

POLITECNICO DI BARI
I FACOLTÀ DI INGEGNERIA - BARI
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

PROGETTO DI
LINGUAGGI FORMALI E COMPILATORI

IMPLEMENTAZIONE DI UN COMPILATORE
ADA 95
E TRADUZIONE ADA95 - C

Docente: **PISCITELLI Giacomo**

Studenti: **MOLENDINI Vincenzo**
TRIGIANTE Giuseppe

Anno Accademico 2010/2011

INDICE

Introduzione	2-5
Limitazioni	6-7
Script di compilazione	8
Analisi lessicale	9-14
Analisi sintattica	15-47
Analisi semantica e traduzione	48-113
Casi di test ed errori	114-126
Conclusioni	127
Bibliografia e Sitografia	128

Introduzione

Il lavoro svolto consiste nella realizzazione di un compilatore per il linguaggio di programmazione **ADA 95** limitando la grammatica di ADA 95 ed effettuando alcune semplificazioni per la realizzazione didattica del progetto.

Infatti non si è creato un compilatore per l'intero linguaggio di programmazione ADA 95, ma per alcune sue caratteristiche modificando in alcuni casi la grammatica di ADA 95.

Quindi si è realizzato un semplificato compilatore di un ADA 95.

Il progetto prevede inoltre **la traduzione di un programma ADA 95 in C**, dopo aver verificato la sua correttezza sintattica e semantica.

Il compilatore implementato è stato chiamato “*adac*”, per sottolineare la traduzione di un programma ADA 95 (programma sorgente) in un programma C (programma target).

Si è realizzata una traduzione ADA95 – C per poter conoscere ed imparare le caratteristiche di ADA 95, cioè un nuovo linguaggio di programmazione sconosciuto a noi studenti e molto importante poiché usato per la realizzazioni di software da parte del Dipartimento della Difesa (DOD) degli Stati Uniti.

Infatti ADA è un linguaggio di programmazione “*general-purpose*”, inizialmente sviluppato verso la fine degli anni settanta su iniziativa del Dipartimento della Difesa (DOD) degli Stati Uniti.

Inoltre si è scelto proprio la traduzione ADA95 – C per portare un programma scritto in ADA in un programma scritto in C, linguaggio di programmazione maggiormente conosciuto e più usato a livello didattico.

L'implementazione del progetto è avvenuta in ambiente Linux usando la distribuzione OpenSuse 11.3 versione rilasciata il 15 luglio 2010.

Per l'implementazione del compilatore ADA 95 sono stati usati due tool molto importanti che hanno semplificato la realizzazione del compilatore.

I tool usati sono **Flex** e **Bison**.

Flex è uno strumento che consente di realizzare un **analizzatore lessicale**, uno scanner; mentre **Bison** è un tool che consente di realizzare un **analizzatore sintattico**, un parser.

Flex si occupa della parte relativa all'analisi lessicale e ci permette di implementare un analizzatore lessicale (*scanner*) che effettua il riconoscimento di pattern all'interno di uno stream di input.

Bison si occupa di convertire una grammatica libera da contesto scritta in **BNF** (*Backus Naur Form*) nel corrispettivo "*parser*" producendo una funzione C che svolga questo compito.

Nella sezione "*Limitazioni*" sono descritte le limitazioni del compilatore "*adac*", in quanto come detto prima si è realizzato un compilatore ADA 95-C semplificato con delle limitazioni per la realizzazione del progetto didattico.

Nella sezione "*Script di compilazione*" è descritto lo script di comandi Linux usati per la realizzazione di "*adac*".

Nella sezione "*Analisi lessicale*" è descritto il concetto di analisi lessicale, il tool Flex e il file "*scanner.l*" che rappresenta l'analizzatore lessicale usato nel compilatore "*adac*" che effettua l'analisi lessicale del programma in linguaggio ADA 95 fornito in input.

Nella sezione "*Analisi sintattica*" è descritto il concetto di analisi sintattica, il tool Bison e il file "*parser.y*" che effettua l'analisi sintattica del programma in linguaggio ADA 95 fornito in input ad "*adac*".

Nella sezione "*Analisi semantica e traduzione*" è descritto il concetto di analisi semantica e di traduzione per un compilatore, la tecnica usata ed inoltre è analizzato il file "*semtrad.h*" che è una libreria inclusa nel file "*parser.y*" per effettuare l'analisi semantica e la traduzione del programma sorgente di "*adac*".

La libreria "*semtrad.h*" ha al suo interno delle funzioni che consentono al compilatore implementato di effettuare la traduzione del programma ADA fornito in input, in un file C che sarà l'output del compilatore, in assenza di errori sintattici o semantici.

La logica seguita per la realizzazione del compilatore "*adac*" è rappresentata dalle seguenti fasi:

1. L'analizzatore lessicale esegue la scansione del file sorgente individuando i vari **token** del programma sorgente, questi vengono passati di volta in volta al parser che cerca di riconoscere una regola di derivazione della grammatica implementata.
2. Riconosciuta una regola di derivazione dall'analizzatore sintattico la si verifica: se è una regola che genera errore sintattico allora viene visualizzato il messaggio di errore

sintattico in output da “adac” e il compilatore termina la sua esecuzione, altrimenti se è stata riconosciuta una regola di derivazione non associata ad un errore sintattico, quindi una regola associata ad uno statement o ad una operazione di definizione o di dichiarazione del linguaggio ADA 95 il compilatore analizza questa regola, effettua l’analisi semantica della regola riconosciuta e sintatticamente corretta.

3. L’analisi semantica della regola riconosciuta da “adac” avviene non tramite un albero astratto **AST** (*Abstract Syntax Tree*), ma si usa uno schema **SDT** (*Syntax Directed Translation*) uno schema di traduzione diretto per l’analisi semantica.

Uno schema SDT permette di eseguire azioni semantiche direttamente all’interno delle regole sintattiche che caratterizzano l’analisi sintattica, contenute nel file “parser.y”.

Si verifica che la regola di derivazione riconosciuta sia corretta oltre che sintatticamente anche semanticamente usando delle funzioni di checking (di controllo) definite nella libreria “semtrad.h”.

L’analisi semantica viene effettuata usando una lista, una struttura definita nella libreria “semtrad.h” che conterrà un campo “tes” (campo testo) è un puntatore all’elemento successivo della lista.

La lista conterrà tutti quei token e altre informazioni aggiuntive che caratterizzano la regola di produzione da analizzare dal punto di vista semantico.

Per la regola di produzione riconosciuta dal parser si verifica la sua coerenza dal punto di vista semantico, se questa regola non è semanticamente corretta a video appare all’utente un messaggio di errore semantico e l’esecuzione del compilatore termina.

4. Se la regola di derivazione riconosciuta è semanticamente corretta viene effettuata la traduzione di questa regola dal linguaggio ADA 95 al C. Per la fase di traduzione del compilatore “adac”, che consente di tradurre un programma in ADA 95 in un programma C, si è usato un approccio semplificato (una passata), infatti si sintetizza l’output, si traduce il programma in input ad “adac” direttamente durante l’analisi sintattica, senza costruire un albero sintattico astratto. Infatti la traduzione viene effettuata usando la lista usata per l’analisi semantica contenente tutti quei token e quelle informazioni aggiuntive che caratterizzano la regola di produzione da tradurre. La traduzione è effettuata con delle funzioni dichiarate nella libreria “semtrad.h”.

5. La traduzione del programma nel linguaggio sorgente (ADA 95) al programma nel linguaggio target (C) crea un file di output con lo stesso nome del file di input, ma con estensione diversa. Infatti supponendo di fornire in input ad “adac” il file di nome “*calcolaMedia.adb*” (.adb estensione di un file ADA) il compilatore fornisce in uscita il file “*calcolaMedia.c*” (.c estensione di un file C), contenente la traduzione in C del file in input scritto in linguaggio ADA 95. Se nel file in input ad “adac” sono definite delle funzioni o delle procedure, queste sono definite nella parte dichiarativa della procedura fondamentale del programma ADA in input. Le varie funzioni o procedure che vengono definite nella procedura fondamentale del programma in input ad “adac”, sono tradotte in un file con nome “*nomeFunzione.c*” o “*nomeProcedura.c*” mentre la procedura principale è tradotta nel file “*main.c*”.

Il file in C fornito in output da “adac” avrà nelle righe precedenti l’istruzione “*void main()*”, le funzioni o procedure tradotte definite nel programma ADA in input al compilatore e contenute nei file “*nomeFunzione.c*” e “*nomeProcedura.c*”.

Poi a partire dall’istruzione *void main()* ci sarà la traduzione della procedura principale del programma in input ad “adac”, ovvero sarà copiato il contenuto del file “*main.c*”.

Se non sono state definite delle funzioni o delle procedure nella procedura principale del programma in input ad “adac” allora nel file in output di “adac” sarà copiato solo il contenuto del file “*main.c*”.

Nella sezione “*Casi di test ed errori*” sono descritti alcuni test effettuati sul compilatore “adac” con alcuni file di test e riportando gli errori che si verificano.

Infine nella sezione “*Bibliografia e Sitografia*” sono indicati i riferimenti usati per la realizzazione del progetto.

Limitazioni

Il compilatore “*adac*” implementato ha delle limitazioni in quanto la realizzazione di un compilatore per l’intero linguaggio ADA 95 e la traduzione di un programma ADA 95 in C richiederebbe un lavoro più complesso con un ampio gruppo di programmatori.

Dato che il progetto realizzato è un progetto a livello didattico non si è realizzato un completo compilatore per l’ADA 95, ma un compilatore che effettua analisi lessicale, sintattica e semantica e una traduzione di una parte del linguaggio ADA 95 cercando di usare la maggior parte dei costrutti e delle istruzioni possibili.

Le limitazioni previste nel programma ADA in input ad “*adac*” sono le seguenti:

- i tipi usati sono “*integer*”, “*float*” e “*character*”;
- non è previsto l’uso di stringhe, quindi della keyword “*string*” ma definiamo vettori di “*character*”, così come definiamo vettori di “*integer*” e di “*float*”;
- definiamo nuovi tipi di dati con il comando “*type*” che possono essere solo “*array*”, quindi definiamo vettori di “*integer*”, di “*float*” e di “*character*”;
- è possibile definire delle funzioni o procedure solo all’interno della procedura principale del programma in ADA 95 in input ad “*adac*”, quindi non possiamo avere delle funzioni o procedure all’interno di altre funzioni o procedure, non sono possibili sottofunzioni all’interno di altre sottofunzioni;
- nelle procedure definite i parametri sono solo di modo “*in*” così come nelle function, nelle function in ADA 95 i parametri sono sempre di modo “*in*” mentre nelle procedure i parametri possono essere di modo: “*in*”, “*out*” e “*in out*”.

Quando un parametro è di modo “*in*” può essere solo letto (il suo valore non può essere modificato), quando un parametro è di modo “*out*” può essere scritto (gli si può assegnare un valore, inizialmente quando è passato alla funzione non ha valore) infine un parametro “*in out*” è un parametro che può essere letto e scritto;

- le funzioni e le procedure che sono senza parametri sono richiamate dalla procedura principale del programma sorgente di “adac” nel seguente modo:

variabile:=nomeFunzione() oppure *nomeProcedura()*

mentre in ADA 95 queste funzioni e procedure senza parametri sono richiamate senza usare le parentesi tonde:

variabile:=nomeFunzione oppure *nomeProcedura*

- nei cicli **for** in cui sono modificati tutti gli elementi dei vettori si hanno delle incoerenze nella traduzione in C. Infatti supponendo di avere vettori con range di indici da 1 a 4, la variabile contatore del ciclo for per poter modificare tutti i valori va da 1 a 4, l’istruzione “*for i in range 1..4*” in C sarà tradotta in “*for(i=1;i<=4;i++)*”.

In C i primi quattro elementi di un vettore hanno indici da 0 a 3.

Nel ciclo **for** in “adac” non si controlla che ci sia un’istruzione che modifica gli elementi di un vettore. Questo controllo non è stato implementato in “adac” poiché nel ciclo for si potrebbero avere anche delle operazioni non associate a dei vettori per cui quel range di valore può essere accettabile, quindi in questo caso si ha un errore a run time, un errore logico nel file in output ad “adac”.

Per non avere incoerenze nella traduzione in C bisognerebbe definire vettori con range di indici da 0 a n-1 (il vettore definito avrà dimensione n) nel file ADA in input ad “adac”;

- infine nelle istruzioni “*if*” e “*while*” non sono effettuati dei controlli semantici sulle condizioni dei costrutti.

Inoltre un’altra limitazione è che nel compilatore implementato non sono stati gestiti gli errori lessicali, ma solo errori sintattici e semantici.

Lo script di compilazione

Una volta installati i tool **Flex** e **Bison** su *OpenSuse 11.3*, è stato necessario per la realizzazione del compilatore “adac” creare il file “*compile.sh*” che contiene le seguenti direttive:

- **bison -k -d parser.y**

crea il file *parser.tab.c* a partire da *parser.y* (file usato per l’analisi sintattica) con i seguenti parametri:

–**k**: include una tabella dei nomi dei token

–**d**: produce un file header, indispensabile per il funzionamento combinato con l’analizzatore lessicale

- **gcc -c parser.tab.c**

genera il solo codice oggetto corrispondente a *parser.tab.c*

- **flex scanner.l**

crea il file *lex.yy.c* a partire da *scanner.l* (file usato per l’analisi lessicale)

- **gcc -c lex.yy.c**

genera il solo codice oggetto corrispondente a *lex.yy.c*

- **gcc parser.tab.o lex.yy.o -o adac**

genera l’*eseguibile adac* a partire dai due codici oggetto *parser.tab.o* e *lex.yy.o*, *adac* è il nome del compilatore ADA 95 implementato che esegue la traduzione in C.

Analisi lessicale

L'**analisi lessicale** in un compilatore permette di identificare nel programma in input le parole chiave, le variabili, le costanti, le funzioni, le procedure, etc.

L'analisi lessicale consente l'identificazione dei vari elementi lessicali del programma in input al compilatore poichè prevede quelle regole per la costruzione dei nomi a cui è attribuito un certo significato nel programma da compilare.

Il compito fondamentale di un analizzatore lessicale è rappresentato dalle seguenti fasi:

- data una sequenza di **caratteri** (*stringa*) di un alfabeto, l'analizzatore lessicale deve verificare se la sequenza può essere decomposta in una sequenza di **lessemi** (*parole*), tale che ogni lessema appartiene al lessico del linguaggio sorgente (ADA 95);
- in caso positivo, lo scanner deve restituire in uscita la sequenza di token per ciascun lessema del programma sorgente;
- in caso negativo deve restituire un errore lessicale (nel compilatore "adac" implementato non sono stati previsti errori lessicali);

Non esiste alcuna necessità di separare l'analisi lessicale dall'analisi sintattica, poichè per la realizzazione di un compilatore queste due fasi sono molto legate tra loro.

Quando lo scanner e il parser cooperano, un analizzatore lessicale non ritorna una lista di token, ma ritorna un token quando il parser lo richiede.

Poichè lo scanner è la parte del compilatore che legge il "*sorgente*", esso può svolgere altri compiti oltre a quello d'identificare dei lessemi e quindi dei token da fornire al parser:

- eliminare gli spazi bianchi e i commenti (implementato nel file "*scanner.l*");
- inserire i simboli nella tabella dei simboli;
- cancellare/inserire/sostituire caratteri nell'input;
- numerare le linee di codice, così da associare ad un eventuale messaggio di errore il numero di linea di codice corrispondente (implementato nel file "*scanner.l*")

L'analizzatore lessicale riconosce come token la stringa più lunga possibile.

I delimitatori/spaziatori sono i caratteri sui quali una stringa cessa di rappresentare un

simbolo (“*confini*” dei token).

Per l’implementazione dell’analizzatore lessicale e quindi per l’analisi lessicale si è usato il tool **Flex**, un tool open source.

Flex lavora riconoscendo una serie di token (pattern) descritti attraverso una serie di espressioni regolari (rules) eventualmente corredate dal codice C da eseguire in corrispondenza del pattern riconosciuto.

L’output di Flex è un sorgente C chiamato di default “*lex.yy.c*”, il quale definisce la routine *yylex()* che esegue l’analisi lessicale del file in input al file “*scanner.l*” (“*scanner.l*” è il file usato per la realizzazione dell’analizzatore lessicale di “*adac*”). L’input di Flex (*file .l*) si compone di tre sezioni separate da una riga di codice contenente il simbolo “%%”.

- La **Sezione 1** (che può anche essere vuota) contiene:
 - racchiuse tra i caratteri `%{` e `%}`, le **#include** di libreria, le definizioni di costanti e/o macro personalizzate del programma C che si vuole realizzare. Questa parte di testo verrà letteralmente copiata nel programma C generato. Quando lo scanner viene usato in combinazione con il parser, va inserita la **#include “parser.tab.h”** (poichè in “*adac*” il file “*parser.y*” esegue l’analisi sintattica), che è il file generato dal parser e che contiene la definizione dei token multi-carattere ai fini dell’analisi sintattica;
 - le definizioni di base usate nella seconda sezione per descrivere una espressione regolare.

La **Sezione 1** contiene le dichiarazioni facoltative di nomi che identificano pattern predefiniti (ad esempio un digit può essere identificato dall’espressione regolare `[0-9]`).

- La **Sezione 2** contiene la definizione dei pattern con le relative azioni da intraprendere, sotto forma di coppie **pattern-azione**. Le azioni devono iniziare sulla stessa riga in cui termina l’espressione regolare e sono separate tramite spazi o tabulazioni. Questa sezione contiene una serie di regole “*condizione azione*”, dove la condizione è identificata mediante un’espressione regolare e l’azione è implementata nel nostro caso

con un `return NOME_TOKEN;` dove `NOME_TOKEN` è una macro definita automaticamente dai tool `FLEX` e `BISON`.

- La **Sezione 3** (che può anche essere vuota) contiene le procedure di supporto di cui il programmatore intende servirsi per associarle alle azioni indicate nella seconda sezione: se è vuota, il separatore `%%` viene omissso.

Questa è la sezione conclusiva dove il programmatore ha la possibilità di implementare logiche differenti da quelle di default per quel che riguarda le azioni compiute dallo scanner.

Invece di produrre il semplice eseguibile dello scanner, la routine **yylex()** viene richiamata a runtime dall'analizzatore sintattico implementato con il file *"parser.y"*, generato mediante l'ausilio del tool **Bison**, versione open source del più celebre **Yacc** (*Yet Another Compiler of Compilers*).

L'implementazione dell'analizzatore lessicale è stata effettuata attraverso il file *"scanner.l"*, le cui caratteristiche vengono descritte di seguito e sono anche specificate nel file stesso, riportato di seguito, dai commenti evidenziati in colore giallo.

L'analizzatore lessicale esegue la scansione del file sorgente individuando i vari token del programma in input ad *"adac"*, questi vengono passati di volta in volta al parser che cerca di riconoscere una regola di derivazione della grammatica implementata.

scanner.l

```
%{
#include "parser.tab.h"/* Si è incluso il file "parser.tab.h" al fine di
permettere il funzionamento combinato di Flex e Bison*/

#include <stdio.h>
#include <string.h>
int numeroriga=1;/*numeroriga serve per contare le righe del programma
sorgente in modo da poter indicare la riga in cui si verifica un errore
sintattico o semantico all'utente*/
}%

/*option noyywrap serve per disabilitare la funzione yywrap(),funzione che
viene chiamata quando lo stream di input raggiunge l'EOF (End Of File)
tipo_token indica il tipo di una variabile,di un array,di una costante e il
tipo restituito da una function
interi serve per definire gli interi positivi (senza segno +)
interi_sign serve per definire gli interi negativi (con segno - e racchiusi
tra le parentesi tonde)
floating serve per definire i floating number positivi (senza segno +)
floating_sign serve per definire i floating number negativi (con segno - e
racchiusi tra le parentesi tonde)
s_char serve per definire i singoli caratteri
identificatore ci permette di rappresentare gli identificatori usati in ADA
95
oprel indica una operazione relazionale*/
%option noyywrap
tipo_token integer|INTEGER|float|FLOAT|character|CHARACTER|string|STRING
interi [0-9]+
interi_sign ("(-")[0-9]+(")")
floating ([0-9]+"."[0-9]+)
floating_sign ("(-")([0-9]+"."[0-9]+)(")")
s_char "'\".\""
identificatore [a-zA-Z]([a-zA-Z]*[0-9]*_?)*
oprel ("="| ">"| "<"| "/"="| ">="| "<=")

%%

" " ;
[ \t]+ ;
"--"(.)*"\n" {numeroriga++;}
":=" {return(ASSIGN);}
"constant"|"CONSTANT" {return(CONSTANT);}
"procedure"|"PROCEDURE" {return(PROCEDURE);}
"function"|"FUNCTION" {return(FUNCTION);}
"return"|"RETURN" {return(RETURN);}
"end"|"END" {return(END);}
"begin"|"BEGIN" {return(BG);}
"is"|"IS" {return(IS);}
"type"|"TYPE" {return(TP);}
"array"|"ARRAY" {return(ARRAY);}
"of"|"OF" {return(OF);}
```

```

"if"|"IF"           {return(IF);}
"else"|"ELSE"       {return(ELSE);}
"then"|"THEN"       {return(THEN);}
"and"|"AND"         {return(AND);}
"or"|"OR"           {return(OR);}
"not"|"NOT"         {return(NOT);}
"while"|"WHILE"     {return(WHILE);}
"for"|"FOR"         {return(FOR);}
"in"|"IN"           {return(IN);}
"out"|"OUT"         {return(OUT);}
"loop"|"LOOP"       {return(LOOP);}
"with"|"WITH"       {return(WITH);}
"Ada\Text_IO"       {return(TEXT);}
"Ada\Integer_Text_IO" {return(TEXT_INTEGER);}
"Ada\Float_Text_IO" {return(TEXT_FLOAT);}
"Ada\Text_IO\Put"   {return(PUT_CHARACTER);}
"Ada\Text_IO\Get"   {return(GET_CHARACTER);}
"Ada\Integer_Text_IO\Put" {return(PUT_INTEGER);}
"Ada\Integer_Text_IO\Get" {return(GET_INTEGER);}
"Ada\Float_Text_IO\Put" {return(PUT_FLOAT);}
"Ada\Float_Text_IO\Get" {return(GET_FLOAT);}
"+"                {return(PIU);}
"-"                {return(MENO);}
"*"                {return(PER);}
"/"                {return(DIVISO);}
{interi}           {
/*il testo riconosciuto dallo scanner viene accumulato nella variabile
yytext, yylval indica il valore del token corrente nello stack*/
    yylval.intval=atoi(yytext);
    return(INT_NUMBER);
}
{interi_sign}      {
//preleviamo il valore intero negativo dall'espressione: (intero negativo)
//eliminiamo le parentesi tonde e recuperiamo il valore dell'intero
//negativo da yytext
    char *s=(char *)malloc((strlen(yytext)-1)*sizeof(char));
    int j=0;
    int i;
    for(i=1;i<strlen(yytext)-1;i++)
    {
        s[j]=yytext[i];
        j++;
    }
    s[strlen(s)]='\0';
    yylval.intval=atoi(s);
    return(INT_SIGN_NUMBER);
}
{floating_sign}    {
//preleviamo il valore float negativo dall'espressione: (float negativo)
//eliminiamo le parentesi tonde e recuperiamo il valore del float negativo
//da yytext
    char *s=(char *)malloc((strlen(yytext)-1)*sizeof(char));
    int j=0;
    int i;

```

```

        for (i=1; i<strlen(yytext)-1; i++)
        {
            s[j]=yytext[i];
            j++;
        }
        s[strlen(s)]='\0';
        yylval.float_value=atof(s);
        return(FLOAT_SIGN_NUMBER);
    }
{floating}
{
    yylval.float_value=atof(yytext);
    return(FLOAT_NUMBER);
}
{s_char}
{
    yylval.id = (char *)strdup(yytext);
    return S_CHAR;
}
{tipo_token}
{
    yylval.id = (char *)strdup(yytext);
    return(TIPO);
}
{identificatore}
{
    yylval.id = (char *)strdup(yytext);
    return(ID);
}

{oprel}
{
    yylval.id = (char *)strdup(yytext);
    return OPREL;
}
";"
{return(PUNTOEVIRGOLA);}
":"
{return(DUEPUNTI);}
","
{return(VIRGOLA);}
"("
{return(TONDAAPERTA);}
")"
{return(TONDACHIUSA);}
"\.\."
{return(PUNTOPUNTO);}
"\n"
{numeroriga++;}
"."
{return(yytext[0]);}

%%

```

I return consentono l'invio del simbolo riconosciuto al corrispondente token definito nel parser. Per questioni di semplicità e per le limitazioni previste si è usato un minor numero di pattern rispetto ai pattern che supporta ADA 95.

Analisi sintattica

L'**analisi sintattica** di un compilatore è l'insieme delle regole di produzione che caratterizzano una frase del linguaggio sorgente, nel compilatore "adac" è l'insieme delle regole di produzione che caratterizzano una frase del linguaggio di programmazione ADA 95.

La sintassi di un compilatore indica la forma del programma in input al compilatore.

Il problema di base dell'analisi sintattica è il seguente: data una sequenza di token e una specifica della sintassi, **stabilire se la sequenza è una frase ammessa dalla sintassi**.

La sintassi è in genere specificata in termini di un linguaggio non contestuale (*context free*), in cui ogni token costituisce un simbolo atomico del linguaggio.

Data una sequenza *s* di token, l'analizzatore sintattico verifica se la sequenza appartiene al linguaggio generato dalla grammatica **G**.

A questo scopo, l'analizzatore sintattico cerca di costruire l'albero sintattico di *s*:

- in caso positivo, restituisce in uscita l'albero sintattico per la sequenza di input nella grammatica **G**;
- in caso negativo, restituisce un errore (errore sintattico).

Un **parser** deve riconoscere la struttura di una stringa di ingresso, che è fornita in termini di regole di produzione di una **Context Free Grammar (CFG)** o di una **BNF**.

Un parser è una macchina "*astratta*" che raggruppa input in accordo con regole grammaticali.

La **sintassi** è costituita da un insieme di regole che definiscono le frasi formalmente corrette e allo stesso tempo permettono di assegnare ad esse una struttura (albero sintattico) che ne indica la decomposizione nei costituenti immediati.

Ad esempio, la struttura di una frase (ovvero di un programma) di un linguaggio di programmazione ha come costituenti le parti dichiarative e quelle esecutive, infatti in un programma ADA 95 ci sono in genere parti dichiarative e parti esecutive.

Le parti dichiarative definiscono i dati usati dal programma.

Le parti esecutive si articolano nelle istruzioni, che possono essere di vari tipi: assegnamenti, istruzioni condizionali, frasi di lettura, etc.

Quello che si può ottenere con una sintassi libera è limitato e spesso insufficiente: la sintassi non basta a definire le frasi corrette del linguaggio di programmazione, perchè una stringa sintatticamente corretta non è detto che lo sia in assoluto.

Infatti essa potrebbe violare altre condizioni non esprimibili nel modello noncontestuale.

Ad esempio è noto che la sintassi non può esprimere le seguenti condizioni:

- in un programma ogni identificatore di variabile deve comparire in una dichiarazione;
- il tipo del parametro attuale di una funzione o procedura deve essere compatibile con quello del parametro formale corrispondente.

Condizioni, quelle precedenti, che sono giustificate dalla necessità di **attribuire un significato** agli elementi sintattici e di **stabilire delle regole semantiche e dei controlli semantici** per gli elementi sintattici.

Un altro limite del modello sintattico libero riguarda la traduzione.

Se si guardano le traduzioni definibili con i soli metodi sintattici, esse sono decisamente povere, inadeguate ai problemi reali (compilazione di un linguaggio programmatico nel linguaggio macchina), ma utile nel caso di semplici traduzioni e di traduzioni da un linguaggio di alto livello ad un altro di alto livello come nel caso della traduzione implementata in “adac” ADA95 - C.

Da queste critiche sarebbe sbagliato concludere che i metodi sintattici sono inutili: al contrario essi sono indispensabili come supporto (concettuale e progettuale) su cui poggiare i più completi strumenti della semantica.

La realizzazione dell’analizzatore sintattico è stata effettuata mediante l’uso di un tool open source: **Bison**.

Un file di input di Bison che ha estensione “.y” è formato da tre sezioni:

- il **prologo** (o parte dedicata alle dichiarazioni) che è la sezione preliminare che contiene la definizione delle macro, la dichiarazione di funzioni e variabili che saranno richiamate in seguito (in maniera analoga a quanto accade per un programma C).

Il prologo può comprendere:

- una “*black box*” di definizioni ausiliarie, costituita da dichiarazioni C racchiuse tra i delimitatori **%{** e **%}** ad esempio:

%{ #include e altre macro e direttive C, costanti, variabili %}

- una “white box”, costituita da nomi di simboli terminali e non terminali, regole di precedenza/associatività tra simboli, etc.

In questa “white box” ci sono le dichiarazioni che definiscono i simboli terminali e non terminali associati alla grammatica e che, nell’uso combinato con Flex, devono contenere un’istruzione del tipo: `%token NOME_TOKEN` per ogni valore di ritorno dello scanner.

- le **regole** (o parte dedicata alla descrizione delle strutture sintattiche della grammatica context-free considerata) che definisce le regole di derivazione tramite le quali il parser è in grado di generare l’albero sintattico.

Una regola è espressa nella forma:

`NOME_REGOLA : espr1 {codice1} | espr2 {codice2} ... | esprN {codiceN};`

- l’**epilogo** (o parte dedicata alle routine ausiliarie), che contiene ulteriore codice C che verrà copiato in coda al parser generato e che, generalmente, è utilizzata per ridefinire le funzioni che altrimenti sarebbero limitate a quelle di default.

Ogni sezione è delimitata da “%%”.

L’output di Bison per il file “*parser.y*” nel quale si è implementato l’analizzatore sintattico, è un sorgente C chiamato “*parser.tab.c*”, il quale definisce la routine **yyparse()** che esegue l’analisi sintattica. Nel momento in cui viene chiamata la funzione **yyparse()**, quest’ultima provvede alla chiamata di **yylex()** affinché sia possibile iniziare la lettura dell’input segmentato in token. Bison inserisce in una pila i token ricevuti (operazione di **shift**) e, nel frattempo, cerca di raggrupparli per ridurli (operazione di **reduce**) secondo una delle regole definite dalla grammatica.

Partendo dalle foglie dell’albero quindi, si cerca di ridurre progressivamente le regole riconosciute verso l’assioma seguendo ovviamente un approccio **bottom up**.

Bison inoltre prevede un **lookahead** di un token, pertanto, anziché ridurre con la prima regola possibile, cerca di esaminare il token successivo per decidere la migliore strategia di derivazione da effettuare. Nel caso in cui un token di lookahead non sia sufficiente a discriminare in maniera opportuna una o più alternative dell’albero sintattico, Bison genera un

warning chiamato “*shift/reduce*” che non implica necessariamente la presenza di ambiguità nella grammatica, in quanto può essere generato anche da semplici ricorsioni che si rivelano necessarie durante la scrittura della grammatica stessa.

Un errore assolutamente da evitare, invece, è il conflitto “*reduce/reduce*” che indica una situazione per la quale vi sono effettivamente due o più strade possibili da seguire nell’albero per ridurre una stessa regola e che quindi identificano una situazione di forte ambiguità.

L’analizzatore sintattico del compilatore “adac” è stato implementato nel file “*parser.y*” le cui caratteristiche vengono descritte di seguito e sono anche specificate nel file stesso, riportato di seguito, dai commenti evidenziati in colore giallo.

Nell’analisi sintattica è gestita l’analisi semantica usando lo schema **SDT**, uno schema di traduzione diretta per l’analisi semantica.

Nell’analisi sintattica una volta che l’analizzatore sintattico ha riconosciuto una regola di derivazione essa la si verifica: se è una regola che genera errore sintattico allora viene visualizzato il messaggio di errore sintattico in output da “adac” e il compilatore termina la sua esecuzione, altrimenti se è stata riconosciuta una regola di derivazione non associata ad un errore sintattico si effettua l’analisi semantica della regola riconosciuta e sintatticamente corretta.

Nel file “*parser.y*” non sono stati gestiti gli errori sintattici per tutte le regole di produzione implementate e caratterizzanti la grammatica usata.

Inoltre per alcune regole di produzione non sono stati gestiti tutti i possibili errori sintattici.

Quando il parser riconosce la definizione di una variabile, di una costante, di una funzione, di una procedura, di un nuovo tipo di vettore, di un array o di una variabile contatore usata in un ciclo **for** viene richiamata la funzione **install** della libreria “*semtrad.h*” che consente l’inserimento delle strutture riconosciute nella **Symbol Table**.

La Symbol Table è una struttura dati usata da un compilatore per tener traccia del significato, della semantica e delle caratteristiche dei nomi dichiarati nel programma in input al compilatore (“adac” nel nostro caso).

Alcune delle caratteristiche che possono essere gestite sono le caratteristiche riguardanti il tipo del dato, lo scope (che indica quando il dato viene usato) o l’indirizzo in cui è memorizzato il dato.

La Symbol Table può essere realizzata in diversi modi:

- **Lista non ordinata:**
 - nel caso in cui si usa un piccolo set di variabili;
 - implementazione semplice, ma le performance non sono buone con un gran numero di variabili.
- **Lista ordinata:**
 - è usata una ricerca binaria;
 - l'inserimento e la cancellazione sono costosi dal punto di vista computazionale;
 - implementazione relativamente semplice.
- **Binary search tree:**
 - tempo per le operazioni (ricerca, inserimento o cancellazione) pari a $O(\log n)$;
 - implementazione difficile.
- **Hash table:**
 - usata in genere;
 - performance non buone se sfortunati, ovvero se ci sono molti sinonimi da inserire nella Symbol Table o questa è piena;
 - l'implementazione non è molto difficile.

Nel nostro caso si è realizzata una **Symbol Table** basata su una **Hash Table**, usando il **metodo della divisione**.

Una Hash Table è una struttura dati usata per mettere in corrispondenza una data **chiave** con un dato **valore** (*valore hash*) una posizione della Hash Table.

Una Hash Table è un array di dimensione m in cui in posizione i -esima inserisco un elemento con una chiave che corrisponde ad un valore ovvero alla posizione i -esima della Hash Table in cui inserisco questo elemento.

La corrispondenza tra chiave e valore hash è ricavata attraverso una funzione hash.

La chiave che si considera è il nome del token da inserire nella Symbol Table.

La funzione hash è una funzione non invertibile che trasforma un testo di lunghezza arbitraria in una stringa di lunghezza fissa, in un numero che va da 0 ad $m-1$.

Due chiavi distinte possono produrre lo stesso valore hash e in questo caso si ha una collisione e le due chiavi sono dei sinonimi. Nel caso in cui vi è una collisione la seconda chiave sinonima da inserire nella Hash Table è inserita nella posizione della Hash Table calcolata con la funzione hash e sarà legata alla prima chiave sinonima inserita mediante una lista.

La funzione hash usata è stata realizzata mediante il metodo della divisione ed è quella funzione che calcola il resto della divisione tra la somma dei valori *ascii* dei caratteri che costituiscono la chiave diviso *m*.

I campi degli elementi della Symbol Table usati sono i seguenti:

- **int valh:** indice della Symbol Table in cui si inserisce il token riconosciuto dallo scanner;
- **char *nameToken:** nome del token che si inserisce nella Symbol Table;
- **int dim_vet:** dimensione del vettore che si inserisce nella Symbol Table altrimenti questo campo ha valore 0;
- **int esI:** estremo inferiore del vettore che si inserisce nella Symbol Table altrimenti questo campo ha valore 0;
- **int esS:** estremo superiore del vettore che si inserisce nella Symbol Table altrimenti questo campo ha valore 0;
- **struct el_ST *next:** puntatore ad un elemento della Symbol Table, serve per gestire le liste che si creano nel caso in cui si verificano delle collisioni;
- **tipo t:** tipo di token o del simbolo inserito nella Symbol Table, il simbolo inserito può essere variabile, constant, type, array, contatore, funzione;
- **char *tv:** tipo di variabile, costante, array o tipo restituito da una funzione in generale questo campo assume valori: "integer", "float" o "character";
- **char *funz:** nome della funzione o della procedura in cui è definito un token;
- **int pf:** se settata a 1 (parametro di modo "in") indica che la variabile a cui si riferisce è un parametro di una funzione o procedura, altrimenti se è settata a 0 la variabile che si considera è una semplice variabile usata in una funzione o procedura;
- **int nparam:** indica il numero di parametri di una funzione o di una procedura;

- **int nump**: indica l'ordine con cui dichiaro i parametri di una funzione o di una procedura.

Nel file “*parser.y*” quando si riconosce la definizione o la dichiarazione di una variabile, di una costante, di un nuovo tipo di vettore, di un array, di una variabile contatore usata nel ciclo for, di una funzione o di una procedura viene eseguita la funzione **install** della libreria “*semtrad.h*” che permette di inserire uno degli elementi precedentemente indicati nella Symbol Table.

parser.y

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "semtrad.h" //libreria per l'analisi semantica e la traduzione
extern char * yytext; /*con la keyword extern Bison permette di accedere
alle variabili esterne, alle variabili di scanner.1 infatti yytext è una
variabile di Flex*/
int yylex();
void yyerror(char *);
extern numeroriga;
char buff[256]; /*buff array di caratteri contenente i valori interi o float
letti dallo scanner*/
el ST ST[MAXDIMST]; /*ST è un array contenente MAXDIMST(128) struct di tipo
el ST, ST è la Symbol Table*/
testo *T; /*T è una lista di tipo testo, è la testa della lista usata per
effettuare l'analisi semantica e la traduzione delle generiche istruzioni
ADA 95 tranne che per le istruzioni riguardanti la firma delle funzioni e
delle procedure di ADA 95*/
testo *Tf=NULL; /*Tf è una lista di tipo testo, è la testa della lista usata
per effettuare l'analisi semantica e la traduzione della firma delle
funzioni e delle procedure di ADA 95*/
FILE *out; /*out è un puntatore al file in cui ci sarà la traduzione in C
del programma in input al compilatore adac, sarà uguale al valore di m o
di fz*/
FILE *m; /*m è un puntatore al file in cui ci sarà la traduzione in C della
procedura fondamentale del programma ADA 95 in input ad adac, punterà al
file "main.c"*/
FILE *fz; /*fz è un puntatore al file in cui ci sarà la traduzione in C
delle funzioni o procedure definite nel programma ADA 95 in input ad
adac, punterà a file con nome "nomeProcedura.c" o "nomeFunzione.c"*/
char *filename; /*filename è un array di caratteri contenente il nome del
file in output ovvero del file in cui c'è la traduzione in C del file in
input ad adac*/
```

```

int costante=0; /*costante variabile usata per vedere se nel programma in
input ad adac è stata definita una costante (costante=1) o no(costante=0)*/
int aggr=0; /*aggr variabile usata per capire se il parser ha identificato
una operazione di aggregazione di un vettore. 0 operazione di aggregazione
non riconosciuta 1 operazione di aggregazione riconosciuta*/
int arr=0; /*la variabile arr serve per vedere se nel programma in input ad
adac è stato definito un array(arr=1) o no(arr=0)*/
int dimvet=0; /*dimvet è una variabile che contiene la dimensione di un
array altrimenti è uguale a zero, è usata nell'inserimento di un nuovo
simbolo nella Symbol Table*/
int estremoI=0; /*estremoI è una variabile che contiene l'estremo inferiore
del range di indici di un array altrimenti è uguale a zero.È una variabile
usata nell'inserimento di un nuovo simbolo nella Symbol Table*/
int estremoS=0; /*estremoS è una variabile che contiene l'estremo superiore
del range di indici di un array altrimenti è uguale a zero.È una variabile
usata nell'inserimento di un nuovo simbolo nella Symbol Table*/
int text=0; /*text è una variabile che serve per vedere se è stata inclusa
la libreria Ada.Text_IO(text=1) o no(text=0)*/
int text_integer=0; /*text_integer è una variabile che serve per vedere se è
stata inclusa la libreria Ada.Integer_Text_IO(text_integer=1)
o no(text_integer=0)*/
int text_float=0; /*text_float è una variabile che serve per vedere se è
stata inclusa la libreria Ada.Float_Text_IO(text_float=1)
o no(text_float=0)*/
int r=0; /*la variabile r serve per verificare se si sta facendo l'analisi
semantica o la traduzione della firma di una funzione(r=1) o l'analisi
semantica o la traduzione di un'altra istruzione(r=0) del programma in
input ad adac*/
char *nomeF; /*nomeF è un array di caratteri contenente il nome della
procedura o funzione che viene definita in ADA 95*/
char *mn; /*mn è un array di caratteri contenente il nome della procedura
principale del programma ADA 95 in input ad adac*/
char *nf=NULL; /*nf array di caratteri, contiene il nome del file in cui
effettuare la traduzione in C delle funzioni o procedure definite nel
programma in input ad adac*/
int pr=0; /*pr è una variabile che serve per vedere se si è in una
funzione(pr=0) o in una procedura(pr=1) nell'analisi del file in input ad
"adac" e se la keyword return è rispettivamente ammessa e non ammessa*/
char *parpg; /*parpg è un array di caratteri contenente le stringhe
"integer","float" o "character" per capire se si sta eseguendo
una funzione Put o Get rispettivamente delle librerie
Ada.Integer_Text_IO,Ada.Float_Text_IO o Ada.Text_IO*/
int paramf=0; /*paramf è una variabile settata a 1 se si stanno definendo i
parametri di una funzione o di una procedura altrimenti è settata a 0,
quando paramf è uguale a 1 serve per impostare il modo di un parametro di
una funzione o di una procedura a 1(parametro di modo in) è per contare i
parametri di una funzione o procedura che saranno contenuti nella variabile
"numpar"*/
int numpar=0; /*numpar è una variabile che contiene il numero di parametri
di una funzione o di una procedura*/
int paramlista=0; /*paramlista è una variabile che è settata a 1 se si
stanno definendo i parametri di una funzione o di una procedura

```

```

altrimenti è settata a 0, quando paramlista è uguale a 1 permette di
ricavare il numero d'ordine del parametro di una funzione o di una
procedura che sarà contenuto nella variabile "np"*/
int np=0; /*np indica il numero d'ordine del parametro di una funzione o di
una procedura*/
char *tiporet; /*tiporet è un array di caratteri contenente il tipo
restituito da una funzione definita nel programma in input ad adac*/
testo *nomifile=NULL; /*nomifile è un puntatore ad una lista testo
contenente il nome dei file in cui ci sono le traduzioni in C delle
funzioni e delle procedure definite nel programma in input ad adac*/
int sf=0; /*sf serve per settare paramf a 1 (sf=1)*/
int vet=0; /*la variabile vet serve per capire se si sta facendo una
operazione di assegnazione con un vettore a destra o a sinistra
nell'operazione di assegnazione (vet=1) oppure una chiamata a
funzione (vet=0) per poter effettuare l'analisi semantica e la traduzione
in C opportuna, dato che la chiamata a funzione o procedura e l'accesso ad
un elemento del vettore hanno la stessa sintassi*/
char *contat; /*è un array di caratteri che conterrà il nome della
variabile contatore da eliminare dalla Symbol Table, quando il ciclo for ha
cui è associata è terminato*/
/*La funzione inserisci_var è una funzione che permette di inserire un
nuovo simbolo nella Symbol Table.
La funzione inserisci_var ha un parametro: tvar che è un puntatore ad un
array di caratteri contenente il nome del nuovo simbolo da inserire nella
Symbol Table.*/
void inserisci_var(char *tvar)
{
    testo *punt=T;
    while(strcmp(punt->tes,":")!=0)
    {
        if(strcmp(punt->tes,",")!=0)
        {
            if(paramf!=0)
                numpar=numpar+1;
            if(costante)
                install(ST,punt->tes,2,tvar,0,0,0,nomeF,paramf,np,numpar,numeroriga);
            else
            {
                if(arr)
                    install(ST,punt->tes,4,tvar,0,0,0,nomeF,paramf,np,numpar,numeroriga);
                else
                    install(ST,punt->tes,1,tvar,0,0,0,nomeF,paramf,np,numpar,numeroriga);
            }
        }
        punt=punt->next;
    }
}
/*La funzione get_estremi è una funzione che permette di individuare gli
estremi del range di indici ammessi da un vettore.*/
void get_estremi()
{
    testo *punt=T;
    estremoI=atoi(punt->next->next->tes);
    estremoS=atoi(punt->next->next->next->next->tes);
}

```



```

}
/*La funzione get_dim_vet permette di calcolare la dimensione di un array
in base agli estremi del range di indici che caratterizzano l'array.
La funzione get_dim_vet ha un parametro: val che contiene il valore
dell'estremo superiore del range di indici ammessi dall'array.*/
void get_dim_vet(int val)
{
    int trov=0;
    testo *punt=T;
    while (punt!=NULL&&!trov)
    {
        if(strcmp(punt->tes,"array")==0)
            trov=1;
        punt=punt->next;
    }
    if(trov)
    {
        testo *ei=punt->next;
        int trovp=0;
        punt=punt->next;
        while (punt!=NULL&&!trovp)
        {
            if(strcmp(punt->tes,"..")==0)
                trovp=1;
            else
                punt=punt->next;
        }
        if(trovp&&punt->next==NULL)
        {
            estremoI=atoi(ei->tes);
            estremoS=val;
            if(estremoI>0&&estremoS>0)
                dimvet=estremoS;
            else
            {
                if(estremoI>=0&&estremoS>0)
                    dimvet=estremoS+1;
                else
                {
                    if(estremoI<0&&estremoS<0)
                        dimvet=estremoI;
                    else
                    {
                        if(estremoI<0&&estremoS<=0)
                            dimvet=estremoI+1;
                        else
                        {
                            if(estremoI<0&&estremoS>0)
                                dimvet=labs(estremoI)+estremoS+1;
                        }
                    }
                }
            }
        }
    }
}
/*La funzione crea_file serve per creare i puntatori ai file che
conterranno la traduzione in C delle funzioni o delle procedure definite
nel programma in input ad adac.
I puntatori ai file creati, punteranno a file con nome "nomeProcedura.c" o
"nomeFunzione.c"*/
void crea_file()

```

```

{
    char *nome;
    int crea=1;
    int i=0;
    int n;
    if(nf!=NULL)
    {
        n=strlen(nf);
        n=n-2;
        nome=(char *)malloc((n+1)*sizeof(char));
        strncpy(nome,nf,n);
        nome[n]='\0';
        if(strcmp(nome,nomeF)==0)
            crea=0;
    }
    if(crea)
    {
        n=strlen(nomeF);
        nf=(char *)malloc((n+3)*sizeof(char));
        strcpy(nf,nomeF);
        strcat(nf, ".c");
        nf[n+2]='\0';
        put_testo(&nomifile,nf);
        fz=fopen(nf,"w");
    }
}
%}
/*l'opzione %union di Bison consente di specificare una varietà di tipi di
dati che sono usati dalla variabile yylval in Flex per identificare: gli
interi, i float, i character, il tipo ("integer", "float" e "character"),
gli identificatori e gli operatori relazionali
("=", ">", "<", "/=", ">=", "<=") */
%union{
    int intval;
    char *id;
    float float_value;
}

%start input //input è l'assioma della grammatica usata per "adac"

%expect 67/*permette di ignorare i warning relativi a conflitti del tipo
shift/reduce, dato che sono conflitti che non sempre portano ad errori
nelle grammatiche e che potrebbero essere risolti se si disponesse di un
maggior numero di lookahead tokens*/
%error-verbose/*consente la visualizzazione degli errori direttamente sullo
stream di output quando viene richiamata la funzione yyerror()*/
%locations/*permette di ottenere maggiori informazioni circa l'errore
riscontrato, ad esempio consente di far stampare a video anche il token che
ci si aspetterebbe di ricevere */

%token WITH TEXT TEXT_INTEGER TEXT_FLOAT
%token PUNTOEVIRGOLA DUEPUNTI VIRGOLA TONDAAPERTA TONDACHIUSA PUNTOPUNTO
%token PROCEDURE FUNCTION RETURN IS END BG ARRAY OF OUT
%token ASSIGN CONSTANT TP

```

```

%token PIU MENO DIVISO PER
%token IF ELSE THEN AND OR NOT
%token FOR WHILE IN LOOP
%token PUT_INTEGER PUT_CHARACTER PUT_FLOAT
%token GET_INTEGER GET_CHARACTER GET_FLOAT

%token <intval> INT_NUMBER INT_SIGN_NUMBER
%token <float_value> FLOAT_NUMBER FLOAT_SIGN_NUMBER
%token <id> ID TIPO S_CHAR OPREL

%left PIU MENO DIVISO PER

%%

```

```

//input regola start
input: |input inizio
{
    printf("\n\nProgramma ADA riconosciuto CORRETTAMENTE!!!\n\n");
}
;
//regole per poter inserire e gestire nel programma ADA 95 in input ad
//adac le librerie Ada.Integer_Text_IO,Ada.Float_Text_IO e Ada.Text_IO
inizio: with_stmts startprocedure |startprocedure
;
with_stmt: WITH seq_lib PUNTOEVIRGOLA
;
with_stmts: with_stmts with_stmt |with_stmt
;
seq_lib: seq_lib VIRGOLA lib|lib
;
lib: TEXT
{
    if(text==1)
    {
        printf("ERRORE SINTATTICO: errore alla riga numero %d libreria
            \"Ada.Text_IO\" già inclusa\n",numoriga);
        exit(0);
    }
    else
        text=1;
}
|TEXT_INTEGER
{
    if(text_integer==1)
    {
        printf("ERRORE SINTATTICO: errore alla riga numero %d libreria
            \"Ada.Integer_Text_IO\" già inclusa\n",numoriga);
        exit(0);
    }
    else
        text_integer=1;
}
|TEXT_FLOAT
{

```

```

        if(text_float==1)
        {
            printf("ERRORE SINTATTICO: errore alla riga numero %d libreria
                \"Ada.Float_Text_IO\" già inclusa\n",numeroriga);
            exit(0);
        }
        else
            text_float=1;
    }
;
//regole per l'identificazione della procedura fondamentale del programma
//ADA 95 in input ad adac
pro_id:PROCEDURE ID
{
    pr=1;
    fprintf(out,"void main()");nomeF=(char *)strdup($2);mn=(char *)strdup($2);
}
;
pro_id_is:pro_id IS{fprintf(out,"\\n{\\n\\n//parte dichiarativa\\n");}
;
pro_id_is_decl_part:pro_id_is decl_part
;
pro_id_is_decl_part_bg:pro_id_is_decl_part BG
{fprintf(out,"\\n\\n//parte esecutiva\\n");}
;
pro_id_is_decl_part_bg_stmt_list:pro_id_is_decl_part_bg stmt_list
;
pro_id_is_decl_part_bg_stmt_list_end:pro_id_is_decl_part_bg_stmt_list
END{fprintf(out,"\\n}");}
;
pro_id_is_decl_part_bg_stmt_list_end_id:pro_id_is_decl_part_bg_stmt_list_end ID
{
    if(strcmp(nomeF,$2)!=0)
    {
        printf("ERRORE SEMANTICO: errore alla riga numero %d NOME FUNZIONE non
            corretto\\n",numeroriga);
        exit(0);
    }
}
;
startprocedure :pro_id_is_decl_part_bg_stmt_list_end_id
PUNTOEVIRGOLA{pr=0;}
    |PROCEDURE IS
    {
        printf("ERRORE SINTATTICO: errore alla riga numero %d NOME
            DELLA PROCEDURA mancante\\n",numeroriga);
        exit(0);
    }
    |ID IS
    {
        printf("errore sintattico: errore alla riga numero %d keyword
            PROCEDURE mancante\\n",numeroriga);
        exit(0);
    }

```

```

    }
    |pro_id decl_part
    {
        printf("errore sintattico: errore alla riga numero %d keyword
            IS mancante\n",numeroriga-1);
        exit(0);
    }
    |pro_id_is_decl_part stmt_list
    {
        if(iskeyword(yytext))
            printf("ERRORE SEMANTICO:errore alla riga %d nome \"%s\"
riservato ad una parola chiave del linguaggio ADA\n",numeroriga,yytext);
        printf("errore sintattico: errore alla riga numero %d keyword
            begin mancante\n",numeroriga-1);
        exit(0);
    }
    |pro_id_is_decl_part_bg_stmt_list ID
    {
        printf("errore sintattico: errore alla riga numero %d keyword
            END mancante\n",numeroriga);
        exit(0);
    }
    |pro_id_is_decl_part_bg_stmt_list_end
    {
        printf("errore sintattico: errore alla riga numero %d NOME
            PROCEDURA mancante\n",numeroriga);
        exit(0);
    }
    |pro_id_is_decl_part_bg_stmt_list_end_id
    {
        printf("errore sintattico: errore alla riga numero %d ;
            mancante\n",numeroriga);
        exit(0);
    }
    }
;
//regole per la definizione di nuove funzioni nella procedura fondamentale
//del programma ADA 95 in input ad adac
fun_id: FUNCTION ID
{
    put_testo(&Tf,"function");nomeF=(char*)strdup($2);put_testo(&Tf,$2);
    crea_file();out=fz;paramlista=1;sf=1;pr=0;
}
;
fun_id_return:fun_id RETURN
{
    put_testo(&Tf,"return");r=1;sf=0;
}
|fun_id{r=1;} r_tipo
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d keyword RETURN
        mancante\n",numeroriga);
    exit(0);
}
|fun_id_param RETURN{put_testo(&Tf,"return");r=1;sf=0;}

```

```

|fun_id_param{r=1;} r_tipo
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d keyword RETURN
        mancante\n",numeroriga);
    exit(0);
}
;
fun_id_param:fun_id TONDAAPERTA seq TONDACHIUSA
|fun_id seq TONDACHIUSA
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d \"(\n"
        mancante\n",numeroriga);
    exit(0);
}
|fun_id TONDAAPERTA seq
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d \")\n"
        mancante\n",numeroriga);
    exit(0);
}
;
fun_id_return_r_tipo:fun_id_return r_tipo
|fun_id_return IS
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d tipo da restituire
        mancante\n",numeroriga);
    exit(0);
}
;
fun_id_return_r_tipo_is:fun_id_return_r_tipo IS
{
    put_testo(&Tf,"is");
    r=0;
    analisi_trad(Tf,ST,out,aggr,vet,nomeF,numeroriga,mn);
    Tf=NULL;
    paramf=0;
    numpar=0;
    install(ST,nomeF,6,tiporet,0,0,0,mn,paramf,np,numpar,numeroriga);
    np=0;
    paramlista=0;
}
;
fun_id_return_decl_part:fun_id_return_r_tipo_is decl_part
;
fun_id_return_decl_part_bg:fun_id_return_decl_part BG
;
fun_id_return_decl_part_bg_stmt_list_f:fun_id_return_decl_part_bg stmt_list
;
fun_id_return_decl_part_bg_stmt_list_f_end:fun_id_return_decl_part_bg_stmt_
list_f END
;
fun_id_return_decl_part_bg_stmt_list_f_end_id:fun_id_return_decl_part_bg_st
mt_list_f_end ID
{

```

```

    if(strcmp(nomeF,$2)!=0)
    {
        printf("ERRORE SEMANTICO: errore alla riga numero %d NOME FUNZIONE non
            corretto\n",numeroriga);
        exit(0);
    }
}
;
fun:fun_id_return_decl_part_bg_stmt_list_f_end_id PUNTOEVIRGOLA
{
    pr=1;
    fprintf(out,"}");
    nomeF=(char *)strdup(mn);
    fclose(out);
    out=m;
}
|FUNCTION
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d NOME FUNZIONE
        mancante\n",numeroriga);
    exit(0);
}
|fun_id_return_r_tipo
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d keyword IS
        mancante\n",numeroriga-1);
    exit(0);
}

|fun_id_return_decl_part_stmt_list
{
    if(iskeyword(yytext))
        printf("ERRORE SEMANTICO:errore alla riga %d nome \"%s\" riservato ad
            una parola chiave del linguaggio ADA\n",numeroriga,yytext);
    printf("ERRORE SINTATTICO: errore alla riga numero %d keyword begin
        mancante\n",numeroriga);
    exit(0);
}
|fun_id_return_decl_part_bg_stmt_list ID
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d keyword END
        mancante\n",numeroriga);
    exit(0);
}
|fun_id_return_decl_part_bg_stmt_list_f_end PUNTOEVIRGOLA
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d NOME FUNZIONE
        mancante\n",numeroriga);
    exit(0);
}
|fun_id_return_decl_part_bg_stmt_list_f_end_id
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d ;
        mancante\n",numeroriga-1);
}

```

```

    exit(0);
}
;
//regole per la definizione di nuove procedure nella procedura fondamentale
//del programma ADA 95 in input ad adac
pr_id:PROCEDURE ID
{
    pr=1;
    put_testo(&Tf,"procedure");
    nomeF=(char*)strdup($2);
    put_testo(&Tf,$2);
    crea_file();
    out=fz;
    paramlista=1;
    sf=1;
}
;
pr_id_is:pr_id IS
{
    put_testo(&Tf,"is");
    analisi_trad(Tf,ST,out,aggr,vet,nomeF,numeroriga,mn);
    Tf=NULL;
    paramf=0;
    numpar=0;
    install(ST,nomeF,6,"procedure",0,0,0,mn,paramf,np,numpar,numeroriga);
    np=0;
    paramlista=0;
    sf=0;
}
|pr_id_param IS
{
    put_testo(&Tf,"is");
    analisi_trad(Tf,ST,out,aggr,vet,nomeF,numeroriga,mn);
    Tf=NULL;
    paramf=0;
    numpar=0;
    install(ST,nomeF,6,"procedure",0,0,0,mn,paramf,np,numpar,numeroriga);
    np=0;
    paramlista=0;
    sf=0;
}
;
pr_id_param:pr_id TONDAAPERTA seq TONDACHIUSA
|pr_id seq TONDACHIUSA
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d \"(\n"
        mancante\n",numeroriga);
    exit(0);
}
|pr_id TONDAAPERTA seq
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d \"(\n"
        mancante\n",numeroriga);
    exit(0);
}

```



```

}
;
pr_id_is_decl_part:pr_id_is_decl_part
;
pr_id_is_decl_part_bg:pr_id_is_decl_part BG
;
pr_id_is_decl_part_bg_stmt_list_f:pr_id_is_decl_part_bg stmt_list
;
pr_id_is_decl_part_bg_stmt_list_f_end:pr_id_is_decl_part_bg_stmt_list_f END
;
pr_id_is_decl_part_bg_stmt_list_f_end_id:pr_id_is_decl_part_bg_stmt_list_f_
end ID
{
    if(strcmp(nomeF,$2)!=0)
    {
        printf("ERRORE SEMANTICO: errore alla riga numero %d NOME PROCEDURA non
            corretto\n",numeroriga);
        exit(0);
    }
}
;
pr: pr_id_is_decl_part_bg_stmt_list_f_end_id PUNTOEVIRGOLA
{
    fprintf(out,"");
    nomeF=(char *)strdup(mn);
    fclose(out);
    out=m;
    pr=0;
}

|PROCEDURE IS
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d NOME DELLA
        PROCEDURA mancante\n",numeroriga);
    exit(0);
}
|pr_id
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d keyword IS
        mancante\n",numeroriga-1);
    exit(0);
}
|pr_id_param
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d keyword IS
        mancante\n",numeroriga-1);
    exit(0);
}
|pr_id_is_decl_part
{
    if(iskeyword(yytext))
        printf("ERRORE SEMANTICO:errore alla riga %d nome \"%s\" riservato ad
            una parola chiave del linguaggio ADA\n",numeroriga,yytext);
    printf("ERRORE SINTATTICO: errore alla riga numero %d keyword begin

```

```

        mancante\n",numeroriga);
    exit(0);
}
|pr_id_is_decl_part_bg_stmt_list_f
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d keyword END
        mancante\n",numeroriga);
    exit(0);
}
|pr_id_is_decl_part_bg_stmt_list_f_end PUNTOEVIRGOLA
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d NOME DELLA PROCEDURA
        mancante\n",numeroriga);
    exit(0);
}
|pr_id_is_decl_part_bg_stmt_list_f_end_id
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d ;
        mancante\n",numeroriga-1);
    exit(0);
}
;
//regole per la definizione dei parametri di una funzione o di una
//procedura
seq:seq id_seq r_tipo PUNTOEVIRGOLA|id_seq r_tipo PUNTOEVIRGOLA
|id_seq r_tipo
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d ;
        mancante\n",numeroriga);
    exit(0);
}
;
r_tipo: TIPO
{
    put_testo(&T,$1);
    tiporet=(char *)strdup($1);
    if(sf)
        paramf=1;
    if(r==0)
        inserisci_var($1);
    testo *punt=T;
    while(punt!=NULL)
    {
        put_testo(&Tf,punt->tes);
        punt=punt->next;
    }
    T=NULL;
}

|ID
{
    arr=1;
    inserisci_var($1);
    tiporet=(char *)strdup($1);

```

```

put_testo(&T,$1);
testo *punt=T;
while (punt!=NULL)
{
    put_testo(&Tf,punt->tes);
    punt=punt->next;
}
T=NULL;
r=0;
arr=0;
}
;
//regole per le chiamate delle funzioni o delle procedure dalla procedura
//principale del programma ADA 95 in input ad adac
id_ta:ID TONDAAPERTA
{
    put_testo(&T,$1);
    put_testo(&T,"(");
} term
;
c_funz: id_ta seq_par TONDACHIUSA {put_testo(&T,"")};}
|ID TONDAAPERTA TONDACHIUSA
{
    put_testo(&T,$1);
    put_testo(&T,"(");
    put_testo(&T,"")");
}
|id_ta TONDACHIUSA {put_testo(&T,"")");}
;
seq_par: VIRGOLA term seq_par
|VIRGOLA term
|VIRGOLA
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d valore o
        identificatore mancante\n",numeroriga);
    exit(0);
}
;
/*regole per la parte dichiarativa della procedura principale, delle
funzioni e delle procedure definite nel programma
ADA 95 in input ad adac */
decl_part: decl_part decl_s_t | decl_s_t
;
decl_s_t: fun
|pr
|decl_s PUNTOEVIRGOLA
{
    analisi_trad(T,ST,out,aggr,vet,nomeF,numeroriga,mn);
    costante=0;
    T=NULL;
}
|type_decl PUNTOEVIRGOLA
{
    testo *punt=T;

```

```

while (punt->next!=NULL)
    punt=punt->next;
install(ST,T->next->tes,3,
punt->tes,dimvet,estremoI,estremoS,nomeF,paramf,np,numpar,numeroriga);
T=NULL;
}
|id_seq ID
{
    arr=1;
    put_testo(&T,$2);
    context_check(ST,$2,nomeF,mn,numeroriga);
    inserisci_var($2);
} PUNTOEVIRGOLA
{
    analisi_trad(T,ST,out,aggr,vet,nomeF,numeroriga,mn);
    arr=0;
    T=NULL;
}
|id_seq ID{put_testo(&T,$2);} decl_c
{
    arr=1;
    context_check(ST,$2,nomeF,mn,numeroriga);
    inserisci_var($2);
} PUNTOEVIRGOLA
{
    analisi_trad(T,ST,m,aggr,vet,nomeF,numeroriga,mn);
    arr=0;
    T=NULL;
}
|type_decl
{
    printf("errore sintattico: errore alla riga numero %d ;
           mancante\n",numeroriga-1);
    exit(0);
}
|decl_s
{
    printf("errore sintattico: errore alla riga numero %d ;
           mancante\n",numeroriga-1);
    exit(0);
}
;
decl_s: id_seq decl
;
decl:decl_c |TIPO
{
    inserisci_var($1);
    put_testo(&T,$1);
}
;

```

```

//regola per la definizioni di costanti
decl_c: CONSTANT
{
    put_testo(&T,"constant");
    costante=1;
} ass
| ass
| CONSTANT
{
    put_testo(&T,"constant");
    costante=1;
}
ass_a
| ass_a
;

//regola per le operazioni di assegnazione
ass: TIPO{put_testo(&T,$1);} ASSIGN{put_testo(&T,":=");} term
{inserisci_var($1)}
;

//regole per l'operazione di aggregazione di un vettore
ass_a: ASSIGN{put_testo(&T,":=");} ass_agg
;
ass_agg: TONDAAPERTA{put_testo(&T,"{");} aggregato TONDACHIUSA
{put_testo(&T,"")};
;
aggregato: aggregato VIRGOLA {put_testo(&T,",");aggr=1;} term|term
;

/*regola per definire una sequenza di parametri di una funzione o di una
procedura ed è anche la regola usata per definire e dichiarare
una sequenza di variabili, costanti o array*/
id_seq : ID
{
    put_testo(&T,$1);
    if(paramlista)
        np=np+1;
} DUEPUNTI{put_testo(&T,":");}
| ID
{
    put_testo(&T,$1);
    if(paramlista)
        np=np+1;
} VIRGOLA{put_testo(&T,",");} id_seq
| ID
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d : o keyword
        \"function\" \"procedure\" mancanti oppure , o begin
        mancante\n",numeroriga);
    remove(filename);
    exit(0);
}

```

```

|ID IS
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d keyword type
           mancante\n",numeroriga);
    remove(filename);
    exit(0);
}
;
//regole per creare un nuovo tipo di array
type_decl:td1 TIPO{put_testo(&T,$2);}
|td1 {T=NULL;}
{
    printf("errore sintattico: errore alla riga numero %d tipo
           mancante\n",numeroriga);
    exit(0);
}
;
td1:td2 OF{put_testo(&T,"of");}
|td2 {T=NULL;}
{
    printf("errore sintattico: errore alla riga numero %d keyword of
           mancante\n",numeroriga);
    exit(0);
}
;
td2:td3 range
|td3 {T=NULL;}
{
    printf("errore sintattico: errore alla riga numero %d range
           mancante\n",numeroriga);
    exit(0);
}
;
td3:td4 ARRAY{put_testo(&T,"array");}
|td4 {T=NULL;}
{
    printf("errore sintattico: errore alla riga numero %d keyword array
           mancante\n",numeroriga);
    exit(0);
}
;
td4:td5 IS{put_testo(&T,"is");}
|TP IS {T=NULL;}
{
    printf("errore sintattico: errore alla riga numero %d IDENTIFICATORE
           mancante\n",numeroriga);
    exit(0);
}
|td5 ARRAY {T=NULL;}
{
    printf("errore sintattico: errore alla riga numero %d keyword IS
           mancante\n",numeroriga);
    exit(0);
}

```

```

;
td5:TP{put_testo(&T,"type");} ID{put_testo(&T,$3);}
;
range: TONDAAPERTA{put_testo(&T,"(");} range_int
TONDACHIUSA{put_testo(&T,")");}
|range_int TONDACHIUSA
{
    printf("ERRORE SINTATTICO: errore alla riga %d ( mancante",numeroriga);
    remove(filename);
    exit(0);
}
;
range_int: estremi PUNTOPUNTO{put_testo(&T,"..");} estremi
|PUNTOPUNTO estremi
{
    printf("ERRORE SINTATTICO: errore alla riga %d estremo inferiore del range
           mancante",numeroriga);
    remove(filename);
    exit(0);
}
|estremi PUNTOPUNTO{put_testo(&T,"..");}
;
estremi: INT_NUMBER
{
    get_dim_vet($1);
    sprintf(buff,"%d",$1);
    put_testo(&T,buff);
}
|INT_SIGN_NUMBER
{
    get_dim_vet($1);
    sprintf(buff,"%d",$1);
    put_testo(&T,buff);
}
;
/*regola per riconoscere i simboli terminali:gli identificatori,i
caratteri,i valori interi,i valori interi con segno,
i valori float,i valori float con segno*/
term:ID{put_testo(&T,$1);}
|INT_NUMBER
{
    sprintf(buff,"%d",$1);
    put_testo(&T,buff);
}
|INT_SIGN_NUMBER
{
    sprintf(buff,"(%d)", $1);
    put_testo(&T,buff);
}
|FLOAT_NUMBER
{
    sprintf(buff,"%f", $1);
    put_testo(&T,buff);
}

```

```

|FLOAT_SIGN_NUMBER
{
    sprintf(buff,"%f",$1);
    put_testo(&T,buff);
}
|S_CHAR{put_testo(&T,$1);}
;
//regole che ci permettono di riconoscere gli statement della procedura
//fondamentale, delle procedure e delle funzioni del programma in input ad
adac
stmt_list : /*empty*/| stmts
;
stmts : stmts stmt| stmt
;
//regola per riconoscere l'istruzione return associata ad una funzione
r_stmt: RETURN{put_testo(&T,"return");} term PUNTOEVIRGOLA
{
    if(pr==0)
    {
        return_checking(T,ST,nomeF,numeroriga,mn);
        analisi_trad(T,ST,out,aggr,vet,nomeF,numeroriga,mn);
    }
    else
    {
        printf("ERRORE SEMANTICO: errore alla riga numero %d keyword RETURN non
            prevista\n",numeroriga);
        exit(0);
    }
    T=NULL;
}
;
/*regola che ci permette di riconoscere le istruzioni che sono usate nella
procedura principale, nelle procedure e nelle funzioni
del programma in input ad adac*/
stmt:assign_stmt|if_stmt|for_stmt|while_stmt|put_get TONDAAPERTA
{put_testo(&T,"(");} term1 TONDACHIUSA{put_testo(&T,")");} PUNTOEVIRGOLA
{
    put_get_checking(ST,T,parpg,nomeF,numeroriga,mn);
    analisi_trad(T,ST,out,aggr,vet,nomeF,numeroriga,mn);
    T=NULL;
}|r_stmt
;
/*regole per riconoscere uno statement di assegnazione, nella regola
assign_stmt viene richiamata la regola della chiamata ad una procedura
o ad una funzione dato che questa a sintassi simile in alcuni casi (cioè
quando la funzione o la procedura hanno un solo parametro)
all'istruzione di accesso ad un elemento di un vettore.La chiamata ad una
funzione o ad una procedura è effettuata in questa regola per
poter effettuare a seconda che si tratti di un vettore o di una funzione
l'analisi semantica e la traduzione corretta*/
id_a:ID{put_testo(&T,$1);} ASSIGN{put_testo(&T,"=");}
;
assign_stmt: id_a expr PUNTOEVIRGOLA

```



```

{
    type_checking(T, ST, nomeF, numeroriga, mn);
    analisi_trad(T, ST, out, aggr, vet, nomeF, numeroriga, mn);
    T=NULL;
    aggr=0;
}
|c_funz ASSIGN{put_testo(&T, ":=");} expr PUNTOEVIRGOLA
{
    vet=1;
    type_checking(T, ST, nomeF, numeroriga, mn);
    analisi_trad(T, ST, out, aggr, vet, nomeF, numeroriga, mn);
    T=NULL;
    vet=0;
}
|id_a c_funz PUNTOEVIRGOLA
{
    testo *punt=T;
    punt=punt->next->next;
    el_ST e=getsym(ST, punt->tes, nomeF);
    if(e.t==6)
        function_checking(T, ST, nomeF, numeroriga);
    else
    {
        vet=1;
        type_checking(T, ST, nomeF, numeroriga, mn);
    }
    analisi_trad(T, ST, out, aggr, vet, nomeF, numeroriga, mn);
    T=NULL;
    vet=0;
}
|id_a expr
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d ;
           mancante\n", numeroriga-1);
    exit(0);
}
|c_funz PUNTOEVIRGOLA
{
    function_checking(T, ST, nomeF, numeroriga);
    analisi_trad(T, ST, out, aggr, vet, nomeF, numeroriga, mn);
    T=NULL;
}

|id_a c_funz op_m expr PUNTOEVIRGOLA
{
    testo *punt=T;
    punt=punt->next->next;
    el_ST e=getsym(ST, punt->tes, nomeF);
    if(e.t==4)
        vet=1;
    type_checking(T, ST, nomeF, numeroriga, mn);
    analisi_trad(T, ST, out, aggr, vet, nomeF, numeroriga, mn);
    T=NULL;
    aggr=0;
}

```

```

}
;
op_m:PIU{put_testo(&T,"+");}
|MENO{put_testo(&T,"-");}
|DIVISO{put_testo(&T,"/");}
|PER{put_testo(&T,"*");}
;
expr: expr PIU{put_testo(&T,"+");} expr
| expr MENO{put_testo(&T,"-");} expr
| expr DIVISO{put_testo(&T,"/");} expr
| expr PER{put_testo(&T,"*");} expr
| TONDAAPERTA{put_testo(&T,"(");} expr TONDACHIUSA{put_testo(&T,")");}
| term_el_v
| aggregato
;
//regole per riconoscere l'istruzione if e if-else
if_rel_m:IF{put_testo(&T,"if");} rel_m
|IF PUNTOEVIRGOLA
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d END
           mancante\n",numeroriga);
    exit(0);
}
;
if_rel_m_then: if_rel_m THEN
{
    put_testo(&T,"then");
    analisi_trad(T,ST,out,aggr,vet,nomeF,numeroriga,mn);
    T=NULL;
    fprintf(out,"\n{\n");
}
;
if_rel_m_then_stmt_list: if_rel_m_then stmt_list
;
if_stmt:if_rel_m_then_stmt_list else_stmt
|rel_m
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d keyword IF
           mancante\n",numeroriga);
    exit(0);
}

|IF THEN
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d condizione
           mancante\n",numeroriga);
    exit(0);
}
|if_rel_m
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d keyword THEN
           mancante\n",numeroriga-1);
    exit(0);
}

```

```

}
|if_rel_m_then else_stmt
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d lista statement
           mancante\n",numeroriga);
    exit(0);
}
;
rel: term1 OPREL{put_testo(&T,$2);} expr
;
rel_m: rel_m oplog THEN{put_testo(&T,"then");} rel
|rel
;
else_stmt:END IF PUNTOEVIRGOLA{fprintf(out,"\n}\n");}
|ELSE
{
    fprintf(out,"else");
    fprintf(out,"\n{\n");
} stmt_list END IF PUNTOEVIRGOLA{fprintf(out,"\n}\n}\n");}
|END IF
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d ;
           mancante\n",numeroriga);
    exit(0);
}
|END PUNTOEVIRGOLA{fprintf(out,"\n}\n");}
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d keyword IF
           mancante\n",numeroriga);
    exit(0);
}
;
//regola per riconoscere gli operatori logici
oplog: AND{put_testo(&T,"&&");}
|OR{put_testo(&T,"||");}
|NOT{put_testo(&T,"!");}
;
//regola per riconoscere un identificatore o un vettore
term1: ID{put_testo(&T,$1);} |term_el_v
;
//regole per riconoscere un vettore
term_el_v:ID{put_testo(&T,$1);} TONDAAPERTA{put_testo(&T,"[");} id_sign_n
TONDACHIUSA{put_testo(&T,"]");}
;
id_sign_n: ID{put_testo(&T,$1);}
|INT_NUMBER
{
    sprintf(buff,"%d", $1);
    put_testo(&T,buff);
}
|INT_SIGN_NUMBER
{
    sprintf(buff,"(%d)", $1);
    put_testo(&T,buff);
}

```

```

}
;
//regole per riconoscere il ciclo for
for_id: FOR{put_testo(&T,"for");} ID
{put_testo(&T,$3);contat=(char *)strdup($3);}
;
for_id_in:for_id IN
;
for_id_in_range_int: for_id_in range_int
|for_id range_int
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d keyword IN
           mancante\n",numeroriga);
    exit(0);
}
;
for_id_in_range_int_loop: for_id_in_range_int LOOP
{
    get_estremi();
    install(ST,T->next->tes,
    5,"integer",0,estremoI,estremoS,nomeF,paramf,np,numpar,numeroriga);
    put_testo(&T,"loop");analisi_trad(T,ST,out,aggr,vet,nomeF,numeroriga,mn);
    T=NULL;
    fprintf(out,"\n{\n");
}
;
for_id_in_range_int_loop_stmt_list: for_id_in_range_int_loop stmt_list
;
for_id_in_range_int_loop_stmt_list_end: for_id_in_range_int_loop_stmt_list
END
;
for_id_in_range_int_loop_stmt_list_end_loop:
for_id_in_range_int_loop_stmt_list_end LOOP
;
for_stmt:for_id_in_range_int_loop_stmt_list_end LOOP
PUNTOEVIRGOLA{delete_contatore(&ST,contat,nomeF);fprintf(out,"\n{\n");}
|ID PUNTOEVIRGOLA
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d END
           mancante\n",numeroriga);
    exit(0);
}
|ID
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d keyword FOR
           mancante\n",numeroriga);
    exit(0);
}
|FOR IN
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d NOME CONTATORE
           mancante\n",numeroriga);
    exit(0);
}
}

```

```

|for_id_in LOOP
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d range
           mancante\n",numeroriga);
    exit(0);
}
|for_id_in_range_int
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d keyword LOOP
           mancante\n",numeroriga-1);
    exit(0);
}
|for_id_in_range_int_loop END
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d lista statement
           mancante\n",numeroriga);
    remove(filename);
    exit(0);
}
|for_id_in_range_int_loop_stmt_list LOOP
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d keyword END
           mancante\n",numeroriga);
    exit(0);
}
|for_id_in_range_int_loop_stmt_list_end PUNTOEVIRGOLA
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d keyword LOOP
           mancante\n",numeroriga);
    exit(0);
}
|for_id_in_range_int_loop_stmt_list_end_loop
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d ;
           mancante\n",numeroriga-1);
    exit(0);
}
;
//regole per riconoscere il ciclo while
while_rel_m:WHILE{put_testo(&T,"while");} rel_m
;
while_rel_m_loop: while_rel_m LOOP
{
    put_testo(&T,"loop");
    analisi_trad(T,ST,out,aggr,vet,nomeF,numeroriga,mn);
    T=NULL;
    fprintf(out,"\n{\n");
}
;
while_rel_m_loop_stmt_list:while_rel_m_loop stmt_list
;
while_rel_m_loop_stmt_list_end:while_rel_m_loop_stmt_list END
;
while_rel_m_loop_stmt_list_end_loop:while_rel_m_loop_stmt_list_end LOOP

```

```

;
while_stmt:while_rel_m_loop_stmt_list_end LOOP
PUNTOEVIRGOLA{fprintf(out, "\n\n");}
|WHILE LOOP
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d relazione
           mancante\n", numeroriga);
    exit(0);
}
|rel_m LOOP
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d keyword WHILE
           mancante\n", numeroriga);
    exit(0);
}
|while_rel_m
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d keyword LOOP
           mancante\n", numeroriga-1);
    exit(0);
}
|while_rel_m_loop END
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d lista statement
           mancante\n", numeroriga);
    remove(filename);
    exit(0);
}
|while_rel_m_loop_stmt_list LOOP
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d keyword END
           mancante\n", numeroriga);
    exit(0);
}
|while_rel_m_loop_stmt_list_end
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d keyword LOOP
           mancante\n", numeroriga);
    exit(0);
}
|while_rel_m_loop_stmt_list_end_loop
{
    printf("ERRORE SINTATTICO: errore alla riga numero %d ;
           mancante\n", numeroriga-1);
    exit(0);
}
;

//regole per la gestione delle funzioni Put e Get delle librerie
//Ada.Integer_Text_IO, Ada.Float_Text_IO e Ada.Text_IO
put_get:put{put_testo(&T, "printf");}
|get{put_testo(&T, "scanf");}
;
put: PUT_INTEGER{parpg="integer";}
|PUT_CHARACTER{parpg="character";}

```

```

|PUT_FLOAT{parpg="float";}
;
get: GET_INTEGER{parpg="integer";}
|GET_CHARACTER{parpg="character";}
|GET_FLOAT{parpg="float";}
;

%%

/*Ultima sezione del parser, il prologo:
Nella funzione principale del parser (main) è creato il file in output ad
"adac" che avrà il seguente nome "nomeFile.c" (nomeFile è il nome del file
in input ad "adac" ). In questa funzione si copia il contenuto dei file
"nomeFunzione.c" o "nomeProcedura.c" (se nel file in input ad "adac" sono
definite delle funzioni o delle procedure) e del file "main.c" nel file
"nomeFile.c".*/
extern FILE *yyin;
void main( int argc, char **argv)
{
    int i;
    el_ST *punt;
    argc--;
    if ( argc > 0 )
        yyin= fopen( argv[1], "r" );
    else
        yyin = stdin;
    m=fopen("main.c","w");
    out=m;
    inizializza_ST(ST,&T);
    yyparse();
    fclose(out);
    /*sequenza di istruzioni per inserire nel file di output di "adac" la
traduzione delle procedure e delle funzioni usate nel programma in input ad
adac*/
    int n=strlen(argv[1]);
    n=n-4;
    filename=(char *)malloc((n+3)*sizeof(char));
    strncpy(filename,argv[1],n);
    filename[n]='\0';
    strcat(filename,".c");
    filename[n+2]='\0';
    out=fopen(filename,"w");
    if(text||text_integer||text_float)
        fprintf(out,"#include<stdio.h>\n");
    FILE *ff;
    char riga[200];
    while(nomifile!=NULL)
    {
        ff=fopen(nomifile->tes,"r");
        while(fgets(riga,200,ff)!=NULL)
            fputs(riga,out);
        fputs("\n",out);
        fclose(ff);
        remove(nomifile->tes); // se == -1 ho errore
    }
}

```

```
        nomifile=nomifile->next;
    }
    ff=fopen("main.c","r");
    while(fgets(riga,200,ff)!=NULL)
        fputs(riga,out);
    fclose(ff);
    remove("main.c");
    fclose(out);
}
void yyerror(char *s)
{
    /*errore segnalato per quelli errori sintattici non gestiti in "adac"*/
    printf ("ERRORE SINTATTICO:errore alla riga %d %s\n",numeroriga,s);
}
```


Analisi semantica e traduzione

L'**analisi semantica** e la **traduzione** del compilatore “*adac*” sono quelle parti del compilatore che permettono rispettivamente di individuare possibili “*errori semantici*” nel programma ADA 95 in input ad “*adac*” e di effettuare la traduzione del programma in input ad “*adac*” in un programma in **linguaggio C** che sarà l'output del compilatore “*adac*” quando non vengono individuati dal compilatore gli errori sintattici e gli errori semantici previsti.

Per **analisi semantica** si intende un'analisi del **significato del programma**.

L'analisi semantica di un compilatore non è rigidamente determinata da regole, a differenza dell'analisi sintattica.

In genere rientrano nell'analisi semantica i controlli dei tipi delle variabili in un'operazione di assegnazione (si cerca di verificare se le variabili usate nell'operazione di assegnazione sono tutte dello stesso tipo, ad esempio si verificano se sono tutte degli interi o dei float), il controllo dei parametri attuali e formali di una funzione o di una procedura e così via.

Per l'analisi semantica nel compilatore “*adac*” si è usato uno schema **SDT** (*Syntax Directed Translation*), uno schema di traduzione diretto.

Uno schema SDT serve per associare frammenti di programma o azioni semantiche alle produzioni di una grammatica, ovvero permette di eseguire azioni semantiche direttamente all'interno delle regole sintattiche che caratterizzano l'analisi sintattica, implementata nel file “*parser.y*”.

Le azioni semantiche sono eseguite quando la regola di produzione è stata riconosciuta durante l'analisi sintattica.

L'analisi semantica del file in input ad “*adac*” dipende dalla semantica del linguaggio sorgente, nel nostro caso dalla semantica del **linguaggio ADA 95**.

La traduzione in un compilatore dipende sia dalla sintassi che dalla semantica dei linguaggi sorgente e destinazione.

Per la fase di traduzione del compilatore “*adac*” che consente di **tradurre** un programma in ADA 95 in un programma C si è usato un **approccio semplificato** (*una passata*), infatti si sintetizza l'output, si traduce il programma in input ad “*adac*” direttamente durante l'analisi sintattica, senza costruire un albero sintattico astratto **AST** (*Abstract Syntax Tree*).

L'approccio semplificato ad una passata funziona solo nei casi “*semplici*” di traduzione e di elaborazione dell'input, per ciò per il progetto didattico implementato che ha delle limitazioni come indicato nella sezione “*Limitazioni*”, si è pensato di usare questo approccio.

Invece, con un AST, si può implementare un insieme molto più grande di funzioni di traduzione e di elaborazione.

Nel progetto implementato si è usata una **traduzione guidata dalla sintassi**, infatti sarebbe difficile progettare in modo globale un algoritmo di traduzione di un linguaggio complesso senza prima analizzare la frase sorgente nei suoi costituenti definiti dalla sintassi.

L'analisi sintattica riduce la complessità del problema della traduzione, scomponendolo in sottoproblemi più semplici.

Per ogni parte di una frase del programma sorgente il traduttore esegue le azioni appropriate per preparare la traduzione, azioni che consistono nel raccogliere informazioni per emettere certe parti delle frasi del linguaggio target.

In altre parole il lavoro del traduttore è modularizzato secondo la struttura descritta dall'albero sintattico nel file “*parser.y*”.

Perciò questo tipo di traduttori è detto **guidato dalla sintassi**.

Si parla di traduzione guidata dalla sintassi nei casi in cui è possibile definire il processo di sintesi dell'output (*traduzione*) sulla base della struttura sintattica dell'input.

In questi casi l'operazione di traduzione è fortemente legata alla sintassi dell'input, pertanto la traduzione può essere definita in modo “*parallelo*” alla definizione della sintassi dell'input.

Anche l'esecuzione di un controllo semantico così come nel caso della traduzione avviene in parallelo all'applicazione della corrispondente regola sintattica nell'analisi sintattica.

Dunque l'ordine di applicazione delle regole semantiche e dei controlli semantici è conseguenza dell'ordine di applicazione delle regole di produzione nell'analisi sintattica.

Le operazioni per l'analisi semantica e la traduzione del compilatore “*adac*” sono state implementate nel file “*semtrad.h*” le cui caratteristiche vengono descritte di seguito dai commenti evidenziati in colore giallo.

semtrad.h

```

/*L'istruzione enum segue la seguente logica:
creo una lista di valori che saranno trattati come valori interi a partire
da 0 quindi: 0,1,2,3..
i valori che caratterizzano la lista non sono delle stringhe che possono
essere stampate così come sono ma solo dei valori interi quindi per la enum
seguente è possibile definire una variabile: tipo t=variabile; (che
corrisponde a tipo t=1;contatore è la variabile usata in un ciclo for)...*/
typedef enum tipo {variabile=1,constant,type,array,contatore,funzione}
tipo;//il valore 1 dopo variabile indica che si parte dall'intero 1
/*struct testo è una lista con un campo tes che è un vettore di caratteri e
un campo next puntatore ad un elemento della lista*/
typedef struct testo
{
    char *tes;
    struct testo *next;
} testo;
//campi di un elemento della symbol table
typedef struct el_ST
{
    int valh;/*indice della Symbol Table in cui si inserisce il token
riconosciuto dallo scanner*/
    char *nameToken;//nome del token che si inserisce nella Symbol Table
    int dim_vet;/*dimensione del vettore che si inserisce nella Symbol Table
altrimenti questo campo ha valore 0*/
    int esI;/*estremo inferiore del vettore che si inserisce nella Symbol
Table altrimenti questo campo ha valore 0*/
    int esS;/*estremo superiore del vettore che si inserisce nella Symbol
Table altrimenti questo campo ha valore 0*/
    struct el_ST *next;/*puntatore ad un elemento della Symbol Table, serve
per gestire le liste che si creano nel caso in cui si verificano delle
collisioni*/
    tipo t;/*tipo di token o del simbolo inserito nella Symbol Table, il
simbolo inserito può essere variabile, constant, type, array, contatore,
funzione*/
    char *tv;/*tipo di variabile, costante, array o tipo restituito da una
funzione in generale questo campo assume valori: "integer", "float" o
"character"*/
    char *funz;/*nome della funzione o della procedura in cui è definito un
token*/

```

```

int pf; /* se settata a 1 (parametro di modo in) indica che la variabile
a cui si riferisce è un parametro di una funzione o procedura,
altrimenti se è settata a 0 la variabile che si considera è una semplice
variabile usata in una funzione o procedura */

int nparam; /* indica il numero di parametri di una funzione o di una
procedura */

int nump; /* indica l'ordine con cui dichiaro i parametri di una funzione
o di una procedura */

}el_ST;
//struttura contenente le parole chiavi usate dell' ADA 95
typedef struct KEY_TABLE
{
    char *kwM;
    char *kwm;
} KEY_TABLE;
#define NUM_KEYWORDS 22 //numero di parole chiavi usate
#define MAXDIMST 128 //dimensione della Symbol Table
//parole chiavi dell'ADA 95 usate nel progetto
KEY_TABLE key_tab[NUM_KEYWORDS] = {
    {"AND", "and"},
    {"ARRAY", "array"},
    {"BEGIN", "begin"},
    {"CONSTANT", "constant"},
    {"ELSE", "else"},
    {"END", "end"},
    {"FOR", "for"},
    {"FUNCTION", "function"},
    {"IF", "if"},
    {"IN", "in"},
    {"IS", "is"},
    {"LOOP", "loop"},
    {"NOT", "not"},
    {"OF", "of"},
    {"OR", "or"},
    {"PROCEDURE", "procedure"},
    {"RANGE", "range"},
    {"RETURN", "return"},
    {"THEN", "then"},
    {"TYPE", "type"},
    {"WHILE", "while"},
    {"WITH", "with"}
};

```

```

/*La funzione calcola_valh è una funzione che calcola il valh della Symbol
Table associato ad un token riconosciuto dall'analizzatore lessicale, ha
come parametro un puntatore a caratteri contenente il token (il simbolo)
per cui calcolare il valh*/
int calcola_valh(char *nome)
{
    int somma=0;
    int n=strlen(nome);
    int i;
    int k;
    for(i=0;i<n;i++)
        somma=somma+nome[i];
    k=somma%MAXDIMST;
    return k;
}
/*La funzione putsym permette di inserire un nuovo elemento nella symbol
table è una funzione che ha i seguenti parametri:
el_ST *s che è il puntatore alla Symbol Table, char *sym_name contiene
l'identificatore da inserire nella Symbol Table,
tipo valt campo che indica il tipo dell'identificatore
(variabile,constant,type,array,contatore,funzione),
char *tvar indica il tipo "integer","float","character" di
variabile,constant,type e funzione,
int dim indica la dimensione del vettore se l'identificatore che si
inserisce è un nuovo type rappresentante un array,
int ei indica l'estremo inferiore del type di array, int es indica
l'estremo superiore del type di array,
char *funzione indica la funzione alla quale appartiene l'identificatore,
int parf settato a 1 indica che l'identificatore è un parametro di una
funzione o procedura di modo in (non può essere modificato all'interno
della funzione o procedura di cui è parametro),
int np indica il numero di parametri di una funzione o procedura,
int numpar è un valore che incdica l'ordine con cui dichiaro i parametri di
una funzione o procedura,
L'inserimento nella Symbol Table avviene usando la tecnica hash table.*/

```

```

void putsym (el_ST *s, char *sym_name, tipo valt, char *tvar, int dim, int
ei, int es, char *funzione, int parf, int np, int numpar)
{
    int k=calcola_valh(sym_name);
    el_ST *p_el_ST;
    el_ST *punt;
    if(s[k].valh==-1)
    {
        s[k].valh=k;
        s[k].nameToken=(char *)strdup(sym_name);
        s[k].t=valt;
        s[k].dim_vet=dim;
        s[k].esI=ei;
        s[k].esS=es;
        s[k].tv=(char *)strdup(tvar);
        s[k].funz=(char *)strdup(funzione);
        s[k].pf=parf;
        s[k].nparam=np;
        s[k].nump=numpar;
        s[k].next=NULL;
    }
    else
    {
        p_el_ST=(el_ST *)malloc(sizeof(el_ST));
        p_el_ST->nameToken=(char *)strdup(sym_name);
        p_el_ST->t=valt;
        p_el_ST->tv=(char *)strdup(tvar);
        p_el_ST->funz=(char *)strdup(funzione);
        p_el_ST->pf=parf;
        p_el_ST->dim_vet=dim;
        p_el_ST->esI=ei;
        p_el_ST->esS=es;
        p_el_ST->valh=k;
        p_el_ST->nparam=np;
        p_el_ST->nump=numpar;
        p_el_ST->next=NULL;
    }
}

```

```

        if(s[k].next==NULL)
            s[k].next=p_el_ST;
        else
        {
            punt=s[k].next;
            while(punt->next!=NULL)
                punt=punt->next;
            punt->next=p_el_ST;
        }
    }
}

/*La funzione getsym restituisce un elemento della Symbol Table.
Gli argomenti che la caratterizzano sono: s il puntatore alla Symbol
Table, sym_name è il nome del simbolo che vogliamo prelevare
dalla Symbol Table e funzione è il nome della funzione in cui viene usato
il simbolo che preleviamo.*/
el_ST getsym (el_ST *s,char *sym_name,char *funzione)
{
    int k=calcola_valh(sym_name);
    int trov=0;
    el_ST *punt;
    if(s[k].valh!=-1)
    {

if(strcmp(s[k].nameToken,sym_name)==0&&strcmp(s[k].funz,funzione)==0)
        return s[k];
    else
    {
        punt=s[k].next;
        while(punt!=NULL&&!trov)
        {
            if(strcmp(punt->nameToken,sym_name)==0&&strcmp(punt->funz,funzione)==0)
                trov=1;
            else
                punt=punt->next;
        }
    }
}
}

```

```

        }
    }
}
if(trov)
    return *punt;
else
{
    el_ST y;
    y.valh=-1;
    return y;
}
}
/*La funzione inizializza_ST serve per inizializzare la Symbol Table
impostando il campo valh a -1*/
void inizializza_ST(el_ST *s,testo **TT)
{
    int i;
    *TT=NULL;
    for(i=0;i<MAXDIMST;i++)
        s[i].valh=-1;
}
/*La funzione install serve per effettuare un controllo semantico, prima
che un nuovo simbolo sia inserito nella Symbol Table si verifica
se questo non è stato già inserito.
Se il simbolo da inserire nella symbol table è già presente viene segnalato
un errore semantico altrimenti si inserisce il simbolo nella Symbol Table.
L'obiettivo della funzione è evitare di ridefinire una variabile già
definita.
I parametri della funzione sono:
el_ST *s che è il puntatore alla Symbol Table, char *sym_name contiene il
nuovo simbolo da inserire nella Symbol Table,
tipo valt campo che indica il tipo del simbolo riconosciuto dallo scanner
(variabile,constant,type,array,contatore,funzione),
char *tvar indica il tipo "integer","float","character" di
variabile,constant,type e funzione,

```



```

int dim indica la dimensione del vettore se l'identificatore che si
inserisce è un nuovo type rappresentante un array,
int ei indica l'estremo inferiore del type di array, int es indica
l'estremo superiore del type di array,
char *funzione indica la funzione alla quale appartiene l'identificatore,
int parf settato a 1 indica che l'identificatore è un parametro di una
funzione o procedura di modo in
(non può essere modificato all'interno della funzione o procedura di cui è
parametro),
int np indica il numero di parametri di una funzione o procedura,
int numpar è un valore che indica l'ordine con cui dichiaro i parametri di
una funzione o procedura
int nr è un valore intero positivo da 1...n (n numero di righe del programma
ADA) indica la riga alla quale è definita la variabile */
void install (el_ST *s,char *sym_name, tipo valt,char *tvar,int dim,int
ei,int es,char *funzione,int parf,int np,int numpar,int nr)
{
    el_ST rg=getsym(s,sym_name,funzione);
    if(rg.valh==-1)
        putsym (s,sym_name,valt,tvar,dim,ei,es,funzione,parf,np,numpar);
    else
    {
        printf( "ERRORE SEMANTICO:errore alla riga %d la variabile \"%s\"
                è già\ ' stata definita\r\n",nr,sym_name);
        exit(0);
    }
}

/*La funzione context_check serve per verificare se un nuovo tipo di dato o
una variabile o una funzione è stata definita o no,
oppure serve per vedere se si è definito un nuovo tipo di dato (array)
nella funzione principale che può essere usato anche in una
sottofunzione della funzione principale.*/
el_ST context_check(el_ST *s,char *symname,char *funzione,char *mn,int nr)
{
    el_ST r=getsym(s,symname,funzione);
    if(r.valh==-1)

```

```

{
    r=getsym(s,symname,mn);
    if(r.valh==-1)
    {
        printf( "\nERRORE SEMANTICO:errore alla riga %d tipo o variabile
                o funzione \"%s\" non definita\n",nr,symname);
        exit(0);
    }
    /*errore quando in una funzione o procedura cerco di usare una
       variabile della funzione principale.Della funzione principale
       si può usare solo l'identificatore associato ad un nuovo tipo
       di dato (array)*/
    if(r.valh!=-1&& r.t!=3)
    {
        printf( "\nERRORE SEMANTICO:errore alla riga %d tipo o variabile
                o funzione \"%s\" non definita\n",nr,symname);
        exit(0);
    }
}
return r;
}

/*La funzione delete_contatore è una funzione che serve per eliminare dalla
Symbol Table le variabili contatori quando il ciclo for a cui sono
associate le variabili contatori termina.
La funzione delete_contatore ha tre parametri:
el_ST (*s) [MAXDIMST]: puntatore alla Symbol Table che è passata non per
valore, ma per indirizzo poichè deve essere modificata;
char *cont: è il nome della variabile contatore da eliminare dalla Symbol
Table;
char *funzione: è la funzione o procedura in cui viene usata la variabile
contatore.*/
void delete_contatore(el_ST (*s) [MAXDIMST],char *cont,char *funzione)
{
    int k=calcola_valh(cont);
    if((*s)[k].valh!=-1)
    {

```

```

if(strcmp((*(s)[k].nameToken, cont)==0)
{
    if((*(s)[k].next==NULL)
        (*(s)[k].valh=-1;
    else
    {
        el_ST *punt=(*(s)[k].next;
        (*(s)[k].valh=punt->valh;
        (*(s)[k].nameToken=(char *)strdup(punt->nameToken);
        (*(s)[k].t=punt->t;
        (*(s)[k].dim_vet=punt->dim_vet;
        (*(s)[k].esI=punt->esI;
        (*(s)[k].esS=punt->esS;
        (*(s)[k].tv=(char *)strdup(punt->tv);
        (*(s)[k].funz=(char *)strdup(punt->funz);
        (*(s)[k].pf=punt->pf;
        (*(s)[k].nparam=punt->nparam;
        (*(s)[k].nump=punt->nump;
        if(punt->next==NULL)
            (*(s)[k].next=NULL;
        else
            (*(s)[k].next=punt->next;
        free(punt);
    }
}
else
{
    el_ST *succ,*prec;
    int trov=0;
    prec=(*(s)[k].next;
    if(strcmp(prec->nameToken, cont)==0)
    {
        if(prec->next==NULL)
            (*(s)[k].next=NULL;
        else
            (*(s)[k].next=prec->next;
    }
}

```

```

    free(prec);
}
else
{
    succ=prec->next;
    while(succ!=NULL&&!trov)
    {
        if(strcmp(succ->nameToken,cont)==0)
            trov=1;
        else
        {
            prec=succ;
            succ=succ->next;
        }
    }
    if(succ->next==NULL)
        prec->next=NULL;
    else
        prec->next=succ->next;
    free(succ);
}
}
}
}

/*La funzione getpar serve per trovare nella Symbol Table i parametri
formali di una funzione definiti secondo un certo ordine all'interno della
funzione. Infatti la funzione getpar ha come parametri:
el_ST *s che è il puntatore alla Symbol Table,
int nump che è un intero positivo che va da 1...n (n numero di parametri
della sottofunzione) indica in che posizione nella definizione
dei parametri della sottofunzione si trova il parametro da ricercare,
char *funzione indica la funzione alla quale appartiene il parametro da
ricercare.
La funzione getpar restituisce un elemento della Symbol Table un elemento
di tipo el_ST*
el_ST getpar(el_ST *s,int nump,char *funzione)

```

```

{
    el_ST *punt;
    int i;
    for(i=0;i<MAXDIMST;i++)
    {
        if(s[i].valh!=-1)
        {
            if(strcmp(s[i].funz,funzione)==0)
            {
                if(nump==s[i].nump)
                    return s[i];
            }
            else
            {
                if(s[i].next!=NULL)
                {
                    punt=s[i].next;
                    while(punt!=NULL)
                    {
                        if(strcmp(punt->funz,funzione)==0)
                        {
                            if (nump==punt->nump)
                                return *punt;
                        }
                        punt=punt->next;
                    }
                }
            }
        }
    }
}

/*La funzione numterminal serve per verificare se il suo parametro punt
contiene nel campo tes un valore intero o un valore float.
Il parametro formale testo *punt è un puntatore alla lista testo, nel campo
tes conterrà ho un valore intero o un valore float.

```

La funzione numterminal restituisce un char * ovvero un puntatore ad una sequenza di caratteri contenente la stringa "integer" o "float" poichè nel campo tes puntato da punt ci sarà rispettivamente un valore intero o un valore float.*/

```
char * numterminal(testo *punt)
{
    if(isdigit(punt->tes[0]) || ((punt->tes[0]=='(')&&(punt->tes[1]=='-'
    '&&(punt->tes[strlen(punt->tes)-1]==')'))))
    {
        char *valore=(char *)strdup(punt->tes);
        int trov=0,i=1;
        while(i<strlen(valore)&&!trov)
        {
            if(valore[i]=='.')
                trov=1;
            i=i+1;
        }
        char *v;
        if(trov)
            v="float";
        else
            v="integer";
        return v;
    }
    return NULL;
}
```

/*La funzione **return_checking** è una funzione che esegue dei controlli semantici sul tipo di dato restituito da una function definita in un programma ADA. La funzione return_checking ha i seguenti parametri: **testo *T** che è un puntatore ad una lista testo e contiene il testo riconosciuto dal parser per la regola associato allo statement return e che serve per controllare se l'istruzione di return è fatta in maniera coerente o no, **el_ST *s** che è il puntatore alla Symbol Table, **char *funzione** indica la funzione alla quale appartiene l'istruzione return,

```

int nr indica il numero della riga alla quale è definita l'istruzione
return,
char *mn indica la funzione principale del programma ADA*/
void return_checking(testo *T,el_ST *s,char *funzione,int nr,char *mn)
{
    testo *punt=T;
    punt=punt->next;
    el_ST f=context_check(s,funzione,mn,mn,nr);
    if(isalpha(punt->tes[0]))
    {
        el_ST r=context_check(s,punt->tes,funzione,mn,nr);
        if(strcmp(r.tv,f.tv)!=0)
        {
            printf( "\nERRORE SEMANTICO:errore alla riga %d il tipo restituito
                dalla funzione \"%s\" non è di tipo \"%s\"\\n",nr,f.nameToken,r.tv);
            exit(0);
        }
    }
    char *t=numterminal(punt);
    if(t!=NULL)
    {
        if(strcmp(t,f.tv)!=0)
        {
            printf( "\nERRORE SEMANTICO:errore alla riga %d il tipo restituito
                dalla funzione \"%s\" non è di tipo \"%s\"\\n",nr,f.nameToken,t);
            exit(0);
        }
    }
    if(punt->tes[0]=='\\')
    {
        if(strcmp("character",f.tv)!=0)
        {
            printf( "\nERRORE SEMANTICO:errore alla riga %d il tipo restituito
                dalla funzione \"%s\" non è di tipo \"character\"\\n",nr,f.nameToken);
            exit(0);
        }
    }
}

```

```

    }
}
/*La funzione function_checking è una funzione che esegue dei controlli
semantici sulle function o procedure definite in un programma ADA,
quando queste sono richiamate nella funzione principale.
La funzione return_checking ha i seguenti parametri:
testo *T che è un puntatore ad una lista testo e contiene il testo
riconosciuto dal parser per la regola associato allo statement
di chiamata di una funzione, è necessario per controllare se la chiamata ad
una function o procedure è fatta in maniera coerente o no,
el_ST *s che è il puntatore alla Symbol Table,
char *funzione contiene il nome della funzione principale del programma ADA
nel quale viene richiamata una function o una procedure,
int nr indica il numero della riga alla quale è definita l'istruzione di
chiamata a funzione*/
void function_checking(testo *T,el_ST *s,char *funzione,int nr)
{
    testo *punt=T;
    int trovapar=0;
    int ass=0;
    el_ST ell,elf,el2;
    punt=punt->next;
    if(strcmp(punt->tes,":")==0)
    {
        ell=getsym(s,T->tes,funzione);
        if(ell.valh==-1)
        {
            printf( "\nERRORE SEMANTICO:errore alla riga %d la variabile \"%s\"
                    non è stata definita\n",nr,T->tes);
            exit(0);
        }
        punt=punt->next;
        elf=getsym(s,punt->tes,funzione);
        if(elf.valh==-1)
        {

```



```

    printf( "\nERRORE SEMANTICO:errore alla riga %d la function \"%s\"
non è stata definita\n",nr,punt->tes);
    exit(0);
}
if(strcmp(elf.tv,"procedure")==0)
{
    printf( "\nERRORE SEMANTICO:errore alla riga %d la procedura \"%s\"
    non è una \"function\" è una \"procedure\"\n",nr,elf.nameToken);
    exit(0);
}
if(strcmp(elf1.tv,elf.tv)!=0)
{
    printf( "\nERRORE SEMANTICO:errore alla riga %d la variabile \"%s\" è
    di tipo \"%s\" mentre la funzione \"%s\" restituisce un
    \"%s\"\n",nr, elf1.nameToken,elf1.tv,elf.nameToken,elf.tv);
    exit(0);
}
ass=1;
}
if(ass)
    punt=punt->next;
else
{
    elf=getsym(s,T->tes,funzione);
    if(elf.valh==-1)
    {
        printf( "\nERRORE SEMANTICO:errore alla riga %d la procedure \"%s\"
        non è stata definita\n",nr,T->tes);
        exit(0);
    }
    if(strcmp(elf.tv,"procedure")!=0)
    {
        printf( "\nERRORE SEMANTICO:errore alla riga %d la funzione \"%s\"
        restituisce un \"%s\" non è una \"procedure\"\n"
        ,nr,elf.nameToken,elf.tv);
        exit(0);
    }
}

```

```

    }
}
punt=punt->next;
if(strcmp(punt->tes,"")==0)
{
    if(elf.nparam!=0)
    {
        printf( "\nERRORE SEMANTICO:errore alla riga %d la funzione \"%s\"
                prevede %d parametri\n",nr,elf.nameToken,elf.nparam);
        exit(0);
    }
}
else
{
    testo *puntapp=punt;
    while(strcmp(punt->tes,"")!=0)
    {
        trovapar=trovapar+1;
        punt=punt->next;
    }
    if(elf.nparam!=trovapar)
    {
        printf( "\nERRORE SEMANTICO:errore alla riga %d la funzione \"%s\"
                prevede %d parametri\n",nr,elf.nameToken,elf.nparam);
        exit(0);
    }
    punt=puntapp;
    trovapar=0;
    while(strcmp(punt->tes,"")!=0)
    {
        trovapar=trovapar+1;
        ell=getpar(s,trovapar,elf.nameToken); //in ell ho parametro formale
        char *v=numterminal(punt);
        if(v!=NULL)
        {
            if(strcmp(ell.tv,v)!=0)

```

```

{
    printf( "\nERRORE SEMANTICO:errore alla riga %d la variabile di
            \"%s\" \"%s\" è \"%s\" accetta solo valori
            \"%s\"\n",nr,el1.funz,el1.nameToken,el1.tv,el1.tv);
    exit(0);
}
}
else
{
    if(punt->tes[0]=='\'' )
    {
        if(strcmp(el1.tv,"character")!=0)
        {
            printf( "\nERRORE SEMANTICO:errore alla riga %d la variabile di
                    \"%s\" \"%s\" è \"%s\" non accetta valori
                    \"character\"\n",nr,el1.funz,el1.nameToken,el1.tv);
            exit(0);
        }
    }
    else
    {
        el2=getsym(s,punt->tes,funzione);//in el2 ho parametro attuale
        if(el2.valh==-1)
        {
            printf( "\nERRORE SEMANTICO:errore alla riga %d il parametro
                    attuale \"%s\" non è stato definito\n",nr,punt->tes);
            exit(0);
        }
        if(el1.t!=el2.t)
        {
            printf( "\nERRORE SEMANTICO:errore alla riga %d il parametro
                    attuale \"%s\" non è dello stesso tipo del parametro formale
                    \"%s\"\n",nr,el2.nameToken,el1.nameToken);
            exit(0);
        }
        if(strcmp(el1.tv,el2.tv)!=0)

```

```

    {
        printf( "\nERRORE SEMANTICO:errore alla riga %d la variabile di
            \"%s\" \"%s\" è \"%s\" mentre la variabile di \"%s\" \"%s\" è
            \"%s\"\\n",nr,el1.funz,el1.nameToken,el1.tv,el2.funz,
            el2.nameToken,el2.tv);
        exit(0);
    }
}
}
punt=punt->next;
}
}
}
/*La funzione index_checking viene usata per verificare se l'indice di un
vettore è nel range di indici che il vettore consente.
La funzione index_checking ha i seguenti parametri:
el_ST e è l'elemento della Symbol Table in cui è memorizzato il nuovo tipo
di array associato al vettore in analisi,
testo *punt è un puntatore ad una lista testo e contiene il valore o la
variabile contatore per cui verificare se rispettano il range di indici
ammessi dal vettore,
int nr indica il numero della riga alla quale è definita un'istruzione
contenente un'operazione  $v(i) := \dots, v(1) := \dots, a := v(1) * \dots$  dove  $v$  è un
vettore e  $a$  una variabile,
el_ST evet è l'elemento della Symbol Table in cui è memorizzato il vettore,
char *funzione contiene il nome della funzione nella quale viene usato un
elemento del vettore  $v(i)$  o  $v(1)$  per esempio,
el_ST *s che è il puntatore alla Symbol Table.*/
void index_checking(el_ST e,int nr,testo *punt,el_ST evet,char
*funzione,el_ST *s)
{
    char *str;
    if(isdigit(punt->tes[0]))
    {
        int val=atoi(punt->tes);
        if(val<e.esI||val>e.esS)

```

```

{
    printf( "\nERRORE SEMANTICO:errore alla riga %d valore \"%d\" fuori
           dal range del vettore \"%s\" \n",nr, val, evet.nameToken );
    exit(0);
}
}
else
{
    if (punt->tes[0]=='(' && punt->tes[1]=='-' && punt->tes[strlen(punt->tes)-
1]=='')
    {
        str=(char *)malloc((strlen(punt->tes)-1)*sizeof(char));
        int j=0;
        int i;
        for(i=1;i<strlen(punt->tes)-1;i++)
        {
            str[j]=punt->tes[i];
            j++;
        }
        str[strlen(str)]='\0';
        int val=atoi(str);
        if(val<e.esI || val>e.esS)
        {
            printf( "\nERRORE SEMANTICO:errore alla riga %d valore \"%d\" fuori
                   dal range del vettore \"%s\" \n",nr, val, evet.nameToken );
            exit(0);
        }
    }
    else
    {
        if (strcmp (punt->tes,")")==0)
        {
            printf( "\nERRORE SEMANTICO:errore alla riga %d indice per il vettore
                   \"%s\" non presente\n",nr, evet.nameToken);
            exit(0);
        }
    }
}

```

```

else
{
el_ST el=getsym(s,punt->tes,funzione);
if(el.valh==-1)
{
printf( "\nERRORE SEMANTICO:errore alla riga %d variabile o tipo
        \"%s\" non definito\n",nr,punt->tes);
exit(0);
}
if(el.t==5)
{
if(e.esI!=el.esI||e.esS!=el.esS)
{
printf( "\nERRORE SEMANTICO:errore alla riga %d la variabile
        contatore \"%s\" è fuori dal range del vettore \"%s\" \n",
        nr,el.nameToken,evet.nameToken);
exit(0);
}
}
if(el.t==1)
{
if(strcmp(el.tv,"integer")!=0)
{
printf( "\nERRORE SEMANTICO:errore alla riga %d un vettore può
        avere come indici solo variabili \"integer\" o
        \"contatori\"\n",nr, el.nameToken,evet.nameToken );
exit(0);
}
}
}
}
}

/*La funzione put_get_checking serve per effettuare i controlli semantici
sui metodi Put e Get delle istruzioni Ada.Integer_Text_IO,
Ada.Float_Text_IO e Ada.Text_IO.

```

La funzione `put_get_checking` ha i seguenti parametri:

el_ST *s che è il puntatore alla Symbol Table,

testo *T che è un puntatore ad una lista testo e contiene il testo

riconosciuto dal parser per le regola associate agli statement

`Ada.Integer_Text_IO.Put`, `Ada.Integer_Text_IO.Get`, `Ada.Float_Text_IO.Put`, `Ada.F`

`loat_Text_IO.Get`, `Ada.Text_IO.Put` e `Ada.Text_IO.Get` è necessario

per i controlli semantici di queste istruzioni,

char *str è un puntatore a caratteri contenente la stringa "integer" se si

devono analizzare le istruzioni `Ada.Integer_Text_IO.Put` e

`Ada.Integer_Text_IO.Get`, la stringa "float" se si devono analizzare le

istruzioni `Ada.Float_Text_IO.Put` e `Ada.Float_Text_IO.Get`,

e la stringa "character" se si devono analizzare le istruzioni

`Ada.Text_IO.Put` e `Ada.Text_IO.Get`,

char *funzione indica la funzione alla quale appartengono le istruzioni

`Ada.Integer_Text_IO.Put`, `Ada.Integer_Text_IO.Get`, `Ada.Float_Text_IO.Put`, `Ada.F`

`loat_Text_IO.Get`, `Ada.Text_IO.Put` e `Ada.Text_IO.Get`,

int nr indica il numero della riga alla quale sono definite le istruzioni:

`Ada.Integer_Text_IO.Put`, `Ada.Integer_Text_IO.Get`, `Ada.Float_Text_IO.Put`, `Ada.F`

`loat_Text_IO.Get`, `Ada.Text_IO.Put` e `Ada.Text_IO.Get`,

char *mn indica la funzione principale del programma ADA*/

`void put_get_checking(el_ST *s, testo *T, char *str, char *funzione, int`

`nr, char *mn)`

```
{
    testo *punt=T;
    punt=punt->next->next;
    el_ST el=context_check(s,punt->tes,funzione,mn,nr);
    if(el.t==1)
    {
        if(strcmp(el.tv,str)!=0)
        {
            if(strcmp(str,"integer")==0)
                printf( "\nERRORE SEMANTICO:errore alla riga %d Ada.Integer_Text_IO
                    opera con \"%s\" non con \"%s\"\\n",nr,str,el.tv);
            if(strcmp(str,"float")==0)
                printf( "\nERRORE SEMANTICO:errore alla riga %d Ada.Float_Text_IO
                    opera con \"%s\" non con \"%s\"\\n",nr,str,el.tv);
```

```

        if(strcmp(str,"character")==0)
        printf( "\nERRORE SEMANTICO:errore alla riga %d Ada.Text_IO opera con
                \"%s\" non con \"%s\"\\n",nr,str,el.tv);
        exit(0);
    }
}
if(el.t==4)
{
    char *vet=(char *)strdup(punt->tes);
    el_ST evet=context_check(s,vet,funzione,mn,nr);
    el_ST e=context_check(s,evet.tv,funzione,mn,nr);
    punt=punt->next;
    if(strcmp(punt->tes,"[")!=0)
    {
        printf( "\nERRORE SEMANTICO:errore alla riga %d operazione non
                consentita\\n",nr);
        exit(0);
    }
    punt=punt->next;
    index_checking(e,nr,punt,evet,funzione,s);
    if(strcmp(e.tv,str)!=0)
    {
        if(strcmp(str,"integer")==0)
        printf( "\nERRORE SEMANTICO:errore alla riga %d Ada.Integer_Text_IO
                opera con \"%s\" non con \"%s\"\\n",nr,str,e.tv);
        if(strcmp(str,"float")==0)
        printf( "\nERRORE SEMANTICO:errore alla riga %d Ada.Float_Text_IO
                opera con \"%s\" non con \"%s\"\\n",nr,str,e.tv);
        if(strcmp(str,"character")==0)
        printf( "\nERRORE SEMANTICO:errore alla riga %d Ada.Text_IO opera con
                \"%s\" non con \"%s\"\\n",nr,str,e.tv);
        exit(0);
    }
}
}
}

```



```

/*La funzione verifica serve per effettuare dei controlli semantici nel
type checking.
Questa funzione richiamata dalla funzione type_checking serve per
controllare che non ci sono problemi di type checking nell'istruzione
di assegnazione ovvero che tutte le variabili e i valori usati
nell'operazione di assegnazione devono essere dello stesso tipo.
La funzione verifica ha i seguenti parametri:
testo *punt è un puntatore alla lista testo contenente quel testo
necessario per i controlli semantici nel type checking e conterrà le
variabili a destra nell'operazione di assegnazione per cui effettuare i
controlli di type checking,
el_ST el è l'elemento della Symbol Table a sinistra nell'operazione di
assegnazione da confrontare con le variabili, i vettori e i valori
a destra dell'operazione di assegnazione per poter effettuare i controlli
semantici di type checking,
int nr indica il numero della riga alla quale occorre effettuare i
controlli semantici di type checking,
el_ST *s che è il puntatore alla Symbol Table,
char *funzione indica la funzione nella quale occorre effettuare i
controlli semantici di type checking,
char *str è uguale a NULL oppure contiene il nome di un vettore a destra in
una operazione di assegnazione e si verifica che il tipo del vettore è
coerente con il tipo della variabile o del vettore a sinistra
nell'operazione di assegnazione,
char *mn contiene il nome della funzione principale del programma ADA*/
void verifica(testo *punt,el_ST el,int nr,el_ST *s,char *funzione,char
*str,char *mn)
{

    char *v=numterminal(punt);
    if(v!=NULL)
    {
        if(strcmp(el.tv,v)!=0)
        {
            if(str!=NULL)
            {

```

```

printf( "\nERRORE SEMANTICO:errore alla riga %d il vettore \"%s\" è
        un vettore che non accetta valori \"%s\" accetta valori
        \"%s\"\\n",nr,str,v,el.tv);
exit(0);
}
else
{
printf( "\nERRORE SEMANTICO:errore alla riga %d ci si aspetta solo
        valori \"%s\" per modificare la variabile
        \"%s\"\\n",nr,el.tv,el.nameToken);
exit(0);
}
}
else
{
    if(strcmp(punt->tes,"+")!=0&&strcmp(punt->tes,"-")!=0&&strcmp(punt-
>tes,"*")!=0&&strcmp(punt->tes,"/")!=0&&strcmp(punt-
>tes,"(")!=0&&strcmp(punt->tes,",")!=0&&strcmp(punt->tes,")")!=0)
    {
        if(punt->tes[0]=='\'' )
        {
            if(strcmp(el.tv,"character")!=0)
            {
                if(str!=NULL)
                {
                    printf( "\nERRORE SEMANTICO:errore alla riga %d il vettore
                            \"%s\" è un vettore che non accetta valori \"character\"
                            accetta valori \"%s\"\\n",nr,str,el.tv);
                    exit(0);
                }
            }
            else
            {
                printf( "\nERRORE SEMANTICO:errore alla riga %d ci si aspetta
                        solo valori \"%s\" per modificare la variabile
                        \"%s\"\\n",nr,el.tv,el.nameToken);
            }
        }
    }
}

```

```

        exit(0);
    }
}
else
{
    if(str==NULL)
    {
        if(punt->next!=NULL)
        {
            printf( "\nERRORE SEMANTICO:errore alla riga %d operazione non
                    consentita\n",nr);
            exit(0);
        }
    }
}
else
{
    el_ST e=getsym(s,punt->tes,funzione);
    if(e.valh==-1)
    {
        printf( "\nERRORE SEMANTICO:errore alla riga %d tipo o
                variabile o funzione \"%s\" non definita\n",nr,punt->tes);
        exit(0);
    }
    if(strcmp(el.tv,e.tv)!=0)
    {
        if(e.t==1||e.t==2)
        {
            if(str!=NULL)
            {
                printf( "\nERRORE SEMANTICO:errore alla riga %d il vettore
                        \"%s\" è un vettore di \"%s\" mentre la variabile \"%s\"
                        è \"%s\"\n",nr,str,el.tv,e.nameToken,e.tv);
                exit(0);
            }
        }
    }
}

```

```

else
{
printf( "\nERRORE SEMANTICO:errore alla riga %d la variabile
        \"%s\" è \"%s\" mentre la variabile \"%s\" è
        \"%s\"\n",nr,el.nameToken,el.tv,e.nameToken,e.tv);
exit(0);
}
}
else
{
el_ST evet=getsym(s,e.tv,funzione);
if(evet.valh==-1)
{
evet=getsym(s,e.tv,mn);
if(evet.valh==-1)
{
printf("ERRORE SEMANTICO:errore alla riga %d variabile o tipo
        \"%s\" non definito\n",e.tv);
exit(0);
}
else
{
if(strcmp(el.tv,evet.tv)!=0)
{
printf( "\nERRORE SEMANTICO:errore alla riga %d i type
        \"%s\" e \"%s\" non sono dello stesso
        tipo\n",nr,el.nameToken,evet.nameToken);
exit(0);
}
}
}
else
{
if(strcmp(el.tv,evet.tv)!=0)
{
printf( "\nERRORE SEMANTICO:errore alla riga %d i type \"%s\"

```

```
e \"%s\" non sono dello stesso tipo\\n\",nr,  
el.nameToken,evet.nameToken);  
  
exit(0);  
  
}  
  
}  
  
}  
  
}  
  
}  
  
}
```

/*La funzione **type_checking** è una funzione per i controlli semantici di type checking.

La funzione **type_checking** ha i seguenti parametri:

testo *T che è un puntatore ad una lista testo e contiene il testo riconosciuto dal parser per la regola associato allo statement di assegnazione e che serve per controllare se l'istruzione di assegnazione è fatta in maniera coerente o no.

Questo parametro serve per controllare che non ci sono problemi di type checking nell'istruzione di assegnazione ovvero che tutte le variabili e i valori usati nell'operazione di assegnazione devono essere dello stesso tipo,

el_ST *s che è il puntatore alla Symbol Table,

char *funzione indica la funzione alla quale appartiene l'istruzione di assegnazione,

int nr indica il numero della riga alla quale è definita l'istruzione di assegnazione,

char *mn contiene il nome della funzione principale del programma ADA*/

```
void type_checking(testo *T,el_ST *s,char *funzione,int nr,char *mn)  
{  
    testo *punt=T;  
    el_ST el=context_check(s,punt->tes,funzione,mn,nr);  
    if(el.pf==1)  
    {  
        printf(" \\nERRORE SEMANTICO:errore alla riga %d la variabile \"%s\" non  
            può essere modificata poichè di modo \"in\"\\n\",nr,el.nameToken );
```

```

    exit(0);
}
if(el.t==5)
{
    printf( "\nERRORE SEMANTICO:errore alla riga %d variabile \"%s\" di
        tipo contatore non può essere modificata\n", nr,el.nameToken );
    exit(0);
}
if(el.t==4)
{
    int isvet=0;
    char *vet=(char *)strdup(punt->tes);
    testo *punta=punt;
    punt=punt->next;
    el_ST evet=context_check(s,vet,funzione,mn,nr);
    el_ST e=context_check(s,evet.tv,funzione,mn,nr);
    if(strcmp(punt->tes,":")==0)
    {
        punt=punt->next;
        int i=1;
        while(punt!=NULL)
        {
            if(strcmp(punt->tes,"+")==0||strcmp(punt->tes,"-")==0||strcmp(punt->tes,"*")==0||strcmp(punt->tes,"/")==0)
            {
                printf( "\nERRORE SEMANTICO:errore alla riga %d è possibile solo
                    un'operazione di aggregazione\n", nr);
                exit(0);
            }
            if(strcmp(punt->tes,"(")==0)
                punt=punt->next;
            if(strcmp(punt->tes,",")==0)
            {
                i++;
                if(i>e.dim_vet)
                {

```

```

    printf( "\nERRORE SEMANTICO:errore alla riga %d aggregazione
           fuori dal range del vettore \"%s\"\n", nr, evet.nameToken);
    exit(0);
}
else
    punt=punt->next;
}
if(strcmp(punt->tes,")")!=0)
    verifica(punt,e,nr,s,funzione,evet.nameToken,mn);
punt=punt->next;
}
}
else
{
    punt=punt->next;
    index_checking(e,nr,punt,evet,funzione,s);
    punt=punt->next->next->next;
    el_ST ee,eevet;
    while(punt!=NULL)
    {
        if(punt->next!=NULL)
        {
            if(isalpha(punt->tes[0])&&strcmp(punt->next->tes,"[")==0)
            {
                char *vet=(char *)strdup(punt->tes);
                punt=punt->next->next;
                eevet=context_check(s,vet,funzione,mn,nr);
                ee=context_check(s,eevet.tv,funzione,mn,nr);
                index_checking(ee,nr,punt,eevet,funzione,s);
                verifica(punta,ee,nr,s,funzione,evet.nameToken,mn);
                isvet=1;
            }
        }
        else
        {
            if(strcmp(punt->tes,",")==0)
            {

```

```

        printf( "\nERRORE SEMANTICO:errore alla riga %d operazione di
                aggregazione non consentita\n",nr);
        exit(0);
    }
    if(strcmp(punt->tes,"(")!=0&&strcmp(punt-
>tes,")")!=0&&strcmp(punt->tes,"[")!=0&&strcmp(punt-
>tes,"]")!=0&&strcmp(punt->tes,"+")!=0&&strcmp(punt->tes,"-
")!=0&&strcmp(punt->tes,"*")!=0&&strcmp(punt->tes,"/")!=0)
        verifica(punt,e,nr,s,funzione,evet.nameToken,mn);
    }
}
else
{
    if(strcmp(punt->tes,",")==0)
    {
        printf( "\nERRORE SEMANTICO:errore alla riga %d operazione di
                aggregazione non consentita\n",nr);
        exit(0);
    }
    if(strcmp(punt->tes,"(")!=0&&strcmp(punt->tes,")")!=0&&strcmp(punt-
>tes,"[")!=0&&strcmp(punt->tes,"]")!=0&&strcmp(punt-
>tes,"+")!=0&&strcmp(punt->tes,"-")!=0&&strcmp(punt-
>tes,"*")!=0&&strcmp(punt->tes,"/")!=0)
        verifica(punt,e,nr,s,funzione,evet.nameToken,mn);
    }
    punt=punt->next;
    if(isvet)
    {
        isvet=0;
        if(strcmp(e.tv,"character")==0)
        {
            if(punt->next!=NULL)
            {
                printf( "\nERRORE SEMANTICO:errore alla riga %d operazione non
                        consentita\n",nr);
                exit(0);
            }
        }
    }
}

```



```

        }
    }
}
}
}
}
}
if(el.t==2)
{
    printf( "\nERRORE SEMANTICO:errore alla riga %d il valore della
        variabile \"%s\" è costante non può essere modificato\n",nr,
        el.nameToken);
    exit(0);
}
if(el.t==3)
{
    printf( "\nERRORE SEMANTICO:errore alla riga %d identificatore \"%s\"
usato per la definizione di un nuovo tipo di dato \n",nr, el.nameToken);
    exit(0);
}
if(el.t==1)
{
    testo *punta=punt;
    punt=punt->next->next;
    int isvet=0;
    while (punt!=NULL)
    {
        if (punt->next!=NULL)
        {
            if (isalpha (punt->tes[0]) && (strcmp (punt->next-
>tes, "(")==0 || strcmp (punt->next->tes, "[")==0))
            {
                char *vet=(char *)strdup (punt->tes);
                punt=punt->next->next;
                el_ST evet=context_check(s,vet,funzione,mn,nr);
                el_ST e=context_check(s,evet.tv,funzione,mn,nr);
                index_checking(e,nr,punt,evet,funzione,s);
            }
        }
    }
}

```

```

    verifica(punta,e,nr,s,funzione,evet.nameToken,mn);
    isvet=1;
}
else
{
    if(strcmp(punt->tes,",")==0)
    {
        printf( "\nERRORE SEMANTICO:errore alla riga %d la variabile
            \"%s\" non è un vettore\n",nr,el.nameToken);
        exit(0);
    }
    if(strcmp(punt->tes,"(")!=0&&strcmp(punt->tes,")")!=0&&strcmp(punt-
>tes,"["!=0&&strcmp(punt->tes,"]")!=0&&strcmp(punt-
>tes,"+")!=0&&strcmp(punt->tes,"-")!=0&&strcmp(punt-
>tes,"*")!=0&&strcmp(punt->tes,"/")!=0)
        verifica(punt,el,nr,s,funzione,NULL,mn);
    }
}
else
{
    if(strcmp(punt->tes,",")==0)
    {
        printf( "\nERRORE SEMANTICO:errore alla riga %d la variabile \"%s\"
            non è un vettore\n",nr,el.nameToken);
        exit(0);
    }
    if(strcmp(punt->tes,"(")!=0&&strcmp(punt->tes,")")!=0&&strcmp(punt-
>tes,"["!=0&&strcmp(punt->tes,"]")!=0&&strcmp(punt-
>tes,"+")!=0&&strcmp(punt->tes,"-")!=0&&strcmp(punt-
>tes,"*")!=0&&strcmp(punt->tes,"/")!=0)
        verifica(punt,el,nr,s,funzione,NULL,mn);
    }
    punt=punt->next;
    if(isvet)
    {
        isvet=0;
    }
}

```


/*La funzione **conta_id** è una funzione usata nella traduzione ADA95-C.

Questa funzione conta le variabili definite in una dichiarazione

in ADA 95. Supponendo di avere la seguente dichiarazione:

a,b:integer;

La funzione **conta_id** conta le variabili definite che saranno due a e b e restituisce un int, il numero di variabili dichiarate, che sarà due in questo esempio.

La funzione **conta_id** ha due parametri:

testo *punt che è un puntatore alla lista **testo** contenente quel **testo** associato alla regola di definizione di una variabile riconosciuta dal parser,

int *trov è una variabile che se settata a 1 ci dice che sono stati trovati i due punti nella lista puntata da **punt** è che quindi

si sta operando con una istruzione di definizione di una variabile*/

```
int conta_id(testo *punt,int *trov)
{
    int i=1;
    while(*trov==0&&punt!=NULL)
    {
        if(strcmp(punt->tes,":")==0)
            *trov=1;
        else
        {
            if(strcmp(punt->tes,",")==0)
                i++;
            punt=punt->next;
        }
    }
    if(trov)
        return i;
    else
        return 0;
}
```

/*La funzione **trov_val** è una funzione usata nella traduzione ADA95-C. Questa funzione serve per vedere qual è il valore assegnato ad una variabile quando questa viene definita.

Ha un solo parametro: **testo *punt** che è un puntatore alla lista **testo** contenente quel testo associato alla regola di definizione di una variabile riconosciuta dal parser necessario per capire se viene assegnato un valore ad una variabile quando questa è definita.

La funzione **trov_val** è una funzione che restituisce un **char *** ovvero il valore assegnato alla variabile quando questa è definita*/

```
char * trov_val(testo *punt)
{
    int trov=0;
    char *str;
    while(trov==0&&punt!=NULL)
    {
        if(strcmp(punt->tes,":")==0)
            trov=1;
        punt=punt->next;
    }
    if(trov)
    {
        str=(char *)strdup(punt->tes);
        return str;
    }
    return NULL;
}
```

/*La funzione **iskeyword** è una funzione usata nella traduzione ADA95-C. Questa funzione serve per vedere se la prima parola nella lista è una keyword di ADA95 o no.

Ha un solo parametro: **testo *p** che è un puntatore alla lista **testo** contenente quella stringa per cui occorre verificare se è una parola chiave di ADA95 o no.

La funzione **iskeyword** restituisce un **int** che sarà 0 se la stringa da controllare non è una keyword di ADA mentre sarà 1 se la stringa da controllare è una keyword di ADA*/

```
int iskeyword(char *p)
{
    int i;
    int trov=0;
```

```

for(i=0;i<NUM_KEYWORDS;i++)
{
    if(strcmp(p,key_tab[i].kwM)==0)
        trov=1;
    if(strcmp(p,key_tab[i].kwm)==0)
        trov=1;
}
return trov;
}
/*La funzione val_neg è una funzione usata nella traduzione ADA95-C e serve
per la traduzione dei valori negativi dall'ADA 95 al C.
La funzione val_neg ha due parametri:
char *val che contiene il valore negativo da tradurre,
FILE *f che è il puntatore al file nel quale scrivere il valore negativo
tradotto.*/
void val_neg(char *val,FILE *f)
{
    char *s=(char *)malloc((strlen(val)-1)*sizeof(char));
    int j=0;
    int i;
    int trov=0;
    for(i=1;i<strlen(val)-1;i++)
    {
        s[j]=val[i];
        j++;
    }
    s[strlen(s)]='\0';
    i=0;
    while(i<strlen(s)&&!trov)
    {
        if(s[i]=='.')
            trov=1;
        i++;
    }
    fprintf(f,"(");
    if(!trov)

```

```

        fprintf(f,"%d",atoi(s));
    else
        fprintf(f,"%f",atof(s));
    fprintf(f,"");
}
/*La funzione traducipr è una funzione usata nella traduzione ADA95-C che
serve per la traduzioni delle chiamate alle procedure e alle function di
ADA 95.
La funzione traducipr ha due parametri:
testo *punt che contiene l'istruzione di chiamata ad una function o ad una
procedure da tradurre,
FILE *f che è il puntatore al file nel quale scrivere la chiamata ad una
function o ad una procedure tradotta.*/
void traducipr(testo *punt,FILE *f)
{
    fprintf(f,"%s",punt->tes);
    punt=punt->next;
    fprintf(f,"%s",punt->tes);
    punt=punt->next;
    if(strcmp(punt->tes,"")==0)
    {
        fprintf(f,"%s",punt->tes);
        fprintf(f,";\n");
    }
    else
    {
        fprintf(f,"%s",punt->tes);
        punt=punt->next;
        while(punt!=NULL)
        {
            if(strcmp(punt->tes,"")!=0)
            {
                fprintf(f,"");
                fprintf(f,"%s",punt->tes);
            }
            else

```

```

    {
        fprintf(f,"%s",punt->tes);
        fprintf(f,";\n");
    }
    punt=punt->next;
}
}
}
/*La funzione trovfc è una funzione usata nella traduzione ADA95-C e serve
per vedere se un valore è un integer,un float o un character.
La funzione trovfc ha tre parametri:
char *v contiene il valore da verificare,
int *tf ci dice se il valore è un float(*tf=1) o no(*tf=0),
int *tc ci dice se il valore è un character(*tc=1) o no(*tc=0).
Il valore da verificare è un integer se *tf=0 e anche *tc=0*/
void trovfc(char *v,int *tf,int *tc)
{
    int i=0,trov=0;
    while(i<strlen(v)&&!trov)
    {
        if(v[i]=='.')
            trov=1;
        i++;
    }
    if(trov)
        *tf=1;
    trov=0;
    i=0;
    while(i<strlen(v)&&!trov)
    {
        if(v[i]=='\\')
            trov=1;
        i++;
    }
    if(trov)
        *tc=1;

```



```

}
/*La funzione var_cost è una funzione usata nella traduzione ADA95-C e
serve per la traduzione delle operazioni di dichiarazione
delle variabili e delle costanti di ADA95 e di assegnazione delle
variabili.
La funzione var_cost ha i seguenti parametri:
testo *TT è un puntatore ad una lista testo contenente il testo
riconosciuto dal parser per le regole di dichiarazione delle variabili
e delle costanti e di assegnazione delle variabili,
el_ST e è un elemento della Symbol Table ed è il primo elemento nella lista
puntata da *TT per cui si deve fare l'operazione di traduzione,
FILE *f è il puntatore al file nel quale scrivere l'operazione di
dichiarazione e di assegnazione associate alle variabili e alle
costanti da tradurre,
el_ST *s è un puntatore alla Symbol Table,
char *funzione è un puntatore a caratteri che contiene il nome della
funzione in cui vi è l'operazione di dichiarazione e di assegnazione
di una variabile o di una costante da tradurre,
int v è 1 se si sta traducendo un vettore e 0 se si deve tradurre una
chiamata ad una funzione o ad una procedura,
int nr indica il numero della riga alla quale è definita l'istruzione di
dichiarazione delle variabili e delle costanti di ADA95 e di assegnazione
delle variabili.*/
void var_cost(testo *TT,el_ST e,FILE *f,el_ST *s,char *funzione,int v,int
nr)
{
    int i;
    int trov=0;
    int j;
    testo *punt=TT;
    i=conta_id(punt,&trov);
    if(trov)
    {
        if(strcmp(e.tv,"integer")==0||strcmp(e.tv,"INTEGER")==0)
        {
            if(e.t==1)

```

```

    fprintf(f,"int ");
    else
        fprintf(f,"const int ");
}
if(strcmp(e.tv,"character")==0||strcmp(e.tv,"CHARACTER")==0)
{
    if(e.t==1)
        fprintf(f,"char ");
    else
        fprintf(f,"const char ");
}
if(strcmp(e.tv,"float")==0||strcmp(e.tv,"FLOAT")==0)
{
    if(e.t==1)
        fprintf(f,"float ");
    else
        fprintf(f,"const float ");
}
trov=0;
char *val=trov_val(punt);
if(val==NULL)
{
    j=1;
    while(strcmp(punt->tes,":")!=0)
    {
        fprintf(f,"%s",punt->tes);
        if(j==i)
            fprintf(f,";\n");
        if(strcmp(punt->tes,",")==0)
            j++;
        punt=punt->next;
    }
}
else
{
    j=1;

```

```

int tf=0,tc=0;
while(strcmp(punt->tes,":")!=0)
{
if(strcmp(punt->tes,",")==0)
{
if(strcmp(e.tv,"integer")==0||strcmp(e.tv,"INTEGER")==0)
{
trovfc(val,&tf,&tc);
if(tf||tc)
{
printf( "\nERRORE SEMANTICO:errore alla riga %d la variabile
        \"%s\" accetta solo valori \"integer\"\n",nr,e.nameToken);
exit(0);
}
if(val[0]=='(' && val[strlen(val)-1]==')')
{
fprintf(f,"=");
val_neg(val,f);
}
else
fprintf(f,"%d",atoi(val));
}
if(strcmp(e.tv,"float")==0||strcmp(e.tv,"FLOAT")==0)
{
trovfc(val,&tf,&tc);
if(!tf||tc)
{
printf( "\nERRORE SEMANTICO:errore alla riga %d la variabile
        \"%s\" accetta solo valori \"float\"\n",nr,e.nameToken);
exit(0);
}
if(val[0]=='(' && val[strlen(val)-1]==')')
{
fprintf(f,"=");
val_neg(val,f);
}
}
}
}

```

```

else
    fprintf(f,"%f",atof(val));
}
if(strcmp(e.tv,"character")==0||strcmp(e.tv,"CHARACTER")==0)
{
    trovfc(val,&tf,&tc);
    if(!tc)
    {
        printf( "\nERRORE SEMANTICO:errore alla riga %d la variabile
        \"%s\" accetta solo valori \"character\\n\",nr,e.nameToken);
        exit(0);
    }
    if(val[0]=='(' && val[strlen(val)-1]==')')
    {
        fprintf(f,"=");
        val_neg(val,f);
    }
    else
        fprintf(f,"%s",val);
}
j++;
}
else
{
    fprintf(f,"%s",punt->tes);
    if(j==i)
    {
        if(strcmp(e.tv,"integer")==0||strcmp(e.tv,"INTEGER")==0)
        {
            trovfc(val,&tf,&tc);
            if(tf||tc)
            {
                printf( "\nERRORE SEMANTICO:errore alla riga %d la variabile
                \"%s\" accetta solo valori \"integer\\n\",nr,e.nameToken);
                exit(0);
            }

```

```

    if(val[0]==' ('&&val[strlen(val)-1]==')')
    {
        fprintf(f,"=");
        val_neg(val,f);
    }
    else
        fprintf(f,"%d",atoi(val));
}
if(strcmp(e.tv,"float")==0||strcmp(e.tv,"FLOAT")==0)
{
    trovfc(val,&tf,&tc);
    if(!tf||tc)
    {
        printf( "\nERRORE SEMANTICO:errore alla riga %d la variabile
                \"%s\" accetta solo valori \"float\"\n",nr,e.nameToken);
        exit(0);
    }
    if(val[0]==' ('&&val[strlen(val)-1]==')')
    {
        fprintf(f,"=");
        val_neg(val,f);
    }
    else
        fprintf(f,"%f",atof(val));
}
if(strcmp(e.tv,"character")==0||strcmp(e.tv,"CHARACTER")==0)
{
    trovfc(val,&tf,&tc);
    if(!tc)
    {
        printf( "\nERRORE SEMANTICO:errore alla riga %d la variabile
                \"%s\" accetta solo valori \"character\"\n",nr,e.nameToken);
        exit(0);
    }
    if(val[0]==' ('&&val[strlen(val)-1]==')')
    {

```

```

        fprintf(f, "=");
        val_neg(val, f);
    }
    else
        fprintf(f, "%s", val);
    }
    fprintf(f, ";\n");
}
}
punt=punt->next;
}
}
else
{
    punt=TT;
    int trovf=0;
    //operazione di assegnazione con espressione in genere
    while(punt!=NULL&&!trovf)
    {
        testo *pvar;
        if(strcmp(punt->tes, ":") != 0)
        {
            if(punt->tes[0] == '(' && punt->tes[strlen(punt->tes)-1] == ')')
                val_neg(punt->tes, f);
            else
            {
                if(strcmp(punt->tes, "(") == 0 && v == 1)
                    fprintf(f, "[");
                else
                {
                    if(strcmp(punt->tes, ")") == 0 && v == 1)
                    {
                        fprintf(f, "]");
                        v=0;
                    }
                }
            }
        }
    }
}

```

```

        else
            fprintf(f,"%s",punt->tes);

    }
}
}
else
{
    fprintf(f,"=");
    testo *punft=punt;
    puntf=punft->next;
    if(isalpha(punft->tes[0]))
    {
        el_ST ell=getsym(s,punft->tes,funzione);
        if(ell.t==6)
        {
            traducipr(punft,f);
            trovf=1;
        }
    }
    puntf=punft->next;
}
if(!trovf)
    fprintf(f,";\n");
}
}

```

/*La funzione **param_funz** è una funzione usata nella traduzione ADA95-C che serve per la traduzione dei parametri di una funzione o di una procedura di ADA95.

La funzione **param_funz** ha i seguenti parametri:

testo *punt che è un puntatore ad una lista testo e contiene il testo riconosciuto dal parser per la regola associato allo statement di definizione di una funzione o di una procedura,
el_ST *s che è il puntatore alla Symbol Table,

char *funzione contiene il nome della function o della procedure definita nel programma principale di ADA95,

FILE *f è il puntatore al file nel quale scrivere la definizione della funzione o della procedura,

char *mn contiene il nome della funzione principale del programma ADA nella quale viene definita una function o una procedure,

int nr indica il numero della riga alla quale è definita l'istruzione di definizione di una funzione o di una procedura di ADA95.

La funzione `param_funz` restituisce un testo * ovvero un puntatore alla lista testo che contiene il testo riconosciuto dal parser per la regola associato allo statement di definizione di una funzione o di una procedura il cui indirizzo sarà diverso da quello di `punt.` */

```
testo * param_funz(testo *punt,el_ST *s,char *funzione,FILE *f,char *mn,int
nr)
{
    el_ST el=getsym(s,punt->tes,funzione);
    char *tp;
    if(el.t==4)
        el=context_check(s,el.tv,funzione,mn,nr);
    if(strcmp(el.tv,"integer")==0||strcmp(el.tv,"INTEGER")==0)
    {
        if(el.t==1)
        {
            fprintf(f,"int ");
            tp=(char *)strdup("int");
        }
        else
        {
            if(el.t==2)
            {
                fprintf(f,"const int ");
                tp=(char *)strdup("const int");
            }
            else
            {
                fprintf(f,"int *");
            }
        }
    }
}
```



```

        tp=(char *)strdup("int *");
    }
}
}
if(strcmp(el.tv,"character")==0||strcmp(el.tv,"CHARACTER")==0)
{
    if(el.t==1)
    {
        fprintf(f,"char ");
        tp=(char *)strdup("char");
    }
    else
    {
        if(el.t==2)
        {
            fprintf(f,"const char ");
            tp=(char *)strdup("const char");
        }
        else
        {
            fprintf(f,"char *");
            tp=(char *)strdup("char *");
        }
    }
}
}
if(strcmp(el.tv,"float")==0||strcmp(el.tv,"FLOAT")==0)
{
    if(el.t==1)
    {
        fprintf(f,"float ");
        tp=(char *)strdup("float");
    }
    else
    {
        if(el.t==2)
        {

```

```

    fprintf(f,"const float ");
    tp=(char *)strdup("const float");
}
else
{
    fprintf(f,"float *");
    tp=(char *)strdup("float *");
}
}
}
while(strcmp(punt->tes,":")!=0)
{
    fprintf(f,"%s",punt->tes);
    if(strcmp(punt->tes,",")==0)
        fprintf(f,"%s ",tp);
    punt=punt->next;
}
return punt;
}
/*La funzione aggrcheck serve per verificare se in una operazione di
aggregazione associata ad un vettore, al vettore vengono assegnati
valori coerenti con il suo tipo di array precedentemente definito nel
programma ADA95.
La funzione aggrcheck ha i seguenti parametri:
el_ST e elemento della Symbol Table che rappresenta il tipo di array
associato al vettore per cui si effettua l'operazione di aggregazione,
el_ST el elemento della Symbol Table che rappresenta il vettore per cui si
effettua l'operazione di aggregazione,
int nr è indica il numero della riga alla quale si effettua l'operazione
di aggregazione,
int tf è 1 se l'elemento associato al vettore nell'operazione di
aggregazione è un float altrimenti tf è 0,
int tc è 1 se l'elemento associato al vettore nell'operazione di
aggregazione è un character altrimenti tc è 0*/
void aggrcheck(el_ST e,el_ST el,int nr,int tf,int tc)
{

```

```

if(strcmp(e.tv,"integer")==0)
{
    if(tf||tc)
    {
        printf( "\nERRORE SEMANTICO:errore alla riga %d il vettore \"%s\"
                accetta solo valori \"integer\"\\n",nr,el.nameToken);
        exit(0);
    }
}
if(strcmp(e.tv,"float")==0)
{
    if(!tf||tc)
    {
        printf( "\nERRORE SEMANTICO:errore alla riga %d il vettore \"%s\"
                accetta solo valori \"float\"\\n",nr,el.nameToken);
        exit(0);
    }
}
if(strcmp(e.tv,"character")==0)
{
    if(!tc)
    {
        printf( "\nERRORE SEMANTICO:errore alla riga %d il vettore \"%s\"
                accetta solo valori \"character\"\\n",nr,el.nameToken);
        exit(0);
    }
}
}
/*La funzione analisi_trad è la funzione principale per la traduzione
ADA95-C.

```

La funzione **analisi_trad** ha i seguenti parametri:

testo *TT è un puntatore alla lista testo contenente il testo da tradurre,

el_ST *s è un puntatore alla Symbol Table,

FILE *f è il puntatore al file nel quale effettuare la traduzione,

int a è 1 se si sta effettuando la traduzione di un'operazione di
 aggregazione altrimenti è 0,

```

int v è 1 se si sta traducendo un vettore e 0 se si deve tradurre una
chiamata ad una funzione o ad una procedura,
char *funzione contiene il nome della function o della procedure definita
nel programma principale di ADA95 a cui è associato il testo da tradurre,
int nr indica il numero della riga alla quale si trova l'istruzione
da tradurre,
char *mn contiene il nome della funzione principale del programma ADA.*/
void analisi_trad(testo *TT,el_ST *s,FILE *f,int a,int v,char *funzione,int
nr,char *mn)
{
    testo *punt=TT;
    int i;
    int j;
    int trov=0;
    //stampa_testo(*TT);
    int kw=iskeyword(punt->tes);
    el_ST el;
    if(kw==0)
    {
        if(strcmp(punt->tes,"printf")==0)
        {
            fprintf(f,"%s",punt->tes);
            punt=punt->next;
            fprintf(f,"%s",punt->tes);
            punt=punt->next;
            el=getsym(s,punt->tes,funzione);
            int car=37;
            if(strcmp(el.tv,"integer")==0||strcmp(el.tv,"INTEGER")==0)
                fprintf(f,"%cd\\",",",car);
            else
                if(strcmp(el.tv,"character")==0||strcmp(el.tv,"CHARACTER")==0)
                    fprintf(f,"%cc\\",",",car);
            else
                if(strcmp(el.tv,"float")==0||strcmp(el.tv,"FLOAT")==0)
                    fprintf(f,"%cf\\",",",car);
            else

```

```

{
    el=getsym(s,el.tv,funzione);
    if(strcmp(el.tv,"integer")==0||strcmp(el.tv,"INTEGER")==0)
        fprintf(f,"%cd\\",",",car);
    if(strcmp(el.tv,"character")==0||strcmp(el.tv,"CHARACTER")==0)
        fprintf(f,"%cc\\",",",car);
    if(strcmp(el.tv,"float")==0||strcmp(el.tv,"FLOAT")==0)
        fprintf(f,"%cf\\",",",car);
    fprintf(f,"%s",punt->tes);
    punt=punt->next;
    if(strcmp(punt->tes,"[")==0)
    {
        fprintf(f,"%s",punt->tes);
        punt=punt->next;
        fprintf(f,"%s",punt->tes);
        punt=punt->next;
    }
    else
    {
        printf("ERRORE SEMANTICO:impossibile stampare la variabile
            indicata");
        exit(0);
    }
}

fprintf(f,"%s",punt->tes);
fprintf(f,"");\n");
}

else
{
    if(strcmp(punt->tes,"scanf")==0)
    {
        fprintf(f,"%s",punt->tes);
        punt=punt->next;
        fprintf(f,"%s",punt->tes);
        punt=punt->next;
        el=getsym(s,punt->tes,funzione);
    }
}

```

```

int car=37;
if(strcmp(el.tv,"integer")==0||strcmp(el.tv,"INTEGER")==0)
{
    fprintf(f,"%cd\\",",",car);
    fprintf(f,"%&");
}
else
    if(strcmp(el.tv,"character")==0||strcmp(el.tv,"CHARACTER")==0)
    {
        fprintf(f,"%cc\\",",",car);
        fprintf(f,"%&");
    }
else
    if(strcmp(el.tv,"float")==0||strcmp(el.tv,"FLOAT")==0)
    {
        fprintf(f,"%cf\\",",",car);
        fprintf(f,"%&");
    }
else
    {
        fprintf(f,"%&");
        fprintf(f,"%s",punt->tes);
        punt=punt->next;
        if(strcmp(punt->tes,"[")==0)
        {
            fprintf(f,"%s",punt->tes);
            punt=punt->next;
            fprintf(f,"%s",punt->tes);
            punt=punt->next;
        }
        else
        {
            printf("ERRORE SEMANTICO:impossibile effettuare tale
                acquisizione");
            exit(0);
        }
    }

```

```

    }
    fprintf(f,"%s",punt->tes);
    fprintf(f,");\n");
}
else
{
    el=getsym(s,punt->tes,funzione);
    switch(el.t)
    {
        case 1:
            var_cost(punt,el,f,s,funzione,v,nr);
            break;
        case 2:
            var_cost(punt,el,f,s,funzione,v,nr);
            break;
        case 6:
            traducipr(punt,f);
            break;
        case 4:
            i=conta_id(punt,&trov);
            el_ST e;
            if(trov)
            {
                int ch=0;
                e=getsym(s,el.tv,funzione);
                if(strcmp(e.tv,"integer")==0||strcmp(e.tv,"INTEGER")==0)
                    fprintf(f,"int ");
                if(strcmp(e.tv,"character")==0||strcmp(e.tv,"CHARACTER")==0)
                {
                    fprintf(f,"char ");
                    ch=1;
                }
                if(strcmp(e.tv,"float")==0||strcmp(e.tv,"FLOAT")==0)
                    fprintf(f,"float ");
                punt=TT;
                j=0;
            }
    }
}

```

```

char *val=trov_val(punt);
if(val==NULL)
{
    while(strcmp(punt->tes,":")!=0)
    {
        if(ch)
        {
            if(strcmp(punt->tes,",")!=0)
            {
                fprintf(f,"%s[%d]",punt->tes,e.dim_vet+1);
                j++;
            }
            else
                fprintf(f,",");
        }
        else
        {
            if(strcmp(punt->tes,",")!=0)
            {
                fprintf(f,"%s[%d]",punt->tes,e.dim_vet);
                j++;
            }
            else
                fprintf(f,",");
        }
        if(j==i)
            fprintf(f,";\n");
        punt=punt->next;
    }
}
else
{
    int tf=0,tc=0;
    int i=1;
    testo *punt_a=TT;
    while(strcmp(punt_a->tes,":")!=0)

```



```

    punt_a=punt_a->next;
punt_a=punt_a->next;
while(strcmp(punt->tes,":")!=0)
{
    if(ch)
    {
        if(strcmp(punt->tes,",")!=0)
        {
            fprintf(f,"%s[%d]",punt->tes,e.dim_vet+1);
            j++;
        }
        else
        {
            fprintf(f,"=");
            while(punt_a!=NULL)
            {
                if(strcmp(punt_a->tes,",")!=0&&strcmp(punt_a-
>tes,"{")!=0&&strcmp(punt_a->tes,"}")!=0)
                {
                    tf=0;
                    tc=0;
                    trovfc(punt_a->tes,&tf,&tc);
                    aggrcheck(e,el,nr,tf,tc);
                }
                printf("%s\n",punt_a->tes);
                if(strcmp(punt_a->tes,",")==0)
                {
                    i++;
                    if(i>e.dim_vet)
                    {
                        printf( "\nERRORE SEMANTICO:errore alla riga %d
                        aggregazione fuori dal range del vettore \"%s\"\\n",
                        nr,el.nameToken);
                        exit(0);
                    }
                }
            }

```

```

    fprintf(f,"%s",punt_a->tes);
    punt_a=punt_a->next;

}
fprintf(f,",");
if(strcmp(punt_a->tes,",")==0)
{
    i++;
    if(i>e.dim_vet)
    {
        printf( "\nERRORE SEMANTICO:errore alla riga %d
                aggregazione fuori dal range del vettore \"%s\\\"\\n",
                nr,el.nameToken);
        exit(0);
    }
}
}
}
else
{
    if(strcmp(punt->tes,",")!=0)
    {
        fprintf(f,"%s[%d]",punt->tes,e.dim_vet);
        j++;
    }
    else
    {
        fprintf(f,"=");
        while(punt_a!=NULL)
        {
            if(strcmp(punt_a->tes,",")!=0&&strcmp(punt_a->tes,"{")
!=0&&strcmp(punt_a->tes,"}")!=0)
            {
                tf=0;
                tc=0;
                trovfc(punt_a->tes,&tf,&tc);

```

```

        aggrcheck(e,el,nr,tf,tc);
    }
    printf("%s\n",punt_a->tes);
    if(strcmp(punt_a->tes,",")==0)
    {
        i++;
        if(i>e.dim_vet)
        {
            printf( "\nERRORE SEMANTICO:errore alla riga %d
                aggregazione fuori dal range del vettore \"%s\"\\n",
                nr,el.nameToken);
            exit(0);
        }
    }
    fprintf(f,"%s",punt_a->tes);
    punt_a=punt_a->next;

}
fprintf(f,",");
if(strcmp(punt_a->tes,",")==0)
{
    printf("%d\\n",i);
    i++;
    if(i>e.dim_vet)
    {
        printf( "\nERRORE SEMANTICO:errore alla riga %d
            aggregazione fuori dal range del vettore \"%s\"\\n",
            nr,el.nameToken);
        exit(0);
    }
}
}
}
if(j==i)
{
    fprintf(f,"=");

```

```

while (punt_a!=NULL)
{
    if (strcmp (punt_a->tes, ",") !=0 && strcmp (punt_a-
>tes, "{") !=0 && strcmp (punt_a->tes, "}") !=0)
    {
        tf=0;
        tc=0;
        trovfc (punt_a->tes, &tf, &tc);
        aggrcheck (e, el, nr, tf, tc);
    }
    if (strcmp (punt_a->tes, ",") ==0)
    {
        i++;
        if (i>e.dim_vet)
        {
            printf( "\nERRORE SEMANTICO:errore alla riga %d
                aggregazione fuori dal range del vettore \"%s\"\\n",
                nr, el.nameToken);
            exit(0);
        }
    }
    fprintf(f, "%s", punt_a->tes);
    punt_a=punt_a->next;
}
fprintf(f, ";\\n");
}
punt=punt->next;
}
}
else
{
    punt=TT;
    //operazione di assegnazione con espressione in genere
    while (punt!=NULL)
    {

```

```

if (strcmp (punt->tes, "!=") != 0)
{
    if (strcmp (punt->tes, "(") == 0 && a == 1)
        fprintf (f, "{");
    else
    {
        if (strcmp (punt->tes, ")") == 0 && a == 1)
            fprintf (f, "}");
        else
        {
            if (strcmp (punt->tes, "(") == 0 && v == 1)
                fprintf (f, "[");
            else
            {
                if (strcmp (punt->tes, ")") == 0 && v == 1)
                {
                    fprintf (f, "]");
                    v = 0;
                }
                else
                {
                    if (punt->tes[0] == '(' && punt->tes[strlen (punt->tes) -
1] == ')')

                        val_neg (punt->tes, f);
                    else
                        fprintf (f, "%s", punt->tes);
                }
            }
        }
    }
    else
        fprintf (f, "=");
    punt = punt->next;
}
fprintf (f, ";\n");

```

```

        }
        break;
    }
}
}
else
{
    if (strcmp (punt->tes, "if")==0)
    {
        while (punt!=NULL)
        {
            if (strcmp (punt->tes, "then")==0)
            {
                if (punt->next==NULL)
                    fprintf (f, " ");
            }
            else
            {
                if (strcmp (punt->tes, "if")==0)
                    fprintf (f, "%s(", punt->tes);
                else
                {
                    if (strcmp (punt->tes, "/"==0)
                        fprintf (f, "!=");
                    else
                    {
                        if (strcmp (punt->tes, "=")==0)
                            fprintf (f, "==" );
                        else
                            fprintf (f, "%s", punt->tes);
                    }
                }
            }
        }
        punt=punt->next;
    }
}

```

```

}
else
{
    if (strcmp (punt->tes, "for") == 0)
    {
        fprintf (f, "%s(", punt->tes);
        punt = punt->next;
        char *id = (char *) strdup (punt->tes);
        fprintf (f, "%s=", id);
        punt = punt->next;
        fprintf (f, "%d;%s<=", atoi (punt->tes), id);
        punt = punt->next;
        punt = punt->next; // perchè considero ..
        fprintf (f, "%d;%s++", atoi (punt->tes), id);
        fprintf (f, ")\n");
    }
    else
    {
        if (strcmp (punt->tes, "while") == 0)
        {
            while (punt != NULL)
            {
                if (strcmp (punt->tes, "loop") == 0)
                    fprintf (f, ")\n");
                else
                {
                    if (strcmp (punt->tes, "then") != 0)
                    {
                        if (strcmp (punt->tes, "while") == 0)
                            fprintf (f, "%s(", punt->tes);
                        else
                        {
                            if (strcmp (punt->tes, "/=") == 0)
                                fprintf (f, "!=");
                            else
                                {

```

```

        if (strcmp (punt->tes, "=") == 0)
            fprintf (f, "=");
        else
            fprintf (f, "%s", punt->tes);
    }
}
}
}
punt=punt->next;
}
}
else
{
    if (strcmp (punt->tes, "procedure") == 0)
    {
        fprintf (f, "void ");
        punt=punt->next;
        fprintf (f, "%s", punt->tes);
        fprintf (f, "(");
        punt=punt->next;
        while (strcmp (punt->tes, "is") != 0)
        {
            punt=param_funz (punt, s, funzione, f, mn, nr);
            punt=punt->next;
            punt=punt->next;
            if (strcmp (punt->tes, "is") != 0)
                fprintf (f, ",");
        }
        fprintf (f, ")");
        fprintf (f, "\n{\n");
    }
}
else
{
    if (strcmp (punt->tes, "function") == 0)
    {
        while (strcmp (punt->tes, "return") != 0)

```



```

punt=punt->next;
punt=punt->next;
if(strcmp(punt->tes,"integer")==0)
    fprintf(f,"int ");
else
    if(strcmp(punt->tes,"character")==0)
        fprintf(f,"char ");
    else
        if(strcmp(punt->tes,"float")==0)
            fprintf(f,"float ");
        else
            {
                el=getsym(s,punt->tes,funzione);
                if(strcmp(el.tv,"integer")==0||strcmp(el.tv,"INTEGER")==0)
                    fprintf(f,"int * ");
            }

if(strcmp(el.tv,"character")==0||strcmp(el.tv,"CHARACTER")==0)
    fprintf(f,"char * ");
if(strcmp(el.tv,"float")==0||strcmp(el.tv,"FLOAT")==0)
    fprintf(f,"float * ");
}
punt=TT;
punt=punt->next;
fprintf(f,"%s",punt->tes);
fprintf(f,"(");
punt=punt->next;
while(strcmp(punt->tes,"return")!=0)
{
    punt=param_funz(punt,s,funzione,f,mn,nr);
    punt=punt->next;
    punt=punt->next;
    if(strcmp(punt->tes,"return")!=0)
        fprintf(f,",");
}
fprintf(f,")");
fprintf(f,"\n{\n");

```

```
    }  
    else  
    {  
        if (strcmp(punt->tes, "return")==0)  
        {  
            punt=punt->next;  
            fprintf(f, "return ");  
            fprintf(f, "%s", punt->tes);  
            fprintf(f, ";\n");  
        }  
    }  
}  
}  
}  
}  
}
```

Casi di test ed errori

Per testare il compilatore “*adac*” si sono utilizzati tre programmi in ADA 95, che sono i seguenti:

1. *function_parametri.adb*
2. *vettore.adb*
3. *calcola_media.adb*

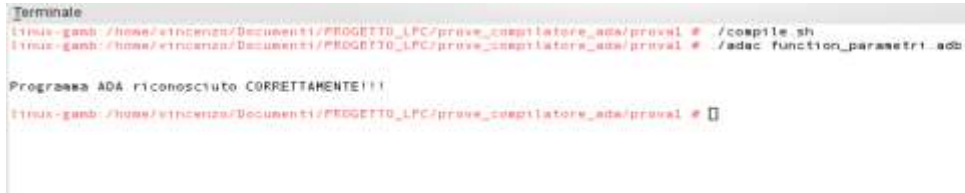
1. *function_parametri.adb*

```

1  with Ada.Text_IO, Ada.Integer_Text_IO;
2
3  procedure Proc_param is
4      cani,gatti,animali:integer;
5
6
7
8      function Numero_Totale_Animali(speciel:integer;specie2:integer;) return integer is
9          totale:integer;
10
11      begin
12
13          totale := speciel + specie2;
14          return totale;
15      end Numero_Totale_Animali;
16  begin
17      cani := 3;
18      gatti := 4;
19      animali:=Numero_Totale_Animali(cani,gatti);
20      Ada.Integer_Text_IO.Put (animali);
21  end Proc_param;
```

1.1 Errori sintattici

- Programma ADA riconosciuto correttamente



```

Terminale
linux-gamb: /home/vincenzo/Documents/PROGETTO_LFC/prove_compilerare_ada/prova1 # /compile.sh
linux-gamb: /home/vincenzo/Documents/PROGETTO_LFC/prove_compilerare_ada/prova1 # /adac function_parametri.adb

Programma ADA riconosciuto CORRETTAMENTE!!!
linux-gamb: /home/vincenzo/Documents/PROGETTO_LFC/prove_compilerare_ada/prova1 # 

```

Di seguito è inserito il file function_parametri.c la traduzione in C del file

function_parametri.adb:

```

#include<stdio.h>
int Numero_Totale_Animali(int specie1,int specie2)
{

//parte dichiarativa
int totale;

//parte esecutiva
totale=specie1+specie2;
return totale;
}
void main()
{

//parte dichiarativa
int cani,gatti,animali;

//parte esecutiva
cani=3;
gatti=4;
animali=Numero_Totale_Animali(cani,gatti);
printf("%d",animali);

}

```

- riga 1: si elimina il “with”, si ha il seguente errore



```

Terminale
linux-gamb: /home/vincenzo/Documents/PROGETTO_LFC/prove_compilerare_ada/prova1 # /compile.sh
linux-gamb: /home/vincenzo/Documents/PROGETTO_LFC/prove_compilerare_ada/prova1 # /adac function_parametri.adb
ERRORE SINTATTICO: errore alla riga 1 syntax error, unexpected TEXT, expecting $end or WITH or PROCEDURE or ID
linux-gamb: /home/vincenzo/Documents/PROGETTO_LFC/prove_compilerare_ada/prova1 # 

```

- riga 3: si elimina la keyword “procedure”, si ha il seguente errore

```

Terminale
linux-gaeb: /home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 # ./compile.sh
linux-gaeb: /home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 # ./adac function_parametri.adb
errore sintattico: errore alla riga numero 3 keyword PROCEDURE mancante
linux-gaeb: /home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 # █

```

- riga 3: si elimina il nome della procedura principale (Proc_param) del programma “function_parametri.adb” e si ha il seguente errore

```

Terminale
linux-gaeb: /home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 # ./compile.sh
linux-gaeb: /home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 # ./adac function_parametri.adb
ERRORE SINTATTICO: errore alla riga numero 3: o NOME DELLA PROCEDURA mancante
linux-gaeb: /home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 # █

```

- riga 8: si elimina la keyword “function”, si ha il seguente errore

```

Terminale
linux-gaeb: /home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 # ./compile.sh
linux-gaeb: /home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 # ./adac function_parametri.adb
ERRORE SINTATTICO: errore alla riga numero 8: o keyword "function" "procedure" mancanti oppure o begin mancante
linux-gaeb: /home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 # █

```

- riga 8: si elimina la keyword “return”, si ha il seguente errore

```

Terminale
linux-gaeb: /home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 # ./compile.sh
linux-gaeb: /home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 # ./adac function_parametri.adb
ERRORE SINTATTICO: errore alla riga numero 8 keyword RETURN mancante
linux-gaeb: /home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 # █

```

- riga 15: si elimina la keyword “end”, si ha il seguente errore

```

Terminale
linux-gaeb: /home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 # ./compile.sh
linux-gaeb: /home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 # ./adac function_parametri.adb
ERRORE SINTATTICO: errore alla riga numero 15 END mancante
linux-gaeb: /home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 # █

```

- riga 21: omettendo il nome della procedura principale (Proc_param) del programma in input ad adac, si ha il seguente errore

```

Terminale
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 # ./compile.sh
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 # ./adac function_parametri.adb
errore sintattico: errore alla riga numero 21 NONE PROCEDURA mancante
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 #

```

1.2 Errori semantici

- riga 13: “specie2” non definita, non avendo previsto tale parametro formale nella dichiarazione della funzione “Numero_Totale_Animali” si ha il seguente errore

```

Terminale
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 # ./compile.sh
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 # ./adac function_parametri.adb
ERRORE SEMANTICO:errore alla riga 13 tipo o variabile o funzione "specie2" non definita
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 #

```

- riga 14: si dichiara alla riga 9 la variabile “totale” come float e si ha il seguente errore

```

Terminale
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 # ./compile.sh
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 # ./adac function_parametri.adb
ERRORE SEMANTICO:errore alla riga 14 il tipo restituito dalla funzione "Numero_Totale_Animali" non è di tipo "float"
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 #

```

- riga 17: alla variabile “cani” si è assegnato un valore di tipo float, in contraddizione con il fatto che è stata dichiarata come integer quindi si ha il seguente errore

```

Terminale
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 # ./compile.sh
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 # ./adac function_parametri.adb
ERRORE SEMANTICO:errore alla riga 17 ci si aspetta solo valori "integer" per modificare la variabile "cani"
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compiler_ada/prova1 #

```

- riga19: eliminando un parametro attuale della funzione “Numero_Totale_Animali”, quando questa viene richiamata dalla procedura principale del programma in input ad adac si ha il seguente errore

```
Terminale
linux-gamb: /home/vincenzo/Documents/PROGETTO_LFC/prova_compilatore_ada/prova1 # ./compile.sh
linux-gamb: /home/vincenzo/Documents/PROGETTO_LFC/prova_compilatore_ada/prova1 # ./adac function_parametri.adb
ERRORE SEMANTICO: errore alla riga 19 la funzione "Numero_Totale_Animali" prevede 2 parametri
linux-gamb: /home/vincenzo/Documents/PROGETTO_LFC/prova_compilatore_ada/prova1 #
```

- riga 21: scrivendo in maniera errata il nome della procedura principale del programma in input ad adac si ha il seguente errore

```
Terminale
linux-gamb: /home/vincenzo/Documents/PROGETTO_LFC/prova_compilatore_ada/prova1 # ./compile.sh
linux-gamb: /home/vincenzo/Documents/PROGETTO_LFC/prova_compilatore_ada/prova1 # ./adac function_parametri.adb
ERRORE SEMANTICO: errore alla riga numero 21 NOME FUNZIONE non corretto
linux-gamb: /home/vincenzo/Documents/PROGETTO_LFC/prova_compilatore_ada/prova1 #
```

2. vettore.adb

```
1  with Ada.Text_IO, Ada.Integer_Text_IO;
2
3
4  procedure Array1 is
5
6      i:integer;
7      type MY_ARRAY is array (0..4) of integer;
8      totale:MY_ARRAY;
9      primo:MY_ARRAY;
10     secondo:MY_ARRAY;
11
12     begin
13         primo(0) := 1;
14         primo(1) := 12;
15         primo(2) := 16;
16         primo(3) := primo(2) - primo(1);
17         primo(4) := 16 - 2 * primo(2);
18         i:=0;
19         while i<5 loop
20             secondo(i) := 3 * i + 77;
21             i:=i+1;
22         end loop;
23         i:=0;
24         while i<5 loop
25             totale(i) := primo(i) + secondo(i);
26             Ada.Integer_Text_IO.Put(totale(i));
27             i:=i+1;
28         end loop;
29     end Array1;
```


2.1 Errori sintattici

- Programma ADA riconosciuto correttamente



```

Terminale
linux-gasb: /home/vincenzo/Documents/PROGETTO_LFC/prova_compilatore_ada/prova1 # ./compile.sh
linux-gasb: /home/vincenzo/Documents/PROGETTO_LFC/prova_compilatore_ada/prova1 # ./adac vettore.adb

Programma ADA riconosciuto CORRETTAMENTE!!!

linux-gasb: /home/vincenzo/Documents/PROGETTO_LFC/prova_compilatore_ada/prova1 #
  
```

Di seguito è inserito il file vettore.c la traduzione in C del file vettore.adb:

```

#include<stdio.h>
void main()
{

//parte dichiarativa
int totale[5];
int primo[5];
int secondo[5];
int i;

//parte esecutiva
primo[0]=1;
primo[1]=12;
primo[2]=16;
primo[3]=primo[2]-primo[1];
primo[4]=16-2*primo[2];
i=0;
while(i<5)
{
secondo[i]=3*i+77;
i=i+1;

}
i=0;
while(i<5)
{
totale[i]=primo[i]+secondo[i];
printf("%d",totale[i]);
i=i+1;

}

}
  
```

- riga7: si elimina la keyword “type” e si ha il seguente errore

```

Terminale
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compilatore_ada/prova1 # ./compile.sh
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compilatore_ada/prova1 # ./adac vettore.adb
ERRORE SINTATTICO: errore alla riga numero 7 keyword type mancante
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compilatore_ada/prova1 # 

```

- riga 7: estremo inferiore del range del type array “MY_ARRAY” mancante nella sua dichiarazione

```

Terminale
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compilatore_ada/prova1 # ./adac vettore.adb
ERRORE SINTATTICO: errore alla riga 7 estremo inferiore del range mancante
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compilatore_ada/prova1 # 

```

- riga 7: si elimina la keyword “of” e si ha il seguente errore

```

Terminale
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compilatore_ada/prova1 # ./compile.sh
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compilatore_ada/prova1 # ./adac vettore.adb
errore sintattico: errore alla riga numero 7 keyword of mancante
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compilatore_ada/prova1 # 

```

- riga 15: mancanza di una parentesi tonda chiusa

```

Terminale
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compilatore_ada/prova1 # ./compile.sh
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compilatore_ada/prova1 # ./adac vettore.adb
ERRORE SINTATTICO: errore alla riga 15 syntax error, unexpected MENO, expecting TONDACHIUSA
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compilatore_ada/prova1 # 

```

2.2 Errori semantici

- riga 13: si è assegnato al vettore “primo” un valore float, in contraddizione col fatto che è stato dichiarato come vettore di interi quindi si ha il seguente errore

```
Terminale
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compilatore_ada/prova1 # ./compile.sh
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compilatore_ada/prova1 # ./adac vettore.adb
ERRORE SEMANTICO: errore alla riga 13 il vettore "primo" è un vettore che non accetta valori "float" accetta valori "integer"
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compilatore_ada/prova1 #
```

- riga 29: nome della procedura “Array1” non scritta correttamente

```
Terminale
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compilatore_ada/prova1 # ./compile.sh
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compilatore_ada/prova1 # ./adac vettore.adb
ERRORE SEMANTICO: errore alla riga numero 29 NOME FUNZIONE non corretto
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compilatore_ada/prova1 #
```

3. calcola_media.adb

```

1   with Ada.Text_IO,Ada.Float_Text_IO;
2
3
4   procedure CalcolaMedia is
5
6       numero1, numero2, tot : float;
7
8       function ValoreMedio(x, y : float;) return float is
9           media:float;
10        begin
11
12
13            media:= (x + y)/2.0;
14            return media;
15        end ValoreMedio;
16    begin
17        Ada.Float_Text_IO.Get(numero1);
18        Ada.Float_Text_IO.Get(numero2);
19        tot := ValoreMedio(numero1,numero2);
20        Ada.Float_Text_IO.Put(tot);
21    end CalcolaMedia;
```

3.1 Errori sintattici

- Programma ADA riconosciuto correttamente

```

Terminale
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compilatore_ada/prova1 # ./compile.sh
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compilatore_ada/prova1 # ./adac calcola_media.adb

Programma ADA riconosciuto CORRETTAMENTE!!!
linux-gamb:/home/vincenzo/Documents/PROGETTO_LFC/prove_compilatore_ada/prova1 # █
```

Di seguito è inserito il file `calcola_media.c` la traduzione in C del file `calcola_media.adb`:

```
#include<stdio.h>
float ValoreMedio(float x,float y)
{
float media;
media=(x+y)/2.000000;
return media;
}
void main()
{

//parte dichiarativa
float numero1,numero2,tot;

//parte esecutiva
scanf("%f",&numero1);
scanf("%f",&numero2);
tot=ValoreMedio(numero1,numero2);
printf("%f",tot);

}
```

- riga 15: se non si inserisce il nome della funzione “ValoreMedio” si ha il seguente errore



```
Terminale
linux-gaab: /home/vincenzo/Documents/PROGETTO_LFC/prove_compileratore_ada/prova1 # ./compile.sh
linux-gaab: /home/vincenzo/Documents/PROGETTO_LFC/prove_compileratore_ada/prova1 # ./adac calcola_media.adb
ERRORE SINTATTICO: errore alla riga numero 15 NOME FUNZIONE mancante
linux-gaab: /home/vincenzo/Documents/PROGETTO_LFC/prove_compileratore_ada/prova1 #
```

- riga 21: se manca la keyword “end” si ha il seguente errore



```
Terminale
linux-gaab: /home/vincenzo/Documents/PROGETTO_LFC/prove_compileratore_ada/prova1 # ./compile.sh
linux-gaab: /home/vincenzo/Documents/PROGETTO_LFC/prove_compileratore_ada/prova1 # ./adac calcola_media.adb
ERRORE SINTATTICO: errore alla riga numero 21 END mancante
linux-gaab: /home/vincenzo/Documents/PROGETTO_LFC/prove_compileratore_ada/prova1 #
```

3.2 Errori semantici

- Riga 6: la variabile “tot” è dichiarata come integer, mentre dovrebbe essere di tipo float

```

Terminale
linux-gamb:/home/vincenzo/Documenti/PROGETTO_LFC/prove_compilerare_ada/prova1 # ./compile.sh
linux-gamb:/home/vincenzo/Documenti/PROGETTO_LFC/prove_compilerare_ada/prova1 # ./adac calcola_media.adb
ERRORE SEMANTICO:errore alla riga 19 la variabile "tot" è di tipo "integer" mentre la funzione "ValoreMedio" restituisce un "float"
linux-gamb:/home/vincenzo/Documenti/PROGETTO_LFC/prove_compilerare_ada/prova1 # 

```

- riga 8: omettendo il parametro formale “x” nella definizione della funzione “ValoreMedio” si ha il seguente errore quando si cerca di usare la variabile “x”

```

Terminale
linux-gamb:/home/vincenzo/Documenti/PROGETTO_LFC/prove_compilerare_ada/prova1 # ./compile.sh
linux-gamb:/home/vincenzo/Documenti/PROGETTO_LFC/prove_compilerare_ada/prova1 # ./adac calcola_media.adb
ERRORE SEMANTICO:errore alla riga 13 tipo o variabile o funzione "x" non definita
linux-gamb:/home/vincenzo/Documenti/PROGETTO_LFC/prove_compilerare_ada/prova1 # 

```

- riga 8: dichiariamo i parametri formali della funzione “ValoreMedio” come integer invece che come float

```

Terminale
linux-gamb:/home/vincenzo/Documenti/PROGETTO_LFC/prove_compilerare_ada/prova1 # ./compile.sh
linux-gamb:/home/vincenzo/Documenti/PROGETTO_LFC/prove_compilerare_ada/prova1 # ./adac calcola_media.adb
ERRORE SEMANTICO:errore alla riga 13 la variabile "media" è "float" mentre la variabile "x" è "integer"
linux-gamb:/home/vincenzo/Documenti/PROGETTO_LFC/prove_compilerare_ada/prova1 # 

```

- riga 9: si è dichiarato la variabile “media” come integer, invece dovrebbe essere di tipo float

```

Terminale
linux-gamb:/home/vincenzo/Documenti/PROGETTO_LFC/prove_compilerare_ada/prova1 # ./compile.sh
linux-gamb:/home/vincenzo/Documenti/PROGETTO_LFC/prove_compilerare_ada/prova1 # ./adac calcola_media.adb
ERRORE SEMANTICO:errore alla riga 13 la variabile "media" è "integer" mentre la variabile "x" è "float"
linux-gamb:/home/vincenzo/Documenti/PROGETTO_LFC/prove_compilerare_ada/prova1 # 

```

- riga 13: al denominatore della divisione compare un numero di tipo integer invece che di tipo float

```

Terminale
linux-gamb:/home/vincenzo/Documenti/PROGETTO_LFC/prove_compilatore_ada/prova1 # ./compile.sh
linux-gamb:/home/vincenzo/Documenti/PROGETTO_LFC/prove_compilatore_ada/prova1 # ./adac calcola_media.adb
ERRORE SEMANTICO:errore alla riga 13 ci si aspetta solo valori "float" per modificare la variabile "media"
linux-gamb:/home/vincenzo/Documenti/PROGETTO_LFC/prove_compilatore_ada/prova1 # 

```

- riga 19: la variabile “tot” è stata dichiarata come integer, mentre la funzione “ValoreMedio” restituisce un valore float

```

Terminale
linux-gamb:/home/vincenzo/Documenti/PROGETTO_LFC/prove_compilatore_ada/prova1 # ./compile.sh
linux-gamb:/home/vincenzo/Documenti/PROGETTO_LFC/prove_compilatore_ada/prova1 # ./adac calcola_media.adb
ERRORE SEMANTICO:errore alla riga 19 la variabile "tot" è di tipo "integer" mentre la funzione "ValoreMedio" restituisce un "float"
linux-gamb:/home/vincenzo/Documenti/PROGETTO_LFC/prove_compilatore_ada/prova1 # 

```

- riga 19: mancanza di un parametro attuale nella funzione “ValoreMedio”

```

Terminale
linux-gamb:/home/vincenzo/Documenti/PROGETTO_LFC/prove_compilatore_ada/prova1 # ./compile.sh
linux-gamb:/home/vincenzo/Documenti/PROGETTO_LFC/prove_compilatore_ada/prova1 # ./adac calcola_media.adb
ERRORE SEMANTICO:errore alla riga 19 la funzione "ValoreMedio" prevede 2 parametri
linux-gamb:/home/vincenzo/Documenti/PROGETTO_LFC/prove_compilatore_ada/prova1 # 

```

Conclusioni

Il lavoro svolto ha permesso di comprendere l'importanza e la difficoltà della creazione di un compilatore.

Grazie al progetto sviluppato si sono apprese in pieno le potenzialità dei tool **Flex** e **Bison**, ovvero software free che semplificano la realizzazione di un compilatore e che permettono la **generazione automatica del codice C** indispensabile per la realizzazione di un analizzatore lessicale e sintattico (componenti fondamentali di un compilatore).

Flex e Bison sono dei software che permettono di alleggerire il lavoro del programmatore che non deve creare del codice per l'analisi lessicale e sintattica del compilatore che si implementa.

Attraverso l'implementazione del compilatore “*adac*” è stato possibile apprendere il linguaggio di programma ADA 95 che è un linguaggio estremamente utile e che ci permette di arricchire le nostre conoscenze informatiche.

Con lo studio del linguaggio ADA 95 e con la realizzazione del compilatore “*adac*” sono state apprese nuove tecniche di programmazione (implementazione di una **Symbol Table** con la *Hash Table*) utili per la formazioni di noi studenti.

Il progetto realizzato ha consentito la realizzazione di un compilatore didattico “*adac*” con discrete potenzialità e discreti risultati per la compilazione di programmi in linguaggio ADA 95, nonostante le limitazioni previste.

Mediante questo lavoro è stato possibile comprendere le potenzialità e le caratteristiche di un compilatore.

Infine con il progetto implementato si è compreso che la realizzazione di un compilatore prevede un lavoro molto sofisticato ed attento.

Bibliografia e Sitografia

- Prof. Giacomo Piscitelli, *Dispense del corso di Linguaggi Formali e Compilatori*, a.a. 2010/2011
- C. Donnelly and R. Stallman, *The Yacc-compatible Parser Generator*, Bison Version 2.4.3, 5 August 2010
- Anthony A. Aaby , *Compiler Construction using Flex and Bison*, Walla Walla College cs.wwc.edu - aabyan@wwc.edu, Version of February 25, 2004
- Claudio Marsan, *Programmazione in Ada 95*, Claudio Marsan, http://www.liceomendrisio.ch/~marsan/informatica/ada95/programmazione_in_ada95.pdf