スペースにまつわるエトセトラ

私立文系 初級ユーザー

2022 年 12 月 5 日

TEX & LATEX Advent Calendar 2022

あらかじめのお詫び

この文書のテーマは、ざっくりと言って、スペースにまつわるお話です。今年の 6 月に TEX Forum で空白トークンの扱いに関する質問があったり、10 月に界隈で "`\xobeylines`" というのが話題になったりしていたので、そのときどきに The TEXbook の関係していそうなところを拾い読みした際のメモを、Advent Calendar 用にまとめ直したものです。

私は私立文系出身の万年初級ユーザーでして、私の知識にはいろいろと間違いや思い違いが含まれている可能性が高いです。なので、この文書の内容につきましても、「本当にそうなの？」と疑いながらお読みいただいたほうがいいです（すいません）。また、中級以上の方にとっては、当たり前のことしか書いてないと思います（すいません）。

最初に少し一般的な準備をして（「はじめに」）、それから、本題に移ろうかと思います（「1 スペーストークン」、「2 パターンマッチ」、「3 カテゴリーコード」）。そして、最後には「オマケ」を付けています。

はじめに

本論のほうで "`^^M`" という表記ですとか、「implicit なスペース」というような言葉が出てきますので、まずは、そのあたりについて簡単に確認をしておきたいと思います。

ASCII コードの入力法

The TEXbook の "Appendix C: Character Codes"（367 ページ）には、次のような ASCII (American Standard Code for Information Interchange) のコード表が載っています：

	′0	′1	′2	′3	′4	′5	′6	′7	
′00x	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	″0x
′01x	BS	HT	LF	VT	FF	CR	SO	SI	
′02x	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	″1x
′03x	CAN	EM	SUB	ESC	FS	GS	RS	US	
′04x	SP	!	"	#	$	%	&	’	″2x
′05x	(	)	*	+	,	-	.	/	
′06x	0	1	2	3	4	5	6	7	″3x
′07x	8	9	:	;	<	=	>	?	
′10x	@	A	B	C	D	E	F	G	″4x
′11x	H	I	J	K	L	M	N	O	
′12x	P	Q	R	S	T	U	V	W	″5x
′13x	X	Y	Z	[	\	]	^	_	
′14x	‘	a	b	c	d	e	f	g	″6x
′15x	h	i	j	k	l	m	n	o	
′16x	p	q	r	s	t	u	v	w	″7x
′17x	x	y	z	{	|	}	~	DEL	
	″8	″9	″A	″B	″C	″D	″E	″F	

その説明として、次のようにあります：

> In the ASCII scheme, codes *’000* through *’040* and code *’177* are assigned to special functions; […] The other 94 codes are assigned to visible symbols.

つまり、ASCII では、128 個のスロットのうち、’041 〜 ’176 の 94 スロットは、目に見える文字のコードで、他方’000 〜 ’040 と’177 の計 34 スロットは、目に見える文字ではない制御コードということのようです。

目に見える文字のコードについてならば、キーボードの当該キーを押下すれば入力できますが、それ以外の目に見えない制御コードについては、どうやって表現したり入力したりすればいいのでしょうか（"CR (⟨*return*⟩)"、"HT (⟨*tab*⟩)"、"ESC (⟨*escape*⟩)"、"DEL (⟨*delete*⟩)" くらいなら、手元のキーボードにも一応ありますけれど）。

で、その方法が、次の 368 ページにあります：

	′0	′1	′2	′3	′4	′5	′6	′7	
′00x	^^@	^^A	^^B	^^C	^^D	^^E	^^F	^^G	″0x
′01x	^^H	^^I	^^J	^^K	^^L	^^M	^^N	^^O	
′02x	^^P	^^Q	^^R	^^S	^^T	^^U	^^V	^^W	″1x
′03x	^^X	^^Y	^^Z	^^[	^^\	^^]	^^^	^^_	
′04x	␣	!	"	#	$	%	&	’	″2x
′05x	(	)	*	+	,	-	.	/	
′06x	0	1	2	3	4	5	6	7	″3x
′07x	8	9	:	;	<	=	>	?	
′10x	@	A	B	C	D	E	F	G	″4x
′11x	H	I	J	K	L	M	N	O	
′12x	P	Q	R	S	T	U	V	W	″5x
′13x	X	Y	Z	[	\	]	^	_	
′14x	‘	a	b	c	d	e	f	g	″6x
′15x	h	i	j	k	l	m	n	o	
′16x	p	q	r	s	t	u	v	w	″7x
′17x	x	y	z	{	|	}	~	^^?	
	″8	″9	″A	″B	″C	″D	″E	″F	

目に見える文字については 367 ページの表と同じですが、制御コードが示されていた（スペースを除いた）33 スロットについては：

> "^" を 2 個[1] + 当該スロットの数値を 64（8 進表記だと’100）ズラしたコードの文字

が書かれています。この表記法を使えば、目に見える文字を使って、目に見えない制御コードを入力することができるみたいです

[1] "Here '^^' doesn't necessarily mean two circumflex characters; it means two identical characters whose current `\catcode` is 7." (p. 368)

（尤も、だからと言ってこれら 33 個の制御コードをホントに全部入力したいのかと問われたら、それはそれで答えに窮するのですけれど）。

“Appendix B”（343 ページ）には、次のようにあります：

> When INITEX begins, category 12 (other) has been assigned to all 256 possible characters, except that the 52 letters A...Z and a...z are category 11 (letter), and a few other assignments equivalent to the following have been made:
>
> ```
> \catcode `\\ =0  \catcode`\  =10  \catcode `\% =14
> \catcode`\^^@=9  \catcode`\^^M=5  \catcode`\^^?=15
> ```
>
> Thus ‘\’ is already an escape character, ‘␣’ is a space, and ‘%’ is available for comments on the first line of the file; ASCII ⟨*null*⟩ is ignored, ASCII ⟨*return*⟩ is an end-of-line character, and ASCII ⟨*delete*⟩ is invalid.

つまり、INITEX が、カテゴリーコードの 0、5、9、10、11、12、14、15 を割り当てていて、制御コードの “^^@” や “^^?” はその段階で既に、“ignored” や “invalid” になってるらしいです。

なお、残りのカテゴリーコードは、plain.tex が[2]冒頭で割り当てています（343 ページ．ここでは段組みの幅の関係でコメント部分で改行をしています）：

```
\catcode`\{=1
   % left brace is begin-group character
\catcode`\}=2
   % right brace is end-group character
\catcode`\$=3
   % dollar sign is math shift
\catcode`\&=4
   % ampersand is alignment tab
\catcode`\#=6
   % hash mark is macro parameter character
\catcode`\^=7 \catcode`\^^K=7
   % circumflex and uparrow for superscripts
\catcode`\_=8 \catcode`\^^A=8
   % underline and downarrow for subscripts
\catcode`\^^I=10
   % ASCII tab is treated as a blank space
\chardef\active=13 \catcode`\~=\active
   % tilde is active
\catcode`\^^L=\active \outer\def^^L{\par}
   % ASCII form-feed is \outer\par
```

制御コードに着目しますと、ここでは “^^K”、“^^A”、“^^I”、“^^L” にカテゴリーコードが割り当てられているのが判ります。“^^I” が ASCII ⟨*tab*⟩ で、“^^L” が ASCII ⟨*form-feed*⟩ なのはいいとして、“^^K” が ⟨*uparrow*⟩ で “^^A” が ⟨*downarrow*⟩ というのはどういうことでしょうか。

これについては、369 ページに MIT 版の “extended ASCII code” という表が載っていて、それを見ると判ります：

	′0	′1	′2	′3	′4	′5	′6	′7	
′00x	·	↓	α	β	∧	¬	∈	π	″0x
′01x	λ	γ	δ	↑	±	⊕	∞	∂	
′02x	⊂	⊃	∩	∪	∀	∃	⊗	↮	″1x
′03x	←	→	≠	◇	≤	≥	≡	∨	
′04x		!	"	#	$	%	&	’	″2x
′05x	(	)	*	+	,	-	.	/	
′06x	0	1	2	3	4	5	6	7	″3x
′07x	8	9	:	;	<	=	>	?	
′10x	@	A	B	C	D	E	F	G	″4x
′11x	H	I	J	K	L	M	N	O	
	″8	″9	″A	″B	″C	″D	″E	″F	

	′0	′1	′2	′3	′4	′5	′6	′7	
′12x	P	Q	R	S	T	U	V	W	″5x
′13x	X	Y	Z	[	\	]	^	_	
′14x	‘	a	b	c	d	e	f	g	″6x
′15x	h	i	j	k	l	m	n	o	
′16x	p	q	r	s	t	u	v	w	″7x
′17x	x	y	z	{	|	}	~	∫	
	″8	″9	″A	″B	″C	″D	″E	″F	

この “extended ASCII code” を採用しているキーボードが今でもどこかで使われているのかどうかは分からないのですけれど、それでも、2021 年の plain.tex（\fmtversion{3.1415926535}）でも、“^^K” や “^^A” へのカテゴリーコードの割り当ては残ったままです（他にも、“\let\^^_=\v”、“\let\^^S=\u”、“\let\^^D=\^” といったアクセントの割り当てなんかも今でも残っています）。

ちなみに、MIT 版のほうが Stanford 版よりも “slightly better” だとして現在の The TEXbook に載っているのは MIT 版のコード表ですが、Knuth 先生が TEX78 を開発した当初は、以下のような SUAI (Stanford University Artificial Intelligence Laboratory) code というのが使われてたらしいです（’27 は実際は “↔”）：

	′0	′1	′2	′3	′4	′5	′6	′7	
′00x	NUL	↓	α	β	∧	¬	∈	π	″0x
′01x	λ	TAB	LF	VT	FF	CR	∞	∂	
′02x	⊂	⊃	∩	∪	∀	∃	⊗	↮	″1x
′03x	_	→	~	≠	≤	≥	≡	∨	
′04x	SP	!	"	#	$	%	&	’	″2x
′05x	(	)	*	+	,	-	.	/	
′06x	0	1	2	3	4	5	6	7	″3x
′07x	8	9	:	;	<	=	>	?	
′10x	@	A	B	C	D	E	F	G	″4x
′11x	H	I	J	K	L	M	N	O	
′12x	P	Q	R	S	T	U	V	W	″5x
′13x	X	Y	Z	[	\	]	↑	←	
′14x	‘	a	b	c	d	e	f	g	″6x
′15x	h	i	j	k	l	m	n	o	
′16x	p	q	r	s	t	u	v	w	″7x
′17x	x	y	z	{	|	ALT	}	BS	
	″8	″9	″A	″B	″C	″D	″E	″F	

以上長々とコード表を眺めてみましたが、後々出て来るのは “^^M” だけです。一応試しに：

```
\nopagenumbers
foo^^Mbar
baz
\bye
```

というファイルを作って pdfTEX で処理してみますと、結果は：

> foo baz

となることを、ひとまず確認しておきます。

implicit な文字トークン

The TEXbook のスペース関係のページを拾い読みしていますと、“implicit space” とか “implicit characters”、“implicit character tokens” といった表現が出てきます（“implicit“の的確な訳語を捻り出せないので、英語のママで進めます…）。これは何なのでしょう。

[2] LATEX 2ε だと、ltplain.dtx で、同様の説明がなされていて、ほぼ同じ割り当てがなされています。

まず、普通の「トークン」についての説明は、“Chapter 7: How TEX Reads What You Type”（38 ページ）に、次のようにあります：

> A token is either (a) a single character with an attached category code, or (b) a control sequence. [...] The `hskip` doesn't get a subscript, because it represents a control sequence token instead of a character token.

即ち、トークンには、カテゴリーコードを付与された個々の文字である「文字トークン」と、あと「コントロールシーケンストークン」とがあって、コントロールシーケンスにはカテゴリーコードは付かない、と。ふむふむ。

そして、“implicit characters” については、ずっと後ろの “Chapter 24: Summary of Vertical Mode”（269 ページ）において、ブレースを例にして、以下のように説明されています：

> Control sequences sometimes masquerade as characters, if their meaning has been assigned by `\let` or `\futurelet`. For example, Appendix B says
>
> ```
> \let\bgroup={    \let\egroup=}
> ```
>
> and these commands make `\bgroup` and `\egroup` act somewhat like left and right curly braces. Such control sequences are called “implicit characters”; they are interpreted in the same way as characters, when TEX acts on them as commands, but not always when they appear in arguments to commands.

つまり、普通の一個ずつの文字が「explicit な文字トークン」であるのに対して、それを `\let` や `\futurelet` して作ったコントロールシーケンストークンのことを「implicit な文字トークン」と呼んでいるみたいですね。

続けて、スペースについても、同様の説明がされています：

> The quantity ⟨*space token*⟩, which was used in the syntax of ⟨*optional spaces*⟩ above, stands for an explicit or implicit space. In other words, it denotes either a character token of category 10, or a control sequence or active character whose current meaning has been made equal to such a token by `\let` or `\futurelet`.

なんでこういう「implicit な文字トークン」なんてものが必要なのかを素人なりに考えてみますと、カテゴリーコードが付与されている “$\{_1$” と “$\}_2$” はペアじゃないといけないですけれど、コントロールシーケンスにした “`\bgroup`” や “`\egroup`” なら、ペアにしなくてもエラーにならずに済みます。それに、スペースはそのままだと目に見えないので、それで、適当なコントロールシーケンスに `\let` して、目に見えるようにして扱いやすくしてるんでしょうかね。

* * *

以上で長ったらしい前座を終えて、ここからが、ようやく本論となります。以下、つらつらと The TEXbook のスペースに関連していそうなところを拾い読みしてみます。

1 「スペーストークン」って何をしてるの？

まずは、いかにも素人っぽい疑問なのですけれど、そもそも、「スペーストークン」って、何をしているのでしょうか。

ASCII の 32 番は確かに「スペース」ですけれど、OT1 の 32 番は “`/suppress`” で T1 の 32 番は “`/visualspace`” なので、入力コードがそのまま出力フォントのコードに対応しているわけではなさそうです。それに、文字じゃ伸び縮みできないですし。

スペースの読み飛ばしの規則については、The TEXbook の “Chapter 8: The Characters You Type”（46〜47 ページ）に、次のようにあります（もっと細々とあるのですが、主要な部分のみ抜粋）：

> ⟨*State*⟩
>
> State *N*　Beginning a new line;
> State *M*　Middle of a line;
> State *S*　Skipping blanks.
>
> ⟨*category 0*⟩
> If TEX sees an escape character (category 0) in any state, it scans the entire control sequence name as follows.
>
> - (a) If there are no more characters in the line, the name is empty (like `\csname\endcsname`).
> - Otherwise (b) if the next character is not of category 11 (letter), the name consists of that single symbol.
> - Otherwise (c) the name consists of all letters beginning with the current one and ending just before the first nonletter, or at the end of the line. This name becomes a control sequence token.
>
> TEX goes into state *S* in case (c), or in case (b) with respect to a character of category 10 (space); otherwise TEX goes into state *M*.
>
> ⟨*category 10*⟩
> If TEX sees a character of category 10 (space), the action depends on the current state.
>
> - If TEX is in state *N* or *S*, the character is simply passed by, and TEX remains in the same state.
> - Otherwise TEX is in state *M*; the character is converted to a token of category 10 whose character code is 32, and TEX enters state *S*.
>
> The character code in a space token is always 32.

うーむ。TEX は、入力テキストを順番に読み込んでいく際に、コントロールワードの後ろと、コントロールスペースの後ろだと「状態 S」になって、それに続くスペースはスキップされる、と。そして、「状態 M」のときに出遭った“カテゴリーコード 10 の文字”は、その ASCII コードが 32 番でカテゴリーコードが 10 の ⟨*space token*⟩ に変換されて、それから TEX はまた「状態 S」に移行するので、続く 2 個目以降のスペースはスキップされる、ということのようです。

スペースを読み飛ばすルールについては一応これで判ったことにして、それでは、TEX が読み込んでトークン化したこの ⟨*space token*⟩ というものが一体何をしてるのかを知りたいのですけれど、読み進めていっても、なかなか出て来ません。

“Chapter 12: Glue”（69〜83 ページ）においては、水平方向、垂直方向のボックス間のグルーについて扱われているので、いかにもこのあたりで説明されていそうです。でも、いくら探しても ⟨*space token*⟩ は出て来ません。なんか、いつの間にか “space factor” とか “`\sfcode`” の話になってしまっているのですが、⟨*space token*⟩ は一体どこに消えてしまったのでしょう？

仕方がないのでズルをして索引を見てみましたら、なんと、うんと最後のほうの Chaps. 24–26 で説明されていました：

> Chapter 24: Summary of Vertical Mode (p. 282)
>
> - ⟨*space token*⟩. Spaces have no effect in vertical modes.
>
> Chapter 26: Summary of Math Mode (p. 290)
>
> - ⟨*space token*⟩. Spaces have no effect in math modes.
>
> Chapter 25: Summary of Horizontal Mode (p. 285)
>
> - ⟨*space token*⟩. Spaces append glue to the current list; the exact amount of glue depends on `\spacefactor`, the current font, and the `\spaceskip` and `\xspaceskip` parameters, as described in Chapter 12.

な〜んだ、そういうことでしたか。それだったらこれを、8 章か 12 章にも書いておいてくださったらよかったのになぁ。

`\let` や `\futurelet` でスペースを捕捉する

スペースは、状況によっては読み飛ばされる一方で、行末の ASCII ⟨*return*⟩ はスペースに変換されるので “`%`” で抑制するのを忘れると意外なところにスペースが挿入されてしまっていたりして、いろいろと頭を悩まされますよね。

更には、“Chapter 20: Definitions (also called Macros)”（201 ページ）に：

> After you have said '\def\row#1#2{...}', you are allowed to put spaces between the arguments (e.g., '\row x n'), because TEX doesn't use single spaces as undelimited arguments.

と説明されていますように、スペースは、デリミタが付いてないと、マクロの引数にはなれません。

それで、"Appendix D"（376 ページ）で：

> Notice that '\def\stepthree#1{\stepone}' would not work here, because of TEX's rule that a ␣10 token is bypassed if it would otherwise be treated as an undelimited argument.

と言われているも、このことだと思います。もしもスペースが引数になれるのであれば、"\def\stepthree#1{\stepone}" と定義したら、\stepthree の後ろにスペースが続いている場合には \stepthree がそのスペースを食べてくれて、その後の処理を \stepone に渡せるのですけれど、デリミタの付いてないスペースは引数になれないのでそれが叶わないということでしょう。

"Appendix D" の 376〜377 ページではスペースの扱いの例が取り上げられていますが、ここでは、その中から "\futurenonspacelet" の定義について見てみます。

\futurenonspacelet は \futurelet と大体同じ動きをするのですけれど、スペースでないトークンが見つかるまでスペースを捨てる（呑み込む）ようになっています。

まず、\futurelet についての解説は、"Chapter 20: Definitions (also called Macros)"（207 ページ）にあります：

> TEX also allows the construction
>
> '\futurelet\cs⟨*token1*⟩⟨*token2*⟩',
>
> which has the effect of
>
> '\let\cs = ⟨*token2*⟩⟨*token1*⟩⟨*token2*⟩'.

つまり、\cs には ⟨*token2*⟩ が \let されているので、TEX は \cs を読み込んだ時点で、⟨*token2*⟩ が何なのかを判っていることになります。そこで、続く ⟨*token1*⟩ の展開過程で \cs (= ⟨*token2*⟩) の意味をチェックすれば、⟨*token2*⟩ が何であるのかによって処理を分岐することができるというわけです。

ただ、ここでちょっと困るのは、⟨*token2*⟩ がスペースであって且つそのスペースが読み飛ばされない場合には、\cs にはスペースが \let されてしまうことです。例えば：

```
\nopagenumbers
\let\*\relax % just a dummy as ``token1''
\futurelet\lettoken\* [option]\par
lettoken = \tt\meaning\lettoken
\bye
```

というファイルを pdfTEX で処理すると：

> [option]
> lettoken = blank space

となります。

この例では "\lettoken" が \cs に当たり、"*" が ⟨*token1*⟩ で（且つ "*" はコントロールシンボルなので後ろのスペースは読み飛ばされないので）、続くスペースが ⟨*token2*⟩ に相当してしまっています。そのため、\meaning の結果から判りますように、\lettoken にはスペースが \let されています。邪魔なスペースは捨てて "[" を見つけてくれれば、\lettoken に "[" が \let されるので、\lettoken の意味をチェックするとオプションがあるかどうかの判定が出来るのですけれど、これだとうまくいきません。

そこで登場するのが "Appendix D"（376 ページ）の "\futurenonspacelet" ということになります：

```
\def\futurenonspacelet#1{\def\cs{#1}%
  \afterassignment\stepone\let\nexttoken= }
\def\\{\let\stoken= } \\ % now \stoken is a space token
\def\stepone{\expandafter\futurelet\cs\steptwo}
\def\steptwo{\expandafter\ifx\cs\stoken\let\next=\stepthree
  \else\let\next=\nexttoken\fi \next}
\def\stepthree{\afterassignment\stepone\let\next= }
```

"\\" はコントロールシンボルなのでその後ろのスペースは読み飛ばされないことを利用して、"\stoken" にスペースを \let して「implicit なスペース」を作っていますね。あとは、⟨*token2*⟩ に当たるトークンがスペースの場合には \stepthree へと進むわけですが、その \stepthree では "\next" にスペースを \let することで、スペースを呑み込んでいます。

試しにこの 7 行を書き込んだファイルを仮に nonspacelet.tex として：

```
\nopagenumbers
\input␣nonspacelet

1a:␣\futurenonspacelet\cmdA*␣[\par
1b:␣cmdA␣=␣\meaning\cmdA

2a:␣\futurenonspacelet\cmdB*␣\stoken\stoken\stoken[\par
2b:␣cmdB␣=␣\meaning\cmdB

3a:␣\futurenonspacelet\cmdC*␣\stoken␣␣␣␣␣␣␣\stoken[\par
3b:␣cmdC␣=␣\meaning\cmdC
\bye
```

というファイルを pdfTEX で処理すると：

> 1a: *[
> 1b: cmdA = the character [
> 2a: *[
> 2b: cmdB = the character [
> 3a: *[
> 3b: cmdC = the character [

となります。「explicit なスペース」も「implicit なスペース」も呑み込まれて、"[" が無事 "\cmdA"〜"\cmdC" に \let されていることが確認できます（"*" と "[" の間のスペースも消えています）。

ちなみに、この \stepthree と同じ処理の仕方は、"Appendix D" の 376〜377 ページには都合 4 回も出てきます。\futurenonspacelet 以外の例は：

```
\def\c{\afterassignment\cc\let\next= }
```

と

```
\def\eat{\afterassignment\sanitize \let\next= }
```

なのですけれど、後者のマクロ名が "\eat" だというのがいかにもその役割を表わしていて、面白いですよね。

ちなみにちなみに、この \futurenonspacelet を内部で利用している例としては、Eplain の "\@getoptionalarg" というマクロがあります。また、LATEX 2ε の "\@ifnextchar" では、\afterassignment を使わずに同じような機能を実現しています（あと、余計なスペースを呑み込むのには、スペースをデリミタに使うというテクニックが用いられています）。

2 \def のパターンマッチ

引数が空ないしスペースであるのか、それともそれ以外なのかをチェックできるパッケージとして ifmtarg というのがあります[3]。その中味は、ほんの数行で：

```
\begingroup
\catcode`\Q=3
\long\gdef\@ifmtarg#1{%
  \@xifmtarg#1QQ\@secondoftwo\@firstoftwo\@nil}
\long\gdef\@xifmtarg#1#2Q#3#4#5\@nil{#4}
\long\gdef\@ifnotmtarg#1{%
  \@xifmtarg#1QQ\@firstofone\@gobble\@nil}
\endgroup
```

[3] Copyright Peter Wilson, 1996; Copyright Peter Wilson and Donald Arseneau, 2000.

となっています（段組みの幅の関係で 2 箇所改行していますが、オリジナルはどれも 1 行で書かれています）。ドキュメントによれば、この \@ifmtarg や \@ifnotmtarg は、次のような形で使うとのことです：

```
\@ifmtarg{⟨arg⟩}{⟨Code for arg empty⟩}{⟨Code for arg not empty⟩}
\@ifnotmtarg{⟨arg⟩}{⟨Code for not empty⟩}
```

ifmtarg のポイントは赤字で示した “\@xifmtarg” の引数のパターンマッチの部分だと思われます。なので、まずは \def の引数の取り方について確認をさせてください。

\def の引数の取り方の正確な説明は The TEXbook の “Chapter 20: Definitions (also called Macros)”（203〜204 ページ）の：

> How does TEX determine where an argument stops, you ask. Answer: There are two cases. A *delimited parameter* is . . . . [...] An *undelimited parameter* is . . . .

というところに書かれているのですけれど、ここではうんと雑に、次のようにまとめてみます：

> TEX は、\def によって定義されたマクロが使われているのに出遭った際には、そのマクロが引数をとるのかとらないのか、とるなら何個とるのか、そして、各引数にはデリミタが付いているのかいないのかを、既に知っています。このとき、
>
> - デリミタが付いていないパラメタトークン（“#n”）の場合は、最初に遭遇する 1 個のトークン（文字トークンまたはコントロールシーケンストークン）ないしは 1 個のグループが当該引数に相当します（但し、デリミタが付いていないパラメタトークンの場合には、スペースは引数にはなれないので、スペースに出遭っても、それは引数とはなりません）。
> - デリミタ付きのパラメタトークン（“#n⟨*delim*⟩”）の場合は、0 個や 2 個以上のトークンないしグループが引数に相当する場合があります。
>
>   TEX は “#n” の終端を示す ⟨*delim*⟩ を探しながら入力行を左から順番に読み込んでいきます。直後に ⟨*delim*⟩ が見つかれば、#n に当たる引数は「空（引数に該当するトークンは 0 個）」となります。他方、トークンやグループを 1 個読み込んでも ⟨*delim*⟩ が見つからないときには、2 個目以降のトークンないしグループも ⟨*delim*⟩ が見つかるまで読み込み続けるので、該当する引数は 2 個以上のトークンないしグループとなります。

例えば（定義時も使用時も “\makeatletter” が宣言済みだとして）：

```
\def\cmdA#1#2\@nil{...}
```

と定義した場合、#1 にはデリミタが付いていないので、\cmdA の後に最初に遭遇する 1 個めのトークンが #1 に該当する引数となります。他方、#2 には “\@nil” というデリミタが付いているので、TEX は #2 に当たる引数を決めるに際して \@nil を探します。そのため、もしも、\cmdA が：

```
\cmdA a\@nil
```

という風に使われていたならば、#2 は「空（0 トークン）」になります。#1 に当たる “a” の直後にデリミタ \@nil が見つかったからです。また、もしも、\cmdA が：

```
\cmdA abcde\@nil
```

という風に使われていたならば、今度は #2 は “bcde”（4 トークン）になります。#1 に当たる “a” の後、#2 についてはデリミタ \@nil が見つかるまで読み込み続けるからです。

なお、今の \cmdA の例では、ifmtarg パッケージでも使われている “\@nil” というデリミタを使いましたが、この “\@nil” は、LATEX のカーネルにも、パッケージ内にも、どこにも定義が見当たりません。しかしそれでも、未定義エラーにはなりません。というのも、マクロを定義するときにも、マクロの使用時に引数を読み込むときにも、TEX は展開を抑制しているからです[4]。ですから、例えば、\cmdB を：

```
\def\cmdB #1_#2^#3\my@terminal@marker{...}
```

と定義したとしても、その定義時や使用時に、“_” や “^” が数式モードの外だと言われたり、“\my@terminal@marker” が未定義だと言われたりすることもありません。

もう少しだけ、例を眺めてみます：

```
\nopagenumbers
\def\cmdA #1a{[#1]}
(1) \cmdA aa

\def\cmdB a#1a{[#1]}
(2) \cmdB aaa

\def\cmdC a#1#2a{[#1][#2]}
(3) \cmdC aaa

\def\cmdD #1#2a{[#1][#2]}
(4) \cmdD aa
\bye
```

というファイルを pdfTEX で処理すると、結果は次のようになります：

(1) []a
(2) []a
(3) [a][]
(4) [a][]

(1) の場合は、\cmdA の定義では、#1 に “a” というデリミタがついています。そのため、この \cmdA の使用例では、\cmdA の次にすぐに “a” が来ているので、#1 は空になります。

(2) の場合は、\cmdB の定義では、\cmdB の次には必ず “a” が続かねばならず、そして、#1 には “a” というデリミタがついています。そのため、この \cmdB の使用例では、最初の “a” は \cmdB の次に続かねばならない “a” で、2 番目の “a” は #1 のデリミタなので #1 はやはり空になります。

(3) の場合は、\cmdC の定義では、\cmdC の次には必ず “a” が続かねばならず、そして、#1 にはデリミタが付いていませんので、“a” の次に遭遇するトークンが #1 に相当します。そして、#2 には “a” というデリミタがついています。そのため、この \cmdC の使用例では、最初の “a” は \cmdC の次に続かねばならない “a” で、2 番目の “a” が、#1 に当たり、そして、3 番目の “a” は #2 のデリミタになるので、#2 は空になります。

(4) の場合は、\cmdD の定義では、#1 にはデリミタが付いておらず、#2 には “a” というデリミタがついています。そのため、この \cmdD の使用例では、1 番目の “a” が、#1 に当たり、#2 はやはり空になります。

以上を踏まえて、ifmtarg の “\@xifmtarg” の引数のパターンマッチについて考えてみましょう。\@xifmtarg の定義は：

```
\long\gdef\@xifmtarg#1#2Q#3#4#5\@nil{#4}
```

でしたよね。つまり、引数を 5 個とって、そのうちの 4 番目を返すという動作です。引数の取り方について詳しく見てみますと：

#1	#1 にはデリミタが付いていないので、最初に遭遇する 1 個のトークンが #1 に該当する引数
#2Q	#2 には “Q” というデリミタが付いているので、“Q” の前の 0 個以上のトークン群が #2 に該当する引数
#3	#3 にはデリミタなしなので、次に遭遇する 1 個のトークンが #3 に該当する引数
#4	#4 にはデリミタなしなので、次に遭遇する 1 個のトークンが #4 に該当する引数
#5\@nil	#5 には “\@nil” というデリミタが付いているので、“\@nil” の前の 0 個以上のトークン群が #5 に該当する引数

ということになります。

そして、この \@xifmtarg には、\@ifmtarg を介して：

```
#1QQ\@secondoftwo\@firstoftwo\@nil
```

[4] 次の「3 トークン化のタイミング」のところで、“Chapter 20: Definitions (also called Macros)” の該当箇所を引用しています。

が渡されています（ここで "#1" は、\@ifmtarg の引数です）。
まず、\@ifmtarg の引数 "#1" がスペースか空の場合として：

```
\@ifmtarg{}{YES}{NO}
```

を考えますと、\@xifmtarg には：

```
QQ\@secondoftwo\@firstoftwo\@nil
```

が渡されることになります。即ち：

- #1 <- Q（1 個目の Q）
- #2 <- 空（2 個目の Q の前にはもうトークン群はありません）
- #3 <- \@secondoftwo
- #4 <- \@firstoftwo
- #5 <- 空（\@nil の前にはもうトークン群はありません）

となって、4 番目の引数としては、"\@firstoftwo" が取り出されることになります。以上から、最終的には：

```
\@firstoftwo{YES}{NO}
```

が得られます。
次に、\@ifmtarg の引数 "#1" がスペースや空でない場合、例えば "#1" が "foo" という文字列だとして：

```
\@ifmtarg{foo}{YES}{NO}
```

を考えますと、\@xifmtarg には：

```
fooQQ\@secondoftwo\@firstoftwo\@nil
```

が渡されることになります。即ち：

- #1 <- f
- #2 <- oo（1 個目の Q の前のトークン群）
- #3 <- Q（2 個目の Q）
- #4 <- \@secondoftwo
- #5 <- \@firstoftwo（\@nil の前のトークン）

となって、4 番目の引数としては、"\@secondoftwo" が取り出されることになります。以上から、最終的には：

```
\@secondoftwo{YES}{NO}
```

が得られます。
いやぁ、すごいよく考えられてますよねぇ[5]。

3　トークン化のタイミング

さて、最後に、"\obeylines" を素材にして、カテゴリーコードが付与されるタイミングについて考えてみたいと思います。
plain.tex には、次のように書かれています[6]：

```
\def\space{ }
...
{\catcode`\^^M=\active % these lines must end with %
  \gdef\obeylines{\catcode`\^^M\active \let^^M\par}%
  \global\let^^M\par} % this is in case ^^M appears in a \write
\def\obeyspaces{\catcode`\ \active}
{\obeyspaces\global\let =\space}
```

初見だとブレースの使われ方にちょっと戸惑うのですけれど、ifmtarg パッケージの書き方と同じような書き方に直してみますと、こんな感じになります：

```
\begingroup
\catcode`\^^M=\active %
\gdef\obeylines{\catcode`\^^M=\active \let^^M=\par}%
\global\let^^M=\par %
\endgroup

\def\space{ }
\def\obeyspaces{\catcode`\ =\active}

\begingroup
\obeyspaces % i.e., \catcode`\ =\active
\global\let =\space
\endgroup
```

どちらの場合も、"^^M" や "スペース" のカテゴリーコードを 13 に変更しているので、その影響が外に漏れないようにグルーピングをしていて、そして、\def や \let による定義がグループの外でも有効になるように \global を付けているというわけです。
それで、\obeylines の定義の中で "^^M" を \active (=13) にした上で、その意味を "\par" と同じにしているのはいいのですけれど、その定義の前の行でも "^^M" を \active にしているのは、なぜなのでしょうか。

また少し、The TEXbook をうろうろしてみます。

まず、"Chapter 20: Definitions (also called Macros)"（203 ページ）には、次のようにあります：

> Definitions have the general form
>
> \def⟨*control sequence*⟩⟨*parameter text*⟩{⟨*replacement text*⟩}
>
> [...] For example, [...]. The definition
>
> \def\cs AB#1#2C$#3\$ {#3{ab#1}#1 c##\x #2}
>
> says that the ⟨*control sequence*⟩ \cs is to have a ⟨*parameter text*⟩ consisting of nine tokens
>
> A_{11}, B_{11}, #1, #2, C_{11}, $\$_3$, #3, $\boxed{\$}$, $\sqcup_{10}$
>
> (assuming the category codes of plain TEX), and a ⟨*replacement text*⟩ of twelve tokens
>
> #3, $\{_1$, a_{11}, b_{11}, #1, $\}_2$, #1, $\sqcup_{10}$, c_{11}, $\#_6$, $\boxed{\text{X}}$, #2.

なるほど。もうこれが答えなような気もしますが、"Chapter 7: How TEX Reads What You Type"（39 ページ）にも、次のようにあります：

> In other words, individual characters receive a fixed interpretation as soon as they have been read from a file, based on the category they have at the time of reading.

なるほどなるほど。あと、さきほど、マクロの定義時には ⟨*parameter text*⟩ は展開されないし、マクロの使用時に引数を読み込んでいるときにも展開はされないということに触れましたが、\def や \gdef によるマクロの定義時には ⟨*replacement text*⟩ も展開はされないとのことです（215 ページ）：

> Expansion is suppressed at the following times:
>
> - ...
> - When TEX is reading the arguments of a macro.
> - When TEX is absorbing the ⟨*parameter text*⟩ of a \def, \gdef, \edef, or \xdef.
> - When TEX is absorbing the ⟨*replacement text*⟩ of a \def or \gdef or \read; ...

つまり、TEX が \def の行を読み込む際には、⟨*parameter text*⟩ 内の文字列についても、⟨*replacement text*⟩ 内の文字列についても、その読み込み時点でのカテゴリーコードの割り振りに従って個々の文字の意味は解釈されて[7]、そしてその時点でのカテゴ

---

[5] なお、ifmtarg パッケージのドキュメントには、次のような付記があります："If you need a command to test for emptiness that doesn't include spaces, use the \tl_if_empty:nTF conditional from the expl3 package. \@ifmtarg is equivalent to expl3's \tl_if_blank:nTF."

[6] The TEXbook, "Appendix B"（352 ページ）ではなぜか "\obeyspaces" のほうが先に書かれています。あと、LATEX 2ε ですと ltplain.dtx でまったく同じ定義がなされています。

[7] 即ち、カテゴリーコード "0" の文字は ⟨*escape character*⟩ を意味し、"5" は ⟨*end-of-line character*⟩、"9" は ⟨*ignored character*⟩、"14" は ⟨*comment character*⟩、"15" は ⟨*invalid character*⟩ を意味するということです。cf. The TEXbook, Exercise 8.1 (p. 44), Answer 8.1 (p. 308).

リーコードが振られてトークン化される[8]、ということのようです。また、\def の行を読み込む際には、⟨*parameter text*⟩ 内でも、⟨*replacement text*⟩ 内でも展開は抑制される、とあります。

これらの説明を踏まえて再度、

```
\gdef\obeylines{\catcode`\^^M=\active \let^^M=\par}%
```

という定義について考えてみます。

まず、この \gdef が読み込まれる時点で、"^^M" に通常割り振られているカテゴリーコードは 5 (⟨*end of line*⟩) です。

そして、"\catcode`\^^M=\active" は、実行はされずに、そのまま読み込まれます。つまり、この時点では "^^M" のカテゴリーコードは変更されません[9]。

したがって、続く "\let^^M=\par" の時点でも、"^^M" のカテゴリーコードは 5 のママです。ということは、「はじめに」のところで見ましたように、この "^^M" は ⟨*end-of-line character*⟩ であって、それより後ろの文字列は捨てられてしまいます[10]。

"\let^^M=\par" の時点で "^^M" が ⟨*active character*⟩ であるようにするためには、\gdef が読み込まれる時点で "^^M" を \active にしておかねばなりません。それで：

```
\catcode`\^^M=\active %
\gdef\obeylines{\catcode`\^^M=\active \let^^M=\par}%
```

とする必要があったというわけです。

こうすれば、\gdef による定義段階では、⟨*replacement text*⟩ の

```
\catcode`\^^M=\active \let^^M=\par
```

は実行はされずにそのまま読み込まれてトークン化されて、その後、\obeylines の使用時に、"\catcode`\^^M=\active" と "\let^^M=\par" が順番に処理されるということになります。

ちなみに、"Appendix D" (380 ページ) では、ファイルを読み込んで verbatim に出力するマクロ "\listing" について解説されているのですが、そこで (段組みの幅の関係で 1 箇所 (ry)：

```
\def\listing#1{%
   \par\begingroup\setupverbatim\input#1 \endgroup}
```

> [...] Notice also that the commands '\input#1 \endgroup' will not be listed verbatim, even though they follow \setupverbatim, since they entered TEX's reading mechanism when the \listing macro was expanded (i.e., before the verbatim business was actually set up).

と言われているのも、\def による定義の段階で ⟨*replacement text*⟩ がトークン化されているということを思い出せば、理解できます。

その名前から推測できますように、"\setupverbatim" というマクロは、それに続く文字列に特殊文字が含まれていてもそれらを verbatim に出力できるように、特殊文字のカテゴリーコードをみな 12 に変更するものです。しかし、⟨*replacement text*⟩ の中味は \def の行が読み込まれる時点でトークン化されていて、そのときのカテゴリーコードの割り当てはその後変更できないので、\listing の使用時に \setupverbatim が働きを開始しても、既にトークン化されている "\input#1 \endgroup" はその影響を受けないというわけです。

この、一旦読み込まれてカテゴリーコードを振られてトークン化されると、その後そのカテゴリーコードは変更できないというのは、"Chapter 8: The Characters You Type" (48 ページ) に：

> when the arguments to a macro are first scanned, they are placed into a token list, so their categories are fixed once and for all at that time.

とありますように、マクロの引数についても当てはまります。

LaTeX の \verb コマンドが他のコマンドの引数の中では使えないというのも、これが理由ですよね。

オマケ

スペースとはあまり関係なさそうなのですが、本論の **2** と **3** で扱ったことがらの合わせ技っぽいものが The TEXbook の "Appendix E" (407 ページ) に載ってたので、オマケとさせてください。

"Appendix E" では、"letterformat.tex"、"concert.tex"、"manmac.tex" という 3 つの「マクロパッケージ」が解説されていますが、その最初の "letterformat.tex" に出て来る "\getaddress" というマクロと "\getclosing" というマクロが、目に見えない "^^M" をデリミタに使ったり、\def で再定義したりしていて、びっくりです：

```
{\obeylines\gdef\getaddress #1
  #2
  {#1\gdef\addressee{#2}%
    \global\setbox\theaddress=\vbox\bgroup\raggedright%
    \hsize=\longindentation \everypar{\hangindent2em}#2
    \def\endmode{\egroup\endgroup \copy\theaddress \bigskip}}}

{\obeylines\gdef\getclosing #1
  #2
  {#1\nobreak\bigskip \leftskip=\longindentation #2
    \nobreak\bigskip\bigskip\bigskip % space for signature
    \def
    {\endgraf\nobreak}}}
```

ちなみに、"^^M" をデリミタに使っている例として他に思い付くものとしては、texinfo の "\parseargline" というのもあります：

```
{\obeylines %
  \gdef\parseargline#1^^M{%
    \endgroup % End of the group started in \parsearg.
    \argremovecomment #1\comment\ArgTerm%
  }%
}
```

それでは、**Merry TEXmas & Happy TEXing!**

コンビニのコピー機でプリントをされる際にはご注意を

この pdf は Windows の Adobe Acrobat Reader と Sumatra PDF で閲覧する限りでは正常のように見えたのですが、一部のコンビニのコピー機では、欧文のダブルクォーテーションが正しく出力できないものがあるようです。もしもこの文書をコンビニのコピー機でプリントされる場合には、試しに 1 枚プリントをしてみて、それで正常だったら残りをプリントするようにしてください。

両脇の絵の出典は： Donald E. Knuth, "Icons for TEX and METAFONT," *TUGboat*, 14-4 (1993), pp. 387–389.

---

(8) 即ち、カテゴリーコードが "0" の文字から始まる文字列はコントロールシーケンストークンとなり、カテゴリーコードが "1"、"2"、"3"、"4"、"6"、"7"、"8"、"10"、"11"、"12" の文字は文字トークンとなるということです。カテゴリーコード "13" の ⟨*active character*⟩ は大体はコントロールシーケンストークン扱いだけれど、場合にはよっては文字トークンとして扱われることもあるとのことです。cf. The TEXbook, Exercise 7.3 (p. 39), Answer 7.3 (p. 307).

(9) ちょっとややこしいのですけれど、"^^M" は ⟨*invisible character*⟩ で、"`\^^M" はそのコードを表わす方法であるのに対して、"\^^M" はマクロで、"\def\^^M{\ }" と定義されています。そして、"^^M" のカテゴリーコードを \active にすると、"^^M" は ⟨*active character*⟩ となって、\def や \let できるようになります。

(10) "Chapter 8: The Characters You Type" says: If TEX sees an end-of-line character (category 5), it throws away any other information that might remain on the current line. (p. 47)