags` che indica il tipo di espressione che si vuole memorizzare e quindi quale componente della successiva union è significativa per quel nodo.

La union `u` di `struct expr` viene utilizzata nel seguente modo:

1. se l'espressione è un numero, il campo `n` sulla union contiene il valore del numero;
2. se l'espressione è una variabile, il campo `var` della union contiene le informazioni della variabile;
3. se l'espressione è ottenuta mediante un operatore binario i campi `left` e `right` della struttura `bexp` contengono i puntatori alle espressioni alla sinistra e alla destra dell'operatore binario (l'operatore binario da applicare è deducibile dal campo `tag` del nodo);
4. se l'espressione è ottenuta per applicazione di un operatore unario (in CCINO l'unico operatore unario è **!**), il campo `arg` contiene l'argomento dell'operatore;
5. se l'espressione è la chiamata di una funzione di k argomenti, allora il campo `fun` della struttura `funcall` contiene il puntatore alla funzione, mentre il campo `args` contiene un vettore di k elementi, nel quale l'i-esimo elemento è il puntatore all'espressione corrispondete all'i-esimo argomento della chiamata di funzione.

1.1.3 I comandi

Per quanto riguarda i comandi, in base alla regola (2) in Figura 1, possiamo individuare i seguenti casi:

1. il comando è un'assegnazione (`COMM_ASS`);
2. il comando è un if o un if-else (`COMM_ITE`);
3. il comando è un ciclo while (`COMM_WHILE`);

4. il comando è una read (`COMM_READ`);
5. il comando è una scrittura di un valore su output (`COMM_WRITE`);
6. il comando è una scrittura di una sequenza di spazi (`COMM_WSP`) o newline (`COMM_WNL`);
7. il comando è un return (`COMM_RET`);
8. il comando è un blocco, ovvero, una lista di comandi (`COMM_BLOCK`);
9. il comando è vuoto (`COMM_EMP`).

Per distinguere i pecedenti casi, definiamo un tipo enumerato che associa un tag a ciascuno di essi (il tag è riportato tra parentesi nella precedente lista).

```
/* Tipi di comando */
enum command_tags { COMM_ASS, COMM_ITE, COMM_WHILE, COMM_READ,
                    COMM_WRITE, COMM_WSP, COMM_WNL, COMM_RET,
                    COMM_BLOCK, COMM_EMP
                  };
```

Definiamo quindi una struttura per la memorizzazione di un comando ed una per la memorizzazione di liste di comandi.

```
/* Struttura per la memorizzazione di una lista di comandi
 * Ad esempio, il corpo di una funzione o un blocco.
 */
struct ls_commands {
    struct command *c;
    struct ls_commands *next;
};

/* Nodo dell'albero per la memorizzazione dei comandi */
struct command {
    enum command_tags tag; /* Il tag che individua il tipo di comando */
    union {
        /* assegnazione */
        struct {
            struct variabile lhs; /* variabile alla sin. dell'assegnazione */
            struct exp *rhs; /* espressione alla destra dell'assegnazione */
        } ass;
        /* if-else */
        struct {
            struct exp *cond; /* il test */
            struct command *then_c, /* comando del caso then */
                *else_c;  /* comando del caso else */
        } ite;
        /* while */
        struct {
            struct exp *cond; /* il test */
            struct command *body; /* corpo del ciclo */
        } loop;
        /* letttura */
        struct variabile read_var; /* var. in cui memorizzare il val. letto */
        /* scrittura */
        struct exp *wr_exp; /* espressione da valutare e stampare
                             * oppure numero di spazi o new-line da
                             * stampare */
        /* return */
```

```
        struct exp *ret_exp; /* espressione di cui ritornare il valore */
        /* blocco */
        struct ls_commands *block; /* blocco di comandi */
    } u;
};
```

La struttura per la memorizzazione dei comandi contiene un tag di tipo `enum command_tag` il cui valore indica il tipo del comando da memorizzare e pertanto quale componente della successiva union è significativa.

La union `u` di `struct command` viene quindi utilizzata nel seguente modo:

1. se il comando è un'assegnazione, la struct `ass` contiene nel campo `lhs` l'indice e il tipo (locale o globale) della variabile alla sinistra dell'assegnazione e nel campo `rhs` il puntatore all'espressione alla destra dell'assegnazione;

2. se il comando è un if o un if-else, la struct `ite` contiene nel campo `cond` l'espressione condizionale dell'if-else, nel campo `then_c` il comando corrispondente al ramo then e nel campo `else_c` il comando corrispondente al ramo else (il caso di comando if senza else viene gestito mantenendo pari a `NULL` il campo `else_c`);

3. se il comando è un ciclo while, la struct `loop` coontiene nel campo `cond` l'espressione condizionale che controlla il ciclo e nel campo `body` il comando corrispondente al corpo del ciclo;

4. se il comando è una lettura da input, il campo `read_var` contiene l'indice e il tipo (locale o globale) della variabile da leggere;

5. se il comando è una scrittura (di un dato, di spazi o di new-line), il campo `wr_exp` contiene l'espressione di cui stampare il valore o corrispondente agli spazi o ai new-line da stampare;

6. se il comando è un return, il campo `ret_exp` contiene l'espressione di cui ritornare il valore;

7. se il comando è un blocco, il campo `block` contiene la lista dei comandi che formano il blocco.

Alcune considerazioni sulle parentesi nei comandi e nelle espressioni

La sintassi di CCINO è molto restrittiva nell'uso delle parentesi. Ad esempio, non si può prendere una generica espressione e racchiuderla tra una coppia di parentesi tonde, ovvero, (x) non è un'espressione corretta. Invece, ogni espressione ottenuta mediante un operatore binario deve essere racchiusa tra una coppia di parentesi tonde. Ad esempio, $x > 0$ e $(x+y+z)$ non sono espressioni corrette; le corrispondenti espressioni corrette sono $(x > 0)$ e $((x + y) + z)$ (oppure $(x + (y + z))$ se si associa a destra anzichè a sinistra).

Questa definizione della sintassi semplifica l'implementazione dell'analizzatore sintattico, ma appesantsce la scrittura dei comandi CCINO. Ad esempio,

$$\textbf{if } (x > 0)\ x = (x - 1);$$

non è un comando corretto. Infatti, in base alla sintassi di CCINO, l'if deve essere seguito da un'espressione valida racchiusa tra parentesi, quindi, rimosse le parentesi richieste dalla sintassi dell'if, $x > 0$ dovrebbe essere un'espressione corretta, ma in base a quanto detto precedentemente, tale espressione sarebbe corretta solo se raccchiusa tra parentesi tonde. Concludendo, la forma coretta del precedente comando è

$$\textbf{if } ((x > 0))\ x = (x - 1);$$

dove $x > 0$ è racchiuso tra una doppia coppia di parentesi tonde: una coppia richiesta dalla sintassi dell'if ed una coppia richiesta dalla sintassi delle espressioni.

Lo stesso ragionamento vale in tutti gli altri casi in cui la sintassi del comando richiede un'espressione racchiusa tra parentesi.

1.1.4 Le dichiarazioni di variabili

Le dichiarazioni di variabili, come in C, sono sequenze di nomi di variabile separati da virgole e preceduti dal nome del tipo della variabile. In CCINO, le cose sono ovviamente molto più semplici che in C dato che l'unico tipo è **int** e che non ci sono puntatori o vettori (ciò significa che i nomi di variabile non saranno preceduti o seguiti da modificatori per costruire puntatori o vettori).

1.1.5 Le dichiarazioni di funzioni

La dichiarazione o il prototipo di una funzione deve sempre precedere il primo utilizzo della funzione.

Al momento della lettura del prototipo o della dichiarazione si deve verificare se il nome è già presente nella tavola dei descrittori delle funzioni (vedi il modulo delle tavole dei simboli) ed eventualmente inserirlo.

Per prima cosa, vediamo il caso di nuovo inserimento. La funzione che gestisce la tavola dei descrittori di funzioni ritorna un puntatore ad un nuovo descrittore di funzione, la cui definizione di tipo riportiamo nuovamente qui di seguito[2], ed il cui campo `id` contiene il nome della funzione.

```
/* 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 */
};
```

La scansione della lista dei parametri della funzione porta alla creazione della tavola dei parametri memorizzata nel campo `struct tavola tv_params`—si osservi che le funzioni implementate nel modulo della tavola dei simboli garantiscono che l'i-esimo parametro della funzione è contenuto nella posizione $i-1$ di `tv_params`. Se si sta analizzando il prototipo della funzione, non c'è altro da fare, ed i rimanenti campi del descrittore vanno messi a `NULL`. Nel caso di dichiarazione, si deve invece passare ad analizzare e ad inserire la lista delle variabili locali nella tavola `struct tavola tv_vars` e la lista dei comandi del corpo della funzione nel campo `block`.

L'indice di una variabile locale da usare durante la fase di scansione delle espressioni è:

- per le variabili globali, direttamente la posizione nella corrispondente tavola;
- per le variabili locali e per i parametri, se si tratta di
  - un parametro della funzione, la posizione nella corrispondente tavola;
  - una variabile locale, la posizione nella corrispondente tavola più il numero di parametri della funzione.

  Questo perchè, supponiamo di assegnare un indice alle variabili numerando prima i parametri e quindi, di seguito, le variabili locali.

Nel caso in cui la funzione che si sta analizzando è già presente nella tavola delle funzioni, per semplicità, supponiamo che l'unico caso possibile sia quello in cui si sta analizzando una dichiarazione di funzione della quale si è precedentemente incontrato il prototipo. In questo caso, si deve controllare che la lista dei parametri del prototipo e quella della dichiarazione sotto analisi sono uguali.

1.1.6 I programmi

Un programma è composto da una sequenza di dichiarazioni di variabili globali seguita da una sequenza di dichiarazioni e/o prototipi di funzioni. Per semplificare l'analisi sintattica, le dichiarazioni di variabili precedono le dichiarazioni delle funzioni e le due parti sono separate dalla linea di commento `/* */`.

In base a quanto detto in 1.1.5, la stessa funzione non può essere definita due volte in un programma; inoltre ogni programma corretto deve contenere una funzione `main` che per semplicità supponiamo non avere parametri (ad un programma CCINO non si possono quindi passare parametri sulla linea di comando).

[2]Per mantenere la compatibilità con alcune implementazioni dei moduli I e II sviluppati con una vecchia versione di `struct descr_fun` che eseguivano l'inizializzazione di alcuni campi non più presenti nella corrente versione di `struct descr_fun` la definizione di questa struttura contenuta in `ccinodefs.h` contiene anche questi campi. Tali campi sono del tutto inutili ai fini dello sviluppo del progetto.

1.2 Analisi sintattica di un programma

Partendo dal programma da analizzare letto da input per mezzo di chiamate alla `next_token` sviluppata nel modulo dell'analizzatore lessicale, l'analizzatore sintattico deve produrre una rappresentazione del programma. In particolare, l'analizzatore deve

1. inserire tutte le variabili globali nell'apposita tavola;
2. creare i descrittori di tutti le funzioni dichiarate nel programma;
3. ritornare l'indirizzo del descrittore della funzione `main` contenuta nel programma.

Il tutto, verificando che il programma ricevuto in input è corretto sintaticamente.

Per quanto riguarda la parte di analisi lessicale, si deve quindi scrivere la seguente funzione:

```
struct descr_fun *parse_program(void);
/* Analizzatore sintattico di un programma CCINO
 * Ritorna il puntatore al descrittore della funzione main()
 */
```

I descrittori delle funzioni e le variabili globali del programma CCINO vanno inseriti nelle variabili globali

```
/* Lista dei descrittori delle funzioni */
lista_dfun ls_dfun;

/* Lista dei descrittori delle variabili globali */
lista_dvar ls_dvar;
```

contenute in `ccinodefs.c` e dichiarate esterne in `ccinodefs.h`.

La maniera più semplice per implementare l'analizzatore sintattico è quella di creare una funzione di parsing per ciascuna delle categorie sintattiche della sintassi di CCINO che, in caso di successo, ritornano il puntatore alla struttura corrispondentemente creata. Seguendo questo approccio, si ha una funzione per l'analisi delle espressioni, una per l'analisi dei comandi, una per l'analisi delle dichiarazioni di variabili, ed una per l'analisi delle dichiarazioni di funzioni, oltre ovviamente a quella principale che analizza un intero programma. All'interno di ciascuno dei suddetti casi si possono definire sottofunzioni per l'analisi dei sottocasi che definiscono il corrispondente costrutto sintattico; ad esempio, per l'analisi di un comando, si possono avere una funzione per l'analisi dell'if, una per il while, e così via. Ciascuna delle funzioni corrispondente ad un costrutto sintattico, dopo aver individuato a quale sottocaso appartiene il costrutto, deve chiamare la corrispondente funzione (per fare ciò basta vedere il primo token del costrutto) e verificare che il comando è correttamente terminato da un **;**. Ad esempio, la funzione per il parsing di un comando, se il primo token del costrutto

- è una variabile, procederà con il caso dell'assegnazione, leggendo il successivo = e richiamando la funzione per il parsing di un'espressione;
- se si tratta di un **if**, chiamerà la funzione che esegue il parsing dell'if e dell'if-else;
- se si tratta di un **while**, chiamerà la funzione che esegue l'analisi del ciclo while;
- e così via negli altri casi.

In alcuni casi—ad esempio, per l'analisi di comandi ed espressioni—è necessario conoscere la lista delle variabili locali delle funzioni. Di conseguenza, tale parametro (insieme ad altre eventuali informazioni necessarie per la verifica della correttezza sintattica) deve essere passato alle corrispondenti funzioni di parsing (come già detto, la lista delle variabili globali e quella dei descrittori delle funzioni sono globali e quindi visibili a tutte le funzioni di parsing).

1.3 Stampa

Per verificare che la funzione di analisi lessicale ha correttamente creato la rappresentazione del programma, si deve implementare anche la funzione che stampa il contenuto della tavola delle variabili globali e della tavola dei descrittori delle funzioni.

```
void stampa_program(void);
/* Stampa la lista delle variabili globali e i descrittori delle
 * funzioni memorizzati nelle variabili globali ls_dvar e ls_dfun
 * creando un programma ccino corretto
 * Per garantire la correttezza sintattica dell'output, delle funzioni
 * contenute in ls_dfun viene prima stampata la lista dei prototipi
 * e poi la lista delle dichiarazioni
 */
```

L'output prodotto deve essere un programma CCINO corretto. Per garantire ciò, per prima cosa si dovrà stampare prima la lista delle variabili quindi, preceduta dal commento vuoto di separazione, i prototipi di tutte le funzioni nella tavola delle funzioni, in ultimo, le dichiarazioni di tutte le funzioni. Si osservi che i dati memorizzati nelle tavole delle variabili e delle funzioni garantiscono che i nomi delle variabili contenuti nel programma di output corrispondono a quelli del programma letto in input.

1.4 Verifica

Per verificare l'analizzatore sintattico 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 `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 ccino-III.c ccinodefs.c tavole.c lexan.c syntan.c -o ccino-III
```

Come risultato della compilazione si ottiene l'eseguibile `ccino-III` che legge da stdin ed analizza per mezzo della `parse_program` un programma CCINO e lo ristampa sullo stdout per mezzo della `stampa_program`. Al programma è possibile passare uno o due argomenti sulla linea di comando: il primo è il nome diun file da cui leggere l'input mediante reindirizzamento di stdin; il secondo il nome di un file su cui stampare l'output reindirizzamento di stdout.

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:  <ccino-III.c 04/07/07 15:27:02 guerrini@guerrini.dsi.uniroma1.it> */

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

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

int main(int argc, char *argv[]) {
    /*
     * Esegue il parsing di un programma CCINO letto da stdin e
     * lo ristampa sullo stdout mediante la funzione stampa_program.
     *
     * Puo' essere chiamato nelle seguenti forme
     *    ccino-III fin
     *    ccino-III fin fout
     * dove fin e' un file che contiene il programma da analizzare
     * e fout e' il file dove salvare l'outpu.
     * Il reindirizzamento su fin e fout e' ottenuto mediante
     * assegnazione a stdin e stdout dei file da usare attraverso
     * la freopen.
     */

    struct descr_fun *fd_main;

    --argc; ++argv;
    if (argc > 0) {
        if (freopen(*argv, "r", stdin) == NULL) {
            fprintf(stderr,
                    "Impossibile aprire il file di input \"%s\"\n",
                    *argv);
            exit(-1);
        } else
            fprintf(stderr, "File di input: %s\n", *(argv));
        --argc; ++argv;
        if (argc > 0) {
            if (freopen(*argv, "w", stdout) == NULL) {
                fprintf(stderr,
                        "Impossibile aprire il file di output \"%s\"\n",
                        *argv);
                exit(-1);
            } else {
                fprintf(stderr,
                        "File di output: %s\n",
                        *(argv));
            }
        }
    }

    fd_main = parse_program();

    stampa_program();
}
```