TEX 매크로 작성 기법

(버전 0.1 — 2006년 4월)

작은나무

인터넷 페이지 `http://faq.ktug.or.kr/faq/LittleTree/ReadingTeXbook`를 참고한다.

TEX 매크로 작성 기법

작은나무

TEX이 워드프로세서와 뚜렷이 구별되는 점은 프로그래밍이 가능하다는 점이다. 그리고 그 프로그래밍의 핵심에는 바로 "*매크로*(*Macro*)"가 있다. 워드프로세서에도 매크로 기능이 있지만, TEX의 매크로는 워드프로세서의 그것과는 차원이 다르다. 간단해 보이는 매크로 하나로 환상적인이고 여러가지 기능이 이루어지는 것을 보고 있노라면 매크로를 익히고자 하는 유혹에 빠지지 않을 수 없다. 하지만 매크로를 작성하는 기법을 익히는 것이 그리 쉬운 것은 아니다. 혹시 여유가 있거나 호기심이 있다면 Plain TEX의 포멧 파일을 한 번 열어보라. 그곳에 매크로들을 정의해 놓은 `plain.tex`을 한 번 열어보면 마치 암호문 같다. 그래도 plain TEX은 그나마 나은 편이다. plain TEX의 `plain.tex`에 해당하는 LATEX의 `lplain.tex`을 열어 보면, 그 코드들은 그 정도가 더욱 심하여, TEXnician 쯤 되어야 보고 이해 할 수 듯한 인상을 준다. 하지만 보기에만 그렇다. 그 매크로에 쓰이는 primitive 명령어들을 하나씩 익혀가다 보면, 아무리 어렵고 복잡한 매크로라 할지라도 그 정체를 곧 드러내기 마련이다. 그런데 애석하게도 primitive 명령어를 모두 안다고 해서 매크로를 곧바로 작성할 수 있는 것은 아니다. 어린 아이가 한글을 모두 안다고 해서 모든 책을 읽고 이해 할 수는 없는 것과 마찬가지이다. 매크로를 작성하기 위해서는 TEX의 기본적인 primitive 명령어들을 익히고, 매크로 작성 기법을 익혀야 한다. 작은나무가 살펴 본 바로는 매크로 작성 기법에는 일종의 패턴이 있다. 그 패턴을 익히고, 외우면 비교적 손쉽게 TEX의 매크로를 작성 할 수 있다. 그리고 TEX의 매크로를 알아가면 갈수록 그만큼 더 TEX의 매력에 빠지게 된다.

이 글은 plain TEX에서 간단한 매크로를 만들어 본 경험이 있고, TEX에 대한 약간의 지식이 있으나, 매크로의 위력에 반해 매크로를 작성하고자 하는 사람들을 위한 글이다. 따라서 이 글을 읽는 이는 적어도 매크로 작성에 자주 사용되는 `\global`, `\outer`, `\long`, `\def`, `\edef`, `\let`, `\futurelet`, `\if`, `\expandafter`, ... 등을 한 번쯤 관심있게 본 적이 있다고 가정한다. 또한 이 글은 작은나무가 TEX 공부를 하다가 느낀 어려운 점들을 작은나무와 같은 초보자의 눈높이에서 쓴 글이기도 하다. TEX의 전문가들은 왜 도대체 자신이 작성한 TEX 매크로에 대해 설명을 달아 놓는데 인색한지 모르겠다.

이 글을 작성하는데 있어서, Victor Eijkhout의 *TEX by Topic*과 Amy Hendrickson의 *Getting TEXnical: Insights into TEX Macro Writing Techniques*, TUGboat, Volume 11 (1990), No. 3—1990 Conference Proceedings 을 많이 참조 하였고, 특히 일부는 그 둘을 거의 번역하다시피 하기도 했다.

이 문서는 버전이 붙어 있는데, 이유는 유용한 매크로 작성 기법이 계속해서 이 문서에 덧붙여 질 수 있다는 것을 의미한다. 버전은 0.1부터 시작한다.

꼬리 재귀 기법 (Tail recursion)

꼬리 재귀 기법은 주로 매크로의 인자를 구성하는 토큰들을 하나씩 처리 하는 데에 주로 사용된다. 예를들어, `\foo`라는 매크로가 꼬리 재귀 기법으로 작성된 매크로라고 할 때, '`\foo{AbcDEFghIj}`'라고 하면, 인자로 주어진 '`AbcDEFghIj`'가 `A, b, c, D, E, ...` 와 같이 토큰들로 분해되어, 하나의 문자 토큰 각각에 매크로 `\foo`가 적용되는 경우이다. 이처럼 각각의 토큰에 적용되는 꼬리 재귀 기법은 다른 많은 기법의 기본이 되기도 하고, 이를 응용한 매크로 역시 매우 많다. 즉, 각각의 문자 토큰에 적용된다는 개념을 이용하여, 각각의 토큰들이 아니라 각각의 단어에 적용하기도 한다. 이 기법은 주어진 매크로가 어떤 조건 하에서 인자의 토큰에 하나씩 적용되므로 "*조건, 반복*"에 대한 primitive가 주로 사용된다. 따라서 `\let`, `\if`, `\expandafter` 등의 primitive 에 대한 지식이 요구된다. 이 기법은 대개 아래와 같은 패턴을 가지고 있다.

```
\def\foo#1{\bar#1\end}
\def\bar#1{\ifx#1\end \let\next=\relax
  \else Do Something \let\next=\bar\fi \next}
```

이 패턴은 너무도 많이 쓰이고 유용하기 때문에 익히는 정도가 아니라 반드시 외어 두어야 한다. 먼저 `\foo`가 다음과 같이 정의되었기 때문에,

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

\bar의 인자는 원래 주어진 인자 #1에 \end를 더한 합이 된다. 여기서 \end는 매크로 인자의 구분자(delimiter)로 사용되었기 때문에 정의 되어있지 않아도 상관없다. 여기서 \end는 주어진 원래의 인자 바로 다음에 붙어서 인자의 마지막을 가리키는 토큰이 된다. 따라서 \bar가 토큰들을 하나씩 처리하다가 \end를 만나면, 그 실행을 멈추게 된다. 위의 매크로 정의에서 실제로 일을 하는 매크로는 \bar이고, \foo는 \bar가 처리하는 과정에서 어디까지 실행하라는 그 끝을 알려주는 역할을 한다. \bar의 정의를 보면 알 수 있듯이, \bar가 하는 일은 인자의 토큰을 하나씩 처리하면서 다음에 처리 할 토큰이 \end이면, 더 이상 처리할 토큰이 없으므로, 다음에 실행할 명령어 \next가 아무 일도 하지 말라는 뜻의 primitive, \relax가 되어 그곳에서 실행을 멈추고, 처리 할 토큰이 \end가 아니면 \next가 \bar가 되어 그 다음 토큰을 처리하게 된다. 즉, \bar가 자기 자신을 호출하는 재귀적 방법으로 반복을 구현하고 있다는 것을 확인 할 수 있다.

여기서 눈여겨 보아야 할 것은 \let의 사용법 이다. 만약 \bar가 다음과 같이 정의 되었다면, 무엇이 잘못된 것일까?

```
\def\bar#1{\ifx#1\end \relax \else Do Something \bar\fi}
```

매크로 \bar는 하나의 인자를 갖는데, 위와 같은 정의의 치환문(replacement text)에서는 \fi가 바로 \bar의 인자가 되어버려서 전개가 엉망이 된다. 주의가 필요한 대목이다. "나는 \let을 죽어도 사용하기 싫다." 하는 이는 \expandafter를 사용하여 다음과 같이 하면 된다.

```
\def\bar#1{\ifx#1\end \expandafter\relax
  \else Do Something \expandafter\bar\fi}
```

위와 같은 정의가 가능한 이유는, \expandafter의 성질 상 \bar를 곧 바로 전개하는 것이 아니라 그 다음에 나오는 \fi를 처리한 다음에 \bar가 실행 되어 \fi가 \bar의 인자가 되는 경우가 없기 때문이다. 위의 매크로 정의에서 \expandafter는 \fi를 제거하는 역할을 한다. 이 방법 또한 대단히 유용하고 익혀 둘만한 기법이기는 하지만 \let을 사용하는 경우가 대부분이고 그것이 기본이다.

이제 패턴을 알았으니, 몇 가지 예를 통해서 이 패턴의 사용법을 익혀보자. 먼저 TEXbook, 219쪽에 나와있는 예로, 주어진 인자에서 공백이 아닌 토큰의 갯수를 세는 \length 매크로를 만들어 보자. '\length{abcd efg}'는 공백이 아닌 토큰의 갯수가 7이므로 '7'을 출력하고, \length{argument}는 '8'을 출력한다. 보다 이해를 쉽게하기 위해서, 이 패턴을 하나도 바꾸지 말고 매크로 이름만 바꿔서 그대로 적용해 보자. 위의 패턴에서 \foo에 해당하는 매크로를 \length로 하고, \bar에 해당하는 매크로를 \getlength로 바꾸고 나머지는 그대로 두면 \length 매크로는 다음과 같은 모양을 하게 된다.

```
\def\length#1{\getlength#1\end}
\def\getlength#1{\ifx#1\end \let\next=\relax
  \else Do Something \let\next=\getlength\fi \next}
```

위에 정의한 그대로 '\length{argument}'를 실행하면 결과는 다음과 같다.

Do Something Do Something Do Something Do Something Do Something Do Something Do Something Do Something

위 결과에서 알 수 있듯이 'Do Something'이 8번 출력되었습니다. 즉, \getlength가 여덟 번 실행 되었다는 것이다. 이는 인자 중에서 공백이 아닌 토큰의 갯수가 여덟 개이기 때문에 그렇다. 그렇다면 해야 할 일이 분명해졌다. 'Do Something' 부분을 '8' 이라는 숫자를 출력 할 수 있게끔 해주는 무언가로 바꾸기만 하면 된다. 이 작업은 매우 간단하여 'Do Something'을 '\advance\count0 by1'로 바꾸기만 하면 됩니다. \advance\count0 by1 이 하는 일은 \count0의 값을 1씩 증가시키는 일을 한다. 따라서 \count0의 값을 0으로 초기화 한 후에 1씩 증가시키는 작업을 여덟 번 수행하면 \count0의 값은 8이 된다. 여기서 \length 매크로가 \count0의 값을 0으로 초기화하고, 그 결과 8을 출력하는 역할을 한다.

```
\def\length#1{{\count0=0 \getlength#1\end \number\count0}}
\def\getlength#1{\ifx#1\end \let\next=\relax
  \else\advance\count0 by1 \let\next=\getlength\fi \next}
```

실행해 보면, \length{argument}의 결과는 '8'이 나온다. (조금 감이 잡히지 않는가?) 다시 한 번 말하지만, 이 테크닉은 아주 외워야 한다.

다른 예를 들어보자. 이 번에는 주어진 인자의 문자 토큰들 중에서 대문자의 갯수를 세는 매크로를 한 번 만들어 보겠다. 패턴은 위와 동일하다. 따라서 해결 방법도 위와 매우 유사하고 쉽다. 단지 'Do Something' 부분에 대문자인지 확인을 하고, 대문자이면, \length 매크로에서 처럼 \count0의 값을 하나 증가 시키면 된다. 즉 \length 매크로와 똑같다. 다만 차이점은, 대소문자 구분하는 루틴이 더 들어간다는 점이다.

```
\def\countucletter#1{{\count0=0 \ucact#1\end \number\count0}}
\def\ucact#1{\ifx#1\end \let\next=\relax
  \else\uppercase{\if#1}#1\advance\count0 by1 \fi\let\next=\ucact\fi \next}
```

매크로 \getlength와 유일한 차이점 이면서, 눈여겨 봐야 할 것이 대소문자 판별하는 부분이다.

```
\uppercase{\if#1}#1
```

이 방법 역시 기억해 둘 만한 기법이다. \uppercase는 주어진 인자중 소문자를 대문자로 바꾸는데, 명령어(control sequence)는 변경하지 않는다. 즉 \if는 명령어 이므로 \uppercase의 영향을 받지 않아서, 위의 문장은 결국 다음과 같다.

```
\if#1#1
```

첫번째 #1은 대문자로 바뀐 것이 두번째의 #1는 원래 그대로 이므로, 원래의 #1이 대문자이면 \if 문장이 참이 될 것이고, 소문자 였다면, \if의 값은 거짓이 되는 원리이다.

이 번에는 좀 더 응용하여 인자 각각의 문자 토큰에 적용되는 것이 아니라, 공백으로 구분되는 단어에 적용되는 경우를 살펴보자. \underlinewords'라는 매크로를 정의 하는데, 이 매크로가 하는 일은 주어진 단어에 모두 밑줄을 긋는 것이다.

```
\underlinewords
This is a handbook about \TeX, a new typesetting system intended for ...*
```

이 결과는 다음과 같습니다.

This is a handbook about TEX, a new typesetting system intended for ...

위에서 보면 '*'가 꼬리 재귀 기법의 설명에서 \end가 된다는 것을 알 수 있을 것이다. 구현을 살펴보자. 아마 꼬리 재귀 패턴과 너무도 유사하다는 것을 금방 눈치 챌 수 있을 것이다.

```
\def\aster{*}
\long\def\underlinewords#1*{%
  \def\wstuff{#1 }\leavevmode\expandafter\ulword\wstuff * }
\long\def\ulword\#1 {\def\one{#1}%
  \ifx\one\aster \let\next\relax
  \else\vtop{\hbox{\strut#1}\hrule \relax} \let\next\ulword\fi \next}
```

이 매크로에서 꼬리 재귀 기법에서의 입력 토큰의 마지막을 나타내던 \end의 역할을 하는 토큰은 \aster 이고, 'Do Something'은 '\vtop{\hbox{\strut#1}\hrule \relax}'에 해당한다. 그렇다면 도대체 위의 어떤 부분이 인자를 토큰 단위가 아닌 단어 단위로 처리하게 한다는 말인가? 그 해답은 바로 '␣'에 있다. 눈 여겨 보아야 할 부분이 바로 '\long\def\ulword#1 {\def\one{#1}%'이다. 바로 이 부분이 단어 단위로 처리하도록 해준다. '\ulword#1' 다음의 공백 '␣'은 매우 중요해서, 공백을 거의 무시하는 TEX일지라도 이 경우만은 그렇지 않다. 매개변수 뒤에 나오는 공백은 구분자로 사용되므로 무시하면 안된다. 이 공백이 하는 역할이 결정적이어서 \ulword 매크로는 토큰 단위가 아닌 공백을 구분자로 하는 단어 단위, 즉 'This␣', is␣', 'a␣', 'handbook␣', 'about␣', ... 등을 그 단위로 취급한. 상황이 이러하다보니, \end로 사용되는 * 뒤에도 공백이 필요하여,'{... \wstuff␣*␣}'와 같은 모양이 나온 것이다. 반드시 기억 해 둘만한 테크닉이다.

이상으로 꼬리 재귀 기법에 대해서 알아 보았다. 다음에 공부 할 기법은 바로 이 기법의 약간의 응용으로 매우 유용한 패턴이라고 하겠다.

인자의 갯수가 변하는 매크로

매크로를 작성하다 보면, 종종 그 인자의 갯수를 미리 알 수 없는 경우가 있다. 인자의 갯수를 알 수 없다는 말은 인자의 갯수가 변한다는 말과 같은 뜻 일것이다. 체스에서 말들을 움직이는 문제를 예를 들어서 생각해 보자. 체스 말의 위치를 나타내는 문자열들을 인자로 받는 다음과 같은 매크로는,

```
\White(Ke1,Qd1,Na1,e2,f4)
```

아래와 같은 순서의 명령들을 나열해 놓은 것과 같다.

```
\WhitePiece{K}{e1} \WhitePiece{Q}{d1} \WhitePiece{N}{a1}
\WhitePiece{P}{e2} \WhitePiece{P}{f4}
```

여기서 체스에서 장기의 졸(卒)을 나타내는 'P'는 \White에서 생략된 것을 주목하자.

매크로 \White를 직접 정의해 보자. 먼저 첫번째로 해결해야 할 문제는 \White 매크로의 인자의 갯수가 변할 수 있다는 점이다. 이 것을 해결하기 위해서 꼬리 재귀 기법에서 사용했던 인자의 끝을 나타내는 구분자를 사용하는 테크닉을 이용하자.

```
\def\White(#1){\xWhite#1,xxx,}
\def\endpiece{xxx}
```

꼬리 재귀 기법에서 익힌 그대로 이다. 이상할 것 하나도 없다. \xWhite 매크로는 다음과 같이 정의 된다.

```
\def\xWhite#1,{\def\temp{#1}%
  \ifx\temp\endpiece
  \else \WhitePieceOrPawn#1XY%
        \expandafter\xWhite
  \fi}
```

이것 역시 이상할 것 하나 없다. 다만 \let을 이용하는 대신 \expandafter가 사용됐다는 것이 다른 점인데, 이 역시 꼬리 재귀 기법에서 익힌 그대로 이다.

위에서 예를 든 매크로의 인자에서 보면 알 수 있듯이, 말의 위치를 나타내는 문자열의 길이는 둘 또는 셋이다.('Ke1' 혹은 'e2') 이 둘을 구분하는 방법은 \WhitePieceOrPawn의 인자를 네개로 만드는 것이다 즉 다음과 같은데,

```
\def\WhitePieceOrPawn#1#2#3#4Y{
    \if#3X \WhitePiece{P}{#1#2}%
    \else \WhitePiece{#1}{#2#3}\fi}
```

문자열의 길이가 3일때, 즉 예를 들면 Ke1일때, \xWhite 정의에서 \WhitePieceOrPawn#1XY라고 했으므로 결국은 \WhitePieceOrPawn Ke1XY가 되고, 문자열이 2일때, 예를 들어 e2일때는 \WhitePieceOrPawn e2XY가 된다. 따라서 문자열의 길이가 2이면 #3에 해당하는 토큰은 언제나 X가 되므로 #3의 값이 X인지 아닌지를 알면 문자열의 길이가 2인지 3인지 알 수 있다. 그것이 바로 매크로 \WhitePieceOrPawn의 정의이다.

사실 이 기법은 꼬리 재귀 기법의 응용에 지나지 않는다. 꼬리 재귀 기법에서 살펴보았던, 단어에 밑줄을 긋는 것과 이 매크로와 별반 다를 것이 없다.

인자 검사하기 (Examining the argument)

매크로를 작성하다보면, 주어진 매크로의 인자가 어떤 특정한 요소를 가지고 있는지 검사해야할 경우가 있다. 실제 예로 다음을 한번 생각해보자. 어떤 글의 제목과 저자가 다음과 같이 주어지는 매크로를 가정하자.

```
\title{An angle trisector}
\author{A.B. Cee\footnote*{Research supported by the
Very Big Company of America}}
```

저자가 두 명 이상일때는 다음과 같다.

```
\author{A.B. Cee\footnote*{Supported by NSF grant 1}
        \and
        X.Y. Zee\footnote{**}{Supported by NATO grant 2}}
```

매크로 \title과 \author는 다음과 같이 정의되어 있다고하자.

```
\def\title#1{\def\TheTitle{#1}}  \def\author#1{\def\TheAuthor{#1}}
```

그리고, 위 매크로는 다음과 같은 식으로 사용된다고 하자.

```
\def\ArticleHeading{ ... \TheTitle ... \TheAuthor ... }
```

어떤 저널은 기사의 저자와 제목을 모두 대문자로 표기하기도 한다. 그러한 경우의 구현은 다음과 같다.

```
\def\ArticleCapitalHeading
  { ...
   \uppercase\expandafter{\TheTitle}
    ...
   \uppercase\expandafter{\TheAuthor}
    ...
}
```

위에서 \expandafter는 그 성질상 두번째 인자를 먼저 전개시키므로 \TheTitle과 \TheAuthor가 전개된다. (\expandafter의 첫번째 인자는 '{'이다.) 따라서 \expandafter에 의해서 전개되고난 결과를 \uppercase에 적용하므로 \TheTitle과 \TheAuthor 모두 다 대문자로 바뀐다. 그러나 \TheAuthor의 경우에는 한가지 문제점이 있다. footnote까지 모두 대문자로 바뀐다는 점이다. 이것을 해결해 보자.

먼저 저자가 한 명인 경우를 다루어보자. 그러면 다음과 같은 방법으로 해결 할 수 있다.

```
\expandafter\UCnoFootnote\TheAuthor
```

위의 것은 다음과 같이 전개된다.

```
\UCnoFootnote A.B. Cee\footnote*{Supported ... }
```

그리고 \UCnoFootnote를 다음과 같이 정의하면,

```
\def\UCnoFootnote#1\footnote#2#3{\uppercase{#1}\footnote{#2}{#3}}
```

위의 매크로는 다음과 같이 정확하게 인자가 매치 된다.

```
#1<-A.B. Cee
#2<-*
#3<-Supported ...
```

하지만, 이 매크로의 경우에는 footnote가 없다면, 완전히 잘못 동작할 것이다. 다음과 같이 수정해보자.

```
\expandafter\UCnoFootnote\TheAuthor\footnote 00
\def\stopper{0}
\def\UCnoFootnote#1\footnote#2#3{\uppercase{#1}\def\tester{#2}%
  \ifx\stopper\tester
  \else\footnote{#2}{#3}\fi}
```

위 매크로는 footnote가 없으면 정상 동작하나 하나라도 있다면, 오동작한다. 뭐가 문제인가? 위 매크로를 보면 꼬리 재귀 기법과 유사해 보인다. 그렇다. 꼬리 재귀 기법을 적용하면 footnote가 있건 없건 하나 있건 여러 개 있건 상관 없이 동작한다.

```
\def\stopper{0}
\def\UCnoFootnote#1\footnote#2#3{\uppercase{#1}\def\tester{#2}%
  \ifx\stopper\tester
  \else\footnote{#2}{#3}\expandafter\UCnoFootnote
  \fi}
```

위 꼬리 재귀의 경우는 \let을 사용하지 않고, \expandafter를 이용해서 해결하였다.

\futurelet을 이용한 옵셔널 인자*

먼저 \let과 \futurelet의 의미부터 확실히 하자. 다음의 \let을 이용한 문장에서는

```
\let<control sequence><token1><token2><token3><token...>
```

<control sequence>가 <token1>의 의미를 갖게 되고, 남은 것을 다시 써보면, 다음과 같이 된다.

```
<token2><token3><token...>
```

즉, 위에서 <token1>은 <control sequence>가 되어 사라진다. 하지만 \futurelet은 좀 다르다. 위에서 \let을 \futurelet으로 바꾸면, 다음과 같이 되는데,

```
\futurelet<control sequence><token1><token2><token3><token...>
```

이는 <control sequence>에 <token1>이 아닌 <token2>의 의미를 부여한다. 그리고나서, \let의 경우는 그 의미상 <token1>이 사라진 반면, \futurelet에서는 <token2>가 사라지지 않는다.

```
<token1><token2><token3><token...>
```

위 상태가 되고나면, <token1>은 <token2>가 어떤 녀석인지 알게된다. 그래서 <token1>은 <token2>에 따라서 행동을 취하면 된다. 이제 \futurelet의 의미가 확실히 이해가 된다. 이놈을 어디다 쓰느냐가 문제인데... \futurelet이 쓰이는 대표적인 예가 바로 매크로의 파라미터가 옵셔널한 경우라고 한다. 다음과 같은 매크로 \Com이 있다고 하자.

```
\Com{argument}
\Com[optional]{argument}
```

위에서 처럼 \Com 이라는 명령어에 optional 이라는 파라미터가 있을 수도 있고, 없을 수도 있다. 즉 말 그대로 파라미터가 옵션이라는 뜻이다. 위와 같은 \Com 매크로를 해결하기 위해서 \futurelet이 주로 쓰인다고 한다. 그럼 \Com은 실제로 어떻게 정의 될까?

```
\def\Com{\futurelet\testchar\MaybeOptArgCom}
\def\MaybeOptArgCom{\ifx[\testchar \let\next\OptArgCom
  \else \let\next\NoOptArgCom \fi \next}
\def\OptArgCom[#1]#2{ ... }
\def\NoOptArgCom#1{ ... }
```

모든 것이 분명해지는 순간이다. 결국 아래의 문장에서

```
\Com[optional]{argument}
```

위의 \Com의 정의대로라면,

```
\futurelet\testchar\MaybeOptArgCom[optional]{argument}
```

이 되는데, \futurelet의 정의에 따라 \testchar는 '['를 갖게 된다. 그래서 \MaybeOptArgCom은 이제 \testchar이 어떤 놈인지 알게되어 \testchar가 '['라면 \OptArgCom를 실행하면되고, 아니면 \NoOptArgCom를 전개하면 그만이다.

* 이 글은 작은나무가 KTUG에 쓰고 있는 'The TEXbook 읽기'에서 가져온 글 이다.

재미있는 \csname

\csname의 유용한 기능 중의 하나는 \csname...\endcsname 내에서 control sequece들이 전개될 수 있다는 것이다. 이 말은 매우 중요한 의미를 내포하고 있다. 즉 control sequence의 이름이 주어진 조건에 따라서 동적으로 변할 수 있다는 뜻이다:

```
\expandafter\def\csname\testmacro\endcsname{<definition>...}
```

카운터 레지스터도 사용될 수 있다:

```
\expandafter\def\csname\testcounter\endcsname{<definition>...}
```

로마 숫자를 가진 카운터 레지스터도 사용될 수 있다:

```
\expandafter\def\csname\romannumeral\testcounter\endcsname{<definition>...}
```

심지어는 control sequence의 이름으로 사용될 수 없는, 문자 이외의 기호나 숫자들도 control sequence의 이름이 될 수 있다.

```
\expandafter\def\csname 123&#\endcsname{<definition>...}
```

여기서 한가지 기억해야 할 것은, 이처럼 문자가 아닌 숫자나 다른 기호들로 이루어진 control sequence들은 호출될 때도 정의 할때와 마찬가지로, 반드시 \csname 123&#\endcsname와 같은 형태로 호출 되어야 한다는 것이다. \expandafter와 함께 \csname를 사용하면, 다른 방법으로는 구현할 수 없는 재미있는 일들을 할 수 있다.

예를 들어보자. 어떤 한 매크로가 있고, 이 매크로는 또다른 한 매크로를 정의하고, 이 두번째 매크로의 이름은 주어진 상황에 따라서 변할 수 있다고 하자. 다음의 예에서 매크로 \usearg는 주어진 인자들 중에서 처음 두개의 단어를 각각 인자 #1와 #2로 받아서, 그 둘의 순서를 바꿔서 또다른 하나의 control sequence의 이름으로 하는 매크로를 정의한다. 이렇게 정의된 매크로는 이 매크로의 이름으로 정의된 사람의 이름과 주소를 나타내는데 사용한다고 하자. 여기서 이름의 순서를 바꾸는 이유는 그 이름들은 보조 파일로 들어가서 나중에 이름 순으로 정렬을 하기 위함이다. (Donald Knuth가 인덱스에서는 Knuth, Donald로 나오는 것과 같다.) 그 이름 순으로 정렬된 보조 파일을 잘 활용하면, 메일링 리스트를 만들 수도 있다. 매크로 \usearg를 자세히 살펴보자. 다음에서 \obeylines이 하는 역할은 라인의 끝을 나타내는 문자 ^^M의 카테고리 코드를 13으로 만들어서 매크로 인자의 구분자로 사용할 수 있도록 하고, ^^M을 \par로 정의한다.

```
{\obeylines
\def\usearg#1 #2^^M#3^^M^^M{%
  \expandafter\gdef\csname #2#1\endcsname
  {#1 #2\par #3}}
\usearg George Smith
21 Maple Street
Ogden, Utah 68709
\SmithGeorge
}
```

결과는 다음과 같다.

George Smith
21 Maple Street
Ogden, Utah 68709

이 기법에 대한 자세한 예제는 참고문헌 [3]과 그의 부록에 완벽한 예와 함께 자세히 설명되어 있다. 반드시 참고하기 바란다.

투스텝 매크로: 알고보면 두 단계로 이루어진 매크로

어떤 매크로는 사실 알고보면 두 개의 매크로로 이루어진 경우가 있다. 이러한 경우 한 매크로가 대부분의 일을 하고 다른 한 매크로가 그 매크로가 실행하는데 필요한 여러가지 환경을 초기화 한다. 이러한 매크로를 투스텝 (Two-step) 매크로 라고 한다.

예를 들어보자. 라인의 끝을 구분자(delimeter)로 갖는 인자를 취하는 '`\PickToEol`' 이라는 매크로가 있다고하자. 투 스텝의 첫번째 단계로 인자가 없는 매크로로 라인의 끝에 해당하는 문자의 카테고리 코드를 변경하고나서, 두번째 단계의 매크로를 호출하는 매크로는 다음과 같다.

```
\def\PickToEol{\begingroup\catcode`\^^M=12 \xPickToEol}
```

두번째 단계의 매크로는 라인의 끝까지 모든 토큰들을 하나의 인자로 받을 수 있다.

```
\def\xPickToEol#1^^M{ ... #1 ... \endgroup}
```

이 매크로 정의에는 한 가지 문제점이 있다: `^^M` 문자의 카테고리는 12 이어야 한다는 것입니다. 따라서 이를 바로잡아 매크로를 다시 정의하면 다음과 같다.

```
\def\PickToEol{\begingroup\catcode`\^^M=12 \xPickToEol}
{\catcode`\^^M=12 %
 \gdef\xPickToEol#1^^M{ ... #1 ... \endgroup}%
}
```

여기서 `^^M`의 카테고리 코드의 값은 `\xPickToEol`을 위해서 변경되었다. 주목해야 할 것은 `\PickToEol`에 사용된 `^^M`은 control symbol로 쓰였기 때문에 카테고리 코드와는 상관이 없다는 것이다.

주석 처리 환경

첫번째 기법인 꼬리 재귀 기법과 방금 전에 익힌 투스텝 매크로 기법을 응용한 예로써, 주석 처리 환경을 이루는 매크로를 만들어 보자.

TEX을 이용해서 문서를 만들다 보면, 가끔 디버깅 혹은 테스트 목적으로 텍스트의 일정 부분을 주석처리 해야할 경우가 있다. 그 주석처리 해야할 곳이 작다면 매 줄마다 `%`로 시작해서 해결되겠지만, 그 양이 무척 많다면 `%`도 무용지물이 될 것이다. 그래서 다음과 같은 명령어가 절실히 필요하다.

```
\comment
...
\endcomment
```

이것은 어떻게 구현하면 될까? 가장 간단한 방법은 아마 다음과 같을 것이다.

```
\def\comment#1\endcomment{}
```

위 구현은 간단하면서도 매우 참신한 아이디어 이지만, 몇 가지 결점을 가지고 있다. 예를들면, 주석 처리할 부분에 outer 매크로가 들어있거나 짝이 맞지 않는 괄호들이 있을 때는 위 방법으로 안된다는 것이다. 하지만, 무엇보다도 가장 큰 단점은 TEX이 위의 `\comment`를 처리할 때 `#1`를 모두 인자로 받아들인다는 점이다. TEX은 인자들을 처리하기 위해서 내부적으로 버퍼를 가지고 있을 텐데, `#1`이 무지하게 크고 많다면, 그 버퍼가 다 차버려서 TEX이 작업할 공간이 없어져서 에러 메시지를 뿌려댈 것이다.

가장 큰 단점, 버퍼가 부족할 수 있다는 단점을 해결하는 방법은 `#1`을 한번에 처리하지 말고, 라인 단위로 처리하면 될 것이다. 라인 단위로 처리하기 위한 방법은 꼬리 재귀 방법을 이용하는 것이다. 기본 구조는 다음과 같다.

```
\def\comment#1^^M{... \comment}
```

위 매크로가 제대로 동작하려면, 우선 줄의 끝을 나타내는 ^^M의 카테고리 코드를 변경해야 한다.

```
\def\comment{\begingroup \catcode`\^^M=12 \xcomment}
{\catcode`\^^M=12 \endlinechar=-1 %
  \gdef\xcomment#1^^M{ ... \xcomment}}
```

\endlinechar의 값을 음수로 변경한 이유는 (기본값은 13) 위의 매크로 정의에서 매 라인의 끝마다 %를 넣어야 하는데 그것이 귀찮으므로, 그것을 안해도 되도록 하기 위함이니다. (TEX에서는 \endlinechar의 값이 음수이거나 255보다 크면, 모든 라인의 끝에 %를 삽입한 효과를 낸다.)

위의 매크로는 어느 시점에서는 끝나야한다. 그 방법은 꼬리 재귀 방법에서 자주 사용하던 방법을 사용하면 된다.

```
{\catcode`\^^M=12 \endlinechar=-1 %
  \gdef\xcomment#1^^M{\def\test{#1}
  \ifx\test\endcomment \let\next=\endgroup
  \else \let\next=\xcomment \fi
  \next}
}
\def\endcomment{\endcomment}
```

위에서 \endcomment는 실행되는 것이 아니라, 주석 처리 환경의 끝이라는 것을 나타낸다.

이것으로 TEX의 버퍼가 꽉차버리는 일은 없어졌다. 그렇다면, 앞서 언급한 outer 매크로나 짝이 맞지 않는 괄호를 사용하지 못한다는 단점은 어떤 방법으로 해결하면 될까? 이것은 아주 간단하다. 위의 \comment ... \endcomment를 하는 동안 모든 문자의 카테고리 코드를 12로 만들어 verbatim 모드로 바꾸어 주면 된다! 이를 위해서 plain TeX에는 아주 유용한 매크로 \dospecials이 다음과 같이 정의되어 있다.

```
\def\dospecials{\do\ \do\\\do\{\do\}\do$\do\&%
  \do\#\do\^\do\^^K\do\_\do\^^A\do\%\do\~}
```

이 \dospecials를 이용하면, \comment는 다음과 같다.

```
\def\makeinnocent#1{\catcode`#1=12 }
\def\comment{\begingroup
  \let\do=\makeinnocent \dospecials
  \endlinechar`\^^M \catcode`\^^M=12 \xcomment}
```

이것으로 인자로 어떠한 문자가 와도 해결이 된다. 이것으로 다 끝난 것일까? 문제가 이렇게 쉽게 해결 된다면, 얼마나 좋을까. (사실 그다지 쉽지도 않았지만.)

Verbatim 모드로 바꾸면 한 가지 문제가 발생한다. 그 문제는 공교롭게도 \comment ... \endcomment 환경에서 이 환경을 끝낼 때, 일어난다. 다시 자세히 살펴보면 우리는 \comment를 시작하면서 뒤에 나오는 모든 문자를 verbatim 환경으로 바꿔버렸기 때문에 이 환경을 끝내고자 하는 명령어 \endcomment 마저도 명령어로 보는 것이아니라 단순한 문자들로 보는 것이다. 이를 명령어로 인식하게 하려면 다음과 같이 하면 된다.

```
{\escapechar=-1
  \xdef\endcomment{\string\endcomment}}
```

한 가지 확실히 해둘 것은 위의 매크로를 단순히 아래와 같이 하면 안된다는 것이다.

```
\edef\endcomment{\string\endcomment}}
```

왜냐하면, 이와 같이 하면 \ 뿐만 아니라 endcomment 마저도 카테고리 코드가 12가 되기 때문이다. endcomment는 11이 되어야 한다.

참고문헌

[1] Donald E. Knuth, *The TEXbook* (Addison-Wesley, 1986).

[2] Victor Eijkhout, *TEX by Topic, , A TEXnician's Reference* (Addison-Wesley, 1992).

[3] Amy Hendrickson, *Getting TEXnical: Insights into TEX Macro Writing Techniques*, TUGboat, Volume 11 (1990), No. 3—1990 Conference Proceedings