1 Other inference bars

All the examples of deduction trees in the TUGBoat article use ‘-’s for the inference bars in the ASCII art representation. If we use ‘=’s instead of ‘-’s we get double bars, and if we use ‘:’s we get a line of vertical dots instead of a bar:

$$\begin{array}{c} ,PR \quad ,QR \\ \hline\hline ,PQR \end{array} \quad \Longrightarrow \quad \begin{array}{ccc} & [P]^1 & [Q]^1 \\ \vdots & \vdots & \vdots \\ PQ & R & R \\ \hline & R & \end{array} \; 1$$

You can change the number of vertical dots by redefining the macro ‘\DeduceSym’. For example:

```
\makeatletter
% Original with 4 dots (from proof.sty):
% \def\DeduceSym{\vtop{\baselineskip4\p@ \lineskiplimit\z@
%     \vbox{\hbox{.}\hbox{.}\hbox{.}}\hbox{.}}}
% New, with 3 dots:
\def\DeduceSym{\vtop{\baselineskip4\p@ \lineskiplimit\z@
    \vbox{\hbox{.}\hbox{.}}\hbox{.}}}
\makeatother
```

2 Abbrevs

The first Dednats did not support UTF-8, and the way to write a tree node that would display as ‘*ab*’ was to write it as ‘`a->b`’ after running `addabbrevs("->", "\to ")`. The module `abbrevs.lua` implements this, and `unabbrev(str)` parses `str` from left to right, at each point looking for the longest string starting at that point that is an abbrev and replacing it by its expansion, or leaving that character untouched if it doesn't have an expansion. Here is an example:

```
%L addabbrevs("->", "\\to ")
%:
%:  [a]^1  a->b
%:  -----------
%:       b       b->c
%:       ------------
%:             c
%:          ----1
%:          a->c
%:
%:          ^a->c
%:
$$\pu \ded{a->c}$$
```

$$\to \qquad \begin{array}{c} \begin{array}{c} [a]^1 \quad a\to b \\ \hline b \end{array} \qquad b\to c \\ \hline \begin{array}{c} c \\ \hline a\to c \end{array} \; 1 \end{array}$$

Abbrevs are also used in 2D diagrams, in a more complex way. Section 2.2 of the TUGBoat article explains how the grid words create a table `nodes`, but it doesn't explain how the fields `.tex` and `.TeX` in a node affect how it is displayed. The code below creates nodes whose *tags* are `"A"`, `"B"`, `"C"`, `"D"`, and then changes the fields `.tex` and `.TeX` in some of these nodes. The TEX code for each node is calculated by the function `node_to_TeX`, that expects a node (a table) and returns a string. If `node_to_TeX` receives a node that has a `.TeX` field then it returns that field unchanged, surrounded by '`{}`'s; if it doesn't have a `.TeX` field but it has a `.tex` field then it returns the result of running `unabbrev` on that field and surrounding it with '`{}`'s; otherwise it returns the result of running `unabbrev` on the tag surrounding it with '`{}`'s. For example:

```
%D diagram nodes-and-abbrevs
%D 2Dx     100 +40
%D 2D  100 A -> B
%D 2D      |    |
%D 2D      v    v
%D 2D  +30 C -> D
%D 2D
%D (( B .tex= (a->b)
%D    C              .TeX= (a->b)
%D    D .tex= (a->b) .TeX= (a->b)
%L print("nodes:"); print(nodes)
%L print("A:", node_to_TeX(nodes["A"]))
%L print("B:", node_to_TeX(nodes["B"]))
%L print("C:", node_to_TeX(nodes["C"]))
%L print("D:", node_to_TeX(nodes["D"]))
%D    A B -> A C -> B D -> C D ->
%D ))
%D enddiagram
```

→

$$\begin{tikzcd}
A \arrow[r] \arrow[d] & (a\to b) \arrow[d] \\
(a- > b) \arrow[r] & (a- > b)
\end{tikzcd}$$

The output of the `print()`s is:

```
nodes:
{   1={"noden"=1, "tag"="A", "x"=100, "y"=100},
    2={"noden"=2, "tag"="B", "x"=140, "y"=100, "tex"="(a->b)"},
    3={"noden"=3, "tag"="C", "x"=100, "y"=130,                  "TeX"="(a->b)"},
    4={"noden"=4, "tag"="D", "x"=140, "y"=130, "tex"="(a->b)", "TeX"="(a->b)"},
  "A"={"noden"=1, "tag"="A", "x"=100, "y"=100},
  "B"={"noden"=2, "tag"="B", "x"=140, "y"=100, "tex"="(a->b)"},
  "C"={"noden"=3, "tag"="C", "x"=100, "y"=130,                  "TeX"="(a->b)"},
  "D"={"noden"=4, "tag"="D", "x"=140, "y"=130, "tex"="(a->b)", "TeX"="(a->b)"}
}
A:      {A}
B:      {(a\to b)}
C:      {(a->b)}
D:      {(a->b)}
```

3 Renaming

The word `ren` in the language for 2D diagrams eats the rest of the line, splits it at the '`==>`', and splits the material before the '`==>`' into a list of *tags*, $A_1, \ldots, A_n$, and the material after '`==>`' into a list of *texs*, $B_1, \ldots, B_n$; these two lists must have the same length, and then `ren` runs `nodes[`A_i`].tex =` B_i for each i in $1, \ldots, n$. For example:

```
%D diagram ren
%D 2Dx     100    +30
%D 2D  100 A1 <-| A2
%D 2D      |       |
%D 2D      v       v
%D 2D  +30 A3 |-> A4
%D 2D
%D ren A1 A2 ==> A{\times}B A
%D ren A3 A4 ==> C B{\to}C
%D
%D (( A1 A2 <-| .plabel= a ({\times}B)
%D    A1 A3 -> A2 A4 ->
%D    A3 A4 |-> .plabel= b (B{\to})
%D ))
%D enddiagram
```

→

$$\begin{tikzcd}
A{\times}B \arrow[d] & A \arrow[l, mapsto, "({\times}B)"'] \arrow[d] \\
C \arrow[r, mapsto, "(B{\to})"'] & B{\to}C
\end{tikzcd}$$

4 Arrow modifiers

The language for 2D diagrams in dednat6 has some words for curving and sliding arrows:

```
%D diagram curve-slide
%D 2Dx     100 +40
%D 2D  100 A   B
%D 2D
%D 2D  +40 D   C
%D 2D
%D (( A B ->                .plabel= b \text{one}
%D    A B -> .slide=  5pt  .plabel= a \text{two}
%D    A B -> .slide= 20pt  .plabel= a \text{three}
%D    B C -> .curve= _10pt .plabel= l \text{four}
%D    B C -> .curve=  ^5pt .plabel= r \text{five}
%D    C D ->
%D    C D -> .curve=  _5pt
%D    C D -> .curve=  _5pt .slide= -5pt
%D          .plabel= a \text{six}
%D ))
%D enddiagram
```

→

$$\begin{tikzcd}
A \arrow[r, "\text{one}"'] \arrow[r, shift left=5pt, "\text{two}"] \arrow[r, shift left=20pt, "\text{three}"] & B \arrow[d, bend right, "\text{four}"'] \arrow[d, bend left=15, "\text{five}"] \\
D & C \arrow[l] \arrow[l, bend right=15] \arrow[l, bend right=15, shift right=5pt, "\text{six}"']
\end{tikzcd}$$

The words ‘`sl^^`’, ‘`sl^`’, ‘`sl_`’, and ‘`sl__`’ are abbreviations for “`.slide= 5pt`”, “`.slide= 2.5pt`”, “`.slide= -2.5pt`”, “`.slide= -5pt`” respectively.