1 \use:c

예제

다섯 개의 필수 인자를 받는 명령 \nothing이 있다. 인자가 들어오는 순서대로 \l_a_tl, \l_b_tl, \l_c_tl, \l_d_tl, \l_e_tl로 반환하는 일을 하고 그친다.

한 개의 매크로를 가리키는 인자지시자는 :N이다. 예를 들어

```
\foo:N \l_tmpa_tl
```

이라는 것은 \foo:N이라는 함수(cs)가 \l_tmpa_tl이라는 매크로를 인자로서 취한다는 의미이다.
그런데 가끔은 "매크로"를 가리키기 위해서 그 "이름"만을 제공해야 할 때가 있다. 즉

```
\l_tmpa_tl
{l_tmpa_tl}
```

위의 것은 "매크로"이고 아래의 것은 "매르코의 이름"만을 가진 것이다. 이 매크로의 이름을 "csname"이라고 하고 이를 나타내기 위한 인자지시자가 c이다.
그러므로 \foo:N의 변형인 \foo:c가 있다고 하면

```
\foo:N \l_tmpa_tl
\foo:c {l_tmpa_tl}
```

이 둘은 동일한 의미이다. plain TEX에서 다음 둘이 동일한 것과 같다.

```
\mymacro
\csname mymacro\endcsname
```

이런 것이 언제 필요한가? 예를 들어보자면 다른 명령을 정의하기 위한 명령 \jt가 있다고 하자. 이 명령은 letter 인자를 받아서 그 앞에 \cnt@를 붙여준다고 하자.

```
\jt{home}{20}
```

이 명령의 수행 결과가

```
\def\cnt@home{20}
```

과 같이 되게 하려면 어떻게 해야 하는가?

```
\makeatletter
\def\jt#1#2{\expandafter\def\csname
    cnt@#1\endcsname{#2}}
\jt{home}{20}
\cnt@home
\makeatother
```

20

위의 코드에서 \expandafter가 꼭 필요한데, \def\csname이란 문장은 \csname이라는 매크로를 정의하려 드는 것이라서 허용되지 않기 때문이다.
expl3는 간단히 다음처럼 해도 좋다.

```
\ExplSyntaxOn
\cs_new:Npn \jt_exam:nn #1 #2
```

```
{
    \tl_set:cn { cnt_#1_tl } { #2 }
}

\jt_exam:nn { home } { 20 }
\tl_use:N \cnt_home_tl
\ExplSyntaxOff
```

20

\tl_use:N 대신 \tl_use:c를 쓰면

```
\ExplSyntaxOn
\jt_exam:nn { home } { HOME }
\tl_use:c { cnt_home_tl }
\ExplSyntaxOff
```

HOME

실제로 LaTeX에서 \thesection, \thesubsection과 같은 장절번호 명령, \c@page, \c@chapter와 같은 카운터 명령을 설정하는 것은 모두 위와 비슷한 방법으로 이루어진다.
또한, \tl_set: 명령은 상응하는 \tl_new:N가 없어도 새로운 변수에 값을 할당하는 것이 허용되는데 만약 이것이 되지 않는다면 위와 같은 코드를 쓸 때 엄청나게 불편할 것이다.
TMI이 아닌가 싶지만, LaTeX에서

```
\newcounter{abc}
```

명령을 실행하면 \c@abc라는 카운터 변수가 하나 생겨난다. 다음 코드를 보아라. 같은 결과를 내는 세 가지 방법이 나열되어 있다.

```
\makeatletter
\ExplSyntaxOn
\newcounter{abc}\setcounter{abc}{101}
\int_use:N \c@abc, \the\c@abc, \theabc
\ExplSyntaxOff
\makeatother
```

101,101,101

plain TeX에서

```
\newcount\a
```

라고 하면 \a를 바로 카운터로 쓸 수 있는데 비해 LaTeX에서

```
\newcounter{abc}
```

라고 하면 이 LaTeX 카운터의 조작은 \stepcounter, \setcounter, \addtocounter로 하고 그 결과를 \the...로 표현한다는 것을 우리는 이미 잘 알고 있다.

```
plaintex:
  \newcount\a \a=10 \advance\a by10 \the\a

e-tex:
  \newcount\b \b=\numexpr 10+20\relax \the\b
```

```
latex:
  \newcounter{a}%
  \setcounter{a}{10}\stepcounter{a}%
  \addtocounter{a}{10}%
  \thea

expl3:
\ExplSyntaxOn
  \int_new:N \l_a_int
  \int_set:Nn \l_a_int { 10 + 10 }
  \int_incr:N \l_a_int
  \int_use:N \l_a_int
\ExplSyntaxOff
```

plaintex: 20
e-tex: 30
latex: 21
expl3: 21

예제를 풀어보자. 1, 2, 3 등은 숫자(other)라서 변수 이름에 쓸 수 없으므로 이를 alph로 바꾸어서 넣어야 한다.

```
\ExplSyntaxOn
\NewDocumentCommand \nothing { mmmmm }
{
    \seq_set_from_clist:Nn \l_tmpa_seq { #1, #2, #3, #4, #5 }

    \seq_indexed_map_inline:Nn \l_tmpa_seq
    {
        \tl_set:cn { l_ \int_to_alph:n { ##1 } _tl } { ##2 }
    }
}

\nothing{Tomato}{Potato}{Banana}{Apple}{Kiwi}

\tl_use:N \l_a_tl,~\tl_use:N \l_d_tl
\ExplSyntaxOff
```

Tomato, Apple

2 weird 인자형

예제

`\testcmd{10.25}`와 같이 주어졌을 때 소수점을 기준으로 10과 25를 분리하고 각각 별도의 매크로(또는 tl)에 넣어 반환하여라.

esgutil 패키지에서 `\esg_fp_format:nn` 명령을 구현하기 위해서는 주어진 문제와 같은 일을 해야 했다. 이에 대해 간략히 알아본다. 문제를 간단히 하기 위해 들어오는 인자에 도트(.)가 없는 경우는 일단 고려하지 않기로 한다.

이 문제에 대한 전통적 해법은 다음과 같다.

```
\makeatletter
\def\splitbydot#1{\expandafter\@split@by@dot#1\end}
\def\@split@by@dot#1.#2\end{\def\a{#1}\def\b{#2}}
\makeatother
\splitbydot{10.25}
A = \a \\
B = \b
```

A = 10
B = 25

위의 예에서 `\@split@by@dot`과 같이 사용자 인터페이스와 무관하고 내부적으로 사용되는 함수에 @문자를 섞어쓰는 관행이 LaTeX에 있다. @문자의 catcode는 letter가 아니라 other이므로 원래는 매크로 이름에 쓸 수 없는데 이를 프로그래밍을 위하여 (임시로) letter로 만들어 매크로 이름의 일부로 쓸 수 있게 하라는 선언이 `\makeatletter`이다. @문자를 원래의 other로 되돌리라는 것은 `\makeatother`이다. 이 범위 안에서는 @을 letter로서 매크로 이름에 쓸 수 있으며, LaTeX style package인 `.sty`는 별도로 선언하지 않아도 `\makeatletter` 상태가 된다. (이 작용은 `\usepackage` 명령이 하는 일이므로 만약 단순히 `\input{foo.sty}`하였다면 그리 되지 않을 것이며, 만약 `.sty` 파일 내부에서 `\makeatother`를 선언하였다면 그 후로는 @이 other가 된다. 스타일 패키지를 만들 때 주의하여야 하는 사항들이다.)

expl3의 입장에서 생각하면 이것은 굉장히 간단한 문제이다. 바로 생각나는 해법은 다음과 같은 것이다.

```
\ExplSyntaxOn
\tl_new:N \l_a_tl
\tl_new:N \l_b_tl
\cs_new:Npn \split_by_dot:n #1
{
    \tl_set:Nn \l_a_tl { #1 }
    \regex_replace_once:nnN { ^ (.+?) \. (.*?) $ } { \1 } \l_a_tl
    \tl_set:Nn \l_b_tl { #1 }
    \regex_replace_once:nnN { ^ (.+?) \. (.*?) $ } { \2 } \l_b_tl
    a ~=~ \tl_use:N \l_a_tl \\
    b ~=~ \tl_use:N \l_b_tl
}
\split_by_dot:n { 10.279 }
\ExplSyntaxOff
```

a = 10
b = 279

실로 명쾌하다고 하겠다.
들어온 수를 "수"로 보고 계산하는 것도 간단하다.

```
\ExplSyntaxOn
\cs_new:Npn \split_by_dot_num:n #1
{
    \fp_set:Nn \l_tmpa_fp { #1 }
    \fp_set:Nn \l_tmpb_fp { trunc ( \l_tmpa_fp ) }
    \tl_set:No \l_a_tl { \fp_use:N \l_tmpb_fp }
    \tl_set:No \l_b_tl { \fp_eval:n { \l_tmpa_fp - \l_tmpb_fp } }
    a ~=~ \tl_use:N \l_a_tl \\
```

```
    b ~=~ \tl_use:N \l_b_tl
}

\split_by_dot_num:n { 10.79 }
\ExplSyntaxOff
```

a = 10
b = 0.79

이 방법은 소수 부분이 정말로 소수로 표현되기 때문에 특정한 용도에 더 적합할 수 있다.
마지막으로 처음에 보인 plain TEX 코드를 expl3로 직접 번역하면 다음처럼 된다.

```
\ExplSyntaxOn
\cs_new:Npn \split_by_dot_exp:n #1
{
    \split_by_dot_exp_subfn:w #1\q_stop
    a ~=~ \tl_use:N \l_a_tl \\
    b ~=~ \tl_use:N \l_b_tl
}

\cs_new:Npn \split_by_dot_exp_subfn:w #1 . #2 \q_stop
{
    \tl_set:Nn \l_a_tl { #1 }
    \tl_set:Nn \l_b_tl { #2 }
}

\split_by_dot_exp:n { 10.297 }

\ExplSyntaxOff
```

a = 10
b = 297

이 기법에 사용된 것은 `w` (weird) arguments를 이용하는 것이다. 특별한 인자 형식으로부터 인자를 넘겨받기 위한 목적으로 사용한다. 종지 위치를 `\q_stop`이라는 quark으로 표시하는 것은 아까 보인 코드에서 `\end`를 주는 것과 동일하다.

3 인자 수

예제

12개의 필수(mandatory) 인자를 받아서 처리하는 명령 `\testtwelve`를 작성해보아라.

`\fbox`하고 콤마로 나열하는 걸 해보겠다.
`#10`이란 게 없기 때문에 9개까지만 인자를 설정할 수 있다. 그 이상이라면 그 이후의 인자를 취하는 명령을 해당 명령 마지막에 두어서 처리하는 것이 가능하다.

```
\ExplSyntaxOn
\NewDocumentCommand \testtwelve { mmmmmm }
{
    \fbox{#1}, \fbox{#2}, \fbox{#3}, \fbox{#4},
```

```
    \fbox{#5}, \fbox{#6},
    \testtwelvenext
}

\NewDocumentCommand \testtwelvenext { mmmmmm }
{
    \fbox{#1}, \fbox{#2}, \fbox{#3}, \fbox{#4},
    \fbox{#5}, \fbox{#6},
}
\ExplSyntaxOff

\testtwelve{a}{b}{c}{d}{e}{f}{g}{h}{i}{j}{k}{l}
```

a,b,c,d,e,f,g,h,i,j,k,l,

expl3의 함수(cs)도 필요하다면 같은 방식으로 할 수 있다.
그런데 가만히 생각해보면 인자를 취하는 것은 전형적인 "재귀" 과정이다. 그러니까,

```
\ExplSyntaxOn
\int_zero:N \l_tmpa_int
\cs_new:Npn \test_twelve:n #1
{
    \fbox{#1}
    \int_incr:N \l_tmpa_int
    \int_compare:nTF { \l_tmpa_int < 12 }
    {
        , \test_twelve:n
    }
    {
        .
    }
}

\test_twelve:n {a}{b}{c}{d}{e}{f}{g}{h}{i}{j}{k}{l}
\ExplSyntaxOff
```

a,b,c,d,e,f,g,h,i,j,k,l.

열두 개의 인자를 취하는 것은 위와 같다.

4 가변개의 인자

예제

새로이 정의하는 명령은 다음과 같은 형식이다.

```
\mycmd{1}{2}{5}{2}
```

그러면 인자로 주어지는 수만큼 점을 찍어준다.
• •• ••••• ••
인자는 몇 개가 주어질지 사전에 알 수 없다. 단 중괄호가 아닌 부호가 오면 그 후로는 인자로 보지 않는다.

plain TEX으로는 다음처럼 할 수 있다. 테스트를 위하여 마지막에 (4)와 같이 중괄호가 아닌 부호를 두었다.

```
\newcount\n
\def\mycmd{\futurelet\next\mycmdsub}
\def\mycmdsub{\ifx\next\bgroup\expandafter\mycmdproc\fi}
\def\mycmdproc#1{\n=1\loop\textbullet\advance\n
    by1\ifnum\n<#1\repeat\quad\mycmd}

\mycmd{2}{5}{3}(4)
```

•• ••••• ••• (4)

이 코드는 설명하지 않겠다.
이런 종류의 "재귀" 코드에서 가장 중요한 것은 종료 조건을 분명하게 하는 것이다. 예를 들어

```
\mycmd{{1}{2}{5}{3}}
```

과 같이 인자 범위를 명확히 설정해주거나

```
\mycmd{1}{2}{5}{3};
```

과 같이 종지부호를 적게 하는 방식으로 명령을 정의하면 그 인자를 처리하는 것이 아주 쉬워진다. 두 번째와 같이 종지부호를 세미콜론으로 하는 경우의 코드를 보자.

```
\ExplSyntaxOn
\NewDocumentCommand \mycmd {u;}
{
    \tl_set:Nn \l_tmpa_tl { #1 }
    \tl_map_inline:Nn \l_tmpa_tl
    {
        \mycmd_dot:n { ##1 }
    }
}

\cs_new:Npn \mycmd_dot:n #1
{
    \int_step_inline:nn { #1 }
    {
        \textbullet
    }
    \quad
}
```

```
\ExplSyntaxOff

\mycmd{1}{2}{5}{3};
```

● ●● ●●●●● ●●●

실용적 목적으로 문서 명령을 작성하는 경우라면 되도록 "명시적 종료 토큰"을 사용자가 입력하게 만드는 것이 좋다.
그러나 어찌됐든 처음에 보였듯이 종지부호 같은 것이 없이 가변 인자를 구현해야 한다면, 어쩔 수 없이 토큰 검사를 하는 도리밖에 없다. 처음에 보인 plain TEX의 코드를 expl3 방식으로 다시 쓰면 다음과 같을 것이다. 마지막의 `(5)`는 테스트를 위하여 중괄호가 아닌 부호를 둔 것이다.

```
\ExplSyntaxOn
\cs_new:Npn \my_cmd:n #1
{
    \mycmd_dot:n { #1 }

    \peek_catcode:NT \c_group_begin_token
    {
        \my_cmd:n
    }
}

\NewDocumentCommand \mycmd { }
{
    \my_cmd:n
}
\ExplSyntaxOff

\mycmd{3}{2}{1} {4}
```

●●● ●● ● 4

이 명령에서 유심히 볼 것은 중괄호가 아닌 것이 오면 끝난다고 했으므로 space token이 와도 마찬가지로 종료된다는 것이다. 만약 space를 무시하고 위와 같은 경우에 마지막 `{4}`도 유효하게 만들려면 `\peek_catcode_ignore_spaces:N`을 쓰면 되기는 하지만 space로 종료하지 못하면 다음 non-space token이 중괄호가 아니어야 하므로, 다른 식의 종료 조건을 반드시 구비하여야 한다. 그렇게 하지 않으면 에러가 발생할 가능성이 너무 높아진다.
문서 명령을 정의하면서 인자 자리를 비우고 재귀 함수인 `\my_cmd:n`을 부르는 것으로 그친 것은 드물지 않은 기법이므로 익숙할 것이다. 재귀함수로써 그 이후에 이어지는 입력 스트림을 처리하는 명령을 작성할 때는 재귀함수의 호출이 이 명령의 "마지막" 토큰이어야 한다는 점을 기억하자. 명시적으로 주어지는 인자는 없지만 `\my_cmd:n`이 어차피 한 개의 토큰을 받아서 (absorb) 처리하기 시작할 것이므로 결과적으로 이후의 인자를 모두 처리하고 종료할 것을 예상할 수 있다. 요컨대, `\futurelet`으로 다음에 올 토큰을 검사하던 것을 expl3에서는 `\peek_catcode:`나 `\peek_charcode:`를 통해 대부분 해결할 수 있다. (다만 다음에 올 토큰이 space token일 경우에 예외적인 취급이 필요할 수 있는데 이에 대해서는 이후 절을 달리하여 토론하기로 한다.)

지금까지 배운 내용으로 다음 문제를 해결할 수 있다.

연습문제

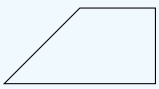
다음과 같은 사용자 TikZ 명령을 작성하려 한다. `\mydraw(0,0)(1,1)(2,1)`라고 하면

```
\draw (0,0)--(1,1)--(2,1)--cycle;
```

로 그려주는 명령이다. 명령의 마지막에 세미콜론이 붙지 않으며 가변 개의 점을 지정할 수 있다.

Special Course는 연습문제를 숙제로 삼지 않으므로 모범답안을 보이면 다음과 같다.

```
\ExplSyntaxOn
\NewDocumentCommand \mydraw {}
{
    \tl_clear:N \l_tmpa_tl
    \my_draw:w
}

\cs_new:Npn \my_draw:w (#1,#2)
{
    \tl_put_right:Nn \l_tmpa_tl { (#1,#2) }
    \peek_charcode:NTF (   %%% if next token is open-paren
    {
        \tl_put_right:Nn \l_tmpa_tl { -- }
        \my_draw:w
    }
    {
        \tl_put_right:Nn \l_tmpa_tl { -- cycle }
        \draw_final:N \l_tmpa_tl
    }
}

\cs_new:Npn \draw_final:N #1
{
    \exp_last_unbraced:No
    \draw #1 ;
}
\ExplSyntaxOff
\begin{tikzpicture}
\mydraw(0,0)(1,1)(2,1)(2,0)
\end{tikzpicture}
```



`\my_draw:w`의 인자형에 대해 조금 해설을 붙이자면 실제로는 `\my_draw:w (#1)`과 같이 하고 `#1`만을 put right해주어도 이 코드에서는 문제를 일으키지 않는다. 그런데 만약 x좌표와 y좌표를 조작해야 할 일이 있다면 저런 형식으로 인자를 받는 것이 좋다. 또한 쉼표가 들어가지 않은 material이 괄호 안에 들어온다고 다 유효한 것으로 취급하는 것보다는 쉼표가 없으면 아예 에러를 토하는 것이 낫다. 예시 코드가 그런 작용을 한다.

5 modulo 연산과 조건식

예제

`\int_compare:nTF { \int_mod:nn { #1 } { 5 } == 0 } {T}{F}`과 같이 쓰는 조건식을 `\my_mod:TF (#1 / 5) {T}{F}`와 비슷하게 줄여쓸 수 없을까? 그런 명령을 정의해보아라.

expl3에 modulo 연산자가 없고 함수만 있기 때문에 함수명을 길게 적는 것이 귀찮은 일일 수 있다.

```
\ExplSyntaxOn

\prg_new_conditional:Npnn \my_modzero:w (#1//#2) { p, T, F, TF }
{
    \int_compare:nTF { \int_mod:nn { #1 } { #2 } == 0 }
    {
        \prg_return_true:
    }
    {
        \prg_return_false:
    }
}

\my_modzero:wTF (15 // 5) { TRUE } { FALSE } \\
\my_modzero:wTF (16 // 5) { TRUE } { FALSE } \\
\bool_if:nTF { \my_modzero_p:w (17//5) } { T } { F }

\ExplSyntaxOff
```

TRUE
FALSE
F

인자형을 `:nn`으로 한다면 두 개의 중괄호 인자가 오는 것으로 약속했고, 이 함수는 `(#1//#2)` 형식, 즉 중괄호를 쓰지 않고 괄호(parenthesis)와 슬래시로 인자를 주기 때문에 인자형을 `:w`로 해두는 편이 낫다. 물론 `#1`이나 `#2` 위치에 중괄호로 둘러싸인 정수 표현식이 오는 것은 상관없다. (다음 보기는 테스트용에 가깝다. 너무 복잡한 식이 오면 나중에 알아보기 어려워진다.)

```
\ExplSyntaxOn
\int_set:Nn \l_tmpa_int { \fp_eval:n {
    round ( sqrt ( 200 ) ) } }
\my_modzero:wTF ( { \l_tmpa_int * ( 3 +
    \int_max:nn { 5 } { 9 } ) } // 3 )
{ divisible~by~3 }
{ not~divisible~by~3 }
\ExplSyntaxOff
```

divisible by 3

`\my_modzero_p:w`는 boolean 값을 반환한다. `\my_modzero:wTF`는 (T 또는 F중 하나는 생략될 수 있음) "주어진 나머지 연산식이 0이면"이라는 조건에 따라 실행할 코드를 줄 수 있다.

6 tl map과 space token

예제

입력되는 문자열의 각 문자에 fbox를 치는 명령을 작성하되 space 위치에 가로 4pt, 세로 7pt의 막대를 그려라.

입력: `\test{Lorem ipsum dolor}`
출력: L o r e m i p s u m d o l o r

tl에서 space는 보존된다. (`...ignore_spaces` 관련 함수를 쓰지 않은 한.)

```
\ExplSyntaxOn
\NewDocumentCommand \foo { m }
{
    \tl_set:Nn \l_tmpa_tl { #1 }
    \l_tmpa_tl
}
\ExplSyntaxOff

\foo{Lorem ipsum dolor sit}
```

Lorem ipsum dolor sit

그러나, `\tl_map_...`에서 next token은 non-space token을 가리킨다.

```
\ExplSyntaxOn
\NewDocumentCommand \foo {m }
{
    \tl_set:Nn \l_tmpa_tl { #1 }
    \tl_map_inline:Nn \l_tmpa_tl { \fbox { ##1 } }
}
\ExplSyntaxOff

\foo{Lorem ipsum dolor sit}
```

L o r e m i p s u m d o l o r s i t

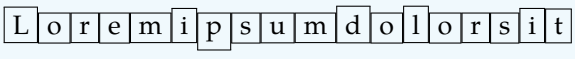
이와 같은 일은 `\peek_catcode:N \c_space_token`에서도 일어난다. 단순 입력 스트림에서는 스페이스 토큰을 peeking하는 것이 가능하기 때문에 다음과 같은 코드는 동작하지만 (이와 같이 중괄호로 묶어서 인자로 전달하지 않고 토큰을 늘어놓는 방식은 앞서 소개한 가변 인자 처리와 사실상 동일한 아이디어임을 상기하여라.)

```
\ExplSyntaxOn
\cs_new:Npn \test_r:n #1
{
    \peek_catcode:NTF \c_space_token
    {
        #1 \rule{4pt}{7pt}
    }
    {
```

```
        \fbox{#1}
        \test_r:n
    }
}

\cs_set_eq:NN \test \test_r:n
\ExplSyntaxOff

\test Lorem ipsum dolor sit
```

`\NewDocumentCommand`의 m인자 안에 들어가고 나면 space를 peek할 수 없게 된다. (`\peek_...` 함수는 다른 함수의 인자로 들어가지 않은 입력 스트림에서 정상적으로 동작한다는 것을 기억하자.) 아래 코드에서 `\peek_catcode` 부분은 항상 false로서 스페이스 검출에 성공하지 못하였다.

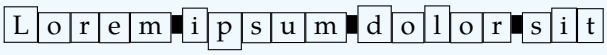
```
\ExplSyntaxOn
\NewDocumentCommand \foo { m }
{
    \foo_recur:n #1 \q_recursion_tail \q_recursion_stop
}

\cs_new:Npn \foo_recur:n #1
{
    \quark_if_recursion_tail_stop:n { #1 }
    \peek_catcode:NTF \c_space_token
    {
        #1 \rule{4pt}{7pt}
    }
    {
        \fbox{#1}
    }
    \foo_recur:n
}

\ExplSyntaxOff
\foo{Lorem ipsum dolor sit}
```



주어진 문제를 풀기 위해서 space를 처리하는 방법은 몇 가지가 있다. space 단위로 seq에 넣어 처리하는 것이 가장 직관적이고

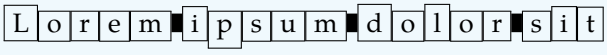
```
\ExplSyntaxOn
\NewDocumentCommand \foo { m }
{
    \seq_set_split:Nnn \l_tmpa_seq { ~ } { #1 }
    \seq_indexed_map_inline:Nn \l_tmpa_seq
    {
        \f_each:n { ##2 }
        \int_compare:nT { ##1 < \seq_count:N \l_tmpa_seq }
```

```
        { \rule{4pt}{7pt} }
    }
}

\cs_new:Npn \f_each:n #1
{
    \tl_set:Nn \l_tmpa_tl { #1 }
    \tl_map_inline:Nn \l_tmpa_tl { \fbox { ##1 } }
}
\ExplSyntaxOff

\foo{Lorem ipsum dolor sit}
```



다음처럼 해볼 수도 있다. space를 특정 부호(여기서는 |)로 미리 바꾸어두는 것이다.

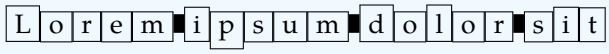
```
\ExplSyntaxOn
\NewDocumentCommand \foo { m }
{
    \tl_set:Nn \l_tmpa_tl { #1 }
    \regex_replace_all:nnN { \s } { | } \l_tmpa_tl
    \tl_map_inline:Nn \l_tmpa_tl
    {
        \str_if_eq:nnTF { ##1 } { | }
        {
            \rule{4pt}{7pt}
        }
        {
            \fbox{##1}
        }
    }
}

\ExplSyntaxOff
\foo{Lorem ipsum dolor sit}
```



또는 "재귀"적으로 정의하는 방법도 있다. `\q_recursion_tail (stop)` 기법은 한 번 써보았으니 여기서는 `\q_nil`을 끝에 두고 종료 조건을 설정하는 방식으로 해보았다.

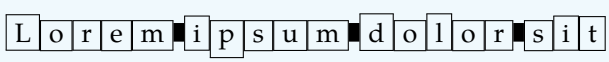
```
\ExplSyntaxOn
\NewDocumentCommand \foo { m }
{
    \tl_set:Nn \l_tmpa_tl { #1 }
    \regex_replace_all:nnN { \s } { | } \l_tmpa_tl
    \exp_last_unbraced:Nx
    \foo_fn:n \l_tmpa_tl \q_nil
}
```

```
\cs_new:Npn \foo_fn:n #1
{
    \quark_if_nil:NF #1
    {
        \token_if_eq_charcode:NNTF #1 |
        {
            \rule{4pt}{7pt}
        }
        {
            \fbox{#1}
        }
        \foo_fn:n
    }
}

\ExplSyntaxOff
\foo{Lorem ipsum dolor sit}
```



이 코드는 동작하지만 `\quark_if_...`로 처리하는 편이 더 안전하다. 연습 삼아 `\quark_if_recursion_...`으로 똑같은 코드를 적어보겠다.

```
\ExplSyntaxOn
\NewDocumentCommand \foo { m }
{
    \tl_set:Nn \l_tmpa_tl { #1 }
    \regex_replace_all:nnN { \s } { | } \l_tmpa_tl
    \exp_last_unbraced:Nx
    \foo_s:n \l_tmpa_tl \q_recursion_tail \q_recursion_stop
}

\cs_new:Npn \foo_s:n #1
{
    \quark_if_recursion_tail_stop:n { #1 }

    \token_if_eq_charcode:NNTF #1 |
    {
        \rule{4pt}{7pt}
    }
    {
        \fbox{#1}
    }
    \foo_s:n
}

\ExplSyntaxOff
\foo{Lorem ipsum dolor sit}
```



마지막으로 다음 코드는 peek catcode로 space token을 검출하여 뭔가를 할 수 있는 "재귀" 함수 코드이다.

여기서는 space token 직전 토큰의 색상을 빨갛게 해보았다. [FS] (Full Stop) 부호는 종료 조건 (스페이스가 더이상 종료 조건이 되지 못하기 때문에 마침표 사용) 을 검출하기 위한 테스트 코드이다.

```
\ExplSyntaxOn

\cs_new:Npn \_chk_and_exit:nN #1 #2
{
    \token_if_eq_catcode:NNF #1 .
    { #2 }
}

\cs_new:Npn \foo_rc:n #1
{
    \str_if_eq:nnTF { #1 } { . }
    { [FS] }
    {
        \peek_catcode:NTF \c_space_token
        {
            { %%% \fboxsep made local
                \dim_set:Nn \fboxsep { 1pt }
                \colorbox{gray!30}{\color{red}\fbox{#1}}
            }
            \rule{4pt}{7pt}
            \_chk_and_exit:nN #1 \foo_rc:n
        }
        {
            \fbox{#1}
            \_chk_and_exit:nN #1 \foo_rc:n
        }
    }
}

\cs_set_eq:NN \foo \foo_rc:n

\ExplSyntaxOff
\foo Lorem ipsum dolor sit. Hello boys.
```

Lorem ipsum dolor sit[FS] Hello boys.

plain TEX에서는 `\@sptoken`만 가지고 next token으로 검출하는 것이 비교적 간단하다. 그런데 expl3에서 살짝 복잡해진 이유는 expl3가 space token을 건드려놓았기 때문이다. `\ExplSyntaxOn`으로 스페이스는 모두 사라진다는 걸 생각해보면 expl3가 space token을 특별하게 취급하고 있다는 것이 짐작될 것이다. 그리고 `...ignore_spaces` 관련 함수도 많고.
그 덕분에 예전에는 할 수 없었던 여러 가지를 에러 없이 손쉽게 하게 된 것도 있고, 이처럼 space 자체를 제어하려면 조금 고민이 필요한 부분도 생겨나고 한 것이라고 보면 되겠다. 전체적으로 보아서 expl3의 스페이스 취급은 실보다는 득이 많은 방향이었다는 판단이므로 여기 소개한 기법을 필요할 때 활용할 수 있으면 그것으로 좋다.

Nova de Hi.