fancy-units

https://github.com/janekfleper/typst-fancy-units
Version 0.1.1
Requires Typst 0.11+

1 Introduction

Since a comparison to the LaTeX package siunitx is inevitable for a units package, I will get this out of the way immediately. I used the same names for the functions `num()`, `unit()`, `qty()` etc. and tried to use same (or at least similar) names for the options. However, this package is not supposed to be a port of siunitx. There are already two Typst packages available that aim to replace siunitx, namely unify and metro. For the formatting of numbers there is also the package zero.

My goal was to create a package to format numbers and units that makes use of the Typst language and the built-in styling as much as possible. This package therefore does not have to be nearly as complex as siunitx, which I consider a good thing. I am definitely planning to implement more features over time, but I kept the initial version rather simple and somewhat opinionated by design.

For the impatient reader I will already show a few examples. Please refer to the later sections for the parameters of the functions and more examples to showcase all the available options.

`num[0.9]`	0.9
`num[-0.9 (*1*)]`	$-0.9 \pm \mathbf{0.1}$
`num[0.9 +-#text(red)[0.1] e1]`	$(0.9 \pm 0.1) \times 10^{1}$
`unit[kg m^2 / s]`	$\mathrm{kg\,m^{2}\,s^{-1}}$
`unit[#math.cancel[µg]]`	$\cancel{\mu g}$
`unit[_E_#sub[rec]]`	E_{rec}
`unit[#sym.planck Hz]`	$h\,\mathrm{Hz}$
`qty[0.9][g]`	$0.9\,\mathrm{g}$
`qty[27][_E_#sub[rec]]`	$27\,E_{\mathrm{rec}}$

The input for numbers and units is just regular Typst content in markup mode that can be styled with the functions that are already available in Typst. Writing the units does not require any variables or macros for the prefixes and base units. The parser strips off the styling and stores the functions together with the number and unit content. During the processing the numbers and units are converted to your desired output format, and the styling is applied again when the content is actually formatted.

In Section 2 I will go into the details of the styling and explain some of the known limitations. I will give a summary of the available configuration options in Section 3. The functions `num()`, `unit()` and `qty()` are then shown in Section 4, Section 5 and Section 6 respectively with many examples to highlight the capabilities of the packages.

If you have found a bug or if you have any suggestions how I could improve the package, please feel free to open an issue or a pull request on https://github.com/janekfleper/typst-fancy-units. I am also active on the Typst forum if you want to reach out to me https://forum.typst.app/u/janekfleper.

2 Styling

This package allows you to wrap parts of the numbers and units into styling functions. During the parsing the content is unwrapped until there is only the actual text left. The styling functions are saved in a stack alongside the text in a so-called content tree. If necessary, the text is then modified according to the format options. During the formatting the styling functions are applied to the text again to get the desired output.

Since the body has to follow the syntax rules of markup content, there are situations where spaces are required when you are using styling functions. There is no way to ignore a syntax error in the body, the content must always be valid before it can be parsed. If you are calling a function in the content of a number or a unit, make sure to put a space in front of succeeding parentheses (or brackets) that are not supposed to be part of the function. For numbers this is only relevant when you are using relative uncertainties. With units this can be an issue whenever you are grouping units with parentheses (or brackets).

2.1 Supported functions

This table gives you an overview of the styling functions that are currently supported for numbers and units. The support for quantities is equivalent to `num[]` and `unit[]` for the respective parts. Which styling functions are actually useful is for you to decide.

function	`num[]`	`unit[]`	Notes
`*bold*`	**0.9**	**kg**	
`_emph_`	*0.9*	*kg*	Support for `num[]` depends on the font
`text(..)[]`	0.9	kg	
`overline[]`		$\overline{\text{kg}}$	
`underline[]`		$\underline{\text{kg}}$	
`strike[]`		~~kg~~	
`sub[]`		kg_{abc}	The subscript will be passed to `attach(br: )`
`super[]`		$^{\text{kg}}$	The superscript is treated like a separate unit
`math.cancel[]`	~~0.9~~	$\cancel{\text{kg}}$	
`math.display[]`		kg^2	The parameter `cramped` has no effect
`math.inline[]`		kg^2	Equivalent to the function `math.display[]`
`math.script[]`	0.9	kg^2	
`math.sscript[]`	0.9	kg^2	
`math.bold[]`	**0.9**	**kg**	Equivalent to the function `*bold*`
`math.italic[]`	*0.9*	*kg*	Equivalent to the function `_emph_`
`math.sans[]`	0.9	kg	Support for `num[]` depends on the font
`math.frak[]`	0.9	𝔨𝔤	Support for `num[]` depends on the font
`math.mono[]`	0.9	kg	Support for `num[]` depends on the font
`math.bb[]`	𝟘.𝟡	𝕜𝕘	Support for `num[]` depends on the font
`math.cal[]`	0.9	$\mathcal{kg}$	Support for `num[]` depends on the font
`math.overline[]`	$\overline{0.9}$	$\overline{\text{kg}}$	Different spacing than the regular `overline[]`
`math.underline[]`	$\underline{0.9}$	$\underline{\text{kg}}$	Different spacing than the regular `underline[]`

3 Configuration

The settings to configure the output format of the numbers and the units are kept in a state and will be used as the default. This state should be set at the beginning of the document to configure the global format. The format can always be changed for individual numbers and units by using the respective function arguments that will take precedence over the state.

```
fancy-units-configure(
  decimal-separator: auto string content,
  uncertainty-mode: string,
  unit-separator: content,
  per-mode: string,
  quantity-separator: content
)
```

decimal-separator `auto` or `string` or `content`

The symbol to separate the integer part from the decimal part.

This only affects the output. The input must always use the decimal point `"."` as separator.

If the separator is set to `auto`, the appropriate symbol based on the text language will be used.[1]

Default: `auto`

uncertainty-mode `string`

The output format for the (symmetric) uncertainties.

See the parameter of `num()` in Section 4.1 for the details.

Default: `"plus-minus"`

unit-separator `content`

The separator between units.

See the parameter of `unit()` in Section 5.1 for the details.

Default: `h(0.2em)`

per-mode `string`

The output format for units with negative exponents.

See the parameter of `unit()` in Section 5.1 for the details.

Default: `"power"`

quantity-separator `content`

The separator between the number and the unit.

See the parameter of `qty()` in Section 6.1 for the details.

Default: `h(0.2em)`

[1] According to https://en.wikipedia.org/wiki/Decimal_separator#Conventions_worldwide

4 Numbers

A number consisting of a value, uncertainties and an exponent.

The value component is required to have a valid number, whereas the uncertainties and the exponent are optional components. The parsing is only successful if everything in the number can be matched. Even if just one of the components has an invalid format, an error will be raised.

`num[0.9]`	0.9
`num[0.9e1]`	0.9×10^{1}
`num[-0.9 +-0.1 e1]`	$(-0.9 \pm 0.1) \times 10^{1}$

4.1 Parameters

```
num(
  decimal-separator: auto string content,
  uncertainty-mode: auto string,
  body: content
) -> content
```

decimal-separator `auto` or `string` or `content`

The symbol to separate the integer part from the decimal part.

By default the separator stored in the `fancy-units-state` will be used, see Section 3 for the details.

Default: `auto`

uncertainty-mode `auto` or `string`

The output format for the (symmetric) uncertainties.

Symmetric uncertainties can be converted to the other format. Asymmetric uncertainties will ignore this option and automatically use the output format `"plus-minus"`. See the examples in Section 4.2 to understand how the conversion between the input format and the output format works.

By default the format stored in the `fancy-units-state` will be used, see Section 3 for the details.

Variant	Details
`"plus-minus"`	The (absolute) uncertainties are preceded by ±.
`"parentheses"`	The (relative) uncertainties are wrapped in parentheses ().
`"conserve"`	The input format of the uncertainties will be conserved (if possible).

Default: `auto`

body `content` *Required Positional*

The actual number to be parsed and formatted.

The number must contain a value, while the uncertainties and the exponent are optional. The uncertainties can be either symmetric or asymmetric and absolute or relative to the value.

`value` `+-` `uncertainty` `e` `exponent` or `value` `(` `uncertainty` `)e` `exponent`

The value can either be an integer or a floating point number.

The exponent is prefixed by an `e` or `E` and must always be at the end of the number. It can either be an integer or a floating point number.

There is no limit to the number of uncertainties, the parser will try to interpret everything after the value (and before the exponent) as uncertainties. If an uncertainty is prefixed by `+-`, it is interpreted as an *absolute* uncertainty. Absolute uncertainties can either be an integer or a floating point number.

An uncertainty wrapped in parentheses `()` will be interpreted *relative* to the value. Relative uncertainties must always be an integer. A floating point number will result in an error due to an invalid number format.

4.2 Examples

4.2.1 `uncertainty-mode`

As explained earlier in Section 4.1 the `uncertainty-mode` only affects the output of the numbers. The input will always be parsed without taking the `uncertainty-mode` into account. Spaces around the signs `+` and `-`, the parentheses `()` and the exponent characters `e` or `E` are allowed and will not affect the output. For absolute uncertainties it is considered best practice to put a space before `+-` to improve the readability. If the uncertainty is asymmetric, the space should be put before both signs.
A space before the exponent character can be useful to highlight that the exponent affects the entire number. This is especially true in the case of absolute uncertainties. Since parentheses around the value and the uncertainties are not required in the number input, a space can highlight that the exponent does not belong to the (last) uncertainty.

`uncertainty-mode:`	`"plus-minus"`	`"parentheses"`	`"conserve"`
`num[0.9 +-0.1]`	0.9 ± 0.1	$0.9(1)$	0.9 ± 0.1
`num[0.9(1)]`	0.9 ± 0.1	$0.9(1)$	$0.9(1)$
`num[0.9 +-0.1 e1]`	$(0.9 \pm 0.1) \times 10^{1}$	$0.9(1) \times 10^{1}$	$(0.9 \pm 0.1) \times 10^{1}$
`num[0.9(1)e1]`	$(0.9 \pm 0.1) \times 10^{1}$	$0.9(1) \times 10^{1}$	$0.9(1) \times 10^{1}$
`num[0.9 +-0.1 +-0.2]`	$0.9 \pm 0.1 \pm 0.2$	$0.9(1)(2)$	$0.9 \pm 0.1 \pm 0.2$
`num[0.9(1)(2)]`	$0.9 \pm 0.1 \pm 0.2$	$0.9(1)(2)$	$0.9(1)(2)$
`num[0.9 +0.1 -0.2]`	$0.9^{+0.1}_{-0.2}$	$0.9^{+0.1}_{-0.2}$	$0.9^{+0.1}_{-0.2}$
`num[0.9(1:2)]`	$0.9^{+0.1}_{-0.2}$	$0.9^{+0.1}_{-0.2}$	$0.9^{+0.1}_{-0.2}$
`num[0.9(1:2)e1]`	$(0.9^{+0.1}_{-0.2}) \times 10^{1}$	$(0.9^{+0.1}_{-0.2}) \times 10^{1}$	$(0.9^{+0.1}_{-0.2}) \times 10^{1}$

4.2.2 Styling

When styling the components in a number, there are a few (syntax) rules to follow. The styling functions are attached to the components before the number is actually parsed. If done correctly, the styling will therefore not affect the interpretation of the number.

It is sufficient to apply the styling to the actual components. The accompanying characters `+-`, `()` or `eE` do not have to be included in the styling functions. In either case only the actual component will be styled in the output. Styling the accompanying characters is (currently) not possible.

`uncertainty-mode:`	`"plus-minus"`	`"parentheses"`	`"conserve"`
`num[#text(red)[-0.9] (1)]`	-0.9 ± 0.1	$-0.9(1)$	$-0.9(1)$
`num[0.9 #text(red)[(1)] e1]`	$(0.9 \pm 0.1) \times 10^{1}$	$0.9(1) \times 10^{1}$	$0.9(1) \times 10^{1}$
`num[0.9 *+-0.1* e1]`	$(0.9 \pm \mathbf{0.1}) \times 10^{1}$	$0.9(\mathbf{1}) \times 10^{1}$	$(0.9 \pm \mathbf{0.1}) \times 10^{1}$
`num[-0.9 (1) #text(red)[e1]]`	$(-0.9 \pm 0.1) \times 10^{1}$	$-0.9(1) \times 10^{1}$	$-0.9(1) \times 10^{1}$
`num[0.9 +0.0 #text(red)[-0.1]]`	$0.9^{+0.0}_{-0.1}$	$0.9^{+0.0}_{-0.1}$	$0.9^{+0.0}_{-0.1}$

5 Units

A unit can be anything from a single character to a complex structure with fractions, brackets and groups. It is not necessary to use variables for the prefixes and units, you can just write them down directly. The parser will figure out the exponents, brackets, etc. and the unit will then be formatted accordingly.

`unit[µg]`	$\mu\mathrm{g}$
`unit[(m s)^2]`	$\mathrm{m}^{2}\,\mathrm{s}^{2}$
`unit[kg m/s^2]`	$\mathrm{kg}\,\mathrm{m}\,\mathrm{s}^{-2}$

5.1 Parameters

```
unit(
  decimal-separator: auto string content,
  unit-separator: auto content,
  per-mode: auto string,
  body: content
) -> content
```

decimal-separator `auto` or `string` or `content`

The symbol to separate the integer part from the decimal part in exponents.

By default the separator stored in the `fancy-units-state` will be used, see Section 3 for the details.

Default: `auto`

unit-separator `auto` or `content`

The separator to join the units.

After the individual units are formatted they are joined by the separator. The most common choice will be a small amount of horizontal space to visually separate the units. Other typical options are the symbols `sym.dot` [·] or `sym.times` [×].

By default the separator stored in the `fancy-units-state` will be used, see Section 3 for the details.

Default: `auto`

per-mode `auto` or `string`

The output format for units with negative exponents.

This option only affects the output format. The parser will not save any information about the input format. The `"conserve"` option therefore does not exist here.

By default the format kept in the `fancy-units-state` will be used, see Section 3 for the details.

Variant	Details
`"power"`	The negative exponent will be applied directly, e.g. $\mathrm{m\,s^{-2}}$
`"fraction"`	An actual fraction is used, e.g. $\frac{\mathrm{m}}{\mathrm{s}^2}$
`"slash"`	A forward slash is used to indicate a fraction, e.g. m/s^2

Default: `auto`

body `content` *Required* *Positional*

The actual unit(s) to be parsed and formatted.

Since the body is just regular content, the usual restrictions of math input do not apply. You can just write down the units separated by spaces and it is not necessary to use variables for the prefixes and units. Unicode characters such as the prefix µ also work directly or as a hexadecimal escape sequence `\u{03bc}`.

The parser will try to match pairs of parentheses, brackets and curly brackets. If not all of them can be matched, an error will be raised. A single pair of parentheses `()` will only group the units inside. This is for example useful to apply an exponent to multiple units. If you want to actually have the parentheses in the output, you have to use two pairs `(())`.

You can apply styling to multiple units, to a single unit or just to a part of a unit, e.g. the prefix. If the styling is only applied to the prefix, you have to use a colon `:` to join the prefix and the unit again. Otherwise the parser will not understand that the two components belong to the same unit.

5.2 Macros

Macros enable you to define units or unit prefixes that will be inserted automatically if you use the macro in a unit or quantity anywhere in your document. This feature is designed for complicated units or units that directly include styling. In that case it can be annoying to write down the entire unit every time. You should not put trivial macros like `meter: [m]` in here to make the package work like `siunitx` and the other Typst unit packages again. The macros are compatible with all unit features such as exponents, subscripts, styling and grouping. See the examples in Section 5.3.4 for more details.

`add-macros(..acros: content string symbol)`

macros `content` or `string` or `symbol`

The names of the macros must only contain alphanumeric characters. Underscores are not allowed since they are used for italic styling in the units parser.

The values of the macros should be content. If a string or a symbol are passed, they will be wrapped inside content automatically. This is required since the macros are interpreted by the same functions as the regular units.

5.3 Examples

5.3.1 `per-mode`

As explained earlier in Section 5.1 the `per-mode` only affects the output of the units. For the input format you most likely want to use a slash `/` to indicate a fraction, but it is also valid to use negative exponents. The slash will only affect the first trailing unit, use parentheses or (curly) brackets to apply the fraction to multiple units.

`per-mode:`	`"power"`	`"fraction"`	`"slash"`
`unit[m / s]`	$\mathrm{m\,s^{-1}}$	$\frac{\mathrm{m}}{\mathrm{s}}$	$\mathrm{m/s}$
`unit[kg^-2]`	$\mathrm{kg^{-2}}$	$\frac{1}{\mathrm{kg}^2}$	$\mathrm{1/kg^2}$
`unit[kg m / s^2]`	$\mathrm{kg\,m\,s^{-2}}$	$\mathrm{kg}\,\frac{\mathrm{m}}{\mathrm{s}^2}$	$\mathrm{kg\,m/s^2}$
`unit[kg / (m s)]`	$\mathrm{kg\,m^{-1}\,s^{-1}}$	$\frac{\mathrm{kg}}{\mathrm{m\,s}}$	$\mathrm{kg/m\,s}$

The `per-mode` `"slash"` can be ambiguous when a slash is followed by multiple units. If you want to prevent this ambiguity, wrap the units in a second pair of parentheses as shown in Section 5.3.2.

5.3.2 Grouping

You can group units with parentheses or (curly) brackets. A single pair of parentheses will *silently* group the units, only the second pair is actually included in the formatted output. This replicates the behaviour of parentheses in a fraction in math mode. Brackets and (curly) brackets are always included in the formatted output.

Since brackets `[]` are also the macro for `content`, this can sometimes lead to unexpected behaviour. This is just something to keep in mind if you absolutely have to use brackets in a unit.

`per-mode:`	`"power"`	`"fraction"`	`"slash"`
`unit[kg / (m s)]`	$\mathrm{kg\,m^{-1}\,s^{-1}}$	$\frac{\mathrm{kg}}{\mathrm{m\,s}}$	$\mathrm{kg/m\,s}$
`unit[kg / ((m s))]`	$\mathrm{kg\,(m\,s)^{-1}}$	$\frac{\mathrm{kg}}{(\mathrm{m\,s})}$	$\mathrm{kg/(m\,s)}$
`unit[(kg m) / s]`	$\mathrm{kg\,m\,s^{-1}}$	$\frac{\mathrm{kg\,m}}{\mathrm{s}}$	$\mathrm{kg\,m/s}$
`unit[[kg m] / s]`	$\mathrm{[kg\,m]\,s^{-1}}$	$\frac{\mathrm{[kg\,m]}}{\mathrm{s}}$	$\mathrm{[kg\,m]/s}$
`unit[{kg m} / s]`	$\mathrm{\{kg\,m\}\,s^{-1}}$	$\frac{\mathrm{\{kg\,m\}}}{\mathrm{s}}$	$\mathrm{\{kg\,m\}/s}$

If you wrap a single unit in parentheses, its power will be *protected* from the `per-mode`. This can be useful if you are using the `"fraction"` mode and you want to prevent nested fractions since they can be difficult to read. In the `"slash"` mode nesting is not possible anyway and protecting individual units will not have any effect on the output. If you already have the `per-mode` set to `"power"`, the behaviour of protected units can be a bit weird since the multiple powers will be attached one by one.

`per-mode:`	`"power"`	`"fraction"`	`"slash"`
`unit[kg / (m^-1 s)]`	$\mathrm{kg\,m\,s^{-1}}$	$\frac{\mathrm{kg}}{\frac{1}{\mathrm{m}}\,\mathrm{s}}$	$\mathrm{kg/m^{-1}\,s}$
`unit[kg / ((m^-1) s)]`	$\mathrm{kg\,m^{-1^{-1}}\,s^{-1}}$	$\frac{\mathrm{kg}}{\mathrm{m^{-1}\,s}}$	$\mathrm{kg/m^{-1}\,s}$

5.3.3 Styling and Joining

You can apply styling to (mulitple) units or just to a part of a unit. The styling functions are attached to the (group of) units and components inside. E.g. if there is a fraction or an exponent in the styling function, they will also be formatted accordingly.

It is also possible to apply the styling only to the base unit or only to the exponent. The parser will just attach an exponent to the previous unit, the separation by the styling functions is therefore not an issue.

If a unit is split up into multiple parts due to the styling, you can use a colon to join the components again. This is useful when you want to apply styling only to the prefix or the base unit. In addition this is also necessary to include a Typst variable in a unit.

Since the underscore character _ is reserved for *italic* styling you have to use the function `sub()` to add a subscript to a unit. As for an exponent, the parser will attach the subscript to the previous unit and the formatter will use the function `math.attach()`. If a unit has both an exponent and a subscript, everything will therefore be formatted correctly.

`unit[*kg* m / s]`	$\mathrm{kg\,m\,s^{-1}}$
`unit[_E_#sub[rec]^2]`	E_{rec}^{2}
`unit[#text(red)[µ]:m^2]`	$\mu\mathrm{m}^{2}$
`unit[m#math.cancel[^2] / (#math.cancel[m] s)]`	$\mathrm{m}^{2}\,\cancel{\mathrm{m}^{-1}}\,\mathrm{s}^{-1}$

5.3.4 Macros

The easiest example for a macro is the prefix µ. If you don't want to type that letter directly (or use `sym.mu`), you can define a macro that replaces the letter u with µ. For the macro to work correctly, you have to join the prefix and the unit with a colon. Writing `unit[u:m]` will then return µm.

```
#add-macros(u: sym.mu)
```

The macros do not care whether something is supposed to be a prefix or a unit, it is up to you to join everything correctly.

If you have a composite unit that you use often, defining this as a macro has another advantage besides making it faster to type. Any changes to the unit will be automatically applied to the entire document, you wont have to update all the instances of the unit manually.

```
#add-macros(
  m2: [m^2],
  aB: [_a_#sub[B]],
  au: [arb. unit],
  verdet: [rad / (T m)],
)
```

With the macros defined above and the macro for the prefix µ, see the follwing examples:

`unit[u#sub[2]]`	μ_{2}
`unit[#text(red)[u]:m2^2]`	$\mu\mathrm{m}^{4}$
`unit[((m2))^2]`	$\left(\mathrm{m}^{2}\right)^{2}$
`unit[aB^2]`	a_{B}^{2}
`unit[au]`	$\mathrm{arb.\ unit}$
`qty[137][verdet]`	$137\ \mathrm{rad\,T^{-1}\,m^{-1}}$

6 Quantities

A quantity combines a number and a unit in a single function. The parsing and formatting of the two components is completely separated. Internally, the function `qty()` just calls the functions `num()` and `unit()` and adds a separator between the two.

`qty[0.9][g]`	$0.9\,\mathrm{g}$
`qty[6][W / kg]`	$6\,\mathrm{W\,kg^{-1}}$
`qty[33][G / cm]`	$33\,\mathrm{G\,cm^{-1}}$

6.1 Parameters

```
qty(
  decimal-separator: auto string content,
  uncertainty-mode: auto string,
  unit-separator: auto content,
  per-mode: auto string,
  quantity-separator: auto content,
  number: content,
  unit: content
) -> content
```

decimal-separator `auto` or `string` or `content`

The symbol to separate the integer part from the decimal part.

See the parameter of `num()` in Section 4.1 for the details.

Default: `auto`

uncertainty-mode `auto` or `string`

The output format for the (symmetric) uncertainties.

See the parameter of `num()` in Section 4.1 for the details.

Default: `auto`

unit-separator `auto` or `content`

The separator to join the units.

See the parameter of `unit()` in Section 5.1 for the details.

Default: `auto`

per-mode `auto` or `string`

The output format for units with negative exponents.

See the parameter of `unit()` in Section 5.1 for the details.

Default: `auto`

quantity-separator `auto` or `content`

The separator to join the number and the unit.

After the number and the unit are parsed and formatted they are joined by the separator. A small horizontal space is probably the only reasonable choice here.

By default the separator stored in the `fancy-units-state` will be used, see Section 3 for the details.

Default: `auto`

number `content` *Required Positional*

The number to be parsed and formatted.

See the `body` of `num()` in Section 4.1 for the details.

unit `content` *Required Positional*

The unit to be parsed and formatted.

See the `body` of `unit()` in Section 5.1 for the details.

6.2 Examples

6.2.1 `quantity-separator`

There are situations where you might want to adjust the space between the number and the unit. If the number has an exponent or the unit is a variable wrapped in `math.emph()`, it can be nice to slightly reduce the spacing. You are of course free to use other symbols to separate the number and the unit, but even `sym.dot` · just does not look right.

```
qty(quantity-separator: h(0.1em))[0.9e-3][kg]
```
$0.9 \times 10^{-3}\,\mathrm{kg}$

```
qty(quantity-separator: h(0.2em))[0.9e-3][kg]
```
$0.9 \times 10^{-3}\,\mathrm{kg}$

```
qty(quantity-separator: h(0.1em))[27][_E_#sub[rec]]
```
$27\,E_{\mathrm{rec}}$

```
qty(quantity-separator: h(0.2em))[27][_E_#sub[rec]]
```
$27\,E_{\mathrm{rec}}$