The `lingmacros` package
Version 2.1

Andrew McKenzie
andrew.mckenzie@ku.edu
http://people.ku.edu/~a326m085

October 1, 2016

Contents

1 About `lingmacros`

The `lingmacros` package is designed to allow easier use of formal symbols used in formal linguistics, especially in formal linguistics. The set arose from the macros that I have been using over the years for papers and class handouts. Suggestions and additions are welcome.

To call the package, type \usepackage{lingmacros} in the preamble of your document. The package has the following options:

`text-semantics`	typesets formal semantic expressions in upright fashion rather than italicized or math fashion
`shortspace`	removes space before, after, and between examples in `exe` environment (see sect. 2.2)
`leftflush`	pulls examples to the left edge in `exe` environment (see sect. 2.2)
`abstract`	combines `shortspace` and `leftflush`; useful for abstracts (see sect. 2.2)

The `lingmacros` package calls for the following packages: `stmaryrd`, `ulem`, `amssymb`, `upgreek`, `gb4e`, `relsize`, and `pbox`. Also included are call commands for phonetic writing (`tipa`) and tree structures (`qtree`), but they are commented out to prevent option clashes (look for %\RequirePackage{qtree}, etc., if you want to turn them on for your version).

Using `lingmacros` is pretty straightforward, since it is just a series of macros. The macros are organized by module of the grammar.

2 General linguistics

2.1 Ordinary macros

The following macros are used pretty generally throughout the subfields.

command	purpose	in source file	in print
\nl	null symbol	\nl	∅
\m{...}	small caps for morpheme gloss	\m{acc.pl}	ACC.PL
\mc{...}	small caps for morpheme gloss	\mc{acc.pl}	ACC.PL
\mb	wider hyphen for morpheme breaks (can be changed globally)	ant\mb s	ant−s
\alert{...}	highlights parts of examples	work\alert{horse}	work**horse**
\term{...}	highlights terminology being introduced	\term{causative}	**causative**
\ix{...}	subscript index (upright text with `text-semantics` option)	Bill\ix{j}, x\ix{cat}	Bill_{j}, x_{cat}
\ux{...}	subscript index with upright text (no matter what)	Bill\ux{j}, x\ux{cat}	Bill_{j}, x_{cat}
\superx{...}	superscript index with upright text	x\superx{3}, y\superx{i}	x^{3}, y^{i}

The \alert{...} command is compatible with the `beamer` class. It puts highlighted text in boldface. This can be changed with `renewcommand`.

The \nl declaration requires the amssymb package. (\null is already used by TEX for empty boxes, so it's best not to replace it).

The \m{...} command sets grammatical morphemes (m) in SMALL CAPS. This is helpful in glosses, notably using gb4e, since it makes the source easier to read. Also, if you need to change the morphemes' typesetting globally, a simple \renewcommand of \m will suffice.

However, if you use the fontspec package for XeLATEX, beware, for it employs \m for various diacritics. So you can comment it out and use \mc{...} for morpheme caps instead. Also, \usepackage{lingmacros} must be placed in your preamble after \usepackage{fontspec}.

2.2 Macros for gb4e environment

The following declarations make the use of the gb4e more streamlined and allow for easy tweaking of example alignment.

declaration	command it replaces	purpose	name origin
\bex	\begin{exe}	begin example environment	begin exe
\fex	\end{exe}	end example environment	finish exe
\bxl	\begin{xlist}	begin example subenvironment	begin xlist
\fxl	\end{xlist}	end example subenvironment	finish xlist

These declarations also define a number of variable widths that can be used to reformat the example environments. The reformatting is done globally by setting the package options leftflush, shortspace, or abstract.

Warning: The package options do not reformat exe environments unless you use the \bex and \bxl declarations. Otherwise, one would need to adjust the gb4e package itself.

The leftflush option puts example numbers to the left, but numbers 1-9 are not all the way to the left. To force them to be, place the declaration \lessthanten in the document before the first example. This will place examples 10 and above too far left, so use the declaration \tenormore to undo this effect.

2.3 The context environment

The context environment typesets the context used to elicit or set-up an example. The typesetting can be changed globally.

> **Context:**
> *Denny arrived at the restaurant, and sat at an empty table. The moment he did so, a waiter approached and asked him:*

(1) # Would you like some more water?

\begin{context}
Denny arrived at the restaurant, and sat at an empty table. The moment he did so, a waiter approached and asked him:
\end{context}

\bex \ex[\#]{Would you like some water?} \fex

3 Phonology

OT Tableaux can be made with a number of packages, each with their own macros for symbols. To write phonological rules, however, you can use the following macros to simplify things.

command	purpose	source	in print
\underlying{...}	the input to the rule	\underlying{+back}	/+back/
\becomes	the arrow	\becomes	→
\spoken{...}	the output	\spoken{–back}	[–back]
\environ	'in the environment of' slash	\environ	/
\spot	the exact spot of the change	\spot	_
\syll	syllable subscript	[\syll]\syll	$[_\sigma$ $]_\sigma$
\fmleft	feature matrix left bracket	\fmleft	
\fmright	feature matrix right bracket	\fmright	
\fmat{...}{...}	feature matrix line	\fmat{+}{coronal} \fmat{-}{voiced}	$\begin{bmatrix} + & \text{coronal} \\ - & \text{voiced} \end{bmatrix}$

Combined, these get a source code like this, for a rule fronting a back vowel between /i/ and any consonant:

\underlying{+back} \becomes \spoken{–back} \environ i\spot C

The commands \prule and \iparule are macros combining the above macros.

About the \prule{...}{...}{...} command: The first command is the underlying form, the second the spoken form, and the third the environment. The \iparule{...}{...}{...} does the same, but puts everything in the rule in IPA. The \iparule command requires the `tipa` package, which you probably already use if you're typesetting phonology.

\prule{+back}{–back}{i\spot C}	/+back/ → [–back] / i_C
\iparule{2}{E}{i\spot *C}	/ʌ/ → [ɛ] / i_C

The \fmat{...}{...}command is for feature matrices. Use \fmleft and \fmright for each bracket, and for each line in the feature matrix, use \fmat{w}ith its two arguments. The first argument will be +/−, and the second will be the feature name.

You can put feature matrices inside a phonological rule command as well.

4 Syntax

For syntax trees, a tree package like `qtree`, `tikz-qtree`, or `parsetree` suffices. The following macros allow quick and regular typing of some common syntactic symbols, in better looking ways than are offered by ordinary distributions and packages.

command	purpose	source	in print
\head{...}	the head circle	\head{V}	V°
\xbar{...}	the bar in X-bar	\xbar{V},\xbar{Asp}	$\overline{\text{V}}$, $\overline{\text{Asp}}$
\lv	little v	\lv	v
\feat{...}	syntactic feature in trees	\feat{fem}	[fem]
\textfeat{...}	syntactic feature in text	\textfeat{fem}	[fem]
\dcopy{...}	deleted copy strike-out[1]	\dcopy{the car}	~~the car~~
\mroot{...}	morpheme root	\mroot{car}	$\sqrt{\text{car}}$
\ufeat{...}	unvalued feature in trees	\ufeat{T}	[$_u$T:_]
\unv{...}	unvalued feature	\feat{\unv{T}}	[$_u$T:_]

The \unv command should be used inside a \feat or \textfeat command, but of course doesn't have to be. If you want to use an upright ϕ symbol (ϕ), use the \upphi declaration in math mode.

Use the \featuresize{⟨size⟩} command to adjust the size of features in \feat{...}.

The \xbar{...} command places a bar over the entire head name. For a prime symbol instead, you can use use the \pri declaration.

For bracket subscript labels, you can use the \ix{...} or \ux{...} commands.

{[}\ux{TP} \head{T} [\ux{VP} \head{V} [\ux{DP} D°]]]

[$_{\text{TP}}$ T° [$_{\text{VP}}$ V° [$_{\text{DP}}$ D°]]]

5 Semantics

Formal semantics uses math mode more clearly than most areas of linguistics. You can use the \form{...} command to put anything into math mode. More recently, semanticists have been writing formulas in text, with mathematical symbols. Using the `text-semantics` option will convert these formulas from math mode to text mode. Some symbols you will want to stay in math mode. Putting them between $...$ often creates errors. Instead, the \ensuremath{...} command will protect them. For short, you can use the \f{...} command for

[1] Requires the `ulem` package

any symbol you want to remain in math mode even as \form{...} is redefined as text mode.

Along with formal expressions, the \readas{...} command is used with denotations to write formal expressions out in metalanguage.

$[\![\textit{every car}]\!] = \lambda Q \in D_{et}.\ \forall y[\ \textit{car}(y) = 1 \rightarrow Q(y) = 1\ ]$
READ: *the function from properties of entities to truth values such that for all y, if y is a car, then Q(y) = 1*

5.1 Operators and Symbols

Operators all require math mode, and putting them in math mode makes source documents hard to read. These macros simplify the writing of common operators, and make the source code more intuitive to read.

command	purpose	source	in print
\lam{*variable*}	lambda operator	\lam{x}	λx
\lamd{*var.*}{*type*}	lambda operator with domain D_{type}	\lam{P}{s,t}	$\lambda P \in D_{s,t}$
\all{*var.*}	universal quantifier	\all{x}	$\forall x$
\some{*var.*}	existential quantifier	\some{x}	$\exists x$
\no{*var.*}	negative quantifier	\no{x}	$\neg\exists x$
\ddet{*var.*}	iota-operator (definite determiner)	\ddet{x}	ιx
\pri	prime symbol in text or math mode	x\pri	x'

5.2 Semantic types

The \type{...} command is used for writing semantic types. It places its argument in ordered pair brackets, in math mode. It can be used inside the arguments of another \type command to get complex types.

\type{e,t}	$\langle e, t\rangle$
\type{e,\type{s,t}}	$\langle e, \langle s, t\rangle\rangle$
\type{\type{e,t},\type{\type{e,t},t}}	$\langle\langle e, t\rangle, \langle\langle e, t\rangle, t\rangle\rangle$

Since \type places its arguments in math mode, it can be used for ordinary ordered pairs as well. For simple types, which don't require ordered pair brackets, simply place the type in math mode: e, t ⇒ e, t

The `text-semantics` option will not put types in text mode. If you really want semantic types with upright letters, use the \uptype{...} command wherever you'd use \type{...}.

5.3 Sets

The \set{...} and \varset{...} commands are used to write sets. The \set command is purely for making the source code more intuitive, since \{ is not exactly hard to type. The \varset command ('variable set') writes an abstracted set. \varset uses a vertical line for 'such that'. For the older colon notation, use the \cvarset command.

\set{a, b, c, d}	$\{ a, b, c, d \}$
\varset{x \f{\in} D}{x is happy}	$\{ x \in D \mid x \ is \ happy \}$
\cvarset{x}{x is happy}	$\{ x : x \ is \ happy \}$

5.4 Functions

Use the command \funcnot{...}{...}{...}{...}{...}, which allows quick writing of functions in an explicit functional notation (hence the name). The first argument is the variable representing the function; the second is the domain of the function, the third is the range, the fourth is the argument variable, and the fifth are the truth conditions.

\funcnot{f}{D}{R}{y}{1 iff \form{y} is a bandit}

With no options : $f : D \to R$
$\forall y \in D, f(y) = 1$ iff y is a bandit

With `text-semantics` option : f : D → R
∀y ∈ D, f(y) = 1 iff y is a bandit

Functions can be embedded in others by putting the second function in the truth-conditions of the first

```
\funcnot{f}{D\ix{e}}{D\ix{et}}{x}{%
\funcnot{g}{D\ix{e}}{D\ix{t}}{y}{1 iff \form{y} is tall}%
}%
```

$f : D_e \to D_{et}$
$\forall x \in D_e, f(x) = g : D_e \to D_t$
$\forall y \in D_e, g(y) = 1$ iff y is tall

To write a function in array format requires math mode and the array environment. This is inconvenient, so the following macros simplify this.

1. The \fleft (function left) declaration gives the left bracket.
2. The \func{*domain*}{*range*} command is used for each line of the function.
3. The \fright (function right) declaration gives the right bracket.

To write the function $\{ \langle a, 1\rangle, \langle b, 2\rangle \}$:

```
\fleft%
\func{a}{1}%
\func{b}{2}%
\fright%
```

$$\left[\begin{array}{lll} \text{a} & \rightarrow & 1 \\ \text{b} & \rightarrow & 2 \end{array}\right]$$

These macros can be used recursively.

```
\fleft%
\func{a}{% range of a
\fleft \func{c}{1}%
       \func{d}{2}%
\fright}% end of range of a
\fright%
```

$$\left[\begin{array}{lll} \text{a} & \rightarrow & \left[\begin{array}{lll} \text{c} & \rightarrow & 1 \\ \text{d} & \rightarrow & 2 \end{array}\right] \end{array}\right]$$

5.5 Scope brackets with \scopebox

The \scopebox{...} command places brackets ([ ]) around an expression to signal its scope. This command allows the use of multi-line scope brackets, to make things easier to read. If you put more than one scope box inside the largest one, you should use \innerscopebox{...} for the inside ones.

Compare the following formulas, with simple brackets, and then with \scopebox{...}.

$\forall x[\, dog(x) \,\&\, on(\text{the car})(x) \rightarrow \exists y[\, cat(y) \,\&\, \exists e[\textsc{perfective}(e) \,\&\, chase(x)(y)(e)]]]$

$$\forall \text{x} \left[\begin{array}{l} \text{dog(x) \& on(the car)(x)} \rightarrow \\ \exists \text{y} \left[\begin{array}{l} \text{cat(y) \&} \\ \exists e[\ \textsc{perf}(\text{e}) \ \&\ \text{chase(x)(y)(e)}\ ] \end{array}\right] \end{array}\right]$$

Note: These commands force their expression to be in text mode.
Note 2: Sometimes you may need to add spaces to the outside scopebox (with ~) to make sure it's the widest.

5.6 Denotation brackets

Several commands involve double brackets for denotations (or interpretation functions). These all require the `stmaryrd` package, which is called by the `lingmacros` package.

Many commands involve assignment modifications. These modifications are already in math mode, so any use of $ in them will lead to an error. If you need to use math mode inside these, use \ensuremath{...} or \f{...}.

command	purpose	source	in print
\den{...}	denotation brackets	\den{car}	$[\![\text{car}]\!]$
\dena{...}{...}	denotation w/ assignment	\dena{car}{g}	$[\![\text{car}]\!]^{g}$
\denac{...}{...}	d.b. w/ asst., context c	\denac{car}{g}	$[\![\text{car}]\!]^{g_c}$
\denamod{..}{..}{..}	d.b. w/ modified assignment	\denamod{car}{g}{x\to 1}	$[\![\text{car}]\!]^{g^{x\to 1}}$
\denacmod{..}{..}{..}	d.b. w/ modified asst, context c	\denacmod{car}{g}{x\to 1}	$[\![\text{car}]\!]^{g_c^{x\to 1}}$

There is also a series of commands which are identical to those of the \den family, except that the text is already in the object language font (\ol). These commands declutter the source code. With \ol set to \itshape:

command	purpose	source	in print
\denol{...}	denotation brackets	\denol{car}	$[\![\mathit{car}]\!]$
\denola{...}{...}	denotation w/ assignment	\denola{car}{g}	$[\![\mathit{car}]\!]^{g}$
\denolac{...}{...}	d.b. w/ asst., context c	\denolac{car}{g}	$[\![\mathit{car}]\!]^{g_c}$
\denolamod{..}{..}{..}	d.b. w/ modified assignment	\denolamod{car}{g}{x\to 1}	$[\![\mathit{car}]\!]^{g^{x\to 1}}$
\denolacmod{..}{..}{..}	d.b. w/ modified asst, context c	\denolacmod{car}{g}{x\to 1}	$[\![\mathit{car}]\!]^{g_c^{x\to 1}}$