

徹底解説！ `\expandafter` 活用術（キホン編）

`\expandafter` is 何

`\expandafter` は \TeX 言語のプリミティブの一種ですが、 \TeX 言語のコードを書くときに限らず、(La) \TeX 者の日常生活における様々な場面で利用されています。


1. お酒のネタにする（例）
2. （特に \TeX 関連の）会議におけるノベルティグッズの意匠にする（例）
3. 絵を描く際のモチーフにする（例 1・例 2）
4. 独自の健康法のモチーフにする（例）
5. 独自の体操のモチーフにする（例 1・例 2）
6. とにかく連呼する（例 1・例 2）
7. \TeX 言語のプログラムにおいて、展開制御のために書く

このうち、最初の6つに関しては、特に難しく考えることは何ともありません。単に本能の赴くがままに `\expandafter` の名を叫べばよいわけですから¹。

これに対して、最後の「展開制御のために書く」についてはそう簡単にはいきません。 \TeX 言語の学習が進んである程度複雑なプログラムを書く段階に至ると、「展開の順番を変えたい」と思って `\expandafter` を試す人は多いようです。しかし、この「展開制御」というのは \TeX 言語特有の概念であるため、数多くのプログラム言語を制覇した強者であっても、マトモな理解を得るのはなかなか容易ではありません²。

世の中には `\expandafter` の挙動を理解するための「チョット変わったアプローチ」を提唱している人もいます。しかし、この記事では特に奇をてらうことなく、`\expandafter` の素朴な定義にホンキで取り組むことでその理解を目指します。

前書きの最後に、念のため、コレを書いておきましょう。

 \TeX 言語注意！

Before `\expandafter`

そもそも「展開制御」が何かの役に立つためには「展開」についての理解が不可欠です。すなわち、 \TeX 言語の「展開」の仕様、またその前提となる「字句解析（トークン化）」の仕様について十分に理解していない間は、`\expandafter` などの「展開制御」に手を出すべきでないのは当然でしょう。

そこで、まずは「`\expandafter` 以前」の理解確認のため、簡単なクイズをやってみましょう。

¹ `\expandafter` の読み方について特に決まりことはありません。「エクスバンドアフター」でも「エキスパンドアフター」でも「エクスぺアアンディアフトゥ!!!!!!」でも好きなように叫んでください。
² 下手に自分の知っているプログラミング言語の概念で「近似」して先に急いで進もうとすると、本来の定義とのズレのために後々になって頭を抱えることになりかねません。

【クイズ1】次の \TeX のソースを字句解析すると何個のトークン³からなると見なされるか。ただし、カテゴリコードの設定は LaTeX の通常通りであると仮定する。

```
\someproc A {B C\relax } {\`\\}\nil %?
```

【クイズ2】次のコードが予め実行されていたとする。

```
\def\foo#1{\foo}
```

この場合、次のトークン列を（先頭で）一回展開した結果のトークン列はどうなるか。

```
\csname\foo\foo\endcsname\foo\foo\relax
```

自信をもって答えを出せたならば合格です⁴。 \expandafter をホンキで理解したい人にとっては楽勝でしたね！

表記のおやくそく

展開制御は、字句解析された結果のトークン列に対して行われるものです⁵。そのため、トークン列を曖昧さなく示すために、本記事では次のような「トークン列の記法」を使うことにします。

- 制御綴⁶のトークンを \csname 、文字トークンを A のように等幅フォントで書きます。特に空白文字トークンを $_$ と書きます。
- 見やすさのため、トークン列は $a\ b\ c$ のように間を空けて書きます。間に空白文字トークンがあるわけではないことに注意してください。空白文字トークンは常に $_$ で表されます。
- トークンを表す変数として A 、 B 、...などの英大文字を使います。さらに、トークン列を $A...$ のように表すことにします。
- 「トークン列 $A...$ を一回展開すると $B...$ になる」という言明を「 $A... \Rightarrow B...$ 」と表記します。

例えば、

```
\someproc A {B C\relax } {\`\\}\nil %?
```

（先の問題に出てきたもの）を字句解析した結果のトークン列は次のようになります。

```
\someprocA{BC\relax}{\`\\}\nil
```

※ 本記事では一貫して「 LaTeX 上の \TeX 言語プログラミング」を取り扱います。また、特に断りがない限り、「 LaTeX の \makeatletter の状態」のカテゴリコード設定を仮定します。

³ 本記事では、 \TeX の用語の「*token*」の訳語に「トークン」を用いることにします。

⁴ チョット気になる人のために正解を書いておきます。【クイズ1】15個です。【クイズ2】「 $\text{\foo\foo\foo\relax}$ 」となります。

⁵ もちろん、実際の \TeX 処理系においては字句解析と展開は同時進行で処理されます（参考記事）。それでも、展開に関する議論を行う場合は、「字句解析に関するややこしい話」が絡むのを避けるために、敢えて「字句解析が済んだ後のトークン列」を前提にする方が適切でしょう。（特に、展開だけが行われている状況では、字句解析に影響を与える「カテゴリコードの変更」が起こらないことに注意しましょう。）

⁶ 本記事では、 \TeX の用語の「*control sequence*」の訳語を「制御綴」とします。他文献では「コントロール・シーケンス」と呼ばれることもあります。

`\expandafter` のキホン

`\expandafter`、コワクナイヨー

一般に難解といわれる `\expandafter` ですが、実はその定義は、次の示す通り、いたって単純なものです。

1

```
B...⇒B'...  
であるとき  
\expandafterA B...⇒A B'...
```

これだけです。しかし、単純だからこそ、その確実な理解が大事なわけです。

`\expandafter` はなぜ動くのか

簡単な例で調べてみましょう。

```
\def\csAA{\csA\csA}  
\def\csBB{\csB\csB}
```

このようにマクロが定義されているとします。この時、トークン列 `\csAA\csBB` を一回展開するとどうなるでしょうか？

2

```
\csAA\csBB  
⇒\csA\csA\csBB
```

これは当たり前ですね。ではこの先頭に `\expandafter` を付けたものを考えましょう。これの一回展開はどうなるでしょうか？

`\expandafter\csAA\csBB`

定義に戻って考えましょう。今の場合、A に相当するのが `\csAA`、B... に相当するのが `\csBB` です。 `\expandafter` の定義に従うと、その展開結果を知るためには、B... の一回展開の結果である B'...を知る必要があります。

3

```
\csBB (←これが B...)  
⇒\csB\csB (←これが B'...)
```

`\expandafter` の展開結果は A B'...ですから、結局 `\csAA\csB\csB` となります。以上の結果を改めて整理してみましょう。

4

```
\expandafter\csAA\csBB  
  [\csBB ⇒\csB\csB]  
⇒\csAA\csB\csB
```

結局 `\expandafter` の追加により、A が展開される前に B が展開されたこととなります。

`\expandafter` のガイドライン

以上の結果を「どういつ場合に `\expandafter` を使うべきか」という観点で改めて整理してみます。

- A B...のようなトークン列があり、このままでは A が実行（展開）される。
- しかし、A の実行（展開）の前に B...を展開したい。
- その場合、A B...の直前に `\expandafter` を置けばよい。

以下ではこれを「`\expandafter` のガイドライン」と呼ぶことにします。

例題：はじめての `\expandafter`

定義が解ったところで、さっそく `\expandafter` を利用したコードを書いてみましょう。

【例題 1】 \TeX のプログラム中に、以下のような `\let` 文があった。

```
% \xName に結果(\xResult)を格納する
\let\xName\xResult
```

ところが、プログラムの改修で、`let`のコピー先の制御綴（`\xName`）を可変にする必要が生じた。このため、コピー先の制御綴を指定するためのマクロ `\xTargetCs` を用意した。

```
% 結果を格納する変数(マクロ名)
\def\xTargetCs{\xName}%
```

このとき、「`\xTargetCs` の内容⁷の制御綴（上例の場合は `\xName`）に `\xResult` をコピー（`\let`）する」というコードを書け。

もちろん以下のコードは正しくありません。

```
\let\xTargetCs\xResult % ダメ
```

これでは（`\xTargetCs` の内容である）`\xName` ではなく `\xTargetCs` そのものが書き換えられてしまいます。この状況を先程の「`\expandafter` のガイドライン」と照合してみましょう。

- `\let\xTargetCs` というトークン列があり、このままでは `\let` が先に実行される。
 - しかし、`\let` の実行の前に `\xTargetCs` を展開したい。
 - その場合、`\let\xTargetCs` の直前に `\expandafter` を置けばよい。
- ピッタリと当てはまりました。つまり、次のようにすればよいわけです。

```
% 例題 1 の解答
\verb{\expandafter}\let\xTargetCs\xResult
```

⁷ この記事では、引数無しのマクロについて、一回展開した結果のトークン列のことを便宜的に「内容」と呼ぶことにします。

期待通りに動くかどうか、^{simulate}シミュレートしてみましょう。

5

```
\expandafter\let\xTargetCs\xResult
[\xTargetCs ⇒ \xName]
⇒ \let\xName\xResult
```

大丈夫ですね。

例題：引数を完全展開したい話

`\expandafter` の最もキホンの的な使い方を心得たので、もう少し応用してみましょう。

【例題 2】 次のように、`\includegraphics` に渡すべき引数がマクロとして与えられている。

```
\def\xImageOpt{\xImageSysOpt,\xImageUserOpt}% \includegraphics の
オプション
\def\xImageSysOpt{width=.8\linewidth}
\def\xImageUserOpt{pagebox=artbox}
\def\xImageFile{image-1.pdf}% \includegraphics の対象のファイル名
```

次のような形の `\includegraphics` 文を実行したいが、引数がマクロのままでは正しく処理されない⁸。

```
% 引数のマクロを展開しないとダメ
\includegraphics[\xImageOpt]{\xImageFile}
```

「引数（だけ）を完全展開してから `\includegraphics` を実行する」ようにするコードを書け。

引数の部分「`[\xImageOpt]{\xImageFile}`」を完全展開したいわけなので、まずそこに `\edef` を適用することを考えます。

※トークン列の完全展開（full expansion）を得るには `\edef` が必要です。`\expandafter` では対処できません。

```
% 引数の部分を完全展開する
\edef\xArgs{[\xImageOpt]{\xImageFile}}
% これで \xArgs ⇒ [width=.8\linewidth,pagebox=artbox]image-1.pdf} となる
```

この `\xArgs` を `\includegraphics` 文のしかるべき位置に置きます。

```
% 未完成 \xArgs は " 引数 " である
\includegraphics\xArgs
```

ここで再び「`\expandafter` のガイドライン」を思い起こしましょう。

⁸ 一般的に、`key=value` 形式の引数について、「`key=value`」全体がマクロになっているとそれは正常にパースされません。

- `\includegraphics\xArgs` というトークン列があり、このままでは `\includegraphics` が先に実行される。
- しかし、`\includegraphics` の実行の前に `\xArgs` を展開したい。
- その場合、`\includegraphics\xArgs` の直前に `\expandafter` を置けばよい。というわけで答えは次のようになります。(最初の `\edef` から書いておきます。)

% 例題 2 の解答

```
\edef\xArgs{\xImageOpt}\xImageFile}
\verb\expandafter\includegraphics\xArgs
```

この「引数を `\edef` する → `\expandafter`」のコンボは応用範囲が広いので、ぜひ身につけておきましょう。

補足：`\expandafter` しない方法

先ほどの例題 2 ですが、次のように `\noexpand` を使っても解決できます。

% 例題 2 の別解

```
\edef\xNext{\noexpand\includegraphics\xImageOpt}\xImageFile}
\xNext
```

これは「次に実行すべき文の正しいトークン列を `\edef` と展開抑止を組み合わせで作る」というパターンです。これも応用が効くので覚えておくといいいでしょう。

チョット注意

トークンとはトークンである

ここでまたクイズです。次のようなコードを考えます。

```
\everypar\verb\expandafter{\the\everypar\xSomething}\xAnother
```

これを実行すると、`\everypar` の実行⁹の後に `\expandafter` が展開されますが、この時に（定義における）A に相当する部分は何でしょうか？

`\expandafter` の直後にあるトークンが A です。なので、この場合は `{`（開き波括弧）となります。これが正解です。

特に難しい話ではないはずですが、ここで \TeX 言語初心者がよくやる間違いは、「A の部分」を「`{\the\everypar\xSomething}`」だと考えてしまうことです。そもそも A は単一のトークンを表す変数であり、それが複数のトークンからなる列 `{\the\everypar\xSomething}` を表すことは絶対にありえません。 \TeX 言語の文法では `{...}` で囲まれた「グループ」を一体のものとして扱う場面がよくありますが、この「グループ」自体は決してトークンではないことに注意してください。

ちなみに、今の場合の `\expandafter` の展開過程は以下のようになります。ただしトークン列パラメタ `\everypar` の現在の値を「`\xFooBar`」とします¹⁰。

⁹ `\everypar` トークンの実行により、「`\everypar` パラメタに対する代入文」が開始されます。
¹⁰ 先頭にあった `\everypar` は `\expandafter` の展開時には既に「実行されてしまっている」ため入力バッファ上にはありません。（このため図では \langle に入れて示しました。）この後まで読んだところで、「`\everypar` の代入文」として「`\everypar{\xFooBar\xSomething}`」が完成することになります。

```

<\everypar> \expandafter{\the\everypar\xSomething}\xAnother
[\the\everypar ⇒ \xFooBar]
⇒ <\everypar> {\xFooBar\xSomething}\xAnother

```

展開不能トークンで `\expandafter` する件

`\expandafter A B...` の展開の際には B が展開されるわけですが、この時もし B が展開不能なトークンだったらどうなるでしょうか？

```

% この場合 Bは'\{'であり展開不能
\verb{\expandafter}\begin{array}\xArgs

```

この場合は、便宜的に「B は B 自身に展開される」と解釈されます。

```

\expandafter\begin{array}\xArgs
[\{ ⇒ \}] (と見なす)
⇒ \begin{array}\xArgs

```

つまり、B が展開不能な場合は `\expandafter` は全くの無駄、ということになります。

それでは、A が展開不能な場合の `\expandafter` はどうでしょうか？一見すると、こちらも同様に無駄であるようにも見えますが、実は違います。現に、最初の例題の正解のコードにおいて、A に相当するのは展開不能なプリミティブである `\let` ですが、それでも `\expandafter` は有意義でした。

```

% Aは'\let'であり展開不能
\verb{\expandafter}\let\xTargetCs\xResult

```

この `\expandafter` がなぜ意味をもつかというと、それは「一旦 `\let` が実行されると、`let` 文が完成するまで展開が抑止される」という性質があるからです。

[`\expandafter`がない場合¹¹]

```

\let\xTargetCs\xResult
→ <\let> \xTargetCs\xResult (let 文開始)
→ <\let\xTargetCs> \xResult (展開されない)

```

[`\expandafter`がある場合]

¹¹ この図の中の「→」は（「⇒」とは異なり）「実行の1ステップ」を表します。また <> は「既に実行されてバッファ上にはないトークン」を表します。「実行」とは何か、についてはあまり深く考えずに直感的に把握しましょう。（えっ）


```

\expandafter\let\xTargetCs\xResult
  [\xTargetCs ⇒ \xName] (展開される)
⇒ \let\xName\xResult
→ <\let> \xName\xResult    (let 文開始)
→ <\let\xName> \xResult

```

このように、構文上で展開抑止が起こる箇所では「Aが展開不能」であっても `\expandafter` は必ずしも無駄にならないのです¹²。

プリミティブで `\expandafter` する件

「展開不能なトークン」に関してよくある誤解は「プリミティブは展開不能である」というものです。確かにプリミティブの多く (`\let` 等) は展開不能ですが、実際には展開可能なプリミティブ (`\the` 等) もあります。例えば、先ほどの `\everypar` の例では `\the\everypar ⇒ \xFooBar` という展開を扱いました。

参考として、 \TeX 言語の展開可能なプリミティブのうち重要なものを列挙しておきます。

- 制御綴構成: `\csname`
- 値取得・文字列化: `\the \string \meaning \number \romannumeral`
- 条件分岐: 各種 if トークン (`\ifnum` 等) `\else \fi`
- 展開制御: `\noexpand \expandafter`

これらは展開可能であるため、`\expandafter` の定義における B の位置に来る場合があります。

例題: `\csname` で `\expandafter` する件

これらの展開可能プリミティブのうち、`\expandafter` と絡んでよく使われるのが `\csname` です。なので、これについて詳しく考えましょう。

【例題 3】例題 1 では、コピー先の制御綴を可変にするために、その制御綴そのものを入れたマクロ `\xTargetCs` を用意した。

```
\def\xTargetCs{\xName}
```

これを少し変えて、次のように「制御綴の名前」¹³ を内容に持たせる仕様にしたい。

```
\def\xTargetCsName{xName}% コピー先の制御綴の★名前★
```

では「`\xTargetCsName` の内容の文字列を名前とする制御綴に `\xResult` をコピー (`\let`) する」というコードを書け。

`\xTargetCsName` (の 内 容) の 名 前 を も つ 制 御 綴 を 作 る た め に `\csname\xTargetCsName\endcsname` と しま しょう。こ こ で 問 題 な の は

¹² もう一つ例を挙げます。先の「`\everypar` の代入文」のケース (`\everypar\expandafter{...}`) では、トークン列パラメタの代入文の文法規則として、`{}` を実行してから `}` を読むまでの間に展開抑止が起こります。従って、(展開不能な) `{}` の前の `\expandafter` が意味をもつわけです。

¹³ \TeX 言語において、制御綴 `\foo` の名前とは、先頭のエスケープ文字 (`\`) を除いた「`foo`」のことを指します。ちなみに、 \LaTeX において「命令の名前」という場合はエスケープ文字を含めた「`\foo`」を指すのが一般的です。

「これの一回展開はどうなるか」ということです。一回展開しただけで目的の制御綴 (`\xName`) になるのでしょうか？

答えはYesです。つまり `\csname~\endcsname` の一回展開は「~」の部分の名前をもつ単一の制御綴となります¹⁴。この際に「~」の部分は完全展開されるという規則になっています。

10

```
\csname\xTargetCsName\endcsname
  [\xTargetCsName — (完全展開) → x N a m e]
⇒ \xName
```

一回展開で十分なことが判れば、あとは簡単ですね。 `\let` の実行より先に `\csname` を一回展開すればよいので以下ようになります。

% 例題 3 の解答

```
\verb{\expandafter}\let\csname\xTargetCsName\endcsname\xResult
```

11

```
\expandafter\let\csname\xTargetCsName\endcsname\xResult
  [\csname\xTargetCsName\endcsname ⇒ \xName]
⇒ \let\xName\xResult
```

練習問題 (キホン編)

以上で、`\expandafter` の最も基本的な使い方 (単発の `\expandafter`) についての解説は終わりです。ここまでの内容をちゃんと理解できたかを確認するため、キホンの練習問題に挑戦してみましょう。

※この練習問題において、カテゴリコードの設定はLaTeXの `\makeatletter` の状態を仮定します。また、`my@` で始まる名前の制御綴 (例えば `\my@val`) は未定義であり自由に使ってよいものとします。

問題 1: `\expandafter` を展開する話

次のようなマクロが定義されているとする。

```
\def\gobble#1{}
\def\twice#1{#1#1}
```

この時、次のトークン列を一回展開した結果はどうなるか。

```
\expandafter\gobble\twice\gobble\twice\twice\gobble
```

¹⁴ 結果の制御綴はそれ以上展開されません。つまり、今の場合、`\xName` が展開可能であったとしても、`\csname...` の一回展開は飽くまで `\xName` となります。

問題 2: `\@namedef` を自作する話

LaTeXには「制御綴の代わりに制御綴の名前を指定してマクロを定義する」ための`\@namedef`という内部マクロが存在する。例えば、次の2つの文は等価な動作を行う。

```
% \@namedef{<名前>}<パラメタテキスト>{<置換テキスト>}
\@namedef{Hoge}{#1#2}\message{#1 and #2}

% ↑は↓と同等な動作をする
\def\Hoge{#1#2}\message{#1 and #2}
```

では、この機能をもつ`\@namedef`を自分で実装せよ。

問題 3: ドライブレターの有無を判定する話

次の機能をもつマクロ`\hasdrivespec`を実装せよ。

- `\hasdrivespec{<文字列>}` : 引数のトークン列を完全展開して得られる文字列について、その2文字目が:であるか（つまりWindowsのドライブレター付きの絶対パス名であるか）否かを判定し、その結果をスイッチ¹⁵`\if@tempswa`に返す。
 - ◇ スイッチ`\if@tempswa`はLaTeXでは予め定義されている。
 - ◇ 引数は「完全展開すると「普通の文字列」（カテゴリコード11か12の文字トークンの列）になる」ことを仮定してよい。

以下に`\hasdrivespec`の使用例を示す。

```
\def\xOneDir{C:/tmp/tex}
\def\xOneFile{advent.txt}
\def\xOnePath{\xOneDir/\xOneFile}

\hasdrivespec{D:/fonts}\if@tempswa Yes\else No\fi
%→ "Yes"と出力
\hasdrivespec{\xOneFile}\if@tempswa Yes\else No\fi
%→ "No"と出力
\hasdrivespec{\xOnePath}\if@tempswa Yes\else No\fi
%→ "Yes"と出力
```

問題 4: 制御綴を得る話

次の機能をもつマクロ`\makecs`を実装せよ。

- `\makecs\制御綴 A{<名前>}` : `\制御綴 A`を、内容が「名前が<名前>である制御綴」であるマクロとして定義する。
 - ◇ 引数は「完全展開すると文字トークンの列になる」ことを仮定してよい。

以下に`\makecs`の使用例を示す。

```
\makecs\xTest{space}
% \xTest \Longrightarrow \space となる
\def\&{and}
\makecs\xTest{exp\&after}
% \xTest \Longrightarrow \verb{\expandafter} となる
```

¹⁵ `\newif`により作成される`\if`トークンのことをスイッチ (switch) といいます。

問題 5：一回展開を調べる話

e-TeX 拡張をもつ TeX エンジン¹⁶は `\showtokens` というプリミティブを持つ。`\showtokens{<トークン列>}` を実行すると、`\show` と同様の情報表示の様式で、引数のトークン列が（展開されずに）端末にそのまま表示される。

```
% latex の対話モード
*\showtokens{\foo\bar\verb{\expandafter}} %← '*' はプロンプト
> \foo \bar \verb{\expandafter} .          %← 引数がそのまま出る
<*> \showtokens{\foo\bar\verb{\expandafter}}

?                                           %← 入力待ちになる
```

では、この `\showtokens` プリミティブを用いて「与えられたトークン列の一回展開がどうなるか」を調べる手順を構成せよ。そしてその手順を利用して問題 1 に対する自分の解答が正しいことを確認せよ。

まとめ

`\expandafter` は単なる TeX のプリミティブです。コワくありません！

「`\expandafter` のガイドライン」を活用して思う存分 `\expandafter` しましょう！

- A B...のようなトークン列があり、このままでは A が実行（展開）される。
- しかし、A の実行（展開）の前に B...を展開したい。
- その場合、A B...の直前に `\expandafter` を置けばよい。

`\expandafter` のホンキが見たい人はこちらへ——徹底解説！ `\expandafter` 活用術（ホンキ編）

¹⁶ 今時の LaTeX は全て、e-TeX 拡張をもつエンジンの上で動作しています。

徹底解説！

`\expandafter` 活用術（ホンキ編）

本記事は「徹底解説！`\expandafter` 活用術（キホン編）」の続きにあたります。「キホン編」では単発の `\expandafter` の使い方を学びましたが、この「ホンキ編」では `\expandafter` を自在に使いこなすのに不可欠な「`\expandafter` の鎖」について解説します。めざせ `\expandafter` マスター！

※「キホン編」で用いた表記や用語をこの記事でも踏襲します。

`\expandafter` の「鎖則」

「キホン編」の最後で「展開可能プリミティブの一覧」を載せましたが、その中には他ならぬ `\expandafter` が含まれています。`\expandafter` が展開可能なのはある意味当然で、なぜなら最初に挙げた「`\expandafter` の定義」は「`\expandafter` プリミティブの展開の規則」そのものだからです。

`\expandafter` で `\expandafter` する件

それでは、`\expandafter` の定義の **B** の位置に `\expandafter` がある場合、つまり「`\expandafter A \expandafter`」を展開するとどうなるか、考えてみましょう。`\expandafter` がある場合、その先に最低 2つのトークンがあるはずですので、次の形を考えます。

1

```
\expandafter A1\expandafter A2 B...  
ただし B...⇒B'...とする
```

「定義」の中の「**B...**」に相当する部分がここでは `\expandafter A2 B...` なので、いつもの図式で考えると以下のようなになるでしょう。

2

```
\expandafter A1\expandafter A2 B...  
  [\expandafter A2 B...⇒??]  
⇒A1 ???
```

?? の部分は何でしょう。これは `\expandafter` の単純な展開です。

3

```
\expandafter A2 B...  
  [B...⇒B'...]  
⇒A2 B'...
```

この結果をそのまま当てはめると、以下ようになります。

4

```
\expandafter A1\expandafter A2 B...
[\expandafter A2 B...
[B...⇒B'...]]
⇒A2 B'...]]
⇒A1 A2 B'...
```

つまり、 A_1 や A_2 は変化せずに B が展開される、という結果になりました。

もっと `\expandafter` を `\expandafter` してみる

では、ここで B がまた `\expandafter` だったらどうなるでしょうか。先と同様に考えると、結果は以下ようになります。

5

```
\expandafter A1\expandafter A2\expandafter A3 B...
[\expandafter A2\expandafter A3 B...
[\expandafter A3 B...
[B...⇒B'...]]
⇒A3 B'...]]
⇒A2 A3 B'...]]
⇒A1 A2 A3 B'...
```

最終的な結果だけ見ると、以下の式が成り立ちます。

- $\text{\expandafter } A_1 \text{\expandafter } A_2 B \dots \Rightarrow A_1 A_2 B' \dots$
- $\text{\expandafter } A_1 \text{\expandafter } A_2 \text{\expandafter } A_3 B \dots \Rightarrow A_1 A_2 A_3 B' \dots$

素敵な表記のおやくそく

さて、もっと先に進みたいわけですが、そうすると、ただでも長い `\expandafter` という制御綴が大量に並ぶことになって、煩わしいことこの上ありません。そこで次のような、チョット素敵な表記規則を設けましょう。

- `\expandafter` の略記として \blacksquare と書く。

この表記法を取り入れると、先の展開規則は次のように書けます。

- $\blacksquare A_1 \blacksquare A_2 B \dots \Rightarrow A_1 A_2 B' \dots$
- $\blacksquare A_1 \blacksquare A_2 \blacksquare A_3 B \dots \Rightarrow A_1 A_2 A_3 B' \dots$

チョット素敵になりましたね！¹⁷

「`\expandafter` の鎖」の法則

state: unknown

本題に戻りましょう。先ほど行った、「 B を `\expandafter A1 B` に置き換えて、全体の展開を考える」という操作は何度でも繰り返すことができます。この方法で

¹⁷ なんだ、結局奇をてらっているじゃん……。

- $\frac{}{\text{name:} A_1} \frac{}{\text{name:} A_2} B \dots \Rightarrow A_1 A_2 B' \dots$

- これを一般化して¹⁸ 得られるのが、次に示す「\expandafter の鎖則」です。

「\expandafter が連なってできた鎖がトークン列に「絡まっている」」ように見えるため「鎖則 (chain law)」の名前がついています。

- A, B のようなトークン列があり、この中では A が実行される

(上記の例の場合、\xBarHook の一回展開が \sscsnewman[muff]or-red\relax\x

まずは「\xPerflock の後に \xMyllock を追加」したもので \xPerflock を更完善する。


```
% 未完成 あくまでも原型
\def\xParHook{\xParHook\xMyHook}
```

もちろんこのままでは \xParHook が無限ループになってしまいます。ここで必要なのは「\def が実行される前に（後ろの） \xParHook を展開する」ことです。この状況を「ガイドライン」を当てはめてみましょう。

- \def...\xParHook...というトークン列があり、このままでは \def が実行される。
- しかし、\def の実行の前に \xParHook を展開したい。
- その場合、\def...の部分にある全てのトークンの直前に \expandafter を置けばよい。

今の場合、「\def...の部分」のトークン列とは「\def\xParHook{\xParHook\xMyHook}」です。従って、この部分に \expandafter の鎖を絡ませればよいわけです。つまり以下のようになります。

```
% 例題 4 の解答
\expandafter\def\expandafter\xParHook\expandafter{\%
\xParHook\xMyHook}
```

例題：キレイキレイにする話

【例題 5】制御綴の列を内容にもつマクロ \xGarbageList がある。

```
% "消したい" 制御綴を入れておく
\def\xGarbageList{\rm\sfftt\bf\it\sl\sc}
```

このとき、LaTeX の内部命令 \@tfor を利用して「\xGarbageList の内容に含まれる各々の制御綴の定義を消去する（未定義に戻す）」コードを書け。

先ほどと同様に、まずは原型となるコードを作って、それから \expandafter を加えていきます。

```
% 未完成
\@tfor\x:=\xGarbageList\do{%
\let\x\@undefined}% \@undefined は未定義の制御綴
```

まずループの中ですが、このままでは \x 自体に代入されてしまいます。「\let の実行の前に \x を展開したい」ので \let の前に \expandafter を置きます。

```
\expandafter\let\x\@undefined
```

次に \xGarbaseList の展開についてですが、「\@tfor の実行の前に \xGarbaseList を展開したい」ということなので、ガイドラインに従うと、「\@tfor\x:=」の部分に \expandafter の鎖を絡ませればよいわけです。

```
% 例題 5 の解答
\expandafter\@tfor\expandafter\x\expandafter:\expandafter=
\xGarbageList\do{\expandafter\let\x\@undefined}
```

`\expandafter` の鎖に潜む罠

「`\expandafter` の鎖」は応用範囲がとても広くて便利なのですが、重大な欠点があります。それは「コード中の鎖が「絡んでいる」部分の可読性が著しく損なわれる」ということです。例えば、先ほどの例題の解答のコードを改めて見返してください。

「`\@tfor\x:=...\do`」というループ構造を示す外見が、ほとんど視認できなくなってしまっています。後で「絡んでいる」部分のコードを改修しようとしても、その作業は困難を極めることになるでしょう。

この場合、鎖が「絡んでいる」部分の「`\@tfor\x:=`」を一度マクロにすると、単発の `\expandafter` で済ませられます。

% 例題 5 の解答

```
\def\xTforXIn{\@tfor\x:=}  
\expandafter\xTforXIn\xGarbageList\do{%  
\expandafter\let\x\@undefined}
```

まだまだ解り難さは残っていますが、少なくとも実際に実行されるコードが「見えて」いるため、将来の改修に対応することができるでしょう。

`\expandafter` の長連の規準

このように、「`\expandafter` の長い鎖」の使用は厳に慎まれるべきです。私自身は以下のような「`\expandafter` の長連の規準」を推奨しているので参考にしてください。

- `\expandafter` の3重を超える鎖が発生した場合は、それを回避する策をホンキで考えよう。
- `\expandafter` の5重を超える鎖は絶対に絶対に絶対に回避しよう。

この規準はかなり厳しいのは確かですが、実際にこのくらいに「長連の回避」を考える機会が得られないと、「回避するコツ」がなかなか身に付かないものです。

`\expandafter` の「ベキ乗則」

これまでの話では、`\expandafter` を（単発でも鎖でも）使うと、後ろにあるトークンが一回展開できる、ということでした。それでは、後ろにあるトークンを「何回も」展開したい場合はどうすればいいのでしょうか。考えましょう。

後ろを2回展開したい話

```
\def\csB{\csBi}  
\def\csBi{\csBii}  
\def\csBii{\csBiii}  
\def\csBiii{\csBiv}  
% つまり \csB ⇒ \csBi ⇒ \csBii ⇒ \csBiii ⇒ \csBiv
```

例えば、こういう定義があったとして、

「`\csA\csB`」の後ろにある `\csB` を2回展開したい（つまり「`\csA\csBii`」に変えたい）

という状況を考えます。どのように `\expandafter` を置けばよいのでしょうか。

「\expandafter では1回しか展開できない」という原則は変えられないので、これを実現するには「展開自体を2回にする」必要があることは確かです。

【概念図】

```

\csA\csB ...①
⇒ \csA\csBi ...①
⇒ \csA\csBii ...②

```

まずは「①⇒②になるように①に \expandafter を加える」という問題を考えましょう。 \csBi の一回展開が \csBii なので、これは単純な単発の \expandafter で解決できます。

【①⇒②は完成】

```

\csA\csB ...①
⇒ \csA\csBi ...①
[\csBi ⇒ \csBii]
⇒ \csA\csBii ...②

```

ここで①も①と同じ形になるように前に \expandafter を置きました。この状態で「①⇒②になるように①に \expandafter を加える」という問題を考えます。よく見ると、これは「\expandafter の鎖のガイドライン」が当てはめられることに気づきます。

- \csA\csB というトークン列があり、このままでは \csB が実行（展開）される。
- しかし、 \csB の実行の前に \csB (⇒ \csBi) を展開したい。
- その場合、 \csA の部分にある全てのトークンの直前に \csB を置けばよい。

つまり、 \csA ⇒ \csB、 \csA ⇒ \csB と置き換える。
 従って結果は \csA \csB \csA となる。

つまり、完成形は以下ようになります。結果的に、先頭に3個の \csB を置けばよいことになります。

【2回展開の完成形】

```

\csA\csB ...①
[\csB ⇒ \csBi] (鎖則)
⇒ \csA\csBi ...①
[\csBi ⇒ \csBii] (単発)
⇒ \csA\csBii ...②

```

後ろを3回展開したい話

2回展開ができたので、次は3回展開を考えてみます。

「\csA\csB」の後ろにある \csB を3回展開したい（つまり「\csA\csBiii」に変えたい）3回展開なので、全体のトークン列も3回展開する必要があります。先ほどと同様に「後ろから順に」考えてみましょう。

【概念図】

```

\csA\csB ...①
⇒ \csA\csBi ...②
⇒ \csA\csBii ...③
⇒ \csA\csBiii ...④

```

②⇒③に単発の \expandafter を適用します。

【②⇒③は完成】

```

\csA\csB ...①
⇒ \csA\csBi ...②
⇒ \csA\csBii ...③
[\csBii⇒\csBiii] (単発)
⇒ \csA\csBiii ...④

```

①⇒②に \expandafter の鎖を適用します。

【①⇒②⇒③は完成】

```

\csA\csB ...①
⇒ \csA\csBi ...②
[\csBi⇒\csBii] (鎖則)
⇒ \csA\csBii ...③
[\csBii⇒\csBiii] (単発)
⇒ \csA\csBiii ...④

```

最後に④⇒①の部分ですが、これも鎖則が適用できる形になっていることが判るでしょう。つまり、`\csA`に`\expandafter`の鎖（もう「`\`」の鎖）でいいよね）を絡ませればよいわけです。

ここで少し一般的に考えてみます。「`\`」が n 個並んだ後にトークン X がある」というトークン列に対して「`\`」の鎖の絡ませる（列に含まれる各々のトークンの前に「`\`」を置く）とどうなるでしょうか。「`\`」が n 個の部分は個数が倍に増えて「`\`」が $2n$ 個になり、さらに X が`\`に変わるので、結局 X の前に $2n+1$ 個の「`\`」がある恰好になります。この結果を「`\expandafter` 倍増の規則」（いや「`\` 倍増の規則」かな？）と呼ぶことにしましょう。

7

```

\ x n X state: unknown
name:
に \ の鎖を絡ませると
\ x (2n+1) X
name:
になる
outCollection/sushi_3d.png

```

19 「`\`」が n 個並んだトークン列」を「`\`」と表記します。
`sushi_3d`
`file: outCollection/sushi_3d.png`
`state: stateknown`
`unknown`

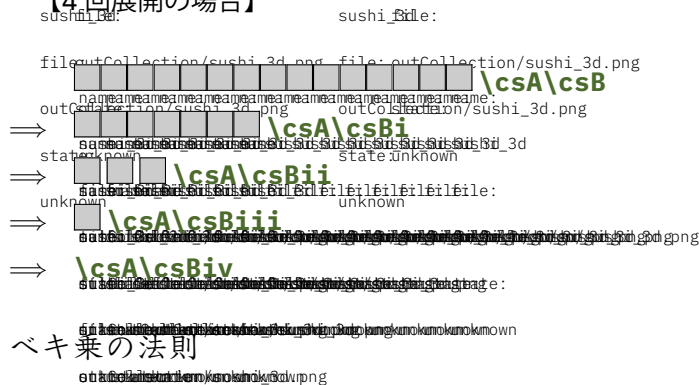
この規則に従うと、 $\square \times 3 \backslash \text{csA}$ に鎖を絡ませた結果は $\square \times 7 \backslash \text{csA}$ となります。従って、結果的に、先頭に7個の \square を置けばよいことになります。

【3回展開の完成図】



さらに一步進めて、4回展開はどうなるでしょうか。これまでの手順と同様に考えると、結局 $\square \times 7 \backslash \text{csA}$ の部分に再度 \square の鎖を絡ませれば済むことが判るでしょう。そして「 \square 倍増の規則」により、 \square の個数は $2 \times 7 + 1$ で15個に増えます。

【4回展開の場合】



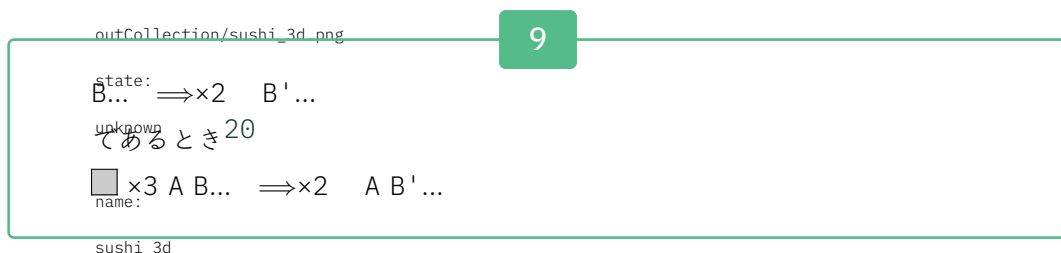
ベキ乗の法則

ここで、今までの考察の結果を、一般的なトークン列に対する規則としてまとめてみましょう。

【1回展開】(\expandafter の定義)



【2回展開】




【3回展開】

²⁰ 「X を n 回展開すると Y になる」という言明を「 $X \Rightarrow \times n Y$ 」と表記します。

B... $\Rightarrow \times 3$ B'...

であるとき

 $\times 7$ A B... $\Rightarrow \times 3$ A B'...


sushi_3d

【4 回展開】


outCollection/sushi_3d.png

state:
B... $\Rightarrow \times 4$ B'...

unknown
であるとき

 $\times 15$ A B... $\Rightarrow \times 4$ A B'...

sushi_3d

これを見ると、先頭に置くべき  の個数は

1 \rightarrow 3 \rightarrow 7 \rightarrow 15 \rightarrow

のように増えています²¹。この数列の第 n 項の値は $2^n - 1$ で求められます。ここから

「 n 回展開」に関する一般的規則を導き出すことができます。


【 n 回展開】

state:

unknown

B... $\Rightarrow \times n$ B'...

であるとき

 $\times (2^n - 1)$ A B... $\Rightarrow \times n$ A B'...

sushi_3d

$\backslash\text{expandafter}$ の個数が 2 のべきに従って増えていく様子から、この規則は「 $\backslash\text{expandafter}$ のべき乗則 (power law)」と呼ばれることがあります。

べき乗則よりも大事なこと

この「べき乗則」は理論的には非常に面白い結果なのですが、しかし私はべき乗則は覚える必要はないと考えています。なぜかという、べき乗則を単純に適用できる状況は実用上はそう多くはないと考えているからです。実際の TeX 言語のプログラミングで「複数回展開する」状況はもっと多様です。

- 複数回展開したいトークンがずっと後ろにある（つまり鎖則と複合する場合）
- そもそも先に展開したいトークンが複数個ある
 - ◇ しかも各々のトークンで必要な展開回数が異なる

従って、「複数回展開する」状況に対応できるようになるために必要なのは、べき乗則を定理として覚えることではなく「それを導出する方法」を習得することだといえます。具体的には、以下のような要素を身につける必要があります。

- 鎖則を自由に使いこなす
- 展開過程を「後ろから順に」構築する
- 「 $\backslash\text{expandafter}$ 倍増の規則」

²¹ 先ほどの「 n 回展開」の規則により、この数列は漸化式 $a_n = 2a_{n-1} + 1$ に従うことになります。

sushi_3d
file:
outCollection/sushi_3d.png
state:
unknown

そして、展開制御を上手に行う上で大事なコツはそもそも「後ろのトークンを何度も展開する状況」を作らないということです。ベキ乗則から判るように、後ろのトークンを複数回展開しようと試みると、`\expandafter` が文字通り指数爆発してしまいます。結果的に、それほど複雑でない状況であってもコードが「`\expandafter` まみれ」になって全く読めなくなる、という悲惨な状況が簡単に発生します。

`\expandafter` の高度な活用法を学習する際には、ぜひとも、他の展開制御の手法（`\edef` による完全展開、など）も一緒に習得して、「`\expandafter` が爆発しないように上手く展開制御する」ことを心がけましょう。

めざせ展開制御マスター！

練習問題（ホンキ編）

※ここの練習問題において、カテゴリコードの設定はLaTeXの `\makeatletter` の状態を仮定します。また、`my@` で始まる名前の制御綴（例えば `\my@val`）は未定義であり自由に使ってよいものとします。

問題 6：`\expandafter` の群れを展開する話

次のようなマクロが定義されているとする。

```
\def\gobble#1{}
\def\twice#1{#1#1}
```

この時、次のトークン列を一回展開した結果はどうなるか。

```
\expandafter\expandafter\expandafter\twice\expandafter
\twice\expandafter{\gobble\expandafter}{\gobble}
```

問題 7: 名前で `\let` する話

etoolbox パッケージでは、`\let` について「制御綴の代わりにその名前を指定する」変種として以下の命令を提供している。

- `\cslet<名前 A>\制御綴 B`：「制御綴 B を「名前が <名前 A> の制御綴」にコピーする。
- `\letcs\制御綴 A<名前 B>`：「名前が <名前 B> の制御綴」を `\制御綴 A` にコピーする。
- `\csletcs<名前 A>{<名前 B>}`：「名前が <名前 B> の制御綴」を「名前が <名前 A> の制御綴」にコピーする。

これらの命令を自分で実装せよ。ただし制御綴の名前の引数は「完全展開すると文字トークンの列になる」ことを仮定してよい。

```
% 以下の4つの文は全て等価になるべき
\let\foo\bar
\cslet{foo}\bar
\letcs\foo{bar}
\csletcs{foo}{bar}
```


問題 8: マクロの前後にナニカを追加する話

次の機能をもつマクロ `\enclose` を実装せよ。

`\enclose\制御綴 A{<トークン列 1>}{<トークン列 2>}` : 引数無しのマクロ `\制御綴 A` について、その内容を、「前に `<トークン列 1>`、後ろに `<トークン列 2>` を追加したトークン列」に置き換える。`\制御綴 A` は引数無しのマクロであると仮定してよい。以下に `\enclose` の使用例を示す。

```
\def\xTest{\ARE\LaTeX}  
\enclose\xTest{\ARE\TeX}{\ARE{TikZ}}  
% \xTest \Longrightarrow \ARE\TeX\ARE\LaTeX\ARE{TikZ}
```

まとめ

`\expandafter` と 5 回唱えたら願いが叶った。

—露伴 (@Rohan_zzz) 2016 年 8 月 17 日

皆さんも `\expandafter` で幸せになりましょう！

