

YAX is Yet Another Key System, or YAX Ain't Keys, however you want it. It has some peculiarities. First, keys are not keys, but attributes, which means they depend on a parameter. You don't set key individually in YAX (although you can do it), you define parameters, which have attributes, which have values. Second, parameters can have definitions, so that when you set it you can also execute a command, or execute it later; thus, YAX is halfway between key management and macro definition. Finally, the syntax of YAX is not traditional: there are as few braces as possible, which you can find tremendously annoying, but it can be changed.

And, of course, YAX is format-independant. Files are provided for LaTeX and ConTeXt, so you can input it with `\usepackage{yax}` or `\usemodule[yax]` respectively; anywhere else just use `\input yax`.

## Setting parameters

Here's an example of how values are set:

```
\setparameter Zappa :
  firstname = Frank
  motto     = "Music is the best"
  hairstyle = \moustache
```

Here we have defined a *parameter*, 'Zappa', which has *attributes* 'firstname', 'motto' and 'hairstyle' with *values* 'Frank', 'Music is the best' and '\moustache' respectively.

**\setparameter** Here's how you set parameters more precisely. This command is the only way to set values. Its simplified syntax is as follows:

```
\setparameter<list of parameters> :
  <attribute> = <value>
  <attribute> = <value>
  ...
\par
```

First, the `\par` command that terminates the declaration: it was not chosen simply because it echoes *parameter*, but above all because it can be implicit in a blank line, i.e. `\setpa-`

parameter can be delimited by a blank line. That's why in the first example there seemed to be nothing delimiting `\setparameter`: it is supposed to be followed by a blank line. I will regularly display parameter setting in this fashion, even though the following would be equally legitimate:

```
\setparameter Zappa : firstname = Frank ... \par
```

barring readability, obviously.

As for the detail, *<list of parameters>* is a list of space-delimited parameters, followed by a colon. Any space before the colon is removed, that's why

```
\setparameter Zappa Boulez: ... \par
\setparameter Zappa Boulez : ... \par
```

set the attributes of the same Zappa and Boulez parameters. Which means, of course, that you can set the attributes of as many parameters as you wish by doing so.

The name of each parameter should be fully expandable. It can contain spaces, but then it should be enclosed between braces, since space delimits parameters in *<list of parameters>*.

Finally, you can use `\setparameter` several times on the same parameter(s), by default it does the same thing as using one big `\setparameter`. I say *by default*, because things might occur between the two calls, e.g. the parameter might be active and have deleted its own attributes after the first call, which anyway is something we'll see later.

Each *<attribute>* is made of the same thing as *<parameter>*: i.e. anything expandable. However, an *<attribute>* cannot have a name beginning with the strings `e:`, `g:` and `x:`. Indeed, these prefixes are used to specify how an attribute is to be defined, and they are similar to `\edef`, `\gdef` and `\xdef` (with no prefix meaning `\def`), i.e.:

```
\setparameter foo :
  one   = \whatever
e: two  = \whatever
g: three = \whatever
x: four  = \whatever
```

defines, for the parameter `foo`, the attributes `one` and `three` as `\whatever` and the attributes `two` and `four` as the full expansion of `\whatever`; moreover, `one` and `two` are locally defined, whereas `three` and `four` are globally defined. The space between the prefix and the name of the parameter is optional.

Finally, *<value>* may be given in three ways. First, it can be delimited by a space, e.g.:

```
\setparameter Zappa :
  first name = Frank
```

```

motto      = {Music is the best}
...

```

here the space is simply the end of the line. The spaces in `Music is the best` aren't seen, because of the braces (which will be removed from the value). Don't forget that control sequences eat the subsequent space, hence a control sequence can't be at the end of a value supposedly delimited by space without braces. Actually, it can't be at the beginning either (see below).

Otherwise, `<value>` can be given between double quotes:

```

\setparameter Zappa :
  first name = "Frank"
  motto      = "Music is the best"
  hairstyle  = "\mustache"

```

which basically act as braces. The space inserted here by the end of the line is optional and the following can be done:

```

\setparameter Zappa:first name="Frank"motto="Music is the best"...\par

```

Finally, there's a special rule. Control sequences gobble the next space, but a `<value>` can be made of one and only one control sequence, e.g.:

```

\setparameter Zappa :
  hairstyle  = \mustache
...

```

i.e. if `\X` sees a control sequence at the beginning of a value it will take this control sequence only as `<value>`, which means that anything thereafter will be considered as belonging to the next `<parameter>` name. Hence, all the following are bad ideas:

```

\setparameter bad values :
    % the space stops the value to "bad"
one   = bad space
    % the control sequence is taken as the only token
two   = \control sequence
    % "\sequence" eats the delimitating space
three = control\sequence
...

```

but they'd all be ok with quotes or braces (which still require a space after):

```

\setparameter good values :
  one   = "bad space"
  two   = "\control sequence"
  three = {control\sequence}
  four  = \LoneLyCommand
  five  = "\Command\Command"
...

```

Finally, there are some exceptions to this rule: first, always put `\par` between braces, otherwise it will be seen as the end of the parameter. Second, always put a character denotation between either braces or quotes, even if it's the only control sequence. If you don't know what character denotation means, never mind. Just remember that `\bgroup` and `\egroup` are character denotations.

Now I'm sure you ask: what is this lousy syntax? An answer is I don't like braces. Another answer is `\YAX` is not designed to store complicated strings, although it can do it. Instead it aims at setting simple values in an orderly fashion, e.g.:

```

\setparameter page :
  pagewidth   = 32cm
  pageheight  = 30cm
  lines       = 45
  whatever    = \foo
...

```

in which case it is very handy, especially when you're in a hurry:

```

\setparameter page: pagewidth=32cm pageheight=30cm whatever=\foo...\par

```

Thus, I find commas to delimit values equally superfluous and find the odd quote better. Anyway setting parameters is far from the whole story.

## The *meta* attribute

You can give any attribute to any parameter (unless they're restricted, but that's not the point for the time being). However, there's one particular attribute which has a special meaning: *meta*. The value of *meta* should be another parameter. Then, when querying the value of an attribute, say *attr*, for a given parameter, say *param*, `\YAX` will do the following: if *param* has *attr*, it is returned. Otherwise, if *param* has a *meta* attribute, whose value is for instance *metaparam*, then *attr* is queried for the value of *metaparam*. And if *metaparam* has no *attr* but has a *meta* attribute, this process continues, until either a parameter is found with *attr* or there are no new *meta*. For instance:

```

\setparameter A : attr = value ... \par
\setparameter B : meta = A ... \par
\setparameter C : meta = B ... \par

```

If parameter C has no attr, then it retrieves it from parameter A via B (or from B if it has one). On the other hand, if C has attr, then its own value is returned. You can also query the value of an attribute for a parameter and forbid the search for meta's, as explained in the next section.

Don't be afraid to create loop with meta. The following is perfectly legitimate:

```

\setparameter A : meta = C ... \par
\setparameter B : meta = A ... \par
\setparameter C : meta = B ... \par

```

and it can even be useful. Y~~A~~X detects loops when searching meta paths and stops in time (with no value, obviously). Finally, you can set meta to a parameter that doesn't exist. It will simply return no value.

The use of meta is useful to create families of parameters and/or to set default values, e.g.:

```


\setparameter mammal :
  egg = no
  fur = yes
  ...
\setparameter cat whale :
  meta = mammal
  ...
\setparameter cat :      \setparameter whale :
  foot = clawed          fur = no
  tooth = fang            foot = flipper
  ...                    tooth = baleen
                        ...
\setparameter tiger mykitty :
  meta = cat
  stripped = yes
  ...
\setparameter tiger :      \setparameter mykitty :
  foot = "very bad news"    foot = "bad news"
  ...                      ...

```

Apart from that, meta behaves as any other attribute, i.e. it can be freely set and queried.

## Using values

Once attributes have been set, they can be queried by the macros that follow. But first, one last bit of odd syntax: `<parameter>:<attribute>` means anything up to the colon as the `<parameter>` and then anything up to the next space as the `<attribute>`. That's the reason why space in attribute names is a bad idea: the space is the main delimiter when using attributes. It is gobbled in the process. On the other hand, any space surrounding the colon is removed, so that 'zappa:hairstyle' and 'zappa : hairstyle' denote the same attribute of the same parameter. Good news, though: if you don't like that syntax, the next section explains how to create commands with the same meaning but a different syntax.

In what follows, fully expandable commands are marked with .

### `\nometa<command>`

All the commands that follow can be prefixed with `\nometa`, except `\deleteattribute`. In this case, `\YA` will return the value of the attribute for the specified parameter, as usual, or return no value, i.e. it will not search meta parameters. So, in what follows, '`<parameter>:<attribute>` is defined' means two things. If `\nometa` isn't used, it means that `<parameter>` has `<attribute>` or it has a metaparameter with `<attribute>`; on the other hand, if `\nometa` is used, it means that `<parameter>` has `<attribute>`, end of story.

### `\ifattribute<parameter>:<attribute><true><false>`

This returns `<true>` if `<parameter>:<attribute>` is defined, `<false>` otherwise. Since all commands below always check whether `<parameter>:<attribute>` is defined before trying to do anything with the value, this command can be avoided most of the time.

```
\setparameter musician : job = music\par          Good, it is.
\setparameter Zappa :                             Too bad.
  meta      = musician
  firstname = Frank

          % See this space?
\ifattribute Zappa : job {Good, it is.}{...}\par
\nometa\ifattribute Zappa : job {...}{Too bad.}
```

### `\usevalue<parameter>:<attribute>`

#### `\usevalueor<parameter>:<attribute><no value>`

#### `\usevalueand<parameter>:<attribute><value exists><no value>`

The first macro returns the value of `<parameter>:<attribute>` if it exists, or does nothing otherwise. Like all the -or and -and variants below, `\usevalueor` executes `<no value>` in case `<parameter>:<attribute>` doesn't exist, while `\usevalueand` returns the value of

`<parameter>:<attribute>` immediately followed by `<value exists>` (no brace added) if `<parameter>:<attribute>` exists, otherwise it executes `<no value>`.

```
Zappa's job was \usevalueand Zappa : job
                        { (and then some!)}
                        {unknown}
and he played the \usevalueor Zappa : instrument
                        {guitar}.
```

Zappa's job was music (and then some!) and he played the guitar.

↪ `\passvalue<code><parameter>:<attribute>`  
 ↪ `\passvalueor<code><parameter>:<attribute><no value>`  
 ↪ `\passvalueand<code><parameter>:<attribute><value exists><no value>`

These return `<code>{<value>}` if `<parameter>:<attribute>` is defined, with the -or and -and variants as above.

```
\def\whichwas#1{(which was #1)}
Zappa's job \passvalue\whichwas Zappa : job \ took
most of his time, because it's a time-consuming
occupation \nometa\passvalueor\whichwas Zappa : job
                        {(you know which)}.
```

Zappa's job (which was music) took most of his time, because it's a time-consuming occupation (you know which).

↪ `\passvaluenobraces<code><parameter>:<attribute>`  
 ↪ `\passvaluenobracesor<code><parameter>:<attribute><no value>`  
 ↪ `\passvaluenobracesand<code><parameter>:<attribute><value exists><no value>`

These are the same as `\passvalue` and variants except the value of the attribute is concatenated to `<code>` without braces (which means that no braces are added in the process, not that braces are removed from the value if it has any).

↪ `\settovalue<dimen or count><parameter>:<attribute>`  
 ↪ `\settovalueor<dimen or count><parameter>:<attribute><no value>`  
 ↪ `\settovalueand<dimen or count><parameter>:<attribute><value exists><no value>`

This sets the first argument to the value of `<parameter>:<attribute>` if it exists. If the first argument is more than one token (e.g. `\count0` vs. `\parindent`), it must be surrounded by braces; and actually it can even be something like `\advance\count0`. Of course `<dimen or count>` must be a dimension or a count, and the value of `<parameter>:<attribute>` must be accordingly a dimension or a number (Y&X doesn't check either of them).

```
\setparameter para : parskip = 2pt \par
Note that
\settovalueor\parskip para : parskip
      {\parskip=1pt\relax}
(\the\parskip) is basically the same thing as
\parskip=\usevalueor para : foo {1pt\relax}
(\the\parskip).
```

Note that (2.0pt) is basically the same thing as (1.0pt).

What the previous example shows is that since `\usevalue` is thoroughly expandable one can say:

```
\mydimen=\usevalueor parameter : attribute {0pt}
```

and it will set `\mydimen` to the value of `<parameter>: <attribute>` or to 0pt. The difference with `\settovalueor` is that in the construction with `\usevalueor` the assignment is obligatorily made (hence the -or variant), whereas with `\settovalueor` the or-clause can do something else (e.g. send an error message). And `\settovalue` insert a prophylactic `\relax`.

```
\storevalue<command><parameter>:<attribute>
\storevalueor<command><parameter>:<attribute><no value>
\storevalueand<command><parameter>:<attribute><value exists><no value>
  These define <command> as the value of <parameter>:<attribute> if it exists.
```

```
\setparameter Zappa : hairstyle = \moustache\par      macro:->\moustache
\storevalue\beard Zappa : hairstyle
\meaning\beard
```

```
↪\ifvalue<parameter>:<attribute>=<value> <true><false>
```

This returns `<true>` if the value of `<parameter>:<attribute>` is `<value>` and `<false>` otherwise (including unexisting `<parameter>:<attribute>`). Note that when comparing the value of `<parameter>:<attribute>` with `<value>`, catcodes aren't part of the picture. Here `<value>` is delimited by the following space, but there might be optional space after the '=' sign. Because of this, it is not possible to test for the emptiness of a value with `\ifvalue`, i.e.

```
\ifvalue foo : bar = {} {true}{false}
```

won't work. Instead, either use `\ifcasevalue` below or `\passvalue` with an emptiness-tester (e.g. `texapi's \ifemptystring`, since `YAX` is based on `texapi` (what, me, self-advertising?)).



```

\bggroup \catcode`\Z=13
\setparameter foo :
  g: bar = Z \par
\egroup
\edef\foobar{%
  \ifvalue foo : bar = Z {yes}{no},
  even though catcodes are different.}
\meaning\foobar

```

macro:->yes, even though catcodes are different.

```

↪\ifcasevalue<parameter>:<attribute>
    \val<value> <code>
    \val<value> <code>
    ...
\elseval<code>
\endval

```

This executes `<code>` following `<value>` matching the value of `<parameter>:<attribute>`. If `<parameter>:<attribute>` doesn't exist, or matches no `<value>`, then `\elseval` is executed. Once again catcodes aren't taken into account when values are compared. The exact syntax is: `<value>` is anything from `\val` to the next space, and `<code>` is anything that follows up to the next `\val`, `\elseval` or `\endval` (any space on the right is removed, so no need to stick `\val` to `<code>`). Apart from `\endval`, everything here is optional: there might be as many `\val`-clauses as needed, including none, and the `\elseval`-clause need not be present (in which case, if no match occurs, nothing happens). Finally, although this is similar to TeX's primitive `\ifcase`, there's no need to jump before anything with `\expandafter` to avoid bumping into conditional structure.

```

\def\doitalic#1{\it#1}
\setparameter type : font = italic \par
\ifcasevalue type : font
  \val italic \doitalic
  \val bold \dobold
\endval{Some text.}

\edef\foo{%
  \ifcasevalue type : font
    \val bold This is bold
    \elseval This is something else
  \endval}
\meaning\foo

```

*Some text.*

macro:->This is something else

`\deleteattribute<parameter>:<attribute>`

This deletes `<parameter>:<attribute>`, which now responds negatively to all previous commands, as if it was never defined. If `<parameter>:<attribute>` doesn't exist, and if `<parameter>` has meta-parameters with `<attribute>`, those are not deleted. Hence `\nometa` can't be prefixed to `\deleteattribute`, because it doesn't make sense.

## Using another syntax

If you don't like Y<sub>A</sub>X's native syntax, and want for instance good old braces to delimit `<parameter>` and `<attribute>`, you might be tempted to do something like:

```
\def\myusevalue#1#2{\usevalue #1:#2 }
```

On the other hand, if you don't mind Y<sub>A</sub>X's syntax but want other names for the commands, then you'll probably go:

```
\let\myusevalue\usevalue
```

Both are bad ideas. Indeed, in neither example will `\myusevalue` work properly with `\nometa`. Besides, you have to create the `-or` and `-and` variants by hand. Not to mention that in the first example `\myusevalue` wastes time calling `\usevalue` (and what if it is redefined?) when it could be in direct relation with internal code. So here's how to circumvent Y<sub>A</sub>X's syntax and/or create new names.

`\newsyntax<syntax>{\<prefix>}`

This creates commands whose names are `\<prefix><command>` and whose syntax for arguments is `<syntax>`. The latter is a parameter text which must contain `#1` and `#2` (for `<parameter>` and `<attribute>` respectively) with whatever to delimit them. For instance:

```
\newsyntax#1#2{x}  
\newsyntax#1 #2!{y}
```

will create among others an `\xusevalue` command whose usage is `\xusevalue<parameter><attribute>` and a `\yusevalue` command whose usage is `\yusevalue<parameter><attribute>!` and both will do the same thing as `\usevalue`. There must be braces around `<prefix>` and none around `<syntax>` (i.e. the latter is delimited by the left brace of the former). If `<prefix>` is empty, you redefine the default commands, which is dangerous.

To be precise, the commands copied are (omitting the `-or` and `-and` variants, which are created too, if any): `\ifattribute`, `\usevalue`, `\passvalue`, `\passvaluenobraces`, `\settovalue`, `\storevalue`, `\ifvalue`, `\ifcasevalue`, `\deleteattribute` and `\restrictattribute` (which you'll learn about in the next section).

**\copysyntax***<prefix1><prefix2>*

This defines all the commands above with *<prefix1>* as those same commands with *<prefix2>*.

**\letyaxcommand***<command1><command2>*

This at the very least `\let<command1> to <command2>`. Besides, if *<command2>* can take a `\nometa` prefix, *<command1>* can too. Finally, if *<command2>* has `-or` and `-and` variants, these are created with *<command1>*. E.g.:

```
\letyaxcommand\defval\storevalue
```

defines `\defval`, `\defvalor` and `\defvaland` as `\storevalue` and its variants. If *<command2>* has been created with `\newsyntax` or `\copysyntax`, *<command1>* of course has the same syntax.

```
\newsyntax#1#2{x}          no
\letyaxcommand\uv\xusevalue
\uvand{noparameter}{noattribute}{yes}{no}
```

## Restrictions on parameters and attributes

**\restrictparameter***<list of parameters>:<list of attributes>\par*

After this declaration, the *<parameter>*'s in *<list of parameters>* (where they are separated by space as in `\setparameter`) can take only those *<attribute>*'s in *<list of attributes>* (which are also separated by space). It affects `\setparameter` only, producing an error message when an *<attribute>* not belonging to *<list of attributes>* is given a value (and the assignment isn't made, of course). Even if it doesn't belong to *<list of attributes>*, the meta attribute is always allowed. Several `\restrictparameter` declarations on the same parameter(s) actually accumulate the allowed attributes in *<list of parameters>*, e.g. after

```
\restrictparameter foo : one two three\par
\restrictparameter foo : four\par
```

one can set the attributes one, two, three and four for foo, and not only four. The idea behind `\restrictparameter` is not so much hiding attributes from the user as making the use of a parameter clearer by indicating which attributes are in use with it, especially if it can be executed (see below).

`\restrictattribute<parameter>:<attribute> <list of values>\par`

This restricts `<parameter>:<attribute>` to take only the values in `<list of values>` (separated by space).

`\restrictallattributes<attribute> <list of values>\par`

This restricts `<attribute>`, whatever the `<parameter>` in which it appears, to take only the values in `<list of values>`. In this command, `<attribute>` is found as anything before the first space, e.g.:

```
\restrictallattributes attribute value1 value2 value3\par
```

```
\restrictallattributes {attri bute} value1 value2 value3\par
```

(the second example if you want to have space in attribute names). Note that if an attribute is restricted with both `\restrictattribute` and `\restrictallattributes`, only the former restriction holds. E.g.:

```
\restrictattribute foo:bar one\par
```

```
\restrictallattributes bar two three\par
```

```
\setparameter foo:
```

```
bar = two % Will produce an error message.
```

```
...
```

## Defining parameters

`\defparameter<list of parameters>{<definition>}`

Parameters aren't just a way of organizing attributes. They can have a definition and act as commands whose arguments are the values of their attributes. The `<list of parameters>` (with parameters once again separated by space) must be braceless, whereas `<definition>` must be enclosed in braces (like a real definition). No parameter text is allowed. However, `<definition>` can contain `'#1'`, which doesn't refer to any argument but to the parameter being defined instead, so that one can use the commands defined in the previous section without specifying the name of the parameter. E.g.

```
\defparameter foo bar {%
```

```
\usevalue #1 : one
```

```
\passvalueor\mycomm #1 : whatever {...}%
```

```
...}
```

defines `foo` and `bar` respectively to

```

\usevalue foo : one
\passvalueor\mycomm foo : whatever {...}%
...
\usevalue bar : one
\passvalueor\mycomm bar : whatever {...}%
...

```

↪ **\executeparameter**<parameter>:

This executes the definition of <parameter> with the the latest values of its attributes. If <parameter> hasn't been defined, nothing happens.

```

\defparameter foo {The value is \usevalue#1:bar }           The value is whatever
\setparameter foo : bar = whatever\par
\executeparameter foo:

```

**\defactiveparameter**<list of parameters>{<definition>}

This does the same thing as \defparameter, i.e. define the parameters in <list of parameters>, but it also set them as 'active,' which means that they're automatically executed each time their attributes are defined with \setparameter. You can still use \executeparameter.

```

\setparameter metasection : skip = 2 font = \it\par      And now we're going to have a new section.
\setparameter mysection : % not yet active
  meta = metasection\par
\defactiveparameter mysection {%                          A new section
  \vskip \usevalueor #1 : skip 0\baselineskip             Fascinating. Once again?
  {\usevalue #1 : font \usevalue #1 : title }%
  \par}

And now we're going to have a new section.               Once again
\setparameter mysection : title = "A new section"\par    Cool.
Fascinating. Once again?
\setparameter mysection : title = "Once again"\par
Cool.

```

```

\setparameter metasection :
  skip    = 2
  inline  = false
  font    = \it

\setparameter mysection mysubsection :
  meta    = metasection

\setparameter mysection :
  font    = \sc

\setparameter mysubsection :
  skip    = 1
  inline  = true

\defparameter mysection mysubsection {%
  \vskip \usevalueor #1 : skip 0\baselineskip
  {\usevalue #1 : font \usevalue #1 : title }%
  \ifvalue #1 : inline = true {. \ignorespaces}%
  \par}%

\def\section#1{%
  \setparameter mysection : title = {#1}\par
  \executeparameter mysection :}

\def\subsection#1{%
  \setparameter mysubsection : title = {#1}\par
  \executeparameter mysubsection :}

... and this is the end of our paragraph.
\section{New ideas}
Here we are going to expose bold new ideas.
\subsection{First bold new idea}
Lore, aim, hip, sum... what do you think about
it? Pig latin, you say?
\subsection{Second bold new idea}
Perhaps Do lore, aim... then, huh?
\section{New new ideas}
Etc.

```

... and this is the end of our paragraph.

NEW IDEAS

Here we are going to expose bold new ideas.

*First bold new idea.* Lore, aim, hip, sum... what do you think about it? Pig latin, you say?

*Second bold new idea.* Perhaps Do lore, aim... then, huh?

NEW NEW IDEAS

Etc.