문제

문자열 인자를 받아서 각 글자마다 `\fbox`를 치는 명령 `\foobox`를 작성하여라.

입력: `\foobox{Korea}`
출력: K o r e a

1 new, set, map, use

expl3의 리스트 자료형 tl, clist, seq에 대하여,

`new` 새로운 변수를 선언한다. 단, 스크래치 변수라면 new가 필요없다.

`set` 변수에 새로운 값을 할당한다. (전역 변수에 대해서는 `gset`.)

`map` 리스트 각 항목(item)에 대하여 순차적으로 어떤 작용(function)을 적용한다.

`use` 리스트 변수를 확장(expand)하여 입력 스트림에 남긴다.

참고 1. 일부 자료형(특히 tl)은 `\tl_new:N`가 없어도 `\tl_set:Nn`할 수 있다. 그러나 `\tl_set:Nn`이 아닌 다른 tl 조작함수를 new하지 않고 부를 수는 없으므로 tmpa, tmpb가 아니고 사용자 변수를 만들어 쓴다면 new하는 것이 안전하다.

참고 2. `\tl_use:N \l_tmpa_tl`은 이것이 입력 스트림에 남겨지는 경우라면 `\l_tmpa_tl`과 결과가 동일하다. 그러나 차이라면 `\tl_use:N`이 쓰이면 대상 tl을 한 번(o) 확장한다는 것이다. 지금은 이 둘이 큰 차이가 없는 상황만을 주로 다루게 된다.

map에는 두 가지 방법이 있다. 하나는 inline이고 다른 하나는 function을 부르는 방식이다. 단순히 `\fbox`를 부르는 경우만을 예로 들면 다음과 같다.
inline mapping:

```
\tl_new:N \l_my_tl
\tl_set:Nn \l_my_tl { We shall overcome }
\tl_map_inline:Nn \l_my_tl
{
    \fbox { #1 }
}
\tl_use:N \l_my_tl
```

function mapping:

```
\cs_new:Npn \myfbox_fn:n #1
{
    \fbox { #1 }
}
```

```
\tl_new:N \l_my_tl
\tl_set:Nn \l_my_tl { We shall overcome }
\tl_map_function:NN \l_my_tl \myfbox_fn:n
\tl_use:N \l_my_tl
```

inline mapping에서 주의할 점은 이 코드가 다른 함수의 정의 범위 안에 들어가 있다면 #1이 아니라 ##1이 되어야 한다는 것이다. expl3는 ###1부터의 # 중첩을 허용하지 않기 때문에 nesting이 필요한 상황에서는 inline mapping을 쓰기 불편하다.
function mapping의 경우는 인자를 한 개 받아들이는 mapping function을 미리 정의해두어야 한다.
주어진 문제는 따라서 다음과 같이 해결할 수 있다.

```
\ExplSyntaxOn
\NewDocumentCommand \foobox { m }
{
    \my_foobox:n { #1 }
}

\cs_new:Npn \f_box_it:n #1
{
    \fbox { #1 }
}

\cs_new:Npn \my_foobox:n #1
{
    \tl_set:Nn \l_tmpa_tl { #1 }
    \tl_map_function:NN \l_tmpa_tl \f_box_it:n
}
\ExplSyntaxOff

\foobox{Korea}
```

K o r e a

clist와 seq를 연습해보자.

```
\ExplSyntaxOn
\clist_new:N \l_my_clist
\clist_set:Nn \l_my_clist { abc,
    123, ABC }
\clist_map_function:NN \l_my_clist
    \myfbox_fn:n
\par
\clist_use:Nn \l_my_clist {,~}
\ExplSyntaxOff
```

abc 123 ABC
abc, 123, ABC

clist는 항목 구분자가 쉼표로 통일되어 있지만 seq는 매우 유연하다. 그 대신 항목 구분자를 반드시 매번 밝혀야 한다.

```
\ExplSyntaxOn
\seq_new:N \l_my_seq
\seq_set_split:Nnn \l_my_seq {|} {
    abc|123|ABC }
\seq_map_function:NN \l_my_seq
    \myfbox_fn:n
\par
\seq_use:Nn \l_my_seq {,~}
\ExplSyntaxOff
```

abc | 123 | ABC
abc, 123, ABC

seq에서 set을 위하여 `\seq_set_split:Nnn`과 `\seq_set_from_clist:Nn`이 흔하게 쓰인다.
seq와 clist는 간단한 db나 stack처럼 활용하게 될 수 있다. 굳이 말하자면 seq/clist는 다른 언어의 리스트에 가깝고 tl은 스트링과 비슷하다. clist와 seq의 다른 활용 가능성은 이어지는 강좌에서 다루게 된다.

map과 꼬리재귀 map 함수는 expl3 언어의 가장 중요한 기능 중의 하나다. 예를 들어 주어진 문자열에서 각 문자마다 fbox를 치는 이 단순한 일은

```
\def\acmd#1{\expandafter\aacmd#1\end}
\def\aacmd#1{\ifx#1\end\let\next\relax
    \else \fbox{#1}\let\next\aacmd\fi\next}
```

이렇게 정의하는 것이 plainTEX스러운 방법일 것이다. 또는 우리가 즐겨 사용하는

```
\def\fifo#1{\ifx#1\ofif\ofif\fi\process#1\fifo}
\def\ofif#1\fifo{\fi}
\def\process#1{\fbox{#1}}
\def\acmd#1{\expandafter\fifo#1\ofif}
```

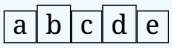
fifo ... ofif 기법을 써도 마찬가지인데, 이 두 방법은 모두 이른바 "TEX의 꼬리재귀"를 적용하고 있다. 실상 리스트 매핑이라는 것은 꼬리재귀를 아주 쉽게 사용하도록 해둔 것이다.
expl3에서는 이른바 "꼬리재귀"를 직접 적용할 수 없는가? 물론 가능하다. 이것을 plainTEX에 비하여 더 안전하게 수행할 수 있도록 *quark* 데이터타입이 정의되어 있다. 당분간 이 자료형을 활용하게 될 일은 없을 것이지만 여기에서 재미삼아 (low-level expl3의) 예를 들어두고 간다.

```
\ExplSyntaxOn
\NewDocumentCommand \test { m }
{
    \test_fn:n #1 \q_recursion_tail \q_recursion_stop
}

\cs_new:Npn \test_fn:n #1
{
```

```
    \quark_if_recursion_tail_stop:n { #1 }
    \myfbox_fn:n { #1 }
    \test_fn:n
}
\ExplSyntaxOff

\test{abcde}
```



tl 관련 기본 함수 다음 함수가 자주 사용할 법한 것이다.

- `\tl_clear:N` 현재 tl 안의 내용을 모두 지운다. 이것은 tl 자체를 삭제하는 것과는 다르다.
- `\tl_put_right:Nn` n 인자의 내용을 현재의 tl의 뒤(오른쪽)에 붙여넣는다. 반면 `\tl_put_left:Nn`은 앞(왼쪽)에 붙인다.
- `\tl_set_eq:NN` `\let\A\B`와 거의 동일하다. 두 개의 tl을 일치시킨다.
- `\tl_count:N` tl의 토큰 개수를 반환한다.
- `\tl_reverse:N` 현재 tl의 토큰을 역순으로 배열한다.

2 정수에 관한 기초 지식

수(numbers)를 본격적으로 다루기 전에 학습의 진행을 위해서 정수(*int*)에 대하여 다음 몇 가지를 알아두자.

new 정수형 변수는 사용 전에 반드시 new 선언을 하여야 한다. 단, tmpa, tmpb라는 스크래치 변수는 이미 정의되어 있다.

set `\int_set:Nn` 명령의 n인자 부분은 `\int_eval:n`으로 확장되어 적용된다.

integer expression 정수 표현식은 `+`, `-`, `*`, `/`과 괄호(`()`)로 이루어진다. 누승(지수)이나 계승(팩토리얼)은 정수와 관련 있음에도 불구하고 fp 자료형에서 담당한다. 참고로 `/` 연산은 (plainTEX의 `\divide`와 달리) truncate가 아니라 round이다. `\int_eval:n { 7/4 }`의 결과는 2. 같은 연산을 `\divide`로 하면,

```
\newcount\tmpcnt\tmpcnt=7
\divide\tmpcnt by4 \the\tmpcnt
```

결과는 1. trunc와 modulo를 위해서는 따로 함수가 준비되어 있다.

compare 정수 비교식은 `>`, `<`, `=` (`==`), `!=`, `>=`, `<=`를 비교 연산자로 하고 bool 값을 반환한다. 이것은 `\int_compare:nTF`의 첫 인자로만 쓰인다. 이 함수 `\int_compare:nTF`는 사실상 expl3의 정수형 if-문이므로 그 사용례를 숙지해두어야 한다.

use tl과는 달리 `\l_tmpa_int`만을 입력 스트림에 남기면 에러가 발생한다. 정수형은 반드시 `\int_use:N \l_tmpa_int` 꼴로 쓰거나 `\int_eval:n`한 다음에야 tl에 할당될 수 있다.

수에 대하여 따로 공부하기 전에 당장 필요한 것은 다음 몇 가지 명령이다.

- `\int_zero:N` 주어지는 변수의 값을 0으로 만든다. `\int_set:Nn \l_tmpa_int {0}`와 동일. 편의를 위한 `\int_zero_new:N`이라는 것도 있다.
- `\int_incr:N` 주어지는 변수의 값을 1 증가시킨다. `\int_add:Nn \l_tmpa_int {1}`과 동일.
- `\int_compare:nTF` n에는 정수 비교식이 온다.
- `\int_use:N`
- `\int_if_odd:nTF`, `\int_if_even:nTF`

연습문제

기본 1. 문제에서 제시한 `\foobox`에서 각 글자에 아래 예시와 같이 색상을 입혀보아라. 각 글자 사이에 1pt의 간격을 둔다.

기본 2. `\foobox`에서 인자로 주어진 글자 수를 세어서 마지막에 괄호와 함께 표현하는 명령 `\barbox`를 작성하여라.

발전 3. 기본 문제 2번의 색상상자를 홀수번째 오는 문자에만 적용하도록 `\baroddbox` 명령을 작성하여라. 문자 사이에는 1pt의 간격을 둔다.

1. 입력: `\foobox{world}`
   출력: w o r l d
2. 입력: `\barbox{world}`
   출력: w o r l d (5)
3. 입력: `\oddbarbox{world}`
   출력: w o r l d (5)

문제

주어지는 문자열을 3개 단위로 끊어서 다음 출력예와 같이 배열하여라. 마지막 항목은 3개가 되지 않을 수 있으며 스페이스는 무시한다.

입력: `\scmd{abcdefg hijklmn}`
출력: abc def
ghi jkl
mn

`\tl_map`할 적에 스페이스(10)는 하나의 토큰(next token)으로 치지 않는다. 스페이스를 다루는 방법에 대하여는 지금 다루지 않는다.

```
\ExplSyntaxOn
\NewDocumentCommand \test { m }
{
    \tl_set:Nn \l_tmpa_tl { #1 }
    \tl_map_inline:Nn \l_tmpa_tl
    {
        \fbox { ##1 }
    }
    (\tl_count:N \l_tmpa_tl)
}
\ExplSyntaxOff
\test{a bc def gh}
```

a b c d e f g h (8)

주어진 문제는 글자 수를 세는 것이 관건이다.

인덱스 카운터를 이용하자 `\tl_map`으로 주어지는 토큰을 하나씩 처리하는 상황을 생각한다. 이 때 임의의 정수 `\l_tmpa_int`를 생각하자.

(1) 첫 번째 아이템이 들어오면 `\l_tmpa_int`를 1 증가시킨다.

(2) 첫 번째 아이템을 임시 tl `\l_tmpb_tl`에 적립(`\tl_put_right:Nn`)한다.

(3) 이 과정을 반복해가다가, `\l_tmpa_int`가 3이 되면 카운터를 0으로 만들고 지금까지 적립된 `\l_tmpb_tl`을 출력(입력 스트림에 남김)한 다음 `\l_tmpb_tl`을 비운다(clear).

```
\ExplSyntaxOn
\NewDocumentCommand \test { m }
{
    \int_zero:N \l_tmpa_int
    \test_process:n { #1 }
}

\cs_new:Npn \test_process:n #1
{
```

```
    \tl_set:Nn \l_tmpa_tl { #1 }
    \tl_map_inline:Nn \l_tmpa_tl
    {
        \int_incr:N \l_tmpa_int
        \tl_put_right:Nn \l_tmpb_tl { ##1 }

        \int_compare:nTF { \l_tmpa_int == 3 }
        {
            \tl_use:N \l_tmpb_tl \par
            \int_zero:N \l_tmpa_int
            \tl_clear:N \l_tmpb_tl
        }
        { }
    }
}
\ExplSyntaxOff
\test{abcdefg hijk}
```

abc
def
ghi

위의 예에서 `\int_compare:nTF`에서 F 부분은 아무 것도 하는 것이 없다. 이럴 경우에 F를 생략하고 해당 부분을 적지 않을 수 있다.
이 코드는 다 좋은데 마지막에 남는 두 글자가 (3을 이루지 못했기 때문에) 식자되지 않았다. 이를 처리하기 위해서 다음 코드를 추가한다.

```
    \int_compare:nT { \tl_count:N \l_tmpb_tl != 0 }
    {
        \tl_use:N \l_tmpb_tl \par
    }
```

이제 마지막으로 출력 모양에 대하여 생각해본다. 위의 예에서 매번 `\l_tmpb_tl \par`를 하여 추려낸 것을 한 줄씩 찍었다. 이것을 어떤 tl에 다음과 같은 모양으로 저장하였다가

```
abc & def \tabularnewline
```

tabular 환경 안에서 확장해주면 될 것 같다.
이를 위하여 `\l_tabitem_int`라는 정수(카운터)와 `\l_tablines_tl`이라는 tl을 하나 마련하였다. 다음 코드를 보고 무슨 일이 일어난 것인지 잘 살펴보도록 하여라.

```
\ExplSyntaxOn
\tl_new:N \l_tablines_tl

\NewDocumentCommand \testa { m }
{
```

```
    \int_zero:N \l_tmpa_int
    \tl_clear:N \l_tablines_tl
    \int_zero_new:N \l_tabitem_int

    \testa_fn:n { #1 }

    \begin{tabular}{ll}
    \tl_use:N \l_tablines_tl
    \end{tabular}
}

\cs_new:Npn \testa_fn:n #1
{
    \tl_set:Nn \l_tmpa_tl { #1 }
    \tl_map_inline:Nn \l_tmpa_tl
    {
        \int_incr:N \l_tmpa_int
        \tl_put_right:Nn \l_tmpb_tl { ##1 }

        \int_compare:nT { \l_tmpa_int == 3 }
        {
            \int_incr:N \l_tabitem_int
            \int_if_odd:nTF { \l_tabitem_int }
            {
                \tl_put_right:Nx \l_tablines_tl { \l_tmpb_tl }
                \tl_put_right:Nn \l_tablines_tl { \c_alignment_token }
            }
            {
                \tl_put_right:Nx \l_tablines_tl { \l_tmpb_tl }
                \tl_put_right:Nn \l_tablines_tl { \tabularnewline }
            }
            \int_zero:N \l_tmpa_int
            \tl_clear:N \l_tmpb_tl
        }
    }
    \int_compare:nT { \tl_count:N \l_tmpb_tl != 0 }
    {
        \int_incr:N \l_tabitem_int
        \int_if_odd:nTF { \l_tabitem_int }
        {
            \tl_put_right:Nx \l_tablines_tl { \l_tmpb_tl }
            \tl_put_right:Nn \l_tablines_tl { \c_alignment_token
    \tabularnewline }
        }
        {
            \tl_put_right:Nx \l_tablines_tl { \l_tmpb_tl }
```

```
            \tl_put_right:Nn \l_tablines_tl { \tabularnewline }
        }
    }
}
\ExplSyntaxOff
\testa{ABCD efghi jklmno p}
```

ABC	Def
ghi	jkl
mno	p

인덱스에 대하여 modulo 연산 위의 예에서는 인덱스가 3이 되면 이를 0으로 되돌리는 방법을 사용하였다. 이렇게 하지 않고 인덱스 카운터를 증가시키면서 그 값의 modulo(3) 연산 결과가 0인가를 검사하는 방법이 있다. 이에 대해서는 따로 코드를 보이지 않을 것이니 꼭 실제 해보기를 바란다. `\int_mod:nn`을 쓰면 간단히 된다.

regex로 해보자 expl3가 plainTEX이나 기존 LATEX에 비하여 가진 중대한 장점이 sorting과 regex 관련 함수를 제공한다는 점이다.
시나리오는 간단하다. 인자로 주어지는 문자열을 regex 조작하여 세 글자 이후마다 어떤 표지(예컨대 쉼표)를 붙인 다음에 이를 seq 또는 clist로 처리하는 것이다. 코드가 매우 세련되고 아름다워지는 장점이 있다. regex 함수들이 실행상 약간의 부하가 걸린다는 주장이 있지만 말단 사용자로서는 크게 신경쓸 일이 아니다.

[붙임 1] regular expression이 아닌 문자(열) 바꾸기라면 l3tl 함수인 `\tl_replace_once:Nnn` 이나 `\tl_replace_all:Nnn`을 쓰면 된다.

sed로 'abcdefghij'에 대하여 세 글자마다 쉼표를 붙이려면

```
$ sed 's/\(.\)\(.\)\(.\)/\1\2\3,/g' < file_in.txt
```

이와 같이 한다. expl3의 l3regex로 간단히 다음과 같이 할 수 있다.

```
\tl_set:Nn \l_tmpa_tl { abcdefgh ijk }
\regex_replace_all:nnN { (.)(.)(.) } { \1\2\3 , } \l_tmpa_tl
```

또는 더 간단하게

```
\ExplSyntaxOn
\tl_set:Nn \l_tmpa_tl { abcdefghijk }
\regex_replace_all:nnN { (.){3} } { \0 , } \l_tmpa_tl
\tl_use:N \l_tmpa_tl
\ExplSyntaxOff
```

abc,def,ghi,jk

이제 `\l_tmpa_tl`에 들어 있는 것은 쉼표로 분리된 토큰열이므로 clist를 사용할 수 있겠다. 물론 쉼표가 아닌 다른 표지를 넣을 수 있으며, 이 때는 seq로 처리하는 것이 어렵지 않다. 분리 표지를 |로 하고 seq로 같은 일을 하도록 하는 코드를 연습해보라.

tabular를 위해서 행하는 조작 부분을 직접 해보는 즐거움을 위해 남겨두고 여기서는 해당 항목을 출력한 후 개행하는 방식의 짧은 코드만 보이기로 한다. (항목만을 출력하고 개행하는 아래 코드는 사실 `\clist_map_...`하지 않아도 간단히 `\clist_use:Nn`으로 처리할 수 있다. 그러나 아래 코드를 그대로 남겨두는 이유는 이 부분에 tabular를 만들기 위한 조작을 넣어야 하기 때문이다.)

```
\ExplSyntaxOn
\NewDocumentCommand \testc { m }
{
    \testc_fn:n { #1 }
}

\cs_new:Npn \testc_fn:n #1
{
    \tl_set:Nn \l_tmpa_tl { #1 }
    \regex_replace_all:nnN { \s } { } \l_tmpa_tl
    \regex_replace_all:nnN { (.){3} } { \0, } \l_tmpa_tl
    \clist_set:No \l_tmpa_clist { \l_tmpa_tl }

    \clist_map_inline:Nn \l_tmpa_clist
    {
        ##1 \par
    }
}
\ExplSyntaxOff

\testc{abcd efgh ijk lmno}
```

```
abc
def
ghi
jkl
mno
```

`\regex_...` 명령이 두 번 쓰였는데 첫 번째 것은 space 문자를 제거하는 것이다. regex 명령에 들어오는 tl 안의 모든 문자가 보존되므로 이와 같이 하여 스페이스를 제거할 필요가 있다.
한편, `\clist_set:No`가 사용되었다. 다음 두 명령은 그 의미가 같다.

```
\clist_set:No \l_tmpa_clist { \l_tmpa_tl }
\clist_set:NV \l_tmpa_clist \l_tmpa_tl
```

요컨대 `\l_tmpa_tl`이라는 매크로 자체를 취하지 말고 그것을 확장하여 그 값(*value*)을 취하라는 것이다.

연습문제

기본 1. 명령 `\acmd`는 두 개의 인자를 받는다. 첫 번째 인자는 숫자이며 두 번째 인자는 임의의 문자열이다. 만약 문자열이 지정된 숫자보다 크다면 앞에서부터 숫자에 해당되는 번째 문자까지만 출력하라. 만약 문자열이 지정된 숫자보다 작다면 문자열의 앞쪽에 _(언더스코어)를 붙여 n개의 문자열이 되도록 하라.

발전 2. Python에는 문자열을 자르는(슬라이싱) 재미있는 기법이 있다. `\myslicing` 명령을 정의하되, 3개의 인자를 받아들이도록 하여 첫 인자로 주어지는 문자열을 #2부터 #3까지 슬라이싱하여 (즉 `mystring[`m`:`n`]`과 비슷) 출력하도록 하여라. 스페이스는 무시한다.

발전 3. 새로운 명령 `\myitemswap`을 정의한다. 이 명령은 네 개의 인자를 받아들이며 첫 번째 인자가 문자열이다. 두 번째와 세 번째는 숫자인데, 주어지는 문자열의 아이템 번호들이다. 마지막 네 번째 인자는 임의의 매크로를 받는다. 주어진 문자열에서 #2번째 항목과 #3번째 항목을 교환(swap)하여 네 번째로 주어진 매크로에 넣어 반환하라. 숫자가 문자열의 범위를 벗어날 때의 에러처리 코드를 포함하라.

---

1. 입력: `\acmd{5}{beautiful}\quad \acmd{5}{abc}`
   출력: beaut __abc
2. 입력: `\myslicing{Hello world}{3}{7}`
   출력: llowo
3. 입력: `\myitemswap{abcde}{2}{4}{\myresult}`
   출력: `\myresult` → adcbe

문제

다음 실행의 결과가 어떠할지 예측해보아라. 실제로 예상과 같은지 확인해보아라.

```
\ExplSyntaxOn
\tl_new:N \l_tmpc_tl

\tl_set:Nn \l_tmpa_tl { foo }
\tl_set:Nn \l_tmpb_tl { \l_tmpa_tl }
\tl_set:No \l_tmpc_tl { \l_tmpa_tl }

\tl_set:Nn \l_tmpa_tl { bar }

\tl_use:N \l_tmpb_tl
\tl_use:N \l_tmpc_tl
\ExplSyntaxOff
```

3 확장(expansion)의 기초

이 코드를 실행하면 다음과 같은 결과가 나온다.

```
\ExplSyntaxOn
\tl_use:N \l_tmpb_tl \par
\tl_use:N \l_tmpc_tl
\ExplSyntaxOff
```

bar
foo

이유는 `\l_tmpb_tl`에는 `{\l_tmpa_tl}`이 들어 있지만 `\l_tmpc_tl`에는 set 명령이 실행되는 시점에서 `\l_tmpa_tl`이 가지고 있던 값이 해동되어서 들어가 있기 때문이다. 추후 `\l_tmpa_tl` 값을 바꾼다면 이 매크로 자체를 가지고 있는 `\l_tmpb_tl`은 당연히 달라진 값을 식자하겠지만 `\l_tmpc_tl`은 그 영향을 받지 않는다.
expl3의 함수 인자형 지시자가 번거롭기만 하고 TMI가 아니냐는 의견이 있는데 전혀 그렇지 않다. expl3로 코딩하는 프로그래머는 자신이 사용하고 있는 매크로(변수)가 특정 시점에서 확장되어야 할지 그렇지 않은지를 항상 세심하게 유념하여야 한다. 또한 인자를 확장시키는 다양한 방법이 존재하는데, 이에 대해서 다음 기회에 더 자세히 다룬다.

다른 한 예를 들어보자. 앞서 tabular 안에 자신이 수집한 매크로를 바로 넣는 방법, 즉

```
\begin{tabular}{ll}
\tl_use:N \l_tablines_tl
\end{tabular}
```

와 같은 코드를 생각해보자. expl3 이전이면 이런 일을 어떻게 하였을까?

```
\makeatletter
\protected\def\tnewlinehline{\tabularnewline\hline}
\def\@tablines{abc & def \tnewlinehline }
\protected@edef\@tablines{\@tablines LMN & OPQ \tnewlinehline }
\protected@edef\@tablines{\@tablines rst & uvw \tnewlinehline }
```

```
\begin{tabular}{l|l}
\hline
\@tablines
\end{tabular}
\makeatother
```

abc	def
LMN	OPQ
rst	uvw

이것이 전통적으로 사용되어온 기법이다. (혹시 쓰일 데가 있을지 모르니 알아두는 것도 좋다.) 이 코드에서 주의할 것은 두 가지인데, 하나는 `\hline`이 fragile한 명령이라서 이것을 `\@tablines`에 직접 집어넣으면 에러가 뜬다는 것이다. 그래서 `\protected\def`으로 한 번 묶어주었다.
그리고 `\protected@edef\@tablines{\@tablines...` 부분을 유의해서 보아야 한다. 즉 기존의 `\@tablines`를 한 번 확장한 다음 그 뒤에 오는 것을 추가하였다.
이 코드에 대응하는 expl3는 다음과 같다.

```
\ExplSyntaxOn
\tl_set:Nn \l_tablines_tl { abc \c_alignment_token def \tabularnewline
    \hline }
\tl_put_right:Nn \l_tablines_tl { LMN \c_alignment_token OPQ \tabularnewline
    \hline }
\tl_put_right:Nn \l_tablines_tl { rst \c_alignment_token uvw \tabularnewline
    \hline }

\begin{tabular}{l|l}
\hline
\tl_use:N \l_tablines_tl
\end{tabular}
\ExplSyntaxOff
```

abc	def
LMN	OPQ
rst	uvw

번거로운 `\protected@edef` 대신 `\tl_put_right:Nn`을 쓰면 된다는 것을 알겠다. `\hline`은 특히 확장되지 않고 들어가도록 (즉 n 인자로) 조심하는 것이 좋다.

`\use:c`에 관하여 `\tl_set:Nn` 대신 `\tl_set:cn`을 쓰면 다음과 같은 일을 할 수 있다.

```
\ExplSyntaxOn
\int_zero:N \l_tmpa_int
\int_incr:N \l_tmpa_int
```

```
\tl_set:cn { mycmd \int_to_Alph:n { \l_tmpa_int } } { result~=~1 }
\int_incr:N \l_tmpa_int
\tl_set:cn { mycmd \int_to_Alph:n { \l_tmpa_int } } { result~=~2 }
\ExplSyntaxOff
\mycmdA, \mycmdB
```

result = 1, result = 2

plainTEX에서 다음과 같이 하던 것에 해당한다.

```
\newcount\mycnt
\mycnt=1
\expandafter\def\csname mycmd\romannumeral\mycnt\endcsname
{ result 1 }
\advance\mycnt by1
\expandafter\def\csname mycmd\romannumeral\mycnt\endcsname
{ result 2 }

\mycmdi, \mycmdii
```

result 1 , result 2

연습문제

기본 1. 주어지는 문자열을 앞에서부터 3개째마다 쉼표를 추가하는 명령을 작성하여라.

입력: `\test{this is just a test}`
출력: thi, sis, jus, tat, est

힌트: 이 문제는 앞서 그 해법이 이미 다루어졌다. 다시 연습문제로 내는 이유는 만약 입력되는 문자열의 개수가 3의 배수이면 마지막에 쉼표가 붙을 수 있는데 이것이 붙지 않도록 하라는 것이다.

문제

새로운 명령 \newcmd는 다음과 같은 형식으로 실행한다.
`\newcmd{abc, de, fgh, i, jkl}`
쉼표로 분리된 각 항목을 enumerate으로 배열하여라.

입력: `\newcmd{abc, de, fgh, i, jkl}`
출력:

1. abc
2. de
3. fgh
4. i
5. jkl

이 문제는 KTUG QnA:236820에서 질문과 답변이 이루어진 것이다. 해당 글타래에 expl3를 사용하는 답변이 없다. 지금까지 학습한 것으로 문제를 해결할 수 있을 것이므로 이를 작성하여 보아라.

4 리스트의 소팅

여기서 배우고자 하는 것은 sorting이다. 숫자로 된 리스트의 소팅은 다음과 같이 한다. (이하 clist만을 다루는데 seq에 대해서도 마찬가지로 동작한다.)

```
\ExplSyntaxOn
\clist_set:Nn \l_tmpa_clist { 29, 8, -3, 6, 12 }
\clist_sort:Nn \l_tmpa_clist
{
    \int_compare:nTF { #1 > #2 }
    { \sort_return_swapped: }
    { \sort_return_same: }
}
\clist_use:Nn \l_tmpa_clist {,~}
\ExplSyntaxOff
```

-3, 6, 8, 12, 29

`\..._sort:Nn` 함수는 그 정의에서 #1과 #2를 비교한다. 이것이 다른 함수 정의 안에서 사용될 적에는 ##1과 ##2가 되는 것에 주의하여야 한다.
문자열의 소팅은 어떻게 할 것인가? 다음처럼 하는 것이 한 가지 방법이다.

```
\ExplSyntaxOn
\clist_set:Nn \l_tmpa_clist { dog, tiger, lion, sheep, cat, mouse }
\clist_sort:Nn \l_tmpa_clist
{
    \int_compare:nTF { \pdftex_strcmp:D { #1 } { #2 } > 0 }
    { \sort_return_swapped: }
```

```
    { \sort_return_same: }
}
\clist_use:Nn \l_tmpa_clist { ,~ }
\ExplSyntaxOff
```

cat, dog, lion, mouse, sheep, tiger

여기서 "Do not use"라고 하는 `\pdftex_strcmp:D`를 사용하였는데 만약 이를 사용하지 않으려면 좀더 복잡한 코딩이 필요하기 때문에 간편하게 이를 빌어 쓰기로 하였다. (참고로 LuaTEX에서는 `l3kernel.strcmp`라는 함수를 사용할 수 있다.)
어떤 clist (book, house, building, beer, cat)를 사전순(alphabetically)으로 소트할 것이 아니라 문자열 길이를 기준으로 소팅하고 싶다면 어떻게 해야 할까? 연습문제 삼아 풀어보기 바란다. 결과는 다음과 같이 나온다. (만약 길이가 같으면 알파벳순으로 정렬하도록 2중 조건을 적용하였다.)

```
\ExplSyntaxOn
\clist_use:Nn \l_tmpa_clist { ,~ }
\ExplSyntaxOff
```

cat, beer, book, house, building

tl의 각 항목도 정렬할 수 있다. 즉 `\tl_sort:Nn`를 이용할 수 있다.
소팅은 요긴하지만 LATEX 프로그래밍의 관점에서는 아주 예외적인 상황에서 필요하다. 실제 문헌목록이나 인덱스 등에서의 소팅은 makeindex, bibTEX 프로그램이 하는 것이므로 이 경우와는 같지 않다.

연습문제

기본 1. 새로운 명령 `\newcmd`는 다음과 같은 형식으로 실행한다.
`\newcmd{this is just a test}`
주어지는 인자를 먼저 세 개마다 쉼표를 붙여 분리하고, 분리된 각 단어를 `\resi`, `\resii`, `\resiii`, `\resiv`, ...에 넣어 반환하라.

발전 2. 인자로 주어지는 단어의 각 문자가 몇 번씩 사용되었는지를 예시와 같이 출력하여라.

1. 입력: `\newcmd{this is just a test}`
   출력: `\resi` → thi, `\resiv` → tat
2. 입력: `\testcmd{abaracadabra}`
   출력: a = 6, b = 2, c = 1, d = 1, r = 2

Nova de Hi.