# LaTeX for authors
# current version
# 著者のための LaTeX
# 最新版



2026-02-12

## Contents 目次





AI Disclosure This translation was produced using multiple machine translation tools, alongside several free, no-registration generative AI services to verify the content and refine the translated text. The Japanese translation follows the English original to facilitate comparison and verification. All marginal notes are comments intended for the translator's own reference.
この翻訳は、複数の機械翻訳を使って行い、内容の確認と訳文の整理のためにユーザ登録不要の無料で使うことのできる複数の生成 AI を使いました。原文との比較と翻訳内容を確認するために英語の原文の後に日本語訳を入れました。すべてのマージンノートは訳者自身のためのコメントです。

# 1 Introduction はじめに

LaTeX 2ε was released in 1994 and added a number of then-new concepts to LaTeX. These are described in `usrguide-historic`, which has largely remained unchanged. Since then, the LaTeX team have worked on a number of ideas, firstly a programming language for LaTeX (expl3) and then a range of tools for document authors which build on that language. Here, we describe *stable* and *widely-usable* concepts that have resulted from that work. These 'new' ideas have been transferred from development packages into the LaTeX 2ε kernel. As such, they are now available to *all* LaTeX users and have the *same stability* as any other part of the kernel. The fact that 'behind the scenes' they are built on expl3 is useful for the development team, but is not directly important to users.

LaTeX 2ε は 1994 年に公開され、その当時新しい多くの概念を LaTeX に導入しました。これらは `usrguide-historic` で説明していますが、その内容は現在にいたるまでほとんど変わっていません。その後に、LaTeX 開発チームは多くのアイデアに取り組んできました。最初は、LaTeX のためのプログラミング言語（つまり expl3）を開発し、次にその言語を土台にした文書作成者向けのさまざまなツールの開発に取り組みました。ここでは、その作業の結果として得られた中から、安定していてそして幅広く利用できる概念と考えかたについて説明します。これらの「新しい」アイデアは、開発パッケージから LaTeX 2ε カーネルに取り込まれました。そのため、すべての LaTeX ユーザーが利用できるようになり、カーネルの他の部分と同じ安定性を備えています。これらが「舞台裏」で expl3 に基づいて構築されているという事実は、開発チームにとっては有益ですが、一般のユーザーにとって直接重要なことではありません。

訳注：expl3 は、L3 プログラミング層と呼ばれることもあるようです。expl3 is apparently also referred to as the L3 programming layer.

# 2 Creating document commands and environments ドキュメントコマンドと環境の作成

## 2.1 Overview 概要

Creating document commands and environments using the LaTeX3 toolset is based around the idea that a common set of descriptions can be used to cover almost all argument types used in real documents. Thus parsing is reduced to a simple description of which arguments a command takes: this description

provides the 'glue' between the document syntax and the implementation of the command.

LaTeX3 のツールセットを使用したドキュメントコマンドと環境の作成は、実際のドキュメントで使用されるほとんどすべての引数タイプをカバーするために、共通の記述セットを使用できるという考えに基づいています。したがって、必要なのはコマンドがどのような引数を取るかを簡潔に記述することだけになります。この記述は、ドキュメントの構文とコマンドの実装の間の「接着剤」となります。

First, we will describe the argument types, then move on to explain how these can be used to create both document commands and environments. Various more specialized features are then described, which allow an even richer application of a simple interface set up.

まず最初に、引数の型について説明し、次にこれらを使ってドキュメントコマンドと環境の両方を作成する方法について説明します。次に、より特殊なさまざまな機能について説明します。これにより、簡潔に設計したインターフェースを、より豊かに応用できるようになります。

The details here are intended to help users create document commands in general. More technical detail, suitable for TeX programmers, is included in `interface3`.

ここで述べる詳細は、一般のユーザーがドキュメントコマンドを作成する際の助けとなることを意図しています。TeX プログラマに適した技術的な詳細は、`interface3` に含まれています。

## 2.2 Describing argument types 引数の型について

In order to allow each argument to be defined independently, the parser does not simply need to know the number of arguments for a function, but also the nature of each one. This is done by constructing an *argument specification*, which defines the number of arguments, the type of each argument and any additional information needed for the parser to read the user input and properly pass it through to internal functions.

各引数を独立に定義できるようにするには、パーサーは関数の引数の個数だけでなく、それぞれの引数の種類も知っていなければなりません。これは、引数の個数、各引数の型、さらにパーサーがユーザー入力を取得して内部関数へ適切に渡すために必要な追加情報を定める引数指定（argument specification）を与えることで実現されます。

The basic form of the argument specifier is a list of letters, where each letter defines a type of argument. As will be described below, some of the types need additional information, such as default values. The argument types can be divided into two, those which define arguments that are mandatory (potentially raising an error if not found) and those which define optional arguments. The mandatory types

引数指定子の基本形は文字が並んだものであり、各文字が 1 つの引数型を表します。後述するように、型によってはデフォルト値などの追加情報が必要な場合があります。引数型は、必須引数を定義するもの（見つからない場合にはエラーになることがある）と、オプション引数を定義するものの 2 つに分けられます。必須の型は以下の通りです。

`m` A standard mandatory argument, which can either be a single token alone or multiple tokens surrounded by curly braces `{}`. Regardless of the input, the argument will be passed to the internal code without the outer braces. This is the type specifier for a normal TEX argument.

標準的な必須引数であり、単一のトークン、または波括弧（brace）`{}`で囲まれた複数のトークンのいずれかとなります。入力がどのような形であれ、引数は外側の波括弧を除いた状態で内部のコードに渡されます。これは通常 TEX 引数のための型指定子です。

訳注：トークン（token）とは日常語では何かの印や切符などの価値の単位を表し単語です。コンピュータでは処理の最小単位のことです。In everyday words, a *token* refers to a mark or a unit of value, such as a ticket. In computing, it represents the smallest unit of processing.

`r` Given as `r`⟨*token1*⟩⟨*token2*⟩, this denotes a 'required' delimited argument, where the delimiters are ⟨*token1*⟩ and ⟨*token2*⟩. If the opening delimiter ⟨*token1*⟩ is missing, the default marker `\NoValue` will be inserted after a suitable error.

`r`⟨*token1*⟩⟨*token2*⟩ の形で与えると、これは区切り付きの必須引数を表します。このとき、区切り記号は ⟨*token1*⟩ と ⟨*token2*⟩ です。開始区切り ⟨*token1*⟩ が欠けている場合には、適切なエラーを出したうえで、既定のマーカー `\NoValue` が挿入されます。

`R` Given as `R`⟨*token1*⟩⟨*token2*⟩`{`⟨*default*⟩`}`, this is a 'required' delimited argument as for `r`, but it has a user-definable recovery ⟨*default*⟩ instead of `\NoValue`.

`R`⟨*token1*⟩⟨*token2*⟩`{`⟨*default*⟩`}`の形で与えると、これは `r` と同様の区切り付き必須引数を表しますが、`\NoValue` の代わりに、ユーザーが指定する回復用の既定値 ⟨*default*⟩ を用います。

`v` Reads an argument 'verbatim', between the following character and its

next occurrence, in a way similar to the argument of the LaTeX $2_\varepsilon$ command `\verb`. Thus a `v`-type argument is read between two identical characters, which cannot be any of `%`, `\`, `#`, `{`, `}` or ␣. The verbatim argument can also be enclosed between braces, `{` and `}`. A command with a verbatim argument will produce an error when it appears within an argument of another command.

LaTeX $2_\varepsilon$ のコマンド`\verb`の引数と同様の方法で、直後の文字を区切り文字として、その次に同じ文字が現れるまでを‘逐語的（verbatim）に読み込みます。つまり、`v`型の引数は2つの同じ文字の間で読み込まれますが、その文字として`%`、`\`、`#`、`{`、`}`、または␣を使うことはできません。逐語的引数は、波括弧`{`と`}`で囲むことも可能です。逐語的引数を持つコマンドは、別のコマンドの引数内で使うとエラーになります。

訳注：“verbatim”はラテン語で「言葉」を意味する語に由来します。そこから「逐語的な」「一言一句そのまま」という意味で使われます。LaTeX では、入力した文字をそのまま出力するということです。`v` 型の引数とは verbatim argument type の略です。セクション 2.14 に説明があります。“verbatim” derives from the Latin word for "word." It is used to mean "verbatim" or "word-for-word." In LaTeX, this means that the input characters are output exactly as they are. The v-type argument stands for "verbatim argument type"; this is explained in Section 2.14.

`b` Only suitable in the argument specification of an environment, it denotes the body of the environment, between `\begin{`⟨*environment*⟩`}` and `\end{`⟨*environment*⟩`}`. See Section 2.11 for details.

環境の引数指定においてのみ使用可能であり、`\begin{environment}`と`\end{environment}`の間に位置する環境の本体を表します。詳細はセクション 2.11 を参照してください。

`c` Only suitable in the argument specification of an environment, it denotes collection of the environment verbatim, between `\begin{`⟨*environment*⟩`}` and `\end{`⟨*environment*⟩`}`. See Section 2.16 for details.

環境の引数指定においてのみ使用可能であり、`\begin{environment}`と`\end{environment}`の間にある環境の内容を逐語的に取得することを表します。詳細はセクション 2.16 を参照してください

The types which define optional arguments are:

オプション引数を定義する型は以下の通りです：

`o` A standard LaTeX optional argument, surrounded with square brackets, which will supply the special `\NoValue` marker if not given (as described later).

標準的な LaTeX のオプション引数で、角括弧（square bracket）で囲まれます。値が指定されない場合、特別なマーカー`\NoValue`が渡されます（後述）。

`d` Given as `d`⟨*token1*⟩⟨*token2*⟩, an optional argument which is delimited by ⟨*token1*⟩ and ⟨*token2*⟩. As with `o`, if no value is given the special marker

`\NoValue` is returned.

`d`⟨*token1*⟩⟨*token2*⟩ の形式で指定されるオプション引数で、⟨*token1*⟩ と ⟨*token2*⟩ によって区切られます。`o` と同様に、値が指定されない場合は特別なマーカー `\NoValue` が返されます。

`O` Given as `O{`⟨*default*⟩`}`, is like `o`, but returns ⟨*default*⟩ if no value is given.

`O{`⟨*default*⟩`}`の形式で指定されます。`o` と同様ですが、値が与えられない場合には ⟨*default*⟩ を返します。

`D` Given as `D`⟨*token1*⟩⟨*token2*⟩`{`⟨*default*⟩`}`, it is as for `d`, but returns ⟨*default*⟩ if no value is given. Internally, the `o`, `d` and `O` types are short-cuts to an appropriated-constructed `D` type argument.

`D`⟨*token1*⟩⟨*token2*⟩`{`⟨*default*⟩`}`の形式で指定されます。`d` と同様ですが、値が指定されない場合は ⟨*default*⟩ が返されます。内部的には、`o`、`d`、`O` は、適切に構成された `D` 型引数の略記です。

`s` An optional star, which will result in a value `\BooleanTrue` if a star is present and `\BooleanFalse` otherwise (as described later).

オプションの星印です。星印が存在する場合は`\BooleanTrue`、存在しない場合は`\BooleanFalse` という値になります（後述）。

`t` An optional ⟨*token*⟩, which will result in a value `\BooleanTrue` if ⟨*token*⟩ is present and `\BooleanFalse` otherwise. Given as `t`⟨*token*⟩.

省略可能な ⟨*token*⟩ を読み取る引数です。⟨*token*⟩ が存在する場合は `\BooleanTrue`、存在しない場合は`\BooleanFalse` になります。`t`⟨*token*⟩ の形で指定します。

`e` Given as `e{`⟨*tokens*⟩`}`, a set of optional *embellishments*, each of which requires a *value*. If an embellishment is not present, `\NoValue` is returned. Each embellishment gives one argument, ordered as for the list of ⟨*tokens*⟩ in the argument specification. All ⟨*tokens*⟩ must be distinct.

`e{`⟨*tokens*⟩`}`の形式で指定される、値を伴う省略可能の修飾子（embellishment）の集りを読み取る引数です。各修飾子には対応する 値（value）が必要です。修飾子が存在しない場合、`\NoValue` が返されます。各修飾子ごとに 1 つの引数が与えられ、その順序は引数指定の中の ⟨*tokens*⟩ の並んだ順序に従います。すべての ⟨*tokens*⟩ は互いに異なるものである必要があります。

`E` As for `e` but returns one or more ⟨*defaults*⟩ if values are not given: `E{`⟨*tokens*⟩`}{`⟨*defaults*⟩`}`. See Section 2.7 for more details.

e と同様ですが、値が与えられない場合は 1 つ以上の ⟨*defaults*⟩ を返します：E{⟨*tokens*⟩}{⟨*defaults*⟩}。詳細はセクション 2.7 を参照してください。

## 2.3 Modifying argument descriptions 引数記述の変更

In addition to the argument *types* discussed above, the argument description also gives special meaning to three other characters.

訳注：“three other characters”, but four. +, !,-, >

前述した引数の型（type）に加えて、引数記述ではさらに 4 つの文字が特別な意味を持ちます。

First, + is used to make an argument long (to accept paragraph tokens). In contrast to \newcommand, this applies on an argument-by-argument basis. So modifying the example to '`s o o +m O{default}`' means that the mandatory argument is now \long, whereas the optional arguments are not.

第一に、+は引数を \long にします。つまり、その引数で段落トークンを受け付けられるようにします。\newcommand とは異なり、この指定は引数ごとに適用されます。したがって、例を '`s o o +m O{default}`' のように変更すると、必須引数だけが \long となり、オプション引数は\long にはなりません。

Secondly, ! is used to control whether spaces are allowed before optional arguments. There are some subtleties to this, as TEX itself has some restrictions on where spaces can be 'detected': more detail is given in Section 2.6.

第二に、!はオプション引数の前に空白を許すかどうかを制御するために使います。ただし、この点には少し微妙なところがあります。TEX 自体に、空白を「検出」できる位置についていくつか制約があるためです。詳細はセクション 2.6 を参照してください。

Thirdly, = is used to declare that the following argument should be interpreted as a series of keyvals. See Section 2.9 for more details.

訳註：Inconsistencies in the notation of 'keyval, 'key–value', 'keyval-form', 'keyval form' in the original text. 原文の keyvals の表記の揺れ。

第三に、=は、その直後の引数をキー–値の並びとして解釈するよう指定するために使います。詳細はセクション 2.9 を参照してください。

訳注：キー–値（keyval）は、何かを表す項目をキー（key）と呼び、キーの具体的な内容を値（value）として対応づける考え方です。

Finally, the character > is used to declare so-called 'argument processors', which can be used to modify the contents of an argument before it is passed to the macro definition. The use of argument processors is a somewhat advanced topic, (or at least a less commonly used feature) and is covered in Section 2.10.

最後に、>は、いわゆる「引数プロセッサ」を指定するために使います。これを用いると、引数の内容をマクロ本体に渡す前に加工できます。引数プロセッサの利

用はやや高度な話題であり、少なくとも比較的あまり使われない機能です。詳しくはセクション 2.10 で説明します。

訳注：「引数プロセッサ（argument processors）」は、マクロに渡される前の引数を、その前後のスペースを削除するなど自動的に整理する仕組みです。*Argument processors* are a mechanism that automatically organizes arguments passed to a macro—such as by removing leading and trailing spaces—before they are processed.

## 2.4 Creating document commands and environments ドキュメントコマンドと環境の作成

`\NewDocumentCommand` {⟨*cmd*⟩} {⟨*arg spec*⟩} {⟨*code*⟩}
`\RenewDocumentCommand` {⟨*cmd*⟩} {⟨*arg spec*⟩} {⟨*code*⟩}
`\ProvideDocumentCommand` {⟨*cmd*⟩} {⟨*arg spec*⟩} {⟨*code*⟩}
`\DeclareDocumentCommand` {⟨*cmd*⟩} {⟨*arg spec*⟩} {⟨*code*⟩}

This family of commands are used to create a ⟨*cmd*⟩. The argument specification for the function is given by ⟨*arg spec*⟩, and the command uses the ⟨*code*⟩ with `#1`, `#2`, etc. replaced by the arguments found by the parser.

これらのコマンド群は、⟨*cmd*⟩ を定義するために用います。そのコマンドの引数仕様は ⟨*arg spec*⟩ で与えます。実行時には、⟨*code*⟩ 中の`#1`、`#2` などが、パーサーによって取得された引数に置き換えられます。

An example:

例えば：

```
\NewDocumentCommand\chapter{s o m}
  {%
    \IfBooleanTF{#1}%
      {\typesetstarchapter{#3}}%
      {\typesetnormalchapter{#2}{#3}}%
  }
```

would be a way to define a `\chapter` command which would essentially behave like the current LaTeX $2_\varepsilon$ command (except that it would accept an optional argument even when a `*` was parsed). The `\typesetnormalchapter` could test its first argument for being `\NoValue` to see if an optional argument was present. (See Section 2.8 for details of `\IfBooleanTF` and testing for `\NoValue`.)

これは、基本的には現在の LaTeX $2_\varepsilon$ の`\chapter` コマンドと同様に動作する `\chapter` を定義する一例です（ただし、`*`が取得された場合でもオプション引数を受け付ける点は異なります）。`\typesetnormalchapter` は、第 1 引数が`\NoValue` かどうかを調べることで、オプション引数が与えられていたかどうかを判定で

きます。(\IfBooleanTF および\NoValue かどうかの判定については、セクション 2.8 を参照してください。)

The difference between the \New... \Renew..., \Provide... and \Declare... versions is the behavior if ⟨*cmd*⟩ is already defined.

\New...、\Renew...、\Provide...、および \Declare... の違いは、⟨*cmd*⟩ がすでに定義されている場合の挙動にあります。

- \NewDocumentCommand will issue an error if ⟨*cmd*⟩ has already been defined.

  \NewDocumentCommand は、⟨*cmd*⟩ がすでに定義されている場合にエラーを出します。

- \RenewDocumentCommand will issue an error if ⟨*cmd*⟩ has not previously been defined.

  \RenewDocumentCommand は、⟨*cmd*⟩ があらかじめ定義されていない場合にエラーを出します。

- \ProvideDocumentCommand creates a new definition for ⟨*cmd*⟩ only if one has not already been given.

  \ProvideDocumentCommand は、⟨*cmd*⟩ がまだ定義されていない場合にのみ、新たに定義を行います。

- \DeclareDocumentCommand will always create the new definition, irrespective of any existing ⟨*cmd*⟩ with the same name. This should be used sparingly.

  \DeclareDocumentCommand は、同名の ⟨*cmd*⟩ がすでに存在していても、常に新しい定義を作成します。このコマンドの使用は必要最小限にとどめるべきです。

If the ⟨*cmd*⟩ can't be provided as a single token but needs "constructing", you can use \ExpandArgs as explained in Section 4 which also gives an example in which this is needed.

⟨*cmd*⟩ を単一のトークンとして与えられず、「組み立てる」必要がある場合は、セクション 4 で説明されている \ExpandArgs を使うことができます。同セクションには、この機能が必要になる例も示されています。

`\NewDocumentEnvironment` {⟨*env*⟩} {⟨*arg spec*⟩} {⟨*beg-code*⟩} {⟨*end-code*⟩}
`\RenewDocumentEnvironment` {⟨*env*⟩} {⟨*arg spec*⟩} {⟨*beg-code*⟩} {⟨*end-code*⟩}
`\ProvideDocumentEnvironment` {⟨*env*⟩} {⟨*arg spec*⟩} {⟨*beg-code*⟩} {⟨*end-code*⟩}
`\DeclareDocumentEnvironment` {⟨*env*⟩} {⟨*arg spec*⟩} {⟨*beg-code*⟩} {⟨*end-code*⟩}

These commands work in the same way as `\NewDocumentCommand`, etc., but create environments (`\begin{`⟨*env*⟩`}` ... `\end{`⟨*env*⟩`}`). Both the ⟨*beg-code*⟩ and ⟨*end-code*⟩ may access the arguments as defined by ⟨*arg spec*⟩. The arguments will be given following `\begin{`⟨*env*⟩`}`. Any spaces at the start and end of the `{`⟨*env*⟩`}` are removed before the definition takes place, thus

これらのコマンドは`\NewDocumentCommand` などと同様に動作しますが、環境（`\begin{`⟨*env*⟩`}` ... `\end{`⟨*env*⟩`}`）を定義します。⟨*beg-code*⟩ と ⟨*end-code*⟩ のいずれも、⟨*arg spec*⟩ で定義された引数にアクセスできます。引数は `\begin{`⟨*env*⟩`}` に続けて指定します。`{`⟨*env*⟩`}`の先頭および末尾にある空白は、定義が行われる前に取り除かれるため、

```
\NewDocumentEnvironment{foo}
```

and

と

```
\NewDocumentEnvironment{ foo }
```

both create the same “`foo`” environment.

は、どちらも同じ“`foo`”環境を定義します。

## 2.5 Optional arguments オプション引数

In contrast to commands created using LaTeX $2_\varepsilon$'s `\newcommand`, optional arguments created using `\NewDocumentCommand` may safely be nested. Thus for example, following

LaTeX $2_\varepsilon$ の`\newcommand` で作成したコマンドとは異なり、`\NewDocumentCommand` で作成したコマンドのオプション引数は、安全に入れ子にできます。例えば、

```
\NewDocumentCommand\foo{om}{I grabbed ‘#1’ and ‘#2’}
\NewDocumentCommand\baz{o}{#1-#1}
```

using the command as
と定義しておき、次のように使うと

```
\foo[\baz[stuff]]{more stuff}
```

will print
これは

> I grabbed ‘stuff-stuff’ and ‘more stuff’

と出力されます。

This is particularly useful when placing a command with an optional argument *inside* the optional argument of a second command.
これは特に、オプション引数を持つコマンドを、別のコマンドのオプション引数の内側に入れて使う場合に便利です。

When an optional argument is followed by a mandatory argument with the same delimiter, the parser issues a warning because the optional argument could not be omitted by the user, thus becoming in effect mandatory. This can apply to `o`, `d`, `O`, `D`, `s`, `t`, `e`, and `E` type arguments followed by `r` or `R`-type required arguments.
オプション引数の直後に、同じ区切り文字を持つ必須引数が続く場合、そのオプション引数はユーザーが省略できず、実質的に必須引数になってしまいます。そのため、パーサーは警告を出します。これは、`r` または `R` 型の必須引数が後続する、`o`、`d`、`O`、`D`、`s`、`t`、`e`、`E` 型の引数に当てはまります。

The default for `O`, `D` and `E` arguments can be the result of grabbing another argument. Thus for example
`O`、`D`、および `E` 型引数のデフォルト値には、別の引数を取得した結果を用いることができます。

```
\NewDocumentCommand\foo{O{#2} m}
```

would use the mandatory argument as the default for the leading optional one.

とした場合、先頭のオプション引数のデフォルト値として必須引数が使われます。

## 2.6 Spacing and optional arguments 空白とオプション引数

TEX will find the first argument after a function name irrespective of any intervening spaces. This is true for both mandatory and optional arguments. So `\foo[arg]` and `\foo␣[arg]` are equivalent. Spaces are also ignored when collecting arguments up to the last mandatory argument to be collected (as it must exist). So after

TEX は、関数名の後に空白が挟まっていても、最初の引数を見つけます。これは必須引数とオプション引数の両方に当てはまります。したがって、`\foo[arg]` と `\foo␣[arg]` は同じです。また、最後に取得する必須引数までは、その途中にある空白も無視されます。必須引数は必ず存在しなければならないからです。そのため、

```
\NewDocumentCommand\foo{m o m}{ ... }
```

the user input `\foo{arg1}[arg2]{arg3}` and `\foo{arg1}␣[arg2]␣{arg3}` will both be parsed in the same way.

と定義した場合、ユーザー入力 `\foo{arg1}[arg2]{arg3}` と `\foo{arg1}␣[arg2]␣{arg3}` は、どちらも同じように解析されます。

The behavior of optional arguments *after* any mandatory arguments is selectable. The standard settings will allow spaces here, and thus with

必須引数の後に現れるオプション引数の扱いは選べます。標準設定では、この位置に空白があっても許されるため、

```
\NewDocumentCommand\foobar{m o}{ ... }
```

both `\foobar{arg1}[arg2]` and `\foobar{arg1}␣[arg2]` will find an optional argument. This can be changed by giving the modified `!` in the argument specification:

とした場合、`\foobar{arg1}[arg2]` と `\foobar{arg1}␣[arg2]` のどちらでも、オプション引数が取得されます。この挙動は、引数指定に修飾記号`!`を付けることで変更できます：

```
\NewDocumentCommand\foobar{m !o}{ ... }
```

where `\foobar{arg1}␣[arg2]` will not find an optional argument.

この場合、`\foobar{arg1}␣[arg2]` ではオプション引数は取得されません。

There is one subtlety here due to the difference in handling by TEX of 'control symbols', where the command name is made up of a single character, such as '`\\`'. Spaces are not ignored by TEX here, and thus it is possible to require an optional argument directly follow such a command. The most common example is the use of `\\` in amsmath environments, which in the terms here would be defined as

ここには、TEX が「制御記号」(control symbols) を処理する方法の違いに由来する、ひとつの微妙な点があります。制御記号とは、たとえば `\\` のように、コマンド名が 1 文字だけでできている命令のことです。この場合、TEX は空白を無視しません。そのため、この種のコマンドの直後にオプション引数を続けるよう要求することが可能になります。もっともよくある例は、amsmath 環境内で使われる`\\`で、ここでの説明に従えば、これは次のように定義されることになります。

```
\NewDocumentCommand\\{!s !o}{ ... }
```

Also notable when using optional arguments in the last position is that TEX will necessarily look ahead for the argument opening token. This means that the value of `\inputlineno` will be 'out by one' if such a trailing optional argument is *not* present and the command ends a line; it will be one greater than the line number containing the last mandatory argument.

末尾にオプション引数を置く場合にもう 1 つ注意すべきなのは、TEX が必ず引数の開始トークンを探して先読みを行うことです。そのため、そのような末尾のオプション引数が存在せず、しかもコマンドが行末で終わる場合には、`\inputlineno` の値が 1 だけずれます。具体的には、最後の必須引数を含む行の番号よりも 1 大きい値となります。

## 2.7 'Embellishments' 修飾子

The `E`-type argument allows one default value per test token. This is achieved by giving a list of defaults for each entry in the list, for example:

`E` 型引数では、各判定用トークンに対して 1 つずつデフォルト値を指定できます。これは、各トークンに対応するデフォルト値を並べたリストを与えることで実現されます。例えば、

```
E{^_}{{UP}{DOWN}}
```

If the list of default values is *shorter* than the list of test tokens, the special `\NoValue` marker will be returned (as for the `e`-type argument). Thus for example

デフォルト値のリストが判定用トークンのリストよりも 短い場合には、(`e` 型引数と同様に) 特別なマーカー`\NoValue` が返されます。したがって、例えば

```
E{^_}{{UP}}
```

has default `UP` for the `^` test character, but will return the `\NoValue` marker as a default for `_`. This allows mixing of explicit defaults with testing for missing values.

この場合、判定用トークン`^`に対するデフォルト値は `UP` ですが、`_`に対してはデフォルトとして`\NoValue` が返されます。これにより、明示的なデフォルト値の指定と、値が与えられていない場合の判定とを併用できます。

## 2.8 Testing special values 特殊な値の判定

Optional arguments make use of dedicated variables to return information about the nature of the argument received.

オプション引数では、受け取った引数の状態に関する情報を返すために、専用の値が用いられます。

`\IfNoValueTF` {⟨*arg*⟩} {⟨*true code*⟩} {⟨*false code*⟩}
`\IfNoValueT` {⟨*arg*⟩} {⟨*true code*⟩}
`\IfNoValueF` {⟨*arg*⟩} {⟨*false code*⟩}

The `\IfNoValue(TF)` tests are used to check if ⟨*argument*⟩ (`#1`, `#2`, *etc.*) is the special `\NoValue` marker. For example

`\IfNoValue(TF)` は、⟨*argument*⟩ (`#1`、`#2` など) が特別な`\NoValue` マーカーであるかどうかを判定するために使います。例えば、

```
\NewDocumentCommand\foo{o m}
  {%
    \IfNoValueTF {#1}%
      {\DoSomethingJustWithMandatoryArgument{#2}}%
      {\DoSomethingWithBothArguments{#1}{#2}}%
  }
```

will use a different internal function if the optional argument is given than if it is not present.

では、オプション引数が与えられた場合と与えられなかった場合とで、異なる内部関数を使います。

Note that three tests are available, depending on which outcome branches are required: `\IfNoValueTF`, `\IfNoValueT` and `\IfNoValueF`.

必要な分岐に応じて、`\IfNoValueTF`、`\IfNoValueT`、`\IfNoValueF` の 3 種類が用意されていることに注意してください。

As the `\IfNoValue(TF)` tests are expandable, it is possible to test these values later, for example at the point of typesetting or in an expansion context.

`\IfNoValue(TF)` による判定は展開可能なので、例えば組版時や、展開が必要な文脈で、後からこれらの値を判定することもできます。

When two optional arguments follow each other (a syntax we typically discourage), it can make sense to allow users of the command to specify only the second argument by providing an empty first argument.If you wish to test if an argument is blank or not, but are not concerned with distinguishing an entirely absent argument from an empty one, use the `O` type specifier with the conditional `\IfBlankTF` (described below).

New description
2022/06/01

2 つのオプション引数が続く場合（一般的には推奨されない構文ですが）、最初の引数を空にすることで、2 番目の引数だけを指定できるようにすると便利なことがあります。引数が空かどうかを判定したいが、引数が完全に省略されている場合と空である場合とを区別する必要がないのであれば、`O` 型指定子と条件分岐コマンド `\IfBlankTF`（後述）を組み合わせて使ってください。

`\IfValueTF` {⟨*arg*⟩} {⟨*true code*⟩} {⟨*false code*⟩}
`\IfValueT` {⟨*arg*⟩} {⟨*true code*⟩}
`\IfValueF` {⟨*arg*⟩} {⟨*false code*⟩}

The reverse form of the `\IfNoValue(TF)` tests are also available as `\IfValue(TF)`. The context will determine which logical form makes the most sense for a given code scenario.

`\IfNoValue(TF)` に対する逆の形式として、`\IfValue(TF)` も利用できます。どちらの論理形式が適切かは、コードの文脈によって決まります。

`\IfBlankTF` {⟨*arg*⟩} {⟨*true code*⟩} {⟨*false code*⟩}
`\IfBlankT` {⟨*arg*⟩} {⟨*true code*⟩}
`\IfBlankF` {⟨*arg*⟩} {⟨*false code*⟩}

New feature
2022/06/01

The `\IfNoValueTF` command chooses the ⟨*true code*⟩ if the optional argument has not been used at all (and it returns the special `\NoValue` marker), but not if it has been given an empty value. In contrast `\IfBlankTF` returns true if its argument is either truly empty or only contains one or more normal blanks. For example

コマンド`\IfNoValueTF` は、オプション引数がまったく使われなかった場合（つまり特別な `\NoValue` マーカーが返される場合）には ⟨*true code*⟩ を選びますが、空の値が与えられた場合にはそうなりません。対照的に、`\IfBlankTF` は、引数が完全に空であるか、あるいは 1 つ以上の通常の空白文字のみを含む場合に「真」を返します。例えば

```
\NewDocumentCommand\foo{m!o}{\par #1:
  \IfNoValueTF{#2}
    {No optional}%
    {%
      \IfBlankTF{#2}
        {Blanks in or empty}%
        {Real content in}%
    }%
  \space argument!}
\foo{1}[bar] \foo{2}[  ] \foo{3}[] \foo{4}[\space] \foo{5} [x]
```

results in the following output:
は次のような出力を生成します：

1: Real content in argument!
2: Blanks in or empty argument!
3: Blanks in or empty argument!
4: Real content in argument!
5: No optional argument! [x]

Note that the `\space` in (4) is considered real content—because it is a command and not a “space” character—even though it results in producing a space. You can also observe in (5) the effect of the `!` specifier, preventing the last `\foo`

from interpreting `[x]` as its optional argument.

注意すべき点として、(4) の `\space` は、結果として空白を生成するにもかかわらず、実際の内容として扱われます。その理由は“space”文字ではなくコマンドだからです。また、(5) では`!` 指定子の効果も確認できます。これにより、最後の `\foo` は `[x]` を自分自身のオプション引数として解釈しなくなります。

`\BooleanFalse`
`\BooleanTrue`

The `true` and `false` flags set when searching for an optional character (using `s` or `t`⟨*char*⟩) have names which are accessible outside of code blocks.

`true` および `false` のフラグは、（`s` や `t`⟨*char*⟩ を使って）オプション文字を探す際に設定されるもので、コードブロックの外からアクセスできる名前を持っています。

`\IfBooleanTF {`⟨*arg*⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`
`\IfBooleanT {`⟨*arg*⟩`} {`⟨*true code*⟩`}`
`\IfBooleanF {`⟨*arg*⟩`} {`⟨*false code*⟩`}`

Used to test if ⟨*argument*⟩ (`#1`, `#2`, *etc.*) is `\BooleanTrue` or `\BooleanFalse`. For example

これらは、⟨*argument*⟩（`#1`、`#2` など）が `\BooleanTrue` か`\BooleanFalse` かを判定するために使います。例えば

```
\NewDocumentCommand\foo{sm}
  {%
    \IfBooleanTF {#1}%
      {\DoSomethingWithStar{#2}}%
      {\DoSomethingWithoutStar{#2}}%
  }
```

checks for a star as the first argument, then chooses the action to take based on this information.

では、第 1 引数が星印かどうかを調べ、その結果に応じて実行する処理を選択します。

## 2.9 Auto-converting to key–value format キー値形式への自動変換

Some document commands have a long history of accepting a 'free text' optional argument, for example `\caption` and the sectioning commands `\section`, etc. Introducing more sophisticated (keyval) options to these commands therefore needs a method to interpret the optional argument *either* as free text *or* as a series of keyvals. This needs to take place during argument grabbing as there is a need for careful treatment of braces to obtain the correct result.

一部のドキュメントコマンド、例えば`\caption` や`\section` などには、オプション引数として「自由テキスト」( キー値形式ではない通常のテキスト）を受け入れてきた長い歴史があります。したがって、これらのコマンドに、より洗練された（キー値 l) オプションを導入するには、オプション引数を自由なテキストとして、またはキー値の並びとしてのいずれかで解釈する方法が必要となります。正しい結果を得るには波括弧を慎重に扱う必要があるため、これは引数の取得時に行わなければなりません。

The = modifier is available to allow ltcmd to correctly implement this process. The modifier guarantees that the argument will be passed to further code as a series of keyvals. To do that, the = should be followed by an argument containing the default key name. This is used as the key in a key–value pair *if* the 'raw' argument does *not* have the correct form to be interpreted as a set of keyvals.

この処理を ltcmd で正しく実装するために、=修飾子が用意されています。この修飾子は、引数がキー値の列として後続のコードに渡されることを保証します。そのためには、=の後に、デフォルトのキー名を含む引数を指定する必要があります。処理していないままの引数がキー値として解釈できる正しい形式で ない場合には、このデフォルトのキー名がキー値として用いられます。

Taking `\caption` as an example, with the demonstration implementation

例として`\caption` をとると、デモ用の実装

```
\DeclareDocumentCommand\caption{s ={short-text} +O{#3} +m}
  {%
    \showtokens{Grabbed arguments:^^J(#2)^^Jand^^J(#3)}%
  }
```

the default key name is `short-text`. When the command `\caption` is then used, if the optional argument is free text such as

では、デフォルトのキー名は short-text となります。このとき、\caption コマンドのオプション引数として、次のような自由形式のテキストを与えると、

```
\caption[Some short text]{A much longer and more detailed text for
  demonstration purposes}
```

then the output will be
を与えると、出力は

```
Grabbed arguments:
(short-text={Some short text})
and
(A much longer and more detailed text for demonstration purposes)
```

On the other hand, if the caption is given with a keyval-form argument
となります。一方、キャプションをキー値形式の引数で指定した場合

```
\caption[label = cap:demo]%
  {A much longer and more detailed text for demonstration purposes}
```

then this will be respected
には、指定はそのまま用いられます。

```
Grabbed arguments:
(label = cap:demo)
and
(A much longer and more detailed text for demonstration purposes)
```

Interpretation as keyval form is determined by the presence of = characters within the argument. Those in inline math mode (enclosed within $...$ or \(...\)) are ignored. An argument can be forced to be read as keyvals by including an empty entry at the start
キー値形式としての解釈は、引数内に =文字が含まれているかどうかによって決定されます。ただし、インライン数式モード（$...$ または\(...\) で囲まれた部分）にある=文字は無視されます。引数の先頭に空の要素を置くことで、強制的にキー値として解釈させることができます。

```
\caption[=,This is now a keyval]%
% ...
\caption[This is not $=$ keyval]%
```

This empty entry is *not* passed to the underlying code, so will not lead to issues with keyval parsers that do not allow an empty key name. Any text-mode = signs will need to be braced to avoid being misinterpreted: this is likely most conveniently handled by bracing the entire argument

この空の項目は内部のコードには渡されないため、空のキー名を許容しないキー値パーサーとの間で問題が生じることはありません。テキストモードで記述された=記号は、誤って解釈されないよう波括弧で囲む必要があります。これに対処する最も便利な方法は、引数全体を波括弧で囲むことでしょう。そうすれば

```
\caption[{Not = to a keyval!}]%
```

which will be passed correctly as

は、正しく

```
Grabbed arguments:
(short-text = {Not = to a keyval!})
```

として渡されます。

If the argument is completely blank, the conversion results in an empty keyval list, not `short-text␣=␣`. This reflects the fact that with a move toward keyval processing, an empty argument is best modelled as an empty keyval list. If the user does want an empty classical argument, using `[{}]` will work with both the new processor code and older formats.

引数が完全に空である場合、変換結果は `short-text␣=␣`ではなく、空のキー値リストとなります。これは、キー値処理へ移行するにあたって、空の引数は空のキー値リストとして扱うのが最も適切であることを反映しています。ユーザーが空の従来型の引数を必要とする場合は、`[{}]` を使うことで、新しいプロセッサコードと古い形式の両方で動作します。

## 2.10 Argument processors 引数プロセッサ

Argument processor are applied to an argument *after* it has been grabbed by the underlying system but before it is passed to ⟨*code*⟩. An argument processor

can therefore be used to regularize input at an early stage, allowing the internal functions to be completely independent of input form. Processors are applied to user input and to default values for optional arguments, but *not* to the special `\NoValue` marker.

引数プロセッサは、基盤となるシステムが引数を取得した後で、それが ⟨*code*⟩ に渡される前に、その引数に対して適用します。したがって、引数プロセッサを用いて入力の形式を早い段階で正規化（regularize）すれば、内部関数を入力形式から完全に独立させることが可能になります。プロセッサはユーザー入力や省略可能なオプション引数のデフォルト値に対しては適用されますが、特殊なマーカーである`\NoValue` に対しては適用されません。

訳註：ここで「正規化（regularize）」とは、表記や形式が異なる入力を、内部処理のために統一された形式へ変換することを意味します。
Here, regularize means to convert inputs with varying notation or format into a unified form for internal processing.

Each argument processor is specified by the syntax `>{`⟨*processor*⟩`}` in the argument specification. Processors are applied from right to left, so that

各引数プロセッサは、引数指定において`>{`⟨*processor*⟩`}`という構文で指定されます。プロセッサは右から左の順に適用されるため、

```
>{\ProcessorB} >{\ProcessorA} m
```

would apply `\ProcessorA` followed by `\ProcessorB` to the tokens grabbed by the `m` argument.

と記述すると、`m` 引数によって取得されたトークンに対し、まず`\ProcessorA` が適用され、続いて `\ProcessorB` が適用されることになります。

`\SplitArgument {`⟨*number*⟩`} {`⟨*token(s)*⟩`}`

This processor splits the argument given at each occurrence of the ⟨*tokens*⟩ up to a maximum of ⟨*number*⟩ tokens (thus dividing the input into ⟨*number*⟩ + 1 parts). An error is given if too many ⟨*tokens*⟩ are present in the input. The processed input is placed inside ⟨*number*⟩ + 1 sets of braces for further use. If there are fewer than `{`⟨*number*⟩`}` of `{`⟨*tokens*⟩`}` in the argument then `\NoValue` markers are added at the end of the processed argument.

このプロセッサは、引数の中に ⟨*tokens*⟩ が現れるたびに、その位置で引数を分割します。ただし分割に使う ⟨*tokens*⟩ の回数は最大で ⟨*number*⟩ 個までです（したがって、入力は ⟨*number*⟩ + 1 個の部分に分けられます）。入力中に ⟨*tokens*⟩ が多すぎる場合はエラーとなります。処理された入力は、その後の利用に備えて ⟨*number*⟩ + 1 組の波括弧で囲まれ形になります。引数内の ⟨*tokens*⟩ の数が ⟨*number*⟩ 未満なら、処理後の引数の末尾に`\NoValue` マーカーが追加されます。

```
\NewDocumentCommand\foo{>{\SplitArgument{2}{;}} m}
  {\InternalFunctionOfThreeArguments#1}
```

If only a single character ⟨*token*⟩ is used for the split, any category code 13 (active) character matching the ⟨*token*⟩ will be replaced before the split takes place. Spaces are trimmed at each end of each item parsed.

分割に単一の ⟨*token*⟩ だけが使われた場合、その ⟨*token*⟩ に一致する カテゴリーコード 13（active）の文字 は、分割処理が行われる前に置き換えられます。また、解析された各項目の先頭と末尾にある空白は取り除かれます。

The `E` argument type is somewhat special, because with a single `E` in the command declaration you may end up with several arguments in a command (one formal argument per embellishment token). Therefore, when an argument processor is applied to an `e`/`E`-type argument, all the arguments pass through that processor before being fed to the ⟨*code*⟩. For example, this command

`E` 型引数はやや特殊です。なぜなら、コマンド宣言で `E` を 1 つだけ書いても、実際にはコマンドが複数の引数を持つことになる（修飾トークンごとに 1 つの形式的引数が生じる）からです。そのため、`e` 型や `E` 型の引数に引数プロセッサを適用する場合、それらすべての引数が ⟨*code*⟩ に渡される前に、そのプロセッサを通ります。例えば、次のコマンドでは、

```
\NewDocumentCommand\foo{ >{\TrimSpaces} e{_^} }
  { [#1](#2) }
```

applies `\TrimSpaces` to both arguments.

両方の引数に対して`\TrimSpaces` が適用されます。

`\SplitList {`⟨*token(s)*⟩`}`

This processor splits the argument given at each occurrence of the ⟨*token(s)*⟩ where the number of items is not fixed. Each item is then wrapped in braces within `#1`. The result is that the processed argument can be further processed using a mapping function (see below).

このプロセッサは、項数があらかじめ決まっていないリストを扱うためのもので、与えられた引数を ⟨*token(s)*⟩ が現れるたびに分割します。その結果、`#1` の中では各項目がそれぞれ波括弧で囲まれた形になります（後述）。

```
\NewDocumentCommand\foo{>{\SplitList{;}} m}
  {\MappingFunction#1}
```

If only a single character ⟨*token*⟩ is used for the split, it will take account of the possibility that the ⟨*token*⟩ has been made active (category code 13) and will split at such tokens. Spaces are trimmed at each end of each item parsed. Exactly one set of braces will be stripped if an entire item is surrounded by them, i.e. the following inputs and outputs result (each separate item as a brace group).

分割する際に単一の文字 ⟨*token*⟩ だけを使った場合、その ⟨*token*⟩ がアクティブ（カテゴリコード 13）になっている可能性が考慮され、そのようなトークンでも分割が行われます。分割された各要素の先頭と末尾にある空白は取り除かれます。要素全体が波括弧で囲まれている場合は、その波括弧がちょうど 1 組だけ取り除かれます。つまり、以下の入力に対して示されるような出力が得られます（各要素は波括弧で囲まれた集まりとして扱われます)。

```
a      ==> {a}
{a}    ==> {a}
{a}b   ==> {{a}b}
a,b    ==> {a}{b}
{a},b  ==> {a}{b}
a,{b}  ==> {a}{b}
a,{b}c ==> {a}{{b}c}
```

`\ProcessList {`⟨*list*⟩`} {`⟨*tokens*⟩`}`

To support `\SplitList`, the function `\ProcessList` is available to apply ⟨*tokens*⟩ to every entry in a ⟨*list*⟩. The ⟨*tokens*⟩ can be arbitrary contents that should expect one argument after it: the list entry. For example

`\SplitList` を補助するものとして、⟨*list*⟩ の各要素に ⟨*tokens*⟩ を適用する `\ProcessList` 関数が用意されています。ここで ⟨*tokens*⟩ には任意のコンテンツを指定できますが、その直後に引数（リストの要素）を 1 つとる形式でなければなりません。例えば

```
\NewDocumentCommand\foo{>{\SplitList{;}} m}
  {\ProcessList{#1}{\SomeDocumentCommand}}
```

or

あるいは、次のようにも書けます。

```
\NewDocumentCommand\foo{>{\SplitList{;}} m}
  {\ProcessList{#1}{Abc \SomeDocumentCommand}}
```

`\ReverseBoolean`

This processor reverses the logic of `\BooleanTrue` and `\BooleanFalse`, so that the example from earlier would become

このプロセッサは`\BooleanTrue`と`\BooleanFalse`の論理を反転させるため、先ほどの例は次のようになります。

```
\NewDocumentCommand\foo{>{\ReverseBoolean} s m}
  {%
    \IfBooleanTF#1%
      {\DoSomethingWithoutStar{#2}}%
      {\DoSomethingWithStar{#2}}%
  }
```

`\TrimSpaces`

Removes any leading and trailing spaces (tokens with character code 32 and category code 10) for the ends of the argument. Thus for example declaring a function

引数の先頭と末尾にある空白（文字コード 32、カテゴリコード 10 のトークン）を除去します。例えば、関数

```
\NewDocumentCommand\foo{>{\TrimSpaces} m}
  {\showtokens{#1}}
```

and using it in a document as

を定義して、これを文書内で

```
\foo{␣hello␣world␣}
```

will show '`hello␣world`' at the terminal, with the space at each end removed. `\TrimSpaces` will remove multiple spaces from the ends of the input in cases where these have been included such that the standard TEX conversion of multiple spaces to a single space does not apply.

として使った場合、端末には '`hello␣world`' と表示され、両端の空白は取り除かれます。通常の TEX では複数の空白は 1 つの空白に変換されますが、そのような変換が適用されない形で入力の両端に複数の空白が含まれている場合でも、`\TrimSpaces` はそれらを除去します。

## 2.11 Body of an environment 環境の本体

While environments `\begin{`⟨*environment*⟩`}` ... `\end{`⟨*environment*⟩`}` are typically used in cases where the code implementing the ⟨*environment*⟩ does not need to access the contents of the environment (its 'body'), it is sometimes useful to have the body as a standard argument.

環境 `\begin{`⟨*environment*⟩`}` ...,`\end{`⟨*environment*⟩`}`は、通常、その環境を実装するコードが環境の内容（すなわち本体）にアクセスする必要がない場合に用いられますが、本体を通常の引数として受け取れると便利なこともあります。

This is achieved by ending the argument specification with `b`, which is a dedicated argument type for this situation. For instance

これは、引数指定の末尾に、この用途専用の引数型である `b` を置くことで実現されます。例えば、

```
\NewDocumentEnvironment{twice}{O{\ttfamily} +b}
  {#2#1#2} {}
\begin{twice}[\itshape]
  Hello world!
\end{twice}
```

typesets 'Hello world!*Hello world!*'.

とします。この例は、'Hello world!*Hello world!*' と組版されます。

The prefix `+` is used to allow multiple paragraphs in the environment's body. Argument processors can also be applied to `b` arguments. By default, spaces are trimmed at both ends of the body: in the example there would otherwise be spaces coming from the ends the lines after `[\itshape]` and `world!`. Putting the prefix `!` before `b` suppresses space-trimming.

接頭辞 `+`を使うと、環境本体に複数の段落を含めることができます。また、`b` 型引数にも引数プロセッサを適用できます。デフォルトでは、本体の前後にある空白は削除（trim）されます。上の例でそうしないと、`[\itshape]` の後の改行や `world!`の後の改行に由来する空白が入り込むことになります。`b` の前に接頭辞 `!` を付けると、この空白削除は抑制されます。

When `b` is used in the argument specification, the last argument of the environment declaration (e.g., `\NewDocumentEnvironment`), which consists of an ⟨*end code*⟩ to insert at `\end{`⟨*environment*⟩`}`, is redundant since one can simply put that code at the end of the ⟨*start code*⟩. Nevertheless this (empty) ⟨*end code*⟩

must be provided.

引数指定に b を用いる場合、環境宣言（例えば \NewDocumentEnvironment）の最後の引数、すなわち \end{⟨*environment*⟩}の位置に挿入される ⟨*end code*⟩ は、実質的には冗長です。というのも、そのコードは単に ⟨*start code*⟩ の末尾に書いてしまえばよいからです。それでも、この（空の）⟨*end code*⟩ 自体は与えなければなりません。

Environments that use this feature can be nested.

この機能を使う環境は、入れ子にすることができます。

## 2.12 Fully-expandable document commands 完全展開可能なドキュメントコマンド

Document commands created using \NewDocumentCommand, etc., are normally created so that they do not expand unexpectedly. This is done using engine features, so is more powerful than LaTeX $2_\varepsilon$'s \protect mechanism. There are *very rare* occasion when it may be useful to create functions using a expansion-only grabber. This imposes a number of restrictions on the nature of the arguments accepted by a function, and the code it implements. This facility should only be used when *necessary*.

\NewDocumentCommand などを用いて作成される通常のドキュメントコマンドは、予期せず展開されないように保護されています。これはエンジンの機能を利用して実現されているため、LaTeX $2_\varepsilon$ の\protect 機構よりも強力な仕組みとなっています。ただし、展開だけで引数を取得する仕組み（expansion-only grabber）を用いて関数を作ることが有用な場合も、ごくまれにあります。この手法を用いると、関数が受け入れる引数の性質や実装するコードに対して、いくつかの制約が課されることになります。したがって、この機能はあくまで必要な場合にのみ使うべきです。

\NewExpandableDocumentCommand {⟨*cmd*⟩} {⟨*arg spec*⟩} {⟨*code*⟩}
\RenewExpandableDocumentCommand {⟨*cmd*⟩} {⟨*arg spec*⟩} {⟨*code*⟩}
\ProvideExpandableDocumentCommand {⟨*cmd*⟩} {⟨*arg spec*⟩} {⟨*code*⟩}
\DeclareExpandableDocumentCommand {⟨*cmd*⟩} {⟨*arg spec*⟩} {⟨*code*⟩}

This family of commands is used to create a document-level ⟨*cmd*⟩, which will grab its arguments in a fully-expandable manner. The argument specification for the function is given by ⟨*arg spec*⟩, and the ⟨*cmd*⟩ will execute ⟨*code*⟩. In

general, ⟨*code*⟩ will also be fully expandable, although it is possible that this will not be the case (for example, a function for use in a table might expand so that `\omit` is the first non-expandable non-space token).

このコマンド群は、引数を完全展開可能な形で取得する文書レベルの ⟨*cmd*⟩ を定義するために使います。この関数の引数仕様は ⟨*arg spec*⟩ で指定され、⟨*cmd*⟩ は ⟨*code*⟩ を実行します。一般に ⟨*code*⟩ も完全に展開可能となりますが、必ずしもそうとは限りません（例えば表組の中で使う関数では、`\omit` が最初の展開不可能な非空白トークンになるように展開される可能性があります）。

Parsing arguments by pure expansion imposes a number of restrictions on both the type of arguments that can be read and the error checking available:

純粋な展開によって引数を解析する場合、取得可能な引数の種類と利用可能なエラーチェックの両面において、いくつかの制約が生じます：

- The last argument (if any are present) must be one of the mandatory types `m`, `r` or `R`.

  最後の引数（引数が存在する場合）は、必須引数型 `m`、`r`、または `R` のいずれかでなければなりません。

- The 'verbatim' argument type `v` is not available.

  'verbatim' 引数型 `v` は使用できません。

- Argument processors (using `>`) are not available.

  引数プロセッサ（`>`を使う）は使用できません。

- It is not possible to differentiate between, for example `\foo[` and `\foo{[}`: in both cases the `[` will be interpreted as the start of an optional argument. As a result, checking for optional arguments is less robust than in the standard version.

  例えば `\foo[`と`\foo{[}`を区別することはできません。どちらの場合も `[`はオプション引数の開始として解釈されます。その結果、オプション引数の判定は標準版ほど頑健ではありません。

## 2.13 Commands at the start of tabular cells 表組のセルの先頭で使うコマンド

Creating commands that are used at the start of tabular cells imposes some restrictions on the underlying implementation. The standard LaTeX tabular

訳注：ここで使った頑健という単語は robust（ロバスト）の訳です。コマンドの性質を表す専門用語です。ここの後 7 で説明される移動引数（moving argument）」の中で使用されても、安全に展開され、不正な TeX コードを生成しないコマンドのことです。「頑健（robust）なコマンド」の反対のコマンドに「脆弱（fragile）なコマンド」があります。The term 頑健 (robust) used here is a technical term describing a specific property of a command. A 'robust command' is one that expands safely, without generating invalid TeX code, even when used within a 'moving argument' (explained later in section 7). The opposite of a "robust command" is a "fragile command."

environments (`tabular`, etc.) use a mechanism which requires that any command wrapping `\multicolumn` or similar must be 'expandable'. This is *not* the case for commands created using `\NewDocumentCommand`, etc., which as detailed in Section 2.12 use an engine feature which prevents such 'expansion'. Therefore, to create such wrappers for use at the start of tabular cells, you must use `\NewExpandableDocumentCommand`, for example

tabular 環境のセルの先頭で使用するコマンドを作成する場合、その内部実装にはいくつかの制約が伴います。標準的な LaTeX の tabular 環境（`tabular` など）では、`\multicolumn` やそれに類するコマンドを包むコマンド（ラッパー）は「展開可能」でなければならない仕組みになっています。しかし、`\NewDocumentCommand` などを用いて作成されたコマンドはこれに該当しません。これは、セクション 2.12 で述べたように、そうした「展開」を防ぐエンジン機能を用いているためです。したがって、tabular 環境のセル先頭で使うその種のラッパーコマンドを作るには、例えば `\NewExpandableDocumentCommand` を使う必要があります。

```
\NewExpandableDocumentCommand\MyMultiCol{m}{\multicolumn{3}{c}{#1}}
\begin{tabular}{lcr}
a & b & c \\
\MyMultiCol{stuff} \\
\end{tabular}
```

## 2.14 Using the verbatim argument types 逐語的引数の使い方

As described above, the v-type argument may be viewed as similar to `\verb`. Before looking at exactly what that means, it is important to highlight some key differences. Most notably, *grabbing* a verbatim-like argument is separate from *typesetting* it: the latter is covered in the next section.

上述の通り、v 型引数は`\verb` と類似したものと見なすことができます。その正確な意味を検討する前に、いくつかの重要な違いを指摘しておく必要があります。とりわけ注目すべき点は、逐語的な引数を取り込む（grabbing）ことと、それを組版することは別個の処理であるという点です。後者については次節で扱います。

When grabbing a v-type argument, LaTeX first uses the kernel command `\dospecials` to turn off the "special" nature of characters. It then makes both spaces and tabs "active", so that they can be given a custom definition. Any other characters are grabbed as-is: this means that if any characters have

been made “special” and are not listed in `\dospecials`, an error will arise (see below).

v 型引数を取り込む際、LaTeX はまずカーネルコマンド `\dospecials` を用いて、文字の「特殊な」性質を無効にします。続いて、空白とタブを「アクティブ」な状態にし、それらに独自の定義を割り当てられるようにします。その他の文字はそのままの状態で読み込まれます。つまり、何らかの文字が「特殊な」ものとして設定されているにもかかわらず `\dospecials` に含まれていない場合、エラーが発生することになります（後述）。

The characters that are grabbed as the argument are all those between two identical: in contrast to `\verb`, the characters `\`, `{`, `}` and `%` *cannot* be used as the delimiter character. If any of the grabbed tokens have “special” meaning, an error will be issued.

引数として取り込まれる文字は、同一の 2 つの文字に挟まれたすべての文字です。ただし、`\verb` とは異なり、`\`、`{`、`}`、`%`といった文字を区切り文字として使用することはできません。取り込まれたトークンの中に「特殊な」意味を持つものが含まれていると、エラーが発生します。

For the `+v`-type argument, which allows line breaks within the argument, newline characters are converted into `\obeyedline` commands. The standard definition of `\obeyedline` is simple `\par`, thus allowing the grabbed tokens to be used directly in typesetting. A local redefinition of `\obeyedline` can be used to achieve other outputs. For example, to retain blank lines whilst typesetting, one could use

引数内での改行を許容する `+v` 型の引数では、改行文字は`\obeyedline` コマンドに変換されます。`\obeyedline` の標準的な定義は単なる`\par` であり、これにより、取得されたトークンをそのまま組版に使用できるようになっています。`\obeyedline` を局所的に再定義することで、異なる出力結果を得ることも可能です。例えば、組版時に空行を保持したい場合には、

```
\renewcommand*\obeyedline{\mbox{}\par}
```

を使用することができます。

More information about using these arguments in typesetting is in the following subsection.

これらの引数を組版で使う方法の詳細は、次のサブセクションで説明します。

Some additional details that may be useful for those with more TeX knowledge: do not worry if this does not make sense to you! Spaces and tabs are

stored as active characters. In 8-bit engines, non-ASCII characters are “active”, whilst other than the letters a–zA–Z, ASCII characters are “other”. In Unicode engines, non-ASCII codepoints will be either letters or “other”, based on the standard LaTeX settings derived from Unicode data. For token-based comparisons, it is likely that the active spaces and tabs should be replaced: this can be done conveniently by expansion.

TeX に関する知識が豊富な方にとって役立つかもしれない補足情報をいくつか挙げます（これらが理解できなくても心配は無用です）。空白やタブはアクティブ文字（active characters）として扱われます。8 ビットエンジンでは、非 ASCII 文字は「アクティブ」となり、ASCII 文字は英字 a–zA–Z を除いて「その他（other）」として扱われます。Unicode エンジンでは、非 ASCII のコードポイントは、Unicode データに基づく標準的な LaTeX 設定に従い、文字または「その他」のいずれかとして扱われます。トークン単位で比較を行う場合には、アクティブな空白やタブを別のものに置き換える必要が生じることがありますが、これは展開によって手軽に行えます。

—-

> **This text is not in the original; it is a note for the translator’s own use.** — I think the explanation is clear enough for those knowledgeable about TeX, but it might be a bit difficult to grasp for beginners or non-native English speakers. I asked a generative AI to interpret it :-) by Y. Fujimura
>
> —
>
> A note for those familiar with TeX internals (no need to worry if this part is unclear):
>
> - Spaces and tabs are preserved as active characters.
> - In 8-bit engines, non-ASCII characters become active, while ASCII characters are treated as “other,” except for the letters a – z and A – Z.
> - In Unicode engines, non-ASCII code points are treated as either “letter” or “other,” in accordance with standard LaTeX settings based on Unicode data.
> - When performing comparisons token-by-token, it may be necessary to replace active spaces or tabs with a different form, but this can be easily achieved through expansion.

TEX の内部に詳しい方向けの補足です（ここが分からなくても心配しなくてよいです）。

- スペースとタブはアクティブ文字として保持されます。
- 8 ビットエンジンでは、非 ASCII 文字はアクティブになり、ASCII 文字は英字 a–zA–Z を除いて「その他」として扱われます。
- Unicode エンジンでは、非 ASCII コードポイントは、Unicode データに基づく標準的な LATEX の設定に従って、「文字」または「その他」のいずれかとして扱われます。
- トークン単位で比較を行う場合には、アクティブなスペースやタブを別の形に置き換える必要が生じることがありますが、これは展開によって容易に行えます。

—-

## 2.15 Typesetting verbatim-like material verbatim 的な内容の組版

In contrast to `\verb`, the `(+)v`-type argument is only about *grabbing* the argument, not *typesetting* it. As such, features that users often associate with "verbatim" are not automatically activated, e.g., selecting a monospaced font. Material grabbed by the `v`-type argument does not automatically suppress ligatures: with modern TEX engines, this largely can be done without the token manipulation which `\verb` uses. (In `\verb`, ligatures are suppressed by making characters active and inserting a zero-width kern before the character itself.)

`\verb` とは対照的に、`(+)v` 型の引数は引数を取りこむことのみを目的としており、組版は行いません。そのため、ユーザーが "verbatim"によく結び付けて考える機能、例えば等幅フォントの選択などは、自動的には有効になりません。`v` 型引数で取得された内容では、合字は自動的には抑制されません。最新の TEX エンジンでは、`\verb` が用いるようなトークン操作をしなくても、たいていの場合は合字の抑制が可能です。（`\verb` では、文字をアクティブにし、その文字自身の前にゼロ幅のカーンを挿入することで合字を抑制します。）

The `\verb` command also selects a monospaced font: this is not intrinsic to verbatim material, so will need to be set up using for example `\ttfamily`. Similarly, the `verbatim` environment sets up the meaning of `\par` suitable for breaking lines.

\verb コマンドは等幅フォントも選択しますが、これは verbatim 的な内容に本質的に備わっている性質ではありません。そのため、必要なら\ttfamily などを用いて設定しなければなりません。例えば \ttfamily を用いて設定する必要があります。同様に、verbatim 環境は、改行に適した \par の動作を設定します。

## 2.16 Verbatim environments verbatim 環境

In some cases, when grabbing the body of an environment you will want the contents to be treated verbatim. This is available using the argument specification c. Like the b specification, this has to be the last one. Thus for example

環境の本体を取り込む際、その内容をそのまま逐語的に扱いたい場合があります。これには引数指定 c を使用します。b 指定と同様、この指定は最後に来る必要があります。したがって、例えば次の定義

```
\NewDocumentEnvironment{MyVerbatim}{!O{\ttfamily} c}
  {\begin{center} #1 #2\end{center}} {}
\begin{MyVerbatim}[\ttfamily\itshape]
  % Some code is shown here
  $y = mx + c$
\end{MyVerbatim}
```

will typeset verbatim the content, thus:

では、内容が逐語的に組版され、結果は次のようになります。

```
␣␣%␣Some␣code␣is␣shown␣here
␣␣$y␣=␣mx␣+␣c$
```

Since grabbing the entire contents verbatim will result in there being no \par tokens, newlines are always permitted: there is no need for a + modifier here. As for the v specification, newlines are stored as \obeyedline. In a similar fashion to the b specification, by default *newlines* are trimmed at both ends of the body. Putting the prefix ! before c suppresses this trimming.

内容をそのまま逐語的に取得する場合、\par トークンは生成されないため、改行は常に許容されます。したがって、ここでは+という修飾子は不要です。v 指定の場合と同様に、改行は\obeyedline として格納されます。b 指定と同様に、デフォルトでは本体の前後にある行が取り除かれますが、c の前に !という接頭辞を付けると、この除去を抑制できます。

Collection of the body takes place on a line-by-line basis: content is collected up to the end-of-line in the source, then examined before storage. This means that the line ending the environment (containing in the example above `\end{MyVerbatim}`) cannot have any text *after* the end of the environment. Text *before* the end of environment is treated normally, but note that there is no trailing `\obeyedline` added if there is text here. Other than optional arguments, no text is allowed on the opening line of the environment.

本体の取り込みは行単位で行われます。つまり、ソース内の行末までが取り込まれ、保存される前に内容が検査されます。このため、環境を終了させる行（上記の例では`\end{MyVerbatim}`を含む行）において、環境の終了を示す記述の後にテキストを置くことはできません。環境終了の記述の前にあるテキストは通常どおり処理されますが、その場合には末尾に`\obeyedline`は追加されない点に注意してください。オプション引数を除いて、環境開始行にはテキストを書いてはいけません。

Special handling is applied to a `o`, `O`, `d` or `D` specification argument immediately before an `c` specification. This means that when the optional argument is absent, the first character of the next line will be read with the correctly applied verbatim category code. Issues may arise if *multiple* optional arguments are used before a `c` specification: this will only work reliably where the optional tokens are "other" characters.

`c`指定の直前にある`o`、`O`、`d`、または`D`指定の引数に対しては、特別な処理が行われます。これは、オプション引数が省略された場合でも、次の行の最初の文字が適切なverbatim用カテゴリコードで読み込まれることを意味します。ただし、`c`指定の前にオプション引数が複数あると問題が生じる可能性があります。この仕組みが確実に機能するのは、それらのオプション引数を示すトークンが「その他の」文字である場合に限られます。

訳注：verbatim用カテゴリコード（verbatim category code）とは、文字を通常の構文解釈から外し、逐語的に扱えるようにするためのカテゴリコード設定を指します。A *verbatim category code* refers to a category code setting that exempts characters from standard syntactic interpretation, allowing them to be treated verbatim.

For technical reasons, we recommend that spaces are *not* ignored when searching for an optional argument before an `c` specification: this can be achieved by adding the `!` modifier as shown in the example. However, this is left as a choice for the user.

技術的な理由から、`c`指定子の前にあるオプション引数を検索する際、空白を無視しないようにすることをお勧めします。これは、例に示すように`!`修飾子を追加することで実現できます。ただし、これを行うかどうかはユーザーの判断に委ねられています。

## 2.17 Performance パフォーマンス

For document commands where the argument specification is entirely comprised of `m` or `+m` entries (or is entirely empty), the internal structure created by `\NewDocumentCommand` is essentially as efficient as provided by `\newcommand(*)`. As such, document commands may replace constructs arising from `\newcommand`, etc., without a need to be concerned about performance. It should be noted that `\newcommand(*)` produces expandable results, so the direct replacement is `\NewExpandableDocumentCommand`; in most cases, however, it is better to use `\NewDocumentCommand` to give more robust structures.

引数仕様が `m` または `+m` だけから成る（あるいは引数が全くない）ドキュメントコマンドでは、`\NewDocumentCommand` が作る内部構造は、実質的に`\newcommand(*)` と同程度に効率的です。したがって、`\newcommand` などで作られていた定義は、性能面を気にせずドキュメントコマンドで置き換えることができます。ただし、`\newcommand(*)` で定義されるコマンドは展開可能なので、厳密に対応する置き換え先は `\NewExpandableDocumentCommand` です。とはいえ、多くの場合には、より堅牢な構造にできる`\NewDocumentCommand` を使う方が望ましいでしょう。

## 2.18 Details about argument delimiters 引数区切り文字に関する詳細

In normal (non-expandable) commands, the delimited types look for the initial delimiter by peeking ahead (using expl3's `\peek_...` functions) looking for the delimiter token. The token has to have the same meaning and 'shape' of the token defined as delimiter. There are three possible cases of delimiters: character tokens, control sequence tokens, and active character tokens. For all practical purposes of this description, active character tokens will behave exactly as control sequence tokens.

通常の（展開可能でない）コマンドでは、区切り付き引数型は、先読み（expl3 の `\peek_...` 関数を使用）によって区切りトークンを探し、先頭の区切り文字を見つけます。そのトークンは、区切り文字として定義されたトークンと同じ意味と「形」（shape）を持っていなければなりません。区切り文字として使えるトークンには、文字トークン、制御シーケンストークン、アクティブ文字トークンの 3 種類があります。本説明の範囲においては、アクティブ文字トークンは制御綴りトークンと全く同様に振る舞います。

### 2.18.1 Character tokens 文字トークン

A character token is characterized by its character code, and its meaning is the category code (`\catcode`). When a command is defined, the meaning of the character token is fixed into the definition of the command and cannot change. A command will correctly see an argument delimiter if the open delimiter has the same character and category codes as at the time of the definition. For example in:

文字トークンは文字コードによって特徴づけられ、その意味はカテゴリコード（`\catcode`）によって決まります。コマンドが定義される際、その文字トークンの意味はコマンドの定義の一部として固定され、後から変更されることはありません。コマンドが引数の区切り文字を正しく認識できるのは、開始区切り文字が定義時と同じ文字コードおよびカテゴリコードを持っている場合だけです。例えば、次のような場合：

```
\NewDocumentCommand { \foobar } { D<>{default} } {(#1)}
\foobar <hello> \par
\char_set_catcode_letter:N <
\foobar <hello>
```

the output would be:

出力は次のようになります：

```
(hello)
(default)<hello>
```

as the open-delimiter `<` changed in meaning between the two calls to `\foobar`, so the second one doesn't see the `<` as a valid delimiter. Commands assume that if a valid open-delimiter was found, a matching close-delimiter will also be there. If it is not (either by being omitted or by changing in meaning), a low-level TEX error is raised and the command call is aborted.

これは、`\foobar` を 2 回呼び出す間に開始区切り文字である`<`の意味が変わってしまったため、2 回目の呼び出しではその`<`が有効な区切り文字として認識されなかったからです。コマンドは、有効な開始区切り文字が見つかった場合、それに対応する終了区切り文字も存在すると想定して動作します。終了区切り文字が存在しない場合（省略されたか、あるいは意味が変わってしまったことによる）、低レベルの TEX エラーが発生し、コマンドの呼び出しは中断されます。

### 2.18.2 Control sequence tokens 制御シーケンストークン

A control sequence (or control character) token is characterized by its name, and its meaning is its definition. A token cannot have two different meanings at the same time. When a control sequence is defined as delimiter in a command, it will be detected as delimiter whenever the control sequence name is found in the document regardless of its current definition. For example in:

制御シーケンストークン（または制御文字トークン）は、その名前によって識別され、その意味は定義によって決まります。一つのトークンが同時に二つの異なる意味を持つことはありません。ある制御シーケンスがコマンド内で区切り文字として定義された場合、文書内でその制御シーケンス名が現れると、その時点での定義にかかわらず、常に区切り文字として認識されます。例えば：

```
\cs_set:Npn \x { abc }
\NewDocumentCommand { \foobar } { D\x\y{default} } {(#1)}
\foobar \x hello\y \par
\cs_set:Npn \x { def }
\foobar \x hello\y
```

the output would be:

とした時の出力は

```
(hello)
(hello)
```

with both calls to the command seeing the delimiter `\x`.

となり、コマンドへの両方の呼び出しにおいて区切り文字`\x` が認識されます。

## 2.19 Creating new argument processors 新しい引数プロセッサの作成

`\ProcessedArgument`

Argument processors allow manipulation of a grabbed argument before it is passed to the underlying code. New processor implementations may be created as functions which take one trailing argument, and which leave their result in the `\ProcessedArgument` variable. For example, `\ReverseBoolean` is defined

as

引数プロセッサは、引数が基になるコードに渡される前に、その引数を操作できるようにします。新しいプロセッサの実装は、末尾に 1 つの引数を取り、結果を\ProcessedArgument 変数に格納する関数として作成できます。例えば、\ReverseBoolean は

```
\ExplSyntaxOn
\cs_new_protected:Npn \ReverseBoolean #1
  {
    \bool_if:NTF #1
      { \tl_set:Nn \ProcessedArgument { \c_false_bool } }
      { \tl_set:Nn \ProcessedArgument { \c_true_bool } }
  }
\ExplSyntaxOff
```

と定義されます。

[As an aside: the code is written in expl3, so we don't have to worry about spaces creeping into the definition.]

［補足：このコードは expl3 で書かれているため、定義に空白が紛れ込む心配はありません。］

# 3 Copying and showing (robust) commands and environments（堅牢な）コマンドや環境をコピーして表示する

If you want to (slightly) alter an existing command you may want to save the current definition under a new name and then use that in a new definition. If the existing command is robust, then the old trick of using the low-level \let for this doesn't work, because it only copies the top-level definition, but not the part that actually does the work. As most LaTeX commands are nowadays robust, LaTeX now offers some high-level declarations for this instead.

既存のコマンドを（少し）変更したい場合、現在の定義を別の名前で保存し、それを新しい定義の中で利用するのがよいでしょう。既存のコマンドが堅牢なものなら、低レベルの\let を使うという従来の手法はうまくいきません。なぜなら、\let は最上位の定義をコピーするだけで、実際に処理を行う部分まではコピーし

ないからです。現在では LaTeX のコマンドの多くが堅牢になっているため、LaTeX ではその代わりに高レベルの宣言が用意されています。

However, please note that it is usually better to make use of available hooks (e.g., the generic command or environment hooks), instead of copying the current definition and thereby freezing it; see the hook management documentation `lthooks-doc.pdf` for details.

ただし、現在の定義をコピーしてその時点の内容に固定してしまうのではなく、利用可能なフック（例えば、汎用的なコマンドフックや環境フックなど）を活用する方が通常は望ましいことに注意してください。詳細については、フック管理に関するドキュメント `lthooks-doc.pdf` を参照してください。

訳注：フック (hook) は、既存の処理の前後や途中の決まった位置に追加のコードを差し込むための仕組みを指す。A *hook* refers to a mechanism for inserting additional code at specific points, before, after, or during, existing processes.

`\NewCommandCopy` {⟨*cmd*⟩} {⟨*existing-cmd*⟩}
`\RenewCommandCopy` {⟨*cmd*⟩} {⟨*existing-cmd*⟩}
`\DeclareCommandCopy` {⟨*cmd*⟩} {⟨*existing-cmd*⟩}

This copies the definition of ⟨*existing-cmd*⟩ to ⟨*cmd*⟩. After this ⟨*existing-cmd*⟩ can be redefined and ⟨*cmd*⟩ still works! This allows you to then provide a new definition for ⟨*existing-cmd*⟩ that makes use of ⟨*cmd*⟩ (i.e., of its old definition). For example, after

⟨*existing-cmd*⟩ の定義を ⟨*cmd*⟩ にコピーします。その後、⟨*existing-cmd*⟩ を再定義しても、⟨*cmd*⟩ は引き続き機能します。これにより、⟨*cmd*⟩（すなわち ⟨*existing-cmd*⟩ の古い定義）を利用する新しい ⟨*existing-cmd*⟩ の定義を与えることができます。例えば、

```
\NewCommandCopy\LaTeXorig\LaTeX
\RenewDocumentCommand\LaTeX{}{\textcolor{blue}{\LaTeXorig}}
```

all LaTeX logos generated with `\LaTeX` will come out in blue (assuming you have a color package loaded).

とした後は、`\LaTeX` で生成される LaTeX ロゴはすべて青色で生成されます（カラー用パッケージが読み込まれているとします）。

The differences between `\New...` and `\Renew...` are as elsewhere: i.e., you get an error depending on whether or not ⟨*cmd*⟩ already exists, or in case of `\Declare...` it is copied regardless. Note that there is no `\Provide...` declaration, because that would be of limited value.

`\New...` と `\Renew...` の違いは他の場合と同様です。すなわち、⟨*cmd*⟩ がすでに存在するかどうかに応じてエラーになるかどうかが決まり、`\Declare...` の場合

はその有無にかかわらずコピーされます。\Provide... という宣言がないことに注意してください。その理由は、それがあまり役に立たないからです。

If the ⟨*cmd*⟩ or ⟨*existing-cmd*⟩ can't be provided as a single token but need "constructing", you can use \ExpandArgs as explained in Section 4.

⟨*cmd*⟩ あるいは ⟨*existing-cmd*⟩ を単一のトークンとして与えられず、「組み立てる」必要がある場合は、セクション 4 で説明されているように\ExpandArgs を使用できます。

\ShowCommand {⟨*cmd*⟩}

This displays the meaning of the ⟨*cmd*⟩ on the terminal and then stops (just like the primitive \show). The difference is that it correctly shows the meaning of more complex commands, e.g., in case of robust commands it displays not only the top-level definition but also the actual payload code and in case of commands declared with \NewDocumentCommand, etc. it also gives you detailed information about the argument signature.

これは、画面に ⟨*cmd*⟩ の意味を表示し、その後停止します (プリミティブな\show と同様です)。違いは、より複雑なコマンドの意味も正しく表示できる点にあります。例えば、堅牢なコマンドの場合、最上位の定義だけでなく実際の本体コードも表示されますし、\NewDocumentCommand などで宣言されたコマンドの場合には、引数のシグネチャに関する詳細情報も提示されます。

訳注 : プリミティブ (primitive) とは TEX が最初から持っている命令のことです。A 'primitive' is a command that TEX possesses from the start.

\NewEnvironmentCopy {⟨*env*⟩} {⟨*existing-env*⟩}
\RenewEnvironmentCopy {⟨*env*⟩} {⟨*existing-env*⟩}
\DeclareEnvironmentCopy {⟨*env*⟩} {⟨*existing-env*⟩}

This copies the definition for environment ⟨*existing-env*⟩ to ⟨*env*⟩ (both the beginning and end code), i.e., it is simply applying \NewCommandCopy twice to the internal commands that define an environment, i.e., \⟨*env*⟩ and \end⟨*env*⟩. The differences between \New..., \Renew..., and \Declare... are the usual ones.

これは、環境 ⟨*existing-env*⟩ の定義 (開始コードと終了コードの両方) を ⟨*env*⟩ にコピーします。言い換えれば、環境を定義する内部コマンド \⟨*env*⟩ と\end⟨*env*⟩ に対して \NewCommandCopy を 2 回適用するのと同じです。\New...、\Renew...、\Declare... の違いは、通常のものと同様です。

`\ShowEnvironment {⟨env⟩}`

This displays the meaning of the begin and end code for environment ⟨*env*⟩.
これは、環境 ⟨*env*⟩ の開始コードと終了コードの意味を表示します。

# 4 Preconstructing command names (or otherwise expanding arguments) コマンド名の事前構築（または引数の展開

When declaring new commands with `\NewDocumentCommand` or `\NewCommandCopy` or similar, it is sometimes necessary to "construct" the csname. As a general mechanism the L3 programming layer has `\exp_args:N...` for this, but there is no mechanism for it if `\ExplSyntaxOn` is not active (and mixing programming and user interface level commands is not a good approach anyhow). We therefore offer a mechanism to access this ability using CamelCase naming.

新しいコマンドを`\NewDocumentCommand` や `\NewCommandCopy` などで宣言する際には、制御綴り（csname）を「構築」する必要が生じることがあります。L3 プログラミング層には、そのための一般的な仕組みとして `\exp_args:N...` がありますが、`\ExplSyntaxOn` が有効でない場合には、それに相当する仕組みがありません（そもそも、プログラミング用コマンドとユーザーインターフェース用コマンドを混在させるのは適切な手法とは言えません）。そこで、CamelCase（キャメルケース）の命名規則を用いることで、この機能を利用できる仕組みを提供しています。

訳註：L3 プログラミング層とは expl3 が提供している仕組みのことです。The L3 programming layer refers to the mechanism provided by expl3.

`\UseName {⟨string⟩}`
`\ExpandArgs {⟨spec⟩} {⟨cmd⟩} {⟨arg1⟩} ...`

`\UseName` turns the ⟨*string*⟩ directly into a csname and then executes it: this is equivalent to the long-standing LaTeX $2_\varepsilon$ internal command `\@nameuse`, or the L3 programming equivalent `\use:c`. `\ExpandArgs` takes a ⟨*spec*⟩ which describes how to expand the ⟨*arguments*⟩, carries out these operations then executes the ⟨*cmd*⟩. The ⟨*spec*⟩ uses the descriptions offered by the L3 programming layer, and the relevant `\exp_args:N...` function must exist. Common cases will have a ⟨*spec*⟩ of `c`, `cc` or `Nc`: see below.

`\UseName` は ⟨*string*⟩ を、そのまま制御綴りに変換して実行します。これは、古くからある LaTeX $2_\varepsilon$ の内部コマンド `\@nameuse` や、L3 プログラミング層にお

ける同等のコマンド `\use:c` と同じものです。`\ExpandArgs` は、⟨*arguments*⟩ をどのように展開するかを指定する ⟨*spec*⟩ を受け取り、その展開処理を行った上で ⟨*cmd*⟩ を実行します。ここで使用される ⟨*spec*⟩ は L3 プログラミング層の仕様に基づいたものであり、対応する `\exp_args:N...` 関数が存在している必要があります。一般的なケースでは、⟨*spec*⟩ として `c`、`cc`、または `Nc` が用いられます (後述)。

訳注 : 展開 (expand) とは、マクロを実際の内容に置き換える処理です。Expansion is the process of replacing a macro with its actual content.

As an example, the following declaration provides a method to generate copyedit commands:

```
\NewDocumentCommand\newcopyedit{mO{red}}
  {%
    \newcounter{todo#1}%
    \ExpandArgs{c}\NewDocumentCommand{#1}{s m}%
      {%
        \stepcounter{todo#1}%
        \IfBooleanTF {##1}%
          {\todo[color=#2!10]{\UseName{thetodo#1}: ##2}}%
          {\todo[inline,color=#2!10]{\UseName{thetodo#1}: ##2}}%
      }%
  }
```

Given that declaration you can then write `\newcopyedit{note}[blue]` which defines the command `\note` and the corresponding counter for you.

その宣言を与えられれば、`\newcopyedit{note}[blue]` と記述することで、`\note` コマンドとそれに対応するカウンタを定義できます。

A second example is to copy a command by string name using `\NewCommandCopy`: here we might need to construct both command names.

2 つ目の例は、`\NewCommandCopy` を使って文字列名でコマンドをコピーする場合です。この場合には、両方のコマンド名を構築する必要が生じることがあります。

```
\NewDocumentCommand\savebyname{m}
  {\ExpandArgs{cc}\NewCommandCopy{saved#1}{#1}}
```

In the ⟨*spec*⟩ each `c` stands for one argument that is turned into a '`c`'ommand. An `n` represents a '`n`'ormal argument that is not altered and `N` stands for a '`N`'ormal argument which is also left unchanged, but one consisting only of a single token (and usually unbraced). Thus, to construct a command from a

string only for the second argument of `\NewCommandCopy` you would write

⟨*spec*⟩ では、各 `c` は、その引数 1 つを制御綴りとして構成することを表します。`n` は変更を加えない通常の引数を表し、`N` も同様に変更しない引数を表しますが、こちらは単一のトークンだけから成る引数（通常は波括弧で囲まれません）です。したがって、`\NewCommandCopy` の第 2 引数だけを文字列からコマンドとして構成したい場合には、次のように書きます。

```
\ExpandArgs{Nc}\NewCommandCopy\mysectionctr{c@section}
```

There are several other single letters supported in the L3 programming layer that *could* be used in the ⟨*spec*⟩ to manipulate arguments in other ways. If you are interested, take a look at the "Argument expansion" section in the L3 programming layer documentation in `interface3.pdf`.

L3 プログラミング層では、このほかにもいくつかの 1 文字指定がサポートされており、⟨*spec*⟩ の中でそれらを使って引数を別の方法で操作することもできます。興味がある方は、`interface3.pdf` にある L3 プログラミング層の文書の "Argument expansion" 節を参照してください。

# 5 Expandable floating point (and other) calculations 展開可能な浮動小数点（およびその他）の計算

The LaTeX3 programming layer which is part of the format offers a rich interface to manipulate floating point variables and values. To allow for (simpler) applications to use this on document-level or in packages otherwise not making use of the L3 programming layer a few interface commands are made available.

フォーマットの一部を成す L3 プログラミング層は、浮動小数点変数や値を操作するための豊富なインターフェースを提供しています。文書レベルや、本来は L3 プログラミング層を利用していないパッケージにおいて、より手軽にこの機能を使えるようにするため、いくつかの利用者向けコマンドが提供されています。

訳注：原文の"The LaTeX3 programming layer"を、他の記述に揃えるために、「L3 プログラミング層」と訳した。The term "The LaTeX3 programming layer" in the original text has been translated as "L3 プログラミング層" to ensure consistency with other descriptions.

`\fpeval {`⟨*floating point expression*⟩`}`

The expandable command `\fpeval` takes as its argument a floating point expression and produces a result using the normal rules of mathematics. As this

command is expandable it can be used where TEX requires a number and for example within a low-level `\edef` operation to give a purely numerical result.

展開可能なコマンド`\fpeval`は、引数として浮動小数点式を受け取り、通常の数学の規則に従って結果を生成します。このコマンドは展開可能であるため、TEXが数値を必要とする場面や、例えば純粋な数値結果を得るための低レベルな`\edef`操作内などで使用することができます。

Briefly, the floating point expressions may comprise:

要約すると、浮動小数点式は以下で構成され得ます：

- Basic arithmetic: addition $x$ `+` $y$, subtraction $x$ `-` $y$, multiplication $x$ `*` $y$, division $x$ `/` $y$, square root `sqrt` $x$, and parentheses.

  基本的な算術演算：加算 $x$ `+` $y$、減算 $x$ `-` $y$、乗算 $x$ `*` $y$、除算 $x$ `/` $y$、平方根 `sqrt` $x$、および丸括弧（parenthese）。

- Comparison operators: $x$ `<` $y$, $x$ `<=` $y$, $x$ `>?` $y$, $x$ `!=` $y$ *etc.*

  比較演算子：$x$ `<` $y$、$x$ `<=` $y$、$x$ `>?` $y$、$x$ `!=` $y$ など。

  The relation $x$ `?` $y$ is true exactly if one or both operands is `NaN` or is a tuple, unless they are equal tuples. Each ⟨*relation*⟩ can be any (non-empty) combination of `<`, `=`, `>`, and `?`, plus an optional leading `!` (which negates the ⟨*relation*⟩), with the restriction that the negated ⟨*relation*⟩ may not start with `?`.

  関係 $x$ `?` $y$ は、両オペランドが等しいタプルである場合を除き、少なくとも一方のオペランドが `NaN` であるか、またはタプルであるときに真になります。各 ⟨*relation*⟩ には、`<`、`=`、`>`、`?`の空でない任意の組合せを使え、さらに先頭に任意で`!`を付けてその関係を否定できます。ただし、否定された ⟨*relation*⟩ が`?`で始まってはならないという制約があります。

- Boolean logic: sign `sign` $x$, negation `!` $x$, conjunction $x$ `&&` $y$, disjunction $x$ `||` $y$, ternary operator $x$ `?` $y$ `:` $z$.

  ブール論理：符号 `sign` $x$、否定 `!` $x$、論理積 $x$ `&&` $y$、論理和 $x$ `||` $y$、三項演算子 $x$ `?` $y$ `:` $z$。

- Exponentials: `exp` $x$, `ln` $x$, $x$`^`$y$.

  指数・対数・べき乗：`exp` $x$、`ln` $x$、$x$`^`$y$。

- Integer factorial: `fact` $x$.

整数の階乗：`fact` $x$。

- Trigonometry: `sin` $x$, `cos` $x$, `tan` $x$, `cot` $x$, `sec` $x$, `csc` $x$ expecting their arguments in radians, and `sind` $x$, `cosd` $x$, `tand` $x$, `cotd` $x$, `secd` $x$, `cscd` $x$ expecting their arguments in degrees.

  三角関数：引数はラジアンでとる `sin` $x$、`cos` $x$、`tan` $x$、`cot` $x$、`sec` $x$、`csc` $x$、および引数を度数法（度）でとる `sind` $x$、`cosd` $x$、`tand` $x$、`cotd` $x$、`secd` $x$、`cscd` $x$。

- Inverse trigonometric functions: `asin` $x$, `acos` $x$, `atan` $x$, `acot` $x$, `asec` $x$, `acsc` $x$ giving a result in radians, and `asind` $x$, `acosd` $x$, `atand` $x$, `acotd` $x$, `asecd` $x$, `acscd` $x$ giving a result in degrees.

  逆三角関数：`asin` $x$, `acos` $x$, `atan` $x$, `acot` $x$, `asec` $x$, `acsc` $x$（結果はラジアン単位)、および `asind` $x$, `acosd` $x$, `atand` $x$, `acotd` $x$, `asecd` $x$, `acscd` $x$（結果は度単位）

- Extrema: `max`$(x_1, x_2, \ldots)$, `min`$(x_1, x_2, \ldots)$, `abs`$(x)$.

  最大値、最小値、絶対値：`max`$(x_1, x_2, \ldots)$, `min`$(x_1, x_2, \ldots)$, `abs`$(x)$.

- Rounding functions, controlled by two optional values, $n$ (number of places, 0 by default) and $t$ (behavior on a tie, `NaN` by default):

  丸め関数は、2 つのオプション値、$n$（桁数、デフォルトは 0）と $t$（同じ場合のデフォルトは `NaN`）によって制御されます。:

  - `trunc`$(x, n)$ rounds towards zero,

    `trunc`$(x, n)$ は 0 方向に丸めます。

  - `floor`$(x, n)$ rounds towards $-\infty$,

    `floor`$(x, n)$ は $-\infty$ 方向に丸めます。

  - `ceil`$(x, n)$ rounds towards $+\infty$,

    `ceil`$(x, n)$ は $+\infty$ 方向に丸めます。

  - `round`$(x, n, t)$ rounds to the closest value, with ties rounded to an even value by default, towards zero if $t = 0$, towards $+\infty$ if $t > 0$ and towards $-\infty$ if $t < 0$.

    `round`$(x, n, t)$ は最も近い値に丸めます。ちょうど中間の場合は、既定では最も近い偶数に丸められます。$t = 0$ なら 0 方向、$t > 0$ なら $+\infty$ の向きへ、$t < 0$ なら $-\infty$ の向きに丸められます。

- Random numbers: `rand`(), `randint`$(m, n)$.
  乱数: `rand`(), `randint`$(m, n)$.
- Constants: `pi`, `deg` (one degree in radians).
  定数: `pi`, `deg`（1 度をラジアンで表したもの)。
- Dimensions, automatically expressed in points, *e.g.*, `pc` is 12.
  寸法：自動的にポイント単位で表されり。例えば `1pc` は 12 として扱われます。
- Automatic conversion (no need for `\number`) of integer, dimension, and skip variables to floating points numbers, expressing dimensions in points and ignoring the stretch and shrink components of skips.
  整数、寸法、およびスキップ変数の浮動小数点数への自動変換（`\number` は不要)。寸法はポイント単位で表され、スキップの伸縮成分 (stretch/shrink) は無視されます。
- Tuples: $(x_1, \ldots, x_n)$ that can be added together, multiplied or divided by a floating point number, and nested.
  タプル：$(x_1, \ldots, x_n)$ の形をとり、相互に加算でき、浮動小数点数による乗算・除算が可能で、入れ子にもできます。

An example of use could be the following:
使用例は以下の通りです：

```
\LaTeX{} can now compute: $ \frac{\sin (3.5)}{2} + 2\cdot 10^{-3}
  = \fpeval{sin(3.5)/2 + 2e-3} $.
```

which produces the following output:
これにより、次のような出力が得られます：

LaTeX can now compute: $\frac{\sin(3.5)}{2} + 2 \cdot 10^{-3} = -0.1733916138448099$.

`\inteval {`⟨*integer expression*⟩`}`

The expandable command `\inteval` takes as its argument an integer expression and produces a result using the normal rules of mathematics with some restrictions, see below. The operations recognized are `+`, `-`, `*` and `/` plus parentheses.

As this command is expandable it can be used where TEX requires a number and for example within a low-level `\edef` operation to give a purely numerical result.

展開可能なコマンド `\inteval` は、引数として整数式を受け取り、通常の数学的規則（ただし一部の制限あり。後述）に従って結果を生成します。利用可能な演算は+, -, *, / および丸括弧です。このコマンドは展開可能であるため、TEX が数値を必要とする場面や、例えば低レベルの`\edef` 処理内で純粋な数値結果を得るために使用することができます。

This is basically a thin wrapper for the primitive `\numexpr` command and therefore has some syntax restrictions. These are:

これは基本的にプリミティブな`\numexpr` コマンドの薄いラッパーであるため、いくつかの構文上の制約があります。それらは以下の通りです：

- `/` denotes division rounded to the closest integer with ties rounded away from zero;

  `/`は、最も近い整数への丸め（ただし、中間値の場合はゼロから遠い方の整数へ丸める）による除算を表します。

- there is an error and the overall expression evaluates to zero whenever the absolute value of any intermediate result exceeds $2^{31} - 1$, except in the case of scaling operations $a*b/c$, for which $a*b$ may be arbitrarily large;

  中間計算結果の絶対値が $2^{31} - 1$ を超えるとエラーとなり、式全体の評価結果はゼロになります。ただし、$a*b/c$ という形式のスケーリング演算（この場合、$a*b$ は任意に大きな値になり得ます）は、そうなりません。

- parentheses may not appear after unary `+` or `-`, namely placing `+(` or `-(` at the start of an expression or after `+`, `-`, `*`, `/` or `(` leads to an error.

  単項の+や-の直後に丸括弧を置くことはできません。すなわち、式の先頭や+、-、*、/、(の直後に+(や-(を記述するとエラーになります。

訳註：ここのスケーリング演算（scaling operation）とは数値の大きさ（スケール）を調整することです。ここでは、大きな数値 $a*b$ を別の数値 $c$ で割ることで、適切な範囲に収める操作です。The scaling operation here refers to adjusting the size (scale) of a number. In this case, it involves dividing a large number $a * b$ by another number $c$ to bring it within a suitable range.

An example of use could be the following.

使用例は以下のとおりです。

```
\LaTeX{} can now compute: The sum of the numbers is $\inteval{1 + 2 + 3}$.
```

which results in

この結果は次のようになります。

"LaTeX can now compute: The sum of the numbers is 6."

`\dimeval {`⟨*dimen expression*⟩`}` `\skipeval {`⟨*skip expression*⟩`}`

Similar to `\inteval` but computing a length (`dimen`) or a rubber length (`skip`) value. Both are thin wrappers around the corresponding engine primitives, which makes them fast, but therefore shows the same syntax peculiars as discussed above. Nevertheless, in practice they are usually sufficient. For example

`\inteval` と同様ですが、長さ（`dimen`）や伸縮可能な長さ（`skip`）の値を計算します。これらはいずれも対応するエンジンのプリミティブに対するごく薄いラッパーなので高速に動作しますが、その反面、前述したのと同じ構文上の制約もあります。とはいえ、実際にはこれらで十分な場合がほとんどです。例えば、

```
\NewDocumentCommand\calculateheight{m}{%
  \setlength\textheight{\dimeval{\topskip+\baselineskip*\inteval{#1-1}}}}
```

sets the `\textheight` to the appropriate value if a page should hold a specific number of text lines. Thus after `\calculateheight{40}` it is set to 595.0pt, given the values `\topskip` (10.0pt) and `\baselineskip` (15.0pt) in the current document. です。これは、1 ページに特定の行数の本文を収めるために、`\textheight` を適切な値に設定します。したがって、`\calculateheight{40}`を実行すると、現在の文書における`\topskip`（10.0pt）と`\baselineskip`（15.0pt）の値に基づいて、`\textheight` は 595.0pt に設定されます。

# 6 Expandable `\input` equivalent 展開可能な `\input` 相当のコマンド

`\expandableinput {`⟨*filename*⟩`}`

The LaTeX definition of `\input` cannot be used in places where TeX is performing expansion: the classic example is at the start of a tabular cell. There are a number of reasons for this: the key ones are that `\input` records which files are read and provides pre- and post-file hooks.

LaTeX の`\input` 定義は、TeX が展開中の箇所では使用できません。典型的な例として、tabular 環境のセルの先頭が挙げられます。これにはいくつかの理由があ

りますが、主なものとしては、どのファイルが読み込まれたかを記録する\input の機能や、ファイル読み込みの前後に実行されるフックを提供する機能を備えている点が挙げられます。

To support the need to carry out file input in expansion contexts, the command \expandableinput is available: this skips recording the file name and does not apply any file hooks, but otherwise behaves like \input. In particular, it still uses \input@path when doing file lookup.

展開中の文脈でもファイル入力を行えるようにするため、\expandableinput というコマンドが用意されています。このコマンドは、ファイル名を記録せず、ファイルフックも適用しませんが、\input と同様に動作します。特に、ファイルの検索時には\input@path が使用されます。

# 7 Case changing 大文字と小文字の変換

\MakeUppercase [⟨*keyvals*⟩] {⟨*text*⟩}
\MakeLowercaes [⟨*keyvals*⟩] {⟨*text*⟩}
\MakeTitlecase [⟨*keyvals*⟩] {⟨*text*⟩}

TEX provides two primitives \uppercase and \lowercase for changing the case of text. However, these have a range of limitations: they only change the case of explicit characters, do not account for the surrounding context, do not support UTF-8 input with 8-bit engines, etc. To overcome this problem, LATEX provides the commands \MakeUppercase, \MakeLowercase and \MakeTitlecase: these offer significant enhancement over the TEX primitives. These commands are engine-robust (\protected), and so can be used in moving arguments.

TEX には、二つのプリミティブとして文字の大文字と小文字とを変換する \uppercase と\lowercase が用意されています。しかし、これらにはいくつかの制限があります：明示的に与えられた文字しか変換できない、周囲の文脈を考慮しない、8 ビットエンジンで UTF-8 入力をサポートしていない、といった点です。この問題を解決するために、LATEX では\MakeUppercase、\MakeLowercase、\MakeTitlecase というコマンドが提供されています。これらは TEX のプリミティブに比べて大幅に機能が強化されています。これらのコマンドは \protected で定義された堅牢なコマンドなので、移動引数（moving arguments）の中で使用することが可能です。

訳注：MakeLowercaes は MakeLowercase のスペルミス。The *misspelling* 'MakeLowercaes' should be 'MakeLowercase'.

訳注：移動引数（moving arguments）とは、目次・見出し・柱・しおりなどように、いったん別の場所へ書き出されて後で再利用される可能性のある引数のことです。A 'moving argument' refers to an argument that is written out to a different location, such as a table of contents, heading, running head, or bookmark, and may be reused later.

Upper- and lower-casing are well-understood in general conversation. Title-casing here follows the definition given by the Unicode Consortium: the first

character of the input will be converted to (broadly) uppercase, and the rest of the input to lowercase. The full range of Unicode UTF-8 input can be supported.

大文字化と小文字化については、一般にもよく知られています。ここで見出し用大文字表記は、Unicode コンソーシアムの定義に従います。つまり、入力の最初の文字は（広義の）大文字に変換され、それ以降の文字は小文字に変換されます。Unicode UTF-8 入力の全範囲をサポートできます。

訳注：*titlecase* を、ここでは見出し用大文字表記と訳しました。日本語には当てはまりませんが、書名や章の名前などの見出しのときに、単語の最初の 1 文字目だけを大文字にする規則のことです。`\MakeTitlecase` は、デフォルトでは、与えられた文字列全体の、最初の 1 文字目だけを大文字にします。Here, *titlecase* has been translated as "capitalized title for headings." Although this doesn't apply to Japanese, it refers to the rule of capitalizing only the first letter of a word when it's used as a heading, such as a book title or chapter title. By default, `\MakeTitlecase` capitalizes only the very first character of the provided string.

```
\MakeUppercase{hello WORLD üé}   HELLO WORLD SSÜÉ
\MakeLowercase{hello WORLD üé}   hello world SSüé
\MakeTitlecase{hello WORLD üé}   Hello WORLD SSüé
```

The case-changing commands take an optional argument which can be used to tailor the output. This optional argument accepts the key `locale`, also available under the alias `lang`, which can be used to give a language identifier in BCP-47 format. This is then applied to select language-specific features during case-changing.

大文字と小文字を変換するコマンドは、出力を調整するためのオプション引数を受け付けます。このオプション引数では、BCP-47 形式の言語識別子を指定するために `locale` キー（`lang` という別名でも指定可能）を使用できます。指定された識別子は、変換処理において言語固有の機能を適用する際に利用されます。

訳註：BCP-47 形式は、言語や地域を表すためのタグの書き方です。たとえば、ja は日本語を指しますが、ja-JP と書けば日本向けの日本語（ロケール `locale`）を表します。同様に、en-US は米国英語、zh-Hant-TW は台湾で使う繁体字中国語のロケールを意味します。この形式で表されるタグは、「どの言語・文字体系・地域か」を明確に示します。このように表されたときは「どの言語・文字体系・地域か」を表します。The BCP-47 format is a standardized way of writing locale tags to represent languages, regions, and scripts. For example, 'ja' denotes the Japanese language, while 'ja-JP' specifies the Japanese locale as used in Japan. Similarly, 'en-US' refers to the US English locale, and 'zh-Hant-TW' indicates the Traditional Chinese locale as used in Taiwan. These tags precisely identify the language, writing system, and regional conventions.

For titlecasing, the key `words` may also be used: this takes a choice of `first` or `all`. The standard setting is `first`, and means that only the very first "letter" is (broadly) uppercased. The alternative, `all`, means that the input is divided at each space, and for each word that results, the first letter is uppercased. For example

見出し用大文字表記への変換には、二つのキーワードが用意されています。それらは `first` と `all` です。デフォルト設定は `first` であり、これは、最初の「文字」だけが（概ね）大文字化されることを意味します。もうひとつの選択肢である `all` は、入力を空白ごとに分割し、分割された各単語の先頭文字を大文字にすることを意味します。例えば、

```
\MakeTitlecase[words = first]{some words}
\MakeTitlecase[words = all]{some words}
```

gives "Some words Some Words".

とすると、"Some words Some Words"となります。

The input given to these commands is “expanded” before case changing is applied. This means that any commands within the input that convert to pure text will be case changed. Mathematical content is automatically excluded, as are the arguments to the commands `\label`, `\ref`, `\cite`, `\begin` and `\end`. Additional exclusions can be added using the command `\AddToNoCaseChangeList`. Input can be excluded from case changing using the command `\NoCaseChange`.

これらのコマンドに渡される入力は、大文字と小文字の変換が適用される前に「展開」されます。つまり、入力に含まれるコマンドのうち、純粋なテキストに変換されるものは変換の対象となります。数式部分は自動的に除外されるほか、`\label`、`\ref`、`\cite`、`\begin`、そして`\end`といったコマンドの引数も除外されます。`\AddToNoCaseChangeList`コマンドを使用すれば、除外対象を追加することも可能です。また、`\NoCaseChange`コマンドを使用することで、特定の入力を変換対象から除外できます。

| | |
|---|---|
| `\MakeUppercase{Some text $y = mx + c$}` | SOME TEXT $y = mx + c$ |
| `\MakeUppercase{\NoCaseChange{iPhone}}` | iPhone |

Individual words can be excluded globally from case changing using the commands `\DeclareLowercaseExclusions`, `\DeclareTitlecaseExclusions`, and `\DeclareUppercaseExclusions`, each of which take a comma-seperated list of words to be excluded. For example, to titlecase in English in the manner often seen for book titles, one might use

特定の単語を大文字と小文字変換の対象から一括して除外するには、コマンド`\DeclareLowercaseExclusions`、`\DeclareTitlecaseExclusions`、および`\DeclareUppercaseExclusions`を使うことで行えます。これらのコマンドは、いずれも除外対象となる単語をカンマで区切ったリストを引数として受け取ります。例えば、書籍のタイトルなどでよく見られる英語の見出し用大文字表記を適用する場合、

```
\DeclareTitlecaseExclusions{a,an,and,on,of,the}
\MakeTitlecase[words = all]{Of mice and men}
\MakeTitlecase[words = all]{The mill on the floss}
\MakeTitlecase[words = all]{The comedy of errors}
```

to obtain

とすると、次のような結果が得られます。

Of Mice and Men
The Mill on the Floss
The Comedy of Errors

To allow robust commands to be used within case changing *and* to produce the expected output, two additional control commands are available. `\CaseSwitch` allows the user to specify the result for the four possible cases

大文字と小文字変換の中でも堅牢なコマンドを使い、さらに期待どおりの出力を得られるようにするために、2 つの追加制御コマンドが用意されています。`\CaseSwitch` を使用すると、次の 4 つの場合

- No case changing
  大文字と小文字の間の変換を行わない
- Uppercasing
  大文字にする
- Lowercasing
  小文字にする
- Titlecasing (only applies for the start of the input)v 見出し用大文字表記（入力の先頭のみに適用）

のそれぞれについて結果を指定できます。

The command `\DeclareCaseChangeEquivalent` provides a way to substitute a command by an alternative version when it is found inside a case changing situation. There are three commands for customising the case changing of codepoints

コマンド `\DeclareCaseChangeEquivalent` は、大文字と小文字との変換が行われる文脈で、あるコマンドを別のバージョンに置き換えるための手段を提供します。コードポイントの大文字と小文字変換をカスタマイズするためのコマンドは 3 つあります。

`\DeclareLowercaseMapping [`⟨*locale*⟩`] {`⟨*codepoint*⟩`} {`⟨*output*⟩`}`
`\DeclareTitlecaseMapping [`⟨*locale*⟩`] {`⟨*codepoint*⟩`} {`⟨*output*⟩`}`
`\DeclareUppercaseMapping [`⟨*locale*⟩`] {`⟨*codepoint*⟩`} {`⟨*output*⟩`}`

All three take a ⟨*codepoint*⟩ (as an integer expression) and will result in the ⟨*output*⟩ being produced under the appropriate case changing operation. The

optional ⟨*locale*⟩ can be given if the mapping should only apply to a specific one: this is given in BCP-47 format (https://en.wikipedia.org/wiki/IETF_language_tag). For example, the kernel customises the mapping for U+01F0 (ǰ) when uppercasing in 8-bit engines:

これら 3 つのコマンドはすべて、引数として ⟨*codepoint*⟩（整数式として与える）を受け取り、適切な大文字と小文字との変換の際に ⟨*output*⟩ を生成します。特定のロケールに対してのみ変換を適用したい場合は、オプションの ⟨*locale*⟩ を指定できます。この指定には BCP-47 形式（https://en.wikipedia.org/wiki/IETF_language_tag）を用います。例えば、8 ビットエンジンで大文字変換を行う際、カーネルは U+01F0 (ǰ) の変換規則を次のようにカスタマイズしています：

```
\DeclareUppercaseMapping{"01F0}{\v{J}}
```

zas there is no pre-composed J̌ character, and this is problematic if the engine does not support Unicode natively. Similarly, to set a locale `xx` to behave in the same way as Turkish and retain the difference between dotted- and dotless-i, one could use for example

合成済み文字（pre-composed character）の J̌ は存在しないため、エンジンが Unicode をネイティブにサポートしていない場合には問題が生じます。同様に、ロケール `xx` をトルコ語と同様に動作させ、ドット付きの i とドットなしの i の区別を維持したい場合は、例えば次のように設定できます。

```
\DeclareLowercaseMapping[xx]{"0049}{\i}
\DeclareLowercaseMapping[xx]{"0130}{i}
\DeclareUppercaseMapping[xx]{"0069}{\.{I}}
\DeclareUppercaseMapping[xx]{"0131}{I}
```

# 8 Text sub- and superscripts テキスト用の下付きと上付き添え字

`\textsubscript {`⟨*text*⟩`}`
`\textsuperscript {`⟨*text*⟩`}`

Semantically, some sub- and superscripts are clearly text rather than math mode, for example footnote markers. To support these textual uses, the commands `\textsubscript` and `\textsuperscript` are available.

意味の上では、下付きや上付きの中には、明らかに数式モードではなくテキストとして扱うべきものがあります。例えば脚注の番号や記号などです。こうしたテキストとしての用途に対応するため、`\textsubscript` および `\textsuperscript` というコマンドが用意されています。

`\textsubscript@offset`
`\textsubscript@space`
`\textsuperscript@offset`
`\textsuperscript@space`

The standard settings for these commands use the same placement parameters as TEX would use in math mode.[1] Two aspects of placement can be adjusted by setting the control commands to updated values:

これらのコマンドの標準設定では、TEX が数式モードで用いるのと同じ配置パラメータが使われます。[1]配置に関する次の 2 つの点は、これらの制御コマンドに新しい値を設定することで調整できます：

- The vertical offset of the sub- or superscript: this is a dimension expression (as described for `\dimeval`), stored in a command so that it is evaluated at the time of use. The standard setting uses the font dimensions of the current math font, but for new documents, using a value based on the text font is recommended. For example

  下付き文字または上付き文字の垂直方向のオフセット：これは寸法を示す式（`\dimeval` で説明されているとおり）であり、コマンドに格納されているため、使用時に評価されます。標準設定では、現在の数式フォントに由来する寸法が使用されますが、新規文書では、テキストフォントに基づいた値を使用することをお勧めします。例えば、

  ```
  \renewcommand*\textsuperscript@offset{0.86ex}
  ```

  will set the vertical offset to 86 % of the x-height of the current font.

  は、垂直方向のオフセットを現在のフォントの x ハイトの 86 パーセントに設定します。

- The space added after the sub- or superscript: the standard setting is a space of the same size as would be used in math mode (`\scriptspace`);

[1] This reflects the history of these commands: previous versions were implemented using math mode internally for placement.
これはこれらのコマンドの履歴を反映しています。以前の版では、配置のために内部的に数式モードを用いて実装されていました。

this space will be retained if the script occurs immediately before a line-break. It may be preferable in new documents to allow the space to be discarded here, setting for example,

下付き文字または上付き文字の後に挿入される空白：標準設定では、数式モードで使用される空白（`\scriptspace`）と同じ大きさの空白が使用されます。この添え字が改行の直前にある場合、この空白は保持されます。新しい文書では、この空白がここで捨てられるようにしておく方が望ましい場合もあります。例えば、次のように設定できます。

```
\renewcommand*\textsuperscript@space{\hspace{0.5pt}}
```

# 9 Support for problem solving 問題解決のための手助け

`\listfiles [`⟨*options*⟩`]`

If this command is placed in the preamble then a list of the files read in (as a result of processing the document) will be displayed on the terminal (and in the log file) at the end of the run. Where possible, a short description will also be produced. These descriptions will (hopefully) include the descriptions, dates and version numbers for package and class files.

このコマンドをプリアンブルに記述すると、文書の処理終了時に、読み込まれたファイルの一覧が端末（およびログファイル）に表示されます。可能な場合には、簡単な説明も併せて出力されます。この説明には、パッケージファイルやクラスファイルに関する解説、日付、バージョン番号などが（運が良ければ）含まれているでしょう。

Sometimes, it may be that a local edit has been made to a package or class file (or rather a copy of such a file). To allow these cases to be identified, `\listfiles` takes an optional argument which allows adjustment of the information printed using a key–value approach

場合によっては、パッケージファイルやクラスファイル（あるいはそのコピー）に対して、ローカルな修正が加えられていることがあります。こうした場合を識別できるように、`\listfiles` コマンドにはオプション引数が用意されており、キー–値の形式を用いて出力情報を調整できるようになっています。

**`hashes`** Adds the MD5 hash for each file to the information printed

各ファイルの MD5 ハッシュを、表示される情報に追加します

**sizes** Adds the file size for each file to the information printed

各ファイルのファイルサイズを、表示される情報に追加します

Note that as Windows and Unix use different line endings (LF *versus* LF CR), the hashes and file sizes from the two systems will not be the same. As such, you should compare these values between operating systems of the same type.

Windows と Unix では改行コードが異なるため（LF に対して LF CR）、ハッシュ値とファイルサイズは一致しません。そのため、これらの値を比較する際は、同じ種類のオペレーティングシステム間で比較するようにしてください。

*Warning*: this command will list only files which were read using LaTeX commands such as `\input{`⟨*file*⟩`}` or `\include{`⟨*file*⟩`}`. If the file was read using the primitive TeX syntax `\input` *file* (without `{ }` braces around the file name) then it will not be listed; failure to use the LaTeX form with the braces can cause more severe problems, possibly leading to overwriting important files, so **always put in the braces**.

警告：このコマンドは、`\input{`⟨*file*⟩`}`や`\include{`⟨*file*⟩`}`などの LaTeX コマンドを使用して読み込まれたファイルのみを一覧表示します。ファイルがプリミティブな TeX 構文`\input` *file*（ファイル名の周りに`{ }`波括弧がない場合）を使用して読み込まれた場合は、一覧表示されません。波括弧を使用した LaTeX 形式を使用しないと、重要なファイルが上書きされるなど、より深刻な問題が発生する可能性があります。そのため、**必ず波括弧を使用してください**。