

# Zpravodaj Československého sdružení uživatelů TeXu

---

Hans Hagen  
Lua in MetaPost

*Zpravodaj Československého sdružení uživatelů TeXu*, Vol. 27 (2017), No. 3-4, 138–154

Persistent URL: <http://dml.cz/dmlcz/150280>

## Terms of use:

© Československé sdružení uživatelů TeXu, 2017

Institute of Mathematics of the Czech Academy of Sciences provides access to digitized documents strictly for personal use. Each copy of any part of this document must contain these *Terms of use*.



This document has been digitized, optimized for electronic delivery and stamped with digital signature within the project *DML-CZ*:  
*The Czech Digital Mathematics Library* <http://dml.cz>

In L<sup>A</sup>T<sub>E</sub>X, it is now possible to run snippets of L<sup>A</sup>U<sup>A</sup> code from within METAPOST. The article describes the mechanism, the low-level interface available in L<sup>A</sup>T<sub>E</sub>X, as well as the high-level interface available in C<sup>O</sup>N<sup>T</sup>E<sup>X</sup>T through example.

**Keywords:** L<sup>A</sup>U<sup>A</sup>, L<sup>A</sup>T<sub>E</sub>X, C<sup>O</sup>N<sup>T</sup>E<sup>X</sup>T, METAPOST, M<sup>P</sup>L<sup>I</sup>B

## Introduction

Already for a some years I have been wondering how it would be if we could escape to L<sup>A</sup>U<sup>A</sup> inside METAPOST, or in practice, in M<sup>P</sup>L<sup>I</sup>B in L<sup>A</sup>T<sub>E</sub>X. The idea is simple: embed L<sup>A</sup>U<sup>A</sup> code in a METAPOST file that gets run as soon as it's seen. In case you wonder why L<sup>A</sup>U<sup>A</sup> code makes sense, imagine generating graphics using external data. The capabilities of L<sup>A</sup>U<sup>A</sup> to deal with that is more flexible and advanced than in METAPOST. Of course we could generate a METAPOST definition of a graphic from data but often it makes more sense to do the reverse. I finally found time and reason to look into this and in the following sections I will describe how it's done.

## The basics

The approach is comparable to L<sup>A</sup>T<sub>E</sub>X's `\directlua`. That primitive can be used to execute L<sup>A</sup>U<sup>A</sup> code and in combination with `tex.print` we can pipe back strings into the T<sub>E</sub>X input stream. There is a complication that we have to be able to operate under different so called catcode regimes: the meaning of characters can differ per regime. We also have to deal with line endings in special ways as they relate to paragraphs and such. In METAPOST we don't have that complication so getting back input into the METAPOST input, we can do so with simple strings. For that a mechanism similar to `scantokens` can be used. That way we can return anything (including nothing) as long as METAPOST can interpret it and as long as it fulfils the expectations.

```
numeric n ;  
n := scantokens("123.456") ;  
A script is run as follows:  
numeric n ;  
n := runscript("return '123.456'") ;
```

This primitive doesn't have the word `lua` in its name so in principle any wrapper around the library can use it as hook. In the case of `LUATEX` the script language is of course `LUA`. At the `METAPOST` end we only expect a string. How that string is constructed is completely up to the `LUA` script. In fact, the user is completely free to implement the runner any way she or he wants, like:

```
local function scriptrunner(code)
  local f = loadstring(code)
  if f then
    return tostring(f())
  else
    return ""
  end
end
```

This is hooked into an instance as follows:

```
local m = mplib.new {
  ...
  run_script = scriptrunner,
  ...
}
```

Now, beware, this is not the `ConTEXt` way. We provide print functions and other helpers, which we will explain in the next section.

## Helpers

After I got this feature up and running I played a bit with possible interfaces at the `ConTEXt` (read: `METAFUN`) end and ended up with a bit more advanced runner where no return value is used. The runner is wrapped in the `lua` macro.

```
numeric n ;
n := lua("mp.print(12.34567)") ;
draw texttext(n) xsized 4cm withcolor maincolor ;
```

This renders as:

12.34567

In case you wonder how efficient calling `LUA` is, don't worry: it's fast enough, especially if you consider suboptimal `LUA` code and the fact that we switch between machineries.

```
draw image (
  lua("statistics.starttiming()") ;
  for i=1 upto 5000 :
```

```

draw lua ("mp.pair
  (math.random(-74,270),
    math.random(-22,22))" ) ;
endfor ;
lua("statistics.stoptiming()") ;
draw texttext(lua
  ("mp.print(
    statistics.elapsedtime()")
  ) ysize 40 ;
) withcolor maincolor
  withpen pencircle scaled 1 ;

```

Here the part:

```

draw lua ("mp.pair
  (math.random(-74,270),
    math.random(-22,22))" ) ;

```

effectively becomes (for instance):

```
draw scantokens "(25,15)" ;
```

which in turn becomes:

```
draw scantokens (25,15) ;
```

The same happens with this:

```

draw texttext (lua
  ("mp.print
    (statistics.elapsedtime()")
  ) ) ...

```

This becomes for instance:

```
draw texttext(scantokens "1.23") ...
```

and therefore:

```
draw texttext(1.23) ...
```

We can use `mp.print` here because the `texttext` macro can deal with numbers.

The next also works:

```

draw texttext(lua
  ("mp.quoted
    (statistics.elapsedtime()")
  )
) ...

```

Now we get (in METAPost speak):

```

draw texttext(scantokens
  (ditto & "1.23" & ditto) ...

```

Here `ditto` represents the double quotes that mark a string. Of course, because we pass the strings directly to `scantokens`, there are no outer quotes at all, but this is how it can be simulated. In the end we have:

```
draw texttext("1.23") ...
```

What you use, `mp.print` or `mp.quoted` depends on what the expected code is: an assignment to a numeric can best be a number or an expression resulting in a number. This graphic becomes:



The runtime on my current machine is some 0.25 seconds without and 0.12 seconds with caching. But to be honest, speed is not really a concern here as the amount of complex METAPOST graphics can be neglected compared to extensive node list manipulation. With `LUAJITTEX` generating the graphic takes 15% less time.<sup>1</sup>

The three print command accumulate their arguments:

```
numeric n ;  
n := lua("mp.print(1) mp.print('+') mp.print(2)") ;  
draw texttext(n) xsize 1cm  
withcolor maincolor ;
```

As expected we get:



Equally valid is:

```
numeric n ;  
n := lua("mp.print(1,'+',2)") ;  
draw texttext(n) xsize 1cm  
withcolor maincolor ;
```

This gives the same result:



---

<sup>1</sup>Processing a small 8 page document like this takes about one second, which includes loading a bunch of fonts.

Of course all kind of action can happen between the prints. It is also legal to have nothing returned as could be seen in the 10.000 dot example: there the timer related code returns nothing so effectively we have `scantokens("")`. Another helper is `mp.quoted`, as in:

```
draw texttext
  (lua
    ("mp.quoted
      ('@0.3f',
        " & decimal n & "
      )"
    )
  )
) withcolor maincolor ;
```

This typesets 3.000. Watch the @. When no percent character is found in the format specifier, we assume that an @ is used instead.

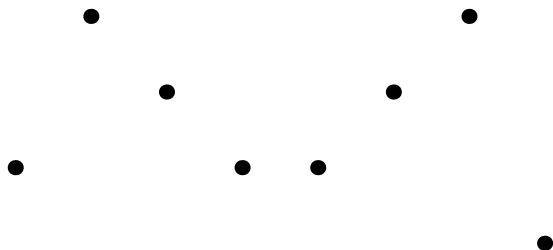
But, the real benefit of embedded LUA is when we deal with data that is stored at the LUA end. First we define a small dataset:

```
\startluacode
table.save("demo-data.lua",
{
  { 1, 2 }, { 2, 4 }, { 3, 3 },
  { 4, 2 }, { 5, 2 }, { 6, 3 },
  { 7, 4 }, { 8, 1 },
}
)
\stopluacode
```

There are several ways to deal with this table. I will show clumsy as well as better looking ways.

```
lua("MP = { }
MP.data = table.load('demo-data.lua')"
) ;
numeric n ;
lua("mp.print('n := ',\#MP.data)") ;
for i=1 upto n :
  drawdot
    lua("mp.pair
      (MP.data[" & decimal i & "]"
    ) scaled cm
    withpen pencircle scaled 2mm
    withcolor maincolor ;
endfor ;
```

Here we load a LUA table and assign the size to a METAPOST numeric. Next we loop over the table entries and draw the coordinates.



We will stepwise improve this code. In the previous examples we omitted wrapper code but here we show it:

```
\startluacode
  MP.data = table.load('demo-data.lua')
  function MP.n()
    mp.print(#MP.data)
  end
  function MP.dot(i)
    mp.pair(MP.data[i])
  end
\stopluacode
```

```
\startMPcode
  numeric n ;
  n := lua("MP.n()") ;
  for i=1 upto n :
    drawdot
      lua("MP.dot
        (" & decimal i & ")")
      ) scaled cm
    withpen pencircle scaled 2mm
    withcolor maincolor ;
  endfor ;
\stopMPcode
```

So, we create a few helpers in the MP table. This table is predefined so normally you don't need to define it. You may however decide to wipe it clean.

• •

• •

• • •

•

You can decide to hide the data:

```
\startluacode
  local data = { }
  function MP.load(name)
    data = table.load(name)
  end
  function MP.n()
    mp.print(#data)
  end
  function MP.dot(i)
    mp.pair(data[i])
  end
\stopluacode
```

It is possible to use less LUA, for instance in:

```
\startluacode
  local data = { }
  function MP.loaded(name)
    data = table.load(name)
    mp.print(#data)
  end
  function MP.dot(i)
    mp.pair(data[i])
  end
\stopluacode
```

```
\startMPcode
  for i=1 upto
    lua
      ("MP.loaded
        ('demo-data.lua')")
    ) :
  drawdot
    lua("MP.dot(",i,")") scaled cm
```

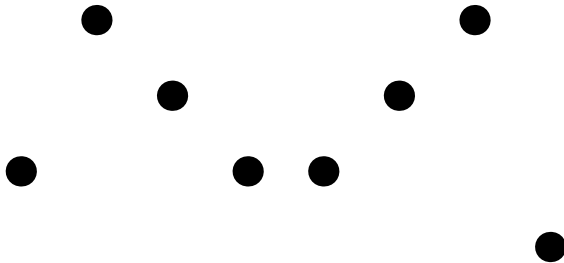


```

        withpen pencircle scaled 4mm
        withcolor maincolor ;
    endfor ;
\stopMPcode

```

Here we also omit the `decimal` because the `lua` macro is clever enough to recognize it as a number.



By using some METAPOST magic we can even go a step further in readability:

```

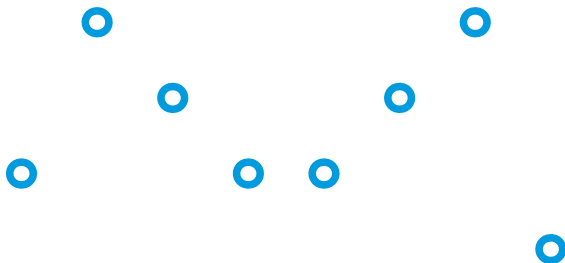
\startMPcode{doublefun}
  cmykcolor maincolor;
  maincolor := (1,.15,0,0);
  lua.MP.load("demo-data.lua") ;

  for i=1 upto lua.MP.n() :
    drawdot lua.MP.dot(i) scaled cm
    withpen pencircle scaled 4mm
    withcolor maincolor ;
  endfor ;

  for i=1 upto MP.n() :
    drawdot MP.dot(i) scaled cm
    withpen pencircle scaled 2mm
    withcolor white ;
  endfor ;
\stopMPcode

```

Here we demonstrate that it also works ok in `double` mode, which makes much sense when processing data from other sources. Watch how we omit the `.lua` prefix: the `MP` macro will deal with that.



So in the end we can simplify the code that we started with to:

```
\startMPcode{doublefun}
  for i=1 upto
    MP.loaded("demo-data.lua") :
    drawdot
      MP.dot(i) scaled cm
      withpen pencircle scaled 2mm
      withcolor maincolor ;
  endfor ;
\stopMPcode
```

## Access to variables

The question with such mechanisms is always: how far should we go. Although METAPost is a macro language it has properties of procedural languages. It also has more introspective features at the user end. For instance, one can loop over the resulting picture and manipulate it. This means that we don't need full access to METAPost internals. However, it makes sense to provide access to basic variables: **numeric**, **string**, and **boolean**.

```
draw texttext(lua
  ("mp.quoted
    ('@0.15f',
      mp.get.numeric('pi')-math.pi
    )"
  )
)
  ysize .5cm
  withcolor maincolor ;
```

In double mode you will get zero printed but in scaled mode we definitely get a difference:

**-0.000006349878856**

In the next example we use `mp.quoted` to make sure that indeed we pass a string. The `texttext` macro can deal with numbers but an unquoted `yes` or `no` is asking for problems.

```
boolean b ;
b := true ;
draw texttext(
  lua
    ("mp.quoted(mp.get.boolean('b')
      and 'yes' or 'no')")
)
ysized 1cm
withcolor maincolor ;
```

Especially when more text is involved it makes sense to predefine a helpers in the MP namespace if only because METAPost (currently) doesn't like newlines in the middle of a string, so a `lua` call has to be on one line.

# yes

Here is an example where `LUA` does something that would be close to impossible, especially if more complex text is involved.

```
string s ;
s := "TEX" ; % ""
draw texttext
  (lua
    ("mp.quoted
      (characters.lower
        (mp.get.string('s'))
      )
    )"
  )
)
ysized 1cm
withcolor maincolor ;
```

As you can see here, the whole repertoire of helper functions can be used in a `METAFUN` definition.

## The library

In ConT<sub>E</sub>Xt we have a dedicated runner, but for the record we mention the low level constructor:

```
local m = mplib.new {  
  ...  
  script_runner = function(s) return  
    loadstring(s)() end,  
  script_error = function(s)  
    print(s) end,  
  ...,  
}
```

An instance (in this case `m`) has a few extra methods. Instead you can use the helpers in the library.

---

<code>m:get_numeric(name)</code>	returns a numeric (double)
<code>m:get_boolean(name)</code>	returns a boolean ( <code>true</code> or <code>false</code> )
<code>m:get_string (name)</code>	returns a string

---

<code>mplib.get_numeric(m,name)</code>	returns a numeric (double)
<code>mplib.get_boolean(m,name)</code>	returns a boolean ( <code>true</code> or <code>false</code> )
<code>mplib.get_string (m,name)</code>	returns a string

---

In ConT<sub>E</sub>Xt the instances are hidden and wrapped in high level macros, so there you cannot use these commands.

## ConT<sub>E</sub>Xt helpers

The `mp` namespace provides the following helpers:

---

<code>print(...)</code>	returns one or more values
<code>pair(x,y)</code> and <code>pair(t)</code>	returns a proper pair
<code>triplet(x,y,z)</code> and <code>triplet(t)</code>	returns an RGB color
<code>quadruple(w,x,y,z)</code> and <code>quadruple(t)</code>	returns an CMYK color
<code>format(fmt,...)</code>	returns a formatted string
<code>quoted(fmt,...)</code> and <code>quoted(s)</code>	returns a (formatted) quoted string
<code>path(t[,connect][,close])</code>	returns a connected (closed) path

---

The `mp.get` namespace provides the following helpers:

---

<code>numeric(name)</code>	gets a numeric from METAPOST
<code>boolean(name)</code>	gets a boolean from METAPOST
<code>string(name)</code>	gets a string from METAPOST

---

## Paths

In the meantime we got several questions on the ConT<sub>E</sub>Xt mailing list about turning coordinates into paths. Now imagine that we have this dataset:

```
10 20 20 20 -- sample 1
30 40 40 60
50 10
```

```
10 10 20 30 % sample 2
30 50 40 50
50 20
```

```
10 20 20 10 # sample 3
30 40 40 20
50 10
```

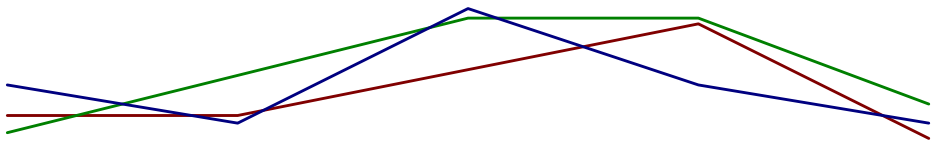
In this case I've put the data in a buffer so that it can be shown here as well as used in a demo. Watch how we can add comments. The following code converts this into a table with three subtables.

```
\startluacode
  MP.myset =
    mp.dataset
      (buffers.getcontent("dataset"))
\stopluacode
```

We use the MP (user) namespace to store the table. Next we turn these subtables into paths:

```
\startMPcode
  for i=1 upto
    lua("mp.print(mp.n(MP.myset))") :
  draw
    lua("mp.path
      (MP.myset[" & decimal i & "]
      )"
    )
    xysized (HSize,10ExHeight)
    withpen
      pencircle scaled .25ExHeight
    withcolor basiccolors[i]/2 ;
  endfor ;
\stopMPcode
```

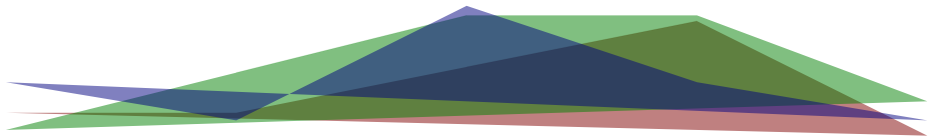
This gives:



Instead we can fill the path in which case we also need to close it. The `true` argument deals with that:

```
\startMPcode
  for i=1 upto
    lua("mp.print(mp.n(MP.myset))") :
  path p ; p :=
    lua("mp.path
      (MP.myset
        [\" & decimal i & \"],
        true
      )"
    )
    xysized (HSize,10ExHeight) ;
  fill p
    withcolor basiccolors[i]/2
    withtransparency (1,.5) ;
  endfor ;
\stopMPcode
```

We get:



The following makes more sense:

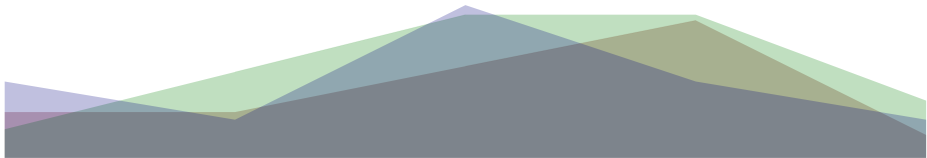
```
\startMPcode
  for i=1 upto
    lua("mp.print
      (mp.n(MP.myset))"
    ) :
  path p ;
  p := lua("mp.path
    (MP.myset[\" & decimal i & \"])")
  )
  xysized (HSize,10ExHeight) ;
  p :=
```

```

(xpart llcorner boundingbox p,0)
-- p --
(xpart lrcorner boundingbox p,0)
-- cycle ;
fill p
  withcolor basiccolors[i]/2
  withtransparency (1,.25) ;
endfor ;
\stopMPcode

```

So this gives:



This (area) fill is so common that we have a helper for it:

```

\startMPcode
for i=1 upto
  lua("mp.size(MP.myset)") :
fill area
  lua("mp.path
    (MP.myset[" & decimal i & "])"
  )
  xysized (HSize,5ExHeight)
  withcolor basiccolors[i]/2
  withtransparency (2,.25) ;
endfor ;
\stopMPcode

```

So this gives:



This snippet of METAPOST code still looks kind of horrible so how can we make it look better? Here is an attempt, First we define a bit more LUA:

```

\startluacode
local data =
  mp.dataset
    (buffers.getcontent("dataset"))

```

```

MP.dataset = {
  Line = function(n) mp.path(data[n]) end,
  Size = function() mp.size(data) end,
}
\stopluacode

```

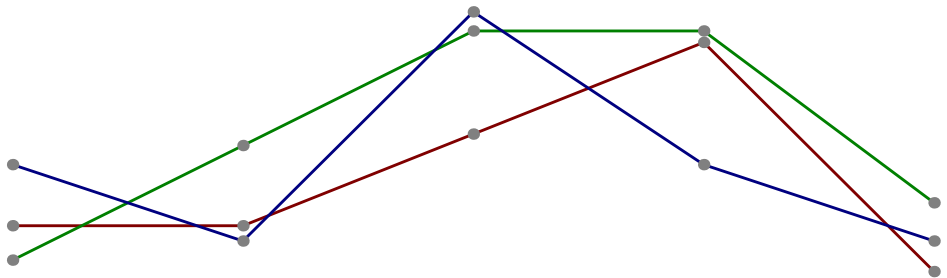
We can now make the METAPOST look more natural. Of course this is possible because in METAFUN the lua macro does some extra work.

```

\startMPcode
  for i=1 upto
    lua.MP.dataset.Size() :
  path p ;
  p := lua.MP.dataset.Line(i)
  xysized (HSize,20ExHeight) ;
  draw
    p
    withpen pencircle scaled .25ExHeight
    withcolor basiccolors[i]/2 ;
  drawpoints
    p
    withpen pencircle scaled ExHeight
    withcolor .5white ;
  endfor ;
\stopMPcode

```

As expected, we get the desired result:



Once we start making things look nicer and more convenient, we quickly end up with helpers like the once in the next example. First we save some demo data in files:

```

\startluacode
  io.savedata("foo.tmp","10 20 20 20 30 40 40 60 50 10")
  io.savedata("bar.tmp","10 10 20 30 30 50 40 50 50 20")
\stopluacode

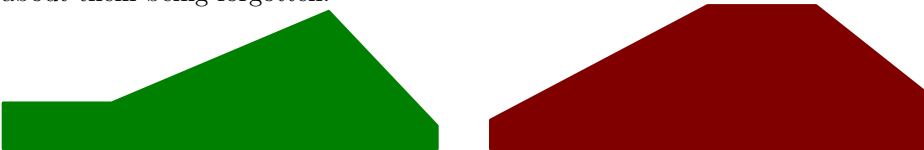
```



We load the data in datasets:

```
\startMPcode
lua.mp.datasets.load("foo","foo.tmp");
lua.mp.datasets.load("bar","bar.tmp");
fill area
lua.mp.datasets.foo.Line()
xysized (HSize/2-EmWidth,10ExHeight)
withpen
  pencircle scaled .25ExHeight
withcolor green/2 ;
fill area
lua.mp.datasets.bar.Line()
xysized (HSize/2-EmWidth,10ExHeight)
shifted (HSize/2+EmWidth,0)
withpen
  pencircle scaled .25ExHeight
withcolor red/2 ;
\stopMPcode
```

Because the datasets are stores by name we can use them without worrying about them being forgotten:



If no tag is given, the filename (without suffix) is used as tag, so the following is valid:

```
\startMPcode
lua.mp.datasets.load("foo.tmp") ;
lua.mp.datasets.load("bar.tmp") ;
\stopMPcode
```

The following methods are defined for a dataset:

method	usage
Size	the number of subsets in a dataset
Line	the joined pairs in a dataset making a non-closed path
Data	the table containing the data (in subsets, so there is always at least one subset)

*Due to limitation is METAPOST suffix handling the methods start with an up-percase character.*

## Remark

The features described here are at this moment still experimental but the interface will not change. There might be a few more accessors and for sure there will be more LUA helpers provided. As usual I need some time to play with it before I make up my mind. It is also possible to optimize the METAPOST-LUA script call a bit but I might do that later.

When we played with this interface we ran into problems with loop variables and macro arguments. These are internally kind of anonymous. Take this:

```
for i=1 upto 100 : draw(i,i) endfor ;
```

The `i` is not really a variable with name `i` but becomes an object (capsule) when the condition is scanned, and a reference to that object when the body is scanned. The body of the for loop gets expanded for each step but at that time there is no longer a variable `i`. The same is true for variables in:

```
def foo(expr x, y, delta) =  
  draw (x+delta,y+delta)  
enddef ;
```

We are still trying to get this right with the LUA interface. Interesting is that when we were exploring this, we ran into quite some cases where we could make METAPOST abort due some memory or stack overflow. Some are just bugs in the new code (due to the new number model) while others come with the design of the system: border cases that never seem to happen in interactive use while the library use assumes no interaction in case of errors.

In ConT<sub>E</sub>Xt there are more features and helpers than shown here but these are discussed in the METAFUN manual.

## Lua v MetaPostu

V L<sup>A</sup>T<sub>E</sub>Xu lze nyní spouštět skripty v jazyce LUA z interpretru jazyka METAPOST. Článek na příkladech popisuje samotný mechanismus, nízkoúrovňové rozhraní na úrovni L<sup>A</sup>T<sub>E</sub>Xu i vysokoúrovňové rozhraní pro formát ConT<sub>E</sub>Xt.

**Klíčová slova:** LUA, L<sup>A</sup>T<sub>E</sub>X, ConT<sub>E</sub>Xt, METAPOST, MPLIB

*Hans Hagen, pragma@wxs.nl*