The dvgloss package

Dan Bridges Velleman

February 12, 2013

Contents

1 What it does

This is a package that lets you set interlinear linguistic examples like the one below:

A linguistic example

K'a tee ka-r-il ri achih ri jun keej xaa maa pwaq
suddenly INC-A3s-see the man the one horse just EXCLAM money

k-u-kisii-j.
INC-A3s-shit-SS

Suddenly, the man saw a horse that was just shitting money.

Mondloch 1981

```
\gl{{K'a tee} ka-r-il ri achih ri jun keej xaa maa pwaq
            k-u-kisii-j.}
   {{suddenly} {\sc inc-a}3s-see the man the one horse
            just {\sc exclam} money {\sc inc-a}3s-shit-{\sc ss}}
\ft{Suddenly, the man saw a horse that was just shitting money.}
\lb{Mondloch 1981}
```

Some characters have been set up as shortcuts for various useful things within an example — inserting a line break (using `/`) putting subscripts on brackets (*e.g.* `]Contr` in the example below), etcetera.

Using a shortcut

No [*moriré*]$_{\text{Contr}}$
not will.die

sino que [*viviré*]$_{\text{Contr}}$
but will.live

You will not die, but will live.

```
\gl{No [ morir\'e ]Contr / {sino que} [ vivir\'e ]Contr}
   {not will.die but will.live}
\ft{You will not die, but will live.}
```

What makes this package different from others that exist so far (and in particular from `expex`, which offers a similar set of shortcuts) is that these shortcuts are *customizable*. You can declare new characters as shortcuts, change the meanings of existing shortcuts, and so on. For instance, if the slash hadn't already been set up as a newline character, you could make it one like this:

```
\makeglshortcut / {\par}
```

If you wanted to repurpose it — let's say you've been using it to mark prosodic boundaries, but you decide you want those to show up as the IPA double-pipe character as in the example below — you could do that too.

Redefining a shortcut

No [*moriré*]$_{\text{Contr}}$ ‖ *sino que* [*viviré*]$_{\text{Contr}}$
not will.die rather will.live

You will not die, but will live.

```
\begingroup
\makeglshortcut / {\unskip\quad$\|$\quad}
\gl{No [ morir\'e ]Contr / {sino que} [ vivir\'e ]Contr}
   {not will.die rather will.live}
\ft{You will not die, but will live.}
\endgroup
```

Finally, for one-off situations where you don't want to define a new shortcut, there is an "escape character" which lets you issue any command you want from within an argument of `\gl`. Suppose you redefine `/` to generate the double-pipe character, but then you run into a tricky formatting situation where you really need to force a line break. Just say `!{\par}` — or, for that matter, `!{\par\vspace{\jot}\mbox{}\qquad}`, if that's what floats your boat — and you'll get what you want. (The default escape character is `!`, but if you're a Khoisanist or otherwise need to start words with a bang, you can redefine it to something else.)

2 What it doesn't do

- It doesn't handle example numbering. There are plenty of other packages that do, so I doubt I'll bother to reinvent that particular wheel any time soon.
- It doesn't do three-line glosses, or anything bigger. This is a dumb limit and could easily be removed — but I probably won't get around to it unless someone else asks, because two-line glosses are all I ever use.
- Shortcuts or escaped commands can take arguments, and can (if you arrange it right) take scope over more than one word of interlinear gloss; but there are limits — in particular, scope-taking commands can't be *nested*. This is a limit that could in principle be removed, but it would take some work.

So for instance, as an example of that last limit: you can define a pair of shortcut characters that will cause a box to be drawn around any number of intervening words, as in the example below; but you will not be able to use those shortcuts to produce nested boxes.

A scope-taking shortcut

Los	*frase-s*	*nominal-es*	*está-n*	*en*	*caja-s.*	*No*	*sé*	*porqué.*
the	phrase-pl	nominal-pl	are-3pl	in	box-pl	not	know.1sg	why

The noun phrases are in boxes. I don't know why.

```
\gl{Los * frase-s nominal-es * est\'a-n en * caja-s. * No s\'e porqu\'e.}
   {the phrase-pl nominal-pl are-3pl in box-pl not know.1sg why}
\ft{The noun phrases are in boxes. I don't know why.}
```

3 How to use it

3.1 The basics

`\gl` `\ft`

The `\gl` macro takes two arguments. It treats each argument as a list of (space-delimited) items, and aligns the items from the first argument atop the items from the second.

The `\ft` macro is for putting a *free translation* after an interlinear gloss. It takes a single argument.

A generic example

dolorem	*ipsum*	*quia*	*dolor*	*sit*	*amet*
pain.acc.sg	itself.acc.sg	because	because	pain.nom.sg	be.3sg.subj

"loves pain itself because it is pain"

```
\gl{dolorem ipsum quia dolor sit amet}
   {pain.acc.sg itself.acc.sg because
          because pain.nom.sg be.3sg.subj love.3sg}
\ft{``loves pain itself because it is pain''}
```

`\lb` The `\lb` macro gives you a "label" — a bit of text set on the same line as what came before (if there's room) but pushed right to the end of the line. You can use this, among other things, for citations.

An example with a label

dolorem	*ipsum*	*quia*	*dolor*	*sit*	*amet*
pain.acc.sg	itself.acc.sg	because	pain.nom.sg	be.3sg.subj	love.3sg

loves pain itself just for being pain — Cicero

```
\gl{dolorem ipsum quia dolor sit amet}
   {pain.acc.sg itself.acc.sg because pain.nom.sg be.3sg.subj love.3sg}
\ft{loves pain itself just for being pain}
\lb{Cicero}
```

If there isn't room for the label on one line, a line break will be added.

An example with a long label

dolorem	*ipsum*	*quia*	*dolor*	*sit*	*amet*
pain.acc.sg	itself.acc.sg	because	pain.nom.sg	be.3sg.subj	love.3sg

loves pain itself just for being pain
Cicero, *de Finibus Bonorum et Malorum*

```
\gl{dolorem ipsum quia dolor sit amet}
   {pain.acc.sg itself.acc.sg because pain.nom.sg be.3sg.subj love.3sg}
\ft{loves pain itself just for being pain}
\lb{Cicero, \emph{de Finibus Bonorum et Malorum}}
```

3.2 Grouping

You can use curly braces to group multiple words together; `\gl` will then treat them like a single word.

Grouping two words together

Edormi crapulam
sleep off hangover

"Sleep off that hangover."
Cicero, Philippic 2 30

```
\gl{Edormi crapulam}
   {{sleep off} hangover}
\ft{``Sleep off that hangover.''}
\lb{Cicero, Philippic 2 30}
```

The `\gl` macro ignores spaces in all the same places that LaTeX normally ignores spaces — for instance after a control sequence. To make sure that a space after a control sequence will be recognized, wrap the control sequence in curly braces, as in the first two words below.

Spaces ignored after control sequences

å *nå* *Blå-ås-en*
to reach blue-ridge-def

to reach Blååsen

```
\gl{{\aa} {n\aa} Bl\aa-\aa s-en}
   {to reach blue-ridge-def}
\ft{to reach Bl\aa\aa sen}
```

3.3 Prebuilt shortcuts

For a shortcut character to do anything at all, (i) it must be in the first argument of `\gl`, and (ii) it must be at the beginning of a word.

If the shortcut character has non-space characters after it, they will be treated as an argument. The right-bracket shortcut which we saw back in §1 is an example of this: if you type `]Contr`, then `Contr` is treated as the shortcut's argument — which in this case means it's typeset as a subscript.

`/` Insert a line break. The argument, if present, specifies an amount of extra vertical space to be added afterward.

`[` Insert a left bracket in the top line, with nothing aligned with it below. The argument, if present, specifies a label to be set as a subscript to the left bracket.

`]` As above, but insert a right bracket.

`* *` Everything between the two asterisks is set with a `\framebox` around it. If an argument is present after the second asterisk, it's set as a label on top of the `\framebox`.

3.4 New shortcuts

`\makeglshortcut` To define a new shortcut along the lines of `/` and `[`, use the command `\makeglshortcut`. It takes two arguments — the first is the character you want to turn into a shortcut, the second is the macro code that will be executed when that character is encountered. If the shortcut character is encountered with other, non-space characters after it, those will be passed to the macro code you supply as an argument — so you can capture those characters as `#1`. (If you try to do something with `#1` but the user hasn't supplied an argument, then `#1`will be empty. You might want to test for this — as in the example below — if this is an issue. If the user does supply an argument, but you don't do anything with it, then that's not a problem — the argument will be silently discarded.)

We've already seen an example of a shortcut definition that doesn't use an argument.

```
\makeglshortcut / {\par}
```

Here's an example of one which does.

```
\makeglshortcut / {\par\ifx#1\empty\else\vskip#1\fi}
```

Shortcuts have to be single characters. `\makeglshortcut{somethingverylong}{code}` will not do what you want.

`\makeglsurround` To define a new shortcut along the lines of `*` — a set of delimiters which can surround multiple words of glossed text and "take scope" over them — use the command `\makeglsurround`. It takes three arguments: the left-hand delimiter, the right-hand delimiter, and the code that will be executed when they're encountered. The two delimiters can be the same — as with `*`, above — but they don't have to be.

If there are extra non-space characters after the left-hand delimiter, they are passed in as argument `#1`. The characters between the two delimiters are passed in as argument `#2`. Extra non-space characters after the right-hand delimiter are passed in as argument `#3`. You'll almost certainly want to do something with `#2`, but ignoring `#1` and `#3` is completely fine.

As an example, here's how the asterisk shortcut was defined:

```
\makeglsurround ** {%
  \rlap{\raisebox{1.5em}{\footnotesize\sf#3}}%
  \fbox{\noglstrut#2\unskip}\glspace}
```

3.5 The escape character

Spacing and line-beaking commands, among others, need to be "escaped" if you want to use them inside of `\gl`. (See §4 for an explanation of why this is. For now, just trust me.) The escape character by default is `!`, and it's best to enclose whatever comes after it in curly braces. So you could use `!{\par)}` to insert a new line if you'd already redefined `/`, or `!{\qquad}` to insert some extra horizontal space.

`\glescape` If you want to change the escape character, redefine `\glescape`. Like shortcuts, the escape has to be a single character. `\def\glescape{averylongescape}` will not do what you want.

4 What's going on

Internally, there is a command called `\glossword` which is responsible for stacking two words on top of one another. The arguments you give to the `\gl` command are converted into a series of `\glosswords`, like so:

```
\gl{a b c d}{1 2 3 4}
   ===>
\glossword{a}{1}
\glossword{b}{2}
\glossword{c}{3}
```

```
\glossword{d}{4}
```

Normally, command sequences that appear inside of `\gl` will be wrapped up inside a `\glossword`. Sometimes this is what you want.

```
\gl{a b \bf c d}{1 2 3 4}
   ===>
\glossword{a}{1}
\glossword{b}{2}
\glossword{\bf c}{3}
\glossword{d}{4}
```

But sometimes it is really not at all what you want.

```
\gl{a b \par c d}{1 2 3 4}
   ===>
\glossword{a}{1}
\glossword{b}{2}
\glossword{\par c}{3}
\glossword{d}{4}
```

So we need a way to get a command like `\par` to show up *outside* of a `\glossword`. That's what the escape character is for.

```
\gl{a b !{\par} c d}{1 2 3 4}
   ===>
\glossword{a}{1}
\glossword{b}{2}
\par
\glossword{c}{3}
\glossword{d}{4}
```

The expansion of a shortcut character also ends up outside of any `\glosswords`.

```
\gl{a b / c d}{1 2 3 4}
   ===>
\glossword{a}{1}
\glossword{b}{2}
\par
\glossword{c}{3}
\glossword{d}{4}
```

5 Code

To begin with we set up temporary token list registers, and some spacing and formatting parameters.

```
\newtoks\ta\newtoks\tb
\newdimen\glhangindent\glhangindent=2em
\newdimen\betweenglskip\betweenglskip=1\jot
\newdimen\withinglskip\withinglskip=0pt
\newdimen\aboveglftskip\aboveglftskip=2\jot
\newdimen\aboveglskip\aboveglftskip=2\jot
\newdimen\glstrutheight\newdimen\glstrutdepth
\def\glspace{\penalty0\hspace{1ex plus 2em minus 2pt}}
\def\everygla{\itshape}
\def\everyglb{}
```

We also specify an escape character.

```
\def\glescape{!}
```

`\addtokens` `\popoff` Here's some helper macros for wrangling lists in which the items are separated by spaces.

```
\long\def\addtokens#1\to#2{\ta={#1}\tb=\expandafter{#2}\edef#2{\the\tb\the\ta}}
\def\pop#1\to#2{\expandafter\popoff#1\to#2\remainderin#1}
\long\def\popoff#1 #2\to#3\remainderin#4{#3={#1}\def#4{#2}}
```

`\split` `\istchar` `\restchars` Here's one for splitting off a single token from a longer string. If you execute `\split{Lorem}`, then the `L` is stored in `\istchar` and the rest of the word — `orem` — is stored in `\restchars`.

```
\long\def\split#1{\expandafter\ssplit#1\xyzzy}
\long\def\ssplit#1#2\xyzzy{\gdef\istchar{#1}\gdef\restchars{#2}}
```

`\ifnotin` There's one truly devious helper macro here: a really tricky way of testing whether one serious of tokens occurs within another. This is cribbed from the doc package, which credits Michael Spivak with the original invention.

```
\def\ifnotin#1#2{%
  \def\@ifnotin##1#1##2##3\zyzzy{\ifx\ifnotfound##2}%
  \expandafter\@ifnotin#2#1\notfound\zyzzy}
```

`\ifspecial` `\checkspecial` `\makespecial` We need this bit of deviousness because we're going to keep a list of characters that have been declared as special shortcuts. `\makespecial` adds something to the list; `\checkspecial` checks whether it's on the list (using `\ifnotin`) and stores the answer in `\ifspecial`.

```
\newif\ifspecial
\def\dvglspecials{}
\def\checkspecial#1{%
  \ifnotin#1{\dvglspecials}\specialfalse\else\specialtrue\fi}
\def\makespecial#1{%
  \xdef\dvglspecials{\dvglspecials#1}}
```

5.1 Glossing

`\glossword` First we define the basic formatting macro: it stacks a word and its gloss atop one another.

```
\def\glossword#1#2{%
  \mbox{\vtop{\halign{##\hfil\cr\everygla#1\strut\cr\everyglb#2\strut\cr}}}%
  \glstrut\glspace}
```

`\zipper` Now we have the rearranging macro. To use a functional programming analogy: `\zipper\a\and\b\to\c` basically means `c = zipWith(glossword) a b`. In other words, if `\a` is a list macro containing the tokens `1 2 3` and `\b` is a list macro containing the tokens `4 5 6`, and we execute `\zipper\a\and\b\to\c`, then `\c` will end up containing `\glossword{1}{4}\glossword{2}{5}\glossword{3}{6}`.

In fact, though, `\zipper` is not quite `zipWith`. It also catches "special" or escaped tokens in its first argument and adds them to the output *without* pairing them off or wrapping them in a `\glossword`.

```
\def\zipper#1\and#2\to#3{%
```

If there's any items left on list #1, pull one off (storing it in `\ta`) and split off its first token so we can peek and see if it's an escape character or a special character.

```
  \ifx\empty#1\else%
    \pop#1\to\ta%
    \def\istchar{}\def\restchar{}%
    \edef\temp{\the\ta}\split\temp%
```

If that first token is an escape character, we pass the rest of the item through to the output.

```
    \ifx\istchar\glescape\expandafter\addtokens\restchars\to#3%
```

If that first token is a special character, we convert it into a control sequence and make the remaining characters in the item into its argument.

```
    \else%
      \expandafter\checkspecial\istchar\ifspecial%
        \expandafter\addtokens\csname gl\istchar\endcsname\to#3%
        \expandafter\addtokens\expandafter<\restchars>\to#3%
```

If neither of those two conditions hold, then we're just building a normal glossword. That means we should also pull an item off of list #2, storing it in `\tb`.

```
      \else
        \ifx\empty#2\else%
          \pop#2\to\tb%
```

Finally, we wrap the contents of `\ta` and `\tb` in a `\glossword` command and pass it to the output. Here we need to be careful: we want to expand `\the\ta` and `\the\tb` *once* — to *get* their contents. But having gotten their contents, we don't want to do any further expansion.[1]

```
          \edef\temp{\noexpand\glossword{\the\ta}{\the\tb}}%
          \expandafter\addtokens\temp\to#3%
        \fi%
      \fi%
```

[1] Why not? Because trying to expand an unexpandable macro here would be messy and unpleasant.

```
    \fi%
    \zipper#1\and#2\to#3%
  \fi%
}
```

`\gl` Based on `\zipper`, we can define a user-friendly gloss command: take two arguments, `\zipper` them together, call `\fixglstrut` (see below) to make sure everything is spaced real nice, and send everything off to be typeset.

```
\long\def\gl#1#2{%
  \begingroup%
  \def\x{#1 }\def\y{#2 }\def\z{}%
  \zipper\x\and\y\to\z%
  \fixglstrut%
  \ifvmode\vskip\aboveglskip\fi\z%
  \endgroup%
}
```

`\fixglstrut` `\noglstrut` The `\fixglstrut` command is issued in `\gl` right before we begin typesetting. It sets `\glstrut` to be as tall as a normal glossbox at the current font size *plus* the current value of `\betweenglskip`. Sometimes we'll also want to be able to suppress `\glstrut` temporarily as a way of preventing the `\betweenglskip` space from being inserted. `\noglstrut` does that. (It's temporary because `\fixglstrut` will be called again at the beginning of the next gloss and things will go back to normal. If you issue `\noglstrut` within a group, it's of course even temporary-er than that.)

```
\def\fixglstrut{%
  \def\glstrut{}%
  \setbox0=\hbox{\glossword{}{}}%
  \glstrutheight=\ht0%
  \glstrutdepth=\dp0%
  \advance\glstrutdepth by \betweenglskip%
  \global\edef\glstrut{\vrule height\glstrutheight width0pt depth\glstrutdepth}}

\def\noglstrut{\def\glstrut{}}
```

`\ft` Usually after an interlinear gloss there's a line of "free translation." This is a user-facing command for setting that free translation line — or, really, for setting *anything* that needs to come on a new line right after an interlinear gloss. The only time this matters is when `betweenglskip` and `belowglskip` are different sizes.

```
\def\ft#1{\nobreak\par\nobreak\vskip-\betweenglskip\vskip\aboveglftskip #1}
```

`\lb` Here's a labeling macro, swiped from the TEXbook.

```
\def\lb#1{{\unskip\nobreak\hfil\penalty0\hskip2em\mbox{}\nobreak\hfill\mbox{#1}}}
```

5.2 Declaring shortcuts

We have a macro `\makespecial` which will tell the parser "treat this character as special."

But that alone won't get us anywhere interesting. We also need to *define* the macro that will be executed when the special character in question is encountered. And it would be nice if we could wrap both steps — calling `\makespecial` and defining the macro — in a user-friendly package. That's what this last section does.

Suppose we want to declare `/` as a shortcut character which inserts a line break. Step one is simple:

```
\makespecial{/}
```

Now let's consider how the parser will respond after we've done this. Suppose we then execute this command:

```
\gl{a b * c}{1 2 3}
```

The output which is sent to be typeset will look like this

```
\glossword{a}{1}
\glossword{b}{2}
      \gl/<>
\glossword{c}{3}
```

where `\gl/` is a control sequence — one which could not normally be input directly. And if we execute the command

```
\gl{a b /argument c}{1 2 3}
```

The output which is sent to be typeset will look like this:

```
\glossword{a}{1}
\glossword{b}{2}
  \gl/<argument>
\glossword{c}{3}
```

Step two, then, is to define a macro that will do something useful in those contexts. If the slash were a normal character, this would be easy:

```
\def\gl/<#1>{\par\ifx#1\empty\else\vskip{#1}\fi}
```

But since the slash isn't a normal character, we need to do it this way instead:

```
\expandafter\def\csname gl/\endcsname<#1>{\par\ifx#1\empty\else\vskip{#1}\fi}
```

`\makeglshortcut` Well, that's just what `\makeglshortcut` does: carries out Step One and Step Two for an arbitrary character and arbitrary macro code.

```
\long\def\makeglshortcut#1#2{%
  \makespecial{#1}%
  \expandafter\gdef\csname gl#1\endcsname<##1>{#2}}
```

/ Here are a few examples of `\makeglshortcut` in action. To make the slash into a newline character, like we did in our example, the user-friendly way to do it is this:

```
\makeglshortcut/{\par\ifx#1\empty\else\vskip{#1}\fi}
```

[ ] We declare that square brackets in the input will give square brackets in the output — but with nice spacing, and an optional argument set as a subscript.

```
\makeglshortcut[{[$_\textrm{\footnotesize#1}$\thinspace}
\makeglshortcut]{\unskip\thinspace]$_\textrm{\footnotesize#1}$\glspace}
```

This is where it gets a little hairy. We also want to be able to set up shortcut characters that turn into the left- and right-hand sides of a *delimited macro*. For instance, suppose we want to be able to use asterisks as shortcuts, such that everything in between two asterisks is put into a `\framebox`. Well, step one is as easy as it was before.

```
\makespecial{*}
```

And if the asterisk were a normal character, step two would also be reasonably easy.

```
\def\gl*<#1>#2\gl*<#3>{\framebox{#2}}
```

makeglsurround But as before, because the asterisk is not a normal character, we need to invoke `\csname`. And this time, it's ugly:

```
\long\def\makeglsurround#1#2#3{%
  \xdef\dvglspecials{\dvglspecials#1#2}%
  \ta=\expandafter{\csname gl#1\endcsname}%
  \tb=\expandafter{\csname gl#2\endcsname}%
  \expandafter\expandafter\expandafter\gdef%
     \expandafter\expandafter\the\ta%
     \expandafter<\expandafter##\expandafter1\expandafter>%
     \expandafter##\expandafter2\the\tb<##3>{#3}}
```

* * Luckily, the user is protected from that ugliness. Here's a slightly more complicated version of that `\framebox`-making macro, with elegant spacing and an optional argument set as a label on top of the frame. Note the use of `\noglstrut` here to prevent extra space from ending up *inside* the frame.

```
\makeglsurround ** {%
  \rlap{\raisebox{1.5em}{\footnotesize\sf#3}}%
  \fbox{\noglstrut#2\unskip}\glspace}
```