# The `semantic` package*

Peter Møller Neergaard†
Arne John Glenstrup‡

January 16, 2004

**Abstract**

The aim of this package is to help people doing programming languages using LaTeX. The package provides commands that facilitates the use of the notation of semantics and compilation in your documents. It provides an easy way to define new ligatures, *eg* making `=>` a short hand for `\RightArrow`. It fascilitates the drawing of inference rules and allows you to draw T-diagrams in the `picture` environment. It supports writing extracts of computer languages in a uniform way. It comes with a predefined set of shorthand suiting most people.

## Contents



---

`semantic` is a LaTeX 2ε package facilitating the writing of notation of programming languages and compilation. To use it, the file `semantic.sty` should be placed so that LaTeX can find it.

The `semantic` package consists of several parts, which can be used independently. The different parts are

**Ligatures** providing an easy way to de-

*This file has version 2.0β and is dated 2000/08/03.

†`turtle@bu.dk`, `http://cs-people.bu.edu/turtle`
‡`panic@diku.dk`, `http://www.diku.dk/~panic`

fine ligatures for often used symbols like $\Rightarrow$ and $\vdash$.

**Inference Rules** fascilitating the presentation of inference rules and derivations using inference rules.

**T-diagrams** providing T-diagrams as an extension the `picture` environment.

**Reserved word**[1] fascilitating getting a uniform appearance of language constructions.

**Short hands** for often used symbols.

In the following we describe the use of the various parts of `semantic` and the installation. We also give a short introduction to the two files `semantic.dtx` and `semantic.ins`.

This package is—like most other computer-programs—provided with several bugs, insuffiencies and inconsistencies. They should be regarded as features of the package. To increase the excitement of using the package these features appear in unpredictable places. If they however get too annoying and seriously reduce your satisfaction with `semantic`, please notify us. You could also drop us a note if you would like to be informed when `semantic` is updated.

# 1 Loading

There is two ways of loading the `semantic` package. You can either load it with all the parts, or to save time and space, you can load, only the parts you will use.

In the first case you just include

```
\usepackage{semantic}
```

in your document preamble.

In the other case you include

```
\usepackage[⟨parts⟩]{semantic}
```

in your document preamble. ⟨*parts*⟩ is a comma separated list of the parts you wants to include. The possibilities are: `ligature`, `inference`, `tdiagram`, `reserved`, and `shorthand`. The different parts are described in detail below.

# 2 Math Ligatures

## 2.1 Defining New Math Ligatures

`\mathlig` When the package is loaded, you can define new ligatures for use in the math environments by using the `\mathlig{`⟨*character sequence*⟩`}{`⟨*ligature commands*⟩`}` command. ⟨*character sequence*⟩ is a sequence of characters[2] that must be entered in the source file to achieve the effect of the ⟨*ligature command*⟩. If for example you write '`\mathlig{->}{\rightarrow}`', subsequently typing '`$->$`' will produce →.

## 2.2 Turning Math Ligatures On and Off

`\mathligson` By default, math ligatures are in effect when the mathligs package is loaded,

`\mathligsoff`

[2]There are some restrictions on the characters you can use. This should be described here but isn't; basically you should stick to using the characters ` ' " ~ ! ? @ * ( ) [ ] < > - + = | : ; . , / 0...9`, and certainly this should suffice for any sane person.

but this can be turned off and on by using the commands `\mathligsoff` and `\mathligson`. Thus, typing '`$->$ \mathligsoff $->$ \mathligson $->$`' will produce $\rightarrow \leftarrow - > < - \rightarrow \leftarrow$.

### 2.3 Protecting Fragile Math Commands

`\mathligprotect` Unfortunately, some macros used in math mode will break when using mathligs, so they need to be turned into protected macros with the declaration `\mathligprotect{`⟨*macro*⟩`}`. *NOTE:* This declaration only needs to be issued once, best in the preamble.

## 3 Inference Rules

`\inference` `\inference*` Inference rules like

$$\text{It(1)}: \frac{\rho \vdash E \Rightarrow \textsc{False}}{\rho \vdash \texttt{while}\ E\ \texttt{do}\ s \Rightarrow \rho} \qquad \text{It(2)}: \frac{\rho \vdash E \Rightarrow \textsc{True} \quad \rho \vdash s \Rightarrow \rho' \quad \rho' \vdash \texttt{while}\ E\ \texttt{do}\ s \Rightarrow \rho''}{\rho \vdash \texttt{while}\ E\ \texttt{do}\ s \Rightarrow \rho''}$$

and

$$\rightarrow^*{}_1 \frac{p, \mathrm{M} \rightarrow^* p', \mathrm{M}' \quad p', \mathrm{M}' \rightarrow p'', \mathrm{M}''}{p, \mathrm{M} \rightarrow^* p'', \mathrm{M}''} \qquad \rightarrow^*{}_2 \frac{}{p, \mathrm{M} \rightarrow^* p, \mathrm{M}}$$

are easily set using `\inference` and `\inference*`. The syntax is

`\inference[`⟨*name*⟩`]{`⟨*line*$_1$⟩ `\\` ... `\\` ⟨*line*$_n$⟩`}{`⟨*conclusion*⟩`}`

and

`\inference*[`⟨*name*⟩`]{`⟨*line*$_1$⟩ `\\` ... `\\` ⟨*line*$_n$⟩`}{`⟨*conclusion*⟩`}`

where $n \geq 0$ so that you can also type axioms. When using `\inference` the bar will be as wide as the conclusion and the premise, whichever is widest; while `\inference*` only will make the bar as wide as the conclusion (It(2) above). The optional names are typeset on the side of the inferences that they appear.

Each line consists of premises seperated by `&`:

⟨*premise*$_1$⟩`&`...`&`⟨*premise*$_m$⟩

Note that $m$ can also be zero, which is used when typing axioms. Each premise and the conclusion are by default set in math mode (*see* however 4).

The rules are set so that the line flushes with the center of small letters in the surrounding text. In this way, secondary conditions or names (like the first example above) can be written in the surrounding text. One may also set the rules in a table as shown below:

$$\text{Transitive (1)}: \quad \frac{p, \mathrm{M} \rightarrow^* p', \mathrm{M}' \quad p', \mathrm{M}' \rightarrow p'', \mathrm{M}''}{p, \mathrm{M} \rightarrow^* p'', \mathrm{M}''}$$

$$\text{Transitive (2)}: \quad \frac{}{p, \mathrm{M} \rightarrow^* p, \mathrm{M}}$$

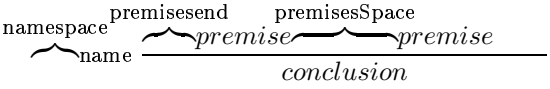
An inference rule can be nested within another rule without problems, like in:

$$\rightarrow^*{}_1 \frac{\rightarrow^*{}_1 \dfrac{\rightarrow^*{}_2 \dfrac{}{p, \mathrm{M} \rightarrow^* p, \mathrm{M}} \quad p, \mathrm{M} \rightarrow^* p', \mathrm{M}'}{p, \mathrm{M} \rightarrow^* p', \mathrm{M}'} \quad p', \mathrm{M}' \rightarrow p'', \mathrm{M}''}{p, \mathrm{M} \rightarrow^* p'', \mathrm{M}''}$$

## 3.1 Controlling the Appearance

`\setpremisesend` `\setpremisesspace` `\setnamespace`

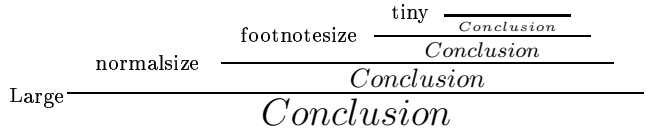
The appearance of the inferences rules can be partly controlled by the following lengths:




The lengths are changed using the three commands `\setnamespace{`⟨*length*⟩`}`, `\setpremisesend{`⟨*length*⟩`}` og `\setpremisesspace{`⟨*length*⟩`}`. ⟨*length*⟩ can be given in both absolute units like `pt` and `cm` and in relative units like `em` and `ex`. The default values are: $1\frac{1}{2}$`em` for `premisesspace`, $\frac{3}{4}$`em` for `premisesend` and $\frac{1}{2}$`em` for `namespace`. Note that the lengths *cannot* be altered using the ordinary LaTeX-commands `\setlength` and `\addtolength`.

Besides that, the appearance of inference rules is like fractions in math: Among other things the premises will *normally* be at same height above the baseline and there is a minimum distance from the line to the bottom of the premises.

Fetching the font information from the math font and the evaluation (in case they are defined in relative units) of the lengths mentioned above is done just before the individual rule is set. This is demonstrated by the following construction (which admittedly is not very useful):




Note that from top to bottom, the leaves get bigger and the names get further from the line below.

## 3.2 Formatting the Entries

`\predicate`

To set up a single predicate (a premise or conclusion) the single-argument command `\predicate` is used. This allows a finer control of the formatting. As an example, all premises and conclusions can be set in mathematics mode by the command:

```
\renewcommand{\prediate}[1]{$ #1 $}
```

`semantic` uses `\predicate` on a premise only when the premise does not contain a nested `\inference`.[3] So even if the declaration above has been given, `\inference` is *never* be executed in math mode. Neither is it used on the premises if you write:

```
\inference{\inference...}{...}
```

`\predicatebegin` `\predicateend`

The default definition of `\predicate` is `\predicatebegin #1\predicateend`, where `\predicatebegin` and `\predicateend` are defined to '`$`'. In this way the premises and conlusions are set in math

The motivation for introducing `\predicatebegin` and `\predicateend` was, however, to use TeX's pattern matching on macro arguments to do even more sophisticated formatting by redefining `\predicatebegin`. If for example, *every expression* is to be evaluated in an *environment* giving a *value*, and you would like to set *all* the *environment's values* in mathematics and the *expressions* in `typewriter`-font, then this could be facilitated by the definition:

[3] What `semantic` precisely does is to append the tokens `\inference \end` to the code of a premise, when it has isolated it. `semantic` then uses TeX's pattern matching to search this new list of tokens for an appearance of the token `\inference`. When this is found the following token is examened, and if it is `\end`, `semantic` concludes that the premise does not contain a nested inference rule

```
\def\predicatebegin#1|-#2=>#3#4\predicateend{%
  $#1 \vdash$\texttt{#2}$\stackrel{#3}{\Rightarrow}_S #4$}
```

Then the inference (borrowed from M. Hennessy, *The Semantics of Programming Languages*)

$$\text{TlR}\frac{D \vdash s \stackrel{v}{\Rightarrow}_S s' \qquad D \vdash s \stackrel{v'}{\Rightarrow}_S s''}{D \vdash \texttt{Tl}(s) \stackrel{v'}{\Rightarrow}_S s''}$$

can be accomplished by

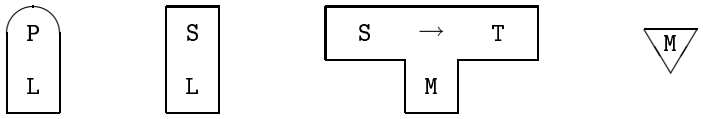
```
\inference[TlR]{D |- $s$ =>{v} s' & D |- $s$ =>{v'} s''}
  {D |- Tl($s$) =>{v'} s''}
```

Please note that the `ligatures` option *has not* been used above.

# 4 T-diagrams

`\compiler` `\interpreter` `\program` `\machine`

To draw T-diagrams describing the result of using one or more compilers, interpreters etc., `semantic` has commands for the diagram:




These commands should only be used in a `picture` environement and are

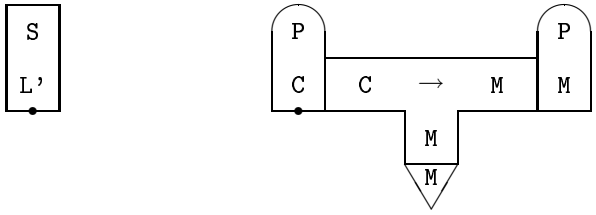
`\program{`⟨*program*⟩`,`⟨*implementation language*⟩`}`
`\interpreter{`⟨*source*⟩`,`⟨*implemenation language*⟩`}`
`\compiler{`⟨*source*⟩`,`⟨*machine*⟩`,`⟨*target*⟩`}`
`\machine{`⟨*machine*⟩`}`

The arguments can be a either a string describing the language (please do not begin the string with a macro name), or one of the four commands. However, combinations taht make no sense—like implementing an interpreter on a program—are excluded, yielding an error message like:

```
! Package semantic Error: A program cannot be at the bottom .

   See the semantic package documentation for explanation.
   Type  H <return>  for immediate help.
    ...
```

When you are use a command as an argument `semantic`, will copy the language from the nested command and automaticly place the two figures in proportion to each other. In this way, big T- diagrams can easily be drawn. The hole construction should be placed using af `\put` command, where the *reference point* is the center of the bottom of the figure corresponding to the outermost command. An example (with the reference point marked by •) will clarify some of these point. The figure



is obtained by the commands

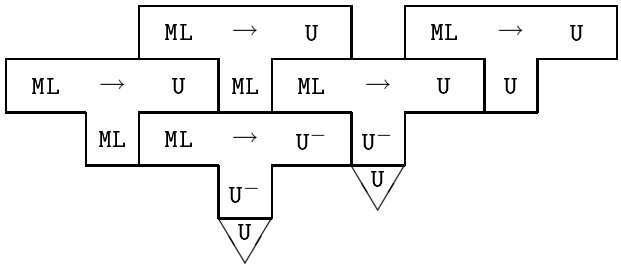
```
\begin{picture}(220,75)(0,-35)
 \put(10,0){\interpreter{S,L'}}
 \put(110,0){\program{P,\compiler{C,\machine{M},\program{P,M}}}}
\end{picture}
```

Note from the second example that when `\compiler` is used as "implementation language"-argument it is by convention attributed to the right of the figure. It is also worth mentioning that there is no strict demand on which command you should choose as the outermost, *ie* the second example could also be written (with a change of the parameters of `\put` due to the new reference point) as

```
\put(160,-20){\compiler{\program{P,C},\machine{M},\program{P,M}}}
```

starting off in the middle instead of using a "left-to-right"-approach. In fact, it is often easier to start in the middle, since this is where you get the least levels of nesting.

Even though most situations may be handled by means of nesting, it is in some rare cases adequate to use different language symbols on the two sides of the line of touch. When *eg* describing bootstrappring the poor U-code implementation can be symbolized by $\mathtt{U}^-$, indicating that the poor implementation is still executed on a U-machine. This can be done by providing the symbol-command with an optional argument immediately after the command name. Thus the bootstrapping




is typed

```
\compiler{\compiler{ML,ML,U},\machine[U$^-$]{U},\compiler{
    \compiler{ML,ML,U},\machine[U$^-$]{U},\compiler{ML,U,U}}}
```

For calculating the dimensions of the `picture`-environment, one needs the dimensions of the individual figures. In units of `\unitlength` they are the following:

compiler: 80*40
machine: 20*17.1
interpreter: 20*40
program: 20*40

# 5 Reserved Words

When describing computer languages, one often wants to typeset commands in one style, expressions in another style, and punctuation characters in yet another style, for instance

$$\textbf{let } x = e \textbf{ in } e$$

`\reservestyle` The `semantic` package supports this by allowing you to reserve a certain *style* for certain language constructs. The fundamental command is

`\reservestyle{\`*stylename*`}{`*formatting*`}`

`reservestyle` reserves `\`*stylename* as the macro to define the language constructs. The language constructs will be set using *formatting*.

The reserved macro `\`*stylename* should be given a comma separated list of words to reserve. For instance to reserve the words **let** and **in** as commands, which all are set using a bold font, you can put

```
\reservestyle{\command}{\textbf}
\command{let,in}
```

in the preamble of your document. Note that there must not any superfloues space in the comma separated list. Thus for instance `\command{let␣,␣in}` would reserve **let**␣ resp. ␣**in** instead of **let** resp. **in**! You can of course reserve several styles and reserve several words within each of the styles.

`\<···>` To refer to a *reserved word* in the text you use the command `\<`*reserved word*`>`, *eg* `\<let>`. If you have reserved several styles, `semantic` will find the style that was used to reserve *reserved word* and use the appropriate formatting commands.

The `\<···>` can be used in both plain text and in math mode. You should, however, decide in the preamble if a given style should be used in math mode or in plain text, as the formatting commands will be different.

`\set`*style* If you only want to type a reserved word a single time, it can seem tedious first to reserve the word and then refer to it once using `\<···>`. Instead you can use the command `\set`*style* that is defined for each style you reserve.

## 5.1 Bells and Whistles: Spacing in Math Mode

In many situations it seems best to use *reserved words* in math mode—after all you get typesetting of expressions for free. The drawback is that it becomes more difficult to get the space correct. One can of course allways insert the space by hand, *eg* `$\<let>\; x=e \;\<in>\; e'$`, However, this soon becomes tedious and `semantic` have several ways to try to work around this.

The first option is to provide `\reservestyle` with an optional spacing command, *eg* `\mathinner`. For instance

```
\reservestyle[\mathinner]{\command}{\mathbf}
```

will force all *commands* to be typeset with spacing of math inner symbols.

You can also provide an optional space command to each reservation of words. For instance

```
\command[\mathrel]{in}
```

will make **in** use the spacing of the relational symbols. The space command is applied to all the words in the reservation. Thus if you would like **in** and **let** to have different space commands, you must specify them in two different `\command`.

The drawback of using the math spacing is that in the rare cases where you use the reserved words in super- or subscripts, most of the spacing will disappear. This can be avoided by defining the replacement text to be the word plus a space, *eg* `\;in\;`. For this end a reservation of a word can be followed by an explicit replacement text in brackets, *eg*

```
\command{let[let\;], in[\;in\;]}
```

The formatting of `\command` (with the setting above: `\mathbf`) will still be used so it is only necessary to provide the replacement text. Note that each word in the reservation can have its own optional replacement text.

The drawback of this method is, that the you also get the space, if you use the reserved word "out of context", for instance refering to the **in** -token! In these cases you can cancel the space by hand using `\!`.

This option is also usefull, if you want to typeset the same word in two different styles. If you for instance sometimes want 'let' to be typeset as a command and sometimes as data, you can define

```
\command{let}
\data{Let[let]}
```

Then `\<let>` will typeset the word 'let' as a command, while `\<Let>` will typeset it as data. Note that in both cases the word appears in lower case.

Unfortunately there is no way to get the right spacing everytime, so you will have to choose which of the two methods serves you the best.

# 6 Often Needed Short Hands

Within the field of semantics there are a tradition for using some special. symbols. These are provided as default as short hand in the `semantic` package. Most of the following symbols are defined as ligatures, and hence the `ligature` option is always implied when the `shorthand` option is provided.

## 6.1 The Meaning of: ⟦ and ⟧

⟦
⟧

The symbols for denoting the meaning of an expression, ⟦ and ⟧ are provided as short hands in math with the ligatures `|[` and `|]`.

## 6.2 Often Needed Symbols

The following ligatures are defined for often needed symbols

| | | | |
|---|---|---|---|
| $\vdash$ | `\|-` | $\models$ | `\|=` |
| $\longleftrightarrow$ | `<->` | $\Longleftrightarrow$ | `<=>` |
| $\rightarrow$ | `->` | $\longrightarrow$ | `-->` |
| $\Rightarrow$ | `=>` | $\Longrightarrow$ | `==>` |
| $\leftarrow$ | `<-` | $\longleftarrow$ | `<--` |
| $\Leftarrow$ | `<=` | $\Longleftarrow$ | `<==` |

All the single directed arrows also comes in a starred and plussed form, *eg* `*<=` gives ${}^*\Leftarrow$ and `-->+` gives $\longrightarrow^+$.

`\eval` `\comp` To support writing denotational, semantics the commands `\comp` and `\eval` are provided to describe the evaluation of programs respectively expressions. They have the same syntax: `\comp{`⟨*command*⟩`}{`⟨*environment*⟩`}`, which yields $\mathcal{C}[\![\langle \textit{command} \rangle]\!]\langle \textit{environment} \rangle$. If you need to describe more than one kind of evaluations, e.g. both $\mathcal{E}$ and $\mathcal{E}^*$, you can provide an optional argument immediately after `\comp` or `\eval`, respectively. As an example a denotational rule for a sequencing two commands

$$\mathcal{C}[\![\texttt{C1 ; C2}]\!]\texttt{d} = \texttt{d}' \quad \texttt{if } \mathcal{C}[\![\texttt{C1}]\!]\texttt{d} = \texttt{d}'' \texttt{ and } \mathcal{C}[\![\texttt{C2}]\!]\texttt{d}'' = \texttt{d}'$$

can be typed

```
\[
  \comp{C1 ; C2}{d} = \mathtt{d'} \quad
   \texttt{if $\comp{C1}{d} = \mathtt{d''}$ and
              $\comp{C2}{d''} = \mathtt{d'}$}
\]
```

\evalsymbol
\compsymbol

As shown above, you can get the evaluation symbol in itself. This is done by `\compsymbol` or `\evalsymbol`, respectively. These commands can also be supplied with an optional argument, e.g. `\evalsymbol[*]` to get $\mathcal{E}^*$.

\exe

The result of executing a program on a machine with som data can be described using `\exe`, which has the syntax `\exe{`⟨*program*⟩`}[`⟨*machine*⟩`]{`⟨*data*⟩`}`, where ⟨*machine*⟩ is optional. The third Futumara projection $\mathtt{cogen} = [\![\mathtt{spec}]\!](\mathtt{spec.spec})$ can be written `$\mathtt{cogen} = \exe{spec}{spec.spec}$`. As an alternative, you can also give the machine `L` explicit:

```
$\mathtt{cogen} = \exe{spec}[L]{spec.spec}$
```

This will result in: $\mathtt{cogen} = [\![\mathtt{spec}]\!]^{\mathtt{L}}(\mathtt{spec.spec})$

# 7 Some Notes about the Files

`semantic` is distributed in two files, `semantic.dtx` and `semantic.ins`. Of these two files, `semantic.dtx` is the most important, as it contains all the essentials—users guide, code and documentation of the code. `semantic.ins` is used only to guide docstrip in generating `semantic.sty` from `semantic.dtx`.

To get ⟦ and ⟧, used in `\comp`, `\eval` and `\exe` `semantic`, tries to load the package bbold written by A. Jeffrey. If this is not installed on your system, the symbols are simulated by drawing together two sharps. However, we recommend that you get bbold from your nearest CTAN-archive.

In addition to the users guide, you can also get the fully documented code. You need this, however, if you want to see how the macros are implemented the macros or if you want to change some part of the package. You should start by editing `semantic.dtx`, change the two lines (near line 3320)

```
\OnlyDescription    % Make to a comment to get the documentation
\DisableCrossrefs   % Remove comment if the index is ready or if
```

into a comment, and remove `%` from the begining of the next two lines:

```
%\CodelineIndex      % Make a index of the command usage
%\RecordChanges      % Make the changes history
```

Then you should run LaTeX twice on the edited file to get a correct table of contents. Then you generate the index and change history, using `makeindex`:

```
makeindex -s gind.ist semantic
makeindex -s gglo.ist -o semantic.gls semantic.glo
```

After another run of LaTeX, then the documentation is ready for printing.

`semantic.dtx` is designed to be used with doc and docstrip. It therefore contains the package code but most of it is a lot of comments. These comments constitute the users guide and the documentation of the code. When TeX is runned on `semantic.ins`, doc is used to remove all these comments, and the

result is written in the package file `semantic.sty`. When you make the users guide by running LaTeX on `semantic.dtx`, LaTeX first skips about 815 lines of comments (containing the source of the users guide) before it gets to the lines normally making up the package file. Then follows nearly 2400 lines of intermixed comments and code defining the commands in the package. At last, LaTeX arrives at a line `\documentclass{ltxdoc}` (ltxdoc is LaTeX $2_\varepsilon$'s standard class for documenting). After a couple of packages loadings and some command definition, it reaches `\begin{document}` and `\docinput{semantic.dtx}`. This makes LaTeX read over `semantic.dtx` once more but this time skipping the %-chars in the first column.