SemanTEX: Object-oriented mathematics (v0.1α)

Sebastian Ørsted (sorsted@gmail.com)

May 30, 2020

The SemanTEX package for LATEX delivers a more semantic, systematized way of writing mathematics, compared to the ordinary math syntax. The system is object-oriented and uses keyval syntax, and everything is highly customizable. At the same time, care has been taken to make it intuitive, natural, practical, and with an easy-to-use and lightweight syntax. **Note: SemanTEX is still in its alpha stage and cannot be considered stable at this point. You are more than welcome to report bugs and come with suggestions!**

Traditional math notation in TEX is not particularly semantic – you usually type the raw *notation* rather than the underlying *meaning* of your math. Take, for instance, the following equations from algebraic geometry:

$$\begin{aligned}(f^{-1}\mathcal{F})_p &= \mathcal{F}_{f(p)},\\ \mathcal{O}_U &= \mathcal{O}_X|_U,\\ \mathcal{H}om(\mathcal{F},\mathcal{G})(U) &= \mathrm{Hom}_{\mathcal{O}_U}(\mathcal{F}|_U,\mathcal{G}|_U),\\ H^0(U;\mathcal{O}_X) &= \mathcal{O}_X(U).\end{aligned}$$

Here, $\mathcal{F}$ and $\mathcal{G}$ are sheaves on some scheme X, $\mathcal{O}_X$ is the structure sheaf, and $U \subset X$ an open subset. In traditional TEX, you would probably define a collection of commands `\sheafF`, `\sheafG`, `\sheafO`, and `\sheafHom` for $\mathcal{F}$, $\mathcal{G}$, $\mathcal{O}$, and $\mathcal{H}om$ and then proceed something like

```
(f^{-1}\sheafF)_{p}=\sheafF_{f(p)},
\sheafO_{U} = \sheafO_{X}|_{U},
\sheafHom( \sheafF , \sheafG)(U)
  = \Hom_{\sheafO_{X} ( \sheafF|_{U} , \sheafG|_{V} ),
  H^{0}(U;\sheafF) = \sheafF(U).
```

For more than 90 % of all mathematicians, this solution will be completely satisfactory; it prints what it is supposed to, and that's that. If this is how you feel, there is absolutely no reason for you to continue reading. This package is for the remaining less than 10 % who would prefer to write something like the following instead:

```
\vf[inverseimage]{ \sheafF }[spar,stalk=\vp]
  = \sheafF[ stalk=\vf{\vp} ] ,
\sheafO[\vU] = \sheafO[ \vX, res=\vU ],
\sheafHom{ \sheafF , \sheafG }{\vU}
  = \Hom[\sheafO[\vU]]{ \sheafF[res=\vU], \sheafG[res=\vU] },
\co{0}{ \vU, coef=\sheafO[\vX] } = \sheafO[\vX]{\vU}.
```

A lot of comments are in order. The whole syntax will be explained in later chapters, but let us take a moment to look at these examples and understand the logic. First of all, what is up with all the `v`'s in the command names `\vf`, `\vX`, `\vU`? The `v` stands for "variable", and it is the prefix I recommend using for all standard variables. So for all letters in the alphabet, uppercase and lowercase, as well as the Greek ones, there will be a command: `\va`, `\vA`, `\vb`, `\vB`, etc. It is not always necessary to use them; for instance, in the above example, both `\vX` and `\vU` could have been replaced by simply `X`, `U` without changing anything. This is because we did not apply any arguments to these symbols. However, for the sake of consistency, I prefer to switch completely to using commands instead of writing the symbols directly. How *you* use the system is completely up to you.

In SemanTEX, all entries are being built up from the inside and out. The basic syntax layout for most SemanTEX commands is

```
\command[options]{argument}
```

Let us try focusing on the first example from above:

```
$\vf[inverseimage]{\sheafF}[
spar,stalk=\vp]$
```

$$(f^{-1}\mathcal{F})_p$$

You always start with a central piece: a *symbol*. In the case of `\vf`, the symbol is f. After the symbol follows the options we apply to it, written in brackets `[...]`. In this case, we the option `inverseimage`. This tells SemanTEX that we want the inverse image functor f^{-1}, so it adds a superscript `-1` to the symbol. After this, we apply the function `\vf[inverseimage]` to something, namely the sheaf $\mathcal{F}$. This is done by enclosing them in braces `{...}`.[1]

Next, we want to take the stalk of this sheaf at the point p. If we simply wrote `\vf[inverseimage]{\sheafF}[stalk=\vp]`, we would get $f^{-1}\mathcal{F}_p$, which looks confusing. So we want to enclose $f^{-1}\mathcal{F}$ in parentheses before taking the stalk. This is done with the key `spar` (an abbreviation for "symbol parentheses"). This key takes whatever has been typed so far, symbol and indices, and adds parentheses around it (of course, type and size is adjustable). This `spar` is a key you fill find yourself using a lot.

1 Getting started

To get started using SemanTEX, load down the package with

```
\usepackage{semantex}
```

The SemanTEX system is object-oriented; all entities are objects of some class. When you load the package, there is only one class by default, which is simply called `semantexvariable`. You should think of this as a low-level class, the parent of all other classes. Therefore, I highly advice against using it directly or modifying it. Instead, we create a new, more high-level variable class. To make notations brief, we call it `var`. We could write `\newvariableclass{var}`, but we choose to pass some options to it in `[...]`:

```
\newvariableclass{var}[output=var]
```

[1] You should be aware that this argument in braces `{...}` is *optional*. You can simply write `\vf[inverseimage]` if you want, and it will produce f^{-1}.

This `output=var` option will be explained better below. Roughly speaking, it tells SemanTEX that everything a variable *outputs* will also be a variable. For instance, if the function `\vf` (i.e. f) is of class `var`, then `\vf{\vx}` (i.e. $f(x)$) will also of class `var`.

Now we have a class, but we do not have any objects. When we create the class `var`, the system automatically defines the command `\newvar` that creates a new object of class `var`. The syntax is

```
\newvar\⟨variable name⟩{⟨variable symbol⟩}[options]
```

For instance, we may write

```
\newvar\vf{f}
\newvar\vx{x}
```

to get two variables `\vf` and `\vx` with symbols f resp. x. Let us perform a stupid test to see if the variables work:

```
$\vf$, $\vx$
```

f, x

Th general syntax of a variable-type command is

```
\command[options]{argument}
```

Both `options` and `argument` are optional arguments (they can be left out if you do not need them). The `options` should consist of a list of options separated by commas, using keyval syntax. On the other hand, `argument` is the actual argument of the function. By design, SemanTEX does not distinguish between variables and functions, so all variables can take arguments. This is a design choice to make the system easier to use; after all, it is fairly common in mathematics that something is first a variable and then a moment later takes an argument. So we may write:

```
$\vf{1}$, $\vf{\vx}$
```

$f(1), f(x)$

So far, we do not have very many options to write in the `options` position, since we have not added any keys yet. However, we do have access to the most important of all options: the *index*. There is a simple shortcut for writing an index: You simply write the index itself in the options tag:

```
$\vf[1]$, $\vf[\vf]$,
$\vf[1,2,\vf]{2}$
```

$f_1, f_f, f_{1,2,f}(2)$

As long as what you write in the options tag is not recognized as a predefined key, it will be printed as the index. Other than that, there are two important predefined keys: `upper` and `lower` which simply add something to the upper and lower index:

```
$\vf[upper=2]$,
$\vf[lower=3]$
```

f^2, f_3

We are soon going to need more variables than just f and x. In fact, I advise you to create a variable for each letter in the Latin and Greek alphabets, both uppercase and lowercase. This is pretty time-consuming, so I did it for you already:

```
\newvar\va{a}
\newvar\vb{b}
\newvar\vc{c}
\newvar\vd{d}
\newvar\ve{e}
\newvar\vf{f}
\newvar\vg{g}
\newvar\vh{h}
\newvar\vi{i}
\newvar\vj{j}
\newvar\vk{k}
\newvar\vl{l}
\newvar\vm{m}
\newvar\vn{n}
\newvar\vo{o}
\newvar\vp{p}
\newvar\vq{q}
\newvar\vr{r}
\newvar\vs{s}
\newvar\vt{t}
\newvar\vu{u}
\newvar\vv{v}
\newvar\vw{w}
\newvar\vx{x}
\newvar\vy{y}
\newvar\vz{z}

\newvar\vA{A}
\newvar\vB{B}
\newvar\vC{C}
\newvar\vD{D}
\newvar\vE{E}
\newvar\vF{F}
\newvar\vG{G}
\newvar\vH{H}
\newvar\vI{I}
\newvar\vJ{J}
\newvar\vK{K}
\newvar\vL{L}
\newvar\vM{M}
\newvar\vN{N}
\newvar\vO{O}
\newvar\vP{P}
\newvar\vQ{Q}
\newvar\vR{R}
\newvar\vS{S}
\newvar\vT{T}
\newvar\vU{U}
\newvar\vV{V}
\newvar\vW{W}
\newvar\vX{X}
\newvar\vY{Y}
\newvar\vZ{Z}
```

```
\newvar\valpha{\alpha}
\newvar\vvaralpha{\varalpha}
\newvar\vbeta{\beta}
\newvar\vgamma{\gamma}
\newvar\vdelta{\delta}
\newvar\vepsilon{\epsilon}
\newvar\vvarepsilon{\varepsilon}
\newvar\vzeta{\zeta}
\newvar\veta{\eta}
\newvar\vtheta{\theta}
\newvar\viota{\iota}
\newvar\vkappa{\kappa}
\newvar\vlambda{\lambda}
\newvar\vmu{\mu}
\newvar\vnu{\nu}
\newvar\vxi{\xi}
\newvar\vpi{\pi}
\newvar\vvarpi{\varpi}
\newvar\vrho{\rho}
\newvar\vsigma{\sigma}
\newvar\vtau{\tau}
\newvar\vupsilon{\upsilon}
\newvar\vphi{\phi}
\newvar\vvarphi{\varphi}
\newvar\vchi{\chi}
\newvar\vpsi{\psi}
\newvar\vomega{\omega}

\newvar\vGamma{\Gamma}
\newvar\vDelta{\Delta}
\newvar\vTheta{\Theta}
\newvar\vLambda{\Lambda}
\newvar\vXi{\Xi}
\newvar\vPi{\Pi}
\newvar\vSigma{\Sigma}
\newvar\vUpsilon{\Upsilon}
\newvar\vPhi{\Phi}
\newvar\vPsi{\Psi}
\newvar\vOmega{\Omega}
```

Just like `\vf`, these can all be regarded as functions, so `\va{\vb}` produces $a(b)$. Importantly, **parentheses can be scaled**. To make parentheses bigger, use the following keys:

```
$\vf{\vx}$,
$\vf[par=\big]{\vx}$,
$\vf[par=\Big]{\vx}$,
$\vf[par=\bigg]{\vx}$,
$\vf[par=\Bigg]{\vx}$,
$\vf[par=auto]{\frac{1}{2}}$
```

$$f(x),\ f\big(x\big),\ f\Big(x\Big),\ f\bigg(x\bigg),\ f\Bigg(x\Bigg),\ f\left(\tfrac{1}{2}\right)$$

Using `par=auto` corresponds to using `\left...\right`. Just as for ordinary math, I advice you to use manual scaling rather than automatic scaling, as TEX has a tendency to scale things wrong. If you do not want parentheses at all, you can pass the key `nopar`:

```
$\vf[nopar]{\vx}$
```

fx

Primes are added via the key `prime` or the keys `'`, `''` and `'''`:

```
$\vf['] = \vf[prime]$,
$\vf[''] = \vf[prime,prime]$
$\vf['''] = \vf[prime,prime,
prime]$
```

$f' = f'$, $f'' = f''$ $f''' = f'''$

So far, so good, but our variables cannot really do anything yet. For this, we need to assign *keys* to them. The more pieces of math notation you need, the more keys you will have to define. Keys are being added via two different keys:

`novaluekeys` and `valuekeys`.

In short, `novaluekeys` is for keys that do *not* take a value (i.e. keys using the syntax `\command[key]`), and `valuekeys` is for keys that *do* take a value (i.e. keys using the syntax `\command[key=value]`)). We explain the syntax for using them in the next section where we show how to make keyval syntax for elementary calculus.

2 Example: Elementary calculus

One thing we might want to do to a variable is *invert* it. We therefore add a key `inv` that adds an upper index `-1` to the symbol. We add this key using the key `novaluekeys`:

```
\setupclass{var}{
  novaluekeys={
    {inv}{ upper={-1} },
  },
}
```

Now the key `inv` has been defined to be equivalent to `upper={-1}`. Now we can do the following:

```
$\va[inv]$, $\vf[inv]$,
$\vg[1,2,inv]$,
$\vh[\va,\vb,inv]$
```

a^{-1}, f^{-1}, $g_{1,2}^{-1}$, $h_{a,b}^{-1}$

The key `novaluekeys` is for keys that take no value, like `inv`.

Other keys might need to take a value. For defining such, we have the command `valuekeys`. For instance, suppose we want a command for deriving a function n times. For this, we add the following extra keyval key `der`:

```
\setupclass{var}{
  novaluekeys={
    {inv}{ upper={-1} },
  },
  valuekeys={
    {der}{ upper={(#1)} },
  },
}
```

The `#1` will contain whatever the user added as the value of the key. Now we can write:

```
$\vf[der=\vn]{\vx}$
```

$f^{(n)}(x)$

Maybe we also want a more elementary operation: raising a variable to some *power*. We could have called the key `power`, but this is long and cumbersome, so let us simply call it `to`:

```
\setupclass{var}{
  novaluekeys={
    {inv}{ upper={-1} },
  },
  valuekeys={
    {der}{ upper={(#1)} },
    {to}{ upper={#1} },
  },
}
```

This allows us to write

```
$\vx[to=2]$,
$\vy[1,to=2] + \vy[2,to=2]$
```

x^2, $y_1^2 + y_2^2$

In the long run, you might want to define a command `squared` or `sq` and make it equivalent to `to=2`.

Let us try doing something a bit more complicated: Adding a key for restricting a function to a smaller subset. For this, we do the following:

```
\setupclass{var}{
  novaluekeys={
    {inv}{ upper={-1} },
  },
  valuekeys={
    {der}{ upper={(#1)} },
    {to}{ upper={#1} },
    {res}{ return,symbolputright={|}, lower={#1} },
  },
}
```

This adds a horizonal line "|" to the right of the symbol followed by a lower index containing whatever you passed to the key (contained in the command `#1`). (There is also an extra key, `return`, which is a bit more advanced and should be taken for granted for now. Roughly speaking, it is there to make sure that the restriction symbol is printed *after* all indices that you might have added before. More details in chapter 6.) Now we may write the following:

```
$\vf[res=\vU]{\vx}$,
$\vg[1,res=\vY]{\vy}$,
$\vh[inv,res=\vT]{\vz}$
```

$f|_U(x)$, $g_1|_Y(y)$, $h^{-1}|_T(z)$

If the reader starts playing around with the SemanTEX functions, they will discover that whenever you apply a function to something, the result becomes a new function that can take an argument itself. This behaviour is both useful and extremely necessary in order for the package to be useful in practice. For instance, you may write

```
$\vf[der=\vn]{\vx}{\vy}{\vz}
=\vg{\vu,\vv,\vw}[3]{
  \vx[1],\vx[2]}[8,1,der=2]{\
  vt}$
```

$$f^{(n)}(x)(y)(z) = g(u,v,w)_3(x_1,x_2)^{(2)}_{8,1}(t)$$

Some people prefer to be able to scale the restriction line. I rarely do that, but for that purpose, we could do the following:

```
\setupclass{var}{
  valuekeys={
    {bigres}{ return, symbolputright=\big\vert, lower={#1} },
    {Bigres}{ return, symbolputright=\Big\vert, lower={#1} },
    {biggres}{ return, symbolputright=\bigg\vert, lower={#1} },
    {Biggres}{ return, symbolputright=\Bigg\vert, lower={#1} },
    {autores}{ return, sparsize=auto, otherspar=.\vert,
    sparsize=normal, lower={#1} },
    % This auto scales the vertical bar. See the chapter on the
     spar
    % key for information about sparsize and otherspar
  },
}
```

So to sum up, we first defined a class `var` via `\newvariableclass` and then used `\setupclass` to add keys to it. In fact, we could have done it all at once by passing these options directly to `\newvariableclass`:

```
  \newvariableclass{var}[
    output=var,
    novaluekeys={
      {inv}{ upper={-1} },
    },
    valuekeys={
      {der}{ upper={(#1)} },
      {to}{ upper={#1} },
      {res}{ rightreturn, symbolputright={|},
        lower={#1} },
    },
  ]
```

As we proceed in this guide, we shall use `\setupclass` to add more and more keys to `var`. However, when you set up your own system, you may as well just add all of the keys like this when you create the class and then be done with it.

Let me add that it is possible to create subclasses of existing classes. You just write `parent=myclass` in the class declaration to tell that `myclass` is the parent class. **But a word of warning:** It is a natural idea to create different classes for different mathematical entities, each with their own keyval syntax that fits whatever class you are in; for instance, you could have one class for algebraic structures like rings and modules with keys for opposite rings and algebraic closure, and you could have another class for topological spaces with keys for closure and interior. However, as the reader can probably imagine, this becomes extremely cumbersome to work with in practice since an algebraic structure might very well also carry a topology. So at the end of the day, I

advice you to use a single superclass `var` that has all the keyval syntax and only use subclasses for further *customization*. We shall see examples of this below.

3 Example: Elementary algebra

Let us try to use the SemanTEX system to build some commands for doing algebra. As an algebraist, one of the first things you might want to do is to create polynomial rings $k[x,y,z]$. Since all variables can already be used as functions (this is a design choice we discussed earlier), all we need to do is find a way to change from using parentheses to square brackets. This can be done the following way:

```
\setupclass{var}{
  novaluekeys={
    {pol}{
      par,  % This tells semantex to use parentheses around
            % the argument in the first place, in case this
            % had been turned off
      leftpar=[,rightpar=],
    },
  },
}
```

Now we may write

```
$\vk[pol]{\vx,\vy,\vz}$
```

$k[x,y,z]$

It is straightforward how to do adjust this to instead write the *field* generated by the variables x,y,z:

```
\setupclass{var}{
  novaluekeys={
    {pol}{
      par,  % This tells semantex to use parentheses around
            % the argument in the first place, in case this
            % had been turned off
      leftpar=[,rightpar=],
    },
    {field}{
      par,
      leftpar=(,rightpar=),
    },
  },
}
```

Now `\vk[field]{\vx,\vy,\vz}` produces $k(x,y,z)$. Of course, leaving out the `field` key would produce the same result with the current configuration of `var`. However, it is still best to use a key for this, both because this makes the semantics more clear, but also because you might later change some settings that would cause the default behaviour to be different.

Adding support for free algebras and fields is almost as easy, but there is a catch:

```
\setupclass{var}{
```

```
  novaluekeys={
    {pol}{
      par,  % This tells semantex to use parentheses around
            % the argument in the first place, in case this
            % had been turned off
      leftpar=[,rightpar=],
    },
    {field}{
      par,
      leftpar=(,rightpar=),
    },
    {freealg}{
      par,
      leftpar=\noexpand\langle,
      rightpar=\noexpand\rangle,
    },
    {powerseries}{
      par,
      leftpar=\noexpand\llbracket,
      rightpar=\noexpand\rrbracket,
    },
    {laurent}{
      par,
      leftpar=(, rightpar=),
      prearg={\!\!\noexpand\semantexdelimsize(},
      postarg={\noexpand\semantexdelimsize)\!\!},
      % These are printed before and after the argument.
      % The command "\semantexdelimsize" is substituted
      % by \big, \Big, ..., or whatever size the
      % argument delimiters have
    },
  },
}
```

For expansion reasons (which I am not completely sure of), we need `\noexpand` before these commands. In general, whenever something fails, try throwing in `\noexpand`'s in front of suspicious-looking commands, and things will usually work out just fine. See for yourself:

```
$\vk[freealg]{\vx}$,
$\vk[powerseries]{\vy}$,
$\vk[laurent]{\vz}$
```

$k\langle x\rangle$, $k[\![y]\!]$, $k(\!(z)\!)$

Let us look at some other algebraic operations that we can control via SemanTEX:

```
\setupclass{var}{
  novaluekeys={
    {op}{upper={\noexpand\mathrm{op}}},
      % opposite groups, rings, categories, etc.
    {algclosure}{overline},
      % algebraic closure
    {conj}{overline},
      % complex conjugation
    {dual}{upper=*},
```

```
      % dual vector space
    {perp}{upper=\perp},
      % orthogonal complement
  },
  valuekeys={
    {mod}{symbolputright={/#1}},
      % for modulo notation like R/I
    {dom}{symbolputleft={#1\backslash}},
      % for left modulo notation like I\R
      % "dom" is "mod" spelled backwards
    {oplus}{upper={\oplus#1}},
      % for notation like R^{\oplus n}
    {tens}{upper={\otimes#1}},
      % for notation like R^{\otimes n}
    {localize}{symbolputright={ \relax [#1^{-1}] }},
      % localization at a multiplicative subset;
      % the \relax is necessary becauese, in some cases,
      % the [...] can be interpreted as an optional argument
    {localizeprime}{lower={#1}},
      % for localization at a prime ideal
  },
}
```

Let us see it in practice:

```
$\vR[op]$, $\vk[algclosure]$,
$\vz[conj]$, $\vV[dual]$,
$\vR[mod=\vI]$,$\vR[dom=\vJ
]$,
$\vR[oplus=\vn]$,
$\vV[tens=\vm]$,
$\vR[localize=\vS]$,
$\vR[localizeprime=\vI]$,
$\vk[freealg]{\vS}[op]$,
$\vV[perp]$
```

R^{op}, $\overline{k}$, $\overline{z}$, V^*, R/I,$J\backslash R$, $R^{\oplus n}$, $V^{\otimes m}$, $R[S^{-1}]$, R_I, $k\langle S\rangle^{\mathrm{op}}$, $V^{\perp}$

4 The spar key

The spar key is one of the most important commands in SemanTEX at all. To understand why we need it, imagine you want to derive a function n times and then invert it. Writing something like

```
$\vf[der=\vn,inv]$
```

$f^{(n)-1}$

does not yield a satisfactory result. However, the spar key saves the day:

```
$\vf[der=\vn,spar,inv]$
```

$(f^{(n)})^{-1}$

So spar simply adds a pair of parentheses around the current symbol, complete with all indices that you may have added to it so for. The name spar stands for "symbol parentheses". You can add as many as you like:

```
$ \vf[1,res=\vV,spar,conj,op,
spar,0,inv,spar,mod=\vI,spar,
dual]{\vx} $
```

$$((((\overline{(f_1|_V)})^{\mathrm{op}})_0^{-1})/I)^*(x)$$

If it becomes too messy, you can scale the parentheses, too. Simply use the syntax `spar=\big`, `spar=\Big`, etc. You can also get auto-scaled parentheses base on `\left...\right`, using the key `spar=auto`:

```
$\vf[spar]$,
$\vf[spar=\big]$,
$\vf[spar=\Big]$,
$\vf[spar=\bigg]$,
$\vf[spar=\Bigg]$,
$\vf[spar=auto]$
```

$$(f), \big(f\big), \Big(f\Big), \bigg(f\bigg), \Bigg(f\Bigg), (f)$$

So returning to the above example, we can write

```
$\vf[1,res=\vV,spar,conj,op,
spar=\big,0,inv,spar=\Big,mod
=\vI,spar=\bigg,dual]{\vx}$
```

$$\bigg(\Big(\big((\overline{(f_1|_V)})^{\mathrm{op}}\big)_0^{-1}\Big)/I\bigg)^*(x)$$

To adjust the type of parentheses, use the `leftspar` and `rightspar` keys:

```
$\vf[leftspar={[},rightspar
={\}},spar,spar=\Bigg]$
```

$$\left[f\right\}$$

Occassionally, it is useful to be able to input a particular kind of parentheses just once, without adjusting any settings. For this purpose, we have the (previously mentioned) `otherspar` key. It uses the syntax `otherspar={opening parenthesis}{closing parenthesis}`:

```
$\vf[otherspar={[}{)},
otherspar={\{}{\rangle},
  spar]$
```

$$(\{[f)\rangle)$$

5 The `command` key

Above, we used the key `overline` a couple of times:

```
$\va[overline]$,
$\vH[overline]$
```

$$\overline{a}, \overline{H}$$

This command applies the command `\overline` to the symbol. In fact, you can create similar commands yourself via the `command` key. In fact, you could have defined the `overline` yourself as follows:

```
\setupclass{var}{
  novaluekeys={
    {overline}{command=\noexpand\overline},
  },
}
```

This is how the key `overline` is defined internally, except it is defined on the level of the superclass `automathvariable` instead. We need the key `\noexpand` in order for everything to expand properly. This is only necessary for some commands, and to tell the truth, I haven't quite figured out the system of which commands need it and which ones do not. However, as usual, if something does not work, try throwing in some `\noexpand`'s and see if it solves the problem. Here are some more examples of predefined keys that use the command key:

```
\setupclass{var}{ % do not add these -- they are already
predefined!
  novalueskeys={
    {smash}{command=\noexpand\smash},
    {tilde}{command=\noexpand\tilde},
    {widetilde}{command=\widetilde},
    {bar}{command=\noexpand\bar},
    {bold}{command=\noexpand\mathbf},
    {roman}{command=\noexpand\mathrm},
  },
}
```

Let us test:

```
$\va[widetilde]$,
$\va[bold]$,
$\va[roman]$,
$\va[bar]$
```

$\widetilde{a}$, $\mathbf{a}$, a, $\bar{a}$

6 The `return` keys

Suppose you want to take the complex conjugate of the variable z_1. Then you might write something like

```
$\vz[1,conj]$
```

$\overline{z}_1$

Notice that the bar has only been added over the z, as is standard mathematical typography; you do not normally write $\overline{z_1}$. This reveals a design choice that has been made in SemanTEX: When you add an index or a command via the `command` key, it is not immediately applied to the symbol. Rather, both commands and indices are added to a register and are then applied at the very last, right before the symbol is printed. This allows us to respect standard mathematical typography, as shown above.

However, there are other times when this behaviour is not what you want. For instance, if you want to comjugate the inverse of a function, the following looks wrong:

```
$\vf[inv,conj]$
```

$\overline{f}^{-1}$

Therefore, there is a command `return` that can be applied at any point to invoke the routine of adding all indices and commands to the symbol. Let us try it out:

```
$\vf[inv,return,conj]$
```

$\overline{f^{-1}}$

In fact, `return` is an umbrella key that invokes three different return routines: `leftreturn`, `innerreturn`, and `rightreturn`. The command `leftreturn` adds the left indices to the symbol (we have not discussed left indices yet, though). The command `innerreturn` adds all commands to the symbol (those defined using the `command` key). Finally, `rightreturn` adds all right indices and arguments to the symbol. In general, the user should probably be satisfied with just using `return`.

7 Example: Algebraic geometry

Let us discuss how to typeset sheaves and operations on morphisms in algebraic geometry. First of all, adding commands for sheaves is not a big deal:

```
\newvar\sheafF{\mathcal{F}}
\newvar\sheafG}{\mathcal{G}}
\newvar\sheafH{\mathcal{H}}
\newvar\sheafO{\mathcal{O}}
\newvar\sheafHom{\mathcal{H}\!\!om}
```

You can of course add as many sheaf commands as you need. Also, to make notations shorter, you could consider calling the commands `\shF`, `\shG`, `\shH`, `\shO`, and `\shHom` instead, as I usually do. You may also want to have a separate command `shreg` for the sheaf $\mathcal{O}$ of regular functions.

Next, for morphisms of schemes $f\colon X \to Y$, we need to be able to typeset comorphisms as well as the one hundred thousand different pullback and pushforward operations. For this, we add some keys to the `var` key:

```
\setupclass{var}{
  novaluekeys={
    {comor}{upper=\#},
      % comorphisms, i.e. f^{\#}
    {inverseimage}{upper={-1},nopar},
      % inverse image of sheaves
    {shpull}{upper=*,nopar},
      % sheaf *-pullback
    {shpush}{lower=*,nopar},
      % sheaf *-pushforward
    {sh!pull}{upper=!,nopar},
      % sheaf !-pullback
    {sh!push}{lower=!,nopar},
      % sheaf !-pushforward
  },
}
```

We have added the command `nopar` to all pullback and pushforward commands since it is custom to write, say, $f^*\mathcal{F}$ rather than $f^*(\mathcal{F})$. Of course, you can decide that for yourself, and in any case, you can write `\vf[shpull,par]{\shF}` if you want to force it to use parentheses in a particular case. Of course, since all SemanTEX variables can be used as functions, so can whatever these pullback and pushforward operations output. So we may write:

```
For a morphism~$ \vf \colon
\vX \to \vY $ with
comorphism~$ \vf[comor]
\colon \sheafO[\vY] \to
\vf[shpush]{\sheafO[\vX]} $,
and for a sheaf~$ \sheafF $
on~$ \vY $, we can define the
pullback~$ \vf[shpull]{
\sheafF} $ by letting~$
\vf[shpull]{ \sheafF}{\vU} =
\cdots $ and the $ ! $-
pullback by letting~$
\vf[sh!pull]{\sheafF}{\vU} =
\cdots $.
```

For a morphism $f\colon X \to Y$ with comorphism $f^{\#}\colon \mathcal{O}_Y \to f_*\mathcal{O}_X$, and for a sheaf $\mathcal{F}$ on Y, we can define the pullback $f^*\mathcal{F}$ by letting $f^*\mathcal{F}(U) = \cdots$ and the !-pullback by letting $f^!\mathcal{F}(U) = \cdots$.

Maybe some people would write `pull`, `push`, etc. instead, but there are other things in math called pullbacks, so I prefer to use the `sh` prefix to show that this is for sheaves. Probably, in the long run, an algebraic geometer might also want to abbreviate `inverseimage` to `invim`.

There are a number of other operations we might want to do for sheaves. We already defined the key `res` for restriction, so there is no need to define this again. However, we might need to stalk, sheafify, take dual sheaves, and twist sheaves. Let us define keys for this:

```
\setupclass{var}{
  valuekeys={
    {stalk}{clower={#1}},
    {twist}{return,symbolputright={(#1)}},
  },
  novaluekeys={
    {sheafify}{upper=+},
    {shdual}{upper=\vee},
  },
}
```

The key `clower` stands for "comma-lower". It is like `lower`, except that it checks whether the index is already non-empty, and if so, it separates the new index from the previous index by a comma. There is, of course, a `cupper` key that does the same with the upper index.

```
$\sheafF[res=\vU,stalk=\vp]$,
$\sheafF[res=\vU,spar,stalk=
\vp]$,
$\sheafO[\vX,stalk=\vp]$,
$\sheafG[sheafify]$,
$\vf[inverseimage]{\sheafO[
\vY]}[spar,stalk=\vx]$
$\sheafG[shdual]$,
$\sheafO[\vX][twist=-1]$,
$\sheafO[twist=1,shdual]$
```

$\mathcal{F}|_{U,p}$, $(\mathcal{F}|_U)_p$, $\mathcal{O}_{X,p}$, $\mathcal{G}^+$, $(f^{-1}\mathcal{O}_Y)_x$ $\mathcal{G}^\vee$, $\mathcal{O}_X(-1)$, $\mathcal{O}(1)^\vee$

8 Example: Homological algebra

Before you venture into homological algebra, you should probably define some keys for the standard constructions:

```
\newvar\Hom{\operatorname{Hom}}
\newvar\Ext{\operatorname{Ext}}
\newvar\Tor{\operatorname{Tor}}
```

Now the ability to easily print indices via the options key will come in handy:

```
$\Hom[\vA]{\vM,\vN}$,
$\Ext[\vA]{\vM,\vN}$
```

$\Hom_A(M,N)$, $\Ext_A(M,N)$

You will probably need several keyval interfaces, some of which will be covered below. But right now, we shall implement a shift operation $X \mapsto X[n]$:

```
\setupclass{var}{
  valuekeys={
    {shift}{ return,symbolputright={ \relax [ {#1} ] } },
    % \relax is necessary since otherwise [...] can
    % occasionally be interpreted as an optional argument
  },
}
```

Let us see that it works:

```
$\vX\mapsto\vX[shift=\vn]$
```

$X \mapsto X[n]$

8.1 The keys `i = index` and `d = deg = degree`

Homological algebra is a place where people have very different opinions about the positions of the gradings. As an algebraist, I am used to *upper* gradings ("cohomological" grading), whereas many topologists prefer *lower* gradings ("homological" grading). The SemanTEX system supports both, but the default is upper gradings (the package author has the privilege to decide). You can adjust this by writing `gradingpos=upper` or `gradingpos=lower`.

We already learned about the keys `upper` and `lower`. There are two more, "relative" keys that print the index either as an upper index or as a lower index, depending on your preference for cohomological or homological grading. They are called

`index` and `degree`

The `degree` is the actual grading in the homological algebra sense. The `index` is an additional index where you can put extra information that you might need. To understand the difference, keep the following two examples in mind: the hom complex $\Hom^\bullet_A$ and the simplicial homology $H^\Delta_\bullet$ (we will define the command `\ho` for homology in the next section):

```
$\Hom[index=\vA,degree=0]$,
$\ho[index=\vDelta,degree=1]$
```

$\Hom^0_A$, H^Δ_1

These names are not perfect; many people would say that the degree is also an index, but feel free to come up with a more satisfactory naming principle, and I shall be happy to consider it. These names probably become a bit too heave to write in the long run, so both keys have abbreviated equivalents:

`i = index` and `d = deg = degree`

Let us see them in action:

```
$ \vX[d=3,i=\vk] $

\setupobject\vX{
  gradingpos=lower
}

$ \vX[d=3,i=\vk] $
```

X_k^3
X_3^k

(We haven't seen the command `\setupobject` before, but I imagine you can guess what it does). If you want to print a bullet as the degree, there is the predefined key `*` for this:

```
$ \vX[*] $

\setupobject\vX{
  gradingpos=lower
}

$ \vX[*] $
```

$X^\bullet$
$X_\bullet$

I guess it is also time to reveal that the previously mentioned shorthand notation `\vx[1]` for indices always prints the `1` on the `index` position. So changing the grading position changes the position of the index:

```
$ \vX[1] $

\setupobject\vX{
  gradingpos=lower
}

$ \vX[1] $
```

X_1
X^1

In other words, in the first example above, we could have written

```
$\Hom[\vA,d=0]$,
$\ho[\vDelta,d=1]$
```

Hom_A^0, H_1^Δ

Note that the use of the short notations `d` and `i` does not mean you cannot write e.g. `\vx[d]` and `\vx[i]`. In fact, this is not the case:

```
$\vf[i]$, $\vf[i=]$,
$\vf[d]$, $\vf[d=]$
```

f_i, f, f_d, f

As we see, it is only when a `d` or `i` key is followed by an equality sign `=` that the routines of these keys are invoked. In fact, SemanTEX carefully separates `valuekeys` from `novaluekeys`.

8.2 The cohomology class type

Now homological algebra is hard unless we can do *cohomology* and *homology*. In principle, this is not hard to do, as we can write e.g. `\vH[d=0]{\vX}` to get $H^0(X)$. However, some people might find it cumbersome to have to write `d=` every time you want to print an index. This is probably the right time to reveal that SemanTEX supports multiple class *types*. So far, we have been exclusively using the `variable` class type, but there are several others. The first one we shall need is the `cohomology` class type, which has a different input syntax that fits cohomology. Let us try to use it:

```
\newcohomologyclass{cohomology}[parent=var,gradingpos=upper]

\newcohomology\co{H}

\newcohomologyclass{homology}[parent=cohomology,gradingpos=
lower]

\newhomology\ho{H}
```

The cohomology command `\co` in general works very much like a command of variable type. However, the input syntax is a bit different:

```
\co[options]{degree}{argument}
```

All three arguments are optional. Let us see it in practice:

```
$\co{0}$, $\co{*}$,
$\co{\vi}{\vX}$,
$\co[\vG]{0}$,
$\co[\vH]{*}$,
$\co[\vDelta]{\vi}{\vX}$
```

H^0, $H^\bullet$, $H^i(X)$, H^0_G, $H^\bullet_H$, $H^i_\Delta(X)$

```
$\ho{0}$, $\ho{*}$,
$\ho{\vi}{\vX}$,
$\ho[\vG]{0}$,
$\ho[\vH]{*}$,
$\ho[\vDelta]{\vi}{\vX}$
```

H_0, $H_\bullet$, $H_i(X)$, H^G_0, $H^H_\bullet$, $H^\Delta_i(X)$

Of course, you can define similar commands for cocycles, coboundaries, and all sorts of other entities that show up in homological algebra.

You might also want to implement feature like reduced cohomology, Čech cohomology, and hypercohomology. This is quite easy with the `command` key:

```
\setupclass{var}{
  novaluekeys={
    {reduced}{command=\widetilde},
    {cech}{command=\noexpand\check},
    {hyper{command=\noexpand\mathbb},
  },
}
```

```
$\co[reduced]{i}$,
$\co[cech]{*}$,
$\co[hyper,cech]{0}{\vX}$
```

$\widetilde{H}^i$, $\check{H}^\bullet$, $\mathbb{\check{H}}^0(X)$

The cohomology class also provides a nice way to implement derived functors:

```
\newcohomology\Lder{\mathbb{L}}[nopar]
\newcohomology\Rder{\mathbb{R}}[nopar]
```

For instance, we can write

```
$\Lder{\vi}{\vf}$,
$\Rder{0}{\vf}$
```

$\mathbb{L}^{i}f$, $\mathbb{R}^{0}f$

Alternatively, the user might prefer to use keyval syntax on the level of the function itself (f in this case). This can be done the following way:

```
\setupclass{var}{
  valuekeys={
    {Lder} {
      innerreturn,leftreturn,
      symbolputleft=\noexpand\mathbb{L}^{#1},
    },
    {Rder} {
      innerreturn,leftreturn,
      symbolputleft=\noexpand\mathbb{R}^{#1},
    },
  },
  novaluekeys={
    {Lder} {
      innerreturn,leftreturn,
      symbolputleft=\noexpand\mathbb{L},
    },
    {Rder} {
      innerreturn,leftreturn,
      symbolputleft=\noexpand\mathbb{R},
    },
  },
}
```

Then the syntax becomes:

```
$\vF[Lder=\vi]$,
$\vF[Lder]{\vX[*]}$,
$\vF[Rder]{\vX[*]}$,
$\Hom[Rder]{\vX,\vY}$
```

$\mathbb{L}^{i}F$, $\mathbb{L}F(X^{\bullet})$, $\mathbb{R}F(X^{\bullet})$, $\mathbb{R}\mathrm{Hom}(X,Y)$

If you get tired of having to write `\Hom[Rder]` all the time, you can create a shortcut:

```
\newvar\RHom[copy=\Hom,Rder]
```

The `copy` key is like the `parent` key, except it allows you to inherit the settings from an *object* rather than a *class*. Notice that we did not specify a symbol; the symbol argument is optional, and in this case, it was unnecessary, as the symbol was inherited from `\Hom`. Let us see it in action:

```
$\RHom{\vX,\vY}$
```

$\mathbb{R}\mathrm{Hom}(X,Y)$

8.3 Keyval syntax in arguments (Example: Cohomology with coefficients)

Imagine we want to do cohomology with coefficients in some ring R. It is common to write this as $H^\bullet(X;R)$ with a semicolon instead of a comma. This can be implemented, too, with the syntax

```
$\co{*}{\vX,coef=\vR}$
```

$H^\bullet(X;R)$

This shows that arguments of functions also support keyval syntax. In order to customize this, there are two extra keys:

`argnovaluekeys` and `argvaluekeys`

These work exactly like `novaluekeys` and `valuekeys`.

```
\setupclass{var}{
  argvaluekeys={
    {coef}{ sep={;}{#1}  },
  },
}
```

(But it will not quite work yet – stay tuned for a moment!) The key `sep` is a key that controls the separator between the current argument and the previous argument (it will only be printed if there was a previous argument). By default, this separator is a comma. So in the syntax `\co{*}{\vX,coef=\vR}`, there are two arguments, `\vX` and `\vR`, and the separator is a semicolon.

However, even with the above setup, the notation `\co{*}{\vX,coef=\vR}` will not work just yet. For the keys you define using `argvaluekeys` are turned off by default. To turn them on for the object `\co`, run the following code:

```
\setupobject\co{
  valuekeys={
    {arg}{argwithkeyval={#1}},
  },
}
```

The reason the keys are turned off by default is that keys in arguments that support values are only used in very rare cases, like cohomology with coefficients. If such keys were turned on in general, it would mess up every occurrence of an equality sign in arguments, and the following would not work:

```
$\Hom[\sheafO[\vU]]{
  \sheafF[res=\vU],
  \sheafG[res=\vU]
}$
```

$\mathrm{Hom}_{\mathcal{O}_U}(\mathcal{F}|_U, \mathcal{G}|_U)$

It should be noted that there are several predefined keys (of type `novaluekey`) which are defined on the level of the class `semantexvariable`. The full list is: FiXme Fatal: Finish this

- slot, ...

We should also talk about the `arg` key.

8.4 The `binary` class type (Example: Derived tensor products and fibre products)

The SemanTEX system has facilities for printing tensor products $\otimes$ as well as derived tensor products $\otimes^L$. For this, we need the `binary` class type. This has exactly the same syntax as the `variable` class type, except that it cannot take an argument. In other words, its syntax is

```
\command[options]
```

Let us try to use it to define tensor products and fibre products:

```
\newbinaryclass{binaryoperator}[
  novaluekeys={
    {Lder}{upper=L},
    {Rder}{upper=R},
  },
  mathbin,
  % this makes sure that the output is wrapped in \mathbin
]

\newbinaryoperator\tens{\otimes}[
  novaluekeys={
    {der}{Lder},
  },
]

\newbinaryoperator\fibre{\times}[
  % Americans are free to call it \fiber instead
  novaluekeys={
    {der}{Rder},
  },
]
```

As you see, this is one of the few cases where I recommend adding keyval syntax on the level of subclasses. Also, notice that it does not have any `parent=var`, as I do not really see any reason to inherit all the keyval syntax from the `var` class. Now we first define keys `Lder` and `Rder` for left and right derived binary operators. Next, we build in a shortcut in both `\tens` and `\fibre` so that we can write simply `der` and get the correct notion of derived functor. Let us see it in action:

```
$\vA \tens \vB$,
$\vX[*] \tens[\vR] \vY[*]$
$\vk \tens[\vA,der] \vk$,
$\vX \fibre[\vY,der] \vX$
```

$A \otimes B$, $X^\bullet \otimes_R Y^\bullet$ $k \otimes^L_A k$, $X \times^R_Y X$

9 The `\⟨classname⟩` command

So far, we have learned that every mathematical entity should be treated as an object of some class. However, then we run into issues the moment we want to write expressions like

$$(f \circ g)^{(n)}(x).$$

We do not want to have to define a new variable with symbol $f \circ g$ in order to write something like this. Fortunately, once you have created the class `var`, you get an extra command `\var` that has the following syntax

```
\var{symbol}[options]{argument}
```

In other words, it allows you to create a variable on the spot and give in an arbitrary symbol. So the above equation can be written

```
$\var{\vf\circ\vg}[
  spar,der=\vn]{\vx}$
```

$(f \circ g)^{(n)}(x)$

More generally, whenever you create a class with name ⟨`classname`⟩, you automatically get a command named `\`⟨`classname`⟩. It has the same input syntax as the class in question, except that, as above, the first argument is the symbol:

```
\⟨classname⟩{symbol}⟨usual syntax of class⟩
```

Actually, now might be the right time to reveal that the low-level machinery in SemanTEX does not actually see the difference between an object and a class. Yep, this is how it has been implemented, and there are probably some object-oriented purists who will say that this goes against some general programming philosophy nonsense. So when you create the class ⟨`classname`⟩, you really create the above object which has a special syntax. And now all other objects of that class simply inherit from this object. (As we saw from above, you can actually inherit from any object; you just write `copy=\objectname` instead of using `parent`.)

10 Class types

The SemanTEX system uses several different *class types*. We have been almost exclusively using the `variable` class type (which is by far the most important one), but in the last section, we were introduced to the `cohomology` and the `binary` class types.

In fact, all class types are identical internally; the low-level machinery of SemanTEX does not "know" what type a class has. The only difference between the class types is the *input syntax*. In other words, it determines which arguments an object of that class can take. The syntax for creating new objects also varies.

The current implementation has the following class types:

- `variable`: A new class is declared with the syntax

  ```
  \newvariableclass{⟨classname⟩}[options]
  ```

  A new object is declared by

  ```
  \new⟨classname⟩\⟨objectname⟩{symbol}[options]
  ```

  The syntax for this object is

  ```
  \⟨objectname⟩[options]{argument}
  ```

- `cohomology`: A new class is declared with the syntax

  ```
  \newcohomologyclassclass{⟨classname⟩}[options]
  ```

A new object is declared by

```
\new⟨classname⟩\⟨objectname⟩{symbol}[options]
```

The syntax for this object is

```
\⟨objectname⟩[options]{degree}{argument}
```

- `binary`: A new class is declared with the syntax

  ```
  \newdelimiterclass{⟨classname⟩}[options]
  ```

  A new object is declared by

  ```
  \new⟨classname⟩\⟨objectname⟩{left bracket}{right
  bracket}[options]
  ```

  The syntax for this object is

  ```
  \⟨objectname⟩[options]{argument}
  ```

- `delimiter`: A new class is declared with the syntax

  ```
  \newtupleclass{⟨classname⟩}[options]
  ```

  A new object is declared by

  ```
  \new⟨classname⟩\⟨objectname⟩{left bracket}{right
  bracket}[options]
  ```

  The syntax for this object is

  ```
  \⟨objectname⟩[options]{argument}
  ```

Let me add that SemanTEX uses a very clear separation between the input syntax and the underlying machinery. Because of this, if the user needs a different kind of class type, it is not very hard to create one. You must simply open the source code of SemanTEX, find the class you want to modify, and then copy the definition of the command `\new⟨class type⟩class` and modify it in whatever way you want.

11 The `delimiter` class type

Delimiters are what they sound like: functions like $\|-\|$ and $\langle -,-\rangle$ that are defined using brackets only. Let us define a class of type delimiter:

```
\newdelimiterclass{delim}[parent=var]
```

Now we get a command `\newdelim` with the following syntax:

```
\newdelim\⟨objectname⟩{left bracket}{right bracket}[options]
```

Now we can do the following:

```
\newdelim\norm{\lVert}{\rVert}
\newdelim\inner{\langle}{\rangle}
```

Indeed:

```
$\norm{\va}$,
$\inner{\va,\vb}$,
$\inner{slot,slot}$
```

$\|a\|$, $\langle a,b\rangle$, $\langle -,-\rangle$

We can also use it for more complicated constructions, like sets.

```
\newcommand\where{
  \nonscript\:
  \semantexdelimsize\vert
  \allowbreak
  \nonscript\:
  \mathopen{}
}

\newdelim\Set{\lbrace}{\rbrace}[
  prearg={\,},postarg={\,},
  % adds \, inside {...}, as recommended by D. Knuth
  valuekeys={
    {arg}{argwithoutkeyval={#1}},
    % this turns off all keyval syntax in the argument
  }
]
```

Now you can use

```
$\Set{ \vx \in \vY \where
\vx \ge 0 }$
```

$\{x \in Y \mid x \geq 0\}$

Don't forget that anything created with SemanTEX outputs as a variable-type object. So you can do stuff like

```
$\Set{
  \vx \in \vY[\vi]
  \where
  \vx \ge 0
}[conj,\vi\in\vI]$
```

$\overline{\{x \in Y_i \mid x \geq 0\}}_{i\in I}$

Tuple-like commands are also possible:

```
\newdelim\tup{(}{)} % tuples
\newdelim\pcoor{[}{]}[ % projective coordinates
  argsep=\mathpunct{:}, % changes the argument separator to :
  argdots=\cdots, % changes what is inserted if you write "..."
]
```

Let us see them in action:

```
$\tup{\va,\vb,...,\vz}$,
$\pcoor{\va,\vb,...,\vz}$
```

$(a,b,\ldots,z)$, $[a{:}b{:}\cdots{:}z]$

One can also use tuples for other, less obvious purposes, like calculus differentials:

```
\newdelimiterclass{calculusdifferential}[
  parent=var,
  argvaluekeys={
    {default}{standardsep={d\!#1}},
  },
```

```
  argdots=\cdots,
  ifpar=false,
]

\newcalculusdifferential\intD{(}{)}[argsep={\,},iffirstarg=
false]

\newcalculusdifferential\wedgeD{(}{)}[argsep=\wedge]


$\int \vf \intD{\vx[1],\vx
[2],...,\vx[n]}$,

$\int \vf \wedgeD{\vx[1],\vx
[2],...,\vx[n]}$
```

$\int f\,dx_1\,dx_2\cdots dx_n$,
$\int f\,dx_1\wedge dx_2\wedge\cdots\wedge dx_n$

12 The `execute` and `parseoptions` keys

As you can see above, SemanTEX has a "waterfall-like" behaviour. It parses keys in the order it receives them. This works fine most of the time, but for some more complicated constructions, it is useful to be able to provide a data set in any order and have them printed in a fixed order. For this purpose, we have the `execute` and `parseoptions` keys.

Suppose we want to be able to write the set of $n\times m$-matrices with entries in k as $\mathrm{Mat}_{n\times m}(k)$. We can in principle do the following:

```
$ \Mat[\vn\times\vm]{\vk} $.
```

$\mathrm{Mat}_{n\times m}(k)$.

However, this is not quite as systematic and semantic as we might have wanted. Indeed, what if later you would like to change the notation to $\mathrm{Mat}_{n,m}(k)$? Therefore, we do something like the following instead (we explain the notation below):

```
\newvar\Mat{\operatorname{Mat}}[
  execute={
    \semantexdataprovide{rows}
    \semantexdataprovide{columns}
     % provides data sets for number of rows and columns
     % for this object
  },
  valuekeys={
    {rows}{
      execute={
        \semantexdataset{rows}{#1}
      },
    },
    {columns}{
      execute={
        \semantexdataset{columns}{#1}
      },
    },
  },
  parseoptions={
```

```
    execute={
      \semantexstrifeq{\semantexdatagetexpnot{columns}}
      {\semantexdatagetexpnot{rows}}
      % tests if rows = columns
      {
        \semantexsetkeysx{
          lower={
            \semantexdatagetexpnot{columns}
          }
        }
      }
      {
        \semantexsetkeysx{
          lower={
            \semantexdatagetexpnot{rows}
            \times
            \semantexdatagetexpnot{colums}
          }
        }
      }
    },
  },
]
```

Now we can do the following:

```
$ \Mat[rows=\vn,columns=\vm
]{\vk} $, $ \Mat[rows=\vn,
columns=\vn]{\vk} $
```

$\mathrm{Mat}_{n\times m}(k)$, $\mathrm{Mat}_{n}(k)$

The key execute is a key that basically just executes code. You can in principle write any TEX code there, and it will be applied right at the spot. However, inside the execute key, you can also use the following locally defined commands. These can be used to handle the data that is associated with the object in question:

```
\semantexdataprovide{name} % provides a data set with this name
\semantexdataset{name}{value} % sets the data set
\semantexdatasetx{name}{value} % sets the data set, but fully
expands the argument
\semantexdataputright{name}{value} % adds something to the
right of the data set
\semantexdataputrightx{name}{value} % the same, but fully
expands first
\semantexdataputleft{name}{value} % adds something to the left
of the data set
\semantexdataputleftx{name}{value} % the same, but fully
expands first
\semantexdataget{name}{value} % outputs the data set
\semantexdatagetexpnot{name}{value} % outputs the data set
wrapped in a \noexpand
\semantexdataclear{name} % clears the data set
\semantexsetkeys{keys} % sets keys
\semantexsetkeysx{keys} % sets keys after expanding
\semantexstrifeq{str1}{str2}{if true}{if false} % tests if str1
 = str2
```

```
\semantexboolprovide{name} % provides a boolean
\semantexboolsettrue{name} % sets the boolean to true
\semantexboolsetfalse{name} % sets the boolean to false
\semantexboolif{name}{if true}{if false} % tests the boolean
```

The key `parseoptions` is a key that is executed right before rendering the object. This is where you write whatever the system is supposed to *do* with the data sets you provide.

13 Bugs

The most important current (known) bug happens if you create a variable whose symbol is a mathematical operator. For instance, write

```
\newvar\Int{\int}
```

```
$\int$ \\
$\Int$ \\
See the difference: \rlap{$\
int$}$\Int$
```

$\int$
$\int$
See the difference: $\iint$

It turns out to be equivalent to the difference between `$\int$` and `${}\int$`. In other words, this `{}` affects the spacing a tiny bit. I more or less know where this bug appears, but cannot really solve it without breaking the expansion somewhere else. Suggestions and advice are more than welcome! Then again, even if it affects the spacing a little bit, it still looks fine, only a bit different.