

GNU MARST

GNU Algol-to-C Translator
User's Guide for Version 2.7
March 2013

Andrew Makhorin

Copyright © 2000, 2001, 2002, 2007, 2013 Free Software Foundation, Inc.

GNU MARST is part of the GNU project released under the aegis of GNU.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions.

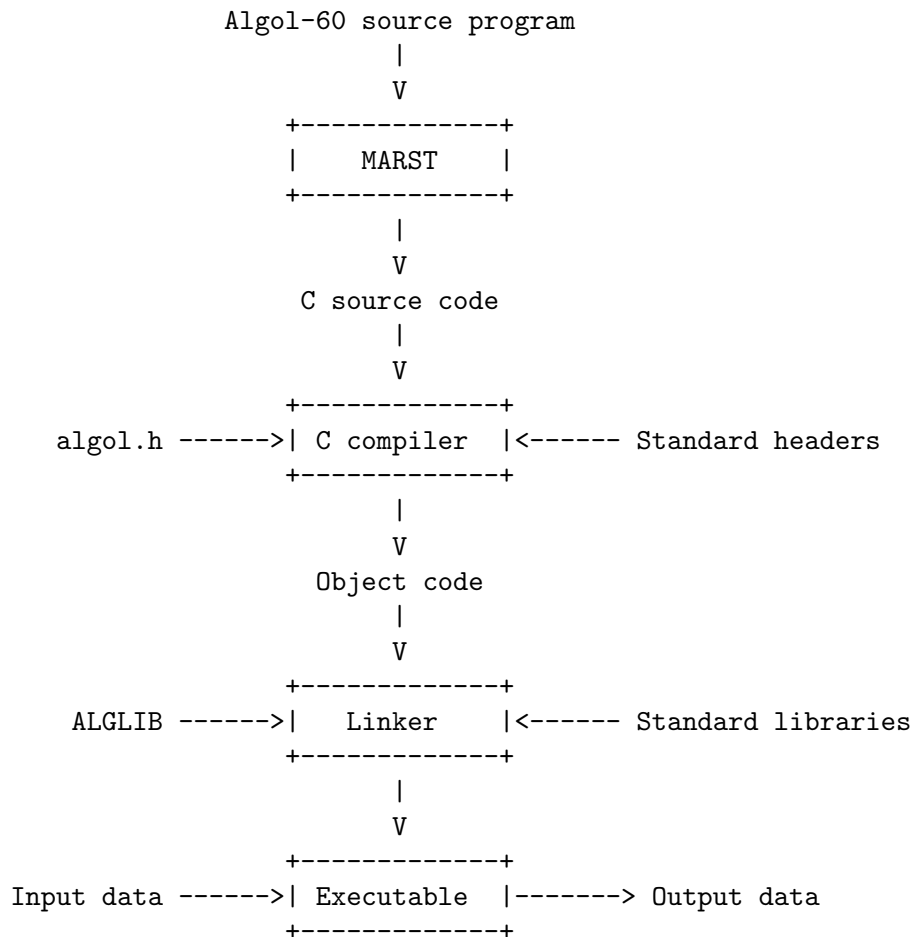
Table of Contents

1	Introduction	1
2	Installation	3
3	Program Invocation	4
4	Usage Example	6
5	Input Language	7
6	Input/Output	11
7	Language Extensions	12
7.1	Modular programming	12
7.2	Pseudo procedure <i>inline</i>	12
7.3	Pseudo procedure <i>print</i>	13
8	Converter Utility	14
	Acknowledgments	19

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 89 programming language.

The processing scheme is 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 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 part of GNU MARST;

Standard headers

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

C compiler

the 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 with the MARST translator. Source code of all library routines is part of GNU MARST. In this distribution the library has the name `libalgol.a`;

Standard libraries

the standard C run-time libraries;

Linker

the linker, a program that resolves external references and produces an executable module;

Executable

a binary file that contains ready-to-run Algol 60 program in an 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

To build and install GNU MARST under GNU/Linux you need to use the standard installation procedure. For details please see file `INSTALL` included in the distribution.

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

<code>marst</code>	as a rule, into <code>usr/local/bin</code> ;
<code>macvt</code>	as a rule, into <code>usr/local/bin</code> ;
<code>algol.h</code>	as a rule, into <code>usr/local/include</code> and/or <code>usr/include</code> ;
<code>libalgol.a</code>	as a rule, into <code>usr/local/lib</code> .

3 Program Invocation

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 specified, the translator emits elementary syntactic units of the 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, if you forgot a comma between the **end** bracket and the next statement.

-e *nnn*, --error-max *nnn*

maximal error allowance

This option specifies maximal error allowance. The translator stops processing after the specified number of errors 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 meaning that the translation continues 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 specifies the 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 larger 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 an output text file to write the produced C code

If this option is not specified, 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 which reflect potential errors and non-standard features used in the source Algol program.

To translate a program written in Algol 60 you need to prepare the program in a plain text file and specify the name of that file 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 can be only a 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 you may use the MARST translator in most practical cases.

At first, you prepare a source Algol 60 program, say, in a text file named `hello.alg`:

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

Then you translate this program to the C programming language:

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

and get the text file named `hello.c`, which you need to compile and link in an usual way (remember about specifying Algol and math libraries for the linker):

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

And finally, you run executable

```
./hello
```

and see what you have. That's all.

5 Input Language

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

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* 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, \dots, z	a, b, ..., z]]
A, B, \dots, Z	A, B, ..., Z	'	"
$0, 1, \dots, 9$	0, 1, ..., 9	,	"
+	+	array	array
—	—	begin	begin
×	*	Boolean	Boolean (boolean)
/	/	code	code
÷	%	comment	comment
↑	^ (**)	do	do
<	<	else	else
≤	<=	end	end
=	=	false	false
≥	>=	for	for
≠	!=	go to	go to (goto)
≡	==	if	if
⊃	->	integer	integer
∨		label	label
∧	&	own	own
¬	!	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 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 when 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. Wichmann. 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* part of GNU MARST.)

To illustrate the input language of the MARST translator, consider the following procedure 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])  $\wedge$  n < 15 then
          begin
            ds := mn/2; n := n + 1;
            mn[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;

```

This procedure may be coded using GNU MARST representation as follows:

```

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 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;
  end
euler := sum
end euler;

```

6 Input/Output

All input/output is performed by the standard Algol 60 procedures. GNU MARST implementation provides up to 16 input/output channels, which have numbers 0, 1, ..., 15. The channel 0 is always connected to `stdin`, so only input from this channel is allowed. Similarly, the channel 1 is always connected to `stdout`, so only output to this channel is allowed. Other channels can be used for both input and output. (The standard procedure *fault* uses the channel 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 0 and 1) are disconnected, i.e. no files are assigned to them.

If an input (output) is required by the Algol program from (to) the channel n , the following actions occur:

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

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

7 Language Extensions

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

7.1 Modular programming

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

First file	Second file
<pre> procedure one(a, b); value a, b; real a, b; begin end; procedure two(x, y); value x, y; array x, y; begin end; </pre>	<pre> 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; </pre>

The procedures *one* and *two* in the first file are called *precompiled procedures*. Declarations of precompiled procedures should be outside of the main program block or compound statement. The procedures *one* and *two* in the second file are called *code procedures*; they have the keyword **code** rather than a procedure body statement. Declarations of code procedures also should be outside of the 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 the main program (the second file in the example above), replacing declarations of corresponding code procedures.

Each code procedure should have the same procedure heading as the corresponding precompiled procedure (however, formal parameter names may differ). Note that mismatched procedure headings cannot 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
<pre> . . . a := 1; b := 2; inline("printf(\"OK\");"); c := 3; . . . </pre>	<pre> . . . dsa_0->a_5 = 1; dsa_0->b_8 = 2; printf("OK"); dsa_0->c_4 = 3; . . . </pre>

The procedure statement *inline* may be used anywhere in the program as an ordinary Algol statement.

7.3 Pseudo procedure *print*

The pseudo procedure *print* is intended mainly for test printing (because the standard Algol input/output is out of any criticism). This procedure has an 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*) in a printable format.

8 Converter Utility

The 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 the resulting code on the output text file. It is assumed that the output code produced by MACVT will be later translated by MARST in an usual way. Note that MACVT performs no syntax checking.

The input language understood by MACVT differs from the 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 a non-official standard, because it was widely used at the time, when Algol 60 was an actual programming language.

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 the input Algol 60 program is coded using a classic representation: all white-space characters are non-significant (except within quoted character strings) and all keywords are enclosed within apostrophes. For details see below.

-f, --free-coding

use free representation

This option allows not to enclose keywords within apostrophes. However, 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 specified, 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***
 name of an output text file, on which the converter writes the converted Algol 60 program
 If this option is not specified, the converter uses the standard output by default.
- s, --old-sc**
 use old (classic) semicolon representation
 This option allows the converter recognizing the diphthong ., (point and comma) as the semicolon (including its usage for terminating comment sequences).
- t, --old-ten**
 use old (classic) ten symbol representation
 This option allows the converter recognizing a single apostrophe, when it is followed by +, -, or digit, as the ten symbol.
- v, --version**
 display the converter version and `exit(0)`

To convert an Algol 60 program you need to prepare it in a plain text file and specify the name of that file 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. Besides, the following additional conventions 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 (e.g. `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, \dots, z	a, b, ..., z	$]$	<code>]</code> <code>/)</code>
A, B, \dots, Z	A, B, ..., Z	<code>'</code>	<code>"</code> <code>'</code>
$0, 1, \dots, 9$	0, 1, ..., 9	<code>,</code>	<code>,</code> <code>,</code>
$+$	<code>+</code>	<code>array</code>	<code>'array'</code>
$-$	<code>-</code>	<code>begin</code>	<code>'begin'</code>
\times	<code>*</code>	<code>Boolean</code>	<code>'boolean'</code>
$/$	<code>/</code>	<code>code</code>	<code>'code'</code>
\div	<code>%</code> <code>'/'</code> <code>'div'</code>	<code>comment</code>	<code>'comment'</code>
\uparrow	<code>^</code> <code>**</code> <code>'power'</code> <code>'pow'</code>	<code>do</code>	<code>'do'</code>
$<$	<code><</code> <code>'less'</code>	<code>else</code>	<code>'else'</code>
\leq	<code><=</code> <code>'notgreater'</code>	<code>end</code>	<code>'end'</code>
$=$	<code>=</code> <code>'equal'</code>	<code>false</code>	<code>'false'</code>
\geq	<code>>=</code> <code>'notless'</code>	<code>for</code>	<code>'for'</code>
\neq	<code>!=</code> <code>'notequal'</code>	<code>go to</code>	<code>'goto'</code>
\equiv	<code>==</code> <code>'equiv'</code>	<code>if</code>	<code>'if'</code>
\supset	<code>-></code> <code>'impl'</code>	<code>integer</code>	<code>'integer'</code>
\vee	<code> </code> <code>'or'</code>	<code>label</code>	<code>'label'</code>
\wedge	<code>&</code> <code>'and'</code>	<code>own</code>	<code>'own'</code>
\neg	<code>!</code> <code>'not'</code>	<code>procedure</code>	<code>'procedure'</code>
<code>,</code>	<code>,</code>	<code>real</code>	<code>'real'</code>
<code>.</code>	<code>.</code>	<code>step</code>	<code>'step'</code>
10	<code>#</code> <code>'</code> <code>'10'</code>	<code>string</code>	<code>'string'</code>
<code>:</code>	<code>:</code> <code>..</code>	<code>switch</code>	<code>'switch'</code>
<code>;</code>	<code>;</code> <code>.,</code>	<code>then</code>	<code>'then'</code>
<code>:=</code>	<code>:=</code> <code>.=</code> <code>..=</code>	<code>true</code>	<code>'true'</code>
<code>(</code>	<code>(</code>	<code>until</code>	<code>'until'</code>
<code>)</code>	<code>)</code>	<code>value</code>	<code>'value'</code>
<code>[</code>	<code>[</code> <code>(/</code>	<code>while</code>	<code>'while'</code>

To illustrate what the MACVT converter does, consider the following Algol 60 procedure, which is coded using an 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 with 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;
```

Acknowledgments

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

The author especially thanks Brian Wichmann <brian.wichmann@totalise.co.uk> for providing a set of Algol 60 validation tests.