

# Dednat6: an extensible (**semi-**)preprocessor for Lua $\text{\LaTeX}$ that understands diagrams in **ASCII art**

Eduardo Ochs - UFF

TUG 2018 - Rio de Janeiro, 20-22 jul 2018

<http://angg.twu.net/dednat6.html>

### Prehistory: `dednat.icn`

My master's thesis was partly about Natural Deduction, and it had lots of tree diagrams like these:

$$\frac{\frac{[x]^1 \quad f}{f(x)} \quad g}{g(f(x))} \quad 1 \qquad \frac{[a]^1 \quad a \rightarrow b}{b} \quad \frac{b \rightarrow c}{c} \quad 1$$

I used `proof.sty` to typeset them, but the code for each diagram was so opaque that I had to keep a 2D ascii art version of each diagram in comments so that I wouldn't get lost...

## Prehistory: dednat.icn (2)

...like this:

$$\frac{\frac{[x]^1 f}{f(x)} g}{g(f(x))} \lambda x.g(f(x))$$

```

%: [x]^1 f
%: -----
%: f(x) g
%: -----
%: g(f(x))
%: -----1
%: \lambda.x.g(f(x))
%:
$$\infer[{}]{ \mathstrut \lambda x.g(f(x)) }{
  \infer[{}]{ \mathstrut g(f(x)) }{
    \infer[{}]{ \mathstrut f(x) }{
      \mathstrut [x]^1 &
      \mathstrut f } &
      \mathstrut g } } }
$$

```

### Prehistory: dednat.icn (3)

...then I realized that I could automate the boring part. I made the syntax of the 2D ascii art trees more rigid and wrote a parser (in **Icon!**) that understood it.

A tree with a name tag like  $\hat{\text{foo}}$  below it

would become a `\defded{foo}{...}` —

**dednat.icn** would only look for trees in ‘%:’-lines,

```
%: [x]^1 f
%: -----
%:      f(x)   g
%:      -----
%:      g(f(x))
%: -----1
%:  λx.g(f(x))
%:
%:  ^foo
```



```
\defded{foo}{
  \infer[1]{ \mathstrut λx.g(f(x)) }{
    \infer[{}]{ \mathstrut g(f(x)) }{
      \infer[{}]{ \mathstrut f(x) }{
        \mathstrut [x]^1 &
        \mathstrut f } &
        \mathstrut g } } }
```

and would put the ‘`\defded`’s in another file...

## Prehistory: dednat.icn (4)

So that I could have this, in `myfile.tex`:

```

\input myfile.auto.dnt
%
%:  [x]^1  f
%:  -----
%:      f(x)  g
%:      -----
%:      g(f(x))
%:  -----1
%:  \lambda.g(f(x))
%:
%:  ~foo
%
$$$$\ded{foo}$$$$

```

$$\longrightarrow \frac{[x]^1 f}{\frac{f(x) g}{g(f(x))} 1} \lambda x.g(f(x))$$

Running ‘`dednat.icn myfile.tex`’ would generate the file `myfile.auto.dnt`.

## Prehistory: dednat4.lua

`dednat.icn` understood one kind of “head”:

‘%:’-lines would be scanned for trees.

`dednat4.lua` understood three kinds of heads:

‘%:’-lines would be scanned for trees,

‘%L’-lines contained Lua code,

‘%D’-lines contained diagrams in a Forth-based language.

New heads could be added **dynamically**.

(Actually I also had a head to define abbreviations like ‘->’ → ‘\to ’)

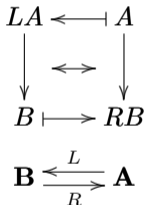


## Dednat4.lua's language for diagrams (2)

```

%D diagram adj
%D 2Dx      100      +25
%D 2D 100 LA <-| A
%D 2D      |         |
%D 2D      | <--> |
%D 2D      v         v
%D 2D +25 B |-> RB
%D 2D
%D 2D +15 \catB \catA
%D 2D
%D (( LA A <-|
%D   LA B -> A RB ->
%D   B RB |->
%D   LA RB harrownodes nil 20 nil <->
%D   \catB \catA <- sl^ .plabel= a L
%D   \catB \catA -> sl_ .plabel= b R
%D ))
%D enddiagram
%D
$$$ \diag{adj} $$$

```





## Dednat4.lua's language for diagrams (3)

(See my “Bootstrapping a Forth in 40 lines of Lua code” in the Lua Gems book... section ‘Modes’)

```
%D diagram adj
%D 2Dx 100 +25
%D 2D 100 LA <-| A
%D 2D | |
%D 2D | <--> |
%D 2D v v
%D 2D +25 B |-> RB
%D 2D
%D 2D +15 \catB \catA
%D 2D
%D (( LA A <-|
%D LA B -> A RB ->
%D B RB |->
%D LA RB harrownodes nil 20 nil <->
%D \catB \catA <- sl^ .plabel= a L
%D \catB \catA -> sl_ .plabel= b R
%D ))
%D enddiagram
%D
$$\diag{adj}$$
```

The words in red

“eat text”.

2D and 2Dx eat

the rest of the line

as a grid, and define  
nodes with coordinates.

.plabel modifies the  
arrow at the top of the  
stack: ‘placement’ ‘label’

## Dednat4.lua's language for diagrams (4)

(See my “Bootstrapping a Forth in 40 lines of Lua code” in the Lua Gems book... section ‘Modes’)

```

%D diagram adj
%D 2Dx 100 +25
%D 2D 100 LA <-| A
%D 2D | |
%D 2D | <--> |
%D 2D v v
%D 2D +25 B |-> RB
%D 2D
%D 2D +15 \catB \catA
%D 2D
%D (( LA A <-|
%D LA B -> A RB ->
%D B RB |->
%D LA RB harrownodes nil 20 nil <->
%D \catB \catA <- sl^ .plabel= a L
%D \catB \catA -> sl_ .plabel= b R
%D ))
%D enddiagram
%D
$$$diag{adj}$$$

```

2D and 2Dx eat the rest of the line as a grid, and define nodes with coordinates. Arrow words connect the two topmost nodes in the stack. harrownodes creates two phantom nodes for a middle horizontal arrow.

## Dednat4.lua's language for diagrams (5)

For the sake of completeness... `diagram` resets several tables, `enddiagram` outputs the table `arrows` as diagxy code, `sl^` and `sl_` slide the topmost arrow in the stack, The `'))` in a `(( ... ))` block drops all top items from the stack until the depth becomes what it was at the `((`, we can put Lua code in `'%L'` lines between `'%D'` lines, and...

```
require "diagforth"

storenode {TeX="a", tag="a", x=100, y=100}
storenode {TeX="b", tag="b", x=140, y=100}
= nodes

storearrow(DxyArrow {from="a", to="b", shape="|->",
                    slide="5pt", label="up",
                    placement="a"})
storearrow(DxyArrow {from="a", to="b", shape=">."})
storearrow(DxyPlace {nodes["a"]})
storearrow(DxyLiteral {"literal foobar"})
= arrows

print(arrow_to_TeX(arrows[1]))
print(arrows[2]:TeX())
print(arrows[3]:TeX())
print(arrows[4]:TeX())
print(arrow_to_TeX(arrows[4]))
```

← this Lua code shows how the low-level functions work...

## Dednat6: a semi-preprocessor

Dednat4 is a real **pre**-processor — it generates a `foo.auto.dnt` from `foo.tex`, and it runs **before** L<sup>A</sup>T<sub>E</sub>X.

In Dednat6 the Lua code that processes the lines with heads like ‘%L’, ‘%:’, ‘%D’, etc, **pretends** to run **at the same time** as T<sub>E</sub>X...

In fact there are synchronization points.

Each tree in a ‘%:’ block generates a ‘\defded’  
 each diagram in a ‘%D’ block generates a ‘\defdiag’...  
 ‘\pu’ means “**process** all pending heads **until** the current line”, and send the defs to L<sup>A</sup>T<sub>E</sub>X—

## Dednat6: a semi-preprocessor (2)

‘\pu’ means “**process** all pending heads **until** the current line”, and send the defs to L<sup>A</sup>T<sub>E</sub>X—

This is implemented using “blocks” with i and j fields for their starting and ending lines.

```
%D diagram triangle
%D 2Dx 100 +20
%D 2D 100 A --> B
%D 2D      \   |
%D 2D      v   v
%D 2D +20      C
%D 2D
%D (( A B -> B C -> A C ->
%D ))
%D enddiagram
$$\pu \diag{triangle}$$

%: A A->B
%: -----
%:      B      C
%:      -----
%:      B/\C
%:      ^a-tree
%:
$$\pu \ded{a-tree}$$
```

‘%D’ block: lines 1–10

First ‘\pu’: line 12

‘%:’ block: lines 15–22

Second ‘\pu’: line 24

Whole .tex file: lines 1–24

## Dednat6: a semi-preprocessor (3)

'\pu' means “**process** all pending heads **until** the current line”, and send the defs to L<sup>A</sup>T<sub>E</sub>X—

This is implemented using “blocks” with i and j fields for their starting and ending lines.

```
%D diagram triangle
%D 2Dx 100 +20
%D 2D 100 A --> B
%D 2D \ |
%D 2D v v
%D 2D +20 C
%D 2D (( A B -> B C -> A C ->
%D ))
%D enddiagram
$$\pu \diag{triangle}$$

%: A A->B
%: -----
%:      B      C
%:      -----
%:      B/\C
%:      ^a-tree
%:
$$\pu \ded{a-tree}$$
```

```
tf = Block {i=1, j=24, nline=1, ...}
```

```
First '\pu': line 12
```

```
processuntil(12)
```

```
processlines(1, 11)
```

```
processblock {head="%D", i=1, j=10}
```

```
output("\defdiag{triangle}{...}")
```

```
nline=13
```

```
tf becomes {i=1, j=24, nline=13, ...}
```

```
Second '\pu': line 24
```

```
processuntil(24)
```

```
processlines(13, 23)
```

```
processblock {head=":%", i=15, j=22}
```

```
output("\defded{a-tree}{...}")
```

```
nline=25
```

## Dednat6: a semi-preprocessor (4)

```
%D diagram triangle
%D 2Dx      100    +20
%D 2D      100 A  --> B
%D 2D              \  |
%D 2D              v  v
%D 2D      +20      C
%D 2D
%D (( A B -> B C -> A C ->
%D ))
%D enddiagram
```

```
$$\pu \diag{triangle}$$
```

```
%.   A   A->B
%.   -----
%.         B       C
%.         -----
%.         B/\C
%.
%.         ^a-tree
%.
%.
%
```

```
$$\pu \ded{a-tree}$$
```

```
tf = Block {i=1, j=24, nline=1, ...}
```

First '`\pu`': line 12

```
processuntil(12)
```

```
processlines(1, 11)
```

```
processblock {head="%D", i=1, j=10}
```

```
output("\defdiag{triangle}{...}")
```

```
nline=13
```

tf becomes {i=1, j=24, nline=13, ...}

Second '`\pu`': line 24

```
processuntil(24)
```

```
processlines(13, 23)
```

```
processblock {head="%.:", i=15, j=22}
```

```
output("\defded{a-tree}{...}")
```

```
nline=25
```

## Downloading and testing

I gave up (temporarily?) keeping a package or a git repo of Dednat6... but if you run something like this in a shell,

```
rm -rfv /tmp/edrx-latex/  
mkdir /tmp/edrx-latex/  
cd /tmp/edrx-latex/  
# See: http://angg.twu.net/LATEX/2017planar-has-1.pdf  
wget http://angg.twu.net/LATEX/2017planar-has-1.tgz  
tar -xvzf 2017planar-has-1.tgz  
lualatex 2017planar-has-1.tex
```

you download and unpack a .tgz with the full source code for `2017planar-has-1.pdf`, including a full version of Dednat6, and all the (non-standard)  $\TeX$  files...

The home page of dednat6

<http://angg.twu.net/dednat6.html>

points to several such .tgzs, both simple and complex.



## Extensions

It is easy to extend Dednat6 with new heads...

For example, for these slides I created a head ‘%V’ for a Dednat6-based verbatim mode...

the Lua code was initially just this:

```
registerhead "%V" {
  name    = "myverbatim",
  action = function ()
    local i,j,verbatimlinesorig = tf:getblock()
    verbatimlines = verbatimlinesorig
  end,
}
```

Dednat6 would take each block of ‘%V’ lines and store its contents in the global variable `verbatimlines`, that I would process in Lua in ‘%L’ lines to generate the  $\text{\LaTeX}$  code that I want...

## Hacking

Hacking something usually consists of these stages:

- 1) “reading”: understanding docs, data structures, code
- 2) making tests, dumping data structures
- 3) “writing”: implementing new things

Here's how to do (1):

Learn a tiny bit of Emacs and eev:

<http://angg.twu.net/#eev>

and run the “eepitch blocks” in the Lua source files...

## Eepitch blocks in comments in Lua files

This is a comment block in `dednat6/diagforth.lua`:

```
--[==[
* (eepitch-lua51)
* (eepitch-kill)
* (eepitch-lua51)
require "diagforth"
storenode {TeX="a", tag="a", x=100, y=100}
storenode {TeX="b", tag="b", x=140, y=100}
= nodes

storearrow(DxyArrow {from="a", to="b", shape="|->",
                    slide="5pt", label="up",
                    placement="a"})
storearrow(DxyArrow {from="a", to="b", shape=".>"})
storearrow(DxyPlace {nodes["a"]})
storearrow(DxyLiteral {"literal foobar"})
= arrows
--] ==]
```

It is an “e-script” — an *executable log* of an experiment that I was doing. It can be “played back” by typing ‘F8’s in Emacs+eev — an ‘F8’ on a red star line runs that line as Lisp code (→ set up a target buffer)...

## Eepitch blocks in comments in Lua files (2)

```
--[==[
* (eepitch-lua51)
* (eepitch-kill)
* (eepitch-lua51)
require "diagforth"
storenode {TeX="a", tag="a", x=100, y=100}
storenode {TeX="b", tag="b", x=140, y=100}
= nodes

(...)
--] ==]
```

An ‘F8’ on a red star line runs that line as Lisp code (→ set up a target buffer with a Lua interpreter) and an ‘F8’ on a non-red star line sends that line to the target buffer as if the user had typed it...

## REPLs

Here's a screenshot.

```

--[[
* (eepitch-lua51)
* (eepitch-kill)
* (eepitch-lua51)
require "diagforth"
storenode {TeX="a", tag="a", x=100, y=100}
storenode {TeX="b", tag="b", x=140, y=100}
= nodes
|

Lua 5.1.5 Copyright (C) 1994-2012 Lua.org, PUC-Rio
> require "diagforth"
> storenode {TeX="a", tag="a", x=100, y=100}
> storenode {TeX="b", tag="b", x=140, y=100}
> = nodes
{ 1={"TeX"="a", "noden"=1, "tag"="a", "x"=100, "y"=100},
  2={"TeX"="b", "noden"=2, "tag"="b", "x"=140, "y"=100},
  "a"={"TeX"="a", "noden"=1, "tag"="a", "x"=100, "y"=100},
  "b"={"TeX"="b", "noden"=2, "tag"="b", "x"=140, "y"=100}
}
>
|

:-- diagforth.lua 94% L346 (Lua)  -:*-* xlua51x All L11 (Comint:run)

```

Left Emacs window: the e-script buffer. The cursor is there: ■.

We have just executed an eepitch block with 'F8's.

Right Emacs window: the target buffer, with a terminal running Lua 5.1 in interactive (Read/Eval/Print/Loop) mode. Blue '>': Lua prompts. Bold white: user input (sent with 'F8's).

Here we used just Lua, not Lua $\LaTeX$ .

## REPLs (2)

It is also possible to run Rob Hoelz's lua-repl from inside Lua<sub>TEX</sub>. Here's a screenshot.

```

\setbox0=\vbox{abc}
\directlua{print();print();syncrun{}}
\def\IGNORETHIS{
* (eepitch-shell)
* (eepitch-kill)
* (eepitch-shell)
luaTeX 2018tug-dednat6.tex
print(tex.box[0])
print(tex.box[0].id,          node.id("hlist"))
print(tex.box[0].list)
print(tex.box[0].list.id,    node.id("glyph"))
print(tex.box[0].list.char,  string.byte("a"))
print(tex.box[0].list.next)
print(tex.box[0].list.next.char, string.byte("b"))
}

>>> print(tex.box[0])
<node nil < 51723 > nil : hlist 2>
>>> print(tex.box[0].id,          node.id("hlist"))
0          0
>>> print(tex.box[0].list)
<node nil < 11817 > 11823 : glyph 256>
>>> print(tex.box[0].list.id,    node.id("glyph"))
29        29
>>> print(tex.box[0].list.char,  string.byte("a"))
97        97
>>> print(tex.box[0].list.next)
<node 11817 < 11823 > 51709 : glyph 256>
>>> print(tex.box[0].list.next.char, string.byte("b"))
98        98
>>> []

```

When you are a Bear of Very Little Brain — like me — Lua<sub>TEX</sub>'s interface to <sub>TEX</sub> boxes looks very hard... lua-repl may help.

## HEY!!!

From <http://angg.twu.net/dednat6.html>:

I've stopped trying to document dednat6 because

- 1) I don't have a mental image of who I am writing for,
- 2) I get *far too little feedback*,
- 3) all of the feedback that I got came from people who felt that I was not writing for them — my approach, tone and choice of pre-requisites were all wrong.

If you would like to try dednat6, get in touch, **let's chat** — *please!*

*Maybe I can typeset in 20 minutes a diagram that took you a day, maybe I can implement an extension that you need...*