Item Paragraphs

Items are hanging paragraphs that "hang off" an identifier. The syntax for this instruction is `\item{\signif}` where ⟨*signifier*⟩ is any letter, number, or symbol with optional punctuation; the braces must be included if the ⟨*signifier*⟩ is more than one character.

A second level of indention for itemized lists is given by `\itemitem`, which indents twice the `\parindent` value. These instructions automatically end the previous paragraph. Refer to Figure 7 for an example.

```
% Figure 7  Items
\parskip 9pt  % spaces between pars
\item{1.}% curly braces contain number
Skillin, Marjorie, Robert Gay, et al.
1964.
Words Into Type.
New York: Appleton-Century-Crofts.
\item{2.}
Carter, Rob, Ben Day, and Philip Megs.
1985.
Typographic Design: Process and
Communication.
New York: Van Nostrand Reinhold Co.
\par
```

Figure 7. Item Paragraphs Source

Items are useful for lists, outlines, and bibliographies. Figure 8 shows a bibliography.

1. Skillin, Marjorie, Robert Gay, et al. 1964. Words Into Type. New York: Appleton-Century-Crofts.

2. Carter, Rob, Ben Day, and Philip Megs. 1985. Typographic Design: Process and Communication. New York: Van Nostrand Reinhold Co.

Figure 8. Formatted Items

The instructions presented in this article create paragraphs. Therefore, you should remember to end each one with a `\par` instruction or a blank line.

There is a lot more to paragraphs, including ragged margins, repetition of instructions for each paragraph, and special shapes, but that will be presented much later. The next of these training tutorials will address the contents of paragraphs: special characters, accents, fonts, and lines.

Macros

A Tutorial on `\futurelet`

Stephan v. Bechtolsheim

This is the second in a series of tutorials by this author. This time we will deal with `\futurelet`, a rather interesting instruction which causes many people unnecessary difficulties. This article is condensed from a draft of my books *Another Look at TEX*. See the end of this article for more information about the books.

Introduction

The `\futurelet` primitive is a TEX instruction allowing the user to look ahead. The term "look ahead" means that TEX will look at a future token and provide a copy of that token **without** absorbing it, i.e. without removing that token from the main token list. This operation allows the programmer to perform a test for "what token is coming" (to express it in a rather informal way) on the main token list. The token looked at through `\futurelet` will be removed later, typically as part of an argument of a later macro call as we will see shortly. It is **not** removed by the action of the `\futurelet` primitive.

Let us be more precise now; the `\futurelet` instruction has the following format:

`\futurelet` ⟨token$_1$⟩ ⟨token$_2$⟩ ⟨token$_3$⟩

Here is what TEX will do:

1. TEX will execute a `\let` ⟨token$_1$⟩ = ⟨token$_3$⟩. We therefore have generated a copy of ⟨token$_3$⟩ stored under the name of ⟨token$_1$⟩.
2. TEX removes ⟨token$_1$⟩ from the main token list.
3. TEX expands ⟨token$_2$⟩. This token is for all practical purposes a macro with the following properties:
   (a) The macro will use ⟨token$_1$⟩, which is a copy of ⟨token$_3$⟩, to find out what ⟨token$_3$⟩ is, in other words what token is to be expected later.
   (b) It will cause another macro to be expanded which will ultimately absorb ⟨token$_3$⟩. This other macro ordinarily depends on what ⟨token$_1$⟩ is.

There are many applications of `\futurelet`. We will here present only one example, although we will present it in quite some detail so the user will know how to apply `\futurelet` in different circumstances.

Using \futurelet in Macros with Optional Arguments

A typical application of \futurelet is the handling of macros with optional arguments as they are used, for instance, in LaTeX. By "optional argument" we mean an argument which in most cases is omitted, and is provided only occasionally in macro calls.

Defining the Problem

Let us give a specific example: we would like to define a macro \xx, which can be called in two different ways:

1. *With* optional argument as in \xx[opt]{arg} where opt is the optional argument enclosed in square brackets and arg is the regular argument.
2. *Without* optional argument as in \xx{arg} where arg is again the regular argument.

Before we discuss how this can be done in TeX, observe that we do not really have to use an optional argument. We could simply define two different macros \xxWithOpt for the case where an optional argument is given, and \xxNoOpt for the case where no optional argument is given:

```
\def\xxWithOpt [#1]#2{...}
\def\xxNoOpt #1{...}
```

How we can use \futurelet to find out whether an optional argument was given or not? We will define a macro \xx whose only function is to check whether there is an opening square bracket (optional argument is present) or not (no optional argument). The \xx macro will, after this has been determined, cause the \xxWithOpt macro to be invoked when there is an optional argument, and the \xxNoOpt macro to be called if there is no opening bracket. In other words the macros \xxWithOpt and \xxNoOpt do the "real work" while the only purpose of the \xx macro is to decide which of the two macros should be invoked.

Here is the completely worked out example.

```
% (1) First define two macros
%   \xxWithOpt and \xxNoOpt which
%   \xx will call.
%   These macros do "the real work".
% \xxWithOpt: optional argument ([#1],
%   enclosed in square brackets), and
%   regular argument ({#2},
%   undelimited).
% \xxNoOpt: assume no opt. argument,
%   but regular argument only {#1}.
\def\xxWithOpt [#1]#2{...}
\def\xxNoOpt #1{...}

% (2) The \xx macro has no parameter!
%   It only uses \futurelet to check
%   whether there is an optional
%   argument or not by checking
%   whether or not '[' follows \xx
%   in the input.
\def\xx {%
    \futurelet\xxLookedAtToken
                        \xxDecide
}

% (3) The \xxDecide macro, based on
%   the lookahead of \xx, calls
%   either \xxWithOpt or \xxNoOpt.
\def\xxDecide {%
    \ifx\xxLookedAtToken [%
        \let\next = \xxWithOpt
    \else
        \let\next = \xxNoOpt
    \fi
    \next
}
```

A Macro Call with Optional Argument

Let us now look at the following macro call of the \xx macro that we have defined: \xx[a]{b}. This generates the following token list:

\xx • [• a •] • { • b • }

Now \xx is expanded, yielding the following token list:

\futurelet • \xxLookedAtToken • \xxDecide • [• a •] • { • b • }

Observe that \xxLookedAtToken corresponds to ⟨token$_1$⟩ of \futurelet, \xxDecide to ⟨token$_2$⟩ and [to ⟨token$_3$⟩ (see the format of \futurelet in the introduction above). Observe especially the value of ⟨token$_3$⟩: this is the token we are interested in. \xxDecide will test this token to check whether or not it is an opening square bracket, in order to decide whether to call \xxWithOpt or \xxNoOpt.

Next TeX executes the \futurelet, assigning [to \xxLookedAtToken, and then expands \xxDecide. This expansion leads to the following main token list:

\ifx • \xxLookedAtToken • [• \let • \next • = • \xxWithOpt • \else • \let • \next • = • \xxNoOpt • \fi • \next • [• a •] • { • b • }

The \ifx conditional evaluates to true because \futurelet has just assigned an opening square bracket to \xxLookedAtToken. Therefore \let\next = \xxWithOpt is executed and the whole conditional (from \ifx through \fi) is removed from the main token list. This leads to the following new main

token list:

```
\next • [ • a • ] • { • b • }
```

Because `\next` is equivalent to `\xxWithOpt` this is equivalent to the following main token list:

```
\xxWithOpt • [ • a • ] • { • b • }
```

And this is of course exactly what we wanted: the macro `\xxWithOpt` is executed and the expansion of this macro will absorb the optional argument enclosed in square brackets and the mandatory argument enclosed in curly braces. Observe that, up to this point, the opening square bracket stayed on the main token list.

A Macro Call without Optional Argument

Let us now look at a macro call of `\xx` with **no** optional argument, as in `\xx{a}`. Here is a short description of what happens. `\xx` is expanded to yield the following token list:

```
\futurelet • \xxLookedAtToken
 • \xxDecide • { • a • }
```

Therefore an opening curly brace, not an opening square bracket, is assigned to `\xxLookedAtToken`. `\xxDecide` is now expanded and the conditional `\ifx • \xxLookedAtToken • [` this time evaluates to false. Therefore the assignment `\next = \xxNoOpt` will be executed. This leads to the following main token list:

```
\next • { • a • }
```

and now `\next` is the same as `\xxNoOpt`, exactly as we wanted it to be.

Looking at the Previous Example Once More, `\DblArg`

There are frequently cases where a macro requires two arguments, but both may be identical. In such a case, a macro may be defined with an optional argument, where the absence of the optional argument in the input is assumed to imply that an optional argument identical to the mandatory argument has been supplied. Using the notation of the previous example, this means that `\xxNoOpt{a}` is equivalent to `\xxWithOpt[a]{a}`.

The previous example can be easily modified to define a generic macro `\DblArg` so that the definition of `\xx` reads as follows:

```
\def\xx{\DblArg{\@xx}}
```

The call `\xx{1}` is converted into `\@xx[1]{1}` and the call `\xx[1]{2}` is converted into `\@xx[1]{2}`. Here is the definition of the macro `\DblArg`:

```
\catcode`@ = 11

% \DblArg
% =======
% #1: the macro to be called
% ultimately (\@xx above).
\def\DblArg #1{%
    \def\@DblArgTemp{#1}%
    \futurelet\@DblArgTok\@DblArg
}

% \@DblArg: if there was an opening
% square bracket then simply continue.
% Otherwise the main argument has
% to be duplicated to also become
% the argument enclosed in square
% brackets.
\def\@DblArg{%
    \ifx \@DblArgTok [%
        % Optional argument!
        \let\@DblArgTempA=\@DblArgTemp
    \else
        % No optional argument:
        % duplicate!
        \let\@DblArgTempA=\@DblArgB
    \fi
    \@DblArgTempA
}

% Read in the argument and duplicate
% to also become an optional argument.
\def\@DblArgB #1{\@DblArgTemp[#1]{#1}}

\catcode`@ = 12
```

Concluding Remark

This article is, as briefly mentioned in the introduction, an adaptation of a section of my books, *Another Look At TEX*, which I am currently finishing. These books, now two volumes totalling almost 1000 pages, grew out of my teaching and consulting experience. The main emphasis of the books is to give concrete and useful examples in all areas of TEX. (The section on `\futurelet` is 18 pages long. The chapter on `\halign` contains over 100 tables.) In *Another Look at TEX* you should be able to find an answer to almost any TEX problem.