SemanTEX: Semantic mathematics (v0.3α)

Sebastian Ørsted (sorsted@gmail.com)

July 21, 2020

The SemanTEX package for LATEX delivers a more semantic, systematized way of writing mathematics, compared to the classical math syntax in LATEX. The system uses keyval syntax and is highly customizable. At the same time, care has been taken to make it the syntax as simple, natural, practical, and lightweight as possible. **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!**

Let us take an example from elementary analysis to demonstrate the idea of the package: Suppose we want to take the complex conjugate of a function f and then derive it n times, i.e. take $\overline{f}^{(n)}$. SemanTEX allows you to typeset this something like this:

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

$\overline{f}^{(n)}$

I shall explain the syntax in detail below, but some immediate comments are in order: First and foremost, the `v` in the command names `\vf` and `\vn` stands for "variable", so these commands are the variables f and n. In SemanTEX, it is usually best to create commands `\va`, `\vA`, `\vb`, `\vB`, ... for each variable you are using, upper- and lowercase. However, it is completely up to the user how to do that and what to call them. Note also that all of the keys `inv`, `res`, etc. are defined by the *user*, and they can be modified and adjusted for all sorts of situations in any kinds of mathematics. In other words, for the most part, you get to choose your own syntax.

Next, suppose we want to invert a function g and restrict it to a subset U, and then apply it to x, i.e. take $g^{-1}|_U(x)$. This can be done by writing

```
$ \vg[inv,res=\vU]{\vx} $
```

$g^{-1}|_U(x)$

Next, let us take an example from algebraic geometry: Suppose $\mathcal{F}$ is a sheaf and h a map, and that we want to typeset the equation $(h^{-1}\mathcal{F})_p = \mathcal{F}_{h(p)}$, saying that the stalk of the inverse image $h^{-1}\mathcal{F}$ at the point p is $\mathcal{F}_{h(p)}$. This can be accomplished by typing

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

$(h^{-1}\mathcal{F})_p = \mathcal{F}_{h(p)}$

Here, `spar` (an abbreviation for "symbol parentheses") is the key that adds the parentheses around $h^{-1}\mathcal{F}$.

Let us see how you could set up all the above notation:

```
\documentclass{article}

\usepackage{amsmath,semantex}

\NewVariableClass\MyVar % creates a new class of variables,
called "\MyVar"

% Now we create a couple of variables of the class \MyVar:
\NewObject\MyVar\vf{f}
\NewObject\MyVar\vg{g}
\NewObject\MyVar\vh{h}
\NewObject\MyVar\vn{n}
\NewObject\MyVar\vp{p}
\NewObject\MyVar\vU{U}
\NewObject\MyVar\vx{x}
\NewObject\MyVar\sheafF{\mathcal{F}}

% Now we set up the class \MyVar:
\SetupClass\MyVar{
  output=\MyVar,  % This means that the output of an object
                  % of class \MyVar is also of class \MyVar
  % We add a few keys for use with the class \MyVar:
  singlekeys={ % keys taking no values
    {inv}{upper={-1}},
    {conj}{overline},
    {inverseimage}{upper={-1},nopar},
  },
  valuekeys={ % keys taking a value
    {der}{upper={(#1)}},
    {stalk}{clower={#1}},
    % "clower" means "comma lower", i.e. lower index
    % separated from any previous lower index by a comma
    {res}{ return ,symbolputright ={|}, lower ={#1} },
  },
}

\begin{document}

$ \vf[conj,der=\vn] $

$ \vg[inv,res=\vU]{\vx} $

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

\end{document}
```

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 `\SemantexBaseObject`. 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. We choose to call it `\MyVar`. It is best to always start class names with uppercase letters to separate them from objects. We could write `\NewVariableClass\MyVar`, but we choose to pass some options to it in `[...]`:

```
\NewVariableClass\MyVar[output=\MyVar]
```

This `output=\MyVar` 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 `\MyVar`, then `\vf{\vx}` (i.e. $f(x)$) will also of class `\MyVar`.

Now we have a class, but we do not have any objects. To create the object `\vf` of class `\MyVar` with symbol f, we write `\NewObject\MyVar\vf{f}`. In general, when you have class `\⟨Class⟩`, you can create objects of that class wtih the syntax

```
\NewObject\⟨Class⟩\⟨object⟩{⟨object symbol⟩}[⟨options⟩]
```

To distinguish objects from classes, it is a good idea to denote objects by lowercase letters.[1] So after writing,

```
\NewObject\MyVar\vf{f}
\NewObject\MyVar\vx{x}
```

we 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 object is

```
\⟨object⟩[⟨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:

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

$f(1), f(x), x(x)$

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:

[1] We shall not follow this convention strictly, as we shall later create objects with names like `\Hom`; using lowercase letters for these would just look weird.

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

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:

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

\NewObject\MyVar\vA{A}
\NewObject\MyVar\vB{B}
\NewObject\MyVar\vC{C}
\NewObject\MyVar\vD{D}
\NewObject\MyVar\vE{E}
\NewObject\MyVar\vF{F}
\NewObject\MyVar\vG{G}
\NewObject\MyVar\vH{H}
\NewObject\MyVar\vI{I}
\NewObject\MyVar\vJ{J}
\NewObject\MyVar\vK{K}
\NewObject\MyVar\vL{L}
```

```
\NewObject\MyVar\vM{M}
\NewObject\MyVar\vN{N}
\NewObject\MyVar\vO{O}
\NewObject\MyVar\vP{P}
\NewObject\MyVar\vQ{Q}
\NewObject\MyVar\vR{R}
\NewObject\MyVar\vS{S}
\NewObject\MyVar\vT{T}
\NewObject\MyVar\vU{U}
\NewObject\MyVar\vV{V}
\NewObject\MyVar\vW{W}
\NewObject\MyVar\vX{X}
\NewObject\MyVar\vY{Y}
\NewObject\MyVar\vZ{Z}

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

\NewObject\MyVar\vGamma{\Gamma}
\NewObject\MyVar\vDelta{\Delta}
\NewObject\MyVar\vTheta{\Theta}
\NewObject\MyVar\vLambda{\Lambda}
\NewObject\MyVar\vXi{\Xi}
\NewObject\MyVar\vPi{\Pi}
\NewObject\MyVar\vSigma{\Sigma}
\NewObject\MyVar\vUpsilon{\Upsilon}
\NewObject\MyVar\vPhi{\Phi}
\NewObject\MyVar\vPsi{\Psi}
\NewObject\MyVar\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` (it will still print parentheses if there is more than one argument, though; to exclude this behaviour, run `neverpar` instead):

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

$$fx$$

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:

`singlekeys` and `valuekeys`.

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

For the rest of the manual, we assume that you have already defined a class `\MyVar` and the variables `\va`, `\vA`, `\vb`, `\vB`, ..., as above.

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 `singlekeys`, which is for keys that do not take a value:

```
\SetupClass\MyVar{
  singlekeys={
    {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}, $g_{1,2}^{-1}$, $h_{a,b}^{-1}$

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 key `der`:

```
\SetupClass\MyVar{
  singlekeys={
    {inv}{ upper={-1} },
  },
  valuekeys={
    {der}{ upper={(#1)} },
  },
}
```

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

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

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

Maybe we also want a more elementary key `power` for raising a variable to a power:

```
\SetupClass\MyVar{
  singlekeys={
    {inv}{ upper={-1} },
  },
  valuekeys={
    {der}{ upper={(#1)} },
    {power}{ upper={#1} },
  },
}
```

This allows us to write

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

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

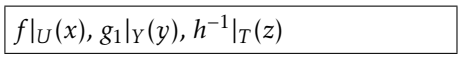
Let us try doing something a bit more complicated: adding a key for restricting a function to a smaller subset. For this, we do the following:

```
\SetupClass\MyVar{
  singlekeys={
    {inv}{ upper={-1} },
  },
  valuekeys={
    {der}{ upper={(#1)} },
    {power}{ 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`).

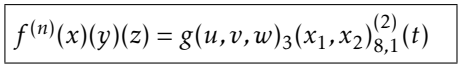
(There is also an extra key, `return`, which is a bit more advanced and should be taken for granted for now. Roughly speaking, it is there to make sure that the restriction symbol is printed *after* all indices that you might have added before. More details in chapter 6.) Now we may write the following:

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



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 is why we wrote `output =\MyVar` in the definition of the class `\MyVar`). This behaviour is both useful and extremely necessary in order for the package to be useful in practice. For instance, you may write

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



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

```
\SetupClass\MyVar{
  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, Otherspar={.}{\vert}{auto},
    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 `\MyVar` 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\MyVar[
  output=\MyVar,  % This means that the output of an object
                  % of class \MyVar is also of class \MyVar
  singlekeys={
    {inv}{ upper={-1} },
  },
  valuekeys={
    {der}{ upper={(#1)} },
    {power}{ upper={#1} },
    {res}{ rightreturn, symbolputright={|},
      lower={#1} },
  },
]
```

As we proceed in this guide, we shall use `\SetupClass` to add more and more keys to `\MyVar`. However, when you set up your own system, you may as well just add all of the keys at once 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=\⟨Class⟩` in the class declaration to tell that `\⟨Class⟩` 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 `\MyVar` that has all the keyval syntax and mainly use subclasses for further customization. We shall see examples of this below.

3 Example: Elementary algebra

Let us try to use SemanTEX 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\MyVar{
  singlekeys={
    {poly}{
      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

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

$k[x,y,z]$

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

```
\SetupClass\MyVar{
  singlekeys={
    {poly}{
      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 the class `\MyVar`. 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, power series, and Laurent series is almost as easy, but there is a catch:

```
\SetupClass\MyVar{
  singlekeys={
    {poly}{
      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=\langle,
      rightpar=\rangle,
    },
    {powerseries}{
      par,
      leftpar=\llbracket,
      rightpar=\rrbracket,
    },
    {laurent}{
      par,
      leftpar=(, rightpar=),
      prearg={\!\!\SemantexDelimiterSize(},
      postarg={\SemantexDelimiterSize)\!\!},
      % These are printed before and after the argument.
      % The command "\SemantexDelimiterSize" is substituted
      % by \big, \Big, ..., or whatever size the
      % argument delimiters have
    },
  },
}
```

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\MyVar{
  singlekeys={
    {op}{upper={\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 far. 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 brackets, 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 brackets just once, without adjusting any settings. For this purpose, we have the otherspar and 0therspar keys. They use the syntax

```
otherspar={⟨opening bracket⟩}{⟨closing bracket⟩}
0therspar={⟨opening bracket⟩}{⟨closing bracket⟩}{⟨size⟩}
```

Let us see them in action:

```
$\vf[otherspar={[}{)},
otherspar={\{}{\rangle},
  0therspar={\langle}{\rangle
  }{\Bigg},spar]$
```
$(\Big\langle\{[f)\rangle\Big\rangle)$

5 The \⟨Class⟩ 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 $f \circ g$ just to write something like this. Fortunately, once you have created the class `\MyVar`, you can actually use `\MyVar` as a command to create a quick instance of the class. More precisely `\MyVar{⟨symbol⟩}` creates a variable on the spot with symbol ⟨symbol⟩. So the above equation can be written

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

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

More generally, when you crate the class `\⟨Class⟩`, you can use it as a command with the following syntax:

```
\⟨Class⟩{⟨symbol⟩}[⟨options⟩]⟨usual syntax of class⟩
```

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 normally do not 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 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\MyVar{
  singlekeys={
    {overline}{command=\overline},
  },
}
```

This is how the key `overline` is defined internally, except it is defined on the level of the superclass `\SemantexBaseObject` instead. Here are some more examples of predefined keys that use the command key:

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

Let us test:

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

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

8 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:

```
\NewObject\MyVar\sheafF{\mathcal{F}}
\NewObject\MyVar\sheafG}{\mathcal{G}}
\NewObject\MyVar\sheafH{\mathcal{H}}
\NewObject\MyVar\sheafreg{\mathcal{O}}
  % sheaf of regular functions
\NewObject\MyVar\sheafHom{\mathop{\mathcal{H}om}}
```

You can of course add as many sheaf commands as you need.

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 `\MyVar` key:

```
\SetupClass\MyVar{
  singlekeys={
    {comorphism}{upper=\#},
```

```
    % comorphisms, i.e. f^{\#}
  {inverseimage}{upper={-1},nopar},
    % inverse image of sheaves
  {sheafpull}{upper=*,nopar},
    % sheaf *-pullback
  {sheafpush}{lower=*,nopar},
    % sheaf *-pushforward
  {sheaf!pull}{upper=!,nopar},
    % sheaf !-pullback
  {sheaf!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[sheafpull,par]{\sheafF}` 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[comorphism]
\colon \sheafreg[\vY] \to
\vf[sheafpush]{\sheafreg[\vX
]} $,
and for a sheaf~$ \sheafF $
on~$ \vY $, we can define the
pullback~$ \vf[sheafpull]{
\sheafF} $ by letting~$
\vf[sheafpull]{ \sheafF}{\vU}
 = \cdots $ and the $ ! $-
pullback by letting~$
\vf[sheaf!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 many different kinds of pullbacks in mathematics, so I prefer to use the `sheaf` 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\MyVar{
  valuekeys={
    {stalk}{clower={#1}},
    % "clower" means "comma lower", i.e. lower index
    % separated from any previous lower index by a comma
    {sheaftwist}{return,symbolputright={(#1)}},
  },
  singlekeys={
    {sheafify}{upper=+},
```

```
    {sheafdual}{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.

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

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

9 Example: Homological algebra

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

```
\NewObject\MyVar\Hom{\operatorname{Hom}}
\NewObject\MyVar\Ext{\operatorname{Ext}}
\NewObject\MyVar\Tor{\operatorname{Tor}}
```

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

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

$\mathrm{Hom}_A(M,N)$, $\mathrm{Ext}_A(M,N)$

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\MyVar{
  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:

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

$X \mapsto X[n]$

Finally, let us define a command for the differential (in the homolgoical algebra sense):

```
\NewObject\MyVar\dif{d}[nopar]

$\dif{\vx} = 0$
```

$dx = 0$

9.1 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 `gradingposition=upper` or `gradingposition=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 $\mathrm{Hom}^{\cdot}_{A}$ and the simplicial homology $H^{\Delta}_{\cdot}$ (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 heavy 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{
  gradingposition=lower
}

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

X^{3}_{k}
X^{k}_{3}

(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{
  gradingposition=lower
}

$ \vX[*] $
```

$X^{\cdot}$
$X_{\cdot}$

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{
  gradingposition=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^0_A, H_1^Δ

Note that the use of the short notations `d` and `i` does not prevent you from writing `\vx[d]` and `\vx[i]`. This still works fine:

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

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 `singlekeys`.

9.2 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, which is what you create when you apply the command `\NewVariableClass`. The first other class type we shall need is the `Cohomology` class type, which has a different input syntax that fits cohomology. Let us try to use it:

```
\NewCohomologyClass\MyCohomology[
  parent=\MyVar,gradingposition=upper
]

\NewObject\MyCohomology\co{H}

\NewCohomologyClass\MyHomology[
  parent=\MyCohomology,gradingposition=lower
]

\NewObject\MyHomology\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:

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

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

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

$H_0, H_\bullet, H_i(X), H_0^G, H_\bullet^H, H_i^\Delta(X)$

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\MyVar{
  singlekeys={
    {reduced}{command=\widetilde},
    {cech}{command=\check},
    {hyper{command=\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 type also provides a nice way to implement derived functors:

```
\NewObject\MyCohomology\Lder{\mathbb{L}}[nopar]
\NewObject\MyCohomology\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\MyVar{
  valuekeys={
    {Lder} {
      innerreturn,leftreturn,
      symbolputleft=\mathbb{L}^{#1},
    },
    {Rder} {
      innerreturn,leftreturn,
      symbolputleft=\mathbb{R}^{#1},
    },
  },
  singlekeys={
    {Lder} {
```

```
      innerreturn,leftreturn,
      symbolputleft=\mathbb{L},
    },
    {Rder} {
      innerreturn,leftreturn,
      symbolputleft=\mathbb{R},
    },
  },
}
```

Then the syntax becomes:

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

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

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

```
\NewObject\MyVar\RHom[clone=\Hom,Rder]
```

The `clone` 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:

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

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

10 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:

`argsinglekeys` and `argvaluekeys`

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

```
\SetupClass\MyVar{
  argkeyval=true, % this turns keyval syntax in arguments on
  argvaluekeys={
    {coef}{ othersep={;}{#1}  },
  },
}
```

The key `othersep` 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.

As you see above, we had to turn keyval syntax on in order for it to work. By default, only singlekeys are turned on in the argument, not valuekeys. The reason is that valuekeys in arguments are only useful 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:

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

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

The key `argkeyval` can take four arguments: `true` (as above), `false` (no singlekeys or valuekeys allowed), `singlekeys` (the default behaviour where only singlekeys are turned on), and `onesinglekey` (only allows one singlekey).

It should be noted that there are several predefined singlekeys which are defined on the level of the class `\SemantexBaseObject`. The full list is:

FiXme Fatal: Finish this

- slot, ...

We should also talk about the `arg` key.

11 Multi-value keys

Sometimes, a key with one value is simply not enough. For instance, if you work in GIT, you will eventually have to take the proj quotient $X/\!\!/_\chi G$ of X with respect to the action of the group G and the character χ. In other words, the proj quotient depends on two parameters, χ and G. For this purpose, we have the key `2valuekeys`. It works exactly like `valuekeys`, except you get to use two arguments instead of one:

```
\SetupClass\MyVar{
  2valuekeys={
    {projquotient}{ symbolputright={ /\!\!/ _ { #1 } #2 } },
  }
}
```

```
$\vX[projquotient={\vchi}{\vG
}] $
```

$X/\!\!/_\chi G$

There are also commands `3valuekeys`, `4valuekeys`, ..., `9valuekeys`. The syntax for these is completely analoguous. There are also commands `arg2valuekeys`, `arg3valuekeys`, ..., `arg9valuekeys` for keys in arguments with multiple values.

12 The `Simple` 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 `Simple` 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

```
\⟨object⟩[⟨options⟩]
```

(You should normally only use it for special constructions like binary operators and not for e.g. variables – the ability to add arguments to variables comes in handy much more often than one might imagine.) Let us try to use it to define tensor products and fibre products:

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

\NewObject\MyBinaryOperator\tensor{\otimes}[
  singlekeys={
    {der}{Lder},
  },
]

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

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

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

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

13 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 chpaters, we were introduced to the `Cohomology` and the `Simple` 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{\⟨Class⟩}[⟨options⟩]
  ```

  A new object is declared by

  ```
  \NewObject\⟨Class⟩\⟨object⟩{⟨symbol⟩}[⟨options⟩]
  ```

  The syntax for this object is

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

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

  ```
  \NewCohomologyClass\⟨Class⟩[⟨options⟩]
  ```

  A new object is declared by

  ```
  \NewObject\⟨Class⟩\⟨object⟩{⟨symbol⟩}[⟨options⟩]
  ```

  The syntax for this object is

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

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

  ```
  \NewSimpleClass\⟨Class⟩[⟨options⟩]
  ```

  A new object is declared by

  ```
  \NewObject\⟨Class⟩\⟨object⟩{⟨symbol⟩}[⟨options⟩]
  ```

  The syntax for this object is

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

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

  ```
  \NewDelimiterClass\⟨Class⟩[⟨options⟩]
  ```

  A new object is declared by

  ```
  \NewObject\⟨Class⟩\⟨object⟩{⟨left bracket⟩}{⟨right
  bracket⟩}[⟨options⟩]
  ```

The syntax for this object is

```
\⟨object⟩[⟨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.

There is another class type, called the `plain` class type, which is the class type of the class `\SemantexBaseObject`. This class is pretty useless as all it does is print its symbol, without allowing any keyval syntax, so don't use it.

14 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\MyDelim[parent=\MyVar]
```

Now we can create instances of the class `\MyDelim` with the following syntax:

```
\NewObject\MyDelim\⟨object⟩{⟨left bracket⟩}{⟨right bracket⟩}[⟨
options⟩
```

Now we can do the following:

```
\NewObject\MyDelim\norm{\lVert}{\rVert}
\NewObject\MyDelim\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. The following is inspired from the `mathtools` package where a similar construction is created using the commands from that package. My impression is that Lars Madsen is the main mastermind behind the code I use for the `\where` command:

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

\NewObject\MyDelim\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:

```
\NewObject\MyDelim\tup{(}{)} % tuples
\NewObject\MyDelim\pcoor{[}{]}[ % projective coordinates
  setargsep=\mathpunct{:},
    % changes the argument separator to colon
  setargdots=\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=\MyVar,
  argvaluekeys={
    {default}{s={d\!#1}},
    % default is the key that is automatically applied by the
    % system to anything you write in the argument. The s key
    % is a key that prints the value of the key with the
    % standard argument separator in front.
  },
  setargdots=\cdots,
  neverpar,
  % neverpar is like nopar, except nopar will still print
  % parentheses when there is more than one argument
  % -- neverpar does not even print parentheses in this case
]

\NewObject\CalculusDifferential\intD{(}{)}[setargsep={\,},
iffirstarg=false]

\NewObject\CalculusDifferential\wedgeD{(}{)}[setargsep=\wedge]

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

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

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

15 The parse routine

As you can see above, SemanTEX has a "waterfall-like" behaviour. It runs 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 parse routine.

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:

```
\NewObject\MyVar\Mat{
  \operatorname{Mat}}
$ \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 $\mathrm{Mat}_{n,m}(k)$? (In this case, you could use multi-value keys, though.) In this chapter, we show how to eanble a syntax like the following instead:

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

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

The important ingredient here is the parse routine. This routine is executed right before the function is being rendered, and you can add code to it via the key parseoptions. However, we need a bit more programming keys to make it work. Let us see it in action and explain the syntax below:

```
\NewObject\MyVar\Mat{\operatorname{Mat}}[
  % We provide data sets "rows" and "columns" to
  % be set up by the user later
  dataprovide={rows},
  dataprovide={columns},
  valuekeys={
    {rows}{ dataset={rows}{#1} }, % set the rows data set
    {columns}{ dataset={columns}{#1} }, % set the columns data
    set
  },
  parseoptions={ % Here we add code to the parse routine
    % We check whether columns = rows. If so, we only write
    % the number once
    ifeqTF={\SemantexDataGetExpNot{columns}}
           {\SemantexDataGetExpNot{rows}}
    {
      % We use a very weird key called "setkeysx" -- this
      % fully executes the content of the keys before
      % setting them
      setkeysx={
        lower={\SemantexDataGetExpNot{columns}},
      },
    }
    {
      setkeysx={
        lower={
          \SemantexDataGetExpNot{rows}
```

```
          \times
          \SemantexDataGetExpNot{columns}
        },
      },
    },
  },
]
```

Here we used a lot of programmking keys. Let us see the full list of them. (An important notice: For some of these keys, such as `boolifTF`, you currently cannot use spaces in the ⟨`bool`⟩ argument, so e.g. `boolifTF{ mybool } { ... } { ... }` will not work; you have to write `boolifTF{mybool}`. I am trying to solve this problem, but have not yet been able to do so.)

```
dataprovide={⟨data⟩}, % provides data
dataset={⟨data⟩}{⟨value⟩}, % sets data
datasetx={⟨data⟩}{⟨value⟩}, % sets data after expanding it
dataputright={⟨data⟩}{⟨value⟩}, % adds to the right of data
dataputrightx={⟨data⟩}{⟨value⟩}, % adds to the right of data
after expanding it
dataputleft={⟨data⟩}{⟨value⟩}, % adds to the left of data
dataputleftx={⟨data⟩}{⟨value⟩}, % adds to the left of data
after expanding it
dataclear={⟨data⟩,} % clears a piece of data
setkeys={⟨keys⟩}, % sets the keys in question, which is rather
useless since you could have just written those keys directly
instead
keysset={⟨keys⟩}, % equivalent to setkeys
setkeysx={⟨keys⟩},  % executes the keys in question after
expanding them
keysset={⟨keys⟩}, % equivalent to setkeysx
ifeqTF={⟨str1⟩}{⟨str2⟩}{⟨if true⟩}{⟨if false⟩}, % checks if
strings are equal
ifeqT={⟨str1⟩}{⟨str2⟩}{⟨if true⟩},
ifeqF={⟨str1⟩}{⟨str2⟩}{⟨if false⟩},
ifblankTF={⟨str⟩}{⟨if true⟩}{⟨if false⟩}, % checks if string is
 blank
ifblankT={⟨str⟩}{⟨if true⟩},
ifblankF={⟨str⟩}{⟨if false⟩},
boolprovide={⟨bool⟩}, % provides a boolean
boolsettrue={⟨bool⟩}, % sets the boolean to true
boolsetfalse={⟨bool⟩}, % sets the boolean to false
boolifTF={⟨bool⟩}{⟨if true⟩}{⟨if false⟩}, % checks if boolean
is true
boolifT={⟨bool⟩}{⟨if true⟩},
boolifF={⟨bool⟩}{⟨if false⟩,
intprovide={⟨int⟩}, % provides an integer
intclear={⟨int⟩}, % sets the integer to 0
intincr={⟨int⟩}, % adds 1 to the integer
intset={⟨int⟩}{⟨value⟩}, % sts the integer
intifgreaterthanTF={⟨num1⟩}{num2⟩}{⟨if true⟩}{⟨if false⟩}, %
checks which number of greater
intifgreaterthanT={⟨num1⟩}⟨{num2⟩}{⟨if true⟩},
intifgreaterthanF={⟨num1⟩}{⟨num2⟩}{⟨if false⟩},
```

```
intifequalTF={⟨num1⟩}{num2⟩}{⟨if true⟩}{⟨if false⟩}, % checks
if the numbers are equal
intifequalT={⟨num1⟩}⟨{num2⟩}{⟨if true⟩},
intifequalF={⟨num1⟩}{⟨num2⟩}{⟨if false⟩},
intiflessthanTF={⟨num1⟩}{num2⟩}{⟨if true⟩}{⟨if false⟩}, %
checks which number of less
intiflessthanT={⟨num1⟩}⟨{num2⟩}{⟨if true⟩},
intiflessthanF={⟨num1⟩}{⟨num2⟩}{⟨if false⟩},
ERRORkeyvaluenotfound={⟨key⟩}{⟨value⟩}, % throws an error
saying that the key has been set to an unkonwn value
ERROR={⟨error text⟩}, % throws a general error with the
provided error test
execute={⟨error text⟩}, % executes the code in question
```

When using these keys (including inside the key execute), you can use a number of commands that provide and manipulate data. Most of them are just command versions of the keys above, and for now, I leave it to the reader to guess what they do based on the above picture:

```
\SemantexDataProvide
\SemantexDataSet
\SemantexDataSetx
\SemantexDataPutRight
\SemantexDataPutRightx
\SemantexDataPutLeft
\SemantexDataPutLeftx
\SemantexDataGet
\SemantexDataGetExpNot
\SemantexDataClear
\SemantexKeysSet
\SemantexKeysSetx
\SemantexStrIfEqTF
\SemantexStrIfEqT
\SemantexStrIfEqF
\SemantexIfBlankTF
\SemantexIfBlankT
\SemantexIfBlankF
\SemantexBoolProvide
\SemantexBoolSetTrue
\SemantexBoolSetFalse
\SemantexBoolIfTF
\SemantexBoolIfT
\SemantexBoolIfF
\SemantexIntProvide
\SemantexIntGet
\SemantexIntClear
\SemantexIntIncr
\SemantexIntSet
\SemantexIntIfGreaterThanTF
\SemantexIntIfGreaterThanT
\SemantexIntIfGreaterThanF
\SemantexIntIfEqualTF
\SemantexIntIfEqualT
\SemantexIntIfEqualF
\SemantexIntIfLessThanTF
```

```
\SemantexIntIfLessThanT
\SemantexIntIfLessThanF
\SemantexExpNot##1
\SemantexERRORKeyValueNotFound
\SemantexERROR
```

Let us look at a more complicated example: Let us create a command for partial derivatives:

```
\NewObject\MyVar\partialdif[
  nopar,
  boolprovide={raisefunction},
  boolsettrue={raisefunction},
  setidots=\cdots,
  setisep=\,,
  valuekeys={
    {default}{
      si={\partial #1},
    },
    {raise}{
      ifeqTF={#1}{true}
      {
        boolsettrue={raisefunction},
      }
      {
        ifeqTF={#1}{false}
        {
          boolsetfalse={raisefunction},
        }
        {
          ERRORkeyvaluenotfound={raise}{#1},
        },
      },
    },
  },
  parseoptions={
    ifblankTF={ \SemantexDataGet{upper} }
    {
      intifgreaterthanTF={ \SemantexIntGet{numberoflowerindices
      } } { 1 }
      {
        setkeysx={
          symbol={
            \frac
            {
              \partial ^ { \SemantexIntGet{numberoflowerindices
              } }
              \SemantexBoolIfT{raisefunction}
              {
                \SemantexDataGet{arg}
              }
            }
            {
              \SemantexDataGet{lower}
            }
```

```
        },
      },
    }
    {
      setkeysx={
        symbol={
          \frac
          {
            \partial
            \SemantexBoolIfT{raisefunction}
            {
              \SemantexDataGet{arg}
            }
          }
          {
            \SemantexDataGet{lower}
          }
        },
      }
    },
  }
  {
    setkeysx={
      symbol={
        \frac
        {
          \partial ^ { \SemantexDataGet{upper} }
          \SemantexBoolIfT{raisefunction}
          {
            \SemantexDataGet{arg}
          }
        }
        {
          \SemantexDataGet{lower}
        }
      },
    },
  },
  dataclear={lower},
  dataclear={upper},
  boolifT={raisefunction}
  {
    dataclear={arg},
    dataclear={numberofarguments},
  },
},
]
```

Let us see it in action:

```
\[
  \partialdif[\vx,\vy,\vz]{
    \vf } ,
  \partialdif[\vu^2,\vv^2,
    d=4]{ \vf },
  \partialdif[\vx[1],
    \vx[2],...,\vx[\vn],
    d=\vn]{ \vf }
\]
\[
  \partialdif[\vx,\vy,\vz,
  raise=false]{ \vf } ,
  \partialdif[\vu^2,\vv^2,
    d=4,raise=false]{
      \vf },
  \partialdif[\vx[1],
    \vx[2],...,\vx[\vn],
    d=\vn,raise=false]{
      \vf }
\]
```

$$\frac{\partial^3 f}{\partial x\,\partial y\,\partial z}, \frac{\partial^4 f}{\partial u^2\,\partial v^2}, \frac{\partial^n f}{\partial x_1\,\partial x_2\cdots\partial x_n}$$

$$\frac{\partial^3}{\partial x\,\partial y\,\partial z} f, \frac{\partial^4}{\partial u^2\,\partial v^2} f, \frac{\partial^n}{\partial x_1\,\partial x_2\cdots\partial x_n} f$$

As you see, we use the `d` key to tell the command what superscript it should put on the ∂ in the enumerator. If it does not receive a `d`, it counts the number of variables you wrote and prints that. That is why the following would give the wrong result:

```
\[
  \partialdif[\vu^2,\vv^2]{
    \vf },
  \partialdif[\vx[1],
    \vx[2],...,\vx[\vn]]{
    \vf }
\]
```

$$\frac{\partial^2 f}{\partial u^2\,\partial v^2}, \frac{\partial^4 f}{\partial x_1\,\partial x_2\cdots\partial x_n}$$

16 Bugs

The biggest unsolved problem I know of is how to correctly strip spaces in programming keys such as `boolifTF`. Similarly, I would also like to allow keys to be defined using the syntax `{ inv } { upper=-1 }` rather than `{inv}{ upper=-1 }`. This will hopefully be solved soon.

For now, the system seems to work fine as long as you do "normal" things. The only real bug that I currently know of occurs if you use the key `0therspar` in a heading. Then it all dies painfully. Then again, why the heck would you do that in the first place? Who scales parentheses in headings?