GNU MARST

GNU Algol-to-C Translator
Version 2.3
User's Guide
October 2001

Andrew Makhorin

Table of Contents

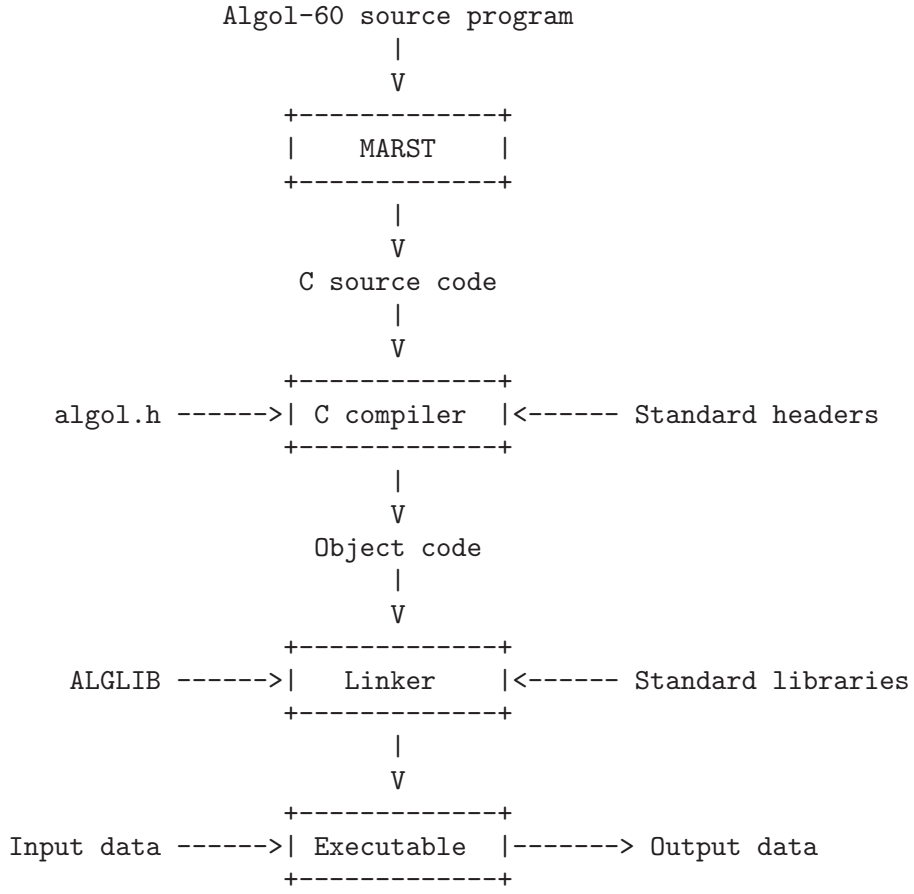
Acknowledgements

The author thanks Erik Schönfelder <schoenfr@gaertner.de> for a lot of useful advices and especially for testing MARST with real Algol 60 programs. The author also thanks Bernhard Treutwein <Bernhard.Treutwein@Verwaltung.Uni-Muenchen.DE> for great help in preparing MARST documentation.

1 Introduction

GNU MARST is an Algol-to-C translator. It automatically translates programs written in the algorithmic language Algol 60 into the ANSI C programming language.

Processing scheme can be understood as the following:



where:

Algol 60 source program
: a text file that contains a program written in the algorithmic language Algol 60 (see below about coding requirements);

MARST
: the MARST translator, a program that converts source Algol program to the C programming language. This program is a part of GNU MARST;

C source code
: a text file that contains the C source code generated by the MARST translator;

`algol.h`
: the header file that contains declarations of all objects used by every program generated by the MARST translator. This file includes some standard headers (`stdio.h`, `stdlib.h`, etc.), however, no other headers are used explicitly in the generated code. This file is a part of GNU MARST;

Standard headers
: standard header files (they are used only in the header file `algol.h`);

C compiler
: C compiler, a program that converts C program to machine instructions;

Object code
: a binary file that contains object code produced by the C compiler;

ALGLIB
: the library (archive) file that contains object code for all standard and library routines used by Algol programs. Some of these routines, which correspond to standard Algol procedures (*ininteger*, *outreal*, etc.) are written in Algol 60 and translated to the C programming language by means of the MARST translator. Source code of all library routines is a part of GNU MARST. In this distribution the library has the name `libalgol.a`;

Standard libraries
: standard C run-time libraries;

Linker
: linker, a program that resolves external references and produces executable module;

Executable
: a binary file that contains ready-to-run Algol 60 program in the loadable (executable) form;

Input data
: input text file(s) read by Algol program;

Output data
: output text file(s) written by Algol program.

2 Installation

In order to install GNU MARST under GNU/Linux the standard installation procedure should be used. For details see the file `INSTALL` included in the distribution.

As a result of installation the following four components will be installed:

`marst` as a rule, into `usr/local/bin`;

`macvt` as a rule, into `usr/local/bin`;

`algol.h` as a rule, into `usr/local/include` and/or `usr/include`;

`libalgol.a`
as a rule, into `usr/local/lib`.

3 Program Invocation

In order to invoke the MARST translator the following syntax should be used:

`marst` [*options ...*] [*filename*]

Options:

`-d`, `--debug`
: run translator in debug mode

  If this option is set, the translator emits elementary syntactic units of source Algol program to the output C code in the form of comments.

  This option is useful for localizing syntax errors more precisely. For example, Algol 60 allows comments of three kinds: ordinary comments, **end-end** comments, and extended parameter delimiters. Therefore it is easy to make a mistake, for example, forgetting a comma between the **end** bracket and the next statement.

`-e` *nnn*, `--error-max` *nnn*
: maximal error allowance

  This option sets maximal error allowance. The translator stops processing after the specified number of errors has been detected. The value of *nnn* should be in the range from 0 to 255. If this option is not specified, the default option `-e 0` is used and means that the translation is continued until the end of the input file.

`-h`, `--help`
: display help information and `exit(0)`

`-l` *nnn*, `--linewidth` *nnn*
: desirable output line width

  This option sets desirable line width for the output C code produced by the translator. The value *nnn* should be in the range from 50 to 255. If this option is not specified, the default option `-l 72` is used.

  Note that the actual line width may happen to be greater than *nnn*, because the translator is not able to break the output text at any place. However, this happens relatively seldom.

`-o` *filename*, `--output` *filename*
: name of the output text file, to which the translator sends the produced C code

  If this option is not set, the translator uses the standard output by default.

`-t`, `--notimestamp`
: don't write the time stamp to the output C code

  By default the translator writes date and time of translation to the output C code as a comment.

`-v`, `--version`
: display translator version and `exit(0)`

`-w`, `--nowarn`

don't display warning messages

By default the translator displays warning messages that reflect potential errors and non-standard features used in the source Algol program.

In order to translate a program written in Algol 60, it should be prepared as a plain text file, and the name of this file should be specified in the command line. If the name of the input text file is not specified, the translator uses the standard input by default.

Note that the translator reads the input file *twice*, therefore this file should be only regular file, but not a pipe, terminal input, etc. Thus, if the standard input is used, it should be redirected to a regular file.

For one run the translator is able to process only one input text file.

4 Usage Example

The following example shows how the MARST translator may be used in most cases. At first we prepare source Algol 60 program, say, in the text file named '`hello.alg`':

```
begin
   outstring(1, "Hello, world!\n")
end
```

Now we translate this program to the C programming language:

```
marst hello.alg -o hello.c
```

and get the text file named '`hello.c`', which then we compile and link in the usual way (we should remember about Algol and math libraries):

```
gcc hello.c -lalgol -lm -o hello
```

Finally, we run executable

```
./hello
```

and see what we have. That's all.

5 Input Language

The input language of the MARST translator is hardware representation of the reference language Algol 60 described in the following IFIP document[1]:

Modified Report on the Algorithmic Language ALGOL 60. *The Computer Journal*, Vol. 19, No. 4, Nov. 1976, pp. 364—79. (This document is an official IFIP standard. It is *not* a part of GNU MARST.)

Source Algol 60 program is coded as a plain text file using ASCII character set.

Basic symbols should be coded as follows:

Basic symbol	Representation	Basic symbol	Representation
$a, b, \ldots, z$	`a, b, ..., z`	]	`]`
$A, B, \ldots, Z$	`A, B, ..., Z`	‘	`"`
$0, 1, \ldots, 9$	`0, 1, ..., 9`	’	`"`
$+$	`+`	**array**	`array`
$-$	`-`	**begin**	`begin`
$\times$	`*`	**Boolean**	`Boolean (boolean)`
$/$	`/`	**code**	`code`
$\div$	`%`	**comment**	`comment`
$\uparrow$	`^ (**)`	**do**	`do`
$<$	`<`	**else**	`else`
$\leq$	`<=`	**end**	`end`
$=$	`=`	**false**	`false`
$\geq$	`>=`	**for**	`for`
$\neq$	`!=`	**go to**	`go to (goto)`
$\equiv$	`==`	**if**	`if`
$\supset$	`->`	**integer**	`integer`
$\vee$	`|`	**label**	`label`
$\wedge$	`&`	**own**	`own`
$\neg$	`!`	**procedure**	`procedure`
,	`,`	**real**	`real`
.	`.`	**step**	`step`
10	`#`	**string**	`string`
:	`:`	**switch**	`switch`
;	`;`	**then**	`then`
:=	`:=`	**true**	`true`
(	`(`	**until**	`until`
)	`)`	**value**	`value`
[	`[`	**while**	`while`

Any symbol can be surrounded by any number of white-space characters (i.e. by spaces, `HT`, `CR`, `LF`, `FF`, and `VT`). However, any multi-character symbol should contain no white-space characters. Moreover, a letter sequence is recognized as a keyword if and only if there is

[1] In order to obtain the verbatim reprint of this document in Postscript or in PDF format please contact the author.

no letter or digit that immediately precedes or follows the sequence (except the keyword '`go to`' that may contain zero or more spaces between '`go`' and '`to`').

For example:

`... 123 then abc ...`
: '`then`' will be recognized as **then** symbol

`... 123then abc ...`

`... 123 thenabc ...`
: '`then`' will be recognized as letters *t*, *h*, *e*, *n*, but not as **then** symbol

`... 123 th en abc ...`
: '`th en`' will be recognized as letters *t*, *h*, *e*, *n*

Note that identifiers and numbers can contain white-space characters. This feature may be used in the case if an identifier is the same as keyword. For example, identifier *label* may be coded as '`la bel`' or '`lab el`'. Note also that white-space characters are non-significant (except when they are used within character strings), so '`abc`' and '`a b c`' denote the same identifier *abc*.

Identifiers and numbers can consist of *arbitrary* number of characters, all of which (except internal white-space characters) are significant.

All letters are case sensitive (except the first `"b"` in the keyword **Boolean**). This means that '`abc`' and '`ABC`' are different identifiers, and '`Then`' will not be recognized as the keyword **then**.

Quoted character string are coded in the C style. For example:

```
outstring(1, "This\tis a string\n");

outstring(1, "This\tis a st"   "ring\n");

outstring(1, "This\tis all one st"
   "ring\n");
```

Within a string (i.e. between double quotes that enclose the string body) escape sequences may be used (as '`\t`' and '`\n`' in the example above). Double quote and backslash within string should be coded as '`\"`' and '`\\`' respectively. Between parts of a string any number of white-space characters is allowed.

Except coding character strings there are no other differences between the syntax of the reference language and the syntax of GNU MARST input language.

Note that there are some differences between the Revised Report on Algol 60 and the Modified Report on Algol 60, because the latter is a result of application of the following IFIP document to the former:

R. M. De Morgan, I. D. Hill, and B. A. Whichman. A Supplement to the ALGOL 60 Revised Report. *The Computer Journal*, Vol. 19, No. 3, 1976, pp. 276—88. (This document is an official IFIP standard. It is *not* a part of GNU MARST.)

In order to illustrate what is the input language of the MARST translator let's consider the following procedure, which is written in the reference language:

```
real procedure euler(fct, eps, tim);
   value eps, tim;
   real procedure fct; real eps; integer tim;
   comment euler computes the sum of fct(i) for i from zero up to
   infinity by means of a suitably refined euler transformation. The
   summation is stopped as soon as tim times in succession the absolute
   value of the terms of the transformed series are found to be less
   than eps. Hence one should provide a function fct with one integer
   argument, an upper bound eps, and an integer tim. euler is
```

```
   particularly efficient in the case of a slowly convergent or
   divergent alternating series;
   begin
   integer i, k, n, t;
   array m[0:15];
   real mn, mp, ds, sum;
   n := t := 0;
   m[0] := fct(0); sum := m[0] / 2;
   for i := 1, i+1 while t < tim do
      begin
      mn := fct(i);
      for k := 0 step 1 until n do
         begin
         mp := (mn + m[k]) / 2;
         m[k] := mn; mn := mp
      end means;
   if abs(mn) < abs(m[n]) & n < 15 then
      begin
      ds := mn / 2; n := n + 1;
      m[n] := mn
      end accept
   else
      ds := mn;
   sum := sum + ds;
   t := if abs(ds) < eps then t + 1 else 0
   end;
euler := sum
end euler;
```

6 Input/Output

All input/output is performed by standard Algol 60 procedures.

GNU MARST implementation provides up to 16 input/output channels, which have numbers 0, 1, . . ., 15. The channel number 0 is always connected to `stdin`, so only input from this channel is allowed. Analogously, the channel number 1 is always connected to `stdout`, so only output to this channel is allowed. Other channels allow both input and output. (The standard procedure *fault* uses the channel number Σ, which is not available to the programmer. This latent channel is always connected to `stderr`.)

Before Algol program startup all channels (except the channels number 0 and 1) are disconnected, i.e. no files are assigned to them.

If input (output) is required by the Algol program from (to) the channel number n, the following actions are taken:

1. if the channel number n is connected for output (input), the I/O routine closes the file assigned to this channel, making it be disconnected;
2. if the channel number n is disconnected, the I/O routine opens the corresponding file in read (write) mode and assigns this file to the channel, making it be connected;
3. finally, the I/O routine performs an input (an output) operation on the channel number n. If an end-of-file has been detected, the I/O routine signals an error condition and terminates execution of the Algol program.

In order to determine the name of file, which should be assigned to the channel number n, the I/O routine checks for environment variable named '`FILE_n`'. If such variable exists, its value is used as filename. Otherwise, its name (i.e. the character string `"FILE_n"`) is used as filename.

7 Language Extensions

The MARST translator provides some extensions of the reference language in order to make the package be more convenient for the programmer.

7.1 Modular programming

The feature of modular programming can be illustrated by the following example:

First file

```
procedure one(a, b);
value a, b; real a, b;
begin
.  .  .  .  .  .
end;

procedure two(x, y);
value x, y; array x, y;
begin
.  .  .  .  .  .
end;
```

Second file

```
procedure one(a, b);
value a, b; real a, b;
code;

procedure two(x, y);
value x, y; array x, y;
code;

begin
<main program>
end;
```

The procedures *one* and *two* in the first file are called *precompiled procedures*. Declarations of precompiled procedures should be outside of main program block or compound statement. The procedures *one* and *two* in the second file are called *code procedures*; they have the keyword **code** instead a procedure body statement. Declarations of code procedures also should be outside of main program block or compound statement.

This mechanism allows translating precompiled procedures independently on the main program. Moreover, precompiled procedures may be programmed in any other C compatible programming language. The programmer can consider that directly before Algol program startup declarations of all precompiled procedures are substituted into the file, which contains main program (the second file in the example above), instead declarations of corresponding code procedures.

Each code procedure should have the same procedure heading as the corresponding precompiled procedure (however, names of parameters may be altered). Note that mismatched procedure headings can't be detected by the MARST translator, because they are placed in different files.

7.2 Pseudo procedure *inline*

The pseudo procedure *inline* has the following (implicit) heading:

```
procedure inline(str);
string str;
```

A procedure statement that refers to the *inline* pseudo procedure is translated into the code, which is the string *str* without enclosing quotes. For example:

Source program	Output C code
`. . .`	`. . .`
`a := 1;`	`dsa_0->a_5 = 1;`
`b := 2;`	`dsa_0->b_8 = 2;`
`inline("printf(\"OK\");");`	`printf("OK");`
`c := 3;`	`dsa_0->c_4 = 3;`
`. . .`	`. . .`

Procedure statement *inline* may be used anywhere in the program as an oridinary Algol statement.

7.3 Pseudo procedure *print*

The pseudo procedure *print* is intended mainly for test printing (because standard Algol input/output is out of criticism). This procedure has unspecified heading and variable parameter list. For example:

```
real a, b; integer c; Boolean d;
array u, v[1:10], w[-5:5,-10:10];
. . .
print(a, b, u);
print(c);
. . .
print("test shot", (a+b)*c, !d & u[1] > v[1], u, v, w);
. . .
```

Each actual parameter passed to the pseudo procedure *print* is sent to the channel number 1 (`stdout`) using printable format.

8 Converter Utility

Algol converter utility is MACVT. It is an auxiliary program, which is intended for converting Algol 60 programs from some other representation to the MARST representation. Such conversion is usually needed when existing Algol programs should be adjusted in order to translate them with GNU MARST.

MACVT is not a translator itself. This program just reads an original code of Algol 60 program from the input text file, converts main symbols to the MARST representation (see Section 5. Input Language), and writes resulting code to the output text file. It is assumed that the output code produced by MACVT will be later translated by MARST in usual way. Note that MACVT performs no syntax checking.

Input language understood by MACVT differs from GNU MARST input language only in representation of basic symbols. Should note that in this sense GNU MARST input language is a subset of the MACVT input language.

Representation of basic symbols implemented in MACVT is based mainly on well known (in 1960s) Algol 60 compiler developed by IBM first for IBM 7090 and later for System/360. This representation may be considered as non-official standard, because it was widely used at the time, when Algol 60 was actual programming language.

In order to invoke the MACVT converter the following syntax should be used:

`macvt` [*options* ...] [*filename*]

Options:

`-c`, `--classic`
: use classic representation

  This option is used by default until other representation is chosen. It assumes that input Algol 60 program is coded using classic representation: all white-space characters are non-significant (except within quoted character strings) and keywords should be enclosed by apostrophes. For details see below.

`-f`, `--free-coding`
: use free representation

  If this option is set, it is allowed not to enclose keywords by apostrophes. But in this case white-space characters should not be used within multi-character basic symbols. See below for details.

`-h`, `--help`
: display help information and `exit(0)`

`-i`, `--ignore-case`
: convert letters to lower case

  If this option is set, all letters (except within comments and character strings) are converted to lower case, i.e. conversion is case-insensitive.

`-m`, `--more-free`
: use more free representation

  This option is the same as `--free-coding`, but additionally keywords for arithmetic, logical, and relational operators can be coded without apostrophes. For details see below.

`-o` *filename*, `--output` *filename*
: the name of output text file, to which the converter sends the converted Algol 60 program

  If this option is not set, the converter uses the standard output by default.

`-s`, `--old-sc`
: use old (classic) semicolon representation

  This option allows the converter recognizing diphthong ., (point and comma) as semicolon (including its usage for terminating comment sequences).

`-t`, `--old-ten`
: use old (classic) ten symbol representation

  This option allows the converter recognizing single apostrophe (when it is followed by `+`, `-`, or digit) as ten symbol.

`-v`, `--version`
: display the converter version and `exit(0)`

In order to convert an Algol 60 program, it should be prepared as a plain text file, and the name of this file should be specified in the command line. If the name of the input text file is not specified, the converter uses the standard input by default.

For one run the converter is able to process only one input text file.

In the table shown on the next page one or more valid representation are given for each basic symbol. Thereto the following additional rules are assumed:

1. Classic (apostrophized) form of keywords and some other basic symbols are allowed for any (i.e. for classic as well as free) representation.
2. In case of classic representation all white-space characters (except their usage within comments and quoted strings) are ignored anywhere.
3. Basic symbol enclosed by apostrophes may contain white-space characters, which are ignored. Besides, all letters are case-insensitive.
4. Basic symbols may be coded in the free form (without apostrophes) only if the free representation (`--free-coding` or `--more-free`) is used.
5. In case of free representation any multi-character basic symbol should contain no white-space characters.
6. Free form of keywords that denote arithmetic, logical, and relational operators (i.e. `greater` instead `'greater'`) is allowed only if the option `--more-free` is used.
7. Single apostrophe is recognized as ten symbol only if the option `--old-ten` is used. Note that in this case the sequence `'10'` is not recognized as ten symbol.
8. Diphthong ., (point and comma) is recognized as semicolon only in the case if the option `--old-sc` is used.
9. If an opening quote is coded as `"` (double quote), the corresponding closing quote should be coded as `"` (double quote). If an opening quote is coded as `‘` (diacritic mark), the corresponding closing quote should be coded as `’` (single apostrophe).

Basic symbol	Representation				Basic symbol	Representation
$a, b, \ldots, z$	a, b, ..., z				]	] /)
$A, B, \ldots, Z$	A, B, ..., Z				‘	" ‘
$0, 1, \ldots, 9$	0, 1, ..., 9				’	’ ’
$+$	+				**array**	’array’
$-$	-				**begin**	’begin’
$\times$	*				**Boolean**	’boolean’
$/$	/				**code**	’code’
$\div$	%	’/’		’div’	**comment**	’comment’
$\uparrow$	^ **	’power’		’pow’	**do**	’do’
$<$	<	’less’			**else**	’else’
$\leq$	<=	’notgreater’			**end**	’end’
$=$	=	’equal’			**false**	’false’
$\geq$	>=	’notless’			**for**	’for’
$\neq$	!=	’notequal’			**go to**	’goto’
$\equiv$	==	’equiv’			**if**	’if’
$\supset$	->	’impl’			**integer**	’integer’
$\vee$	|	’or’			**label**	’label’
$\wedge$	&	’and’			**own**	’own’
$\neg$	!	’not’			**procedure**	’procedure’
,	,				**real**	’real’
.	.				**step**	’step’
$_{10}$	#	’		’10’	**string**	’string’
:	:	..			**switch**	’switch’
;	;	.,			**then**	’then’
$:=$	:=	.=		..=	**true**	’true’
(	(				**until**	’until’
)	)				**value**	’value’
[	[	(/			**while**	’while’

In order to illustrate what the MACVT converter does, we can consider the following Algol 60 procedure, which is coded using old (classic) representation:

```
’PROCEDURE’EULER(FCT,SUM,EPS,TIM).,’VALUE’EPS,TIM.,
’INTEGER’ TIM., ’REAL’ ’PROCEDURE’ FCT., ’REAL’ SUM, EPS.,
’COMMENT’ EULER COMPUTES THE SUM OF FCT (I) FOR I
 FROM ZERO UP TO INFINITY BY MEANS OF A SUITABLY
 REFINED EULER TRANSFORMATION. THE SUMMATION IS
 STOPPED AS SOON AS TIM TIMES IN SUCCESSION THE ABSOLUTE
 VALUE OF THE TERMS OF THE TRANSFORMED SERIES IS
 FOUND TO BE LESS THAN EPS, HENCE ONE SHOULD PROVIDE
 A FUNCTION FCT WITH ONE INTEGER ARGUMENT, AN UPPER
 BOUND EPS, AND AN INTEGER TIM. THE OUTPUT IS THE SUM SUM.
 EULER IS PARTICULARLY EFFICIENT IN THE CASE OF A SLOWLY
 CONVERGENT OR DIVERGENT ALTERNATING SERIES.,
```

```
'BEGIN''INTEGER' I,K,N,T.,'ARRAY' M(/0..15/).,
'REAL' MN, MP, DS.,
 I.=N.=T.=0.,M(/0/).=FCT(0).,SUM.=M(/0/)/2.,
 NEXTTERM..I.=I+1.,MN.=FCT(1).,
   'FOR' K.=0'STEP'1'UNTIL'N'DO'
        'BEGIN' MP.=(MN+M(/K/))/2.,M(/K/).=MN.,
           MN.=MP'END'MEANS.,
   'IF' (ABS(MN)'LESS' ABS (M(/N/))'AND'N'LESS'15)'THEN'
        'BEGIN'DS.=MN/2.,N.=N+1.,
           M(/N/).=MN'END' ACCEPT
        'ELSE' DS.=MN.,
         SUM.=SUM+DS.,
        'IF' ABS(DS)'LESS'EPS'THEN'T.=T+1'ELSE'T.=0.,
        'IF'T'LESS'TIM'THEN''GOTO'NEXTTERM
        'END'EULER;
```

This code can be converted to the GNU MARST input language using the following command:

```
macvt -i -s euler.txt -o euler.alg
```

The verbatim result of conversion is the following:

```
procedure euler(fct,sum,eps,tim);value eps,tim;
integer tim; real procedure fct; real sum, eps;
comment EULER COMPUTES THE SUM OF FCT (I) FOR I
 FROM ZERO UP TO INFINITY BY MEANS OF A SUITABLY
 REFINED EULER TRANSFORMATION .THE SUMMATION IS
 STOPPED AS SOON AS TIM TIMES IN SUCCESSION THE ABSOLUTE
 VALUE OF THE TERMS OF THE TRANSFORMED SERIES IS
 FOUND TO BE LESS THAN EPS, HENCE ONE SHOULD PROVIDE
 A FUNCTION FCT WITH ONE INTEGER ARGUMENT, AN UPPER
 BOUND EPS, AND AN INTEGER TIM .THE OUTPUT IS THE SUM SUM
 .EULER IS PARTICULARLY EFFICIENT IN THE CASE OF A SLOWLY
 CONVERGENT OR DIVERGENT ALTERNATING SERIES;
 begin integer i,k,n,t;array m[0:15];
 real mn, mp, ds;
  i:=n:=t:=0;m[0]:=fct(0);sum:=m[0]/2;
  nextterm:i:=i+1;mn:=fct(1);
    for k:=0 step 1 until n do
         begin mp:=(mn+m[k])/2;m[k]:=mn;
            mn:=mp end means;
    if (abs(mn)< abs (m[n])&n<15)then
         begin ds:=mn/2;n:=n+1;
            m[n]:=mn end accept
         else ds:=mn;
          sum:=sum+ds;
         if abs(ds)<eps then t:=t+1 else t:=0;
         if t<tim then go to nextterm
         end euler;
```