Dednat6: An extensible (semi-)preprocessor for LuaLaTeX that understands diagrams in ASCII art

Eduardo Ochs

1 Prehistory

Many, many years ago, when I was writing my master's thesis, I realized that I was typesetting too many natural deduction trees, and that this was driving me mad. The code (in `proof.sty`) for a small tree like this one

$$\dfrac{\dfrac{\dfrac{[a]^1 \quad a \to b}{b} \qquad b \to c}{c}}{a \to c}\,1$$

was this:

```
\infer[{1}]{ a\to c }{
 \infer[{}]{ c }{
  \infer[{}]{ b }{
   [a]^1 &
   a\to b } &
  b\to c } } }
```

This was somewhat manageable, but the code for bigger trees was very hard to understand and to debug. I started to add 2D representations of the typeset trees above the code, and I defined a macro `\defded` to let me define the code for several trees at once, and a macro `\ded` to invoke that code later:

```
%  [a]^1  a->b
%  -----------
%       b       b->c
%       ------------
%            c
%          ----1
%          a->c
%
%          ^a->c
%
\defded{a->c}{
 \infer[{1}]{ a\to c }{
  \infer[{}]{ c }{
   \infer[{}]{ b }{
    [a]^1 &
    a\to b } &
   b\to c } } }
%
$$\ded{a->c}$$
```

Then I realized that if I made the syntax of my 2D representations a bit more rigid, I could write a preprocessor that would understand them directly, and write all the '`\defded`'s itself to an auxiliary file. If a file `foo.tex` had this (note: I will omit all header and footer code, like `\begin{document}` and `\end{document}`, from the examples),

```
\input foo.dnt

%:  [a]^1  a->b
%:  -----------
%:       b       b->c
%:       ------------
%:            c
%:          ----1
%:          a->c
%:
%:          ^a->c

$$\ded{a->c}$$
```

then I just had to run "`dednat.icn foo.tex;latex foo.tex`" instead of "`latex foo.tex`".

2 `dednat.lua`

A few years after that, I learned Lua, fell in love with it, and ported dednat.icn from Icon — which was a *compiled* language — to Lua.

The first novel feature in `dednat.lua` was a way to run arbitrary Lua code from the `.tex` file being preprocessed, and so extend the preprocessor dynamically. `dednat.lua` treated blocks of lines starting with '`%:`' as specifications of trees, and blocks of lines starting with '`%L`' as Lua code. More precisely, the initial set of *heads* was `{"%:", "%L", "%D"}`, and `dednat.lua` processed each block of contiguous lines starting with the same head in a way that depended on the head.

The second novel feature in `dednat.lua` was a way to generate code for categorical diagrams, or "2D diagrams" for short, automatically, analogous to what we did for trees. I wanted to make the preprocessor write the '`\defdiag`'s seen here itself:

```
%   LA <-| A
%   |      |
%   v      v
%   B |-> RB
%
\defdiag{adj_L-|R}{
  \morphism(0,0)/<-|/<400,0>[LA`A;]
  \morphism(0,0)/->/<0,-400>[LA`B;]
  \morphism(400,0)/->/<0,-400>[A`RB;]
  \morphism(0,-400)/|->/<400,0>[B`RB;]
}
$$\diag{adj_L-|R}$$
```

where '`\morphism`' is the main macro in `diagxy`, Michael Barr's front-end for XY-pic.

After months of experimentation I arrived at a good syntax for 2D diagrams. This code:

```
%D diagram adj_L-|R
%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 (( LA A <-|
%D    LA B -> A RB ->
%D    B RB |->
%D ))
%D enddiagram
%D
$$\diag{adj_L-|R}$$
```

generates this:

$$\begin{tikzcd} LA \arrow[d] & A \arrow[l, mapsto] \arrow[d] \\ B \arrow[r, mapsto] & RB \end{tikzcd}$$

The lines with '`%D 2Dx`' and '`%D 2D`' define a grid with coordinates and nodes, and the lines between '`%D ((`' and '`%D ))`' connect these nodes with arrows.

2.1 A Forth-based language for 2D diagrams — low-level ideas

The article "Bootstrapping a Forth in 40 lines of Lua code" [1] describes how a Forth-like language can be reduced to a minimal extensible core, and bootstrapped from it. The most basic feature in [1] is "words that eat text"; the fact that Forth is a stack-based language is secondary — stacks are added later. The code for '`%D`'-lines is based on [1].

A "Forth" — actually the "outer interpreter" of a Forth, but let's call it simply a "Forth" — works on one line of input at a time, reads each "word" in it and executes it as soon as it is read. A "word" is any sequence of one of more non-whitespace characters, and an input line is made of words separated by whitespace. The "outer interpreter" of Forth does essentially this on each line, in pseudocode:

```
while true do
  word = getword()
  if not word then break end
  execute(word)
end
```

Note that `word` is a global variable. The current input line is stored in `subj` and the current position of the parser is stored in `pos`; `subj` and `pos` are also global variables — which means the `execute(word)` can change them!

The function `getword()` parses whitespace in `subj` starting at `pos`, then parses a word and returns it, and advances `pos` to the position after that word. There is a similar function called `getrestofline()` that returns all the rest of the line from `pos` onwards, and advances `pos` to the end of the line.

One of the simplest Forth words is '`#`' ("comment"). It is defined as:

```
forths["#"] = function ()
    getrestofline()
  end
```

It simply runs `getrestofline()`, discards its return value, and returns. We say that `#` "eats the rest of the line".

In a "real" Forth we can define words using '`:`' and '`;`', like this:

```
 : SQUARE DUP * ;
```

but the Forth-based language in `dednat.lua` is so minimalistic that we don't have '`:`' and '`;`' — we define words by storing their Lua code in the table `forths`.

2.2 A Forth-based language for 2D diagrams — code for diagrams

Let's look at an example. This code

```
%D diagram T:F->G
%D 2Dx     100 +20 +20
%D 2D  100      A
%D 2D          /|\
%D 2D         v v v
%D 2D  +30 FA --> GA
%D 2D
%D (( A FA |-> A GA |->
%D    FA GA -> .plabel= b TA
%D    A FA GA midpoint |->
%D ))
%D enddiagram
%D
$$\diag{T:F->G}$$
```

yields this:

$$\begin{tikzcd} & A \arrow[dl, mapsto] \arrow[dr, mapsto] \arrow[d, mapsto, dashed] & \\ FA \arrow[rr, "TA"'] & {} & GA \end{tikzcd}$$

The word `diagram` eats a word — the name of the diagram — and sets `diagramname` to it. The word `2Dx` eats the rest of the line, and uses it to attribute x-coordinates to some columns. The word `2D` also eats the rest of the line; when it is followed by nnn or $+nnn$ that number gives the y-coordinate of that line, and the words that intersect a point that has both an x-coordinate and a y-coordinate become *nodes*. When a `2D` is not followed by an nnn

or $+nnn$ then this is a line without a y-coordinate, and it is ignored.

In a sequence like "`A FA |->`", both `A` and `FA` put nodes on the stack, and `|->` creates an arrow joining the two nodes on the top of the stack, without dropping the nodes from the stack. In a sequence like "`FA GA midpoint`" the `midpoint` creates a phantom node halfway between the two nodes on the top of the stack, drops (pops) them and pushes the phantom node in their place. The word `.plabel=` eats two words, a *placement* and a *label*, and modifies the arrow at the top of the stack by setting the arrow's label and placement attributes with them. The word '`((`' remembers the depth of the stack — 42, say — and the word '`))`' pops elements from the top of the stack; if the depth at '`))`' is 200 then '`))`' pops 200 − 42 elements to make the depth become 42 again.

The word `enddiagram` defines a diagram with the name stored in `diagramname`; each arrow that was created, even the ones that were dropped from the stack, becomes a call to `\morphism` — the main macro in `diagxy` — in the body of the diagram.

A good way to understand in detail how everything works is to inspect the data structures. Let's modify the code of the example to add some '`print`'s in '`%L`'-lines in the middle of the '`%D`'-code:

```
%D diagram T:F->G
%D 2Dx     100 +20 +20
%L print("xs:"); print(xs)
%D 2D  100      A
%D 2D          /|\
%D 2D         v v v
%D 2D  +30 FA --> GA
%L print("nodes:"); print(nodes)
%D 2D
%D (( A FA |-> A GA |->
%D    FA GA -> .plabel= b TA
%D    A FA GA midpoint -->
%L print("ds:"); print(ds)
%D ))
%L print("arrows:"); print(arrows)
%D enddiagram
```

The preprocessor outputs this on stdout:

```
xs:
{12=100, 16=120, 20=140}
nodes:
{ 1={"noden"=1, "tag"="A", "x"=120, "y"=100},
  2={"noden"=2, "tag"="FA", "x"=100, "y"=130},
  3={"noden"=3, "tag"="-->", "x"=120, "y"=130},
  4={"noden"=4, "tag"="GA", "x"=140, "y"=130},
  "-->"={"noden"=3, "tag"="-->", "x"=120, "y"=130},
  "A"={"noden"=1, "tag"="A", "x"=120, "y"=100},
  "FA"={"noden"=2, "tag"="FA", "x"=100, "y"=130},
  "GA"={"noden"=4, "tag"="GA", "x"=140, "y"=130}
}
ds:
12={"arrow"=4, "from"=1, "shape"="-->", "to"=5}
11={"TeX"="\\phantom{O}", "noden"=5, "x"=120,
    "y"=130}
10={"noden"=1, "tag"="A", "x"=120, "y"=100}
9={"arrow"=3, "from"=2, "label"="TA",
   "placement"="b", "shape"="->", "to"=4}
8={"noden"=4, "tag"="GA", "x"=140, "y"=130}
7={"noden"=2, "tag"="FA", "x"=100, "y"=130}
6={"arrow"=2, "from"=1, "shape"="|->", "to"=4}
5={"noden"=4, "tag"="GA", "x"=140, "y"=130}
4={"noden"=1, "tag"="A", "x"=120, "y"=100}
3={"arrow"=1, "from"=1, "shape"="|->", "to"=2}
2={"noden"=2, "tag"="FA", "x"=100, "y"=130}
1={"noden"=1, "tag"="A", "x"=120, "y"=100}
arrows:
{ 1={"arrow"=1, "from"=1, "shape"="|->", "to"=2},
  2={"arrow"=2, "from"=1, "shape"="|->", "to"=4},
  3={"arrow"=3, "from"=2, "label"="TA",
     "placement"="b", "shape"="->", "to"=4},
  4={"arrow"=4, "from"=1, "shape"="-->", "to"=5}
}
```

3 Semi-preprocessors

`dednat.icn`, `dednat.lua` and all its successors until `dednat5.lua` were preprocessors in the usual sense — they had to be run *outside* `latex` and *before* `latex`. With dednat6 this changed; dednat6 can still be run as a preprocessor, but the recommended way to run it on, say, `foo.tex`, is to put a line like

```
\directlua{dofile "dednat6load.lua"}
```

somewhere near the beginning of `foo.tex`, add some calls to `\pu` at some points — as we will explain soon — and compile `foo.tex` with `lualatex` instead of `latex`, to make `foo.tex` be processed "in parallel" by TeX and by Lua. That "in parallel" is a simplification, though; consider this example:

```
%:
%:  a  b
%:  ----
%:   c
%:
%:   ^my-tree
%:
$$\pu\ded{my-tree}$$
%:
%:  d  e  f
%:  -------
%:     g
%:
%:     ^my-tree
%:
$$\pu\ded{my-tree}$$
```

Suppose that this fragment starts at line 20. (As mentioned above, we are omitting the header and footer — e.g., `\begin{document}` and `\directlua {dofile "dednat6load.lua"}`.)

We have a `%:`-block from lines 20–26, a call to `\pu` at line 27, another `%:`-block from lines 28-34, and another call to `\pu` at line 35.

The output of the first `%:`-block above is a `\defded{my-tree}`, and the output of the second `%:`-block above is a *different* `\defded{my-tree}`.

'`\pu`' means "process until" — or, more precisely, *make dednat6 process everything until this point that it hasn't processed yet*. The first `\pu` processes the lines 1–26 of `foo.tex`, and "outputs" — i.e., sends to TEX — the first `\defded{my-tree}`; the second `\pu` processes the lines 28–34 of `foo.tex`, and "outputs" the second `\defded{my-tree}`. Thus, it is not technically true that TEX and dednat6 process `foo.tex` in parallel; dednat6 goes later, and each `\pu` is a synchronization point.

3.1 Heads and blocks

In order to understand how this idea — "semi-preprocessors" — is implemented in dednat6 we need some terminology.

The initial *set of heads* is `{"%:", "%L", "%D"}`. It may be extended with other heads, but we may only add heads that start with '`%`'.

A *block* is a set of contiguous lines in the current `.tex` file. This code

```
Block {i=42, j=99}
```

creates and returns a block that starts on line 42 and ends on line 99. The Lua function `Block` receives a table, changes its metatable to make it a "block object", and returns the modified table.

A *head block* is a (maximal) set of contiguous lines all with same head. Head blocks are implemented as blocks with an extra field `head`. For example:

```
Block {i=20, j=26, head="%:"}
```

A block is *bad* when it contains a part of a head block but not the whole of it. We avoid dealing with bad blocks — dednat6 never creates a block object that is "bad".

Each head has a *processor*. *Executing* a head block means running it through the processor associated with its head. Executing an arbitrary (non-bad) block means executing each head block in it, one at a time, in order. Note: the code for executing non-bad arbitrary blocks was a bit tricky to implement, as executing a '`%L`'-block may change the set of heads and the processors associated to heads.

A *texfile block* is a block that refers to the whole of the current `.tex` file, and that has an extra field `nline` that points to the first line that dednat6 hasn't processed yet. If `foo.tex` has 234 lines then the texfile block for `foo.tex` starts as:

```
Block {i=1, j=234, nline=1}
```

We saw in sections 1 and 2.2 that the "output" of a `%:`-block is a series of '`\defded`'s and the "output" of a `%D`-block is a series of '`\defdiags`'s. We can generalize this. For example, the "output" of

```
%L output [[\def\Foo{FOO}]]
%L output [[\def\Bar{BAR}]]
```

is

```
\def\Foo{FOO}
\def\Bar{BAR}
```

The *output* of a head block is the concatenation of the strings sent to `output()` when that block is executed. The output of an arbitrary (non-bad) block is the concatenation of the strings sent to `output()` by its head blocks when the arbitrary block is executed.

A `\pu`-*block* is created by dednat6 when a `\pu` is executed, pointing to the lines between this `\pu` and the previous `\pu`. If `foo.tex` has a `\pu` at line 27 and another at line 35 then the first `\pu` creates this block,

```
Block {i=1, j=26}
```

and the second `\pu` creates this:

```
Block {i=28, j=34}
```

As '`\pu`'s only happen in non-comment lines, `\pu`-blocks are never bad.

3.2 The implementation of `\pu`

The macro `\pu` is defined as

```
\def\pu{\directlua{
    processuntil(tex.inputlineno)
  }}
```

in LATEX, and `processuntil()` is this (in Lua):

```
processuntil = function (puline)
    local publock =
      Block {i=tf.nline, j=puline-1}
    publock:process()
    tf.nline = puline + 1
  end
```

Here's a high-level explanation. When dednat6 is loaded and initialized it creates a texfile block for the current `.tex` file — with `nline=1` — and stores it in the global variable `tf`. The macro `\pu` creates a `\pu`-block that starts at line `tf.nline` and ends at line `tex.inputlineno - 1`, executes it, and advances `tf.nline` — i.e., sets it to `tex.inputlineno + 1`.

The code above looks simple because the line `publock:process()` does all the hard work.

4 Creating new heads

New heads can be created with `registerhead`, and they are recognized immediately. For example, this

```
%L eval = function (str)
%L     return assert(loadstring(str))()
%L   end
%L expr = function (str)
%L     return eval("return "..str)
%L   end
%L
%L registerhead "%A" {
%L   name   = "eval-angle-brackets",
%L   action = function ()
%L       local i,j,str = tf:getblockstr()
%L       str = str:gsub("<(.-)>", expr)
%L       output(str)
%L     end,
%L }
%A $2+3 = <2+3>$
\pu
```

produces "$2 + 3 = 5$"; that looks trivial, but it is easy to write bigger examples of '`%A`'-blocks with `pict2e` code in them, in which the Lua expressions in '`<...>`'s generate '`\polyline`'s and '`\puts`'s whose coordinates are all calculated by Lua.

5 A read-eval-print-loop (REPL)

Dednat6 uses only one function from the LuaTEX libraries — `tex.print` — and two variables, `status.filename` and `tex.inputlineno`, but it includes a nice way to play with the other functions and variables in the libraries.

Dednat6 includes a copy of `lua-repl` (by Rob Hoelz, `github.com/hoelzro/lua-repl`), and we can invoke it by running `luarepl()`. If we put this in our `foo.tex`,

```
\setbox0=\hbox{abc}
\directlua{luarepl()}
```

then running `lualatex foo.tex` will print lots of stuff, and then the prompt '`>>>`' of the `lua-repl` inside dednat6; if we send these commands to the REPL,

```
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"))
```

we get this in the terminal:

```
>>> print(tex.box[0])
<node    nil <  35981 >    nil : hlist 2>
>>> print(tex.box[0].id,        node.id("hlist"))
0 0
>>> print(tex.box[0].list)
<node    nil <   6107 >   6114 : 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   6107 <   6114 >  32849 : glyph 256>
>>> print(tex.box[0].list.next.char,
>>>>                            string.byte("b"))
98 98
>>>
```

The best way to use `luarepl()` — in my not so humble opinion — is from Emacs, with the `eev` library. The tutorial of eev at

`http://angg.twu.net/eev-intros/find-eev-quick-intro.html`

explains, in the section "Controlling shell-like programs", how we can edit the commands to be sent to `lualatex` in a buffer, called the "notes buffer", and send them line by line to another buffer that runs `lualatex foo.tex` in a shell — the "target buffer"; each time that we type the F8 key Emacs sends the current line to the program running in the target buffer, *as if the user had typed it*.

6 Availability

Dednat6 is not in CTAN yet (as of October, 2018). Until it gets there you can download it from:

`http://angg.twu.net/dednat6.html`

References

[1] E. Ochs: *Bootstrapping a Forth in 40 Lines of Lua Code*. Chapter 6 (pp. 57–70) of *Lua Programming Gems*, L.H. de Figueiredo, W. Celes, and R. Ierusalimschy, eds. `lua.org/gems`, 2008. Available from `http://angg.twu.net/miniforth-article.html`.

◇ Eduardo Ochs
eduardoochs (at) gmail dot com
http://angg.twu.net/dednat6.html