POLITECNICO DI BARI

FACOLTÀ DI INGEGNERIA
CORSO DI LAUREA SPECIALISTICA IN INGEGNERIA INFORMATICA

Tema d'anno in

"COMPILATORI E INTERPRETI"

C-ANALYZER

ANALIZZATORE LESSICALE E SINTATTICO

Docente:
Chiar.mo Prof. Giacomo PISCITELLI

Studenti:
Leonardo DIBITONTO
Giacomo M. FIORELLA
Stefania LEOCI

ANNO ACCADEMICO 2009/2010

SOMMARIO

INTRODUZIONE

C-Analyzer è un software che consente di effettuare l'analisi lessicale, sintattica e semantica di un codice scritto nel linguaggio di programmazione C, in particolare in un dialetto del C; infatti, come suggerito dal docente durante il corso, date le vaste specifiche grammaticali e sintattiche del C si è deciso di ridurre il lessico e la sintassi del linguaggio preservando comunque il teorema di BOHM-JACOBINI secondo cui: "*ogni algoritmo può essere scritto utilizzando le tre strutture principali: Sequenza, Selezione e Ciclo".*

Il C-Analyzer è dotato di una interfaccia grafica semplice e intuitiva, che consente all'utente di caricare da un file esterno il proprio codice sorgente in un'area di testo e di evidenziare le parti che presentano errori lessicali, sintattici e alcuni errori semantici. I messaggi di diagnostica di tali errori sono mostrati in una text area presente nella finestra principale del programma e consentono di individuare al meglio i tipi di errore che si sono verificati. Un'altra funzionalità molto importante, tipica degli editor di testo grafici comunemente utilizzati, è il riconoscimento dei token e la colorazione degli

stessi in modo differente e in dipendenza dalla loro tipologia, durante il caricamento del codice.
Data l'enorme complessità legata allo sviluppo procedurale e tabulare delle funzionalità di analisi lessicale e sintattica, ci siamo avvalsi dell'uso di un parser generator e di uno scanner generator.
Più precisamente per la realizzazione di C-Analyzer si sono utilizzati i seguenti software:

- Flex 2.5.4a-1: per la creazione dell'analizzatore lessicale
- Bison 2.4.1: per la realizzazione dell'analizzatore sintattico
- NetBeans 6.8: potente ambiente di sviluppo adoperato per realizzare l'intera interfaccia grafica in Java (data la maggior esperienza degli autori con questo linguaggio di programmazione)

La fig. 1 mostra uno screenshot dell'applicazione. Il codice sorgente in linguaggio C da esaminare va immesso nella textarea codice; gli indici di linea sono generati automaticamente e riportati a sinistra e permettono di rintracciare più facilmente gli eventuali errori quando vengono segnalati.
Il menù File consente di espletare le azioni di reset delle aree di testo del codice, di caricamento del contenuto di un file, di inizio dell'analisi e di uscita dal programma (operazioni riportate anche nel pannello principale per una migliore usabilità). L'avvio dell'analisi comporta la segnalazione dei risultati nella textarea output.
Il menù Help, invece, visualizza solo un'informativa sul progetto (fig. 2)

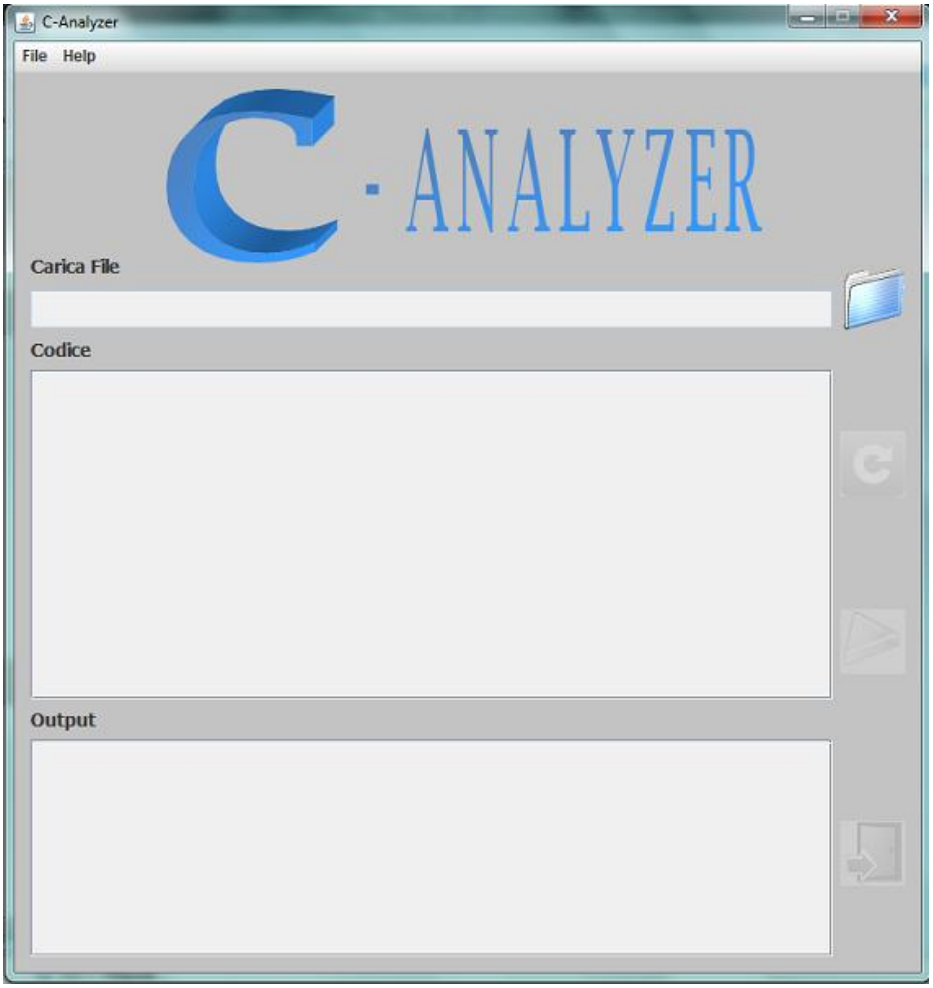

Figura 1

Figura 2

Nel caso in cui le verifiche vadano tutte a buon fine, verrà visualizzato a video che il programma è stato riconosciuto correttamente (vedi fig. 3); in caso di codice errato invece, verranno visualizzati dei messaggi che riporteranno i tipi di errore. Inoltre per agevolare la visibilità ed aumentare l'impatto visivo, le righe interessate verranno colorate di rosso.

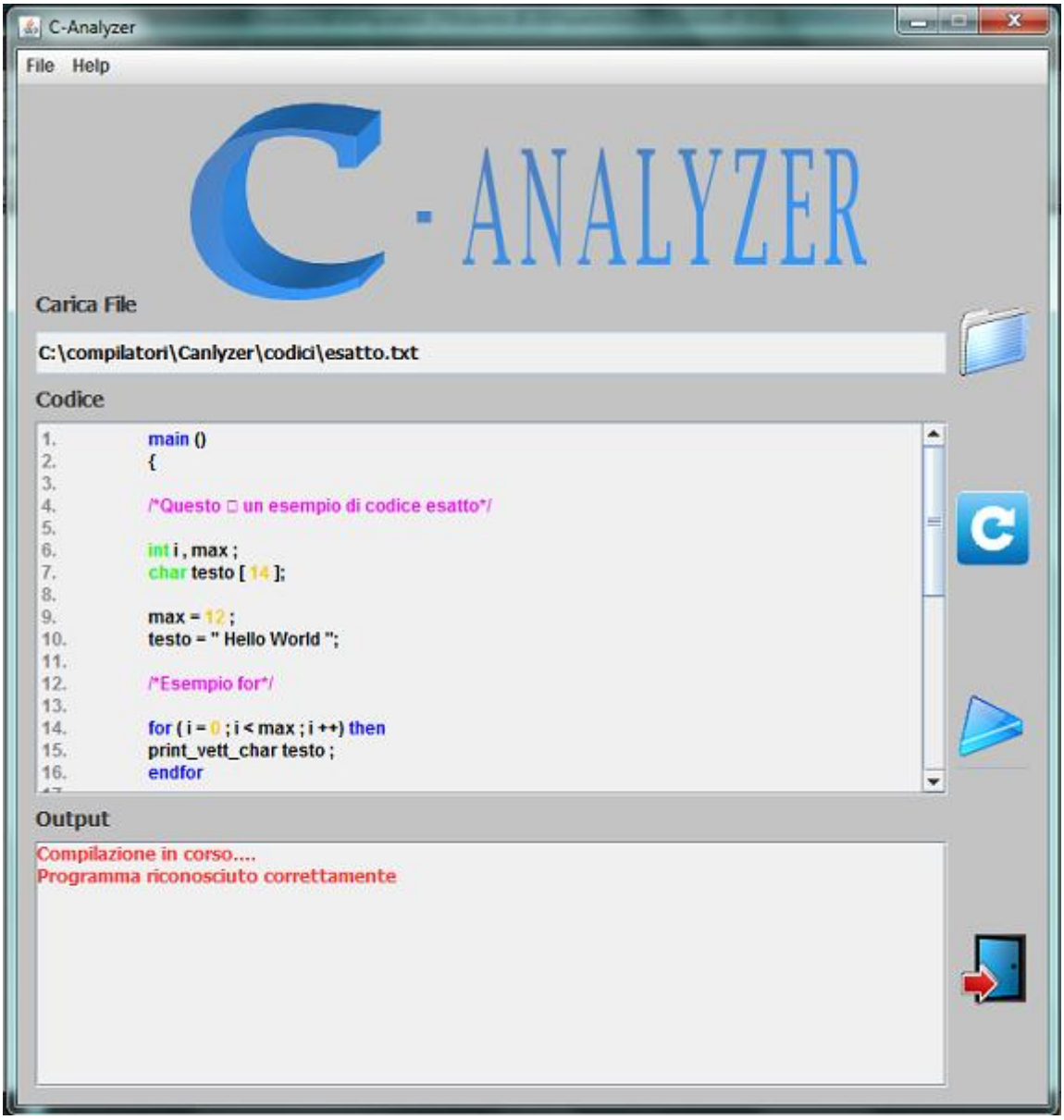

Figura 3

ANALISI LESSICALE

L'analisi lessicale è svolta dallo scanner, implementato dal file sorgente *scanner.l* . In appendice è riportato il contenuto del file.

La diagnostica degli errori lessicali è stata realizzata tenendo conto delle modalità di funzionamento dello scanner generator Flex.

Esso prende come input un file di testo contenente definizioni, regole e codice C di supporto per l'analisi lessicale e fornisce in uscita un file sorgente C denominato generalmente *yy.lex.c*, il quale implementa un automa a stati finiti deterministico (DFA) che consente di riconoscere i token. In particolare il DFA è creato con stati e azioni che dipendono dalle regole indicate nel file input di Flex, le quali sono indicate in questa forma

espressione_regolare *azione*

Ogni volta che l'analizzatore lessicale trova una sequenza di simboli che è implicata dall'espressione regolare indicata in una regola, esercita l'azione corrispondente.

Questo comportamento potrebbe però dar luogo a due tipologie di ambiguità, per risolvere le quali sono state stabilite due convenzioni:

1) quando la parte iniziale di una sequenza di caratteri è riconosciuta da due espressioni regolari è eseguita l'azione associata alla sequenza più lunga;
2) quando la stessa sequenza di simboli è riconosciuta da due espressioni regolari distinte è eseguita l'azione relativa alla regola dichiarata per prima.

Due tipologie di errori lessicali molto comuni sono:

- la denominazione scorretta degli identificatori, che in C possono iniziare unicamente con un underscore o con una lettera ma non con una cifra,
- la mancanza di chiusura delle virgolette nelle stringhe di caratteri.

Per identificare e gestire questi errori sono state inserite le seguenti due regole:

- ([0-9])+([A-Za-z0-9_])* {erroreLessicale(1);}
- L?\"(\\.|[^\\\"\n])* { erroreLessicale(2); }

in cui *erroreLessicale(int)* è una funzione che in base al valore passato come parametro, mostra un messaggio con la descrizione dell'errore lessicale. Si noti come nel caso in cui le virgolette siano invece regolarmente chiuse, non verrà visualizzato alcun messaggio di errore. La fig. 4 mostra un esempio di quanto detto sopra.

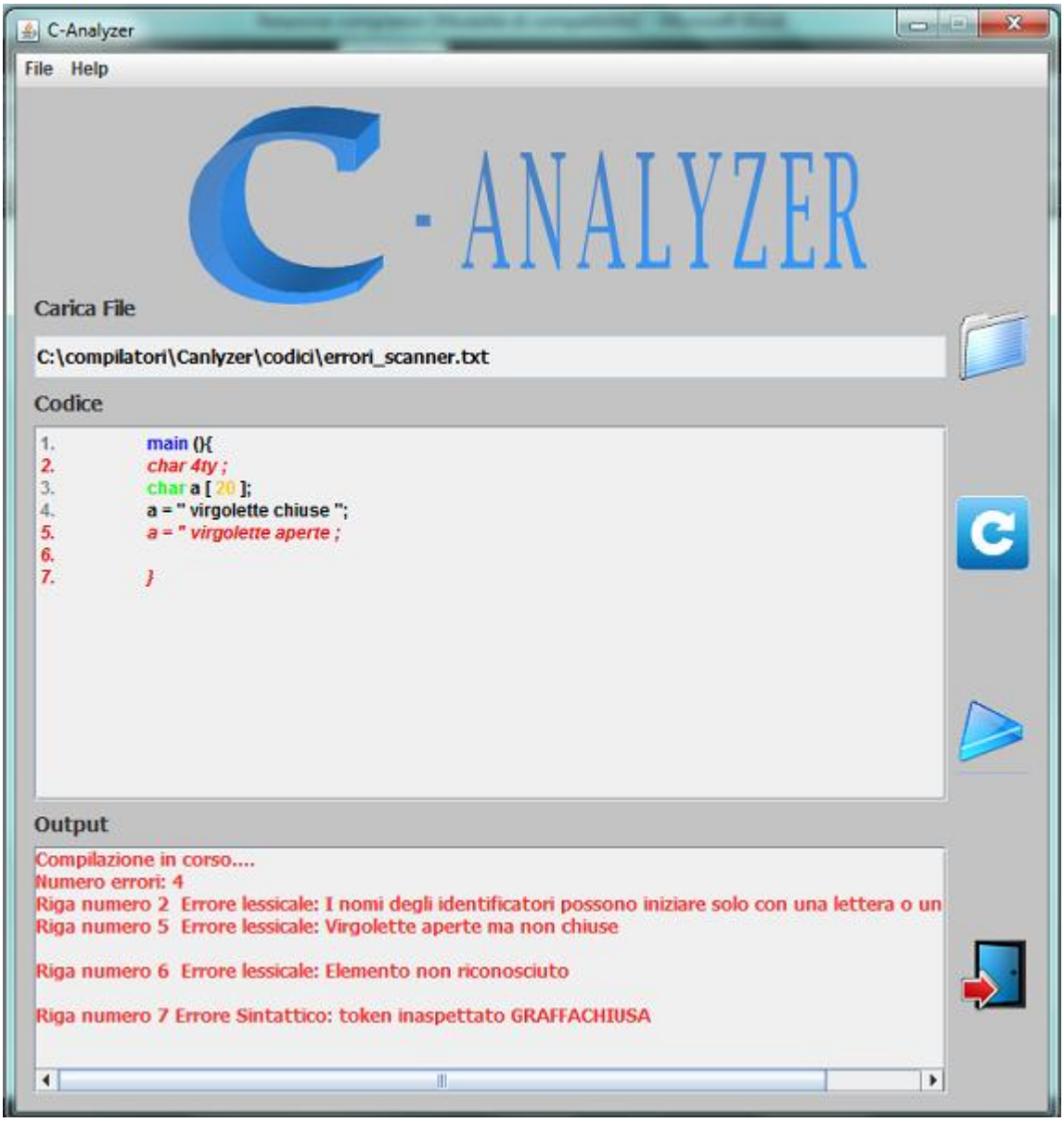

Figura 4

Le righe incriminate sono la 2 e la 5, e come previsto sono evidenziate in rosso dal programma. Ovviamente questo provoca anche l'avviso di altri errori sintattici, secondo un procedimento a cascata, in quanto il parser, alla luce della mancata chiusura delle virgolette, non riesce a spiegarsi il simbolo di parentesi graffa chiusa.
Infine come ultima regola dello scanner è stata inserita la seguente:

- **.** {erroreLessicale(0);}

nella quale il carattere "." è un'espressione regolare che indica un simbolo qualsiasi, eccezion fatta per il newline, perciò tale regola sarà attivata soltanto quando nessuna di tutte le altre regole precedenti risulterà essere verificata e verrà visualizzato a video il messaggio: "Errore lessicale: Elemento non riconosciuto."; come mostrato in fig. 5.

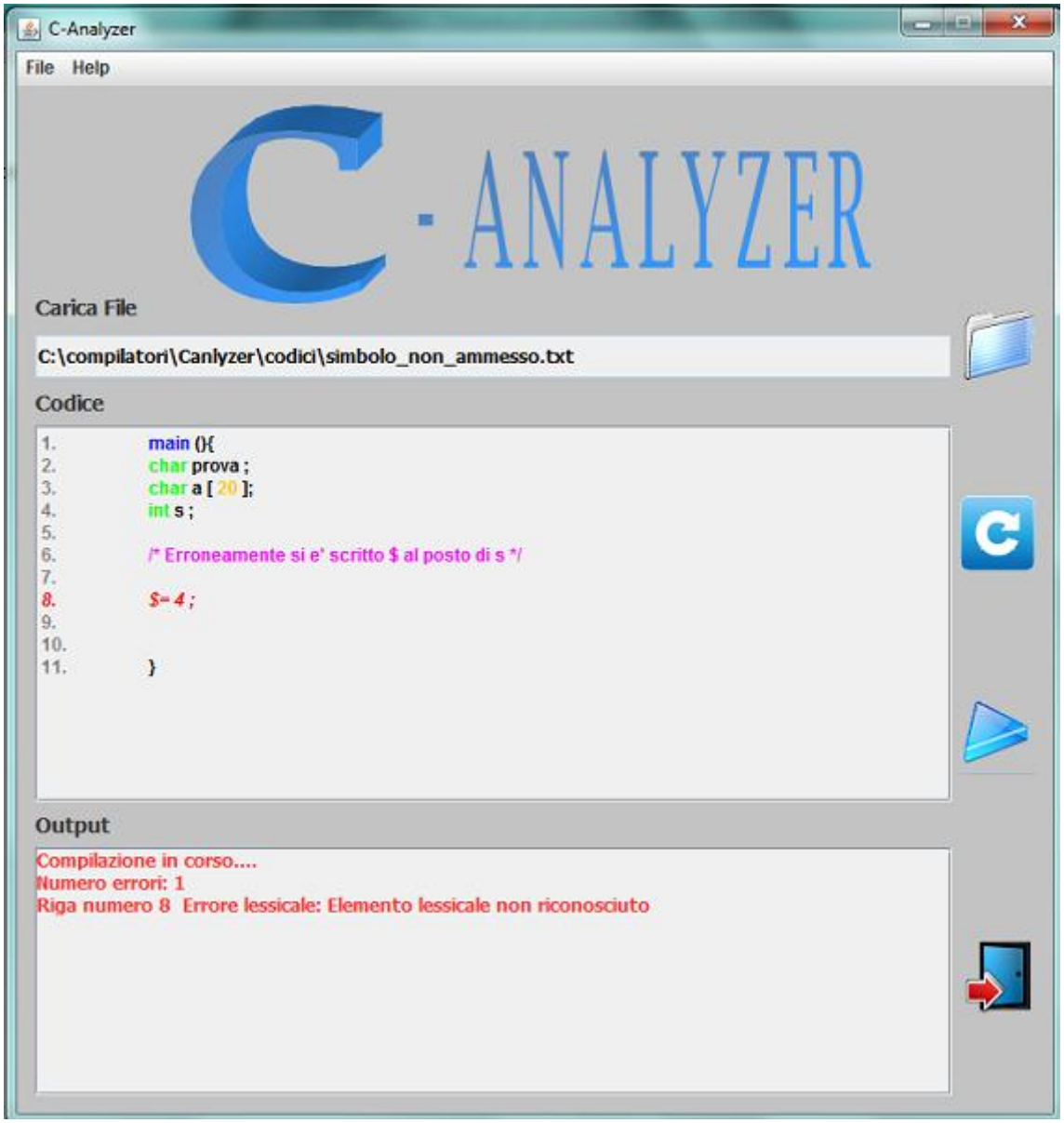
C-Analyzer
File Help
C - ANALYZER
Carica File
C:\compilatori\Canlyzer\codici\simbolo_non_ammesso.txt
Codice
1. main (){
2. char prova ;
3. char a [20];
4. int s ;
5.
6. /* Erroneamente si e' scritto $ al posto di s */
7.
8. $= 4 ;
9.
10.
11. }
Output
Compilazione in corso....
Numero errori: 1
Riga numero 8 Errore lessicale: Elemento lessicale non riconosciuto

Figura 5

ANALISI SINTATTICA

L'analisi sintattica è effettuata dal parser, implementato dal file sorgente parser.y. In appendice è riportato l'intero contenuto del file.
La rilevazione e la gestione degli errori sintattici sono tuttora delle tematiche aperte nell'ambito della ricerca sui compilatori. Al fine di supportare al meglio il lavoro di sviluppo da parte del programmatore, l'analizzatore sintattico dovrebbe consentire la comprensione immediata degli errori compiuti, tuttavia spesse volte ciò risulta essere molto arduo.
C-Analyzer, a fronte di un errore sintattico, visualizza il numero presunto di riga errata e la descrizione testuale dell'errore stesso.
La diagnostica degli errori sintattici è stata realizzata tenendo conto delle modalità di funzionamento del parser generator Bison.
Esso prende come input un file riportante quattro sezioni:

```
%{
Prologo
%}
Dichiarazioni
%%
Regole della grammatica
%%
Epilogo
```

Nel "Prologo" vi sono gli import delle librerie necessarie, nonché la definizione di alcune variabili che serviranno per realizzare al meglio le azioni sintattiche.
La parte "Dichiarazioni" definisce i simboli della grammatica e i valori semantici ad essi associati. I simboli sono suddivisi in due gruppi: i simboli introdotti da *%token* sono simboli terminali e provengono dallo scanner. Quelli introdotti da *%left* servono a definire le precedenze tra i simboli terminali.

La parte contenente le regole della grammatica è il cuore del programma.
Le regole, seguendo una notazione simile alla Backus-Naur Form, sono espresse secondo la formula

A: BODY azione;

ove A è un simbolo non terminale e BODY rappresenta una sequenza di uno o più simboli terminali o non terminali. Se per lo stesso simbolo non terminale A sono previste più produzioni, è possibile intervallarle con il simbolo simbolo "|". Durante l'utilizzo del parser, viene chiamato continuamente lo scanner che fornisce in uscita i token identificati in modo da poter stabilire se la sequenza è una frase ammessa dalla sintassi. In questa sezione vengono gestiti anche i controlli semantici con delle chiamate a funzione, definite nel file *semantica.h*, nel corpo delle azioni. Inoltre, ogni qual volta viene riconosciuta una produzione, si esegue l'azione ad essa associata. Quando una sequenza di token non può essere riscritta sulla base di alcuna regola sintattica-semantica, viene rilevato un errore e il parser chiama la funzione *yyerror()*.
Nell'ultima sezione si trova il corpo della funzione *yyerror()* e del main(). Nel *main()* sono gestiti gli errori legati all'apertura dei file che si vuole analizzare, e viene chiamata la funzione *yyparse()* che a sua volta chiama la *yyerror().*
Bisogna ricordare, che essendo questo un analizzatore di un dialetto del C è necessario rispettare alcune regole per non incorrere in errori sintattici apparentemente inesistenti per chi è abituato a scrivere codice in C. Innanzitutto un programma scritto nel dialetto C deve forzatamente iniziare con la parola chiave ***main(){… corpo del programma…*** e deve terminare con la parentesi graffa chiusa ***}*** (vedi fig. 6). Inoltre non è stata prevista nessuna direttiva del pre-processore. Quindi le ***#include*** e le ***#define*** non sono riconosciute da questo linguaggio; perciò per gestire le operazioni di I/O sono state create delle printf e scanf ad hoc per la sintassi del dialetto C (vedi appendice).

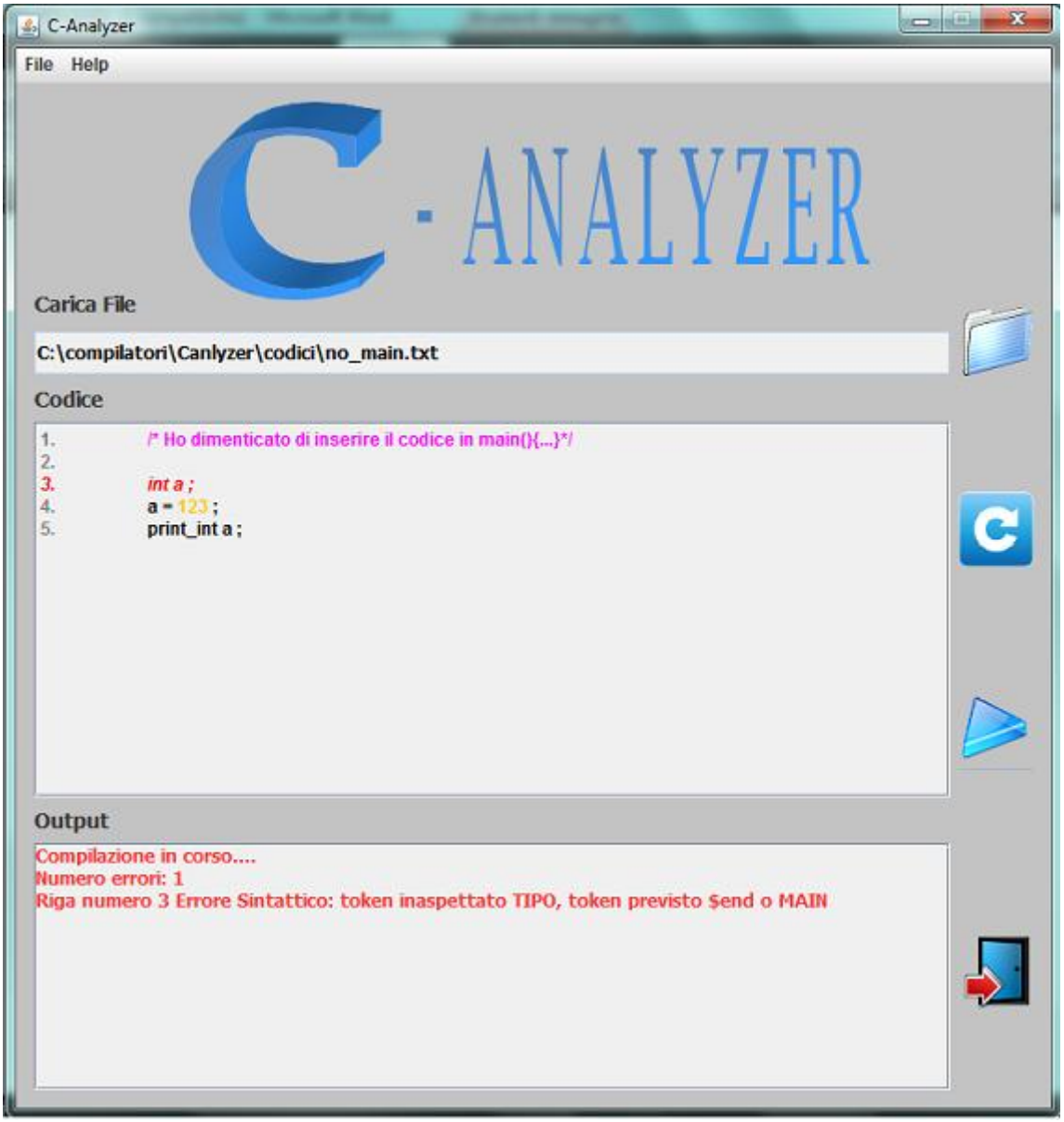

Figura 6

Infine le variabili utilizzate nel main() debbono necessariamente essere dichiarate tutte nella parte iniziale del codice. Nel caso in cui vi sia una dichiarazione di variabile immediatamente dopo un'istruzione che non sia di dichiarazione, verrà segnalato un errore di sintassi come mostrato dall'immagine seguente.

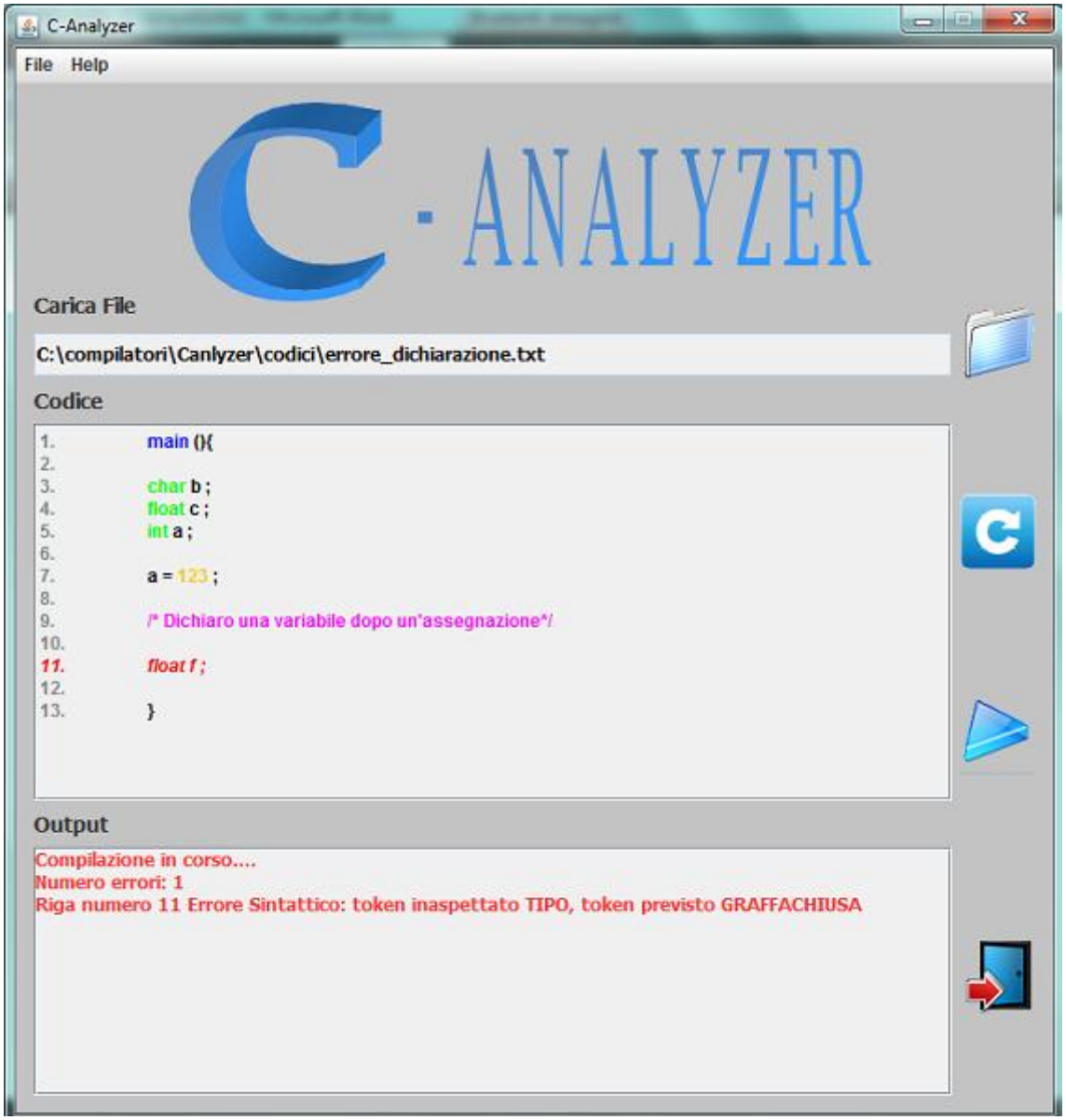

Figura 7

Riportiamo di seguito alcuni esempi di errori sintattici tra i più comuni :

- In questo esempio gli errori commessi sono tre:
    1. Mancanza di un “;”
    2. Omissione di tipo
    3. Omissione di parentesi

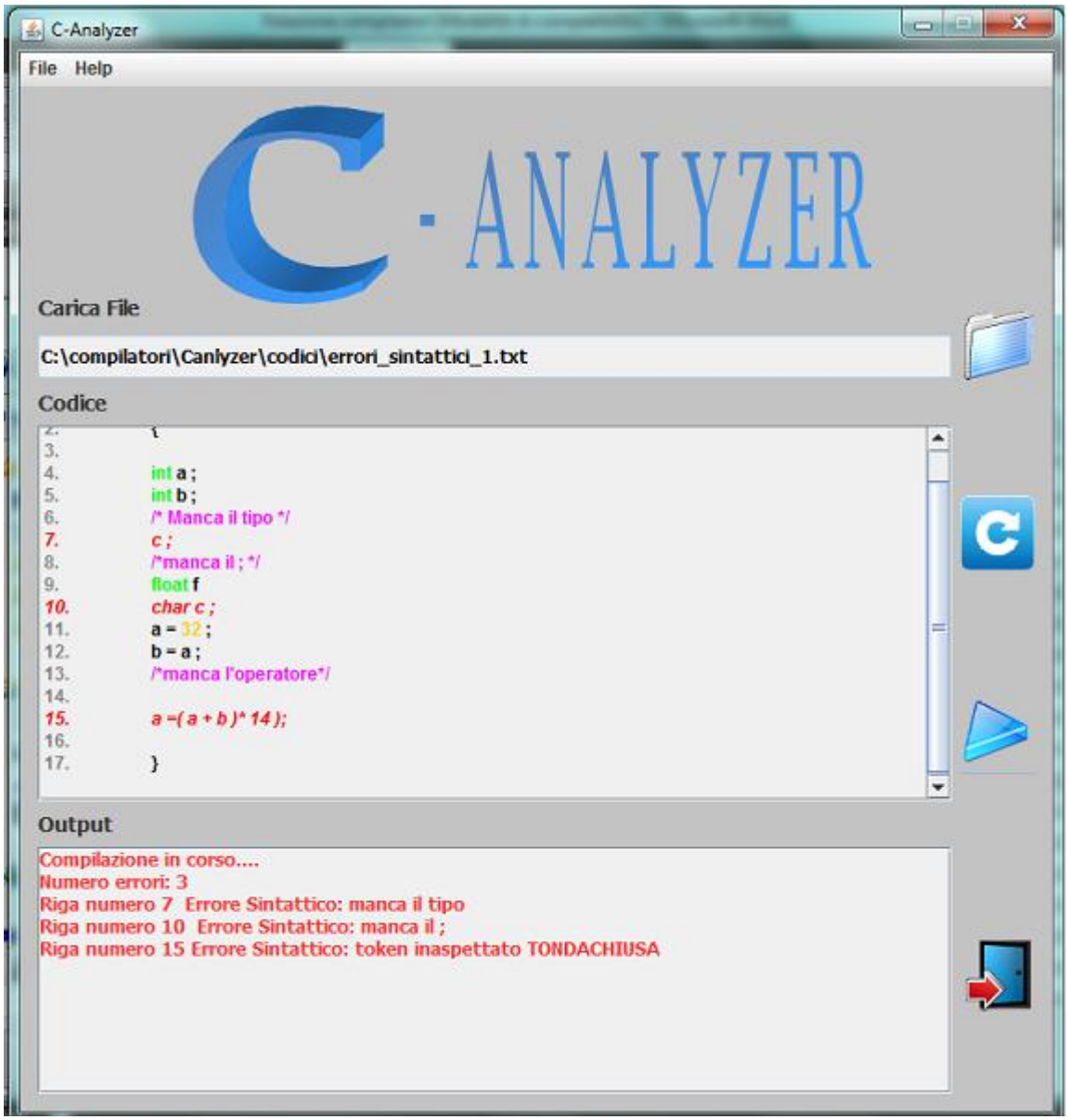

Figura 8

Come si può notare dalla figura a volte è arduo individuare il punto esatto dell’errore sintattico, infatti, anche se il “;” manca alla riga 9 l’errore è segnalato alla riga 10.

- In questo esempio gli errori commessi sono due:
    1. Mancanza dell'identificatore
    2. Mancanza operatore matematico

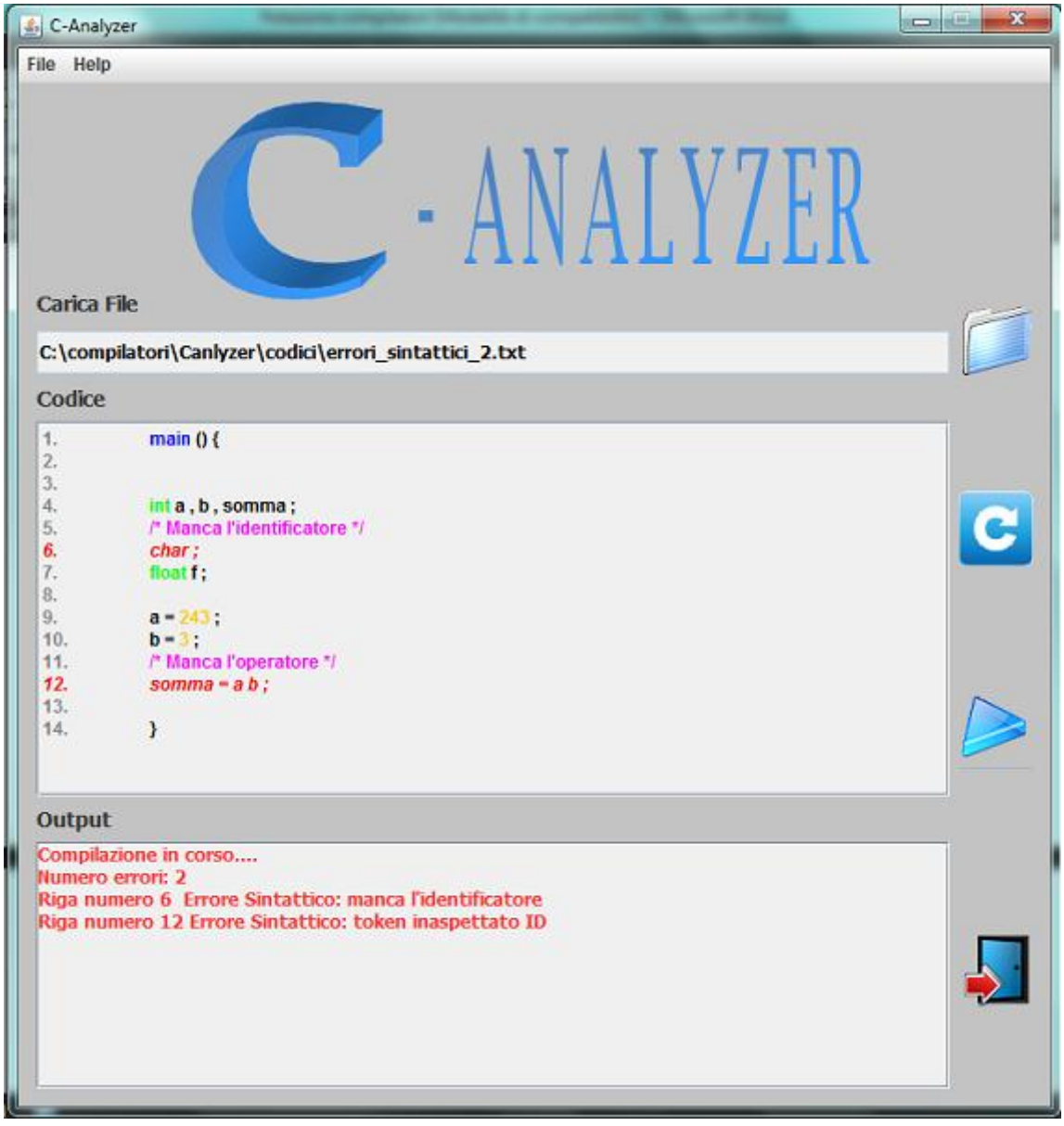

Figura 9

- In questo ultimo esempio sono presentati alcuni errori tipici dell'istruzione if

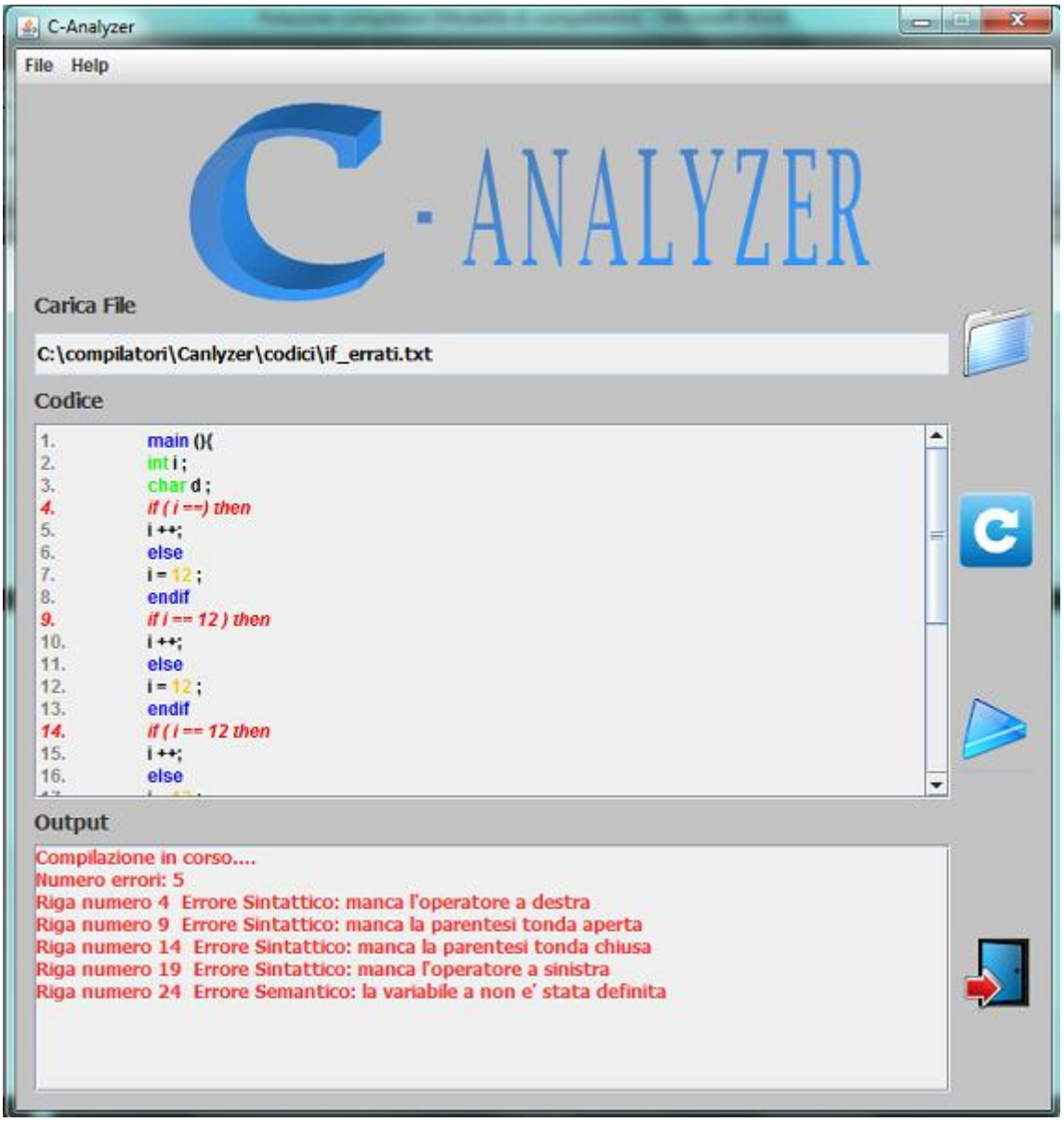

Figura 10

ANALISI SEMANTICA

L'analisi semantica è eseguita dal parser in collaborazione con la symbol table, la quale è implementata nel file sematica.h riportato in appendice.
Durante il check della grammatica tutte le variabili dichiarate insieme ai loro attributi semantici (identificatore, valore e dimensione) sono caricate nella symbol table; C-Analyzer, grazie a queste informazioni, sarà in grado di gestire quattro tipologie di errori:

- se una stessa variabile è dichiarata più di una volta

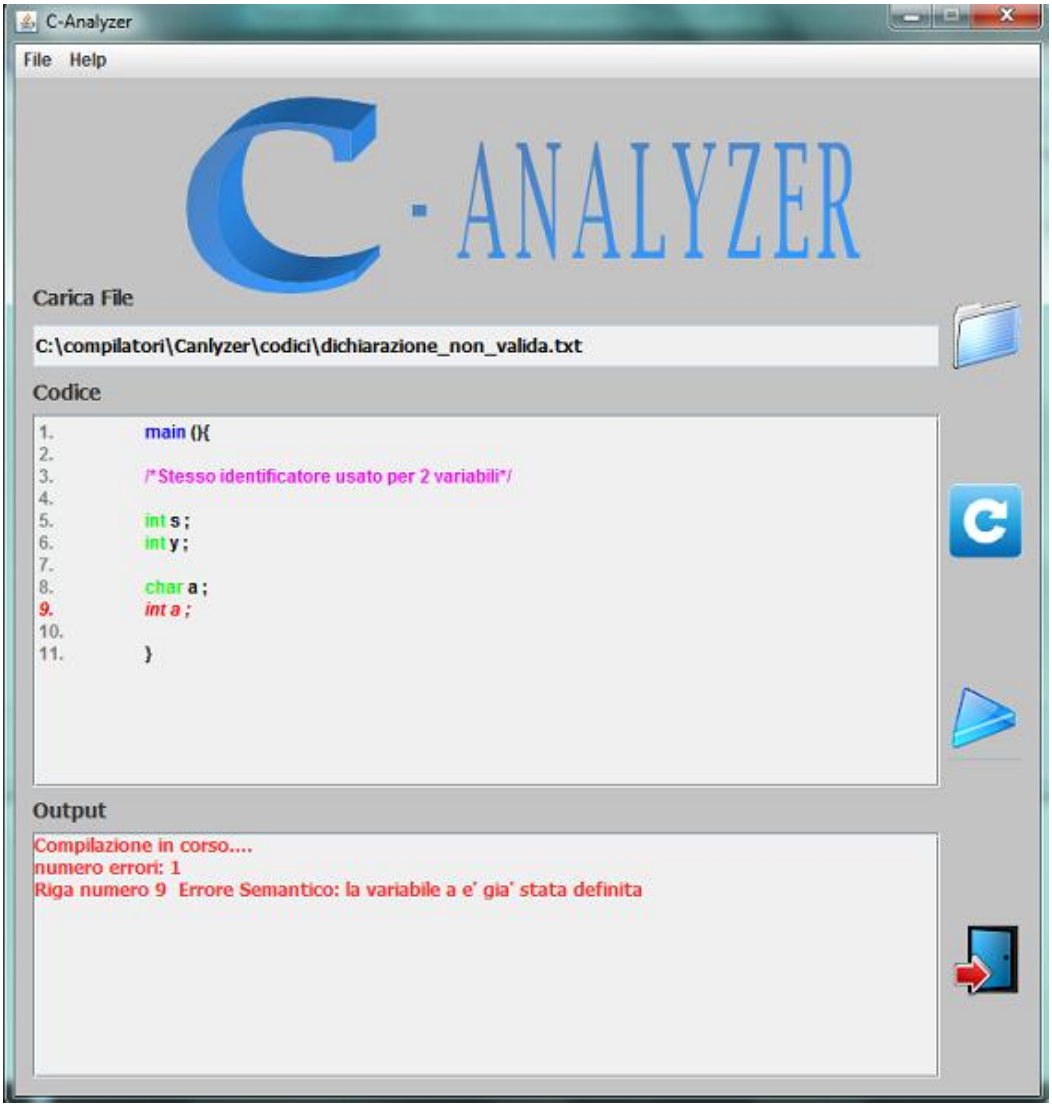

Figura 11

- se una variabile è utilizzata in maniera semanticamente errata (controllo dei tipi)

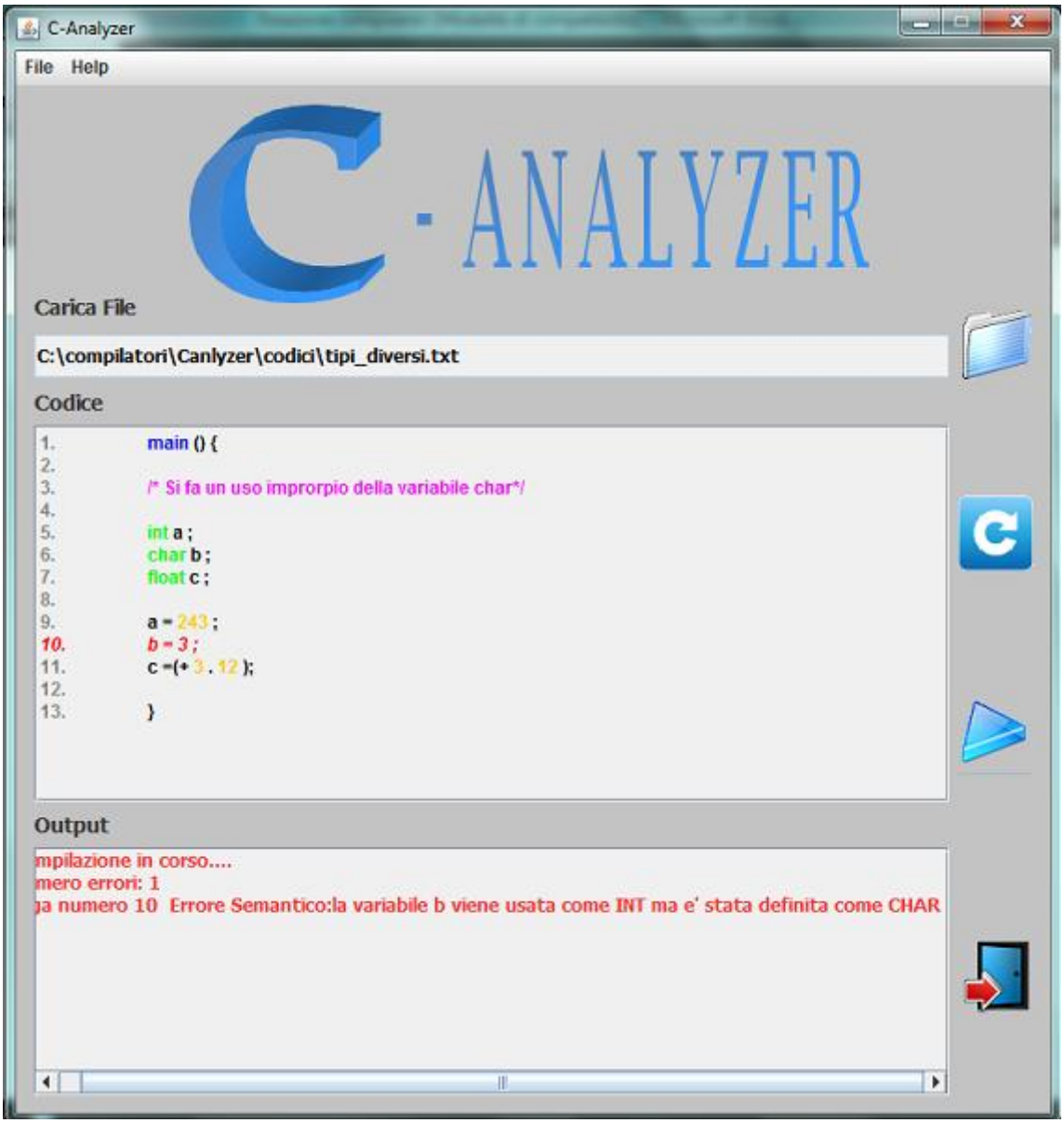

Figura 12

- se si cerca di leggere o scrivere fuori dalla dimensione dichiarata nel vettore

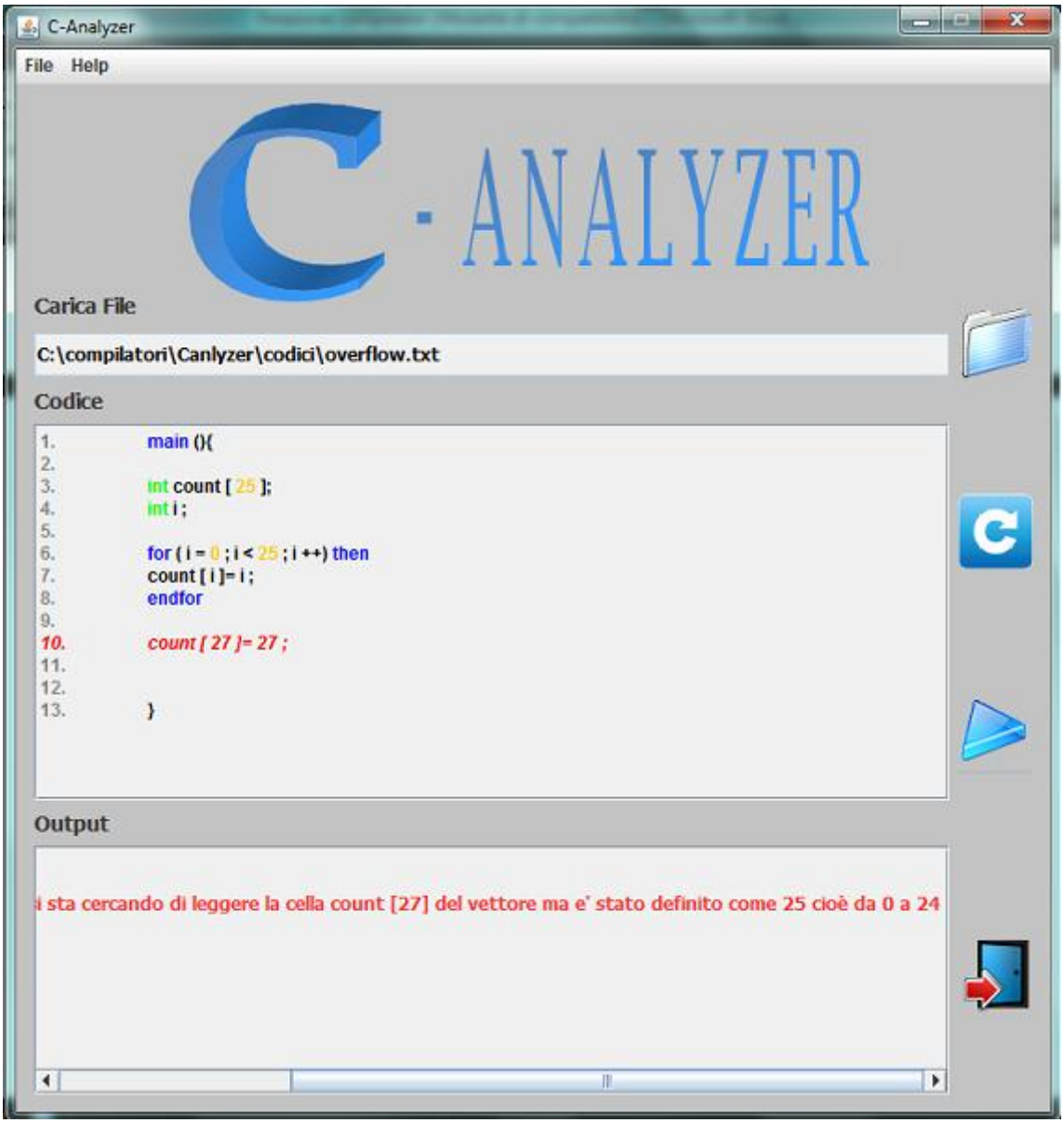

Figura 13

- se una variabile non viene dichiarata nella sezione dedicata alla dichiarazione delle variabili

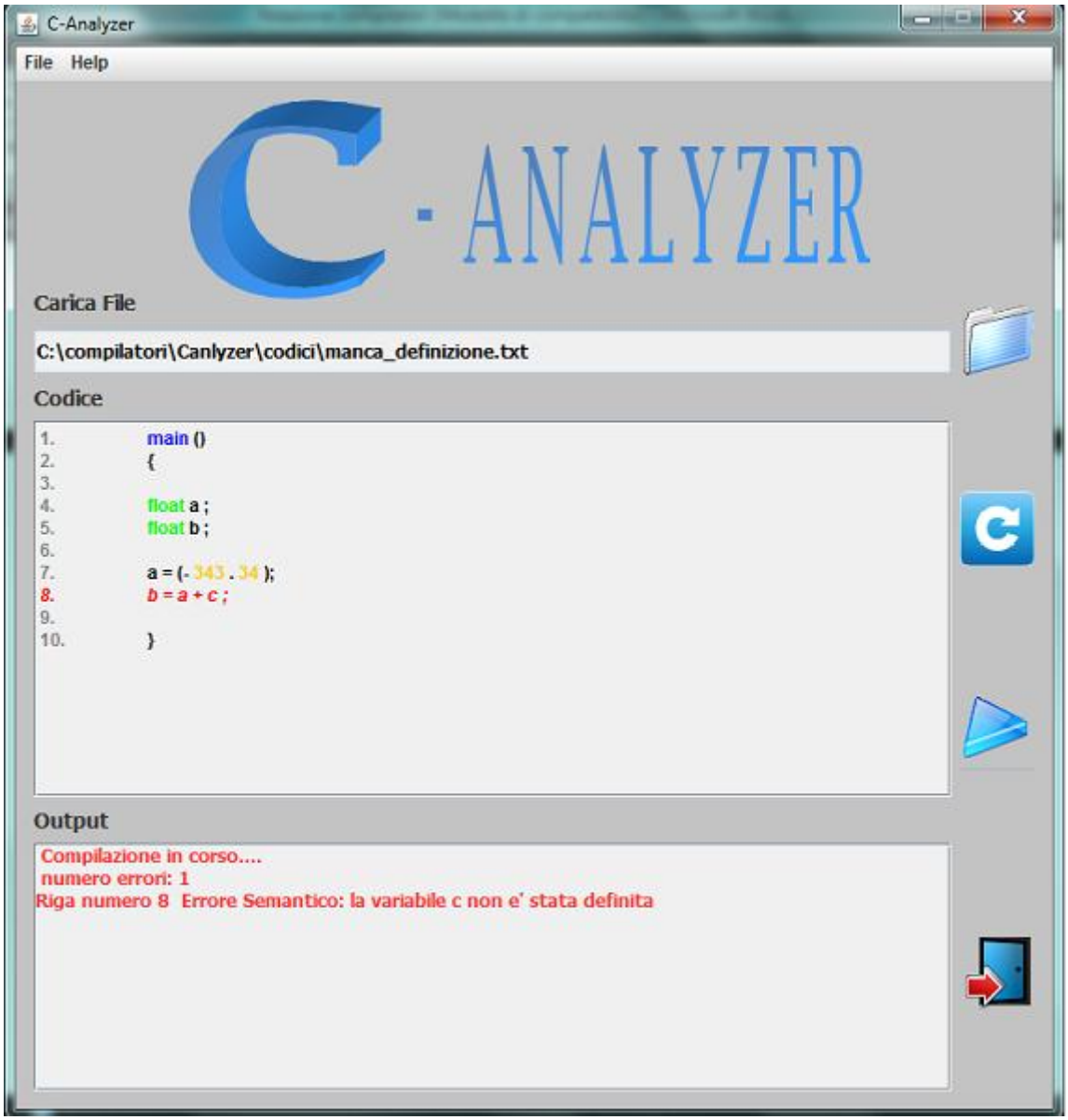

Figura 14

APPENDICE

Comandi per la compilazione e l'avvio

C-Analyzer può funzionare in due modalità:

- testuale: dal prompt dei comandi
- interfaccia grafica: lanciando il file canalyzer.jar presente nella cartella compilatori\Canalyzer\dist.

Per il corretto funzionamento dell'interfaccia grafica bisogna copiare la cartella del progetto in C:\; per quanto concerne la modalità da prompt dei comandi invece bisogna preliminarmente installare i software Flex, Bison, Gcc presenti nella cartella compilatori\setup. Dopo aver installato i software è necessario aprire il pannello di controllo, fare doppio click sull'icona "Sistema"; a questo punto cliccare su "impostazioni di sistema avanzate"; dopo di che selezionare il bottone "Variabili d'ambiente". In questa nuova finestra, selezionare dal pannello "Variabili di sistema", la variabile "Path" e aggiungere la seguente riga:

C:\Programmi\GnuWin32\bin;C:\MinGw\bin;

infine cliccare su ok per applicare la modifica effettuata.

In questa maniera sarà possibile compilare i sorgenti, semplicemente selezionando il path contenente questi ultimi. Sarà quindi evitata la procedura di selezionare i path che contengono i software Flex, Bison e gcc fondamentali per la compilazione.

A questo punto per compilare ed avviare C-Analyzer, è necessario aprire il prompt dei comandi, posizionarsi nella cartella C:\compilatori\Canalyzer\codici contenente i sorgenti *scanner.l* e *parser.y,* ed eseguire i seguenti comandi:

1. **bison –d parser.y** (genera i file **parser.tab.c** e **parser.tab.h**)
2. **gcc –c parser.tab.c** (compila il file **parser.tab.c** e genera **parser.tab.o**)
3. **flex scanner.l** (genera il file **lex.yy.c**)
4. **gcc –c lex.yy.c** (compila il file **lex.yy.c** e genera **lex.yy.o**)
5. **gcc parser.tab.o lex.yy.o –o nomeseguibile.exe** (compila i file **parser.tab.o** e **lex.yy.o** e genera l'eseguibile "**nomeseguibile.exe**")
6. **nome_eseguibile.exe nome_file_da analizzare** (avvia il programma generato "nome seguibile.exe" e passa come parametro d'ingresso il file in dialetto-C da analizzare)

Scanner.l

```
/*Definizioni*/

D                       [0-9]

L                       [a-zA-Z_]


%{
#include "parser.tab.h"
#include <stdio.h>
#include<string.h>
char vett[255];
void commento();
void erroreLessicale(int);
void yyerror(char*);
int numlines=1;
int num();

%}

/* %option main */

%option noyywrap

tipo_token          int|float|char
unsigned_number         [0-9]+
signed_number           ("(+"|"(-")[0-9]+(")")
floating            ("(+"|"(-")(([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+))(")")
stringa             \"[A-Za-z_][A-Za-z0-9_ ]*\"
str_char            \'[A-Za-z]\'
identificatore          [A-Za-z_][A-Za-z0-9_]*
relop               ("=="|">"|"<"|"<>")



%%

"/*"                    { commento(); }

" "  ;
"\r\n" ;
"\r" ;
"\n" ;              { ++numlines; }

"print_int"             return PRINT_INT;
"print_char"            return PRINT_CHAR;
"print_float"           return PRINT_FLOAT;
"print_vett_int"        return PRINT_VETT_INT;
"print_vett_char"       return PRINT_VETT_CHAR;
```

```
"print_vett_float"      return PRINT_VETT_FLOAT;
"scan_int"              return SCAN_INT;
"scan_char"             return SCAN_CHAR;
"scan_float"            return SCAN_FLOAT;
"scan_vett_int"         return SCAN_VETT_INT;
"scan_vett_char"        return SCAN_VETT_CHAR;
"scan_vett_float"       return SCAN_VETT_FLOAT;
"main"                  return MAIN;
"if"                    return IF;
"else"                  return ELSE;
"endif"                 return ENDIF;
"while"                 return WHILE;
"endwhile"              return ENDWHILE;
"switch"                return SWITCH;
"default"               return DEFAULT;
"endswitch"             return ENDSWITCH;
"case"                  return CASE;
"break"                 return BREAK;
"for"                   return FOR;
"then"                  return THEN;
"endfor"                return ENDFOR;
"exit"                  return EXIT;
"++"                    return INCR;
"--"                    return DECR;
"("                     return TONDAPERTA;
")"                     return TONDACHIUSA;
"-="                    return MENO_UG;
"+="                    return PIU_UG;


"+"                     return PIU;
"-"                     return MENO;
"*"                     return PER;
"/"                     return DIVISO;
"="                     return UGUALE;


"["                     return QUADRAPERTA;
"]"                     return QUADRACHIUSA;
"{"                     return GRAFFAPERTA;
"}"                     return GRAFFACHIUSA;


";"                     return PUNTOEVIRGOLA;
","                     return VIRGOLA;
":"                     return DUEPUNTI;

L?\"(\\.|[^\\"\n])*        { erroreLessicale(2);  }
```

```
{relop}                 {
                        yylval.id = (char *)strdup(yytext);
                        return RELOP;
                }

{tipo_token}    {
                        yylval.id = (char *)strdup(yytext);
                        return TIPO;
                }

{stringa}       {
                        yylval.id = (char *)strdup(yytext);
                        return STRINGA;
                }

{str_char}      {
                        strcpy(vett,&yytext[1]);
                        vett[strlen(vett)-1]='\0';
                        yylval.id = (char *)strdup(vett);
                        return STR_CHAR;
                }

{signed_number} {
                        strcpy(vett,&yytext[1]);
                        vett[strlen(vett)-1]='\0';
                        yylval.intval=atoi(vett);
                        return SIGNED_NUMBER;
                }

{unsigned_number}       {
                        yylval.intval=atoi(yytext);
                        return UNSIGNED_NUMBER;
                }

{floating}      {
                        strcpy(vett,&yytext[1]);
                        vett[strlen(vett)-1]='\0';
                        yylval.float_value=atof(vett);
                        return FLOAT_NUMBER;
                }


([0-9])+([A-Za-z0-9_])* {erroreLessicale(1);}


{identificatore}        {
                        yylval.id = (char *)strdup(yytext);
                        return ID;
                }
```

```
.                       {erroreLessicale(0);}


%%

void commento()
{
        char c, prev = 0;

        while ((c = input()) != 0)
        {
                if(c=='\n') numlines++;

                if (c == '/' && prev == '*') return;

                prev = c;
        }

        yyerror("Commenti non terminati\n");
}


int num() { return numlines;}


void erroreLessicale(int idErrore)
{
        char messaggioErrore[1024];
        int errore_riga_lessicale;

        switch(idErrore)
        {
                case 1:
                        sprintf(messaggioErrore, " Errore lessicale: I nomi degli identificatori
possono iniziare solo con una lettera o un underscore");
                        break;

                case 2:
                        sprintf(messaggioErrore, " Errore lessicale: Virgolette aperte ma non
chiuse\n");
                        break;

                case 0:
                        sprintf(messaggioErrore, " Errore lessicale: Elemento non riconosciuto\n");
        }

        yyerror(messaggioErrore);
```

```
}
```

Parser.y

```
/* PROLOGO */

%{
#include <stdio.h>     /*Serve per le operazioni di I/O*/
#include <stdlib.h>    /* Serve per allocare memoria nella tabella dei simboli*/
#include <string.h>    /* Serve per la strcmp  nella tabella dei simboli */
#include <ctype.h>
#include <conio.h>
#include "semantica.h"          /* Tabella dei simboli */
#define FALSE 0
#define TRUE 1
#define MAXBUF 255

int m=0;
int k=0;
int b=0;

struct message
{
        int numero_linea;
        char nome[1024];

} message[100];



int num();

int yylex(void);
void yyerror(char *);
void error_grave(char *);
char tmp[MAXBUF],var[MAXBUF];
int int_tmp=0,i=0,vettore=0;
void yyerrorr(char*);

%}

/* DICHIARAZIONI */

%union semrec {
        int intval;             /* Integer values */
        char *id;               /* Identifiers */
        float float_value;
}

%start input
```

```
%token PRINT_INT PRINT_CHAR PRINT_VETT_INT PRINT_VETT_CHAR
%token SCAN_INT SCAN_CHAR SCAN_VETT_INT SCAN_VETT_CHAR
PRINT_VETT_FLOAT SCAN_VETT_FLOAT
%token SCAN_FLOAT PRINT_FLOAT
%token IF ELSE WHILE SWITCH CASE FOR BREAK EXIT DEFAULT
%token PIU MENO DIVISO PER INCR DECR UGUALE
%token TONDAPERTA TONDACHIUSA QUADRAPERTA QUADRACHIUSA
GRAFFAPERTA GRAFFACHIUSA
%token PUNTOEVIRGOLA DUEPUNTI VIRGOLA
%token MAIN FREE ENDIF ENDWHILE ENDFOR THEN ENDSWITCH MENO_UG PIU_UG
%token <intval> SIGNED_NUMBER UNSIGNED_NUMBER
%token <float_value> FLOAT_NUMBER
%token <id> ID TIPO RELOP STR_CHAR STRINGA


%left PIU MENO
%left DIVISO PER
%left INCR DECR

%error-verbose

%%

/* REGOLE */

/* Simbolo iniziale della grammatica */

input:  /*niente*/

        |input start
;

start : MAIN TONDAPERTA TONDACHIUSA  GRAFFAPERTA  parameter_list stmt_list
GRAFFACHIUSA


/*inizio gestione errori sintattici*/

        |MAIN  TONDACHIUSA  {yyerror(" Errore Sintattico: manca la parentesi tonda aperta ");}


        |MAIN GRAFFAPERTA {yyerror(" Errore Sintattico: mancano le parentesi tonde ");}
;
/*fine gestione errori sintattici*/



parameter_list: parameter_list parameter
        | parameter
;
```

```
parameter : TIPO ID PUNTOEVIRGOLA
        {
                if(strcmp($1,"int")==0)         { install($2,INT,1);}

                if(strcmp($1,"char")==0)        { install($2,CHAR,1);}

                if(strcmp($1,"float")==0)       {install($2,FLOAT,1);}
        }


/*inizio sezione gestione errori sintattici*/

        |TIPO ID                  { yyerror(" Errore Sintattico: manca il ;");}

        |TIPO PUNTOEVIRGOLA   { yyerror(" Errore Sintattico: manca l'identificatore");}

        |ID PUNTOEVIRGOLA       { yyerror(" Errore Sintattico: manca il tipo"); }

/*fine sezione gestione errori sintattici*/

        |TIPO ID
        {
                if(strcmp($1,"int")==0)                 {install($2,INT,1);}

                if(strcmp($1,"char")==0)        {install($2,CHAR,1);}

                if(strcmp($1,"float")==0)       {install($2,FLOAT,1);}

                strcpy(tmp,$1);
        }
        p_lists  PUNTOEVIRGOLA


        |TIPO ID QUADRAPERTA UNSIGNED_NUMBER QUADRACHIUSA
PUNTOEVIRGOLA
        {
                if(strcmp($1,"int")==0)                 {install($2,V_INT,$4);}

                if(strcmp($1,"char")==0)        {install($2,V_CHARS,$4);}

                if(strcmp($1,"float")==0)       {install($2,V_FLOAT,$4);}

        }


/*inizio sezione gestione errori sintattici*/

        | TIPO ID QUADRAPERTA UNSIGNED_NUMBER QUADRACHIUSA          {yyerror("
Errore Sintattico: manca il ;");}
```

```
        | TIPO ID UNSIGNED_NUMBER QUADRACHIUSA PUNTOEVIRGOLA      {yyerror("
Errore Sintattico: manca la parentesi quadra aperta");}

        | TIPO ID QUADRAPERTA UNSIGNED_NUMBER PUNTOEVIRGOLA      {yyerror("
Errore Sintattico: manca la parentesi quadra chiusa");}

/*fine sezione gestione errori sintattici*/


        | TIPO ID
        {
                if(strcmp($1,"int")==0)         {       strcpy(var,$2);
                                                install($2,V_INT,1);
                                        }

                if(strcmp($1,"char")==0)
                                        {
                                                strcpy(var,$2);
                                                install($2,V_CHARS,1);
                                                //sprintf(message,"Errore nella dichiarazione di
\"%s\"",$2);
                                yyerror("Errore Sintattico: errore nella dichiarazione della variabile");
                                        }

                if(strcmp($1,"float")==0)
                                        {
                                                strcpy(var,$2);
                                                install($2,V_FLOAT,1);
                                        }
        }
        GRAFFAPERTA  vet_lists GRAFFACHIUSA PUNTOEVIRGOLA
;

p_lists : p_lists p_list
        |p_list
;


p_list  : VIRGOLA ID
        {
                if(strcmp(tmp,"int")==0)        {install($2,INT,1);}

                if(strcmp(tmp,"char")==0)       {install($2,CHAR,1);}

                if(strcmp(tmp,"float")==0)      {install($2,FLOAT,1);}
        }

        |VIRGOLA ID QUADRAPERTA UNSIGNED_NUMBER QUADRACHIUSA
        {
                if(strcmp(tmp,"int")==0)        {install($2,V_INT,$4);}
```

```
                if(strcmp(tmp,"char")==0)     {install($2,V_CHARS,$4);}

                if(strcmp(tmp,"float")==0)    {install($2,V_FLOAT,$4);}

        }
;


vet_lists : vet_lists VIRGOLA  vet_list
        |vet_list
;

vet_list: SIGNED_NUMBER                 {incr_dim(var);}

      |UNSIGNED_NUMBER          {incr_dim(var);}

        |FLOAT_NUMBER             {incr_dim(var);}

;

stmt_list : stmts
                        | /* niente */
;

stmts : stmts stmt
        |stmt
;

stmt:  assign_stmt

        | if_stmt

        | WHILE rel_expr  THEN  stmt_list ENDWHILE

        | EXIT  PUNTOEVIRGOLA

        | FOR  TONDAPERTA  assign_stmt for_cond PUNTOEVIRGOLA  expr_for
TONDACHIUSA THEN stmt_list ENDFOR

        | SWITCH TONDAPERTA ID { context_check($3,-1,0);} TONDACHIUSA  THEN
case_list ENDSWITCH

        | print_and_scan PUNTOEVIRGOLA


/*inizio gestione errori sintattici*/

        | print_and_scan  {yyerror(" Errore Sintattico: manca il ;");}

/*fine gestione errori sintattici*/
```

```
;

print_and_scan: PRINT_INT ID      {context_check($2,INT,1);}
        | PRINT_FLOAT ID          {context_check($2,FLOAT,1);}
        | PRINT_CHAR ID   {context_check($2,CHAR,1);}
        | PRINT_VETT_INT ID       {context_check($2,V_INT,1);}
        | PRINT_VETT_FLOAT ID {context_check($2,V_FLOAT,1);}
        | PRINT_VETT_CHAR ID  {context_check($2,V_CHARS,1);}
        | SCAN_INT ID             {context_check($2,INT,1);}
        | SCAN_FLOAT ID   {context_check($2,FLOAT,1);}
        | SCAN_CHAR ID    {context_check($2,CHAR,1);}
        | SCAN_VETT_INT ID        {context_check($2,V_INT,1);}
        | SCAN_VETT_FLOAT ID  {context_check($2,V_FLOAT,1);}
        | SCAN_VETT_CHAR ID   {context_check($2,V_CHARS,1);}
;




for_cond: espressione
;

assign_stmt:  ID UGUALE
              {
                    int_tmp=context_check($1,-1,0);
                    vettore=FALSE;
                    strcpy(var,$1);
              }
              expr PUNTOEVIRGOLA

              | ID MENO_UG
              {
                    int_tmp=context_check($1,-1,0);
                    vettore=FALSE;
                    strcpy(var,$1);
              }
              expr PUNTOEVIRGOLA
```

```
| ID PIU_UG
{
        int_tmp=context_check($1,-1,0);
        vettore=FALSE;
        strcpy(var,$1);
}
expr PUNTOEVIRGOLA

| ID QUADRAPERTA UNSIGNED_NUMBER QUADRACHIUSA UGUALE
{
        int_tmp=context_check($1,-2,$3);
        vettore=TRUE;
        strcpy(var,$1);
}
expr PUNTOEVIRGOLA


| ID QUADRAPERTA ID QUADRACHIUSA UGUALE
{
        int_tmp=context_check($1,-1,0);
        vettore=TRUE;
        context_check($3,INT,1);
        strcpy(var,$1);
}
expr PUNTOEVIRGOLA

| ID QUADRAPERTA UNSIGNED_NUMBER QUADRACHIUSA PIU_UG
{
        int_tmp=context_check($1,-2,$3);
        vettore=TRUE;
        if(int_tmp!=V_INT && int_tmp!=V_FLOAT)
        context_check($1,V_INT,1);
        strcpy(var,$1);
}
expr PUNTOEVIRGOLA

| ID QUADRAPERTA UNSIGNED_NUMBER QUADRACHIUSA MENO_UG
{
        int_tmp=context_check($1,-2,$3);
        vettore=TRUE;
        if(int_tmp!=V_INT && int_tmp!=V_FLOAT)
                context_check($1,V_INT,$3);
        strcpy(var,$1);
}
expr PUNTOEVIRGOLA

| ID QUADRAPERTA ID QUADRACHIUSA PIU_UG
{
int_tmp=context_check($1,-1,0);
        vettore=TRUE;
        context_check($3,INT,1);
```

```
                if(int_tmp!=V_INT && int_tmp!=V_FLOAT)
                        context_check($1,V_INT,1);
                strcpy(var,$1);
        }
        expr PUNTOEVIRGOLA

        | ID QUADRAPERTA ID QUADRACHIUSA MENO_UG
        {
                int_tmp=context_check($1,-1,0);
                vettore=TRUE;
                context_check($3,INT,1);
                if(int_tmp!=V_INT && int_tmp!=V_FLOAT)
                        context_check($1,V_INT,1);
                strcpy(var,$1);
        }
        expr PUNTOEVIRGOLA

        | ID INCR PUNTOEVIRGOLA
        {
                context_check($1,INT,1);
        }

        | ID DECR PUNTOEVIRGOLA
        {
                context_check($1,INT,1);
        }


/*inizio gestione errori sintattici*/

        | ID INCR       { yyerror(" Errore Sintattico: manca il ; ");}

        | ID DECR       {yyerror(" Errore Sintattico: manca il ; ");}

/*fine gestione errori sintattici*/


        | ID QUADRAPERTA UNSIGNED_NUMBER QUADRACHIUSA DECR
PUNTOEVIRGOLA
        {
                context_check($1,V_INT,1);
                context_check($1,-2,$3);
        }

        | ID QUADRAPERTA UNSIGNED_NUMBER QUADRACHIUSA INCR
PUNTOEVIRGOLA
        {
                context_check($1,V_INT,1);
                context_check($1,-2,$3);
        }
```

```
        | ID QUADRAPERTA ID QUADRACHIUSA DECR PUNTOEVIRGOLA
        {
                context_check($3,INT,1);
                context_check($1,V_INT,1);
        }

        | ID QUADRAPERTA ID QUADRACHIUSA INCR PUNTOEVIRGOLA
        {
                context_check($3,INT,1);
                context_check($1,V_INT,1);
        }
;


term:   ID
        {
                i=context_check($1,-1,1);
                if(int_tmp!=i && i!=int_tmp-4)
                        context_check($1,int_tmp,1);
        }

        | SIGNED_NUMBER
        {
                if(vettore==FALSE && int_tmp!=INT)
                        context_check(var,INT,1);
                if(vettore==TRUE && int_tmp!=V_INT)
                        context_check(var,V_INT,1);
        }

        | UNSIGNED_NUMBER
        {
                if(vettore==FALSE && int_tmp!=INT)
                        context_check(var,INT,1);
                if(vettore==TRUE && int_tmp!=V_INT)
                        context_check(var,V_INT,1);
        }

        |FLOAT_NUMBER
        {
                if(vettore==FALSE && int_tmp!=FLOAT)
                        context_check(var,FLOAT,1);
                if(vettore==TRUE && int_tmp!=V_FLOAT)
                        context_check(var,V_FLOAT,1);
        }

        | ID QUADRAPERTA UNSIGNED_NUMBER QUADRACHIUSA
        {
                i=context_check($1,-1,1);
                context_check($1,-2,$3);
                if(int_tmp!=i && int_tmp!=i-4){yyerror(" Errore Sintattico: Le due variabili usate
nella condizione non sono di 2 tipi compatibili");}
```

```
        }

        | ID QUADRAPERTA ID QUADRACHIUSA
        {
                i=context_check($1,-1,1);
                context_check($3,INT,1);
        }

        | STRINGA
        {
                context_check(var,V_CHARS,1);
        }

        | STR_CHAR
        {
                context_check(var,CHAR,1);
        }
;

expr:   expr PIU  expr
        | expr MENO  expr
        | expr DIVISO  expr
        | expr PER  expr
        | TONDAPERTA  expr TONDACHIUSA
        | term
;


if_stmt:        IF rel_expr THEN stmt_list else_stmt
;

espressione:    term1 RELOP expr

                |term1 expr { yyerror(" Errore Sintattico: manca l'operatore relazionale");}

                |term1 RELOP{ yyerror(" Errore Sintattico: manca l'operatore a destra");}

                |RELOP expr { yyerror(" Errore Sintattico: manca l'operatore a sinistra");}

                |term1 { yyerror(" Errore Sintattico: espressione incompleta");}
;

rel_expr:       TONDAPERTA espressione TONDACHIUSA

        |TONDAPERTA espressione { yyerror(" Errore Sintattico: manca la parentesi tonda
chiusa");}

                |espressione TONDACHIUSA { yyerror(" Errore Sintattico: manca la parentesi
tonda aperta");}

                |espressione { yyerror(" Errore Sintattico: mancano le parentesi tonde");}
```

```
;

else_stmt:      ENDIF

                |ELSE   stmt_list ENDIF
;


term1: ID
        {
                strcpy(var,$1);
                int_tmp=context_check($1,-1,1);
                vettore=FALSE;
        }

        |ID QUADRAPERTA ID QUADRACHIUSA
        {
                int_tmp=context_check($1,-1,1);
                context_check($3,-1,1);
                vettore=TRUE;
                strcpy(var,$1);
        }

        |ID QUADRAPERTA UNSIGNED_NUMBER QUADRACHIUSA
        {
                int_tmp=context_check($1,-1,1);
                context_check($1,-2,$3);
                vettore=TRUE;
                strcpy(var,$1);
        }
;

case_list:      case_list case_stmt  BREAK  PUNTOEVIRGOLA

                | case_stmt  BREAK  PUNTOEVIRGOLA

/*inizio gestione errori sintattici*/

                |case_list case_stmt  BREAK          {yyerror(" Errore Sintattico: manca il ; ");}

                | case_stmt  BREAK           {yyerror(" Errore Sintattico: manca il ; ");}

/*fine gestione errori sintattici*/
;

case_stmt:      CASE UNSIGNED_NUMBER DUEPUNTI stmt_list

                |CASE SIGNED_NUMBER DUEPUNTI stmt_list

                | CASE ID DUEPUNTI
                {
```

```
                context_check($2,-1,1);
            }
            stmt_list

            | CASE STRINGA DUEPUNTI stmt_list

            | CASE STR_CHAR DUEPUNTI stmt_list

            | DEFAULT DUEPUNTI stmt_list
;

expr_for:   ID INCR             {context_check($1,INT,1);}

            |ID DECR    {context_check($1,INT,1);}
;



%%

/* EPILOGO */

int main( int argc, char *argv[] )
{
    int j;
    extern FILE *yyin;

    if(argc!=2)
    {
        printf("Errore!\nPer far partire il programma serve il nome\ndel file con il codice C
che si vuole analizzare\n");
        printf("ESEMPIO: %s <file_da_analizzare> \n",argv[0]);
        return 0;
    }

    --argc;
    ++argv;

    if((yyin = fopen( argv[0], "r" ))==NULL)
    {
        printf("Errore nell'apertura del file, controllare\n");
        printf("che la path inserita (%s) sia corretta\n", argv[0]);
        return 0;
    }

    yyparse ();


    if(k<=0)
    {
        printf("\nCompilazione in corso....\n");
```

```
		printf("Programma riconosciuto correttamente");
	}

	else
	{
		printf("\nCompilazione in corso....\n");
		printf("Numero errori: %d \n",k);

		for (j=0;j<k;j++)
		{
			printf ("Riga numero %d ", message[j].numero_linea);
			printf ("%s ", message[j].nome);
			printf ("\n");
		}

	}

	return 0;

}


void yyerror ( char *s ) /* chiamata da yyparse in caso di errore */
{
	int a=0;
	a=num();


	if(a!=message[k-1].numero_linea)
	{
		strcpy(message[k].nome,s);
		message[k].numero_linea=a;
		k=k+1;
	}
}


void error_grave(char *s)
{
	int a=0;
	int j=0;
	int count=0;
	count=k+1;

	printf("\nCompilazione in corso....\n");
	printf("Numero errori: %d \n",count);

	for (j=0;j<k;j++)
	{
		printf ("Riga numero %d ", message[j].numero_linea);
		printf ("%s ", message[j].nome);
```

```
                printf ("\n");
        }

        a=num();

        printf("Riga numero %d ", a);
        printf("%s", s);
        exit(0);
}
```

Semantica.h

/* Definiamo la variabile tipo_var, la quale può assumere uno dei qualsiasi valori indicati tra graffe. Creiamo un nuovo tipo di dato, ovvero tipo_variaible, il quale è di tipo tipo_var, che a sua volta può assumere uno dei valori racchiusi tra graffe*/

```
enum tipo_var {INT=1,FLOAT,CHAR,vet,V_INT,V_FLOAT,V_CHARS,ESPRESSIONE};
typedef enum tipo_var tipo_variabile;


#include <string.h>
#include <conio.h>
#include <stdlib.h>
#include <stdio.h>

void yyerror(char *);
void error_grave(char *);
int num();
int numero_errori();
int nn =0 ;
int lines = 0;
char
*tipo_var_char[]={"","INT","FLOAT","CHAR","vet","V_INT","V_FLOAT","V_CHARS","ESPR
ESSIONE"};

/*---------------------------------------------------------------------
RECORD DELLA TABELLA DEI SIMBOLI
---------------------------------------------------------------------*/

//un record della tabella dei simboli è una struttura avente i seguenti campi:

struct symrec
{
        char *name;             /* nome del simbolo */
        int dim_vett;
        struct symrec *next; /* puntatore a un campo della struttura */
        tipo_variabile type;
};

typedef struct symrec symrec;
```

```
/* sym_table è un puntatore alla struttura symrec (la tabella dei simboli) e lo inizializziamo a zero.
*/

symrec *sym_table = (symrec *)0;

//le operazioni possibili sono: putsym, getsym


/*con la funzione putsym restituiamo il puntatore alla tabella dei simboli, che punta al nuovo record
inserito.*/

symrec * putsym (char *sym_name, int val,int dim)
{
	symrec *ptr;
	ptr = (symrec *) malloc (sizeof(symrec));
	ptr->name = (char *) malloc (strlen(sym_name)+1);
	strcpy (ptr->name,sym_name);
	ptr->type=val;
	ptr->dim_vett=dim;
	ptr->next = (struct symrec *)sym_table;
	sym_table = ptr;
	return ptr;
}

/*con la funzione getsym restituiamo un puntatore alla tabella dei simboli.*/

symrec * getsym (char *sym_name)
{
	symrec *ptr;
	for ( ptr = sym_table; ptr!=(symrec *)0; ptr=(symrec *)ptr->next)
		if (strcmp (ptr->name,sym_name) == 0)
			return ptr;
	return 0;
}


/* con questa funzione viene svuotata la tabella dei simboli a partire dal record di testa.*/

void svuota_ST()
{
	symrec *ptr,*next;
	for ( ptr = sym_table; ptr!=(symrec *)0; )
	{
		next=ptr;
		ptr=(symrec *)ptr->next;
		free(next);
	}
}
```

```
/* con la funzione install effettuiamo un controllo semantico.
l'obiettivo della funzione è evitare di ridefinire  una variabile già definita.
*/


void install ( char *sym_name, int tipo_var,int dim)
{
        char stro[80];
        symrec *s;
        s = getsym (sym_name);
        if (s == 0)
                s = putsym (sym_name,tipo_var,dim);
        else {
                sprintf(stro," Errore Semantico: la variabile %s e' gia' stata definita",sym_name);

                yyerror(stro);
         }

}

/* con la funzione incr_dim viene gestita la dichiarazione e la contemporanea
 istanziazione degli elementi di un vettore*/
void incr_dim( char *sym_name)
{
        symrec *s;
        s = getsym (sym_name);
        s->dim_vett++;

}

/* con la funzione context_check vengono gestiti altri 3 tipi di errori semantici.
 variabile non definita.
 errore semantico di tipi
 overflow
*/

int context_check( char *sym_name, int val ,int dim)
{
int va;
int vi;
int vu;

va = dim;

char sip[50];
char sim[50];
char strf[180];

char ci[10];
char pi[10];
char si[10];
```

```
char msg[60];
itoa(va,ci,10);

strcpy(sim,sym_name);

symrec *identifier;



        identifier = getsym( sym_name );

        if ( identifier == 0 )
        {

           sprintf(sip," Errore Semantico: la variabile %s non e' stata definita", sym_name);

                error_grave(sip);

        }

        if(val==-1)
                return identifier->type;
        else if(val==-2)
        {
                if(identifier->dim_vett<=dim)
                {


                vi=identifier->dim_vett;

                vu= vi-1;

                itoa(vi,pi,10);


                itoa(vu,si,10);

                sprintf(strf," Errore Semantico: overflow, si si sta cercando di leggere la cella
%s[%s] del vettore ma e\' stato definito da 0 a %s",sym_name,ci,si);


                yyerror( strf );

                }
                return identifier->type;
        }
        else
        {
                if (identifier->type!=val)
                {
```

```
                    sprintf(msg," ERRORE: la variabile \"%s\" viene usata come %s ma e\' definita come %s",sym_name,tipo_var_char[val],tipo_var_char[identifier->type]);

                    yyerror(msg);

                }
        }


        return 0;
}
```

Istruzioni I/O ad-hoc

```
print_int  VAR;
print_char  VAR;
print_float   VAR;
print_vett_int  VAR;
print_vett_char VAR;
print_vett_float  VAR;
scan_int  VAR;
scan_char  VAR;
scan_float  VAR;
scan_vett_int  VAR;
scan_vett_char  VAR;
scan_vett_float  VAR;
```

BIBLIOGRAFIA

[1] Slide del corso “Compilatori e interpreti”, Prof. Giacomo Piscitelli

[2] A compact guide to LEX & YACC