SemanTEX: semantic, keyval-based mathematics (v0.525)

Sebastian Ørsted (sorsted@gmail.com)

January 23, 2023

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 the user can define their own keys and customize the system down to the last detail. At the same time, care has been taken to make the syntax as simple, natural, practical, and lightweight as possible.

Furthermore, the package has a companion package, called `stripsemantex`, which allows you to completely strip your documents of SemanTEX markup to prepare them e.g. for publication.

The package is still in beta, but is considered feature-complete and more or less stable, so using it at this point should be safe. Still, suggestions, ideas, and bug reports are more than welcome!

Contents

Chapter 1

Introduction

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] $
```

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

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} $
```

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

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

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

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:
  define keys={ % we define a few keys
    {inv}{upper={-1}},
    {conj}{command=\overline}, % Applies \overline to the symbol
    {inverse image}{upper={-1},no par},
  },
  define keys[1]={ % we define keys taking 1 value
    {der}{upper={(#1)}},
    {stalk}{sep lower={#1}},
    % "sep lower" means "separator + lower", i.e. lower index
    % separated from any previous lower index by a separator,
    % which by default is a comma
    {res}{ right return, symbol put right={|}, lower={#1} },
  },
}

\begin{document}

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

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

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

\end{document}
```

Chapter 2

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 create this class by writing `\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 be 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⟩}
```

---

[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.

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. Naturally, ⟨*argument*⟩ is the actual argument of the function.

By a design choice, SemanTEX does not distinguish between variables and functions, so all variables can take arguments. This makes 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:

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

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

As long as what you write in the options tag is not recognized as a defined 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

In fact, there are quite a few keys for manipulating upper and lower indices. Right now, apart from `upper` and `lower`, we shall only need a couple more: `sep upper` and `sep lower` mean “separator + upper” and “separator + lower”. These are like `upper` and `lower`, but if there already was an upper or lower index, the new index will be separated from the old one by a separator. By default, this separator is a comma. There are also two more commands, `comma upper` and `comma lower`. These will use a comma as separator, even if you have changed the default separator.

2.1 Next step: Defining more variables

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:

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

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

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 `no par` (it will still print parentheses if there is more than one argument, though; to exclude this behaviour, run `never par` instead):

```
$\vf[no par]{\vx}$,
$\vf[no par]{\vx,\vy}$,
$\vf[never par]{\vx}$,
$\vf[never par]{\vx,\vy}$
```

$f x, f(x,y), f x, f x,y$

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

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

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

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.

2.2 Defining keys

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. To define keys, we use the command `\SetupClass` (or `\SetupObject` if you want to define it for an individual object) and the key `define keys`. The syntax is as follows:

```
\SetupClass\MyVar{
  define keys={
    {⟨key name₁⟩}{ ⟨keys to run⟩ },
    {⟨key name₂⟩}{ ⟨keys to run⟩ },
    {⟨key name₃⟩}{ ⟨keys to run⟩ },
    ...,
  },
}
```

For instance, you can do

```
\SetupClass\MyVar{
  define keys={
    {key1}{ upper=3, lower=7 },
    {key2}{ lower=6, upper=4 },
  },
}
```

Quite often, we shall also need to define keys that can *take a value*. A key can take up to 8 values (for technical reasons, 9 values are not allowed). To define a key taking n values, use `define keys[`n`]` for $n = 0, 1, 2, \dots, 8$. The syntax is similar to `define keys`, except the values can be accessed by writing `#1`, `#2`, ..., `#8`. Except for a few special cases, you will probably only ever need `define keys[1]`. So you can do

```
\SetupClass\MyVar{
  define keys[1]={
    {key3}{ upper=\{#1\} },
    {key4}{ lower=(#1) },
  },
  define keys[2]={
    {key5}{ upper=3+#1, lower=7-#2 },
    {key6}{ lower=6\cdot#1, upper=4/#2 },
  },
}
```

Let us see these rather ridiculous keys in action:

```
$ \vP[key1,key3=0,key5={3}{4}] $
```

$P^{3,\{0\},3+3}_{7,7-4}$

Chapter 3

Some examples

3.1 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 `define keys` since there is no reason for this key to take a value:

```
\SetupClass\MyVar{
  define keys={
    {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 one value. For defining these, we use a different key, `define keys[1]`. For instance, suppose we want a command for deriving a function n times. For this, we add the key `der`:

```
\SetupClass\MyVar{
  define keys={
    {inv}{ upper={-1} },
  },
  define keys[1]={
    {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{
  define keys={
    {inv}{ upper={-1} },
  },
  define keys[1]={
```

```
    {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{
  define keys={
    {inv}{ upper={-1} },
  },
  define keys[1]={
    {der}{ upper={(#1)} },
    {power}{ upper={#1} },
    {res}{ right return,symbol put right={|}, 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, `right 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 section 4.4.) 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 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

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

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

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

```
\SetupClass\MyVar{
  define keys[1]={
    {big res}{ right return, symbol put right=\big|, lower={#1} },
    {Big res}{ right return, symbol put right=\Big|, lower={#1} },
    {bigg res}{ right return, symbol put right=\bigg|, lower={#1} },
    {Bigg res}{ right return, symbol put right=\Bigg|, lower={#1} },
    {auto res}{
           left return,
           symbol put left=\kern-\nulldelimiterspace,
           Other spar={.}{|}{auto}, symbol put left=\bgroup,
```

```
              symbol put right=\egroup, lower={#1},
    },
    % The last key auto-scales the vertical bar. See section 4.1
    % for information about Other spar.
    % Note that Other spar automatically invokes right return,
    % so no need to run that key twice.
  },
}
```

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
  define keys={
    {inv}{ upper={-1} },
  },
  define keys[1]={
    {der}{ upper={(#1)} },
    {power}{ upper={#1} },
    {res}{ right return, symbol put right={|}, 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.

3.2 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{
  define keys={
    {poly}{
      par,  % This tells semantex to use parentheses around
            % the argument in the first place, in case this
            % had been turned off
```

```
      left par=[,right par=],
    },
  },
}
```

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{
  define keys={
    {poly}{
      par,  % This tells semantex to use parentheses around
            % the argument in the first place, in case this
            % had been turned off
      left par=[,right par=],
    },
    {field}{
      par,
      left par=(,right par=),
    },
  },
}
```

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{
  define keys={
    {poly}{
      par,  % This tells semantex to use parentheses around
            % the argument in the first place, in case this
            % had been turned off
      left par=[,right par=],
    },
    {field}{
      par,
      left par=(,right par=),
    },
    {free alg}{
      par,
      left par=\langle,
      right par=\rangle,
    },
    {power series}{
      par,
      left par=\llbracket,
      right par=\rrbracket,
    },
```

```
    {laurent}{
      par,
      left par=(, right par=),
      pre arg={\!\mathopen{}\SemantexDelimiterSize(},
      post arg={\SemantexDelimiterSize)\mathclose{}\!},
      % The "pre arg" and "post arg" are printed before after
      % the argument, if the argument is non-empty.
      % The command "\SemantexDelimiterSize" is substituted
      % by \big, \Big, ..., or whatever size the
      % argument delimiters have
    },
  },
}
```

See for yourself:

```
$\vk[free alg]{\vx}$,
$\vk[power series]{\vy}$,
$\vk[laurent]{\vz}$
```

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

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

```
\SetupClass\MyVar{
  define keys={
    {op}{upper={\mathrm{op}}},
      % opposite groups, rings, categories, etc.
    {alg closure}{command=\overline},
      % algebraic closure
    {conj}{command=\overline},
      % complex conjugation
    {dual}{upper=*},
      % dual vector space
    {perp}{upper=\perp},
      % orthogonal complement
  },
  define keys[1]={
    {mod}{right return,symbol put right={/#1}},
      % for modulo notation like R/I
    {dom}{left return,symbol put left={#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}{symbol put right={ \lbrack #1^{-1} \rbrack }},
      % localization at a multiplicative subset;
      % we use \lbrack and \rbrack rather than [ and ] since in some
      % cases (using constructions like in section 4.2),
      % the [...] might be interpreted as an optional argument.
    {localize prime}{sep lower={#1}},
      % for localization at a prime ideal
  },
}
```

Let us see it in practice:

```
$\vR[op]$, $\vk[alg closure]$,
$\vz[conj]$, $\vV[dual]$,
$\vR[mod=\vI]$,$\vR[dom=\vJ]$,
$\vR[oplus=\vn]$,
$\vV[tens=\vm]$,
$\vR[localize=\vS]$,
$\vR[localize prime=\vI]$,
$\vk[free alg]{\vS}[op]$,
$\vV[perp]$
```

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

3.3 GIT quotients

We include a slightly more advanced example to show the use of keys with more than one value. Sometimes, a key with one value is simply not enough. For instance, if you work in geometric invariant theory (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 the the key `define keys[2]`:

```
\SetupClass\MyVar{
  define keys[2]={
    {proj quotient}{ symbol put right={ /\!\!/_{#1} #2 } },
  }
}
```

```
$ \vX[proj quotient={\vchi}{\vG}] $
```

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

Chapter 4

Some more techniques

4.1 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 `left spar` and `right spar` keys:

```
$\vf[left spar={[},right spar={\}},
spar,spar=\Bigg]$
```

$$\Bigg[[f\}\Bigg\}$$

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 `other spar` and `Other spar` keys. They use the syntax

`other spar={`⟨*opening bracket*⟩`}{`⟨*closing bracket*⟩`}`
`Other spar={`⟨*opening bracket*⟩`}{`⟨*closing bracket*⟩`}{`⟨normal|auto|*|*other*⟩`}`

The last argument in `Other spar` sets the size of the parentheses. Let us see them in action:

```
$\vf[other spar={[}{)},
  other spar={\{}{\rangle},
  Other spar={\langle}{\rangle}{
  \Bigg},spar]$
```

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

4.2 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*⟩

4.3 The `command` key

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

```
$\va[command=\overline]$,
$\vH[command=\widetilde]$
```

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

This key applies the given command to the symbol. Sometimes, it is useful to put these commands into keys instead. So you can do stuff like

```
\SetupClass\MyVar{
  define keys={
    {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, $\bar{a}$

Note that there is a predefined key, `smash`, which is equivalent to `return, command=\smash`.

4.4 The `return` keys

Let us suppose in this section that we have defined the key `conj` for complex conjugation, like in the introduction. 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 key, called `return`, that can be applied at any point to invoke the routine of adding all current commands, indices, and arguments to the symbol. Let us try it out:

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

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

Before we invoked `return`, the symbol was `f`, and the `-1` was stored as an upper index. But after the `return` routine, the symbol is `f^{-1}`, and consequently, when we apply the `conj` key, you add a line above the whole thing.

There are some cases when you do not want to add all commands, indices, and arguments to the symbol at the same time. Therefore, there exist a few extra, partial `return` keys that only add some of them to the symbol and save the rest of later. We list the most important ones here and refer to section 15.3 for the remaining ones. Most users will probably only ever need the keys `return` and `right return`.

- `return`

  Invokes the return routine, i.e. adds all commands, indices, and arguments to the symbol, if any such exist.

- `inner return`

  Invokes the inner return routine, i.e. adds all commands to the symbol, if any such exist.

- `right return`

  Invokes the right return routine, i.e. adds all commands, right indices, and right arguments to the symbol, if any such exist.

- `left return`

  Invokes the left return routine, i.e. adds all commands, left indices, and left arguments to the symbol, if any such exist.

4.5 Keyval syntax conflicts

You can pass anything you want as key values, including other objects. But you quickly run into the following problem: Imagine you try setting `\vx[1,power=2]` as the lower index of a the object `\va`. If you try

```
$ \va[lower=\vx[1,power=2]] $
```

then the system will break. Indeed, the system will see the object `\va` to which you have passed the two keys

`lower=\vx[1` and `power=2]`.

To avoid this behaviour, you will have to enclose the key value in braces:

```
$ \va[lower={\vx[1,power=2]}] $
```

$a_{x_1^2}$

So far so good, but if you use our favourite shorthand notation for lower indices (simply writing the index in the options, like `\va[1]`), then it still goes wrong:

```
$ \va[{\vx[1,power=2]}] $
```

The reason is that in LaTeX (really, the `xparse` package from LaTeX3) interprets `[{...}]` more or less like `[...]` in this case. To make up for this, you can use either of the following strategies:

```
$ \va[ {\vx[1,power=2]} ] $,
$ \va[\vx[{1,power=2}]] $
```

$a_{x_1^2}, a_{x_1^2}$

There is a similar problem in the arguments, since arguments also allow a kind of keyval syntax (the keys that need equality signs are turned off by default, though; more on that in chapter 7). But it will still react on commas and keys like `...`. Therefore, in order to ensure the correct output, you will also have to enclose any argument containing commas with braces:

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

$f(g_2^3), f(g_2^3)$

As mentioned in chapter 7, you *can* also turn keyval syntax in arguments completely off, avoiding such issues. This can be done by setting

```
\SetupClass\MyVar{
  arg keyval=false,
}
```

Cheating your way around keyval syntax conflicts

If you grow tired of having to deal with such issues all the time, there are solutions to either partly or completely avoid this. The first solution we present does not solve the problem with `\va[\vx[1,power=2}]`, but it does solve problems like

```
$ \va[lower=\vx[lower=3]] $
```

Normally, this will not work, as the underlying keyval machinery of LaTeX3 does not allow key values to contain equality signs. However, there is another keyval package that does: the excellent package `expkv`. To switch to the keyval parser of this package, we do

```
\usepackage{expkv}
\SemantexSetup{
  keyval parser=\ekvparse,
}
```

Now you can do

```
$ \va[lower=\vx[lower=3]] $
```

a_{x_3}

In general, using the key `keyval parser={`⟨*command*⟩`}` sets the keyval parser function to be the command ⟨*command*⟩. The ⟨*command*⟩ must take three arguments: ⟨*command*⟩⟨*function*$_1$⟩⟨*function*$_2$⟩`{`⟨*key-value list*⟩`}`. The ⟨*function*$_1$⟩ must take one argument, while ⟨*function*$_2$⟩ must take two. For a key-value list, ⟨*function*$_1$⟩ will be applied to single keys taking no values, while ⟨*function*$_2$⟩ will be applied to keys taking a value. By default, this key has been set to the command `\keyval_parse:NNn` from LaTeX3. Changing this key will only affect keys for objects and classes, *not* keys for use inside `\SemantexSetup`.

A more drastic solution is to use the package `stricttex`, which has been developed mainly as a companion package to SemanTeX. Unfortunately, it only works in LuaTeX. If you don't know what LuaTeX is, that means that you are not using LuaTeX, and you should note that switching is a rather drastic affair since your existing font settings might very well not work with LuaTeX. Also, SemanTeX does not exactly make your document faster, and LuaTeX makes it even slower, so think carefully before you make the switch just for this.

In any case, with `stricttex`, you will be able to make brackets "strict", which means that any `[` will be replaced by a `[{`, and that any `]` will be replaced by a `}]`. This will make all of the above work just fine:

```
\StrictBracketsOn
$ \va[lower=\vx[lower=3]] $
$ \va[\vx[1,power=2]] $
$ \vf{ \vg[upper=3,lower=2] } $
\StrictBracketsOff
```

There is no demonstration on the right since this manual has not been typeset using LuaTeX, so it would not work.

Chapter 5

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{
  define keys={
    {comorphism}{upper=\#},
      % comorphisms, i.e. f^{\#}
    {inverse image}{upper={-1},no par},
      % inverse image of sheaves
    {sheaf pull}{upper=*,no par},
      % sheaf *-pullback
    {sheaf push}{lower=*,no par},
      % sheaf *-pushforward
    {sheaf !pull}{upper=!,no par},
      % sheaf !-pullback
    {sheaf !push}{lower=!,no par},
      % sheaf !-pushforward
  },
}
```

We have added the command `no par` 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[sheaf pull,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[sheaf push]{\sheafreg[\vX]} $,
and for a sheaf~$ \sheafF $ on~$
\vY $, we can define the
pullback~$ \vf[sheaf pull]{
\sheafF} $ by letting~$
\vf[sheaf pull]{\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 `inverse image` 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{
  define keys[1]={
    {stalk}{sep lower={#1}},
    % "sep lower" means "separator + lower", i.e. lower index
    % separated from any previous lower index by a separator,
    % which by default is a comma
    {sheaf twist}{return,symbol put right={(#1)}},
  },
  define keys={
    {sheafify}{upper=+},
    {sheaf dual}{upper=\vee},
  },
}
```

```
$\sheafF[res=\vU,stalk=\vp]$,
$\sheafF[res=\vU,spar,stalk=
\vp]$,
$\sheafreg[\vX,stalk=\vp]$,
$\sheafG[sheafify]$,
$\vf[inverse image]{\sheafreg[
\vY]}[spar,stalk=\vx]$,
$\sheafG[sheaf dual]$,
$\sheafreg[\vX][sheaf twist=-1]$,
$\sheafreg[sheaf twist=1,sheaf dual
]$
```

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

Chapter 6

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:

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

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

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{
  define keys[1]={
    {shift}{ right return,symbol put right={ \lbrack #1 \rbrack } },
      % we use \lbrack and \rbrack rather than [ and ] since in some
      % cases (using constructions like in section 4.2),
      % the [...] might 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}[no par]
```

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

$dx = 0$

6.1 The d-index and the i-index

In branches of mathematics such as homological algebra, 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. You can adjust this by writing `grading position=upper` or `grading position=lower`.

We already learned about the keys `upper` and `lower`, as well as their friends `sep upper`, `sep lower`, `comma upper`, `comma lower`, etc. There also exist “relative” versions of these 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

`d`, `sep d`, `comma d` and `i`, `sep i`, `comma i`,

and consequently, we shall refer to the indices they correspond to as the “`d`-index” and the “`i`-index”. The `d` stands for “degree” and corresponds to the grading. The `i` stands for “index” and corresponds to the “other” index where you may store additional information.[1]

To understand the difference, keep the following two examples in mind: the hom complex $\operatorname{Hom}^\bullet_A$ and the simplicial homology $H^\Delta_\bullet$:

```
\NewObject\MyVar\Hom{\operatorname{Hom}}
\NewObject\MyVar\ho{H}[grading position=lower] % homology
```

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

Hom^0_A, H^Δ_1

Let us see them in action:

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

\SetupObject\vX{
  grading position=lower
}

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

X^3_k
X^k_3

If you want to print a bullet as the degree, there is the predefined key `*` for this:

```
$ \vX[*] $

\SetupObject\vX{
  grading position=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` in the `i`-index. So changing the grading position changes the position of the index:

```
$ \vX[1] $

\SetupObject\vX{
  grading position=lower
}

$ \vX[1] $
```

X_1
X^1

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

---

[1] These names are not perfect; you might object 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.

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

Hom^0_A, H^Δ_1

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 actions of these keys are invoked. In fact, SemanTeX carefully separates keys taking a value from keys taking no values.

We can similarly define a command for cohomology:

```
\NewObject\MyVar\co{H}[grading position=upper]
    % this is actually unnecessary, as
    % upper grading is the default
```

Let us see `\ho` and `\co` in practise:

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

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

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

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\MyVar{
  define keys={
    {reduced}{command=\widetilde},
    {cech}{command=\check},
    {hyper}{command=\mathbb},
  },
}


$\co[reduced,d=\vi]$,
$\co[cech][*]$,
$\co[hyper,cech,d=0]{\vX}$
```

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

You can use a similar approach to define commands for derived functors:

```
\NewObject\MyVar\Lder{\mathbb{L}}[no par]
\NewObject\MyVar\Rder{\mathbb{R}}[no par]
```

For instance, we can write

```
$\Lder[d=\vi]{\vf}$,
$\Rder[d=0]{\vf}$
```

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

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{
  define keys[1]={
    {Lder} {
      left return, symbol put left=\mathbb{L}^{#1},
    },
    {Rder} {
      left return, symbol put left=\mathbb{R}^{#1},
    },
  },
  define keys={
    {Lder} {
      left return, symbol put left=\mathbb{L},
    },
    {Rder} {
      left return, symbol put left=\mathbb{R},
    },
  },
}
```

Then the syntax becomes:

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

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

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

```
\NewObject\MyVar\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:

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

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

Chapter 7

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. To define argument keys, we use the key `define arg keys`, or `define arg keys[n]` if you want it to be able to take n values for $n = 0, 1, 2, \ldots, 9$. The syntax for these is just like the syntax for the keys `define keys` and `define keys[n]`. However, for reasons we shall see in a moment, argument keys (at least those taking values) are actually turned off by default, so we shall have to turn them on first:

```
\SetupClass\MyVar{
  define arg keys[1]={
    {coef}{ other sep={;}{#1}  },
  },
}
\SetupObject\co{
  arg keyval=true,
}
```

The key `other 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. We shall later (see section 12.2) see another, possibly more natural way to write cohomology with coefficients, and which avoids turning on keyval syntax in the argument.

As mentioned, we had to turn keyval syntax on in order for it to work. By default, only keys taking no values are turned on in the argument. The reason is that argument keys taking values are only useful in very rare cases, such as 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 `arg keyval` can take four arguments: `true` (which we used above, keyval syntax is completely on), `false` (no keys allowed), `single keys` (the default behaviour where only keys taking no values are allowed), and `one single key` (only allows one key, taking no value).

It should be noted that there are several predefined argument keys on the level of the class `\SemantexBaseObject`. The full list can be found in section 15.13.

Chapter 8

Left indices

Left indices are a recurring problem in all TEX-based systems since TEX only has metrics for the positioning of right indices, none for left indices. And it seems that even the later TEX engines are making no attempts at correcting this. So most packages for left indices use variations of the following approach:

```
$ {}^{*} f $
```

${}^{*}f$

Notice the large space between the star and the f. To tackle this problem, the author has written the `leftindex` package which at least attempts to improve this situation:

```
$ \leftindex^{*} {f} $
```

${}^{*}\!f$

Roughly, what it does is to use a “height phantom” and a “slanting phantom” to position the left superscript. The vertical positions of the left indices will be calculated using the height phantom, and the indentation of the left superscript will be calculated using the slanting phantom. More precisely, it will copy the metrics for the positioning of right indices from the slanting phantom and use that to position the left superscript. By default, both phantoms are set to be equal to the symbol, which goes fine sometimes, and at other times, another slanting phantom has to be specified. Below, the `I` is the specified, custom slanting phantom:

```
$ \leftindex^{*} {\Gamma} $,
$ \leftindex[I]^* {\Gamma} $,
$ \leftindex^* {A} $,
$ \leftindex[P]^* {A} $
```

${}^{*}\!\mathit{\Gamma}$, ${}^{*}\mathit{\Gamma}$, ${}^{*}A$, ${}^{*}\!A$

We refer to the manual of the package `leftindex` for details, see

`https://ctan.org/pkg/leftindex`

Our solution for left indices in SemanTEX is based directly on the one from `leftindex`. However, it works much better if you use SemanTEXthan if you just used `leftindex` alone, due to the ability to centrally control all your notation. This allows you to choose height and slanting phantoms once and for all in the preamble and never have to worry about it in your document body.

Just like we have the keys `upper`, `lower`, `sep upper`, `sep lower`, `comma upper`, `comma lower`, we have a similar collection of keys for left indices: `upper left`, `lower left`, `sep upper left`, `sep lower left`, `comma upper left`, `comma lower left`:

```
$ \vf[upper left=*] $,
$ \vGamma[upper left=*] $,
$ \vA[upper left=*] $
```

${}^{*}\!f$, ${}^{*}\!\mathit{\Gamma}$, ${}^{*}A$

When you create a new object in SemanTEX, the height and slanting phantoms will automatically be set to be equal to the symbol. However, as we see above, we sometimes need to change them. This can be done using the keys `height phantom` and `slanting phantom`:

```
\SetupObject\vGamma{
  slanting phantom=I}
\SetupObject\vA{slanting phantom=P}
$ \vf[upper left=*] $,
$ \vGamma[upper left=*] $,
$ \vA[upper left=*] $
```

${}^*f$, ${}^*\Gamma$, ${}^*A$

Sometimes, changing the slanting phantom is not quite enough. In the previous example, the star is still not quite close enough to the A, and there is no slanting phantom that is quite slanted enough to correct this. We solve this using the key `post upper left`. What you add using this key will be printed after the upper left index, provided the upper left index is non-empty and hence will be printed in the first place. There is also a `pre upper left`, and there are similarly `pre lower left`, `post lower left`, `pre upper`, `post upper`, `pre lower`, and `post lower`. Let us see it in action:

```
\SetupObject\vA{
  slanting phantom=P,
  post upper left=\!,
}
$ \vA[upper left=*] $
```

${}^*\!A$

Note that SemanTEX at least does its best to try to guess new height and slanting phantoms when you use operations on objects:

```
$ \vA[spar=\Bigg,upper left=*] $,
$ \vP[command=\overline,return,
  upper left=*] $
```

${}^*\Biggl(A\Biggr)$, ${}^*\overline{P}$

Chapter 9

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

SemanTEX has facilities for printing tensor products $\otimes$ as well as derived tensor products $\otimes^L$. 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 `Symbol` 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 think. Let us try to use it to define tensor products and fibre products:

```
\NewSymbolClass\MyBinaryOperator[
  define keys={
    {Lder}{upper=L},
    {Rder}{upper=R},
  },
]

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

\NewObject\MyBinaryOperator\fibre{\times}[
  % Americans are free to call it \fiber instead
  define keys={
    {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:

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

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

Later (in section 12.4), we shall see another, more advanced solution for binary operators which also allows us to express the application of the operator on n elements.

Chapter 10

Paired delimiters

In this chapter, se show how to define delimiter commands like $\|-\|$ and $\langle -,-\rangle$. This is easy to do via the keys `left par` and `right par`:

```
\NewObject\MyVar\norm[left par=\lVert, right par=\rVert]
\NewObject\MyVar\inner[left par=\langle, right par=\rangle]
```

Indeed:

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

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

In the case where you want to use different kinds of norms, say $\|-\|_2$ or $\|-\|_\infty$, you can use the key `output options={⟨options⟩}`. This allows you to pass the ⟨*options*⟩ to the output class (in this case, `\MyVar`):

```
\SetupObject\norm{
  define keys[1]={
    {default}{ output options={ default={#1} } },
  },
}
```

```
$\norm{\vx}$,
$\norm[2]{\vx}$,
$\norm[\infty]{\vx}$
```

$\|x\|$, $\|x\|_2$, $\|x\|_\infty$

We can also create 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` construction:

```
\newcommand\wherecommand[1]{%
  \nonscript\:%
  #1\vert
  \allowbreak
  \nonscript\:%
  \mathopen{}%
}

\NewObject\MyVar\where{ \wherecommand{\SemantexDelimiterSize} }

\NewObject\MyVar\Set[
  left par=\lbrace, right par=\rbrace,
```

```
  pre arg={\,},post arg={\,},
    % adds \, inside {...}, as recommended by D. Knuth
  arg keyval=false,
    % this turns off all keyval syntax in the argument
]
```

As we briefly mentioned previously, `\SemantexDelimiterSize` is a command that returns the size of the delimiters in the argument. Now you can use

```
$\Set{ \vx\in\vY \where \vx\ge0 }$,
$\Set[par=\big]{ \vx\in\vY \where
\vx\ge0 }$
```

$\{x \in Y \,|\, x \geq 0\}$, $\big\{x \in Y \,\big|\, x \geq 0\big\}$

Don't forget that, because we called `output=\MyVar` in the beginning of this manual, the output of any of these commands also belongs to class `\MyVar`. So you can do stuff like

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

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

Tuple-like commands are also possible:

```
\NewObject\MyVar\tup[left par=(,right par=)] % tuples
\NewObject\MyVar\pcoor[ % projective coordinates
  left par={[}, right par={]},
  set arg sep=\mathbin{:},
    % changes the argument separator to colon
  set arg dots=\dotsb,
    % changes what is inserted if you write "..."
]
```

Let us see them in action:

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

$(a,b,\dots,z)$, $[a : b : \cdots : z]$

One can use similar techniques for other, less obvious purposes, like calculus differentials:

```
\NewVariableClass\CalculusDifferential[
  parent=\MyVar,
  define arg keys[1]={
    {default}{sep={d\!#1}},
    % default is the key that is automatically applied by the
    % system to anything you write in the argument that is
    % not recognized as an argument key. The sep key
    % is a key that prints the value of the key with the
    % standard argument separator in front.
  },
  set arg dots=\dotsm,
  never par,
  % never par is like no par, except no par will still print
  % parentheses when there is more than one argument
  % -- never par does not even print parentheses in this case
]

\NewObject\CalculusDifferential\intD[
```

```
  set arg sep={\,},
  next arg with sep=true,
    % because of this, even the first argument will
    % receive a separator, which in this case
    % is a small space
]

\NewObject\CalculusDifferential\wedgeD[set arg sep=\wedge]
```

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

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

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

Chapter 11

Using SemanTeX in other commands using `\UseClassInCommand`

Sometimes, it is useful to create other commands based on SemanTeXclasses. For instance, if you grow tired of writing `\MyVar{ \frac{...}{...} }` whenever you want to apply keys to a fraction, it could make sense to create a command `\Frac` which automatically wraps the fraction in `\MyVar`. The first guess how to do that would be something like

```
\newcommand\Frac[2]{%
  \MyVar{\frac{#1}{#2}}%
}
```

```
\[
  \Frac{1}{2}[spar=\Big,power=2]
\]
```

$$\Big(\frac{1}{2}\Big)^2$$

Indeed, this will work fine for most people. In fact, the only case where this might cause issues is if you want to use the `stripsemantex` algorithm to strip your document of SemanTeX markup. But in order to prepare yourself for this possibility, I recommend getting used from the start to doing it in a slightly more cumbersome way:

```
\SemantexRecordObject{\Frac}
\newcommand\Frac[2]{%
  \SemantexRecordSource{\Frac{#1}{#2}}%
  \UseClassInCommand\MyVar{\frac{#1}{#2}}%
}
```

```
\[
  \Frac{1}{2}[spar=\Big,power=2]
\]
```

$$\Big(\frac{1}{2}\Big)^2$$

First things first: We used the following command in front of `\MyVar`:

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

So the first advantage to writing `\UseClassInCommand\MyVar` instead of just `\MyVar` is that you can pass an additional set of options to the class first. However, there is a more important difference, namely that this solution makes the command compatible with the `stripsemantex` algorithm.

The reason the first solution was not compatible with `stripsemantex` is that, in this case, the algorithm will desperately look through your document for the code `\MyVar{ \frac{1}{2} }[spar=\Big,power=2]` in order to strip it from your document. But it will find it nowhere, as this code is hidden away in the `\Frac` command. Therefore, we do three things:

- We register the command `\Frac` as a SemanTEX command using the line

  ```
  \SemantexRecordObject{\Frac}
  ```

  After this, SemanTEX "knows" that `\Frac` is part of the family of SemanTEX markup.

- We use the command `\SemantexRecordSource` to "record" the source of the command internally. This way, `stripsemantex` will know what to look for when it moves through the document, trying to strip it of SemanTEX markup. It is therefore important that you record the source exactly like it will be written in the source. (You need not worry about missing braces, though; even if you write `\Frac12` in your document, `stripsemantex` will still recognize the code and strip it as expected.)

- We write `\UseClassInCommand\MyVar` instead of just `\MyVar` in order to correctly record the output code internally. Roughly speaking, when you use the command `\UseClassInCommand`, SemanTEX "knows" that the class `\MyVar` is now used as part of some greater construction.

11.1 Example: Category theory

The above method can be used to create commands for typing categories. First and foremost, it is easy to create objects corresponding to simple categories like Set, Top and Vect:

```
\newcommand\categoryformat[1]{{\operatorname{\mathsf{#1}}}}
  % This means that we write categories with sans-serif fonts;
  % -- but you can change this to your own liking.
  % We use \operatorname since it will allow us to use ordinary
  % dashes (rather than minuses) in math mode.
  % We use an extra pair of braces around \operatorname
  % in order to make it an ordinary symbol (instead of an operator).
  % This solution is inspired by an answer by egreg (obviously),
  % see https://tex.stackexchange.com/a/567886/19809

\NewObject\MyVar\catset{\categoryformat{Set}}
\NewObject\MyVar\cattop{\categoryformat{Top}}
\NewObject\MyVar\catvect{\categoryformat{Vect}}
```

```
$ \catset $,
$ \cattop $,
$ \catvect{\vk} $.
```

Set, Top, Vect(k).

However, we run into issues with categories like R-mod where we shall constantly have to change the ring R. For this, we use the constructions we learned at the introduction to this chapter:

```
\SemantexRecordObject{\catxmod}
\newcommand\catxmod[1]{%
  \SemantexRecordSource{\catxmod{#1}}%
  \UseClassInCommand\MyVar{#1\categoryformat{-mod}}%
}
```

```
$ \catxmod{\vR} $,
$ \catxmod{\vS} $,
$ \catxmod{\vA}[spar,op] $
```

R-mod, S-mod, $(A\text{-mod})^{\mathrm{op}}$

(here, we used the key op which we defined in section 3.2). You can, of course, extend it to all sorts of other situations, like mod-R or R-mod-S:

```
\SemantexRecordObject{\catmodx}
\newcommand\catmodx[1]{%
  \SemantexRecordSource{\catmodx{#1}}%
  \UseClassInCommand\MyVar{\categoryformat{mod-}#1}%
}

\SemantexRecordObject{\catxmody}
\newcommand\catxmody[2]{%
  \SemantexRecordSource{\catxmody{#1}{#2}}%
  \UseClassInCommand\MyVar{#1\categoryformat{-mod-}#2}%
}

\SemantexRecordObject{\catxmodx}
\newcommand\catxmodx[1]{%
  \SemantexRecordSource{\catxmodx{#1}}%
  \UseClassInCommand\MyVar{#1\categoryformat{-mod-}#1}%
}
```

Chapter 12

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 collection of data in any order, and have the system take care of printing them in the right places, according to how you program the object in the preamble. For this purpose, we have the `parse` routine. Using the `parse` routine allows for a comfortable, HTML-like syntax, e.g.:

```
$ \GL[order=\vn,field=\vk] $,
$ \Mat[rows=\vm,columns=\vn,
    field=\vk] $,
$ \co[d=0,coef=\vR,space=\vX] $
```

$\operatorname{GL}_n(k)$, $\operatorname{Mat}_{m\times n}(k)$, $H^0(X;R)$

The `parse` routine is a collection of code which is executed right before an object (or class) is being rendered (but before it outputs). By default, the parse routine contains no code. However, you can add code to it using the key `parse options={`⟨*keys*⟩`}`.

Even though the `parse` routine is automatically invoked right before rendering, you can also invoke it at any time by force using the key `parse`. This will also empty the code from the `parse` routine so that it will not be executed twice:

```
$ \GL[order=\vn,field=\vk,parse,
    spar,op] $
```

$(\operatorname{GL}_n(k))^{\mathrm{op}}$

(here we used the key `op` from section 3.2). Alternatively, you can add a new pair of brackets, which will render the object and invoke the `parse` routine:

```
$\GL[order=\vn,field=\vk][spar,op]$
```

$(\operatorname{GL}_n(k))^{\mathrm{op}}$

Note, though, that in this case, the `spar` and `op` keys are not being applied to the object `\GL` itself, but to the object (of class `\MyVar`) that it outputs. This should not cause any issues in practice, as long as the keys you are using are already defined on the level of `\MyVar`.

12.1 Example: Matrix sets and groups

Suppose we want to be able to write the group of invertible $n\times n$-matrices with entries in k as $\operatorname{GL}_n(k)$. We can in principle do the following:

```
\NewObject\MyVar\GL{\operatorname{GL}}
```

```
$ \GL[\vn]{\vk} $.
```

$\mathrm{GL}_n(k)$.

However, this is not quite as systematic and semantic as we might have wanted. Indeed, what if later we would like to change the notation to $\mathrm{GL}(n,k)$? We could in principle use a key with 2 values for this. However, in this section, we show how to use the `parse` routine to enable the syntax from the introduction to this chapter.

As mentioned there, we need to add code via the `parse` routine. However, to make proper use of it, we need some programming keys and programming commands. You can find an overview of these in sections 15.2 and 15.14.

To set up the notation from above, we do the following:

```
\NewObject\MyVar\GL{\operatorname{GL}}[
  % We provide a few data sets:
  data provide=order, % The "order" will be the number n in GL_n(k)
  data provide=field, % The "field" is of course the k in GL_n(k)
  define keys[1]={
    {order}{ data set={order}{#1} }, % Sets the order
    {field}{ data set={field}{#1} }, % Sets the field
    {arg}{ field={#1} },
      % This way, setting the argument becomes equivalent
      % to setting the field
  },
  parse options={
    set keys x={
      % This means set the keys, but fully expand their values first
      lower={\SemantexDataGetExpNot{order}},
    },
    if blank F={\SemantexDataGetExpNot{field}}
    {
      set arg keys x={
        % Set the argument keys, but fully expand their values first
        sep={\SemantexDataGetExpNot{field}},
      },
    },
  },
]
```

Notice that we changed the `arg` key. This means that specifying the argument becomes equivalent to setting the field. This is what makes the first two pieces of syntax below equivalent:

```
$ \GL[order=\vn,field=\vk] $,
$ \GL[order=\vn]{\vk} $,
$ \GL[order=\vn] $.
```

$\mathrm{GL}_n(k)$, $\mathrm{GL}_n(k)$, GL_n.

Let us look at a more complicated example: The set $\mathrm{Mat}_{n\times m}(k)$ of $n\times m$-matrices with entries in k. What makes this example more complicated is not only that we have an additional piece of data, but that we require that if the number of rows and columns are equal, we want it to print $\mathrm{Mat}_n(k)$ rather than $\mathrm{Mat}_{n\times n}(k)$. We accomplish this by the following:

```
\NewObject\MyVar\Mat{\operatorname{Mat}}[
  % We provide data sets "rows" and "columns" to
  % be set up by the user later
  data provide={rows},
  data provide={columns},
```

```
  data provide={field},
  define keys[1]={
    {rows}{ data set={rows}{#1} }, % set the rows data set
    {columns}{ data set={columns}{#1} }, % set the columns data set
    {field}{ data set={field}{#1} }, % set the underlying field
    {arg}{ field={#1} },
      % this way, setting the argument becomes equivalent
      % to specifying the underlying field
  },
  parse options={ % Here we add code to the parse routine
    % We check whether columns = rows. If so, we only write
    % the number once
    str if eq TF={\SemantexDataGetExpNot{columns}}{
    \SemantexDataGetExpNot{rows}}
    {
      set keys x={
        lower={\SemantexDataGetExpNot{columns}},
      },
    }
    {
      set keys x={
        lower={
          \SemantexDataGetExpNot{rows}
          \times
          \SemantexDataGetExpNot{columns}
        },
      },
    },
    if blank F={\SemantexDataGetExpNot{field}}
    {
      set arg keys x={
        sep={\SemantexDataGetExpNot{field}},
      },
    },
  },
]
```

```
$ \Mat[rows=\vm,columns=\vn,
  field=\vk] $,
$ \Mat[rows=\vn,columns=\vn,
  field=\vk] $.
```

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

12.2 Example: Cohomology with coefficients, revisited

As promised previously, we revisit cohomology with coefficients and show how to set up a syntax like the below:

```
\SetupObject\co{
  data provide=coefficient,
  data provide=space,
  define keys[1]={
    {coef}{ data set={coefficient}{#1} },
    {space}{ data set={space}{#1} },
    {arg}{ space={#1} },
  },
```

```
  parse options={
    if blank F={\SemantexDataGetExpNot{space}}
    {
      set arg keys x={
        sep=\SemantexDataGetExpNot{space},
      },
    },
    if blank F={\SemantexDataGetExpNot{coefficient}}
    {
      set arg keys x={
        other sep={;}{ \SemantexDataGetExpNot{coefficient} },
      },
    },
  },
}
```

```
$\co[d=0]$,
$\co[d=0,space=\vX]$,
$\co[d=0,space=\vX,coef=\vR]$
```

H^0, $H^0(X)$, $H^0(X;R)$

12.3 Example: Partial derivatives

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

```
\NewObject\MyVar\partialdif[
  no par,
  bool provide={raise function},
  bool set true={raise function},
  set i dots=\dotsm,
  set i sep={\,},
  define keys[1]={
    {default}{
      sep i={\partial #1},
    },
    {raise}{
      str if eq TF={#1}{true}
      {
        bool set true={raise function},
      }
      {
        str if eq TF={#1}{false}
        {
          bool set false={raise function},
        }
        {
          ERROR key value not found={raise}{#1},
        },
      },
    },
  },
  parse options={
    if blank TF={ \SemantexDataGetExpNot{upper} }
    {
```

```
  int if greater TF={ \SemantexIntGet{number of lower indices} }
   { 1 }
  {
    set keys x={
      symbol={
        \SemantexExpNot\frac
        {
          \partial ^ { \SemantexIntGet{number of lower indices}
          }
          \SemantexBoolIfT{raise function}
          {
            \SemantexDataGetExpNot{arg}
          }
        }
        {
          \SemantexDataGetExpNot{lower}
        }
      },
    },
  }
  {
    set keys x={
      symbol={
        \SemantexExpNot\frac
        {
          \partial
          \SemantexBoolIfT{raise function}
          {
            \SemantexDataGetExpNot{arg}
          }
        }
        {
          \SemantexDataGetExpNot{lower}
        }
      },
    }
  },
}
{
  set keys x={
    symbol={
      \SemantexExpNot\frac
      {
        \partial ^ { \SemantexDataGetExpNot{upper} }
        \SemantexBoolIfT{raise function}
        {
          \SemantexDataGetExpNot{arg}
        }
      }
      {
        \SemantexDataGetExpNot{lower}
      }
    },
  },
},
data clear={lower},
```

```
    data clear={upper},
    bool if T={raise function}
    {
      data clear={arg},
      int clear={number of arguments},
    },
  },
]
```

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:

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

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

12.4 Example: Smart binary operators

In chapter 9, we saw a simple solution for binary operators using the `Symbol` class. But that solution only printed the operator itself. However, in semantic markup systems, it is ofen desirable to also be able to explicitly typeset applications of the operator on n arguments, like this:

```
Applying the tensor product~$
\tensor$ to $\vn$~elements, we get
\[
  \tensor{ \vx[1], \vx[2], ...,
    \vx[\vn] }.
\]
Applying the multiplication
operator~$\mult$ to $\vn$~elements,
 we get
\[
  \mult{ \vx[1], \vx[2], ...,
    \vx[\vn] }.
\]
```

Applying the tensor product $\otimes$ to n elements, we get

$$x_1 \otimes x_2 \otimes \cdots \otimes x_n.$$

Applying the multiplication operator $\cdot$ to n elements, we get

$$x_1 \cdot x_2 \cdot \cdots \cdot x_n.$$

This can be accomplished using the `parse` routine as follows:

```
\NewVariableClass\MyBinaryOperator[
  set arg dots=\dotsb,
  never par,
  prepend keys[1]={
    {arg}{
      return,
      set keys x={
        set arg sep=\SemantexDataGetExpNot{symbol},
      },
    },
  },
  parse options={
    int if greater T={ \SemantexIntGet{number of arguments} } { 0 }
    {
      symbol={},
      output=\MyVar,
    },
  },
]

\NewObject\MyBinaryOperator\tensor{\otimes}
\NewObject\MyBinaryOperator\mult{\cdot}
```

In mathematical texts, the multiplication symbols in that last equation would usually be omitted. However, if we want to make the syntax fully semantic, we should also type these multiplication operators explicitly, even if they will be invisible in the final output. We do this by changing the argument dots and creating an object with an empty symbol. Alternatively, if we want to insert a space between the variables being multiplied, we can set the symbol to `\,`:

```
\NewVariableClass\InvisibleBinaryOperator[parent=\MyBinaryOperator,
    set arg dots=\dotsm]
\NewObject\InvisibleBinaryOperator\invmult{}
\NewObject\InvisibleBinaryOperator\spacemult{\,}
```

```
In mathematical texts, we denote
the product of \( \vn \)~elements
simply by
\[
  \invmult{ \vx[1], \vx[2], ...,
    \vx[\vn] }
  =
  \spacemult{ \vx[1], \vx[2], ...,
    \vx[\vn] }.
\]
```

In mathematical texts, we denote the product of n elements simply by

$$x_1x_2\cdots x_n = x_1\,x_2\,\cdots\,x_n.$$

Chapter 13

`stripsemantex` – stripping your document of SemanTEX markup

SemanTEX is a big, heavy package, and it might raise eyebrows if you try using it in submissions to journals. On top of that, `arXiv.org` is using TEX Live 2016 at the time of writing this, and it has an old version of LATEX3 that seems unable to run SemanTEX. To address this issue, SemanTEX has a companion package, called `stripsemantex`, which allows you to strip the SemanTEX markup from your document and replace it with raw LATEX code. While no such algorithm will ever be perfect, it generally works very well, even for quite complicated constructions, as long as you use the package in the "normal" and supported way. (If you want proof, have a look at my recent paper which was stripped using the algorithm: `https://arxiv.org/abs/2008.04794`.)

The system has the following limitations:

- It is currently only able to strip the SemanTEX markup from your main document (so it will ignore anything in `\input{...}` and `\include{...}`). So prior to running `stripsemantex`, you should include your entire document body in your main `.tex` file.
- Partly because of the previous point, no attempt is made to remove the *setup* of SemanTEX, so commands like `\NewObject`, `\SetupObject`, and `\SetupClass` will remain in the document body. You will then have to remove these yourself afterwards. But the SemanTEX markup itself should be stripped completely from your document.
- As mentioned, as long as you do normal, supported things, everything should work fine. Non-normal, non-supported things are tings like

  ```
  \va[execute={\vb}]
  ```

- Things might go wrong if you define new keys between `\begin{document}` and `\end{document}` whose definitions make use of other SemanTEX objects or classes, since the algorithm will try to strip these from the definitions. For instance, don't do stuff like this after `\begin{document}`:

  ```
  \SetupObject\va{
    define keys[1]={
      {weirdkey}{ upper=\vb[ {#1} ] }
    },
  }
  ```

If you do, the algorithm will then try and strip this occurrence of `\vb` from the key definition. To avoid such issues, only ever define keys in your preamble, as the algorithm will ignore everything before `\begin{document}`.

- When the document has just been stripped, it will load a small package called `semtex`, which contains a couple of commands that the output will need in order to run. You will be able to replace all of these commands by other commands and then render the package `semtex` unnecessary. More on this in section 13.1.
- When SemanTEX runs, the content of any argument is being wrapped between `\begingroup` and `\endgroup`. This is part of what makes it possible to use the command `\SemantexDelimiterSize`. However, these `\begingroup` and `\endgroup` will not appear in the stripped document. This means that if you do stuff like

  ```
  $ \def\foo{bar} \va{ \def\foo{barbar} \vx } \foo $
  ```

  then this will print $a(x)bar$ before running `stripsemantex`, but $a(x)barbar$ after. In order to avoid this, simply don't define commands inside arguments, which you should never do in the first place (and why would you anyway?).

As a small proof of concept, this is what the example in the introduction would look like when stripped of SemanTEX markup:

```
% Same preamble as before.

\begin{document}

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

$ g^{-1}|_{U} (x) $

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

\end{document}
```

Yes, I know, this was a very simple, unconvincing example. If you want a less trivial example, as mentioned before, you can have a look at my latest paper, which was stripped with (a previous alpha version of) `stripsemantex`:

`https://arxiv.org/abs/2008.04794`

13.1 The `semtex` package

When you have stripped your document and removed all SemanTEX package setup, it should be safe to remove the loading of SemanTEX from your preamble. However, the stripping algorithm will automatically add the following lines to your document right before `\begin{document}`:

```
% The following was added by "stripsemantex":

\usepackage{semtex,leftindex,graphicx}

\providecommand\SemantexLeft{%
  \mathopen{}\mathclose\bgroup\left
```

```
}

\providecommand\SemantexRight{%
  \aftergroup\egroup\right
}

\makeatletter
\DeclareRobustCommand\SemantexBullet{%
  \mathord{\mathpalette\SemantexBullet@{0.5}}%
}
\newcommand\SemantexBullet@[2]{%
  \vcenter{\hbox{\scalebox{#2}{$\m@th#1\bullet$}}}%
}
\DeclareRobustCommand\SemantexDoubleBullet{\SemantexBullet
\SemantexBullet}
\makeatother
```

The package `leftindex` is loaded to take care of any possible left indices. The package `graphicx` is loaded to provide the command `\scalebox`. This package `semtex` is a small package whose sole purpose is to be loaded by stripped SemanTEX documents. All it does is define the four commands `\SemantexLeft`, `\SemantexRight`, `\SemantexBullet`, and `\SemantexDoubleBullet` so that you can remove these definitions from your document and just rely on the package instead.

Let us take a look at the commands defined by `semtex`:

- `\SemantexBullet`, `\SemantexDoubleBullet`

  The commands that contain the bullets we use in SemanTEX, i.e. the superscript in $H^\bullet$. These bullets are smaller (and prettier, in my opinion) than the standard `\bullet` command from LATEX.

- `\SemantexLeft`, `\SemantexRight`

  Like `\left ...\right`, but fixing some spacing issues around these. They are completely equivalent to `\mleft` and `\mright` from the package `mleftright`, so it is safe to just load that package and replace the above commands by `\mleft ...\mright` instead, or use the redefinitions mentioned above.

13.2 The `stripsemantex` algorithm

The stripping algorithm works like this. It will work in any TEX engine (pdfTEX, XƎTEX, LuaTEX, etc.), but along the way, you will have to create a small, separate document and compile it with LuaTEX. Suppose in the following that you TEX document is called `mydoc.tex`.

(1) Make sure to collect all of the SemanTEX markup you want stripped in the main document, `mydoc.tex`. Also make sure to follow the recommendations in chapter 11, in case you have created commands of the form described there.

(2) Put the following somewhere in your preamble, after the loading of SemanTEX:

```
\SemantexSetup{semtexfile=true}
```

(3) Compile your document `mydoc.tex` using your preferred TEX engine (pdfTEX, XƎTEX, LuaTEX, or whatever). Because of the previous step, there will now be a new file, `mydoc.semtex`, in your folder, where the raw output of each SemanTEX command is stored. In a moment, `stripsemantex` will use this information to replace each command by the raw code it outputs.

(4) Create another TEX document in the same folder as `mydoc.tex`, and call it `stripdoc.tex` (or whatever you want). Put the following into it:

```
\documentclass{article}

\usepackage{stripsemantex}

\begin{document}

\StripSemantex{mydoc}

\end{document}
```

Then compile it **with LuaTEX**.

After this step, another document will have been created in the same folder, called `mydoc_prestripped.tex`. It will look just like `mydoc.tex`, but in the document body, each command defined using SemanTEX will now have an expression of the form `\SemantexID{`⟨*a unique ID*⟩`}` preceding it.

(5) Compile the document `mydoc_prestripped.tex` using the same TEX engine as the one you used for `mydoc.tex`.

(6) Compile the document `stripdoc.tex` again, this time also **using LuaTEX**.

(7) After the previous step, some (but usually not all) SemanTEX markup will have been removed from the file `mydoc_prestripped.tex`. If the stripping algorithm has terminated (which it almost never does after a single run), there will now be a new document in your folder, called `mydoc_stripped.tex`. If this document is not there, repeat the steps (5) and (6).

Continue this way until the file `mydoc_stripped.tex` appears. It can easily require three or more iterations, but each iteration will usually be faster than the previous one, and eventually, the file `mydoc_stripped.tex` will appear. (Note that at the point (6), `stripsemantex` will also issue a warning if the algorithm has not yet terminated, asking you to repeat the steps (5) and (6)).

Note again that your SemanTEX **setup** will not be removed, so there will still be commands like `\NewObject`, `\SetupObject`, `\SetupClass`, etc. left. You will then have to remove these few commands from your document manually.

13.3 Stripping comments from the document

Apart from the machinery for stripping SemanTEX markup from documents, the package `stripsemantex` also provides the command `\StripSemantexStripComments`, which is in principle completely unrelated to SemanTEX itself. This command allows you to strip all comments between `\begin{document}` and `\end{document}`. If your document is again called `mydoc.tex`, you can create the following document and compile it **with LuaTEX**:

```
\documentclass{article}

\usepackage{stripsemantex}

\begin{document}

\StripSemantexStripComments{mydoc}

\end{document}
```

This will create a new document, called `mydoc_comments_stripped.tex`, where all comments in the document body have been removed.

Chapter 14

Known bugs

If you write e.g. `Other spar={[}{]}{\Bigg}` in a heading, your command will fail for some reason. It can be solved by omitting the braces around `\Bigg`, i.e. by replacing it by `Other spar={[}{]}\Bigg`.

Chapter 15

The predefined keys, commands, and data

In this chapter, we give a complete list of the predefined keys. Firstly, the keys that can be used inside the command `\SemantexSetup` are:

- `keyval parser={`⟨*command*⟩`}`

  Sets the keyval parser function to ⟨*command*⟩. The ⟨*command*⟩ must take three arguments: ⟨*command*⟩⟨*function*$_1$⟩⟨*function*$_2$⟩`{`⟨*key-value list*⟩`}`. The ⟨*function*$_1$⟩ must take one argument, while ⟨*function*$_2$⟩ must take two. For a key-value list, ⟨*function*$_1$⟩ will be applied to single keys taking no values, while ⟨*function*$_2$⟩ will be applied to keys taking a value. By default, this key has been set to the LaTeX3 command `\keyval_parse:NNn`. Another interesting possibility is the command `\ekvparse` from the package `expkv`. This choice will only affect keys for objects and classes, *not* keys for use inside `\SemantexSetup`.

- `single key parser={`⟨*command*⟩`}`

  Sets the single key parser function to ⟨*command*⟩. The single key parser is the command that parses the content of the argument when you have applied the setting `arg keyval=single keys`. The ⟨*command*⟩ must take two arguments: ⟨*command*⟩⟨*function*⟩`{`⟨*comma list*⟩`}`. The ⟨*function*⟩ must take one argument and will be applied to each entry in the ⟨*comma list*⟩. By default, this key has been set to the LaTeX3 command `\clist_map_function:nN` (but with the arguments in reverse order).

- `semtex file={`⟨true|false⟩`}`

  When turned on, a `.semtex` file will be created while processing the document. This is mainly relevant when using `stripsemantex`.

Apart from this, SemanTeX has a large collection of keys that are predefined for the class `\SemantexBaseObject`. In the following sections, we include the full list.

15.1 Keys for defining and removing keys

- `define keys={`⟨*key definitions*⟩`}`

  Defines keys taking no values. The syntax is

```
define keys={
  {key1}{ upper=3, lower=7 },
  {key2}{ lower=6, upper=4 },
},
```

- `define keys[`n`]={`⟨*key definitions*⟩`}`

  Defines keys taking n values, where $n = 0, 1, 2, \ldots, 8$. The values are accessed by writing `#1`, `#2`, …, `#8`. For technical reasons, nine arguments are not allowed. The syntax is

  ```
  define keys[2]={
    {key1}{ upper=3+#1, lower=7-#2 },
    {key2}{ lower=6\cdot#1, upper=4/#2 },
  },
  ```

- `append keys={`⟨*key definitions*⟩`}`

  Appends keys taking no values, i.e. adds code to the right of that key. The syntax is identical to the one for `define keys`.

- `prepend keys={`⟨*key definitions*⟩`}`

  Prepends keys taking no values, i.e. adds code to the left of that key. The syntax is identical to the one for `define keys`.

- `append keys[`n`]={`⟨*key definitions*⟩`}`

  Appends keys taking n values, where $n = 0, 1, \ldots, 8$, i.e. adds code to the right of that key. The syntax is identical to the one for `define keys[`n`]`.

- `prepend keys[`n`]={`⟨*key definitions*⟩`}`

  Prepends keys taking n values, where $n = 0, 1, \ldots, 8$, i.e. adds code to the left of that key. The syntax is identical to the one for `define keys[`n`]`.

- `remove key=`⟨*key name*⟩

  Removes the key ⟨*key name*⟩ taking no values.

- `remove key[`n`]=`⟨*key name*⟩

  Removes the key ⟨*key name*⟩ taking n values, where $n = 0, 1, 2, \ldots, 8$.

- `define arg keys={`⟨*key definitions*⟩`}`

  Defines argument keys taking no values. The syntax is similar to the one for `define keys`.

- `define arg keys[`n`]={`⟨*key definitions*⟩`}`

  Defines argument keys taking n values, where $n = 0, 1, 2, \ldots, 8$. The syntax is similar to the one for `define keys[`n`]`.

- `append arg keys={`⟨*key definitions*⟩`}`

  Appending argument keys taking no values, i.e. adds code to the right of that key. The syntax is identical to the one for `define arg keys`.

- `prepend arg keys={`⟨*key definitions*⟩`}`

  Prepending argument keys taking no values, i.e. adds code to the left of that key. The syntax is identical to the one for `define arg keys`.

- `append arg keys[`*n*`]={`⟨*key definitions*⟩`}`

  Appending argument keys taking *n* values, where $n = 0, 1, \ldots, 8$, i.e. adds code to the right of that key. The syntax is identical to the one for `define arg keys[`*n*`]`.

- `prepend arg keys[`*n*`]={`⟨*key definitions*⟩`}`

  Prepending argument keys taking *n* values, where $n = 0, 1, \ldots, 8$, i.e. adds code to the left of that key. The syntax is identical to the one for `define arg keys[`*n*`]`.

- `remove arg key=`⟨*key name*⟩

  Removes the argument key ⟨*key name*⟩ taking no values.

- `remove arg key[`*n*`]=`⟨*key name*⟩

  Removes the argument key ⟨*key name*⟩ taking *n* values, where $n = 0, 1, 2, \ldots, 8$.

15.2 Programming keys

- `execute={`⟨*TEX code*⟩`}`

  Executes the ⟨*TEX code*⟩ on the spot.

- `set keys={`⟨*keys*⟩`}`, `keys set={`⟨*keys*⟩`}`

  Sets the keys ⟨*keys*⟩.

- `set keys x={`⟨*keys*⟩`}`, `keys set x={`⟨*keys*⟩`}`

  Sets the keys ⟨*keys*⟩, but fully expands their values.

- `data provide={`⟨*data*⟩`}`

  Provides a new piece of data consisting of a token list.

- `data set={`⟨*data*⟩`}{`⟨*value*⟩`}`

  Sets the ⟨*data*⟩ to ⟨*value*⟩.

- `data set x={`⟨*data*⟩`}{`⟨*value*⟩`}`

  Sets the ⟨*data*⟩ to ⟨*value*⟩, but fully expands the ⟨*value*⟩ first.

- `data put left={`⟨*data*⟩`}{`⟨*value*⟩`}`

  Adds the ⟨*value*⟩ to the left of ⟨*data*⟩.

- `data put left x={`⟨*data*⟩`}{`⟨*value*⟩`}`

  Adds the ⟨*value*⟩ to the left of ⟨*data*⟩, but fully expands the ⟨*value*⟩ first.

- `data put right={`⟨*data*⟩`}{`⟨*value*⟩`}`

  Adds the ⟨*value*⟩ to the right of ⟨*data*⟩.

- `data put right x={`⟨*data*⟩`}{`⟨*value*⟩`}`

  Adds the ⟨*value*⟩ to the right of ⟨*data*⟩, but fully expands the ⟨*value*⟩ first.

- `data clear={`⟨*data*⟩`}`

  Clears the piece of data ⟨*data*⟩.

- `bool provide={`⟨*boolean*⟩`}`

  Provides a new piece of data consisting of a boolean.

- `bool set true=`{⟨*boolean*⟩}

  Sets the ⟨*boolean*⟩ to true.

- `bool set false=`{⟨*boolean*⟩}

  Sets the ⟨*boolean*⟩ to false.

- `bool if TF=`{⟨*boolean*⟩}{⟨*if true*⟩}{⟨*if false*⟩},
  `bool if T=`{⟨*boolean*⟩}{⟨*if true*⟩},
  `bool if TF=`{⟨*boolean*⟩}{⟨*if false*⟩}

  Runs ⟨*if true*⟩ or ⟨*if false*⟩, depending on the value of ⟨*boolean*⟩.

- `int provide=`{⟨*integer*⟩}

  Provides a new piece of data consisting of an integer.

- `int set=`{⟨*integer*⟩}{⟨*value*⟩}

  Sets the ⟨*integer*⟩ to ⟨*value*⟩.

- `int incr=`{⟨*integer*⟩}

  Increases the ⟨*integer*⟩ by 1.

- `int if eq TF=`{⟨*integer*$_1$⟩}{⟨*integer*$_2$⟩}{⟨*if true*⟩}{⟨*if false*⟩},
  `int if eq T=`{⟨*integer*$_1$⟩}{⟨*integer*$_2$⟩}{⟨*if true*⟩},
  `int if eq F=`{⟨*integer*$_1$⟩}{⟨*integer*$_2$⟩}{⟨*if false*⟩}

  Checks whether the integers ⟨*integer*$_1$⟩ and ⟨*integer*$_2$⟩ are equal, and runs ⟨*if true*⟩ or ⟨*if false*⟩ accordingly.

- `int if greater TF=`{⟨*integer*$_1$⟩}{⟨*integer*$_2$⟩}{⟨*if true*⟩}{⟨*if false*⟩},
  `int if greater T=`{⟨*integer*$_1$⟩}{⟨*integer*$_2$⟩}{⟨*if true*⟩},
  `int if greater F=`{⟨*integer*$_1$⟩}{⟨*integer*$_2$⟩}{⟨*if false*⟩}

  Checks whether the integer ⟨*integer*$_1$⟩ is greater than ⟨*integer*$_2$⟩, and runs ⟨*if true*⟩ or ⟨*if false*⟩ accordingly.

- `int if less TF=`{⟨*integer*$_1$⟩}{⟨*integer*$_2$⟩}{⟨*if true*⟩}{⟨*if false*⟩},
  `int if less T=`{⟨*integer*$_1$⟩}{⟨*integer*$_2$⟩}{⟨*if true*⟩},
  `int if less F=`{⟨*integer*$_1$⟩}{⟨*integer*$_2$⟩}{⟨*if false*⟩}

  Checks whether the integer ⟨*integer*$_1$⟩ is less than ⟨*integer*$_2$⟩, and runs ⟨*if true*⟩ or ⟨*if false*⟩ accordingly.

- `int clear=`{⟨*integer*⟩}

  Clears the ⟨*integer*⟩, i.e. sets it to 0.

- `if blank TF=`{⟨*tokens*⟩}{⟨*if true*⟩}{⟨*if false*⟩},
  `if blank T=`{⟨*tokens*⟩}{⟨*if true*⟩},
  `if blank F=`{⟨*tokens*⟩}{⟨*if false*⟩}

  Fully expands the ⟨*tokens*⟩ and checks if it is blank, and runs ⟨*if true*⟩ or ⟨*if false*⟩ according to this.

- `str if eq TF=`{⟨*string*$_1$⟩}{⟨*string*$_2$⟩}{⟨*if true*⟩}{⟨*if false*⟩},
  `str if eq T=`{⟨*string*$_1$⟩}{⟨*string*$_2$⟩}{⟨*if true*⟩},
  `str if eq F=`{⟨*string*$_1$⟩}{⟨*string*$_2$⟩}{⟨*if false*⟩}

  Checks whether the strings ⟨*string*$_1$⟩ and ⟨*string*$_2$⟩ are equal, and runs ⟨*if true*⟩ or ⟨*if false*⟩ accordingly.

- `ERROR={`⟨*error message*⟩`}`

  Issues an generic error message. At the end of the message, it automatically adds "object \⟨*object name*⟩ on line ⟨*line number*⟩" or "class \⟨*Class name*⟩ on line ⟨*line number*⟩".

- `ERROR key value not found={`⟨*key*⟩`}{`⟨*value*⟩`}`

  Issues an error, saying that the key ⟨*key*⟩ was set to the unknown value ⟨*value*⟩.

- `ERROR arg key value not found={`⟨*key*⟩`}{`⟨*value*⟩`}`

  Issues an error, saying that the argument key ⟨*key*⟩ was set to the unknown value ⟨*value*⟩.

15.3 Fundamental keys for class/object information

- `parent={`⟨*Class*⟩`}`

  Sets the class to have parent ⟨*Class*⟩.

- `class={`⟨*Class*⟩`}`

  Sets the object to have class ⟨*Class*⟩.

- `copy={`⟨*object*⟩`}`

  Sets the object to be a copy of ⟨*object*⟩. Then ⟨*object*⟩ works as a "parent object", and all information will be inherited from ⟨*object*⟩ unless modified for the current object.

- `symbol={`⟨*value*⟩`}`

  Sets the symbol to ⟨*value*⟩. At the same time, the height phantom and the slanting phantom are set to the same value.

- `symbol put left={`⟨*value*⟩`}`

  Adds ⟨*value*⟩ to the left of the symbol. No change is made to the height phantom or the slanting phantom.

- `symbol put right={`⟨*value*⟩`}`

  Adds ⟨*value*⟩ to the right of the symbol. No change is made to the height phantom or the slanting phantom.

- `height phantom={`⟨*value*⟩`}`

  Sets the height phantom to ⟨*value*⟩.

- `slanting phantom={`⟨*value*⟩`}`

  Sets the slanting phantom to ⟨*value*⟩.

- `grading position={`⟨upper|lower⟩`}`,
  `grading pos={`⟨upper|lower⟩`}`

  Sets whether to use upper ("cohomological") or lower ("homological") grading. The default is `upper`.

- `command={`⟨*command*⟩`}`

  Applies the ⟨*command*⟩ to the symbol.

- `clear command`

  Clears the list of commands to be applied to the symbol.

- `return`

  Invokes the return routine, i.e. adds all commands, indices, and arguments to the symbol, if any such exist.

- `inner return`

  Invokes the inner return routine, i.e. adds all commands to the symbol, if any such exist.

- `right return`

  Invokes the right return routine, i.e. adds all commands, right indices, and right arguments to the symbol, if any such exist.

- `left return`

  Invokes the left return routine, i.e. adds all commands, left indices, and left arguments to the symbol, if any such exist.

- `left index return`

  Adds the left indices to the symbol, if any such exists.

- `right index return`

  Adds the right indices to the symbol, if any such exists.

- `index return`

  Adds all indices, left and right to the symbol, if any such exists.

- `left arg return`

  Adds the left argument, if any such exists, to the symbol.

- `right arg return`

  Adds the right argument, if any such exists, to the symbol.

- `arg return`

  Adds the argument, if any such exists, to the symbol.

- `output={`⟨*Class*⟩`}`

  Sets the output class to ⟨*Class*⟩.

- `do output={`⟨true|<u>false</u>⟩`}`

  Sets whether the current object/class should output or not. The default is false, but the system will automatically change this when needed. *Never* set this to `true` by default, as this will cause an infinite loop.

- `output options={`⟨*keys*⟩`}`

  Adds the ⟨*keys*⟩ to the output options, i.e. those options passed to the output class.

- `parse options={`⟨*keys*⟩`}`

  Adds the ⟨*keys*⟩ to the parse options, i.e. adds it to the key `parse code`.

- `parse`

  Invokes the parse routine, i.e. runs the key `parse code` and then empties the contents of that key.

- `parse code`

  The key where the parse options are stored. It is emptied when the parse routine is invoked.

- `math class={`⟨*command*⟩`}`

  Sets the TEX math class to be ⟨*command*⟩. The intended values are `\mathord`, `\mathop`, `\mathbin`, `\mathrel`, `\mathopen`, `\mathclose`, and `\mathpunct`.

- `default={`⟨*value*⟩`}`

  This is the key that is applied whenever the user writes something in the options which is not a key, e.g. the `1` in `\vf[1]`. By default, this keys has been set to be equal to `sep i`, but it is meant to be changeable by the user.

- `*`

  Adds a bullet to the `d`-index.

- `**`

  Adds a double bullet to the `d`-index.

- `---`, `slot`

  Adds a slot to the `i`-index.

- `...`, `dots`

  Adds three dots to the `i`-index.

- `* with other sep={`⟨*separator*⟩`}`

  Adds a bullet to the `d`-index, separated by the ⟨*separator*⟩ from any previous `d`-indices.

- `** with other sep={`⟨*separator*⟩`}`

  Adds a double bullet to the `d`-index, separated by the ⟨*separator*⟩ from any previous `d`-indices.

- `arg={`⟨*value*⟩`}`

  The key that is applied whenever the user adds an argument via the standard syntax, e.g. `\vf{\vx}`. By default, it is set to be equal to `set arg single keys`, but it is meant to be changable by the user.

- `smash`

  Applies the command `\smash` to the symbol. Equivalent to `return, command=\smash`.

- `prime`, `'`, `''`, `'''`

  Adds one or more primes to the symbol in the upper index. The first one is equivalent to `upper={\prime},next upper with sep=false`, and the rest are equivalent to multiple iterations of `prime`.

15.4 Keys for the argument parentheses

- `par`

  Turns parentheses on. Equivalent to `use par=true`.

- `no par`

  Turns parentheses off, but still prints them if more than one argument is received. Equivalent to `use par=false`.

- `never par`

  Turns parentheses completely off, even if more than one argument is received. (This is ugly and should only be used for special constructions.) Equivalent to `use par=never`.

- `use par={`⟨true|false|never⟩`}`

  Sets whether or not to use parentheses. If `true`, turns parentheses on (this is the default behaviour). If `false`, turns parentheses off, but still prints them if more than one argument is received. If `never`, turns parentheses completely off, even if more than one argument is received. (This is ugly and should only be used for special constructions.) The default value is `true`.

- `par size={`⟨normal|auto|*|*other*⟩`}`

  Sets the parentheses size. Here, `normal` means normal size parentheses, `auto` and `*` mean auto-scaled parentheses using `\left ...\right`. If another value is received, that value is used for the parenthesis size, so the intended values are `\big`, `\Big`, `\bigg`, `\Bigg`.

- `left par={`⟨*parenthesis*⟩`}`

  Sets the left parenthesis. The default value is `(`.

- `right par={`⟨*parenthesis*⟩`}`

  Sets the right parenthesis. The default value is `)`.

15.5 Keys for the `spar` routine

- `spar`

  Invokes the `spar` routine.

- `spar={`⟨normal|auto|*|*other*⟩`}`

  Invokes the `spar` routine, with the specified parenthesis size. Here, `normal` means normal size parentheses, `auto` and `*` mean auto-scaled parentheses using `\left ...\right`. If another value is received, that value is used for the parenthesis size, so the intended values are `\big`, `\Big`, `\bigg`, `\Bigg`.

- `spar size={`⟨normal|auto|*|*other*⟩`}`

  Sets the `spar` parenthesis size. Here, `normal` means normal size parentheses, `auto` and `*` mean auto-scaled parentheses using `\left ...\right`. If another value is received, that value is used for the parenthesis size, so the intended values are `\big`, `\Big`, `\bigg`, `\Bigg`.

- `left spar=`{⟨*parenthesis*⟩}

  Sets the left parenthesis for the `spar` routine. The default value is `(`.

- `right spar=`{⟨*parenthesis*⟩}

  Sets the right parenthesis for the `spar` routine. The default value is `)`.

- `other spar=`{⟨*left parenthesis*⟩}{⟨*right parenthesis*⟩}

  Invokes the `spar` routine, but with the assigned parentheses.

- `Other spar=`{⟨*left parenthesis*⟩}{⟨*right parenthesis*⟩}{⟨normal|auto|*|*other*⟩}

  Invokes the `spar` routine, but with the assigned parentheses and size. Here, `normal` means normal size parentheses, `auto` and `*` mean auto-scaled parentheses using `\left ...\right`. If another value is received, that value is used for the parenthesis size, so the intended values are `\big`, `\Big`, `\bigg`, `\Bigg`.

15.6 Keys for setting the argument

- `set arg keys=`{⟨*keys*⟩}, `arg keys set=`{⟨*keys*⟩}

  Sets the argument keys ⟨*keys*⟩.

- `set arg keys x=`{⟨*keys*⟩}, `arg keys set x=`{⟨*keys*⟩}

  Sets the argynebt keys ⟨*keys*⟩, but fully expands their values.

- `set arg single keys=`{⟨*keys*⟩}, `arg single keys set=`{⟨*keys*⟩}

  Sets the argument keys ⟨*keys*⟩, but only supports keys taking no values. This allows the arguments to contain equality signs without causing issues.

- `set arg single keys x=`{⟨*keys*⟩}, `arg single keys set x=`{⟨*keys*⟩}

  Sets the argument keys ⟨*keys*⟩, but only supports keys taking no values. If a key is not found, the value is fully expanded and printed. This allows the arguments to contain equality signs without causing issues.

- `set one arg single key=`{⟨*key*⟩}, `one arg single key set=`{⟨*key*⟩}

  Sets one single argument key taking no values. This allows the argument to contain equality signs and commas without cuasing issues.

- `set one arg single key x=`{⟨*key*⟩}, `one arg single key set x=`{⟨*key*⟩}

  Sets one single argument key taking no values, If the key is not found, the value is fully expanded and printed. This allows the argument to contain equality signs and commas without cuasing issues.

- `set arg without keyval=`{⟨*value*⟩}, `arg without keyval set=`{⟨*value*⟩}

  Sets the argument, allowing no keyval syntax.

- `set arg without keyval x=`{⟨*value*⟩}, `arg without keyval set x=`{⟨*value*⟩}

  Sets the argument, fully expanding its value, and allowing no keyval syntax.

- `pre arg=`{⟨*value*⟩}

  Sets the pre-argument.

- `pre arg put left`={⟨*value*⟩}

  Adds to the left of the pre-argument.

- `post arg`={⟨*value*⟩}

  Sets the post-argument.

- `post arg put right`={⟨*value*⟩}

  Adds to the right of the post-argument.

- `set arg sep`={⟨*value*⟩}

  Sets the argument separator. The default value is a comma.

- `set arg slot`={⟨*value*⟩}

  Sets the argument slot. The default value is `{-}`.

- `set arg dots`={⟨*value*⟩}

  Sets the argument dots. The default value is `\dots`.

- `arg keyval`={⟨true|false|single keys|one single key⟩}

  Sets whether to use argument keyval syntax or not. If `true`, `arg` is set equal to `set arg keys`. If `false`, it is set to `set arg without keyval`. If `single keys`, it is set to `set arg single keys`. If `one single key`, it is set to `set one arg single key`. The default value is `single keys`.

- `arg position`={⟨left|right⟩}, `arg pos`={⟨left|right⟩}

  Sets the position of the argument. The default is `right`, so the argument will be printed to the right of the symbol.

- `next arg with sep`={⟨true|false⟩}

  Sets whether the next argument should be separated from the current one with a separator or not. The default is `false`, but the system will automatically change this when needed.

- `sep arg`={⟨*value*⟩}

  Adds ⟨*value*⟩ to the argument, separated from any previous argument by the default argument separator.

- `comma arg`={⟨*value*⟩}

  Adds ⟨*value*⟩ to the argument, separated from any previous argument by a comma.

- `arg with other sep`={⟨*separator*⟩}{⟨*value*⟩}

  Adds ⟨*value*⟩ to the argument, separated from any previous argument by ⟨*separator*⟩.

- `arg ... with other sep`={⟨*separator*⟩}, `arg dots with other sep`={⟨*separator*⟩}

  Adds three dots to the argument, separated from any previous argument by the ⟨*separator*⟩.

- `arg --- with other sep`={⟨*separator*⟩}, `arg slot with other sep`={⟨*separator*⟩}

  Adds a slot to the argument, separated from any previous argument by the ⟨*separator*⟩.

- `arg ...`, `arg dots`

  Adds three dots to the argument, separated from any previous arguments by the standard separator.

- `comma arg ...`, `comma arg dots`

  Adds three dots to the argument, separated from any previous arguments by a comma.

- `arg ---`, `arg slot`

  Adds a slot to the argument, separated from any previous arguments by the standard separator.

- `comma arg ---`, `comma arg slot`

  Adds a slot to the argument, separated from any previous arguments by a comma.

- `clear arg`

  Clears the argument.

- `clear pre arg`

  Clears the pre-argument.

- `clear post arg`

  Clears the post-argument.

15.7 Keys for the upper index

- `upper={`⟨*value*⟩`}`

  Adds to the upper index, with no separator from any previous upper index.

- `sep upper={`⟨*value*⟩`}`

  Adds to the upper index, separated from any previous upper index by the default separator.

- `comma upper={`⟨*value*⟩`}`

  Adds to the upper index, separated from any previous upper index by a comma.

- `pre upper={`⟨*value*⟩`}`

  Sets the pre-upper index.

- `pre upper put left={`⟨*value*⟩`}`

  Adds to the left of the pre-upper index.

- `post upper={`⟨*value*⟩`}`

  Sets the post-upper index.

- `post upper put right={`⟨*value*⟩`}`

  Adds to the right of the post-upper index.

- `upper put left={`⟨*value*⟩`}`

  Adds something to the left of the upper index. As with keys like `upper`, this will also increase the number of registered upper indices by 1, and it will set `next upper with sep=true`.

- `set upper sep={`⟨*value*⟩`}`

  Sets the upper index separator to ⟨*value*⟩. By default, this is a comma.

- `next upper with sep={`⟨true|false⟩`}`

  Sets whether the next upper index should be separated from the current one by a separator.

- `upper with other sep={`⟨*separator*⟩`}{`⟨*value*⟩`}`

  Adds ⟨*value*⟩ to the upper index, separated from any prevous upper index by ⟨*separator*⟩.

- `upper ---`, `upper slot`

  Adds a slot to the upper index, with no separator from any previous upper index.

- `sep upper ---`, `sep upper slot`

  Adds a slot to the upper index, separated from any previous upper index by the default separator.

- `comma upper ---`, `comma upper slot`

  Adds a slot to the upper index, separated from any previous upper index by a comma.

- `set upper slot={`⟨*value*⟩`}`

  Sets the slot for the upper index. By default, this is `{-}`.

- `upper --- with other sep={`⟨*separator*⟩`}`,
  `upper slot with other sep={`⟨*separator*⟩`}`

  Adds a slot to the upper index, separated from any previous upper index by ⟨*separator*⟩.

- `upper ...`, `upper dots`

  Adds three dots to the upper index, with no separator from any previous upper index.

- `sep upper ...`, `sep upper dots`

  Adds three dots to the upper index, separated from any previous upper index by the default separator.

- `comma upper ...`, `comma upper dots`

  Adds three dots to the upper index, separated from any previous upper index by a comma.

- `set upper dots={`⟨*value*⟩`}`

  Sets the dots for the upper index. By default, this is `\dots`.

- `upper ... with other sep={`⟨*separator*⟩`}`,
  `upper dots with other sep={`⟨*separator*⟩`}`

  Adds three dots to the upper index, separated from any previous upper index by ⟨*separator*⟩.

- `upper *`

  Adds a bullet to the upper index, with no separator from any previous upper index.

- `upper **`

  Adds a double bullet to the upper index, with no separator from any previous upper index.

- `sep upper *`

  Adds a bullet to the upper index, separated from any previous upper index by the default separator.

- `sep upper **`

  Adds a double bullet to the upper index, separated from any previous upper index by the default separator.

- `comma upper *`

  Adds a bullet to the upper index, separated from any previous upper index by a comma.

- `comma upper **`

  Adds a double bullet to the upper index, separated from any previous upper index by a comma.

- `upper * with other sep={`⟨*separator*⟩`}`

  Adds a bullet to the upper index, separated from any previous upper index by ⟨*separator*⟩.

- `upper ** with other sep={`⟨*separator*⟩`}`

  Adds a double bullet to the upper index, separated from any previous upper index by ⟨*separator*⟩.

- `clear upper`

  Clears the upper index.

- `clear pre upper`

  Clears the pre-upper index.

- `clear post upper`

  Clears the post-upper index.

15.8 Keys for the lower index

- `lower={`⟨*value*⟩`}`

  Adds to the lower index, with no separator from any previous lower index.

- `sep lower={`⟨*value*⟩`}`

  Adds to the lower index, separated from any previous lower index by the default separator.

- `comma lower={`⟨*value*⟩`}`

  Adds to the lower index, separated from any previous lower index by a comma.

- `pre lower={`⟨*value*⟩`}`

  Sets the pre-lower index.

- `pre lower put left={`⟨*value*⟩`}`

  Adds to the left of the pre-lower index.

- `post lower={`⟨*value*⟩`}`

  Sets the post-lower index.

- `post lower put right={`⟨*value*⟩`}`

  Adds to the right of the post-lower index.

- `lower put left={`⟨*value*⟩`}`

  Adds something to the left of the lower index. As with keys like `lower`, this will also increase the number of registered lower indices by 1, and it will set `next lower with sep=true`.

- `set lower sep={`⟨*value*⟩`}`

  Sets the lower index separator to ⟨*value*⟩. By default, this is a comma.

- `next lower with sep={`⟨true|false⟩`}`

  Sets whether the next lower index should be separated from the current one by a separator.

- `lower with other sep={`⟨*separator*⟩`}{`⟨*value*⟩`}`

  Adds ⟨*value*⟩ to the lower index, separated from any prevous lower index by ⟨*separator*⟩.

- `lower ---`, `lower slot`

  Adds a slot to the lower index, with no separator from any previous lower index.

- `sep lower ---`, `sep lower slot`

  Adds a slot to the lower index, separated from any previous lower index by the default separator.

- `comma lower ---`, `comma lower slot`

  Adds a slot to the lower index, separated from any previous lower index by a comma.

- `set lower slot={`⟨*value*⟩`}`

  Sets the slot for the lower index. By default, this is `{-}`.

- `lower --- with other sep={`⟨*separator*⟩`}`,
  `lower slot with other sep={`⟨*separator*⟩`}`

  Adds a slot to the lower index, separated from any previous lower index by ⟨*separator*⟩.

- `lower ...`, `lower dots`

  Adds three dots to the lower index, with no separator from any previous lower index.

- `sep lower ...`, `sep lower dots`

  Adds three dots to the lower index, separated from any previous lower index by the default separator.

- `comma lower ...`, `comma lower dots`

  Adds three dots to the lower index, separated from any previous lower index by a comma.

- `set lower dots={`⟨*value*⟩`}`

  Sets the dots for the lower index. By default, this is `\dots`.

- `lower ... with other sep={`⟨*separator*⟩`}`,
  `lower dots with other sep={`⟨*separator*⟩`}`

  Adds three dots to the lower index, separated from any previous lower index by ⟨*separator*⟩.

- `lower *`

  Adds a bullet to the lower index, with no separator from any previous lower index.

- `lower **`

  Adds a double bullet to the lower index, with no separator from any previous lower index.

- `sep lower *`

  Adds a bullet to the lower index, separated from any previous lower index by the default separator.

- `sep lower **`

  Adds a double bullet to the lower index, separated from any previous lower index by the default separator.

- `comma lower *`

  Adds a bullet to the lower index, separated from any previous lower index by a comma.

- `comma lower **`

  Adds a double bullet to the lower index, separated from any previous lower index by a comma.

- `lower * with other sep={`⟨*separator*⟩`}`

  Adds a bullet to the lower index, separated from any previous lower index by ⟨*separator*⟩.

- `lower ** with other sep={`⟨*separator*⟩`}`

  Adds a double bullet to the lower index, separated from any previous lower index by ⟨*separator*⟩.

- `clearlower`

  Clears the lower index.

- `clear pre lower`

  Clears the pre-lower index.

- `clear post lower`

  Clears the post-lower index.

15.9 Keys for the upper left index

- `upper left={`⟨*value*⟩`}`

  Adds to the upper left index, with no separator from any previous upper left index.

- `sep upper left={`⟨*value*⟩`}`

  Adds to the upper left index, separated from any previous upper left index by the default separator.

- `comma upper left={`⟨*value*⟩`}`

  Adds to the upper left index, separated from any previous upper left index by a comma.

- `pre upper left={`⟨*value*⟩`}`

  Sets the pre-upper left index.

- `pre upper left put left={`⟨*value*⟩`}`

  Adds to the left of the pre-upper left index.

- `post upper left={`⟨*value*⟩`}`

  Sets the post-upper left index.

- `post upper left put right={`⟨*value*⟩`}`

  Adds to the right of the post-upper left index.

- `upper left put right={`⟨*value*⟩`}`

  Adds something to the right of the upper left index. As with keys like `upper left`, this will also increase the number of registered upper left indices by 1, and it will set `next upper left with sep=true`.

- `set upper left sep={`⟨*value*⟩`}`

  Sets the upper left index separator to ⟨*value*⟩. By default, this is a comma.

- `next upper left with sep={`⟨true|<u>false</u>⟩`}`

  Sets whether the next upper left index should be separated from the current one by a separator.

- `upper left with other sep={`⟨*separator*⟩`}{`⟨*value*⟩`}`

  Adds ⟨*value*⟩ to the upper left index, separated from any prevous upper left index by ⟨*separator*⟩.

- `upper left ---`, `upper left slot`

  Adds a slot to the upper left index, with no separator from any previous upper left index.

- `sep upper left ---`, `sep upper left slot`

  Adds a slot to the upper left index, separated from any previous upper left index by the default separator.

- `comma upper left ---`, `comma upper left slot`

  Adds a slot to the upper left index, separated from any previous upper left index by a comma.

- `set upper left slot={`⟨*value*⟩`}`

  Sets the slot for the upper left index. By default, this is `{-}`.

- `upper left --- with other sep={`⟨*separator*⟩`}`,
  `upper left slot with other sep={`⟨*separator*⟩`}`

  Adds a slot to the upper left index, separated from any previous upper left index by ⟨*separator*⟩.

- `upper left ...`, `upper left dots`

  Adds three dots to the upper left index, with no separator from any previous upper left index.

- `sep upper left ...`, `sep upper left dots`

  Adds three dots to the upper left index, separated from any previous upper left index by the default separator.

- `comma upper left ...`, `comma upper left dots`

  Adds three dots to the upper left index, separated from any previous upper left index by a comma.

- `set upper left dots={`⟨*value*⟩`}`

  Sets the dots for the upper left index. By default, this is `\dots`.

- `upper left ... with other sep={`⟨*separator*⟩`}`,
  `upper left dots with other sep={`⟨*separator*⟩`}`

  Adds three dots to the upper left index, separated from any previous upper left index by ⟨*separator*⟩.

- `upper left *`

  Adds a bullet to the upper left index, with no separator from any previous upper left index.

- `upper left **`

  Adds a double bullet to the upper left index, with no separator from any previous upper left index.

- `sep upper left *`

  Adds a bullet to the upper left index, separated from any previous upper left index by the default separator.

- `sep upper left **`

  Adds a double bullet to the upper left index, separated from any previous upper left index by the default separator.

- `comma upper left *`

  Adds a bullet to the upper left index, separated from any previous upper left index by a comma.

- `comma upper left **`

  Adds a double bullet to the upper left index, separated from any previous upper left index by a comma.

- `upper left * with other sep={`⟨*separator*⟩`}`

  Adds a bullet to the upper left index, separated from any previous upper left index by ⟨*separator*⟩.

- `upper left ** with other sep={`⟨*separator*⟩`}`

  Adds a double bullet to the upper left index, separated from any previous upper left index by ⟨*separator*⟩.

- `clearupper left`

  Clears the upper left index.

- `clear pre upper left`

  Clears the pre-upper left index.

- `clear post upper left`

  Clears the post-upper left index.

15.10 Keys for the lower left index

- `lower left={`⟨*value*⟩`}`

  Adds to the lower left index, with no separator from any previous lower left index.

- `sep lower left={`⟨*value*⟩`}`

  Adds to the lower left index, separated from any previous lower left index by the default separator.

- `comma lower left={`⟨*value*⟩`}`

  Adds to the lower left index, separated from any previous lower left index by a comma.

- `pre lower left={`⟨*value*⟩`}`

  Sets the pre-lower left index.

- `pre lower left put left={`⟨*value*⟩`}`

  Adds to the left of the pre-lower left index.

- `post lower left={`⟨*value*⟩`}`

  Sets the post-lower left index.

- `post lower left put right={`⟨*value*⟩`}`

  Adds to the right of the post-lower left index.

- `lower left put right={`⟨*value*⟩`}`

  Adds something to the right of the lower left index. As with keys like `lower left`, this will also increase the number of registered lower left indices by 1, and it will set `next lower left with sep=true`.

- `set lower left sep={`⟨*value*⟩`}`

  Sets the lower left index separator to ⟨*value*⟩. By default, this is a comma.

- `next lower left with sep={`⟨true|false⟩`}`

  Sets whether the next lower left index should be separated from the current one by a separator.

- `lower left with other sep={`⟨*separator*⟩`}{`⟨*value*⟩`}`

  Adds ⟨*value*⟩ to the lower left index, separated from any prevous lower left index by ⟨*separator*⟩.

- `lower left ---`, `lower left slot`

  Adds a slot to the lower left index, with no separator from any previous lower left index.

- `sep lower left ---`, `sep lower left slot`

  Adds a slot to the lower left index, separated from any previous lower left index by the default separator.

- `comma lower left ---`, `comma lower left slot`

  Adds a slot to the lower left index, separated from any previous lower left index by a comma.

- `set lower left slot={`⟨*value*⟩`}`

  Sets the slot for the lower left index. By default, this is `{-}`.

- `lower left --- with other sep={`⟨*separator*⟩`}`,
  `lower left slot with other sep={`⟨*separator*⟩`}`

  Adds a slot to the lower left index, separated from any previous lower left index by ⟨*separator*⟩.

- `lower left ...`, `lower left dots`

  Adds three dots to the lower left index, with no separator from any previous lower left index.

- `sep lower left ...`, `sep lower left dots`

  Adds three dots to the lower left index, separated from any previous lower left index by the default separator.

- `comma lower left ...`, `comma lower left dots`

  Adds three dots to the lower left index, separated from any previous lower left index by a comma.

- `set lower left dots={`⟨*value*⟩`}`

  Sets the dots for the lower left index. By default, this is `\dots`.

- `lower left ... with other sep={`⟨*separator*⟩`}`,
  `lower left dots with other sep={`⟨*separator*⟩`}`

  Adds three dots to the lower left index, separated from any previous lower left index by ⟨*separator*⟩.

- `lower left *`

  Adds a bullet to the lower left index, with no separator from any previous lower left index.

- `lower left **`

  Adds a double bullet to the lower left index, with no separator from any previous lower left index.

- `sep lower left *`

  Adds a bullet to the lower left index, separated from any previous lower left index by the default separator.

- `sep lower left **`

  Adds a double bullet to the lower left index, separated from any previous lower left index by the default separator.

- `comma lower left *`

  Adds a bullet to the lower left index, separated from any previous lower left index by a comma.

- `comma lower left **`

  Adds a double bullet to the lower left index, separated from any previous lower left index by a comma.

- `lower left * with other sep={`⟨*separator*⟩`}`

  Adds a bullet to the lower left index, separated from any previous lower left index by ⟨*separator*⟩.

- `lower left ** with other sep={`⟨*separator*⟩`}`

  Adds a double bullet to the lower left index, separated from any previous lower left index by ⟨*separator*⟩.

- `clearlower left`

  Clears the lower left index.

- `clear pre lower left`

  Clears the pre-lower left index.

- `clear post lower left`

  Clears the post-lower left index.

15.11 Keys for the `d`-index

- `d={`⟨*value*⟩`}`

  Adds to the `d`-index, with no separator from any previous `d`-index.

- `sep d={`⟨*value*⟩`}`

  Adds to the `d`-index, separated from any previous `d`-index by the default separator.

- `comma d={`⟨*value*⟩`}`

  Adds to the `d`-index, separated from any previous `d`-index by a comma.

- `pre d={`⟨*value*⟩`}`

  Sets the pre-`d`-index.

- `pre d put left={`⟨*value*⟩`}`

  Adds to the left of the pre-`d`-index.

- `post d={`⟨*value*⟩`}`

  Sets the post-`d`-index.

- `post d put right={`⟨*value*⟩`}`

  Adds to the right of the post-`d`-index.

- `d put left={`⟨*value*⟩`}`

  Adds something to the left of the `d`-index. As with keys like `d`, this will also increase the number of registered `d`-indices by 1, and it will set `next d with sep=true`.

- `set d sep={`⟨*value*⟩`}`

  Sets the `d`-index separator to ⟨*value*⟩. By default, this is a comma.

- `next d with sep={`⟨true|false⟩`}`

  Sets whether the next `d`-index should be separated from the current one by a separator.

- `d with other sep={`⟨*separator*⟩`}{`⟨*value*⟩`}`

  Adds ⟨*value*⟩ to the `d`-index, separated from any prevous `d`-index by ⟨*separator*⟩.

- `d ---`, `d slot`

  Adds a slot to the `d`-index, with no separator from any previous `d`-index.

- `sep d ---`, `sep d slot`

  Adds a slot to the `d`-index, separated from any previous `d`-index by the default separator.

- `comma d ---`, `comma d slot`

  Adds a slot to the `d`-index, separated from any previous `d`-index by a comma.

- `set d slot={`⟨*value*⟩`}`

  Sets the slot for the `d`-index. By default, this is `{-}`.

- `d --- with other sep={`⟨*separator*⟩`}`, `d slot with other sep={`⟨*separator*⟩`}`

  Adds a slot to the `d`-index, separated from any previous `d`-index by ⟨*separator*⟩.

- `d ...`, `d dots`

  Adds three dots to the `d`-index, with no separator from any previous `d`-index.

- `sep d ...`, `sep d dots`

  Adds three dots to the `d`-index, separated from any previous `d`-index by the default separator.

- `comma d ...`, `comma d dots`

  Adds three dots to the `d`-index, separated from any previous `d`-index by a comma.

- `set d dots={`⟨*value*⟩`}`

  Sets the dots for the `d`-index. By default, this is `\dots`.

- `d ... with other sep={`⟨*separator*⟩`}`, `d dots with other sep={`⟨*separator*⟩`}`

  Adds three dots to the `d`-index, separated from any previous `d`-index by ⟨*separator*⟩.

- `d *`

  Adds a bullet to the `d`-index, with no separator from any previous `d`-index.

- `d **`

  Adds a double bullet to the `d`-index, with no separator from any previous `d`-index.

- `sep d *`

  Adds a bullet to the `d`-index, separated from any previous `d`-index by the default separator.

- `sep d **`

  Adds a double bullet to the `d`-index, separated from any previous `d`-index by the default separator.

- `comma d *`

  Adds a bullet to the `d`-index, separated from any previous `d`-index by a comma.

- `comma d **`

  Adds a double bullet to the `d`-index, separated from any previous `d`-index by a comma.

- `d * with other sep={`⟨*separator*⟩`}`

  Adds a bullet to the `d`-index, separated from any previous `d`-index by ⟨*separator*⟩.

- `d ** with other sep={`⟨*separator*⟩`}`

  Adds a double bullet to the `d`-index, separated from any previous `d`-index by ⟨*separator*⟩.

- `clear d`

  Clears the `d`-index.

- `clear pre d`

  Clears the pre-`d`-index.

- `clear post d`

  Clears the post-`d`-index.

15.12 Keys for the `i`-index

- `i={`⟨*value*⟩`}`

  Adds to the `i`-index, with no separator from any previous `i`-index.

- `sep i={`⟨*value*⟩`}`

  Adds to the `i`-index, separated from any previous `i`-index by the default separator.

- `comma i={`⟨*value*⟩`}`

  Adds to the `i`-index, separated from any previous `i`-index by a comma.

- `pre i={`⟨*value*⟩`}`

  Sets the pre-`i`-index.

- `pre i put left={`⟨*value*⟩`}`

  Adds to the left of the pre-`i`-index.

- `post i={`⟨*value*⟩`}`

  Sets the post-`i`-index.

- `post i put right={`⟨*value*⟩`}`

  Adds to the right of the post-`i`-index.

- `i put left={`⟨*value*⟩`}`

  Adds something to the left of the `i`-index. As with keys like `i`, this will also increase the number of registered `i`-indices by 1, and it will set `next i with sep=true`.

- `set i sep={`⟨*value*⟩`}`

  Sets the `i`-index separator to ⟨*value*⟩. By default, this is a comma.

- `next i with sep={`⟨true|false⟩`}`

  Sets whether the next `i`-index should be separated from the current one by a separator.

- `i with other sep={`⟨*separator*⟩`}{`⟨*value*⟩`}`

  Adds ⟨*value*⟩ to the `i`-index, separated from any prevous `i`-index by ⟨*separator*⟩.

- `i ---`, `i slot`

  Adds a slot to the `i`-index, with no separator from any previous `i`-index.

- `sep i ---`, `sep i slot`

  Adds a slot to the `i`-index, separated from any previous `i`-index by the default separator.

- `comma i ---`, `comma i slot`

  Adds a slot to the `i`-index, separated from any previous `i`-index by a comma.

- `set i slot={`⟨*value*⟩`}`

  Sets the slot for the `i`-index. By default, this is `{-}`.

- `i --- with other sep={`⟨*separator*⟩`}`, `i slot with other sep={`⟨*separator*⟩`}`

  Adds a slot to the `i`-index, separated from any previous `i`-index by ⟨*separator*⟩.

- `i ...`, `i dots`

  Adds three dots to the `i`-index, with no separator from any previous `i`-index.

- `sep i ...`, `sep i dots`

  Adds three dots to the `i`-index, separated from any previous `i`-index by the default separator.

- `comma i ...`, `comma i dots`

  Adds three dots to the `i`-index, separated from any previous `i`-index by a comma.

- `set i dots={`⟨*value*⟩`}`

  Sets the dots for the `i`-index. By default, this is `\dots`.

- `i ... with other sep={`⟨*separator*⟩`}`, `i dots with other sep={`⟨*separator*⟩`}`

  Adds three dots to the `i`-index, separated from any previous `i`-index by ⟨*separator*⟩.

- `i *`

  Adds a bullet to the `i`-index, with no separator from any previous `i`-index.

- `i **`

  Adds a double bullet to the `i`-index, with no separator from any previous `i`-index.

- `sep i *`

  Adds a bullet to the `i`-index, separated from any previous `i`-index by the default separator.

- `sep i **`

  Adds a double bullet to the `i`-index, separated from any previous `i`-index by the default separator.

- `comma i *`

  Adds a bullet to the `i`-index, separated from any previous `i`-index by a comma.

- `comma i **`

  Adds a double bullet to the `i`-index, separated from any previous `i`-index by a comma.

- `i * with other sep={`⟨*separator*⟩`}`

  Adds a bullet to the `i`-index, separated from any previous `i`-index by ⟨*separator*⟩.

- `i ** with other sep={`⟨*separator*⟩`}`

  Adds a double bullet to the `i`-index, separated from any previous `i`-index by ⟨*separator*⟩.

- `clear i`

  Clears the `i`-index.

- `clear pre i`

  Clears the pre-`i`-index.

- `clear post i`

  Clears the post-`i`-index.

15.13 The predefined argument keys

These are the predefined keys that work inside the argument.

- `execute={`⟨*code*⟩`}`

  Executes the ⟨*code*⟩ on the spot. This is not strictly speaking a logic key, but this allows you to perform logical operations that are not allowed by the other logic keys.

- `set keys={`⟨*keys*⟩`}`, `keys set={`⟨*keys*⟩`}`

  Sets the keys ⟨*keys*⟩.

- `set keys x={`⟨*keys*⟩`}`, `keys set x={`⟨*keys*⟩`}`

  Sets the keys ⟨*keys*⟩, but fully expands their values.

- `set arg keys={`⟨*keys*⟩`}`, `arg keys set={`⟨*keys*⟩`}`

  Sets the argument keys ⟨*keys*⟩.

- `set arg keys x={`⟨*keys*⟩`}`, `arg keys set x={`⟨*keys*⟩`}`

  Sets the argument keys ⟨*keys*⟩, but fully expands their values.

- `set arg single keys={`⟨*keys*⟩`}`, `arg single keys set={`⟨*keys*⟩`}`

  Sets the argument keys ⟨*keys*⟩, but only supports keys taking no values. This allows the arguments to contain equality signs without causing issues.

- `set arg single keys x={`⟨*keys*⟩`}`, `arg single keys set x={`⟨*keys*⟩`}`

  Sets the argument keys ⟨*keys*⟩, but only supports keys taking no values. If a key is not found, the value is fully expanded and printed. This allows the arguments to contain equality signs without causing issues.

- `set one arg single key={`⟨*key*⟩`}`, `one arg single key set={`⟨*key*⟩`}`

  Sets one single argument key taking no values. This allows the argument to contain equality signs and commas without cuasing issues.

- `set one arg single key x={`⟨*key*⟩`}`, `one arg single key set x={`⟨*key*⟩`}`

  Sets one single argument key taking no values, If the key is not found, the value is fully expanded and printed. This allows the argument to contain equality signs and commas without cuasing issues.

- `set arg without keyval={`⟨*value*⟩`}`, `arg without keyval set={`⟨*value*⟩`}`

  Sets the argument, allowing no keyval syntax.

- `set arg without keyval x={`⟨*value*⟩`}`, `arg without keyval set x={`⟨*value*⟩`}`

  Sets the argument, fully expanding its value, and allowing no keyval syntax.

- `default={`⟨*value*⟩`}`

  This is the value that is applied whenever a value is passed to the argument that is not recognized as a key, e.g. the `\vx` in `\vf{\vx}`. By default, this is set to be equivalent to `sep`.

- `sep={`⟨*value*⟩`}`

  Adds the ⟨*value*⟩ to the argument, separated from any previous argument by the default separator.

- `no sep={`⟨*value*⟩`}`

  Adds the ⟨*value*⟩ to the argument, with no separator from any previous argument.

- `comma={`⟨*value*⟩`}`

  Adds the ⟨*value*⟩ to the argument, separated from any previous argument by a comma.

- `---`, `slot`

  Adds a slot to the argument, separated from any previous argument by the default separator.

- `comma ---`, `comma slot`

  Adds a slot to the argument, separated from any previous argument by a comma.

- `...`, `dots`

  Adds three dots to the argument, separated from any previous argument by the default separator.

- `comma ...`, `comma dots`

  Adds three dots to the argument, separated from any previous argument by a comma.

- `other sep={`⟨*separator*⟩`}{`⟨*value*⟩`}`

  Adds ⟨*value*⟩ to the argument, separated from any previous argument by ⟨*separator*⟩.

- `--- with other sep={`⟨*separator*⟩`}`, `slot with other sep={`⟨*separator*⟩`}`

  Adds a slot to the argument, separated from any previous argument by ⟨*separator*⟩.

- `... with other sep={`⟨*separator*⟩`}` , `dots with other sep={`⟨*separator*⟩`}`

  Adds three dots to the argument, separated from any previous argument by ⟨*separator*⟩.

15.14 The programming commands

The following commands are available for programming inside keys, including `execute={...}`:

- `\SemantexThis`

  Returns the name of the current class or object. It is returned in the format `object_`⟨*name of object without backslash*⟩ and `class_`⟨*name of class without backslash*⟩, which is the way the names are stored internally.

- `\SemantexSetKeys{`⟨*keys*⟩`}`, `\SemantexKeysSet{`⟨*keys*⟩`}`

  Sets the ⟨*keys*⟩.

- `\SemantexSetKeysx{`⟨*keys*⟩`}`, `\SemantexKeysSetx{`⟨*keys*⟩`}`

  Sets the ⟨*keys*⟩, but fully expands their values.

- `\SemantexSetArgKeys{`⟨*keys*⟩`}`, `\SemantexArgKeysSet{`⟨*keys*⟩`}`

  Sets the argument ⟨*keys*⟩.

- `\SemantexSetArgKeysx{`⟨*keys*⟩`}`, `\SemantexArgKeysSetx{`⟨*keys*⟩`}`

  Sets the argument ⟨*keys*⟩, but fully expands their values.

- `\SemantexSetArgSingleKeys{`⟨*keys*⟩`}`, `\SemantexArgSingleKeysSet{`⟨*keys*⟩`}`

  Sets the argument keys ⟨*keys*⟩, but only supports keys taking no values. This allows the arguments to contain equality signs without causing issues.

- `\SemantexSetArgSingleKeysx{`⟨*keys*⟩`}`, `\SemantexArgSingleKeysSetx{`⟨*keys*⟩`}`

  Sets the argument keys ⟨*keys*⟩, but only supports keys taking no values. If a key is not found, the value is fully expanded and printed. This allows the arguments to contain equality signs without causing issues.

- `\SemantexSetOneArgSingleKey{`⟨*keys*⟩`}`, `\SemantexOneSingleArgKeySet{`⟨*keys*⟩`}`

  Sets one single argument key taking no values. This allows the argument to contain equality signs and commas without cuasing issues.

- `\SemantexSetOneArgSingleKeyx{`⟨*keys*⟩`}`, `\SemantexOneSingleArgKeySetx{`⟨*keys*⟩`}`

  Sets one single argument key taking no values, If the key is not found, the value is fully expanded and printed. This allows the argument to contain equality signs and commas without cuasing issues.

- `\SemantexSetArgWithoutKeyval{`⟨*value*⟩`}`, `\SemantexArgWithoutKeyvalSet{`⟨*value*⟩`}`

  Sets the argument, allowing no keyval syntax.

- `\SemantexSetArgWithoutKeyval{⟨value⟩}`, `\SemantexArgWithoutKeyvalSet{⟨value⟩}`

  Sets the argument, fully expanding its value, and allowing no keyval syntax.

- `\SemantexDataProvide{⟨data⟩}`

  Provides a new piece of data consisting of a token list.

- `\SemantexDataSet{⟨data⟩}{⟨value⟩}`

  Sets the ⟨*data*⟩ to ⟨*value*⟩.

- `\SemantexDataSetx{⟨data⟩}{⟨value⟩}`

  Sets the ⟨*data*⟩ to ⟨*value*⟩, but fully expands the ⟨*value*⟩ first.

- `\SemantexDataPutLeft{⟨data⟩}{⟨value⟩}`

  Adds the ⟨*value*⟩ to the left of ⟨*data*⟩.

- `\SemantexDataPutLeftx{⟨data⟩}{⟨value⟩}`

  Adds the ⟨*value*⟩ to the left of ⟨*data*⟩, but fully expands the ⟨*value*⟩ first.

- `\SemantexDataPutRight{⟨data⟩}{⟨value⟩}`

  Adds the ⟨*value*⟩ to the right of ⟨*data*⟩.

- `\SemantexDataPutRightx{⟨data⟩}{⟨value⟩}`

  Adds the ⟨*value*⟩ to the right of ⟨*data*⟩, but fully expands the ⟨*value*⟩ first.

- `\SemantexDataGet{⟨data⟩}`

  Returns the value of ⟨*data*⟩.

- `\SemantexDataGetExpNot{⟨data⟩}`

  Returns the value of ⟨*data*⟩, enclosed in `\unexpanded` so that it can be used within an `x`-type expansion.

- `\SemantexDataClear{⟨data⟩}`

  Clears the piece of data ⟨*data*⟩.

- `\SemantexBoolProvide{⟨boolean⟩}`

  Provides a new piece of data consisting of a boolean.

- `\SemantexBoolSetTrue{⟨boolean⟩}`

  Sets the ⟨*boolean*⟩ to true.

- `\SemantexBoolSetFalse{⟨boolean⟩}`

  Sets the ⟨*boolean*⟩ to false.

- `\SemantexBoolIfTF{⟨boolean⟩}{⟨if true⟩}{⟨if false⟩}`,
  `\SemantexBoolIfT{⟨boolean⟩}{⟨if true⟩}`,
  `\SemantexBoolIfF{⟨boolean⟩}{⟨if false⟩}`

  Runs ⟨*if true*⟩ or ⟨*if false*⟩, depending on the value of ⟨*boolean*⟩.

- `\SemantexIntProvide{⟨integer⟩}`

  Provides a new piece of data consisting of an integer.

- `\SemantexIntGet{`⟨*integer*⟩`}`

  Returns the value of the ⟨*integer*⟩.

- `\SemantexIntSet{`⟨*integer*⟩`}{`⟨*value*⟩`}`

  Sets the ⟨*integer*⟩ to ⟨*value*⟩.

- `\SemantexIntIncr{`⟨*integer*⟩`}`

  Increases the ⟨*integer*⟩ by 1.

- `\SemantexIntIfEqTF{`⟨*integer*$_1$⟩`}{`⟨*integer*$_2$⟩`}{`⟨*if true*⟩`}{`⟨*if false*⟩`}`,
  `\SemantexIntIfEqT`⟨*integer*$_1$⟩`}{`⟨*integer*$_2$⟩`}{`⟨*if true*⟩`}`,
  `\SemantexIntIfEqF{`⟨*integer*$_1$⟩`}{`⟨*integer*$_2$⟩`}{`⟨*if false*⟩`}`

  Checks whether the integers ⟨*integer*$_1$⟩ and ⟨*integer*$_2$⟩ are equal, and runs ⟨*if true*⟩ or ⟨*if false*⟩ accordingly.

- `\SemantexIntIfGreaterTF{`⟨*integer*$_1$⟩`}{`⟨*integer*$_2$⟩`}{`⟨*if true*⟩`}{`⟨*if false*⟩`}`,
  `\SemantexIntIfGreaterT{`⟨*integer*$_1$⟩`}{`⟨*integer*$_2$⟩`}{`⟨*if true*⟩`}`,
  `\SemantexIntIfGreaterF{`⟨*integer*$_1$⟩`}{`⟨*integer*$_2$⟩`}{`⟨*if false*⟩`}`

  Checks whether the integer ⟨*integer*$_1$⟩ is greater than ⟨*integer*$_2$⟩, and runs ⟨*if true*⟩ or ⟨*if false*⟩ accordingly.

- `\SemantexIntIfLessTF{`⟨*integer*$_1$⟩`}{`⟨*integer*$_2$⟩`}{`⟨*if true*⟩`}{`⟨*if false*⟩`}`,
  `\SemantexIntIfLessT{`⟨*integer*$_1$⟩`}{`⟨*integer*$_2$⟩`}{`⟨*if true*⟩`}`,
  `\SemantexIntIfLessF{`⟨*integer*$_1$⟩`}{`⟨*integer*$_2$⟩`}{`⟨*if false*⟩`}`

  Checks whether the integer ⟨*integer*$_1$⟩ is less than ⟨*integer*$_2$⟩, and runs ⟨*if true*⟩ or ⟨*if false*⟩ accordingly.

- `\SemantexIntClear{`⟨*integer*⟩`}`

  Clears the ⟨*integer*⟩, i.e. sets it to 0.

- `\SemantexIfBlankTF{`⟨*tokens*⟩`}{`⟨*if true*⟩`}{`⟨*if false*⟩`}`,
  `\SemantexIfBlankT{`⟨*tokens*⟩`}{`⟨*if true*⟩`}`,
  `\SemantexIfBlankF{`⟨*tokens*⟩`}{`⟨*if false*⟩`}`

  Fully expands the ⟨*tokens*⟩ and checks if it is blank, and runs ⟨*if true*⟩ or ⟨*if false*⟩ according to this.

- `\SemantexStrIfEqTF{`⟨*string*$_1$⟩`}{`⟨*string*$_2$⟩`}{`⟨*if true*⟩`}{`⟨*if false*⟩`}`,
  `\SemantexStrIfEqT{`⟨*string*$_1$⟩`}{`⟨*string*$_2$⟩`}{`⟨*if true*⟩`}`,
  `\SemantexStrIfEqF{`⟨*string*$_1$⟩`}{`⟨*string*$_2$⟩`}{`⟨*if false*⟩`}`

  Checks whether the strings ⟨*string*$_1$⟩ and ⟨*string*$_2$⟩ are equal, and runs ⟨*if true*⟩ or ⟨*if false*⟩ accordingly.

- `\SemantexERROR{`⟨*error message*⟩`}`

  Issues an generic error message. At the end of the message, it automatically adds "object \⟨*object name*⟩ on line ⟨*line number*⟩" or "class \⟨*Class name*⟩ on line ⟨*line number*⟩".

- `\SemantexERRORKeyValueNotFound{`⟨*key*⟩`}{`⟨*value*⟩`}`

  Issues an error, saying that the key ⟨*key*⟩ was set to the unknown value ⟨*value*⟩.

- `\SemantexERRORArgKeyValueNotFound{`⟨*key*⟩`}{`⟨*value*⟩`}`

  Issues an error, saying that the argument key ⟨*key*⟩ was set to the unknown value ⟨*value*⟩.

- `\SemantexExpNot{`⟨*value*⟩`}`

  An alias for `\unexpanded` (also known as `\exp_not:N` in LaTeX3).

15.15 The class types

The SemanTeX system uses several different *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.

Each class has a `\New...` as well as a `\Declare...` variant, the difference being that the `\New...` variant raises an error if the command name is already taken. Both variants will raise an error if a SemanTeX class with the same name already exists. Allowing you to override existing SemanTeX definitions would require fundamental changes to the system which would slow it down significantly.

Similarly, you can use both of the commands `\NewObject` and `\DeclareObject` to define objects, the difference being that `\NewObject` raises an error if the command name is already taken. Both commands will raise an error if a SemanTeX object with the same name already exists. Allowing you to override existing SemanTeX definitions would require fundamental changes to the system which would slow it down significantly.

The current implementation has the following class types:

- `Variable`: A new class is defined with the syntax

  `\NewVariableClass{\`⟨*Class*⟩`}[`⟨*options*⟩`]`
  `\DeclareVariableClass{\`⟨*Class*⟩`}[`⟨*options*⟩`]`

  A new object is declared by

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

  The syntax for this object is

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

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

  `\NewSymbolClass\`⟨*Class*⟩`[`⟨*options*⟩`]`
  `\DeclareSymbolClass\`⟨*Class*⟩`[`⟨*options*⟩`]`

  A new object is declared by

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

  The syntax for this object is

  `\`⟨*object*⟩`[`⟨*options*⟩`]`

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

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

A new object is declared by

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

The syntax for this object is

\⟨*object*⟩

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.

The last class type, called `Simple`, 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 you simply should not use it.

15.16 The predefined data

By default, the following data are defined for each class or object and are accessible via the programming keys and commands:

- `symbol` (token list): the symbol.
- `output` (token list): the name of the output class.
- `output options` (token list): the output options, i.e. the options to be passed to the output class.
- `math class` (token list): the TEX math class command that the final output is evnetually wrapped around; the intended use of this is the TEX commands `\mathord`, `\mathop`, `\mathbin`, `\mathrel`, `\mathopen`, `\mathclose`, and `\mathpunct`.
- `height phantom` (token list): the height phantom that is used for calculating the height of left indices.
- `slanting phantom` (token list): the slanting phantom that is used for calculating the slanting of left indices.
- `par size` (token list): the size of the argument parentheses. The value `normal` means normal size parentheses, `auto` and `*` mean auto-scaled parentheses using `\left ...\right`. If another value is received, that value is used for the parenthesis size, so the intended values are `\big`, `\Big`, `\bigg`, `\Bigg`. The default value is `normal`.
- `left par` (token list): the left argument parenthesis; the default value is `(`.
- `right par` (token list): the right argument parenthesis; the default value is `)`.
- `spar size` (token list): the size of the symbol parentheses (for use with the `spar` routine). The value `normal` means normal size parentheses, `auto` and `*` mean auto-scaled parentheses using `\left ...\right`. If another value is received, that value is used for the parenthesis size, so the intended values are `\big`, `\Big`, `\bigg`, `\Bigg`. The default value is `normal`.

- `left spar` (token list): the left symbol parenthesis (for use with the `spar` routine); the default value is `(`.
- `right spar` (token list): the right symbol parenthesis (for use with the `spar` routine); the default value is `)`.
- `arg` (token list): the argument.
- `pre arg` (token list): to be printed in front of the argument, if the argument is non-empty.
- `post arg` (token list): to be printed after the argument, if the argument is non-empty.
- `arg sep` (token list): the argument separator; comma by default.
- `arg slot` (token list): the argument slot; `{-}` by default.
- `arg dots` (token list): the argument dots; `\dots` by default.
- `upper` (token list): the upper index.
- `pre upper` (token list): the pre-upper index, to be printed in front of the upper index, if the upper index is non-empty.
- `post upper` (token list) the post-upper index, to be printed after the upper index, if the upper index is non-empty.
- `upper sep` (token list): the upper index separator; comma by default.
- `upper dots` (token list): the upper dots; `\dots` by default.
- `upper slot` (token list): the upper slot; `{-}` by default.
- `lower` (token list): the lower index.
- `pre lower` (token list): the pre-lower index, to be printed in front of the lower index, if the lower index is non-empty.
- `post lower` (token list) the post-lower index, to be printed after the lower index, if the lower index is non-empty.
- `lower sep` (token list): the lower index separator; comma by default.
- `lower dots` (token list): the lower dots; `\dots` by default.
- `lower slot` (token list): the lower slot; `{-}` by default.
- `upper left` (token list): the upper left index.
- `pre upper left` (token list): the pre-upper left index, to be printed in front of the upper left index, if the upper left index is non-empty.
- `post upper left` (token list) the post-upper left index, to be printed after the upper left index, if the upper left index is non-empty.
- `upper left sep` (token list): the upper left index separator; comma by default.
- `upper left dots` (token list): the upper left dots; `\dots` by default.
- `upper left slot` (token list): the upper left slot; `{-}` by default.

- `lower left` (token list): the lower left index.
- `pre lower left` (token list): the pre-lower left index, to be printed in front of the lower left index, if the lower left index is non-empty.
- `post lower left` (token list) the post-lower left index, to be printed after the lower left index, if the lower left index is non-empty.
- `lower left sep` (token list): the lower left index separator; comma by default.
- `lower left dots` (token list): the lower left dots; `\dots` by default.
- `lower left slot` (token list): the lower left slot; `{-}` by default.
- `upper grading` (boolean): whether or not to use upper (cohomological) grading; true by default.
- `par` (boolean): whether or not to use parentheses; true by default.
- `flex par` (boolean): if `par` is set to false, setting `flex par` to true will still print a pair of parentheses when there is more than one argument; false by default.
- `left argument` (boolean): if true, the argument (and parentheses) will be printed to the *left* of the symbol; false by default.
- `next arg with sep` (boolean): if true, the next argument will have a separator printed in front of it.
- `next upper with sep` (boolean): If true, the next upper index will have a separator printed in front of it.
- `next lower with sep` (boolean): If true, the next lower index will have a separator printed in front of it.
- `next upper left with sep` (boolean): If true, the next upper left index will have a separator printed in front of it.
- `next lower left with sep` (boolean): If true, the next lower upper index will have a separator printed in front of it.
- `number of arguments` (integer): the number of arguments.
- `number of upper indices` (integer): the number of upper indices.
- `number of lower indices` (integer): the number of lower indices.
- `number of upper left indices` (integer): the number of upper left indices.
- `number of lower left indices` (integer): the number of lower left indices.