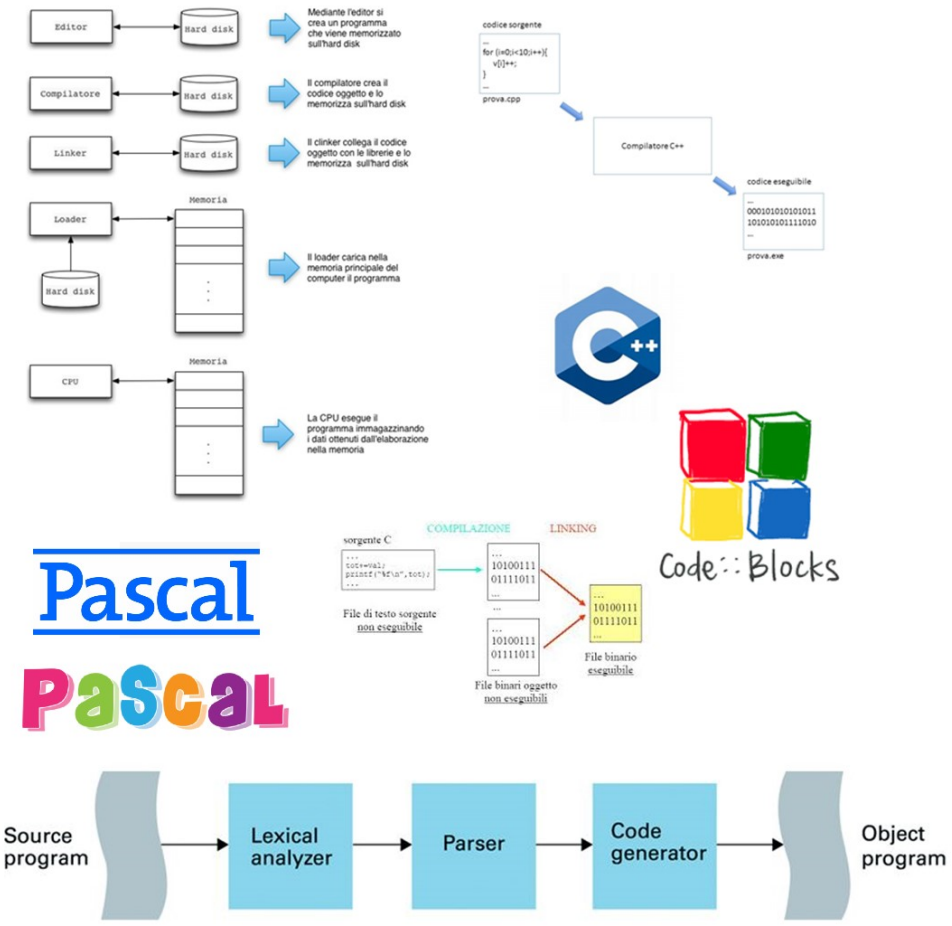
**ITIS-LS "Francesco Giordani" Caserta**

**prof. Ennio Ranucci**

**a.s. 2019-2020**

# *Compilatori e semplici simulazioni*

*Esercizi svolti in ambiente C++ e Pascal*



**Bibliografia:**Giuseppe Callegarin, Lucio Varagnolo “Corso di informatica generale” Ed. CEDAM

**Analisi Lessicale**

L'analisi lessicale è la prima fase di ogni processo di compilazione. Compito dell'analisi lessicale è quello di consentire all'analizzatore sintattico di lavorare su una versione più astratta del testo sorgente.
Per spiegare questo concetto occorre far presente che l'alfabeto terminale utilizzabile dall'utente nella scrittura di un programma è composto tipicamente da lettere, cifre, simboli di punteggiatura, operatori e qualche carattere speciale;cioè un sottoinsieme dei caratteri della tastiera.

L'analizzatore sintattico deve invece operare ad un livello in cui:

a) ciascuna parola chiave (come for, endl, while, do, ecc. ) viene considerata come

secondo simbolo;

b) tutti gli identificatori vengono considerati come un unico simbolo "id" con un

indicazione separata di dove trovare il loro nome individuale;

c) le costanti di un certo tipo "t" vengono considerate come un unico simbolo "const"

con un indicazione di dove trovare il suo valore;

d) operatori composti da più simboli come ":=" sono considerati come un unico

simbolo;

e) sequenze di caratteri inutili al processo di analisi vengono ignorate (spazzi bianchi,

commenti, caratteri di fine linea, ecc.).

L'analizzatore lessicale (o scanner) deve quindi farsi carico di un lavoro di lettura, selezione, raggruppamento e successiva classificazione dei caratteri componenti il testo sorgente.

Il testo subisce quindi una traduzione in un linguaggio su un alfabeto più astratto i cui simboli vengono comunemente chiamati "token".

L'analizzatore lessicale può essere realizzato come un sottoprogramma che restituisce un token tutte le volte che viene invocato dall'analizzatore sintattico. L'analizzatore lessicale partecipa alla gestione della tabella dei simboli, quando viene incontrato un identificatore, esso viene inserito (se non esiste già) nella tabella dei simboli e viene restituito all'analizzatore sintattico un puntatore all'identificatore stesso. Esempio:

Si supponga che gli elementi del lessico di un linguaggio denominato “LingSempl” siano i seguenti:

a) identificatori illimitati composti secondo le usuali regole;

b) costanti intere senza segno;

c) parole chiave LEGGI e SCRIVI;

d) simbolo di assegnazione" := " ;

e) simboli " =, +, * " ;

f) carattere " $ " di fine testo.

Spazi possono comparire ovunque ma non possono spezzare identificatori, costanti, parole chiave o il simbolo ":= " di assegnamento; i commenti iniziano con "{" e terminano con "}".

***ITIS-LS "Francesco Giordani" Caserta***
***Anno scolastico:*** *2019/2020*
***Classe 3^ sez.B spec. Informatica e telecomunicazioni***
***Data:***
***Numero progressivo dell'esercizio:*** *es1*
***Versione:*** *1.0*
***Programmatore/i:***
***Sistema Operativo:*** *Windows 10*
***Compilatore/Interprete:*** *Code::Blocks 17.12*
***Obiettivo didattico:***
*Analisi lessicale*
***Obiettivo del programma:***
*Realizzare l'analizzatore lessicale (o scanner) del linguaggio "LingSempl"*

```
#include <iostream>

#include <string.h>

using namespace std;

/*Un token è il più piccolo elemento di un programma  significativo per il compilatore.

Il parser riconosce questi tipi di token:

identificatori, parole chiave, valori letterali, operatori, segni di punteggiatura e altri separatori.

*/

enum token  {ident,costante,leggi,scrivi,assegna,uguale,add,mul,finetesto} ;

token SIMB;

int PuntatoreCarattere=-1;

string  nomeId, sorgente;

int valoreCostante;

void errore_lessicale()

{

  cout<<"errore lessicale"<<endl;

}

void scanner()

{

  char car;int i=0;

   car=sorgente[PuntatoreCarattere];

   do

    {
```

```
  switch (car)
    {
     case ' ': PuntatoreCarattere++; car=sorgente[PuntatoreCarattere];break;
     case '{': do
                 {
                   PuntatoreCarattere++;
                   car=sorgente[PuntatoreCarattere];
                 }
               while ( sorgente[PuntatoreCarattere]!='}');
               break;// le graffe racchiudono commenti
    }
  }
 while (car==' ' || car=='{');

switch (car)
      {
        case 'A'...'Z':
           i=PuntatoreCarattere; //conserva in i la prima posizione del carattere
           while ((sorgente[PuntatoreCarattere+1] >= '0' && sorgente[PuntatoreCarattere+1] <= '9')||
                 (sorgente[PuntatoreCarattere+1] >= 'A' && sorgente[PuntatoreCarattere+1] <= 'Z'))
            {
             PuntatoreCarattere++;
            };
             nomeId=sorgente.substr(i,PuntatoreCarattere-i+1);//estrae la sottrostringa
             if (nomeId=="LEGGI")   SIMB = leggi;
             else if (nomeId=="SCRIVI") SIMB = scrivi;
                 else SIMB = ident;
        break;
        case '0'...'9':
           SIMB = costante;
           valoreCostante=(int)car-(int)'0';
           while (sorgente[PuntatoreCarattere+1] >= '0' && sorgente[PuntatoreCarattere+1] <= '9')
```

```
            {
            PuntatoreCarattere++;
            car=sorgente[PuntatoreCarattere];
            valoreCostante=valoreCostante*10+(int)car-(int)'0';
            };
        break;
        case ':'     : if (sorgente[PuntatoreCarattere+1]== '=')
                  {
                   PuntatoreCarattere++;
                   SIMB = assegna;//:= equivale ad assegna5
                  }
                 else errore_lessicale();
        break;
        case '='     : SIMB = uguale; break;
        case '+'     : SIMB = add; break;
        case '*'     : SIMB = mul; break;
        case '$'     : SIMB = finetesto;break;
      }
}
int main()
{
 sorgente="LEGGI A B=2*A K=2+3 SCRIVI K$";
 cout<<"Codice sorgente: "<<sorgente<<endl;
 do
  {
   PuntatoreCarattere++;
   scanner();
   switch (SIMB)
    {
     case ident    : cout<<"ident"<<endl;break;
     case costante : cout<<"costante= "<<valoreCostante<<endl;break;
     case leggi    : cout<<"leggi"<<endl;break;
```

```
      case scrivi   : cout<<"scrivi"<<endl;break;
      case assegna  : cout<<"<-"<<endl;break;
      case uguale   : cout<<"="<<endl;break;
      case add      : cout<<"+"<<endl;break;
      case mul      : cout<<"*"<<endl;break;
      case finetesto: cout<<"finetesto";break;
     };
   }
  while(SIMB != finetesto) ;
  return 0;
}
```

**CODIFICA PASCAL**

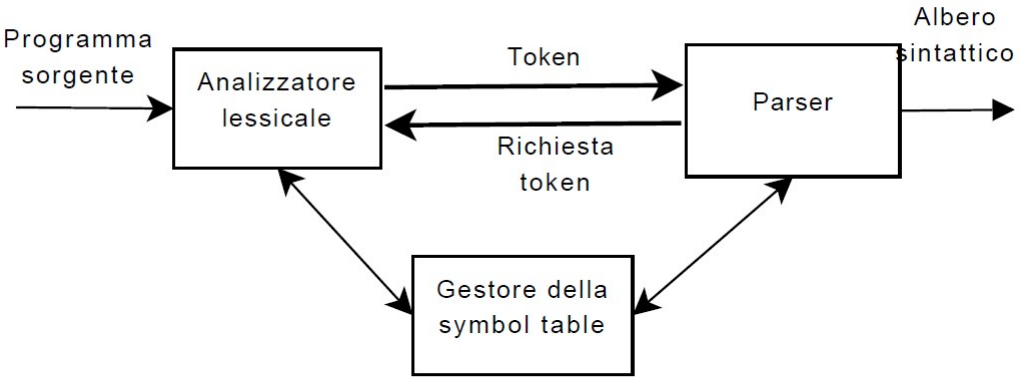
```
{INSERIRE I DATI IN MAIUSCOLO}
program analess;
type
  token=(ident,costante,leggi,scrivi,assegna,uguale,add,mul,finetesto);
var
  simb              :token;
  nomeid,sorgente     :string;
  pc,valorecostante   :integer;
procedure errore_lessicale;
begin
  write('errore lessicale');
  readln
end;
procedure scanner;
var
  car:char;
  i  :integer;
```

```
begin
  repeat
    pc:=pc+1;
    car:=sorgente[pc];
      case car of
        ' ':;
        '{':repeat pc:=pc+1 until sorgente[pc]='}';
      end;
  until (car<>' ') and (car<>'{');
  case car of
    'A'..'Z': begin
                i:=pc;
                while sorgente[pc+1] in ['A'..'Z','_','0'..'9'] do
                pc:=pc+1;
                nomeid:=copy(sorgente,i,pc-i+1);
                if nomeid='LEGGI' then  simb := leggi
                  else if nomeid='SCRIVI' then simb := scrivi
                    else simb := ident
              end;
    '0'..'9': begin
                simb := costante;
                valorecostante:=ord(car)-ord('0');
                while sorgente[pc+1] in ['0'..'9'] do
                begin
                  pc:=pc+1;
                  car:=sorgente[pc];
                  valorecostante:=valorecostante*10+ord(car)-ord('0')
```

```
            end
          end;
   ':'    : if sorgente[pc+1]= '=' then begin
                                  pc:=pc+1;
                                  simb := assegna
                                end
                                else errore_lessicale;
   '='    : simb:= uguale;
   '+'    : simb:= add;
   '*'    : simb:= mul;
   '$'    : simb:= finetesto;
  else errore_lessicale
  end;
end;
BEGIN
{  write('Inserisci testo sorgente : ');
  readln(sorgente);
  sorgente:=sorgente + '$';
}
sorgente:='LEGGI A B=2*A K=2+3 SCRIVI K$';
 pc:=0;
  repeat
    scanner;
    case simb of
      ident    : writeln('ident');
      costante : writeln('costante=',valorecostante,'    +2 = ',
                      valorecostante + 2);
```

```
      leggi    : writeln('leggi');
      scrivi   : writeln('scrivi');
      assegna  : writeln(':=');
      uguale   : writeln('=');
      add      : writeln('+');
      mul      : writeln('*');
      finetesto: write('finetesto');
    end;
  until simb = finetesto
END.
```

- Lo scanner opera come “routine” del parser.




- Quando il parser deve leggere il prossimo simbolo di input esegue una chiamata allo scanner.
- Lo scanner legge la “prossima” porzione di input fino a quando non riconosce un *token*, che viene restituito al parser.

La prima elementare traduzione effettuata durante l'analisi lessicale va in ingresso alla fase di analisi sintattica. Gli analizzatori sintattici sono basati, in genere, su particolari classi di grammatiche che non richiedono il cosiddetto backtracking (ritorno indietro), si segue, cioè all'interno dell'albero delle alternative un certo cammino. Se si arriva ad un punto dal quale non si può continuare, si torna indietro fino alla diramazione nella quale si era scelto il cammino ora interrotto, per sceglierne un altro. Bisogna rilevare che il migliore algoritmo di analisi di una stringa di lunghezza n risulta essere di complessità $n^2$, un costo inaccettabile per le applicazioni pratiche. Per questo motivo sono state cercate ed individuate delle sottoclassi delle grammatiche che consentono un'analisi sintattica di complessità lineare. Quindi l'analizzatore sintattico non controlla eventuali vincoli dipendenti dal contesto che vengono verificati in una fase successiva di analisi semantica, ad esempio:

a) verificare che ogni identificatore non sia dichiarato più di una volta nello stesso

ambiente;

b) verificare che tutte le variabili siano state dichiarate prima del loro uso;

c) verificare che non manchino etichette alle istruzioni a cui si salta e che siano presenti una sola volta;

d) verificare che le interfaccia di chiamata a sottoprogramma siano conformi a quelle richieste in posizione, numero e tipo degli argomenti;

e) verificare che in ogni assegnamento "x□e "il tipo di e sia compatibile con quello di x;

f) stabilire il tipo di un espressione o di una operazione in caso di sovraccarico degli operatori o di uso di espressioni miste.

L'analizzatore semantico potrebbe anche informare al programmatore di istruzioni inutili o irraggiungibili.

Alla fase di analisi semantica può seguire quella di traduzione di codice oggetto ;questa soluzione è adottabile solo se non si ha alcuna pretesa sulla qualità del codice prodotto, cioè sulla sua efficienza in tempo di esecuzione e spazio occupato in memoria.

Questo è uno dei principali motivi per i quali è generalmente preferibile far effettuare prima una traduzione in un linguaggio intermedio fra quello sorgente e quello oggetto. La versione intermedia del programma può infatti essere analizzata molto meglio del codice oggetto per scoprire le trasformazioni da apportare per arrivare alla produzione di un codice efficiente. Esistono diversi tipi di linguaggi intermedi:

- alberi di operatori (o alberi astratti);
- notazione polacca prefissa;
- notazione polacca postfissa;
- liste di quadruple.

In qualunque caso per linguaggio intermedio viene inteso il linguaggio assembler. Un aspetto non secondario della compilazione è quello della segnalazione e del trattamento degli errori che possono avvenire in tutte le fasi.

Essi possono essere classificati in tre categorie:

a) errori lessicali (identificatori troppo lunghi, uso di caratteri non previsti, ecc.);

b) errori di sintassi (espressioni scorrette, parole chiavi mancanti o al posto sbagliato, punteggiatura, ecc.);

c) errori semantici (identificatori non dichiarati, costanti troppo grandi, istruzioni irraggiungibili, ecc.).

***ITIS-LS "Francesco Giordani" Caserta***
***Anno scolastico:*** *2019/2020*
***Classe 4^ sez.B spec. Informatica e telecomunicazioni***
***Data:***
***Numero progressivo dell'esercizio:*** *es2*
***Versione:*** *1.0*
***Programmatore/i:***
***Sistema Operativo:*** *Windows 10*
***Compilatore/Interprete:*** *Code::Blocks 17.12*
***Obiettivo didattico:***
*Analisi lessicale*
***Obiettivo del programma:***
Scrivere un programma che letta in ingresso una sequenza di caratteri maiuscoli terminata da '$', rappresentante una espressione aritmetica valida con operatori binari +, -, *, / infissi ed eventuali parentesi tonde, stampi la stringa in forma polacca postfissa.
<espressione>::= <termine> {(+ | -) <termine>}
<termine>::= <fattore> {(*|/) <fattore>}
<fattore> ::= <lettera> | ( <espressione> )
Esempi:

| | | | | |
|---|---|---|---|---|
| a | a+b | a*b+c | a*b+c/d+e | |
| a | ab+ | ab*c+ | ab*cd/+e+ | polacca |
| a*(b-c)*d | | a*((b-c)/d)+e | | |
| abc-*d* | | abc-d/*e+ | | polacca |

```
#include <iostream>

#include <string.h>

using namespace std;

/*Un token è il più piccolo elemento di un programma  significativo per il compilatore.

Il parser riconosce questi tipi di token:

identificatori, parole chiave, valori letterali, operatori, segni di punteggiatura e altri separatori.

*/

enum token  {variabile,costante,add,sub,mul,dividi,tondaaperta,tondachiusa,finetesto} ;

token SIMB, operatoreMul, operatoreAdd;

int PuntatoreCarattere=-1;

string  nomeId,sorgente;

int valoreCostante;

void compila_espressioni();

void errore_lessicale()

{

  cout<<"errore lessicale"<<endl;

}
```

```
void scanner()
{
 char car;int i=0;
 PuntatoreCarattere++;
 car=sorgente[PuntatoreCarattere];
  do
   {
    switch (car)
      {
       case ' ': PuntatoreCarattere++; car=sorgente[PuntatoreCarattere];break;
       case '{': do
                  {
                    PuntatoreCarattere++;
                  }
                 while ( sorgente[PuntatoreCarattere]!='}');
                 break;// le graffe racchiudono commenti
      }
   }
  while (car==' ' || car=='{');
 car=sorgente[PuntatoreCarattere];
 switch (car)
       {
         case 'A'...'Z':
             nomeId=sorgente[PuntatoreCarattere];
             while ((sorgente[PuntatoreCarattere+1] >= '0' && sorgente[PuntatoreCarattere+1] <=
'9')||
                    (sorgente[PuntatoreCarattere+1] >= 'A' && sorgente[PuntatoreCarattere+1] <= 'Z'))
              {
                nomeId=nomeId+sorgente[PuntatoreCarattere+1];
                PuntatoreCarattere++;
              };
             SIMB = variabile;
         break;
```

```
        case '0'...'9':
           SIMB = costante;
           valoreCostante=(int)car-(int)'0';
           while (sorgente[PuntatoreCarattere+1] >= '0' && sorgente[PuntatoreCarattere+1] <= '9')
            {
            PuntatoreCarattere++;
            car=sorgente[PuntatoreCarattere];
            valoreCostante=valoreCostante*10+(int)car-(int)'0';
            };
        break;
        case '-'    : SIMB = sub; break;
        case '+'    : SIMB = add; break;
        case '*'    : SIMB = mul; break;
        case '/'    : SIMB = dividi;break;
        case '('    : SIMB = tondaaperta;break;
        case ')'    : SIMB = tondachiusa;break;
        case '$'    : SIMB = finetesto;break;
      }
}
void fattore()
 {
   switch (SIMB)
   {
     case variabile :
              cout<<nomeId<<" ";
              scanner();
     break;
     case costante :
               scanner();
               cout<<valoreCostante<<" ";
     break;
     case tondaaperta :
```

```
            scanner();
            compila_espressioni();
            if (SIMB==tondachiusa) scanner();
     break;
    }
   }
void termine()
   {
    fattore();
    while (SIMB==mul || SIMB==dividi)
      {
       operatoreMul=SIMB;
       scanner();
       fattore();
       if (operatoreMul==mul) cout<< "* ";
          else cout<<"/ ";
      }
    }
void compila_espressioni()
{
  if (SIMB==sub)
    {
     scanner();
     termine();
     cout<<"neg ";
    }
  else termine();
  while (SIMB==add || SIMB==sub)
    {
     operatoreAdd=SIMB;
     scanner();
     termine();
```

```
    if (operatoreAdd==add) cout<<"+ ";
      else cout<<"- ";
   }
};
int main()
{
 //sorgente deve essere costituito da lettere maiuscole
 sorgente="(A*(B-C)*D$"; //risultato atteso ABC-*D* altro esempio A*((B-C)/D)+E risultato atteso ABC-D/*E+
 cout<<"Codice sorgente: "<<sorgente<<endl;
 PuntatoreCarattere=-1;
 do
  {
   scanner();
/*
   switch (SIMB)
    {
     case variabile   : cout<<"var"<<endl;break;
     case costante : cout<<"costante= "<<valoreCostante<<endl;break;
     case sub    : cout<<"sub"<<endl;break;
     case tondaaperta  : cout<<"("<<endl;break;
     case tondachiusa  : cout<<")"<<endl;break;
     case dividi   : cout<<"div"<<endl;break;
     case add      : cout<<"+"<<endl;break;
     case mul      : cout<<"*"<<endl;break;
     case finetesto: cout<<"finetesto";break;
    };
    */
    compila_espressioni();
  }
 while(SIMB != finetesto) ;
 return 0;
}
```

## CODIFICA PASCAL

```
{INSERIRE I DATI IN MAIUSCOLO}
program COMPILATORE_ESPRESSIONI;
type
  token=(variabile,costante,add,sub,mul,dividi,
  tondaaperta,tondachiusa,finetesto);
var
  simb                    :token;
  sorgente,parola         :string;
  pc,valorecostante       :integer;
procedure errore;
begin
  write('errore !');
  readln
end;
procedure scanner;
var
  car:char;
  i  :integer;
begin
  repeat
    pc:=pc+1;
    car:=sorgente[pc];
      case car of
        ' ':;
        '{':repeat pc:=pc+1 until sorgente[pc]='}';
      end;
```

```
until (car<>' ') and (car<>'{');
case car of
 'A'..'Z': begin
             parola:=car;
             while sorgente[pc+1] in ['A'..'Z','_','0'..'9'] do
             begin
               parola:=parola + sorgente[pc+1];
               pc:=pc+1
             end;
             simb:=variabile
           end;
 '0'..'9': begin
             simb := costante;
             valorecostante:=ord(car)-ord('0');
             while sorgente[pc+1] in ['0'..'9'] do
             begin
               pc:=pc+1;
               car:=sorgente[pc];
               valorecostante:=valorecostante*10+ord(car)-ord('0')
             end
           end;
 '+'     : simb:= add;
 '-'     : simb:= sub;
 '*'     : simb:= mul;
 '/'     : simb:= dividi;
 '('     : simb:= tondaaperta;
 ')'     : simb:= tondachiusa;
```

```
   '$'     : simb:= finetesto;
  else errore
  end;
end;
procedure compila_espressioni;
var
  operatoreadd:token;
  procedure fattore;
  begin
    case simb of
    variabile : begin
                  write(parola,' ');
                  scanner
                end;
   costante : begin
                  scanner;
                  write(valorecostante,' ');
                end;
   tondaaperta : begin
                   scanner;
                   compila_espressioni;
                   if simb=tondachiusa then scanner;
                 end;
    end
  end;
  procedure termine;
  var
```

```
 operatoremul:token;
begin
  fattore;
  while simb in [mul,dividi] do
    begin
      operatoremul:=simb;
      scanner;
      fattore;
      if operatoremul=mul then write('* ')
                          else write('/ ')
    end
end;


begin
 if simb=sub then begin
                    scanner;
                    termine;
                    write('neg ')
                  end
              else termine;
 while  simb in [add,sub] do
                  begin
                    operatoreadd:=simb;
                    scanner;
                    termine;
                    if operatoreadd=add then write('+ ')
                                        else write('- ')
```

```
            end
end;
BEGIN
 { write('Inserisci testo sorgente : ');
  readln(sorgente);
  sorgente:=sorgente + '$';
}
sorgente:='(A*(B-C)*D$';
  pc:=0;
  repeat
    scanner;
    compila_espressioni;
  until simb = finetesto
END.
```

***ITIS-LS "Francesco Giordani" Caserta***
***Anno scolastico:*** *2019/2020*
***Classe 4^ sez.B spec. Informatica e telecomunicazioni***
***Data:***
***Numero progressivo dell'esercizio:*** *es3*
***Versione:*** *1.0*
***Programmatore/i:***
***Sistema Operativo:*** *Windows 10*
***Compilatore/Interprete:*** *Code::Blocks 17.12*
***Obiettivo didattico:***
*Analisi sintattica*
***Obiettivo del programma:*** *costruire un analizzatore sintattico per il linguaggio "facileLing" definito induttivamente come segue:*

*1) Un identificatore ident (costruito con le usuali regole) appartiene ad "facileLing".*

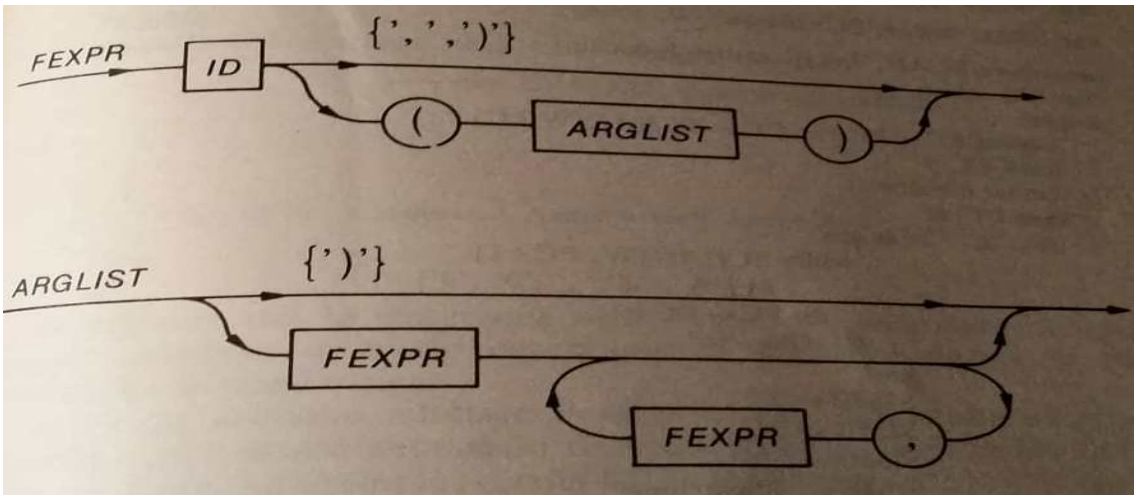
*2) Se W1, W2,… Wn è una sequenza (eventualmente vuota) di stringhe di*

*"facileLing", allora anche ident(W1,W2,...,Wn) è una stringa di "facileLing".*

*Esempi: A, SOM, ZERO(), ADD(X,Y1), MULT(ADD(A,ZERO ()), MULT (X, Y)), SEQUENZA VUOTA()*

*ecc.*

*In altre parole, "facileLing" è il linguaggio costituito da tutte le possibili espressioni (o termini) di un ipotetico linguaggio di programmazione in cui le costanti sono viste come funzioni a zero argomenti e l'usuale notazione funzionale prefissa è l'unica ammessa. Come al solito assumiamo che gli identificatori facciano parte del lessico del linguaggio; la sintassi di "facileLing" può allora limitarsi ai due diagrammi sintattici:*




*Essi corrispondono alle seguenti produzioni in forma estesa:*

*"facileLing"=ident[(argList)]*

*argList ["facileLing"{','"facileLing"}]*

*Le parentesi quadre racchiudono parti che possono mancare (parti opzionali). Si osservi che l'analizzatore non fa alcun controllo sul numero e sul tipo di argomenti in quanto ciò è generalmente di competenza dell'analizzatore semantico. Il trattamento dell'errore consiste nella segnalazione del tipo e del punto in cui è stato riscontrato.*

```
#include <iostream>

#include <string.h>


using namespace std;


/*Un token è il più piccolo elemento di un programma  significativo per il compilatore.

Il parser riconosce questi tipi di token:

identificatori, parole chiave, valori letterali, operatori, segni di punteggiatura e altri separatori.

*/

void analizzatoreSintattico();

void argList();

enum token  {ident,tondaAperta,tondaChiusa,virgola,finetesto} ;

token SIMB;

int PuntatoreCarattere=-1;

string  nomeId, sorgente;

int valoreCostante;

bool erroreFlag=false;

void msgErrore(string err)
```

```
{
  cout<<err<<endl;
  erroreFlag=true;
}
void scanner()
{
  char car;
   PuntatoreCarattere++;
   car=sorgente[PuntatoreCarattere];
   do
    {
     switch (car)
       {
        case ' ': PuntatoreCarattere++; car=sorgente[PuntatoreCarattere];break;
        case '{': do
                  {
                    PuntatoreCarattere++;
                    car=sorgente[PuntatoreCarattere];
                  }
                 while ( sorgente[PuntatoreCarattere]!='}');
                 break;// le graffe racchiudono commenti
       }
    }
   while (car==' ' || car=='{');

  switch (car)
        {
          case 'A'...'Z':
              while  ((sorgente[PuntatoreCarattere+1]  >=  '0'  &&  sorgente[PuntatoreCarattere+1]  <=
'9')||
                   (sorgente[PuntatoreCarattere+1] >= 'A' && sorgente[PuntatoreCarattere+1] <= 'Z'))
                {
                  PuntatoreCarattere++;
```

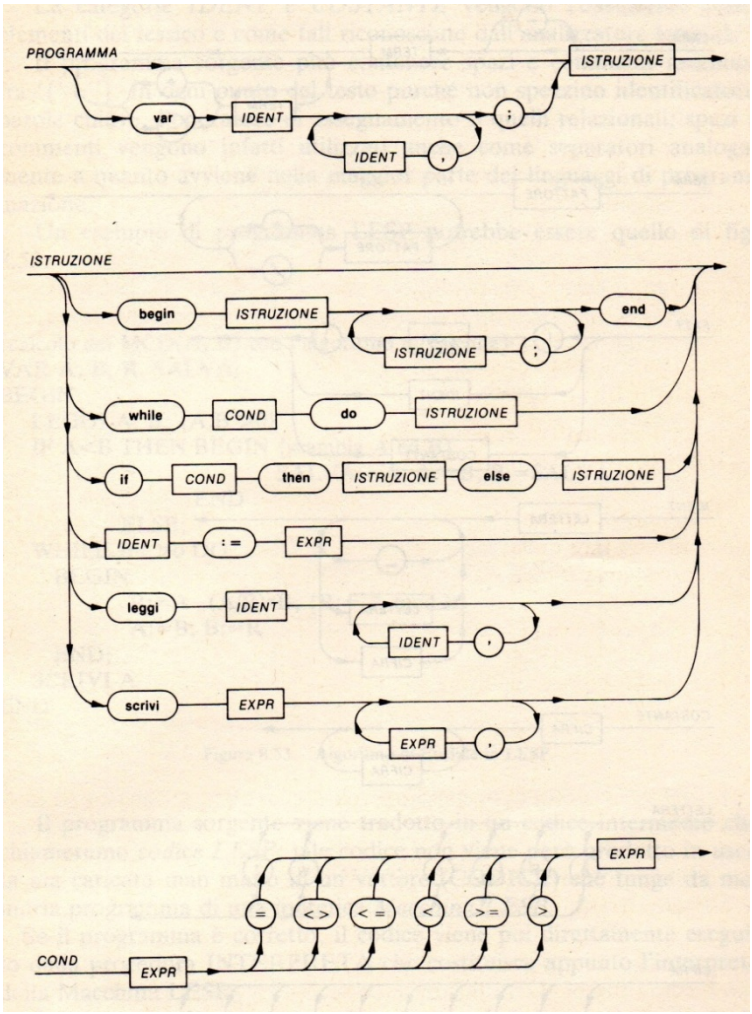
```
            };
            SIMB = ident;
        break;
        case '('    : SIMB = tondaAperta; break;
        case ')'    : SIMB = tondaChiusa; break;
        case ','    : SIMB = virgola; break;
        case '$'    : SIMB = finetesto;break;
        default     : msgErrore("Errore lessicale");break;

      }
}
void analizzatoreSintattico()
{
   if (SIMB==ident)
      {
         scanner();
         if (SIMB==tondaAperta) scanner(); else return ;
         argList();
         if (SIMB==tondaChiusa) scanner(); else msgErrore("manca la parentesi tonda chiusa");
      }
    else msgErrore("manca identificatore");
}
void argList()
{
   if (SIMB==ident)
      {
         analizzatoreSintattico();
         while (SIMB==virgola)
            {
               scanner();
               analizzatoreSintattico();
            }
```

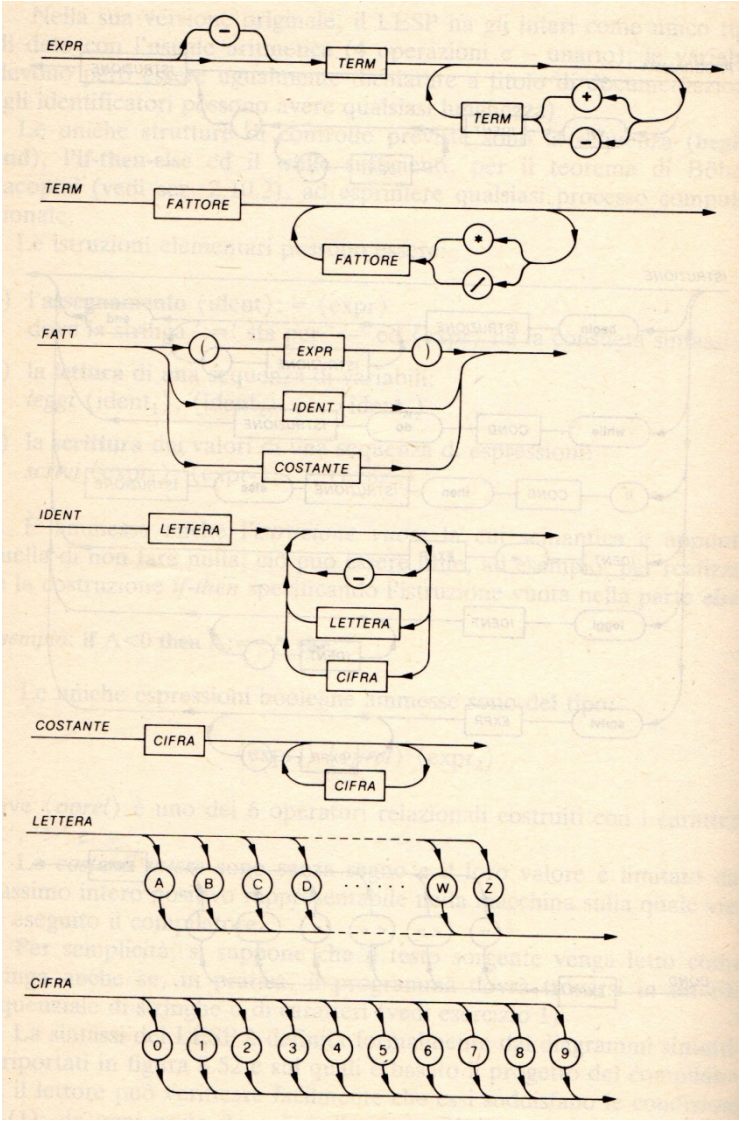
```
        }
}
int main()
{
  sorgente="MULT(ADD(A,ZERO ()))$";
  cout<<"Codice sorgente: "<<sorgente<<endl;
  PuntatoreCarattere++;
  scanner();
  analizzatoreSintattico();
  if (SIMB == finetesto && !erroreFlag) cout<<"Stringa corretta"; else msgErrore("Errori");
  return 0;
}
```

***ITIS-LS "Francesco Giordani" Caserta***
***Anno scolastico:*** *2019/2020*
***Classe 4^ sez.B spec. Informatica e telecomunicazioni***
***Data:***
***Numero progressivo dell'esercizio:*** *es4*
***Versione:*** *1.0*
***Programmatore/i:***
***Sistema Operativo:*** *Windows 10*
***Compilatore/Interprete:*** *Code::Blocks 17.12*
***Obiettivo didattico:***
*Compilatore didattico*
***Obiettivo del programma:*** *realizzare un semplice compilatore che :*

*1)Ha gli interi come unico tipo di dati e le 4 operazioni aritmetiche;*
*2)Le variabili devono essere dichiarate e gli identificatori possono avere qualsiasi lunghezza;*
*3)Le uniche strutture di controllo sono la sequenza, se allora altrimenti ed il while che per il teorema di Bohm jacopini sono sufficient ad esprimere qualsiasi processo computazionale;*
*4)Le istruzioni possono essere:*
*l'assegnamento <ident>:=<espr>;*
*la lettura leggi <ident1>,<ident2>…,<identn>;*
*la scrittura scrivi <ident1>,<ident2>…,<identn>;*
*le espressioni booleane ammesse: <espr> <operatoreRelazionale> <espr> (operatori <>=);*

*Diagrammi sintattici che definiscono formalmente il linguaggio implementato dal "compilatore didattico":*

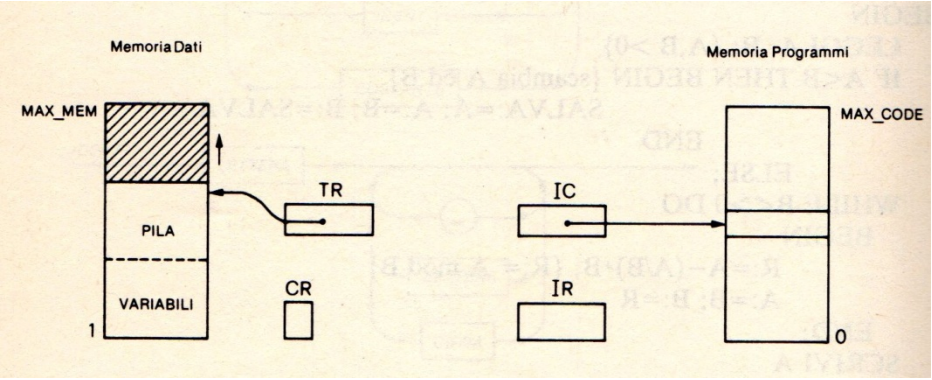





*Di seguito viene riportato l'algoritmo di Euclide per il calcolo del MCD tra due numeri interi*

```
MCD(A,B):
VAR A. B, R, SALVA
BEGIN
  LEGGI A, B; {A,B >0}
  IF A<B THEN BEGIN {scambia A ed B}
                    SALVA: A; A:-B; B:-SALVA
                END
        ELSE:
        WHILE B<>0 DO
            BEGIN
                R:=A-(A/B) B; (R:= A mod B)
                A:-B; B:-R
            END;
   SCRIVI A
END
```

*Il programma sorgente viene tradotto in un codice intermedio che chiameremo codice facileLing; tale codice non viene però prodotto in uscita ma caricato man mano in un vettore (CODICE) che funge da memoria programma di una ipotetica Macchina. Se il programma è corretto, il codice viene poi direttamente eseguito dalla procedura INTERPRETA che costituisce appunto l'interprete della Macchina. La struttura di questa macchina è semplicissima e comprende le seguenti risorse :*
*-una memoria programmi;*
*-una memoria dati (variabili e risultati intermedi) organizzata sia ad accesso random che a pila;*
*-un registro TR (Top Register) che punta all'operando in cima alla pila; il suo valore può essere inizializzato o modificato con l'istruzione LTR;*
*-un registro IC (Instruction Counter) che punta alla prossima istruzione che sarà prelevata dalla memoria programma per essere eseguita;*
*-un registro IR (Instruction Register) che contiene l'istruzionein corso di esecuzione;*
*-un registro CR (Condition Register) di un bit che contiene l'esito di un'operazione di confronto (COMP) fra i due operandi contenuti in cima alla pila.*

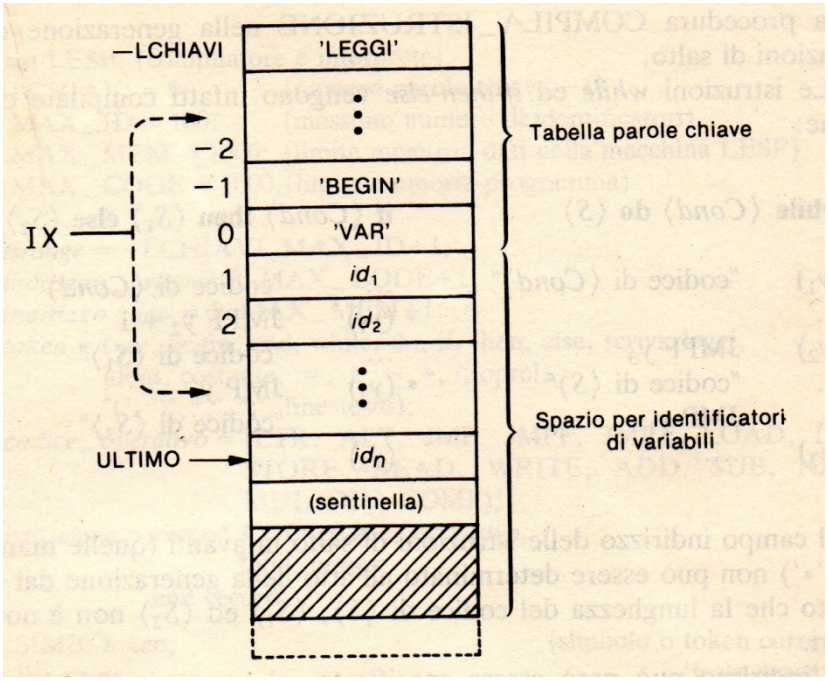



*Le istruzioni macchina hanno un unico formato composto di due soli campi:*
*1) Campo COP, cioè il codice operativo;*
*2) Campo IND (non utilizzato in alcune istruzioni)*
*Il secondo campo può essere interpretato anche come costante intera o come codice di operatore relazionale (istruzione COMP); l'indirizzamento è assoluto sia per la memoria dati che per quella programmi.*
*La pila cresce verso indirizzi crescenti della memoria dati ed è quindi opportuno allocare le variabili prima dell'inizio della pila. Le istruzioni, divise per classi, sono le seguenti:*
*I sta per una costante immediata, x per un indirizzo dati ed y per un indirizzo programma.*
*Trasferimento*
*LTR i : carica TR con il valore i:*
*LIT i: carica i in cima alla pila;*
*LOAD x : carica sulla pila il valore della variabile x;*
*STORE x : estrae il valore sulla pila mettendolo in x;*
*Controllo*
*ALT : arresta l'esecuzione;*
*JMP y: salta incondizionatamente all'istruzione y;*

*JMPT y salta a y se CR= true;*
*JMPF y salta a y se CR false;*
*Operazioni aritmetiche*
*NEG :cambia di segno all'operando in cima alla pila;*
*ADD, SUB, MUL, DIV: sostituisce i primi due operandi in cima alla pila con il risultato della corrispondente operazione; il secondo operando è quello in cima;*
*Confronto*
*COMP (oprel) (con (oprel) 0, 1, 2, 3, 4, 5 corrispondentiagli operatori , #, <, <, , >)*
*Pone il registro CR a true o false a seconda che l'operazione di confronto sui due operandi in cima alla pila sia vera o falsa; in cima alla pila c'è il second operando ed entrambi vengono estratti.*
*Ingresso/Uscita*
*READ x legge un intero nella variabile x*
*WRITE scrive ed estrae il valore in cima alla pila.*
*Di seguito è riportato il codice facileLing (espresso, per maggiore chiarezza, in una forma simbolica tipo Assembler) prodotto dal compilatore come traduzione dell'Algoritmo di Euclide MCD(A,B):*

```
( 0)       LTR    4
( 1)       READ   A
( 2)       READ   B
( 3)       LOAD   A
( 4)       LOAD   B
( 5)       COMP   <
( 6)       JMPF   L1
( 7)       LOAD   A
( 8)       STORE  SALVA
( 9)       LOAD   B
(10)       STORE  A
(11)       LOAD   SALVA
(12)       STORE  B
(13)       JMP    L1
(14)  L1:  LOAD   B
(15)       LIT    0
(16)       COMP   <>
(17)       JMPF   L2
(18)       LOAD   A
(19)       LOAD   A
(20)       LOAD   B
(21)       DIV
(22)       LOAD   B
(23)       MUL
(24)       SUB
(25)       STORE  R
(26)       LOAD   B
(27)       STORE  A
(28)       LOAD   R
(29)       STORE  B
(30)       JMP    L1
(31)  L2:  LOAD   A
(32)       SCRIVI
(33)       ALT
```

*ORGANIZZAZIONE DELLA TABELLA DEI SIMBOLI*



```
//COMPILATORE DIDATTICO Giuseppe Callegarin Lucio Varagnolo CEDAM

#include <iostream>
#include <string.h>
#include <stdlib.h>

using namespace std;

const int numeroparolechiave=9,
        numero_max_identificatori=100,
        max_memoria_dati=1000,
        max_memoria_codice=1000;
int tabellasimbolirange;// =-numeroparolechiave..numero_max_identificatori+1;
int indirizzo_prog;
int  indirizzo_dati; // = 1..max_memoria_dati+1;
enum token { variabile,inizio,fine,mentre,fai,se,allora,altrimenti,
          leggi,scrivi,ident,costante,assegna,addiziona,sottrai,moltiplica,dividi,
          op_relaz,tondaaperta,tondachiusa,virgola,puntovirgola,finetesto};

enum  codice_operativo {LTR,ALT,JMP,JMPF,JMPT,LOAD,LIT,STORE,LEGGERE,SCRIVERE,
                  ADD,SUB,NEG,MUL,DIVID,COMP};

struct istruzione
 {
   codice_operativo cop;
   int ind;
 };

token simb;
string  SORGENTE,SORGENTE1,SORGENTE2;
int  puntatoreCarattere,valoreCostante,i,k;
```

```
string  ts[110];// TS = TABELLA DEI SIMBOLI tabellasimbolirange; =-
numeroparolechiave..numero_max_identificatori+1;

int ultimo;//numero_max_identificatori;ULTIMO = PUNTATORE ALL'ULTIMO ELEMENTO DI TS

int ix; // tabellasimbolirange IX = INDICE ALLA PAROLA CERCATA NELLA TS

int num;

int codice_op_relaz; //0..5;codice operatore relazionale

int  ipi=0; //indirizzo_prog IPI = INDIRIZZO DELLA PROSSIMA ISTRUZIONE

istruzione  codice[999] ;//array[indirizzo_prog]o10f istruzione  CODICE = MEMORIA PROGRAMMA
OGGETTO

//indirizzo_prog; = 0..max_memoria_codice+1;

void errore(string messaggio);

void scanner();

void controlla(token tok, string err_message);

void genera(codice_operativo codop,int arg);

void termine();

void fattore();

void compila_espressioni();

void compila_cond();

void compila_istruzione();

void compila_programma();



void errore(string messaggio)

 {

   cout <<messaggio<<" al carattere "<<puntatoreCarattere<<"\n";

 };

void scanner()

{

 char car;int i=0;string parola;


  do

   {

    puntatoreCarattere++;
```

```
   car=SORGENTE[puntatoreCarattere];

   switch (car)
    {
     case ' ': ;break;
     case '{': do
               {
                 puntatoreCarattere++;
               }
              while ( SORGENTE[puntatoreCarattere]!='}');
              break;// le graffe racchiudono commenti
    }
   }
  while (car==' ' || car=='{');

 switch (car)
      {
        case 'A'...'Z':
           parola=car;
           while ((SORGENTE[puntatoreCarattere+1] >= '0' && SORGENTE[puntatoreCarattere+1] <=
'9')||
                 (SORGENTE[puntatoreCarattere+1] >= 'A' && SORGENTE[puntatoreCarattere+1] <=
'Z'))
              {
                parola=parola+SORGENTE[puntatoreCarattere+1];
                puntatoreCarattere++;
              };
           ts[ultimo+1]=parola;
           ix=0;
            while (ts[ix]!=parola)
              {
                ix++;
              }
```

```
     if (ix<=numeroparolechiave)   simb=token(ix);
         else  simb=ident;

break;

case '0'...'9':
   simb= costante;
   valoreCostante=(int)car-(int)'0';
   while (SORGENTE[puntatoreCarattere+1] >= '0' &&
        SORGENTE[puntatoreCarattere+1] <= '9')

    {
    puntatoreCarattere++;
    car=SORGENTE[puntatoreCarattere];
    valoreCostante=valoreCostante*10+(int)car-(int)'0';
    };
break;
case ':'    : if (SORGENTE[puntatoreCarattere+1]=='=')
             {
               simb=assegna;
               puntatoreCarattere++;
             }
               else errore("a ':' deve seguire '=' ");
break;
case '='    : simb=op_relaz;codice_op_relaz=1;
break;
case '<'    : simb= op_relaz;
           switch (SORGENTE[puntatoreCarattere+1])
            {
             case '>' : puntatoreCarattere++;
                      codice_op_relaz=1;
             break;
```

```
                case '=' : puntatoreCarattere++;
                        codice_op_relaz=2;
                break;
                default  : codice_op_relaz=3;
                break;
               };
        break;
        case '>'    : simb=op_relaz;
                if (SORGENTE[puntatoreCarattere+1]=='=')
                        {
                          puntatoreCarattere++;
                          codice_op_relaz=4;
                        }
                        else codice_op_relaz=5;
        break;
        case '+'    : simb= addiziona; break;
        case '-'    : simb= sottrai; break;
        case '*'    : simb= moltiplica; break;
        case '/'    : simb= dividi;break;
        case '('    : simb= tondaaperta;break;
        case ')'    : simb= tondachiusa;break;
        case ','    : simb= virgola;break;
        case ';'    : simb= puntovirgola;break;
        case '$'    : simb= finetesto;break;
        default     : errore("carattere sconosciuto");

      }
}



void controlla(token tok, string err_message)
```

```
{
  if (simb==tok) scanner();
    else errore(err_message);

}
void genera(codice_operativo codop,int arg)
{
  codice[ipi].cop=codop;
  codice[ipi].ind=arg;
  ipi++;
}
void fattore()
  {
    switch(simb)
    {
       case ident     :
              if (ix>ultimo) errore("variabile non dichiarata ");
              genera(LOAD,ix);
              scanner();
       break;
       case costante :
              scanner();
              genera(LIT,valoreCostante);
       break;
       case tondaaperta :
               scanner();
               compila_espressioni();
               controlla(tondachiusa," parentesi non bilanciate ");
       break;
       default : errore ("espressione scorretta ");
    }
}
```

```
void termine()
 {
  token operatoremul;
  fattore();
  while (simb==moltiplica || simb==dividi)
   {
    operatoremul=simb;
    scanner();
    fattore();
    if (operatoremul==moltiplica)  genera(MUL,0);
                          else genera(DIVID,0);
   }
 }
void compila_espressioni()
{
 token operatoreAdd;
 if (simb==sottrai)
   {
    scanner();
    termine();
    genera(NEG,0);
   }
    else termine();
 while  (simb==addiziona || simb||sottrai)
       {
         operatoreAdd=simb;
         scanner();
         termine();
         if (operatoreAdd==addiziona)  genera(ADD,0);
           else genera(SUB,0);
       }
```

```
}


void compila_cond()
{
 compila_espressioni();
 controlla(op_relaz,"atteso operatore relazionale ");
 compila_espressioni();
 genera(COMP,codice_op_relaz);
}
void compila_istruzione()
{
 int  aiuto,su,ivar;
 switch (simb)
  {
   case inizio  :
            do
             {
              scanner();
              compila_istruzione();
             }
            while (simb!=fine && simb==puntovirgola);
            if (simb==fine)  scanner();
                     else errore("manca ; ");
  break;
  case mentre  :
            scanner();
            su=ipi;
            compila_cond();
            controlla(fai," manca fai ");
            aiuto=ipi;
            genera(JMPF,0);
```

```
        compila_istruzione();
        genera(JMP,su);
        codice[aiuto].ind=ipi;
break;
case se     :
        scanner();
        compila_cond();
        controlla(allora," manca allora ");
        aiuto=ipi;
        genera(JMPF,0);
        compila_istruzione();
        codice[aiuto].ind=ipi+1;
        aiuto=ipi;
        genera(JMP,0);
        controlla(altrimenti," manca else ");
        compila_istruzione();
        codice[aiuto].ind=ipi;
break;
ident   :
        if (ix>ultimo) errore(" variabile non dichiarata ");
        ivar=ix;
        scanner();
        controlla(assegna," manca := ");
        compila_espressioni();
        genera(STORE,ivar);
break;
case leggi    :
         do
         {
          scanner();
          ivar=ix;
          controlla(ident, " atteso identificatore ");
```

```
            if (ivar>ultimo) errore(" variabile non dichiarata ");
            genera(LEGGERE,ivar);
           }
            while (simb==virgola);
   break;
   case scrivi   :
           do
           {
            scanner();
            controlla(ident, " atteso identificatore ");
            if (ivar>ultimo) errore(" variabile non dichiarata ");
            //compila_espressioni();
            genera(SCRIVERE,0);
           }
           while (simb==virgola);
   break;
   case fine    : ;
   break;
   case altrimenti : ;
   break;
   case puntovirgola : ;
   break;
   case finetesto: ;
   break;
   default : errore(" una istruzione non può cominciare cosi' \n");
   break;
  }
}


void compila_programma()
{
```

```
if (simb==variabile)
{
  do
  {
    scanner();
    controlla(ident," atteso identificatore");
    if (ix<=ultimo)  errore(" identificatore gia' dichiarato");
       else ultimo++;
  }
  while (simb==virgola);
  controlla(puntovirgola," manca ; ");
}
genera(LTR,ultimo);
compila_istruzione();
genera(ALT,0);
}
int main()
{
 system("cls");
 ts[0]="VARIABILE";ts[1]="INIZIO";ts[2]="FINE";ts[3]="MENTRE";
 ts[4]="FAI";ts[5]="SE";ts[6]="ALLORA";ts[7]="ALTRIMENTI";
 ts[8]="LEGGI";ts[9]="SCRIVI";
 ipi=0;ultimo=10;
 SORGENTE1="VARIABILE A,B,R,SALVA;INIZIO LEGGI A;  A=B;  SCRIVI A FINE$";
 SORGENTE2="MENTRE B<>0 FAI INIZIO R:=A-(A/B)*B; A:=B;B:=R FINE; SCRIVI A FINE$";
 SORGENTE=SORGENTE1+SORGENTE2;
 puntatoreCarattere=-1;
scanner();
compila_programma();
 for (int i=0; i<=ipi-1;i++)
   {
     switch (codice[i].cop)
```

```
{
 case LTR      : cout<<"("<<i<<")"<<"LTR"<<","<<codice[i].ind<<"  ";
 break;
 case ALT      : cout<<"("<<i<<")"<<"ALT"<<","<<codice[i].ind<<"  ";
 break;
 case JMP      : cout<<"("<<i<<")"<<"JMP"<<","<<codice[i].ind<<"  ";
 break;
 case JMPF     : cout<<"("<<i<<")"<<"JMPF"<<","<<codice[i].ind<<"  ";
 break;
 case JMPT     : cout<<"("<<i<<")"<<"JMPT"<<","<<codice[i].ind<<"  ";
 break;
 case LOAD     : cout<<"("<<i<<")"<<"LOAD"<<","<<codice[i].ind<<"  ";
 break;
 case LIT      : cout<<"("<<i<<")"<<"LIT"<<","<<codice[i].ind<<"  ";
 break;
 case STORE    : cout<<"("<<i<<")"<<"STORE"<<","<<ts[codice[i].ind]<<"  ";
 break;
 case LEGGERE  : cout<<"("<<i<<")"<<"LEGGERE"<<","<<ts[codice[i].ind]<<"  ";
 break;
 case SCRIVERE : cout<<"("<<i<<")"<<"SCRIVERE" <<"  ";
 break;
 case ADD      : cout<<"("<<i<<")"<<"ADD" <<"  ";
 break;
 case SUB      : cout<<"("<<i<<")"<<"SUB" <<"  ";
 break;
 case NEG      : cout<<"("<<i<<")"<<"NEG"<<"  ";
 break;
 case MUL      : cout<<"("<<i<<")"<<"MUL" <<"  ";
 break;
 case DIVID    : cout<<"("<<i<<")"<<"DIVID"  <<"  ";
 break;
 case COMP     : cout<<"("<<i<<")"<<"COMP" <<"  "<<codice[i].ind<<",";
```

```
     break;
    }

if ( i % 3 ==  0)
   {
    cout<<endl;
    for (int k=0; k<=75;k++) cout<<"-";
    cout<<endl;
   }
 }
```

**CODIFICA PASCAL**

```
{ INSERIRE IL CAPS-LOCK  }
program COMPILATORE_DIDATTICO;
uses crt;
const numeroparolechiave       =9;
     numero_max_identificatori=100;
     max_memoria_dati         =1000;
     max_memoria_codice       =1000;
type
  tabellasimbolirange=-numeroparolechiave..numero_max_identificatori+1;
  indirizzo_prog = 0..max_memoria_codice+1;
  indirizzo_dati = 1..max_memoria_dati+1;
  token=(variabile,inizio,fine,mentre,fai,se,allora,altrimenti,
        leggi,scrivi,ident,costante,assegna,addiziona,sottrai,moltiplica,
        dividi,op_relaz,tondaaperta,tondachiusa,virgola,puntovirgola,
        finetesto);
  codice_operativo=(LTR,ALT,JMP,JMPF,JMPT,LOAD,LIT,STORE,LEGGERE,SCRIVERE,
                ADD,SUB,NEG,MUL,DIVID,COMP);
  istruzione = record
               cop:codice_operativo;
               ind:integer
```

```
            end;

var
  simb                          :token;
  sorgente,sorgente1,sorgente2      :string;
  puntcarattere,valorecostante,i,k  :integer;
  ts                            :array[tabellasimbolirange] of string;
        { TS = TABELLA DEI SIMBOLI}
  ultimo                        :0..numero_max_identificatori;
        { ULTIMO = PUNTATORE ALL'ULTIMO ELEMENTO DI TS }
  ix                            :tabellasimbolirange;
        { IX = INDICE ALLA PAROLA CERCATA NELLA TS }
  num                           :integer;
  codice_op_relaz                   :0..5;
  ipi                           :indirizzo_prog;
        { IPI = INDIRIZZO DELLA PROSSIMA ISTRUZIONE }
  codice                        :array[indirizzo_prog]of istruzione;
        { CODICE = MEMORIA PROGRAMMA OGGETTO }

procedure errore(messaggio:string);
begin
  write(messaggio,' al carattere ',puntcarattere)
end;

procedure scanner;
var
  car:char;
  parola:string;
begin
  repeat
    puntcarattere:=puntcarattere+1;
    car:=sorgente[puntcarattere];
```

```
    case car of
      ' ':;
      '{':repeat puntcarattere:=puntcarattere+1
          until  sorgente[puntcarattere]='}';
    end;
until (car<>' ') and (car<>'{');
case car of
  'A'..'Z': begin
              parola:=car;
              while sorgente[puntcarattere+1] in ['A'..'Z','_','0'..'9'] do
              begin
                parola:=parola + sorgente[puntcarattere+1];
                puntcarattere:=puntcarattere+1
              end;
              ts[ultimo+1]:=parola;
              ix:=-numeroparolechiave;
              while ts[ix]<>parola do ix:=ix+1;
              if ix<=0 then   simb:=token(-ix)
                      else simb:=ident
            end;
  '0'..'9': begin
              simb := costante;
              valorecostante:=ord(car)-ord('0');
              while sorgente[puntcarattere+1] in ['0'..'9'] do
              begin
                puntcarattere:=puntcarattere+1;
                car:=sorgente[puntcarattere];
                valorecostante:=valorecostante*10+ord(car)-ord('0')
              end
            end;
  ':'     : if sorgente[puntcarattere+1]='=' then
                          begin
```

```
                    simb:=assegna;
                    puntcarattere:=puntcarattere+1
                   end
                              else errore('deve seguire = ');
'='   : begin  simb:=op_relaz;codice_op_relaz:=1 end;
'<'   : begin
        simb:= op_relaz;
        case
         sorgente[puntcarattere+1] of
         '>':begin puntcarattere:=puntcarattere+1;codice_op_relaz:=1 end;
         '=':begin puntcarattere:=puntcarattere+1;codice_op_relaz:=2 end;
         else codice_op_relaz:=3
        end
       end;
'>'   : begin
        simb:=op_relaz;
        if sorgente[puntcarattere+1]='=' then
                          begin
                           puntcarattere:=puntcarattere+1;
                           codice_op_relaz:=4
                          end
                         else
                           codice_op_relaz:=5
       end;
'+'   : simb:= addiziona;
'-'   : simb:= sottrai;
'*'   : simb:= moltiplica;
'/'   : simb:= dividi;
'('   : simb:= tondaaperta;
')'   : simb:= tondachiusa;
','   : simb:= virgola;
';'   : simb:= puntovirgola;
```

```
   '$'     : simb:= finetesto;
  else errore('carattere sconosciuto');
  end;
end;

procedure controlla(tok:token;err_message:string);
begin
  if simb=tok then scanner
              else errore(err_message);
end;

procedure genera(codop:codice_operativo;arg:integer);
begin
  codice[ipi].cop:=codop;
  codice[ipi].ind:=arg;
  ipi:=ipi+1
end;

procedure compila_espressioni;
var
  operatoreadd:token;
  procedure fattore;
  begin
    case simb of
    ident     : begin
                 if ix>ultimo then errore(' variabile non dichiarata ');
                 genera(LOAD,ix);
                 scanner
                end;
   costante : begin
                 scanner;
                 genera(LIT,valorecostante);
```

```
            end;
  tondaaperta : begin
                  scanner;
                  compila_espressioni;
                  controlla(TONDACHIUSA,' parentesi non bilanciate ');
                end;
   else errore (' espressione scorretta ')
   end
 end;
 procedure termine;
 var
  operatoremul:token;
 begin
   fattore;
   while simb in [moltiplica,dividi] do
     begin
       operatoremul:=simb;
       scanner;
       fattore;
       if operatoremul=moltiplica then genera(MUL,0)
                                  else genera(DIVID,0)
     end
 end;

begin
 if simb=sottrai then begin
                   scanner;
                   termine;
                   genera(NEG,0)
                  end
              else termine;
 while  simb in [addiziona,sottrai] do
```

```
            begin
              operatoreadd:=simb;
              scanner;
              termine;
              if operatoreadd=addiziona then genera(ADD,0)
                                        else genera(SUB,0)
            end
end;

procedure compila_cond;
begin
  compila_espressioni;
  controlla(op_relaz,' atteso operatore relazionale ');
  compila_espressioni;
  genera(comp,codice_op_relaz)
end;

procedure compila_istruzione;
var aiuto,su:indirizzo_prog;ivar:indirizzo_dati;
begin
  case simb of
    inizio  : begin
                repeat
                  scanner;
                  compila_istruzione;
                until (simb=fine) or (simb<>puntovirgola);
                if simb=fine then scanner
                             else errore('manca ; ')
              end;
    mentre  : begin
                scanner;
                su:=ipi;
```

```
          compila_cond;
          controlla(fai,' manca do ');
          aiuto:=ipi;
          genera(jmpf,0);
          compila_istruzione;
          genera(jmp,su);
          codice[aiuto].ind:=ipi
        end;
se      : begin
          scanner;
          compila_cond;
          controlla(allora,' manca then ');
          aiuto:=ipi;
          genera(jmpf,0);
          compila_istruzione;
          codice[aiuto].ind:=ipi+1;
          aiuto:=ipi;
          genera(jmp,0);
          controlla(altrimenti,' manca else ');
          compila_istruzione;
          codice[aiuto].ind:=ipi;
        end;
ident   : begin
          if ix>ultimo then errore(' variabile non dichiarata ');
          ivar:=ix;
          scanner;
          controlla(assegna,' manca := ');
          compila_espressioni;
          genera(store,ivar)
        end;
leggi   : repeat
           scanner;
```

```
            ivar:=ix;
            controlla(ident, ' atteso identificatore ');
            if ivar>ultimo then errore(' variabile non dichiarata ');
            genera(leggere,ivar);
           until simb<>virgola;
   scrivi   : repeat
            scanner;
            compila_espressioni;
            genera(scrivere,0);
           until simb<>virgola;
   fine,altrimenti,puntovirgola,finetesto:;
   else errore(' un"istruzione non pu• cominciare cos   ');
  end;
end;

procedure compila_programma;
begin
  if simb=variabile then
                  begin
                    repeat
                      scanner;
                      controlla(ident,' atteso identificatore');
            if ix<=ultimo then errore(' identificatore gi… dichiarato')
                        else ultimo:=ultimo+1;
                    until simb<>virgola;
                    controlla(puntovirgola,' manca ; ')
                  end;
  genera(ltr,ultimo);
  compila_istruzione;
  genera(alt,0);
end;
```

```
BEGIN
  clrscr;
  ts[0]:='VARIABILE';ts[-1]:='INIZIO';ts[-2]:='FINE';ts[-3]:='MENTRE';
  ts[-4]:='FAI';ts[-5]:='SE';ts[-6]:='ALLORA';ts[-7]:='ALTRIMENTI';
  ts[-8]:='LEGGI';ts[-9]:='SCRIVI';
  ipi:=0;ultimo:=0;
  { write('Inserisci testo sorgente : ');
  readln(sorgente);
  sorgente:=sorgente + '$';}
  SORGENTE1:='VARIABILE A,B,R,SALVA;INIZIO LEGGI A,B;SE A<B ALLORA INIZIO
SALVA:=A;A:=B;B:=SALVA FINE ALTRIMENTI; ';
  SORGENTE2:='MENTRE B<>0 FAI INIZIO R:=A-(A/B)*B;A:=B;B:=R FINE; SCRIVI A
FINE$';
  SORGENTE:=SORGENTE1+SORGENTE2;
  puntcarattere:=0;
  scanner;
  compila_programma;
  for i:=0 to ipi-1 do
  begin
    case codice[i].cop of
    LTR       : write('(',i:2,')',' LTR      ',codice[i].ind:8,'    ');
    ALT       : write('(',i:2,')',' ALT      ','            ');
    JMP       : write('(',i:2,')',' JMP      ',codice[i].ind:8,'    ');
    JMPF      : write('(',i:2,')',' JMPF     ',codice[i].ind:8,'    ');
    JMPT      : write('(',i:2,')',' JMPT     ',codice[i].ind:8,'    ');
    LOAD      : write('(',i:2,')',' LOAD     ',ts[codice[i].ind]:8,'    ');
    LIT       : write('(',i:2,')',' LIT      ',codice[i].ind:8,'    ');
    STORE     : write('(',i:2,')',' STORE    ',ts[codice[i].ind]:8,'    ');
    LEGGERE   : write('(',i:2,')',' LEGGERE  ',ts[codice[i].ind]:8,'    ');
    SCRIVERE  : write('(',i:2,')',' SCRIVERE ','            ');
    ADD       : write('(',i:2,')',' ADD      ','            ');
    SUB       : write('(',i:2,')',' SUB      ','            ');
    NEG       : write('(',i:2,')',' NEG      ','            ');
```

```
   MUL       : write('(',i:2,')',' MUL      ',','            ');
   DIVID     : write('(',i:2,')',' DIVID    ',','            ');
   COMP      : write('(',i:2,')',' COMP     ',codice[i].ind:8,'    ');
   end;
   if i mod 3 =  0 then begin
                      writeln;
                      for k:=0 to 75 do write('-');
                      writeln
                     end;
  end;
  readln;
END.
```

***ITIS-LS "Francesco Giordani" Caserta***
***Anno scolastico:*** *2019/2020*
***Classe 4^ sez.B spec. Informatica e telecomunicazioni***
***Data:***
***Numero progressivo dell'esercizio:*** *es5*
***Versione:*** *1.0*
***Programmatore/i:***
***Sistema Operativo:*** *Windows 10*
***Compilatore/Interprete:*** *Code::Blocks 17.12*
***Obiettivo didattico:***
*Compilatore didattico*
***Obiettivo del programma:*** *realizzare un semplice compilatore ed interprete*

```
{ INSERIRE IL CAPS-LOCK  }

program COMPILATORE_DIDATTICO;

uses crt;

const numeroparolechiave       =9;

     numero_max_identificatori=100;

     max_memoria_dati         =1000;

     max_memoria_codice       =1000;

type

  tabellasimbolirange=-numeroparolechiave..numero_max_identificatori+1;

  indirizzo_prog = 0..max_memoria_codice+1;

  indirizzo_dati = 1..max_memoria_dati+1;
```

```
  token=(variabile,inizio,fine,mentre,fai,se,allora,altrimenti,
        leggi,scrivi,ident,costante,assegna,addiziona,sottrai,moltiplica,
        dividi,op_relaz,tondaaperta,tondachiusa,virgola,puntovirgola,
        finetesto);
  codice_operativo=(LTR,ALT,JMP,JMPF,JMPT,LOAD,LIT,STORE,LEGGERE,SCRIVERE,
                ADD,SUB,NEG,MUL,DIVID,COMP);
  istruzione = record
               cop:codice_operativo;
               ind:integer
              end;

var
  simb                          :token;
  sorgente,sorgente1,sorgente2        :string;
  puntcarattere,valorecostante,i,k    :integer;
  ts                            :array[tabellasimbolirange] of string;
        { TS = TABELLA DEI SIMBOLI}
  ultimo                         :0..numero_max_identificatori;
        { ULTIMO = PUNTATORE ALL'ULTIMO ELEMENTO DI TS }
  ix                            :tabellasimbolirange;
        { IX = INDICE ALLA PAROLA CERCATA NELLA TS }
  num                           :integer;
  codice_op_relaz                     :0..5;
  ipi                           :indirizzo_prog;
        { IPI = INDIRIZZO DELLA PROSSIMA ISTRUZIONE }
  codice                        :array[indirizzo_prog]of istruzione;
        { CODICE = MEMORIA PROGRAMMA OGGETTO }
```

```
procedure errore(messaggio:string);
begin
  write(messaggio,' al carattere ',puntcarattere)
end;

procedure scanner;
var
  car:char;
  parola:string;
begin
  repeat
    puntcarattere:=puntcarattere+1;
    car:=sorgente[puntcarattere];
      case car of
        ' ':;
        '{':repeat puntcarattere:=puntcarattere+1
            until  sorgente[puntcarattere]='}';
      end;
  until (car<>' ') and (car<>'{');
  case car of
    'A'..'Z': begin
                parola:=car;
                while sorgente[puntcarattere+1] in ['A'..'Z','_','0'..'9'] do
                begin
                  parola:=parola + sorgente[puntcarattere+1];
                  puntcarattere:=puntcarattere+1
```

```
            end;
            ts[ultimo+1]:=parola;
            ix:=-numeroparolechiave;
            while ts[ix]<>parola do ix:=ix+1;
            if ix<=0 then   simb:=token(-ix)
                  else simb:=ident
          end;
'0'..'9': begin
            simb := costante;
            valorecostante:=ord(car)-ord('0');
            while sorgente[puntcarattere+1] in ['0'..'9'] do
            begin
              puntcarattere:=puntcarattere+1;
              car:=sorgente[puntcarattere];
              valorecostante:=valorecostante*10+ord(car)-ord('0')
            end
           end;
':'     : if sorgente[puntcarattere+1]='=' then
                        begin
                          simb:=assegna;
                          puntcarattere:=puntcarattere+1
                        end
                              else errore('deve seguire = ');
'='     : begin  simb:=op_relaz;codice_op_relaz:=1 end;
'<'     : begin
            simb:= op_relaz;
            case
```

```
            sorgente[puntcarattere+1] of
            '>':begin puntcarattere:=puntcarattere+1;codice_op_relaz:=1 end;
            '=':begin puntcarattere:=puntcarattere+1;codice_op_relaz:=2 end;
            else codice_op_relaz:=3
          end
        end;
 '>'    : begin
          simb:=op_relaz;
          if sorgente[puntcarattere+1]='=' then
                          begin
                            puntcarattere:=puntcarattere+1;
                            codice_op_relaz:=4
                          end
                         else
                            codice_op_relaz:=5
        end;
 '+'    : simb:= addiziona;
 '-'    : simb:= sottrai;
 '*'    : simb:= moltiplica;
 '/'    : simb:= dividi;
 '('    : simb:= tondaaperta;
 ')'    : simb:= tondachiusa;
 ','    : simb:= virgola;
 ';'    : simb:= puntovirgola;
 '$'    : simb:= finetesto;
else errore('carattere sconosciuto');
end;
```

```
end;

procedure controlla(tok:token;err_message:string);
begin
  if simb=tok then scanner
              else errore(err_message);
end;

procedure genera(codop:codice_operativo;arg:integer);
begin
  codice[ipi].cop:=codop;
  codice[ipi].ind:=arg;
  ipi:=ipi+1
end;

procedure compila_espressioni;
var
  operatoreadd:token;
  procedure fattore;
  begin
    case simb of
    ident     : begin
                  if ix>ultimo then errore(' variabile non dichiarata ');
                  genera(LOAD,ix);
                  scanner
                end;
   costante : begin
```

```
                scanner;
                genera(LIT,valorecostante);
              end;
  tondaaperta : begin
                scanner;
                compila_espressioni;
                controlla(TONDACHIUSA,' parentesi non bilanciate ');
              end;
   else errore (' espressione scorretta ')
   end
 end;
 procedure termine;
 var
  operatoremul:token;
 begin
   fattore;
   while simb in [moltiplica,dividi] do
     begin
       operatoremul:=simb;
       scanner;
       fattore;
       if operatoremul=moltiplica then genera(MUL,0)
                                  else genera(DIVID,0)
     end
 end;


begin
```

```
  if simb=sottrai then begin
                    scanner;
                    termine;
                    genera(NEG,0)
                   end
              else termine;
  while  simb in [addiziona,sottrai] do
                   begin
                    operatoreadd:=simb;
                    scanner;
                    termine;
                    if operatoreadd=addiziona then genera(ADD,0)
                                              else genera(SUB,0)
                   end
end;

procedure compila_cond;
begin
  compila_espressioni;
  controlla(op_relaz,' atteso operatore relazionale ');
  compila_espressioni;
  genera(comp,codice_op_relaz)
end;

procedure compila_istruzione;
var aiuto,su:indirizzo_prog;ivar:indirizzo_dati;
begin
```

```
case simb of
  inizio  : begin
              repeat
                scanner;
                compila_istruzione;
              until (simb=fine) or (simb<>puntovirgola);
              if simb=fine then scanner
                           else errore('manca ; ')
            end;
  mentre  : begin
              scanner;
              su:=ipi;
              compila_cond;
              controlla(fai,' manca do ');
              aiuto:=ipi;
              genera(jmpf,0);
              compila_istruzione;
              genera(jmp,su);
              codice[aiuto].ind:=ipi
            end;
  se      : begin
              scanner;
              compila_cond;
              controlla(allora,' manca then ');
              aiuto:=ipi;
              genera(jmpf,0);
              compila_istruzione;
```

```
          codice[aiuto].ind:=ipi+1;
          aiuto:=ipi;
          genera(jmp,0);
          controlla(altrimenti,' manca else ');
          compila_istruzione;
          codice[aiuto].ind:=ipi;
        end;
ident   : begin
          if ix>ultimo then errore(' variabile non dichiarata ');
          ivar:=ix;
          scanner;
          controlla(assegna,' manca := ');
          compila_espressioni;
          genera(store,ivar)
        end;
leggi   : repeat
           scanner;
           ivar:=ix;
           controlla(ident, ' atteso identificatore ');
           if ivar>ultimo then errore(' variabile non dichiarata ');
           genera(leggere,ivar);
         until simb<>virgola;
scrivi  : repeat
           scanner;
           compila_espressioni;
           genera(scrivere,0);
         until simb<>virgola;
```

```
   fine,altrimenti,puntovirgola,finetesto:;
   else errore(' un''istruzione non pu• cominciare cos   ');
  end;
end;

procedure compila_programma;
begin
  if simb=variabile then
                    begin
                      repeat
                        scanner;
                        controlla(ident,' atteso identificatore');
               if ix<=ultimo then errore(' identificatore gi… dichiarato')
                          else ultimo:=ultimo+1;
                      until simb<>virgola;
                      controlla(puntovirgola,' manca ; ')
                    end;
  genera(ltr,ultimo);
  compila_istruzione;
  genera(alt,0);
end;

procedure esegui;
var
  memoria_dati :array[indirizzo_dati] of integer;
  ir          :istruzione;                  {registro istruzioni}
  ic          :indirizzo_prog;              {contatore istruzioni}
```

```
  tr          :indirizzo_dati;                { registro limite}
  cr,zerodividi:boolean;                        { registro condizione}
begin
  zerodividi:=false;
  ic:=0;
  repeat
    ir:=codice[ic];
    ic:=ic+1;
    with ir do
    case cop of
      LTR      :tr:=ind;
      ALT      :;
      JMP      :ic:=ind;
      JMPF     :if not cr then ic:=ind;
      JMPT     :if     cr then ic:=ind;
      LEGGERE  :readln (memoria_dati[ind]);
      SCRIVERE :begin write (memoria_dati[tr]); tr:=tr-1 end;
      ADD      :begin
                  memoria_dati[tr-1]:=memoria_dati[tr-1]+memoria_dati[tr];
                  tr:=tr-1
                end;
     SUB       :begin
                  memoria_dati[tr-1]:=memoria_dati[tr-1]-memoria_dati[tr];
                  tr:=tr-1
                end;
     MUL       :begin
                   memoria_dati[tr-1]:=memoria_dati[tr-1]*memoria_dati[tr];
```

```
          tr:=tr-1
        end;
DIVID    :if memoria_dati[tr]=0 then
                          begin
                            write(' DIVISIONE PER ZERO ');
                            readln;
                            zerodividi:=true
                          end
                        else
        begin
          memoria_dati[tr-1]:=memoria_dati[tr-1] div memoria_dati[tr];
          tr:=tr-1
        end;
NEG      :memoria_dati[tr]:=-memoria_dati[tr];
LIT      :begin tr:=tr+1; memoria_dati[tr]:=ind end;
LOAD     :begin tr:=tr+1; memoria_dati[tr]:=memoria_dati[ind] end;
STORE    :begin memoria_dati[ind]:=memoria_dati[tr]; tr:=tr-1 end;
COMP     :begin
          case ind of
            0:cr:=(memoria_dati[tr-1]=memoria_dati[tr]);
            1:cr:=(memoria_dati[tr-1]<>memoria_dati[tr]);
            2:cr:=(memoria_dati[tr-1]<=memoria_dati[tr]);
            3:cr:=(memoria_dati[tr-1]<memoria_dati[tr]);
            4:cr:=(memoria_dati[tr-1]>=memoria_dati[tr]);
            5:cr:=(memoria_dati[tr-1]>memoria_dati[tr]);
          end;
          tr:=tr-2
```

```
            end
   end
  until (codice[ic].cop=ALT) or zerodividi;
end;

BEGIN
  clrscr;
  ts[0]:='VARIABILE';ts[-1]:='INIZIO';ts[-2]:='FINE';ts[-3]:='MENTRE';
  ts[-4]:='FAI';ts[-5]:='SE';ts[-6]:='ALLORA';ts[-7]:='ALTRIMENTI';
  ts[-8]:='LEGGI';ts[-9]:='SCRIVI';
  ipi:=0;ultimo:=0;
  { write('Inserisci testo sorgente : ');
  readln(sorgente);
  sorgente:=sorgente + '$';}
  SORGENTE1:='VARIABILE A,B,R,SALVA;INIZIO LEGGI A,B;SE A<B ALLORA INIZIO
SALVA:=A;A:=B;B:=SALVA FINE ALTRIMENTI; ';
  SORGENTE2:='MENTRE B<>0 FAI INIZIO R:=A-(A/B)*B;A:=B;B:=R FINE; SCRIVI A
FINE$';
  SORGENTE:=SORGENTE1+SORGENTE2;
  puntcarattere:=0;
  scanner;
  compila_programma;
  for i:=0 to ipi-1 do
  begin
    case codice[i].cop of
    LTR       : write('(',i:2,')',' LTR      ',codice[i].ind:8,'    ');
    ALT       : write('(',i:2,')',' ALT      ','            ');
    JMP       : write('(',i:2,')',' JMP      ',codice[i].ind:8,'    ');
```

```
  JMPF      : write('(',i:2,')',' JMPF     ',codice[i].ind:8,'    ');
  JMPT      : write('(',i:2,')',' JMPT     ',codice[i].ind:8,'    ');
  LOAD      : write('(',i:2,')',' LOAD     ',ts[codice[i].ind]:8,'    ');
  LIT       : write('(',i:2,')',' LIT      ',codice[i].ind:8,'    ');
  STORE     : write('(',i:2,')',' STORE    ',ts[codice[i].ind]:8,'    ');
  LEGGERE   : write('(',i:2,')',' LEGGERE  ',ts[codice[i].ind]:8,'    ');
  SCRIVERE  : write('(',i:2,')',' SCRIVERE ','           ');
  ADD       : write('(',i:2,')',' ADD      ','           ');
  SUB       : write('(',i:2,')',' SUB      ','           ');
  NEG       : write('(',i:2,')',' NEG      ','           ');
  MUL       : write('(',i:2,')',' MUL      ','           ');
  DIVID     : write('(',i:2,')',' DIVID    ','           ');
  COMP      : write('(',i:2,')',' COMP     ',codice[i].ind:8,'    ');
  end;
  if i mod 3 =  0 then begin
                      writeln;
                      for k:=0 to 75 do write('-');
                      writeln
                     end;
 end;
 write('premi un tasto');
 repeat until keypressed;
 esegui;
 readln
END.
```