Reading *TEXbook*

작은나무

2006년 3월 30일

차 례

1 control sequence 구별

작은나무 2006-03-04T11:27:17

9–11쪽: plain TeX에는 900여개의 control sequence(cs)가 있다고 합니다. 그 중 300여개는 “primitive”이고, 나머지 600여개는 macro라고 합니다. primitive는 더 이상 전개 하 수 없는 최소의 실행 단위 cs이고, macro는 primitive 혹은 다른 매크로 들로 구성된 cs라네요. 그렇다면, 주어진 cs가 primitive인지 macro인지 어떻게 구분할 수 있을까요? Knuth는 두가지 구분 방법을 귀띔해줍니다. 첫번째 방법은 그 책의 인덱스중 cs를 나열해 놓은 곳을 보면 ‘*’가 붙은 cs가 있는데, 바로 그것들이 primitive라고 하네요.(가장 간단한 방법이지만 책 구입에 돈이 든다는 단점이 있습니다.) 두번째 방법이 그럴듯한데, `\show`라는 primitive로 판별하는 방법입니다.
예를들어 TeX을 실행시키고 `\show\thinspace`라고 입력하면 `\thinspace`는 macro라는 표시와 그것의 정의가 주루룩 나옵니다. (재미있습니다.)

2 futurelet

작은나무 2006-03-04T11:27:18

207쪽: 제20장(Definitions)를 공부하다가 요상한 cs를 하나 발견했습니다. `\futurelet`. 도대체 이것을 어디에 사용해야 잘 사용했다고 소문이 나는지 모르겠습니다. “미래`\let`”(?) 도대체 이런 primitive를 왜 만들었을까요? 207쪽 중간에 이 cs의 설명이 있는데, ”dangerous bend” 기호가 두 개나 붙어있습니다. 
언젠가는 이해할 날이 오겠죠.

“`\futurelet\cs<token1><token2>`”는 “`\let\cs=<token2><token1><token1>`” 이랑 같다고 하는데, 그렇다면 문법은 이해가 가는데, 의미가 영 와닿지 않네요.

▶ `\futurelet\cs<token1><token2>`는 `\let\cs=<token2><token1><token2>` *와 동일합니다. – DohyunKim (2006-03-29T17:04:35)*

3 숫자 뒤 공백

작은나무 2006-03-05T05:15:06

208쪽: ”For best result, ”always put a blank space after numeric constant;””

`\count` 레지스터와 같은 숫자 상수 뒤에 반드시 공백을 하나 두라는 것인데,

```
\def\statement{\ifnum\balance=0 \fullypaid
  \else\ifnum\balance>0 \overpaid
          \else\underpaid
          \fi
  \fi}
```

위의 예에서 첫번째줄 0과 \fullypaid 사이의 공백이 매우 중요하다고 합니다. 만약

```
...=0\fullypaid...
```

처럼 한다면 전개되지 않아도 될 \fullypaid가 전개되기 때문이라네요. 즉 결과에는 큰 영향을 미치지 않지만, 쓸데 없는 전개를 하므로 실행시간이 느려져서 비효율적이라고 합니다. (간혹, 괜한 전개로 에러가 발생하기도 합니다.) 숫자 상수 뒤에 공백을 넣음으로 TeX으로 하여금 여기까지가 숫자 상수라고 확실히 못박아둘 수 있습니다.

마지막 Knuth의 말이 재미있습니다. 자신은 독자에게 반드시 숫자 상수 뒤에 공백을 넣으라고 하면서, 정작 본인은 가끔 공백을 넣지 않는답니다. 그 이유인즉

> *because extra spaces do look funny in the file; aesthetics are more important than efficiency*

Knuth, 이 할아버지 재미있는 분임에 틀림없습니다.:))

4 rlap, llap

작은나무 2006-03-09T15:07:34

82쪽: If you understand boxes and glues, you're ready to learn the **rlap** and **llap** macros of plain TeX; these names are abbreviations for "right overlap" and "left overlap"

평소에 \rlap과 \llap 매크로를 사용하면서 참 재미있는 매크로구나 하고 생각했는데, 마침 그것들에 대한 내용이 있네요.

\rlap을 구현하기 위해서 위에 설명한 의미에 충실히 하여 조금만 생각하면,

```
\def\rlap#1{{\setbox0=\hbox{#1}\copy0\kern-\wd0}}
```

위와 같은 정의를 얻을 수 있는데, 사실은 이런 복잡한 계산은 필요 없다고 합니다.

```
\def\rlap#1{\hbox to 0pt{#1\hss}}
```

아름답지 않습니까? 위 정의를 보고, 잠시 음미할 시간을 갖는 것도 괜찮을 듯 합니다.

그렇다면, `\llap`은 어떻게 정의 하면 될까요?

```
\def\llap#1{\hbox to 0pt{\hss#1}}
```

5 TeX logo Texture

작은나무 2006-03-11T10:47:16

225쪽: TeX 로고로 도배를 한 재미있는 텍스쳐(TeXture)가 있습니다.

도배하는 방법은 아래와 같습니다.

```
$$\hbox to 4.5in%
  {\cleaders \vbox to .5in{\cleaders\hbox{\TeX}\vfil}\hfil}$$
```

참 간단합니다! 단 한 줄의 벽지로 위와 같은 멋진 도배를 할 수 있습니다. 이것이 TeX의 힘이고, 우리들이 TeX을 사용하는 이유가 아닌가 합니다.

6 hanging paragraph

작은나무 2006-03-14T14:34:10

102쪽: `\hangindent=<dimen>` 와 `\hangafter=<number>`. plain TeX에서는 굉장히 많이 사용되는 control sequence 들 입니다.

x와 n을 각각 `\hangindent`와 `\hangafter`의 값이라고 하고, h를 `\hsize`의 값이라고 하자. 그때, $n \geq 0$이면, hanging 들여쓰기는 글 문단의 $n+1, n+2, \ldots$

번째 줄에서 나타나고, $n < 0$이면, $1, 2, \ldots |n|$ 에서 나타납니다. Hanging 들여쓰기란 각 줄의 폭이 일반적인 폭인 h가 아니라 $h - |x|$ 인 것을 뜻합니다; 만일 $x \geq 0$이면, 각 줄들은 왼쪽에서 들여쓰기가 되고, 그렇지 않으면 오른쪽 들여쓰기가 됩니다. 무슨 말인지 감이 오십니까? 상황이 이러하니, 참고서가 필요하겠지요?

동해물과 백두산이 마르고 닳도록 하느님이 보우하사 우리나라 만세. 무궁화 삼천리 화려강산 대한사람 대한으로 길이 보전하세. 동해물과 백두산이 마르고 닳도록 하느님이 보우하사 우리나라 만세. 무궁화 삼천리 화려강산 대한사람 대한으로 길이 보전하세.

다음과 같은 입력으로 위의 결과를 얻을 수 있습니다.

```
\hangindent=2cm \hangafter=-2 \noindent
동해물과 백두산이 마르고 닳도록 하느님이 보우하사 우리나라 만세.
무궁화 삼천리 화려강산 대한사람 대한으로 길이 보전하세.
동해물과 백두산이 마르고 닳도록 하느님이 보우하사 우리나라 만세.
무궁화 삼천리 화려강산 대한사람 대한으로 길이 보전하세.
```

7 penalty

작은나무 2006-03-17T15:16:51

96–97쪽: 문단(paragraph)을 쪼개어 줄(line)들을 만드는 과정에는 매우 흥미롭고, 과학적인 방법이 숨어 있습니다. 그 방법을 이해하기 위해서는 몇가지 개념과 그 개념에 해당하는 용어를 알아야 합니다. 먼저 "penalty"에 대해서 알아보겠습니다. 문단에서 줄들을 만들기 위해서 줄바꿈하는 지점을 breakpoint라고 합니다. breakpoint를 선정하는 방법은 문단을 줄바꿈하여 만들어진 줄들이 최대한 우리 눈에 보기 좋은 지점으로 하는 것입니다.

각 breakpoint들의 후보들은 penalty라고 불리는 어떤 값을 가지고 있는데, penalty는 그 지점에서 줄바꿈으로 해서 발생하는 미학적 비용(aesthetic cost)입니다. 예를 들어 어떤 breakpoint의 penalty가 100이라면(`\penalty100`), 그 지점에서 줄바꿈으로 발생하는 미학적 비용이 100만큼 든다는 말입니다. 즉 penalty가 크면 클수록 비용이 많이들므로, 줄바꿈이 일어날 확률이 적어 질 것이고, penalty가 작아서 음수가 된다면, 음수의 penalty는 양수의 bonus 이므로, 그 지점에서는 줄바꿈이 발생할 확률이 크겠지요.

TeX에서의 10000이라는 숫자는 무한대를 뜻합니다. 그래서 `\penalty10000` 이라는 뜻은 줄바꿈의 비용이 무한대로 들어간다는 소리이므로, 절대로 줄바꿈이 일어나지 않는다는 뜻입니다. 즉 `\nobreak` 이지요. 제가 이해하는 penalty는 이렇습니다.

8 badness and tolerance

작은나무 2006-03-18T16:33:08

28–29, 96–97쪽: penalty 설명에서 breakpoint의 설정은 만들어진 줄들이 최대한 우리 눈에 보기좋게 하는 곳으로 설정된다고 했습니다. 그렇다면, 어떤 줄들이 보기 좋은 것일까요? 그리고 그 기준은 무엇일까요? TeX은 이에 대한 수학적 기준으로 badness라는 것을 도입했습니다. 각각의 줄은 badness라는 값을 가지고 있습니다. badness는 주어진 줄에서 단어들 사이의 간격을 수치화한 값입니다. badness 굳이 우리말로 하자면 '나쁨정도'쯤 되므로, 그 값이 작을 수록 좋은 것이겠지요. 그래서 어떤 줄의 badness가 0일때, 그 줄의 단어들 사이의 간격이 매우 적절하여 우리 눈에 퍼펙트하게 보입니다. 따라서 badness는 0과 무한대인 10000 사이의 값을 갖습니다. badness를 구하는 식은 대략 $\min(100r^3, 10000)$입니다. 여기서 r은 어떤 비율(ratio)을 나타내는데, 자세한 설명은 생략하겠습니다. (자세한 수식보다는 badness의 의미를 파악하는 것이 더 중요합니다.)

Plain TeX에서는 각 줄의 badness가 200을 넘어서는 안되도록 정하고 있습니다. 이 말은 Plain TeX은 자기 기준에서는 badness가 200을 넘으면 그 줄에서 단어들 사이의 간격이 너무 좁거나, 넓어서 안 예뻐보이므로, badness가 200이 될 때 까지는 참아줄 수 있다는 것입니다. badness가 200을 넘는 순간 Plain TeX은 그 줄의 단어 사이의 간격들을 차마 눈뜨고 볼 수 없을 정도여서, 더이상 참지 못하고, 못생긴 줄이라고 불평을 합니다.

Plain TeX은 "나는 badness가 200까지 참을 수 있어."는 표시로 `\tolerance=200`이라고 정합니다. primitive `\tolerance`의 뜻을 아시겠죠? 한가지 더 알아 두어야 할 것은 영어에는 hyphenation을 이용해서 영어 단어가 두 줄에 걸쳐 나올 경우, 그 단어를 쪼갤 수 있습니다. Plain TeX은 일단 hyphenation을 이용하지 않고, 줄을 만들 수 있는지 먼저 체크합니다. 이처럼 먼저 확인 한다는 의미에서 접두어 'pre'를 사용하여, 이때의 badness의 값을 `\pretolerance`로 설정합니다. Plain TeX은 `\pretolerance=100`으로 정합니다. 즉, 먼저 badness 100으로 정하고 영어 단어를 쪼개는 일 없이 줄들을 만들수 있는지 체크하고, 안된다면 hyphenation을 이용하면서 badness를 200으로 하고 다시 확인합니다.

9 demerits

작은나무 2006-03-19T00:30:28

97–98쪽: 이제 문단에서 줄들을 만들어내는 과정을 어느정도 정리할 때가 된 것 같습니다. 지난 두번에 걸쳐서 쓴 내용을 정리하면, TeX은 각 줄의 badness가

주어진 tolerance를 넘지 않도록 하면서 breakpoint를 설정합니다. 이 과정을 좀 더 수학적으로 표현하면, 각 줄들은 "*demerit*"라는 것을 가지고 있는데, *각 줄들의 demerits의 합이 최소가 되도록 줄들을 만듭니다.* demerit들의 총합이 최소가 될때, 그 문단의 줄들이 보기 좋은가 봅니다. 그렇다면 demerit가 무엇이냐?, 저도 이러고 싶지는 않습니다만, 어쩔 수 없이 수식을 좀 써야겠습니다. 각 줄의 demerit는 아래의 수식과 같이 계산합니다. (TeX에서 수식을 조판하는 일은 언제나 즐겁죠.:))

$$d = \begin{cases} (l+b)^2 + p^2, & \text{if } 0 \le p < 10000; \\ (l+b)^2 - p^2, & \text{if } -10000 < p < 0; \\ (l+b)^2, & \text{if } p \le -10000. \end{cases}$$

d는 demerit, b는 badness, p는 penalty, 마지막으로 l은 linepenalty를 나타냅니다. linepenalty는 말그대로 line이 갖는 penalty입니다. 그래서 이 값을 크게하면, TeX으로 하여금 문단에서 가능한한 최소 갯수의 줄들을 만들어 내도록 힘들게 한답니다. Plain TeX에서의 linepenalty의 기본 값은 10입니다. 예를들어, 어떤 줄의 badness가 20이고, breakpoint가 glue에서 일어났다면, penalty가 없는 것이므로, 그 줄의 demerit는 위 식에 의해서 $(10+20)^2 = 900$이 됩니다. 이 말에서 볼때, penalty는 줄에 기본적으로 주어지는 값이 아닌, 어떤 독립적인 값인가 봅니다.(glue에 대해서는 언제 언급할 날이 있을것입니다. 그 놈이 아주 집착이 강한 놈이거든요.)

마지막으로 Knuth는 우리들이 수학을 싫어하는 것을 잘 알아서 위 수식을 알아 듣기 쉬운 말로 설명을 합니다. 즉, 처음에 언급한 *각 줄들의 demerits의 합이 최소가 되도록 줄들을 만든다*는 의미는 basness들과 penalty들의 제곱의 합을 최소화하는 것과 같은 의미인데, 이 뜻은 그 문단의 모든 breakpoint들에 걸쳐서, 어떤 한 줄의 최대 badness 또한 최소화 한다는 의미입니다. badness가 최소화 된다는 것은 그만큼 우리 눈에 보기 좋다는 것이니까요. 이상으로 문단에서 줄들을 만들어 내는 과정을 살펴봤습니다. 좀 어려운 면도 있었지만, 재미있었습니다. TeX을 좀 더 알게되어 TeX과 점점 친해져 가는 느낌입니다. TeX! 우리 좀 더 친해집시다. :) "텍도없는 소리 말라고?, 아직 멀었다구?'' :(

▶ *이 곳 위키에서는 TeX으로 수식 조판이 되는 것으로 알고있는데요, 실제로도 몇번 사용해 봤고요. 그런데, 위 수식을 시도했더니, 수식 중간에 이상한* `##Blog` *가는 것이 삽입되면서, 수식이 박살이 났습니다. "미리보기"할 때도 세줄의 수식을 감싸는 왼쪽 괄호"{"가 안나왔지만 그것은 그런대로 참을만하여 (저의 쯤 되나봅니다. :)) "저장"을 눌렀더니, 이상하게 나오더군요. 그래서 그러고 싶지는 않았지만, 그림으로 대체했습니다. 아마도 제가 LaTeX에서는 수식 조판*

을 해 본일이 없어서, 실수가 있었을지도 모릅니다. :) – Anonymous (2006-03-19T00:50:43)

▶ *저 혼자 텍북을 읽을 때보다 훨씬 재미있습니다. 사실 저는 이 책을 편식했습니다. 대충 둘러보고 작년에서야 16–19장을 집중적으로 읽었거든요. 의미도 모르는 채 익숙했던 것들 조금씩 와 닿는 것 같습니다. —badness, demerit 등. 참, TeX에서 수식을 입력하는 것은 언제나 즐겁다는 것에 공감합니다. :) – [Progress] (2006-03-20T04:39:13)*

10 long, outer, global, gdef

작은나무 2006-03-20T07:56:22

저는 TeX 디자이너나 TeXnician이 되고자 하는 마음은 눈꼽 만큼도 없지만, 가끔 plain.tex, lplain.tex과 같은 봐서는 안 될 파일들을 열어봅니다. 사실 저 같은 TeX 초보자들은 그 같은 파일들은 열어 볼 필요도 없고, 열 이유도 없지만, 그저 호기심에 열어봅니다. 하지만, 간단한 명령어(control sequence) 하나로 여러 다양한 기능을 하는 것을 보면, 그 원리가 궁금해지는 것도 사실입니다. 그래서 한 번 읽어는 보는데... 어렵습니다. 특히, 매크로 정의 할때 쓰이는 `\def` 앞에는 뭐가 그리도 많이 붙는지..., 예를들면, `\long`, `\global`, `\outer` 더구나 매크로 정의할때도 `\def` 외에 뭐가 그리도 많은지, `\edef`, `\gdef`, `\xdef`, `\chardef`, `\csname`, `\let`, ... 이 외에도 `\expandafter`, `\noexpand` ... 쩝. 그래서 하나씩 알아보기로 굳게 맘먹었습니다. `\def`는 대충 아니까...

205–206쪽: `\def`를 이용해서 매크로 정의를 할때, 실수로 괄호 '{'나 '}'를 빼먹거나 더해서, 여닫는 괄호가 매치가 안될 경우가 있습니다. 이 경우 그러한 `\def`로 정의된 명령어를 전개할 때, TeX이 충직해서 언제 이 놈의 정의가 어디서 끝나는지 몰라서 계속 처리한다면, 아마도 TeX은 입렵 파일의 끝을 보거나 아니면, 그 전에 TeX이 계속 입력되는 토큰을 받아들여 메모리가 부족해서 컴퓨터가 먹통이 될지도 모릅니다. `:(` 단순한 타이핑 에러로 그러한 사태까지 가면 안되죠. 단지 '{'나 '}' 하나 빼먹을 뿐인데요. 다행히 TeX은 그다지 충직한 편이 못되고 여우같은 면이 있어서, 이런 경우에 대비해서 나름대로의 규칙을 가지고 있습니다. 그 규칙은 `\def`의 argument에는 `\par` 토큰을 사용할 수 없다는 것입니다. 명시적으로 여러분이 "나 `\par`를 arguement에서도 사용하겠으니, 군소리 말고 하라는대로 하라"라고 말하지 않는 이상 말입니다. 그래서 TeX은 매크로를 전개해 나가다가 `\par`를 만나면, "runaway argument"라고 말하고, 하던 일을 그만둡니다. 똑똑합니다. `:)`

하지만, 매크로를 만들다보면, argement에 `\par` 토큰을 사용할 일이 있답니다. 그 경우 어쩌나요? 그때 사용하는 것이 바로 `\def`앞에 붙여주는 `\long`

입니다. 마치 TeX한테 "네가 앞으로 전개할 명령어의 argument가 길 수도 있으니까, 게의치 말고, 그냥하라"라는 뜻인것 같습니다. (이것으로 하나 해치웠습니다. 위에서 언급한 정복해야 할 control sequence 목록에서 두 줄로 찍찍그어 `\long`은 지우세요.:))

위에서 설명한 '`\par`-금지 법칙' 만으로는 좀 모자란 감이 있습니다. 오로지 계속 전개해 나가다가 `\par`이 나와야만 멈추니까 말입니다. 더 빨리 TeX으로 하여금 "너가 잘못된 명령어를 전개하고 있다"라고 알려주어야 할때가 있습니다. 주로 TeX이 하이 스피드로 토큰들을 처리해야 하는 경우가 그 때입니다. 이 경우에 TeX은 `\par`가 나올때 까지도 못기다립니다. 따라서 잘못된 명령어는 아니지만, 빠른 스피드를 요하는 곳에서는 이 명령어를 사용하지 말라는 의미로 `\long` 처럼 `\def` 앞에 붙이는 것으로 '`\outer`'가 있습니다. 즉 특정한 상황에서 TeX이 `\outer`가 앞에 붙은 명령어를 전개하려 한다면, TeX은 전개해 보지도 않고, 하던 일을 멈추고 불평을 해댑니다. 그렇다면, TeX이 매우 빠르게 처리해야하는 경우, 즉 `\outer`가 붙은 명령어가 쓰이면 안되는 경우가 어떤 경우 일까요?

- argument 안에서
- 어떤 매크로 정의의 parameter text 또는 replacement text 안에서
- 표나 배열할때 사용하는 alignment의 preamble 안에서
- 조건에 의해서 실행이 안될 수도 있는 조건문에서 (if... then ... else...)

만약 위와 같은 환경안에서 `\outer`가 붙은 명령어를 사용한다면 TeX은 "runaway" 상황이거나, "incomplete" 조건문이라고 불평하며, 하던 일을 관둡니다.

마지막 `\global`이 남았습니다. 컴퓨터 프로그래밍 관점으로 보면, 전역변수 설정하는 것이네요. 특정 그룹 안에서 정의된 매크로는 그 그룹안에서만 효력이 있고, 그 그룹 밖에서는 그 효력을 읽는데, 그룹 밖에서도 그 의미를 유지하려면 `\def` 앞에 `\global` 이라고 붙여주면 됩니다. 그리고 `\gdef`는 `\global\def`와 동일한 의미입니다. 지금까지 알아본 `\long`, `\outer`, `\global`은 모두 `\def` 앞에 붙는 접두사 같은데, 그 순서는 아무렇게나 와도 되고, 또한 같은 접두사가 하나 이상이 와도 됩니다. 예를들어,

```
\long\outer\global\long\def
```

위의 선언은 `\outer\long\gdef`와 같은 의미입니다.

11 let

작은나무 2006-03-21T01:06:26

206–207쪽: TeX에서 편리하면서도 중요한 명령어 중의 하나가 바로 `\let`입니다. 이는 plain.tex이나 lplain.tex에서 숱하게 쓰입니다. `\let`은 다음과 같은 형식을 취하는데,

```
\let\cs=<token>
```

이는 `\cs`에 `<token>`이 현재 가지고 있는 의미를 부여한다는 뜻입니다. 재미있는 점은 `<token>`이 다른 명령어(control sequence)토큰 일때 인데, 이 때도 마찬가지로 그 명령어가 가지고 있는 기능 그대로를 `\cs`에게 물려줍니다. 예를들어, '`\let\a=\def`'라고 하고, '`\a\b...{...}`'라고 하면, `\a`가 `\def`의 의미와 기능을 물려받아서, 매크로 `\b`를 정의하는 것이 됩니다. 또, 아래의 의미는

```
\let\a=\b \let\b=\c \let\c=\a
```

명령어 `\b`와 `\c`의 의미를 바꾸는 것이 됩니다. 그리고, 다음과 같은 문장은

```
\outer\def\a#1.{#1:}
\let\b=\a
```

'`\outer\def\b#1.{#1:} \let\a=\b`'와 완전 동일합니다.

`\let`에 사용된 `<token>`이 단일 문자(single character)이고, 그 문자가 character code와 category code의 쌍이라면, 어느 정도까지는 그 문자와 같은 역할을 하지만, 약간 차이점이 있다네요. 예를들어, '`\let\zero=0`'라면, `\zero`는 숫자 0의 의미를 갖지만, 0이 그렇듯이, `\zero`는 숫자 상수 안에서는 사용될 수 없습니다. 왜냐하면, TeX은 숫자 상수에 매크로가 쓰였을 경우, 그 매크로를 전개하고 나면 매크로들이 모두 숫자들로 바뀌어야 하는데, `\zero`는 매크로가 아니기 때문에 전개가 되지않으므로, 숫자 상수안에서는 사용될 수 없다고 합니다. 무슨 뜻인지 알듯 말듯 하지만, 저자의 말대로라면, 나중에 다시 살펴볼 기회가 있다네요. 믿어야죠.

책에 나오는 문제 하나만 풀어봅시다.

- **Q:** '`\let\a=\b`'와 '`\def\a{\b}`' 둘 사이의 차이점이 있기는 있는건가요? 있다면 어떤 차이점이 있나요?

- **A:** 차이가 있어도 아주 많다네요.(Yes indeed.) 첫번째의 경우의 \a는 \let이 실행되는 시점의 \b의 의미를 받고, 두번째의 경우의 \a는 매크로 이므로 \a가 사용될 때마다 전개되어 \b로 바뀝니다. 그래서 그 때의 \a는 \a가 전개되는 시점의 \b의 의미를 갖습니다.

▶ *제가 본 예 중에 제일 쉬웠던 것을 하나 예로 들어두겠습니다. 성실한 독자로서 :) – [Karnes] (2006-03-23T00:24:55)*

```
\let\endpara\par
\def\newerpara{\par}
\def\par{\endpara\vskip2pt\hrule\vskip2pt}
\let\anotherpara\par

There are a variety of ways to create commands.
The one descriptoin given so far is to use def. \par
There are a variety of ways to create commands.
The one descriptoin given so far is to use def. \endpara
There are a variety of ways to create commands.
\let\par\endpara
The one descriptoin given so far is to use def. \anotherpara
There are a variety of ways to create commands.
The one descriptoin given so far is to use def. \newerpara
There are a variety of ways to create commands.
The one descriptoin given so far is to use def.

\bye
```

▶ *감사합니다. 머리에 쏙쏙들어옵니다. 아하! 제가 쓰고 있는 글에 분명히 틀린 부분이 많을 것입니다. 틀린 곳은 바로 잡아 주시고, 제 설명이 부족하다거나 모호하다 싶으시면, 이번 처럼 가르침을 주시기를 부탁드립니다. "I hope to pick your brain about TeX." :) – Anonymous (2006-03-23T02:58:19)*

12 edef, xdef, noexpand

작은나무 2006-03-22T00:39:10

215–216쪽: \xdef를 먼저 시작합니다. 왠지 이름에서 강렬함을 느낄 수 있지 않나요? xdef??? \xdef는 \global\edef와 동일합니다. \global\def를 \gdef로 하는 것을 보면 \xdef보다는 \gedef가 되어야 할것 같은데, \xdef라고 하네요. 아마도 expanded의 x에서 따왔나봅니다. 어쨌든, \global의 뜻은 알고 있으니, \edef만 알면 \xdef도 자동 해결됩니다.

글쎄요... 한 번도 그런 경우를 경험해 보지 못해서 그런데, 저자가 말하기를 매크로를 정의하다보면, `\def`가 그렇듯이 단순히 replacement text를 글자 그대로 카피하는 것이 아니라, 현재 상황에 맞추어 replacement text에 들어있는 매크로를 그대로 다 전개시켜야 하는 경우가 있다고 합니다. (도대체 언제?) 이런 경우에 사용되는 명령어가 `\edef`, 이름하야 expanded definition 입니다. 형식은 `\def`와 똑같고, 즉

```
\def<control sequence><parameter text>{replacement text}
```

와 같은 형식이지만, 앞서 말했듯이 TeX은 무작정 replacement text에 있는 모든 전개할 수 있는 토큰들을 전개합니다. 예를들면 아래와 같은데,

```
\def\double#1{#1#1}
\edef\a{\double{xy}} \edef\a{\double\a}
```

첫번째 `\edef`는 '`\def\a{xyxy}`'와 동일하고, 두번째는 '`\def\a{xyxyxyxy}`'와 동일합니다. 이 외에 다른 모든 종류의 전개도 마찬가지 인데, 조건문이 사용된 것을 예로들면,

```
\edef\b#1#2{\ifmmode#1\else#2\fi}
```

TeX이 위의 `\edef`와 만나는 순간 수학 모드에 있다면, 위는 '`\def\b#1#2{#1}`'과 동일하고, 아니면, '`\def\b#1#2{#2}`'과 동일합니다. 언제 `\edef`를 사용할지 모르겠으나, 일단 이해는 됩니다.

`\edef`나 `\xdef`는 replacement text에 더이상 전개할 토큰이 남아있지 않을 때 까지 줄기차게 전개를 해나가는데, 이에 예외가 있죠. 늘 이 놈의 예외가 문제입니다. 문제는 이렇습니다. 참 별난 사람이 있어서 혹은 별난 상황이 되어서 `\edef`를 이용하면서도 replacement text안에 있는 어떤 매크로는 전개시키고 싶지 않다는 것입니다. 이때 사용하는 것이 `\noexpand` 입니다. 이름 그대로 입니다. 다음과 같은 상황을 예를 들어 봅시다. 매크로 `\a`를 정의하는데, `\a`는 전개된 `\b`, 전개되지 않는 `\c`, 전개된 `\d`라고 하고, `\b`와 `\d`는 파라미터가 없는 단순한 매크로 라고 합시다. 그러면 `\a`는 아래와 같은 두가지 방법으로 정의할 수 있습니다.

```
\edef\a{\b\noexpand\c\d}
\toks0={\c} \edef\a{\b\the\toks0 \d}
```

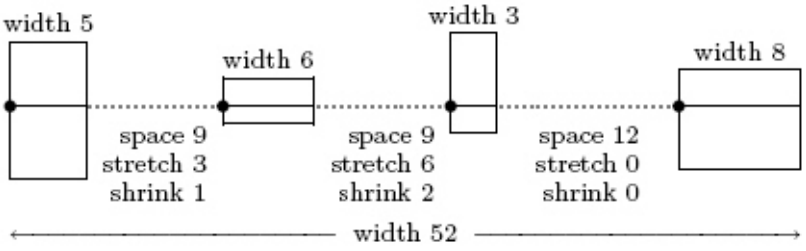
두번째는 `\the`에 의해서 만들어지는 토큰 역시 `\edef` 안에서라도 더이상 전개되지 않는다고 합니다.

13 매직 모르타르 Glue

작은나무 2006-03-22T07:29:04

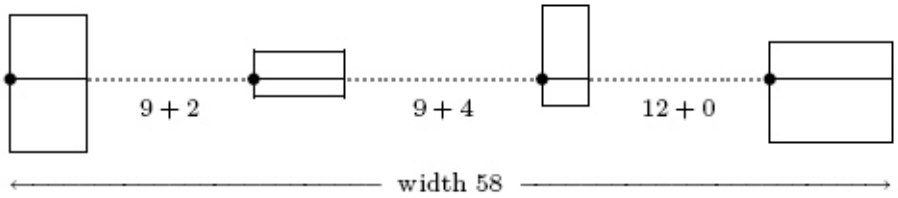
69–70쪽: Glue는 여러 box들 서로 붙이는데 사용합니다. 하나의 줄(line)에 보면 단어들 사이에 간격이 있습니다. 이 간격이 바로 glue입니다. 이 glue는 신기하게도 줄어들거나 늘어날 수 있어서 단어들 사이의 간격 즉 glue를 잘 조절하여 페이지의 오른쪽 여백이 일직선으로 보이도록 해주죠.

TeX은 작은 여러 개의 박스들로 부터 가로로된 한줄의 커다란 박스를 만들때, 그 작은 박스들 사이에 glue를 넣어서 적절한 간격을 유지하도록 해줍니다. 바로 이 방법이 여러 개의 단어를 가지고 한 줄을 만드는 방법입니다. Glue는 세가지 속성을 가지고 있습니다: 주어진 고유의 간격(natural space), 팽창정도(stretchability), 수축정도(shrinkability)



위 그림에는 네 개의 박스와 세 개의 glue가 쓰였네요. 첫번째 glue는 9만큼의 고유 간격을 가지고 있고, 3만큼의 팽창성, 1만큼의 수축성을 가지고 있습니다. 두번째는 9만큼의 간격, 6만큼의 팽창성, 2만큼의 수축성을, 마지막 glue는 12만큼의 고유간격을 가지고 있고, 수축성, 팽창성은 없습니다.

이 예에서 박스의 너비와 glue의 총합은 $5+9+6+9+3+12+8=52$입니다. 이 52를 가로로 긴 박스의 고유의 너비라고 하고, 위와 같은 박스들과 glue들로 만들 수 있는 가장 보기 좋은 조합입니다. 가장 보기 좋다고 하니까 뭐 생각나는것 없으세요? 그렇죠. 바로 badness가 0이라는 소리입니다. :) 그런데, 위의 네 개의 박스와 세 개의 glue로 58만큼의 너비의 긴 박스를 만들려면 어떻게 해야 할까요? 일단 주문은 고유의 너비보다 6만큼 깁니다. 박스는 늘어날 수 있는 것이 아니기 때문에 glue를 늘여야 하는데, 다행히 주어진 glue들은 팽창성이 있기때문에 glue들을 6만큼 늘여서 58을 맞추면 됩니다. 그럼 어떤 glue를 얼만큼 늘이면 될까요? 쉽죠? 첫번째 glue는 3만큼, 두번째는 6만큼, 늘어날 수 있고, 세번째는 늘어날 수 없으니까 그 비율이 3:6:0 즉 1:2:0이네요. 따라서 6을 1:2:0으로 나누면 2:4:0이 되므로 첫번째 glue를 2만큼, 두번째 glue를 4만큼 늘이면 아래 그림처럼 됩니다.



너비가 58이 되었을때의 badness는 얼마일까요? 지난번에 badness를 설명할 때, 어떤 줄의 badness 구하는 식이 $\min(100r^3, 10000)$ 이라고 하고 r은 어떤 비율이라고만 하고 자세한 설명은 하지 않았는데, 지금이 좋은 기회인것 같습니다. r을 'glue set ratio'라고 부르며, 수축(팽창)성의 총합분의 실제 수축(팽창)glue길이 이므로 badness는 30 ($100 \times (6/9)^3 = 29.7$) 입니다.

14 expandafter (1)

작은나무 2006-03-23T02:39:43

213쪽: *TEXbook*에 `\expandafter`의 설명은 아래의 설명이 전부입니다. 예제도 없습니다. 저자가 너무도 무심하고 배려가 없습니다. :(

> `\expandafter<token>`. TeX first reads the token that comes immediately after `\expandafter`, without expanding it; let's call this token "t". Then TEX reads the token that comes after "t" (and possibly more tokens, if that token has an argument), replacing it by its expansion. Finally TeX puts "t" back in front of that expansion.

저렇게만 설명해놓고, 예제도 안들어 놓면, 저 같은 凡人은 저걸 무슨 수로 이해합니까. '설마 예제가 없을까?' 하는 의심에 인덱스를 찾아보니 곳곳에 `\expandafter`가 사용된 예제가 있었습니다. 제일 먼저 나오는 예제는 40쪽에 있습니다. 그것을 본 순간, "`\expandafter` 같은 어려운(?) 명령어가 40쪽에 나와?" 했습니다. 물론 설명 앞에는 두 개의 dangerous bend가 어김없이 붙어있었습니다. 공포의 기호가 아닐 수 없습니다.

바로 이 점이 *TEXbook*의 묘미인것 같습니다. 저자는 책을 써나가면서 한 주제에 대해서 최대한 자세히 설명을 하는데, 난이도를 두어 "지금은 설명을 위해서 다른 chapter에서 배우게 될 어려운 것을 어쩔 수 없이 사용하는데, 지금은 봐봐야 모를 것이다. 나중에 네 실력이 되건든 다시 봐라. 그럼 내 말을 이해 할 수 있을 것이다." 라고 말하는 것 같았습니다.

누가 그랬던가요? "아는 만큼만 보인다"고. 책의 초반부라고 할 수 있는 40쪽에 나온 내용을 초보자가 읽을때와 TeXnician이 읽을때와는 다르겠죠. 오

히려 저자는 친절하게도 난이도를 두어 초보자가 괜히 읽어서 좌절하거나 혼란에 빠지는 것을 사전에 예방하는 것 같습니다. :) 책 구성이 볼 수록 마음에 듭니다. 한 번 읽고 말 책은 분명히 아닙니다.

이야기가 다른 곳으로 빠졌습니다. 빠진 김에 오늘은 여기까지. :)

마음을 가다듬고 다음에 같이 \expandafter를 이해해 보도록 합시다. *TEX book* 이곳 저곳에 나와있는 \expandafter 예제를 잘 공부한 다음에 뵙겠습니다. ;)

15 expandafter(2): 시간차 공격

작은나무 2006-03-25T06:14:27

213쪽: 명령어 \expandafter는 굉장히 재미있으면서도 보기보다는 그다지 어려운 명령어가 아니었습니다. 물론 *TEXbook*에 나와있는 예제는 쉽지만은 않아서 이 명령어의 의미를 알아차리기 그다지 쉽지만은 않았는데, 다행이 아주 쉽고 적절한 예제를 찾았습니다. 예제의 출처는 "Getting TeXnical: Insights into TeX Macro Writing Techniques," *TUGboat*, Volume 11 (1990), No.3 입니다. TeX 매크로 작성하는 기교(?)를 설명하는 paper인데, 약간 어려운 면이 있지만, 매우 재미있습니다. 관심있는 분은 한번 읽어보기를 꼭 권합니다.

이해를 돕기위해서, 배구에 비유를 해서 먼저 개념을 설명하면 :) 배구에 "시간차 공격" 이라는 공격 기술이 있죠? 세터가 배구 공을 토스를 하면 두 명의 공격수가 기다리고 있다가, 첫번째 공격수가 마치 스파이크를 할것 처럼 하다가 두번째 공격수가 실제로 스파이크 공격을 합니다. \expandafter는 바로 이 배구의 "시간차 전개"에 해당합니다. \expandafter가 세터에 해당하고 그 뒤에 따라 나오는 두개의 명령어(control sequence)가 두 명의 공격수에 해당합니다.

\expandafter는 첫번째 나오는 매크로를 전개시키는 것이 아니라 두번째 나오는 매크로를 전개 시키고, 그 전개된 결과를 첫번째 매크로가 이용합니다. 매우 재미있습니다.

그럼, 약속대로 간단하면서도 이해하기 쉬운 예를 들어보겠습니다.

```
\def\letters{xyz}
\def\lookatletters#1#2#3{First arg=#1, Second arg=#2,%
     Third arg=#3 }
```

위와 같이 \lookatletter 매크로가 정의 되어 있을때,

```
\lookatletters\letters ? !
```

는 어떻게 전개될까요? `\lookatletters` 의 첫번째 인자가 `\letter`이고 두번째 인자는 `?`, 세번째 인자는 `!` 인데, `\letter` 매크로는 `xyz`로 전개 되기 때문에

```
First arg=xyz, Second arg=?, Third arg=!
```

과 같이 전개됩니다. 이것은 누구나 알수 있습니다. 이제 `\expandafter`를 이용해서 시간차 공격에 들어갑니다.

```
\expandafter\lookatletters\letters ? !
```

는 어떻게 전개될까요? 먼저 `\expandafter`는 `\letters`를 먼저 전개 시킵니다. 그래서 위의 문장은 결국 다음과 같이 됩니다.

```
\lookatletters xyz ? !
```

그래서 최종 결과는

```
First arg=x, Second arg=y, Third arg=z ? !
```

입니다. 재미있죠? 그리고 쉽습니다.

16 주의 환기

작은나무 2006-03-26T06:29:39

201쪽: 매크로 작성시 가끔 실수하는데, 다시는 그런 일 없기를 바라며 적어봅니다. :)

> When you define a macro with simple parameters, as in these examples, you must be careful not to put blank spaces before the '`{`' that begins the replacement text. For example, `\def\row #1 #2 {...}` will not give the same result as `\def\row#1#2{...}`, because the spaces after `#1` and `#2` tell TeX to look for arguments that are followed by

spaces. (Arguments can be "delimited" in a fairly general way, as explained below.) But the space after `\row` is optional, as usual, because TeX always disregards spaces after control words. 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.

17 begin과 end

작은나무 2006-03-27T12:50:04

21쪽: 연습문제5.7: 다음 문장은 올바른 문장이지만,

```
\beginthe{beguine}
  \beginthe{waltz}\endthe{waltz}
\endthe{beguine}
```

다음 문장은 틀린 것이라고 할때,

```
\beginthe{beguine}
  \beginthe{waltz}
  \endthe{beguine}\endthe{waltz}
```

위와 같은 역할을 하는 명령어 `\beginthe<block name>`와 `\endthe<block name>`를 정의하라.

```
\def\beginthe#1{\begingroup\def\blockname{#1}}
\def\endthe#1{\def\test{#1}%
  \ifx\test\blockname\endgroup
  \else\errmessage{You should have said
    \string\endthe{\blockname}}\fi}
```

참 아이디어가 좋습니다. 모르긴 몰라도 LaTeX의 `\begin{...}`과 `\end{...}`가 위와 같은 식으로 구현되지 않았을까요? :)

18 count0

작은나무 2006-03-28T08:45:22

252쪽: 매크로 공부를 하다가 재미삼아 1부터 n까지 합을 구하는 매크로를 작성해 보기로 하고, 다음과 같이 코딩했습니다.

```
\newcount\n
\def\sum#1{\n=#1 \count0=0 \addto}
\def\addto{%
  \ifnum\n>0 \advance\count0 by\n \advance\n by-1 \addto
  \else\number\count0\fi}

\sum{10}
\bye
```

그랬더니, 아주 재미있는 결과를 얻었습니다.

```
This is TeX, Version 3.141592 (MiKTeX 2.4)
(sum.tex [55] )
Output written on sum.dvi (1 page, 212 bytes).
Transcript written on sum.log.
```

위 결과를 보면 한페이지의 출력물에 그 페이지 번호가 구하고자 하는 10까지의 합인 55가 나왔습니다. 한 쪽짜리라면 페이지 번호가 1이 나와야 할텐데, 55가 나왔습니다. :)

그래서 책을 뒤져보니...

> ... TeX makes `\pageno` an abbreviation for `\count0` ...

ㅎㅎㅎ; `\count0` 함부로 사용하지 마세요. 페이지번호가 이상하게됩니다. 하지만 위의 경우는 구하고자 하는 결과를 sum.dvi를 보지 않고도 55라는 것을 금방 알 수 있었습니다. :)

19 Tail recursion

작은나무 2006-03-29T03:14:41

219쪽: 다음과 같은 매크로 정의가 있다고 합시다.

```
\def\foo{어쩌구 저쩌구 ... \bar ... 저쩌구 어쩌구}
```

즉 매크로 `\foo`의 replacement text 안에 다른 매크로 `\bar`가 있는 경우입니다. 이 경우, 아마도 TeX은 `\foo` 매크로를 전개하다가 `\bar` 매크로를 만나는

순간 \bar 매크로의 전개를 마친 다음, 남아 있는 \foo 매크로를 계속 전개하기 위해서 \foo 매크로의 replacement text 내의 \bar 매크로 다음의 토큰 위치를 기억해 두어야 할 것입니다. 만약에 \bar 매크로 역시 그의 replacement text 안에 다른 매크로가 있다면, \foo 와 마찬가지로 메모리의 어딘가에 계속 전개할 토큰의 위치를 적어두어야 할 것입니다.

그런데, 문제는 위의 예에서 \bar가 \foo인 경우입니다. 즉, 아래와 같은 경우인데,

```
\def\foo{어쩌구 저쩌구 ... \foo ... 저쩌구 어쩌구}
```

이러한 경우를 recursion 이라고 합니다. 우리말로 '재귀'라고 합니다. 간단히 말해서 어떤 것을 정의하는데, 그 정의에 어떤 것 즉 자기 자신이 이용되는 것입니다. 예를 들어 "'GNU"' 모두 아시죠? 이 GNU의 정의가 "GNU is Not Unix"의 약자라고 합니다. GNU를 정의하는데 GNU가 사용되었기 때문에 이 역시 recursion 이라고 할 수 있습니다. (온통 재귀로 이루어진, 재귀가 생명인 컴퓨터 프로그래밍 언어 Lisp과 그 lisp으로 모든 것을 다 할 수 있는 emacs, 그리고 이 emacs를 만든 스톨만, 스톨만이 깊이 관여하고 있는 GNU. 이 모두가 GNU의 재귀적 정의를 갖게 된 것과 무관하지 않을 것입니다. :) 제 추측입니다.)

위의 \foo와 같은 매크로를 전개할때 마다, TeX은 replacement text 내의 \foo 다음의 토큰 위치를 기억하기 메모리를 계속 사용할 것입니다. 아마도 컴퓨터 프로그래밍에서 자주 사용되는 스택이라는 것을 이용할 것같은데, \foo가 전개될때마다 스택에는 \foo의 다음 토큰 위치를 기억하기 위해서 그 메모리 번지가 많이 쌓일 것입니다. 스택의 의미를 몰라도 그저 컴퓨터의 메모리를 많이 사용한다고 이해하면 될 것입니다. 컴퓨터의 메모리 또한 컴퓨터의 자원이고, 컴퓨터가 자원을 많이 사용한다는 의미는 시간과 자원 측면에서 그리 효율적이지 못합니다.

하지만, 같은 recursive 매크로라도 다음과 같은 경우는 어떨까요?

```
\def\foo{어쩌구 저쩌구 ... \foo}
```

위의 재귀와 다른 점은 replacement text내의 \foo 다음에 아무런 토큰도 없다는 점입니다. 이러한 재귀의 장점은 TeX이 \foo 다음의 위치를 기억할 필요가 없다는 점입니다. 왜냐하면 \foo의 전개를 마치고 그 다음을 계속 전개하기 위해서 그 위치로 왔지만, 전개할 것이 없기 때문입니다. 따라서 이경우는 스택에 다음에 전개할 토큰의 위치를 기억할 필요도 없습니다.!!! 이러한

경우 처럼 재귀 호출이 맨 마지막에 있다고 해서 "tail recursion" 이라고 합니다. 우리말로 '꼬리재귀'라고도 합니다. 이제 *TEXbook*에 이 tail recursion을 설명하면서 하는 말,

> ..., and the memory does not fill up during a long loop. ...

을 이해할 수 있을 것입니다.

또한 *TeX by Topic*이라는 책의 11.8.3 Tail recursion에 나와있는 다음과 같은 말도 쉽게 이해할 수 있을 것입니다.

> In general this 'stack build-up' is a necessary evil, but it can be prevented if the nested macro call is the last token in the replacement text of the original macro. After the last token no further tokens need to be considered, so one might as well clear the top item from the input stack before a new one is put there. This is what TeX does.

어렵게만 보이던 *TeX by Topic* 이라는 책이 만만해 보입니다. 신납니다. :)

이제 아주 인상깊은 매크로 `\loop ... \repeat`를 이해할 수 있을 것입니다.

```
\def\loop#1\repeat{\def\body{#1}\iterate}
\def\iterate{\body\let\next=\iterate\else
    \let\next=\relax\fi\next}
```

TeX을 공부하면서 저 매크로를 보고 이해하는 순간은 아주 감동적이었습니다. 수학자들이 흔히 말하는 것 처럼 "아름답습니다."

마지막으로 저~~~ 아래에서 언급한 Knuth의 어록을 마지막으로 이번 독후감을 마치겠습니다. :) 위의 설명에서 처럼 recursion은 매우 엘레강스하고 간단하지만 컴퓨터 입장에서 보면 그리 효율적이지 못합니다. 스택이라는 메모리를 사용해야되고, 위치를 저장해야되고... 그리고 대개 재귀적으로 정의된 문제는 반복적(iterate)으로 해결할 수 있습니다. 하지만, 프로그래밍을 하다보면 가끔 recursion을 사용합니다. 그 간단 명료함에 반해서 이지요. 같은 맥락으로 Knuth도 말했지요?

> *"... aesthetics are more important than efficiency"*

▶ *글은 재미있게 읽었는데 인상깊은 매크로는 이해가 안되는 군요 OTL – [hermian] (2006-03-29T03:37:09)*

▶ *노현정 아나운서가 이렇게 말했을 것입니다. "[hermian]님, 공부하세요" ;) – [작은나무] (2006-03-29T05:13:19)*

▶ *도움이 될까 해서... 1부터 100까지 한줄씩 출력하기: – DohyunKim (2006-03-29T14:48:52)*

```
\let\endgraph\par
\count255=0
\loop
  \advance\count255 by1
  \the\count255 \endgraph
  \ifnum\count255<100 \repeat
```

이 문서에 관하여

이 문서는 KTUG Faq 위키에 기고한 글을 모은 것입니다. 시간순으로 배열되어 있습니다. `http://faq.ktug.or.kr/faq/LittleTree`