AROUND THE BEND

A Collection of TeX Challenges by

MICHAEL DOWNES

◆

edited by

Peter Wilson

The Herries Press
July 2008

Contents

Preface

In the early 90's the late and much missed Michael Downes (1958–2003) ran a column in the INFO-TeX mailing list called *Around The Bend* where he proposed macro-related problems and then posted submitted solutions. Although it was archived on CTAN in `info/aro-bend` it is not well known which is a shame as it provides answers to many problems that keep cropping up. (The archive is now at `info/challenges/aro-bend`). This is an attempt to make his work more accessible by providing the collection as a single document.

As much as possible what follows is what Michael wrote; I have tried to limit myself to marking up the original ASCII text emails but I have not repeated administrative elements such as email headers.

In some cases the original TeX code was replete with comments explaining what was going on. Where the comments were long with respect to the code I have set them in the regular body type so as to make the actual code more obvious; this has a side effect of slightly decreasing the amount of paper required to print the document. If you want to use the code solutions I suggest that you cut and paste them from the original archived versions.

I thought that there were eighteen Around the Bends as that is all that are archived on CTAN. However I googled the Google Groups `comp.text.tex` group and found three more, nos. 19, 20 and 21. I have included what I could find of these, but answers to no. 19 appear to be missing, which is a pity as I think that I could have put them to use. Perhaps some of you might be willing to take up the challenge on this, or on any of the others.

PW
July, 2008

Introduction

(Ed: This is Michael's introduction to his scheme, originally posted on 1991/10/10 as the initial portion of exercise 1.)

```
Date: Thu 10 Oct 91 09:51:32-EST
From: Michael Downes <MJD@MATH.AMS.COM>
Subject: Around the bend
To: info-tex@shsu.edu
```

Proposal for a regular feature:

AROUND THE BEND

With the encouragement of George Greenwade (the INFO-TeX list owner), I would like to propose a regular department for INFO-TeX, called 'Around the bend'. It will consist of macro-writing challenges on the level of the dangerous-bend exercises in the *TeXbook*, with interested parties invited to collaborate and/or compete to find the best solution. My motivation for doing this is partly selfish: to get more feedback from other macro writers about some of the interesting macro-writing problems that I run into.

I originally approached George for advice about setting up a separate mailing list, but he thought that INFO-TeX and comp.text.tex readers would be interested. Since INFO-TeX mail is also channeled to comp.text.tex, readers of the latter should let me know if they don't want the extra traffic (although I don't expect it to be that much). I don't currently have access to read comp.text.tex directly, although George has been investigating the possibility of piping it through the INFO-TeX mailing list. So if you object by posting to comp.text.tex, I may not see your objection; send me mail, instead.

The sample below should give a pretty good idea of what 'Around the bend' would be like. Solutions should be sent to me instead of to INFO-TeX or comp.text.tex, on the premise that people usually won't want to read others' solutions until they've had a chance to try their own hand. A summary of the results would then be posted to the INFO-TeX list after two or three weeks; to those who submit solutions before the deadline, I could forward without delay solutions submitted by other people, for comparison.

I will try to keep the difficulty of the exercises down to something reasonable, let's say, on the level of a homework assignment which a university student must complete in two weeks, finding time in the normal way from the usual busy schedule of other homework, class attendance, sports, and social life. However, be warned that the challenges will be hard. I'm planning to follow a 'hard and fast' format: one or two hard questions, followed by one or two fast questions, where if you don't know the answer off the top of your head, you can either look it up in the *TeXbook* or find it by running a quick test.

1 Expansion

1.1 EXERCISE (HARD)

(Ed: Originally posted on 1991/10/10. Archived as `exercise.001`*.)*

Given arbitrary `\b`, `\c`, `\d` (macros without arguments), for example

```
\def\b{\c\c}          \def\c{*}            \def\d{\b\c}
```

figure out how to define `\a` so that its replacement text consists of `\b` fully expanded plus `\c` not expanded plus `\d` expanded exactly once. I.e., with the above definitions the replacement text of `\a` should be

```
**\c\b\c
```

You may not use `\the` or `\noexpand` in your solution. This is Exercise 20.16 in the *TeXbook*, except that there's an added restriction: your answer must also not use the `\halign...\span` method given in the answer to 20.16. (Yes, that means you can't use `\valign` either!)

Why would anyone want to do such a hard exercise? Answer: advanced macro writing requires a thorough knowledge of expansion control principles.

1.2 ANSWERS

(Ed: Originally posted on 1991/10/25. Archived as `answer.001`*.)*

The restrictions leave us with (essentially) three expansion-control commands: `\expandafter`, `\edef` and `\def`.

Solution 1 (Peter Schmitt)

```
\edef\B{\b}
\def\defA#1{\def\defa##1##2{\def\a{#1##2##1}}}
\expandafter\defA\expandafter{\B}
\expandafter\defa\expandafter{\d}{\c}
```

End solution

Solution 2 (Donald Arseneau)

```
\edef\e{\b}
\expandafter \expandafter \expandafter \def\expandafter \expandafter
\expandafter \a\expandafter \expandafter \expandafter {\expandafter
\e\expandafter \c\d}
```

End solution

Solution 3 (mine)

```
\edef\a{\b}
\expandafter\expandafter\expandafter\def
\expandafter\expandafter\expandafter\a
\expandafter\expandafter\expandafter{\expandafter\a\expandafter\c\d}
```

End solution

My solution differed from Arseneau's only in using `\a` rather than `\e` in the first step.

2 Empty argument

2.1 EXERCISE (HARD)

(Ed: Originally posted on 1991/10/10. Archived as `exercise.002`*.)*

Define an ‘ifempty’ macro that takes one argument and resolves essentially to `\iftrue` if the argument is empty, and `\iffalse` otherwise. This is useful for handling arguments given by users to commands defined in a macro package.

Plain TeX or LaTeX-style solutions are both acceptable, that is,

```
    \ifempty{...}TRUE CASE\else FALSE CASE\fi
```

or

```
    \ifempty{...}{TRUE CASE}{FALSE CASE}
```

(In the former case you will need to do something to avoid problems in the situation

```
  \iffalse ... \ifempty{...} ... \fi ... \fi
```

there are different possibilities here, so I will refrain from indicating any particular one.)

Use the following test suite to verify the robustness of your solution:

```
\long\def\test#1{\begingroup \toks0{[#1]}%
  \newlinechar`\/\message{/\the\toks0:
%    LaTeX-style solution; modify the following line according
%    to the syntax of your solution.
  \ifempty{#1}{EMPTY}{NOT empty}%
}\endgroup}

 \test{}                              \test{ }
 \test{aabc}                          \test{-}
 \test{$}                             \test{\empty}
 \test{\endinput}                     \test{\iftrue a\else b\fi}
 \test{\else}                         \test{#}
 \test{\par}                          \halign{#\cr\test{&}\cr}
 \test{\relax}                        \test{\relax\relax\relax}
 \expandafter\iffalse\test{x}\fi      \test{{}}
```

The two tests on the first line should produce a message ‘EMPTY’ and the remaining ones, ‘NOT empty’. The reason for saying that the second test should return ‘EMPTY’ is that (1) this is the ideal behavior for the applications I’ve encountered so far; (2) at least one other person working independently arrived before me at a solution essentially identical to mine, including this behavior. The details and credit to the other guy will be given at solution time.

2.2 ANSWERS

(Ed: Originally posted on 1991/10/25. Archived as `answer.002`*.)*

The LaTeX-style solution that I had prepared was, I thought, pretty good, but Donald Arseneau observed that it fails the test

```
\test{{\iftrue a\else b\fi}}
```

which was not in my list of tests.

Solution 1 (mine)

```
\catcode`\@=11
%    \@car is actually already defined in latex.tex, but for
%    maximum robustness it needs to have the \long prefix:
\long\def\@car#1#2\@nil{#1}
\long\def\@first#1#2{#1}
\long\def\@second#1#2{#2}
\long\def\ifempty#1{\expandafter\ifx\@car#1@\@nil @\@empty
  \expandafter\@first\else\expandafter\@second\fi}
\catcode`\@=12

\long\def\test#1{\begingroup \toks0{[#1]}%
  \newlinechar`\/\message{/\the\toks0:
  \ifempty{#1}{EMPTY}{NOT empty}%
}\endgroup}
```

End solution

The advantage of using the auxiliary macros `\@first` and `\@second`, together with the `\expandafter`'s, is that it allows the true and/or false cases to end with arbitrary things, even macros that require arguments that have not yet been read (any number of arguments, even delimited arguments).

From here it is easy to implement an `\ifnotempty` test that has a null false case. This is often useful in dealing with user-supplied arguments: 'If #1 is empty, do nothing; otherwise, do the following with #1: ...'

```
\long\def\ifnotempty#1{\ifempty{#1}{}}
```

Solution 2 (Donald Arseneau) Don Arseneau came up with a plain TeX style solution, using an ingenious device with `\then` to pass the test case

```
\expandafter\iffalse\test{x}\fi
```

The comments in the solution are his.

```
% \ifblank{...}\then Test if a parameter is blank (null or spaces).
% Use the inaccessable "letter" @ to separate parameters. The two cases are:
%  _text_is_not_blank_             _text_is_blank_
%  #1<- whatever                   #1<-@
%  #2<- whatever (possibly null)   #2<-
%  #3<- @                          #3<-.
%  #4<- ..                         #4<-.
%  \if @.. {false}                 \if .. {true}
%  In the {false} case, the extra period is skipped so it doesn't hurt.

\catcode`\@=11 % as in plain.tex
\let\then\iftrue
\long\def\ifblank#1\then{\Ifbl@nk#1@@..\then}%
\long\def\Ifbl@nk#1#2@#3#4\then{\if#3#4}
\catcode`\@=12
```

```
\long\def\test#1{\begingroup \toks0{[#1]}%
  \newlinechar`\/\message{/\the\toks0:
  \ifblank{#1}\then EMPTY\else NOT empty\fi%
}\endgroup}
```

End solution

The good thing about this solution is that it doesn't subject any part of the user-supplied argument to the `\ifx` test. Using @ with category code of 11 as a delimiter for the user-supplied text is extremely safe because even in internal code @ doesn't appear by itself, only as part of control sequence names. In a partial solution, Peter Schmitt pushed the idea a little further by using space with category code 3 as the delimiter.

There is another way of handling the problematic `\iffalse` test, in a plain-TeX style solution, by using a suggestion of Donald Knuth that appeared in TeXhax a while ago, in reply to a query of Stephan von Bechtolsheim (texhax89, #38 (post from svb, 17 Apr 89)).

Solution 3 (Arseneau/Knuth)

```
% Usage: \if\blank{#1}...\else...\fi

\catcode`\@=11 % as in plain.tex
\long\def\blank#1{\bl@nk#1@@..\bl@nk}%
\long\def\bl@nk#1#2@#3#4\bl@nk{#3#4}
\catcode`\@=12

\long\def\test#1{\begingroup \toks0{[#1]}%
  \newlinechar`\/\message{/\the\toks0:
  \if\blank{#1}EMPTY\else NOT empty\fi%
}\endgroup}
```

End solution

At the end of Exercise 2 I wrote:

> The two tests on the first line should produce a message 'EMPTY' and the remaining ones, 'NOT empty'. The reason for saying that the second test should return 'EMPTY' is that (1) this is the ideal behavior for the applications I've encountered so far; (2) at least one other person working independently arrived before me at a solution essentially identical to mine, including this behavior.
> The details and credit to the other guy will be given at solution time.

The name of the 'other guy' is Michael Wester; a listing of his macros was published in the preprints for the July 1991 TUG meeting in Dedham, Massachusetts ('Form Letter in LaTeX with 3-across Mailing Labels Capability', joint paper with Jackie Damrau). In rereading the preprint recently, it seems to me the presentation is more different from Exercise 2 and its solutions than I had previously imagined, but the essential ideas are there. See `\wcar`, `\wcdr` and related macros.

By the way, if anyone came up with a fully expandable test (suitable for use inside a `\message`) for which `\test{ }` came up false instead of true, I would be interested to hear about it. I didn't mean to eliminate that possibility in my original statement of the problem.

3 Discretionary

3.1 EXERCISE (FAST)

(Ed: Originally posted on 1991/10/10. Archived as `exercise.003`*.)*

What's the most important difference between `\-` and `\discretionary{-}{}{}` ?

3.2 ANSWERS

(Ed: Originally posted on 1991/10/25. Archived as `answer.003`*.)*

The most important difference between `\-` and `\discretionary{-}{}{}` is that the latter always puts in the character from font position 45 ("2D, '55) of the current font when a word must be broken at the end of a line; `\-` puts in the character from font position `\hyphenchar` of the current font, which is NOT NECESSARILY position 45. It would be rather unusual for `\hyphenchar` to be something other than 45; in certain special applications, however (possibly in some foreign languages as well?) a variant value of `\hyphenchar` can be useful. I have an idea for using this in a future exercise...

Credit to Donald Arseneau for a correct answer. Thanks to Peter Schmitt for providing the perfect opening for another point I wanted to make:

> The *TeXbook* states explicitly:
> `\-` is equivalent to `\discretionary{-}{}{}`
> and both are internal.
>
> I do not see where to the question aims:
>
> - control symbol : control sequence
> - no paramaters : three parameters
> - two characters : 21 characters to type
> - ???

Schmitt is quoting from the last page of Chapter 25; the point is, that in newer versions of the *TeXbook* that sentence has been revised. I'm not sure what the latest printing says, since I don't have a copy, but I think it simply refers the reader to Appendix H, where the significance of `\hyphenchar` is explained. `\hyphenchar` is a feature that was added late in the development of TeX82 (`TeX82.bug` reveals that is was not added until May 25, 1983). Even if the source files for the *TeXbook* were immediately updated by Knuth at that time, the changes did not appear in the published version being sold to the general public until some time later when the first revised edition was published, which was no earlier than October 1984, the date of the *TeXbook* copy that I have on hand, and probably later.

The statement of purpose in 'Around the bend' #1 said something about finding the 'best solution', but conspicuously failed to define what 'best' should mean in this context. It was my intention to address this question in future exercises; for now, let me just say that I don't intend to arbitrarily rule out of consideration answers such as Schmitt's 'two characters : 21 characters to type', since depending on how you look at it, it could be argued that this is much more significant than dumb old `\hyphenchar` minutiae. I promised that these exercises would be challenging; that means, among other things, that they won't always be well-defined, well-bounded, or well-behaved, and part of the job of finding the 'best solution' will be to decide what parts of the problem need to be specified further, and to examine the ramifications of alternatives.

4 What is ‘best’?

4.1 EXERCISE (ESSAY)

(Ed: Originally posted on 1991/11/04. Archived as `exercise.004`*.)*

The statement of purpose in ‘Around the bend’ #1 said something about finding the ‘best solution’, but failed to define what ‘best’ should mean when comparing pieces of TeX code. I’ll start by throwing out a few ideas.

Simplicity A good solution gets hold of the essential idea of the problem and attacks it directly, rather than beating around the bush and resorting to separate clauses to handle troublesome subcases.

Economy If two solutions compare equal in other respects, then the better solution is the one that uses less of TeX’s resources (main memory, hash table, string pool, and so forth). Therefore I (immodestly) say that my solution to Exercise 1 was ever so slightly better than the other two given, because it avoided introducing any auxiliary macros that were not included in the original statement of the problem.

Robustness If a solution only works under limited friendly circumstances, and otherwise blows up with an error message, that’s not good. My solution to Exercise 2 was flawed in this respect, since D.A. found a test case that caused it to go wrong.

*** Exercise 4 (essay):

What should ‘best’ mean when comparing solutions to an ‘Around the bend’ exercise? What qualities of a good solution are most important? Why? How can they be objectively measured? (Or can they?) On the negative side, what qualities indicate an inferior solution?

Table of special characters, to verify accurate transmission:

```
ASCII 33:  ! exclamation point       ASCII 60:  < left elbow
ASCII 34:  " double quote            ASCII 61:  = equals sign
ASCII 35:  # number/pound sign       ASCII 62:  > right elbow
ASCII 36:  $ dollar sign             ASCII 63:  ? question mark
ASCII 37:  % percent sign            ASCII 64:  @ at sign
ASCII 38:  & ampersand               ASCII 91:  [ left square bracket
ASCII 39:  ' right quote/apostrophe  ASCII 92:  \ backslash
ASCII 40:  ( left parenthesis        ASCII 93:  ] right square bracket
ASCII 41:  ) right parenthesis       ASCII 94:  ^ circumflex/hat/caret
ASCII 42:  * star/asterisk           ASCII 95:  _ underscore
ASCII 45:  - hyphen                  ASCII 96:  ` left quote
ASCII 47:  / slash                   ASCII 123:  { left curly brace
ASCII 58:  : colon                   ASCII 124:  | vert bar
ASCII 59:  ; semicolon               ASCII 125:  } right curly brace
                                     ASCII 126:  ~ tilde
```

4.2 ANSWERS

(Ed: Originally posted on 1991/12/10. Archived as `answer.004`*.)*

Peter Schmitt writes:

What is to be rated as 'best' clearly depends on the function used to measure quality. And therefore the question makes sense only with respect to some particular rating function. Seemingly nothing is gained by this statement: Instead of discussing what qualities are required for a good solution one has to discuss how the rating system should be defined. But nevertheless this shifted point of view has an important an important advantage. It makes clear that there is no unique answer: Quality is not an absolute notion but a notion relative to some (agreed) measure. This measure is not independent of the context — under different conditions different rating functions may be used.

One further important point must not be forgotten: If matters of personal taste are to be excluded than the measuring function has to be precisely defined — demanding simplicity, without giving this notion a precise (formal) meaning, is not sufficient.

Therefore I would like to split the original question into two seperate questions:

(a) What (formal and informal) rating functions are likely to be useful, and under what circumstances?

(b) With respect to some formal rating function, is there always a best solution?

Some answers to the first questions are the following (no completeness claimed or even intended):

(1) the first solution:

If some special effect is needed for a single application then the best solution is the first solution (the solution that can be realized with the least effort). This is, however, a purely individual criterion that cannot be formalized.

(2) the most economic (in some sense) solution:

Economic considerations are important if a code is used frequently, Depending on the nature of the applications running time, memory usage, and others, may be relevant. But the time spent for finding a good solution still cannot be neglected in a real world situation. Of course, for theoretical investigations the time spent for research does not matter.

(3) the more robust solution:

If some set of macros is used by a large number of people who not always know how to use them correctly (or even do not care to know) then it is certainly an advantage if they are robust, i.e. work in as many cases (even strange ones) as possible. But again, one has to decide what price (in terms of resources) is acceptable for this robustness. (In many cases the item (4) below will be more important.)

(4) ease-of-use:

If a set of macros is used frequently (by one or more persons) then ease-of-use is certainly a mark of quality: easy to remember syntax, short commands, natural and good readable embedding into the surrounding text, and similar criteria, decide about this.

(5) simplicity:

Simple solutions certainly have a strong appeal — but what is a simple solution? Again this is hard to formalize, since simplicity basically is an aesthetic value, closely related to the concepts of elegance and beauty. (This is similar

> to the situation in mathematics.) But be careful: Simple is not equivalent to short!
>
> (6) the shortest solution:
>
> This may seem to be an easy rating function, but is it? Should length be measured by the number of characters (probably not!), or by the number of tokens, or by the number of control sequences? Or by something else?
>
> Most of the measures mentioned are difficult to formalize, or cannot be formalized at all. Only the resources used (in (2)) and the length of a code (in (6)) can be precisely defined. Therefore, with respect to one of these cases two solutions of the same problem can be compared. Furthermore, in many cases it will be possible to proof that an optimal solution exists. (For instance, since the length of a code (in any interpretation) is a positive integer, there must exist one or more solutions with minimal length, provided there is at least one solution.) But unfortunately this does not imply that one is able to construct an optimal solution, or to decide whether a given piece of code is an optimal solution (or at least near to one). And in some cases it may happen that no optimal solution exists, e.g. if to every solution there is better — but longer! — one.
>
> What is the conclusion of all this? That there may be a best solution relative to some side conditions. But that there is no globally best solution. This statement is, of course, not very satisfying. One would rather prefer to have at least some notion (even a tentative one) of a best solution than none at all. I propose therefore the following informal definition (often subject to personal taste): If some code is optimal or near-optimal in more than one category then it is probably as near to a globally optimal solution as this is possible.

My comments:

I propose the following list, based on (1) [my interpretation of] Knuth's ideas about good macro writing as demonstrated in the *TeXbook* and plain.tex, (2) various articles in TUGboat, (3) Schmitt's comments, (4) discussions I've had in the past with other macro writers, and so forth.

The characteristics of a good solution to an 'Around the bend' exercise are (in order of decreasing importance):

1. Robustness
2. Brevity (= minimal usage of TeX's main memory)3
3. Simplicity
4. Ease of use
5. Suitable commentary
6. Speed
7. Minimal hash table load
8. Minimal save stack load
9. Minimal load in other categories of TeX's memory
10. Comprehensive test suite (when applicable)

Schmitt's point about 'first solution' is well taken but does not apply to 'Around the bend' exercises, because of the stated goal of finding a 'best' solution, with the presumption that normally more than one solution will be found.

Measurement of these qualities is not too difficult, I think, except for 3 and 5. Here's how I see the measurements:

1. **Robustness** A solution is robust if no one who reads it offers a counterexample that causes it to fail. If two solutions both fail, the one with more counterexamples is less robust; if two solutions have different counterexamples, the solution whose counterexample is more likely to occur in normal use is the less robust solution.
2. **Brevity** Of two different solutions, the one that is briefer/shorter/more compact is the one that uses less of TeX's main memory as measured by `\tracingstats`.
3. **Simplicity** Of two different solutions, the shorter one (in the sense of the previous item) is usually the simpler one, but not always. A solution that condenses all the necessary operations into a dense, incomprehensible Gordian knot is less simple than a longer solution that lays out the operations in a series of easily comprehended steps. A solution that relies on arcane dirty tricks is less simple than a solution that uses better-known techniques in a straightforward approach.
4. **Ease of use** I believe this will not be extremely hard to measure in the context of the particular application; it can't sensibly be discussed out of context.
5. **Suitable commentary** The commentary surrounding a solution should explicitly mention any necessary assumptions. If the code is complex, the commentary should give an outline or overview of the intended algorithm. It should explain the operation of any macro if its operation is not evident from the code. If an unusual construction is used where a different construction would normally be expected, the commentary should give the reason.
6. **Speed** Of two solutions, the speedier one is the one that runs faster on common computer systems. If one solution runs faster and slower than another, depending on the system . . . well, let's not cross that bridge unless it turns out to be real.

7,8,9. Minimal hash table load, save stack load, etc. These can be measured by `\tracingstats`.

10. **Comprehensive test suite** If two solutions are equal in other respects, the one whose accompanying test suite covers more distinct cases than the other's is better by that much.

It may be argued that I have not sufficiently answered the question of subjectivity. For example, who's to decide what's an 'arcane dirty trick' and what's not? What does 'suitable' mean in number 5? The answer is that I will say that something is an 'arcane dirty trick' if I think so, and anyone else can do the same. In most cases I believe that there will be general agreement on such a question; if not, and an ensuing discussion fails to reach a clear settlement, then each of the solutions in question will be decreed 'subjectively just as good as the others'.

Other qualities of a good solution can be expressed in terms of the ones listed above. For example, self-sufficiency may be considered an aspect of robustness—if a solution is not entirely self-sufficient, it can easily be shown to be not robust by giving a counterexample that exploits the assumption that makes the solution non-self-sufficient. Elegance? If a solution is simple and easy to use, then I say it is elegant. A solution doesn't necessarily have to be robust in order to be elegant, nor even short (although of two solutions that are otherwise equal, the shorter one is undoubtedly more elegant).

5 `\string` tokens

5.1 EXERCISE (FAST)

(Ed: Originally posted on 1991/11/04. Archived as `exercise.005`*.)*

Assuming a normal value for `\escapechar`

```
\string\a
```

produces two character tokens. What is the category code of the second? Write an experiment (as short as possible) to demonstrate the correctness of your answer.

5.2 ANSWERS

(Ed: Originally posted on 1991/12/05. Archived as `answer.005`*.)*

The category of the 'a' token is 12. All tokens produced by `\string` have category 12, except for space tokens, which have category 10.

Solution 1 (mine)

```
\def\answercheck#1#2{\message{#2: \ifcat0#2\else NOT \fi Category 12}}
\expandafter\answercheck\string\a
\answercheck bb
```

This produces on screen the following message:

```
a: Category 12 b: NOT Category 12
```

End solution

Solution 2 (Peter Schmitt)

```
\def\test#1#2#3{%
  \message{\ifcat#2#3 #2 and #3 have the same category code
                \else #2 and #3 have not the same category code
           \fi}}

\def\Test#1#2#3{%
  \ifcat#2#3 \message{#2 and #3 have the same category code}
       \else \message{#2 and #3 have not the same category code}
  \fi}

\catcode`\A12
\test 1aA
\Test 1aA
\expandafter\test\string\a A
\expandafter\Test\string\a A
```

Comment:

I have given two essentially equivalent Tests — `\test` and `\Test`.

(i) `\test` is slightly more simple because it contains only one `\message` command, but I think that `\Test` is more adequate because it avoids to perform the test inside the `\message` — there might be some side effect one is not aware off.

(ii) Both tests are not as short as possible — the `\true` and `\false` cases could be much shorter, e.g. a T (for true) and a F (for false) would suffice — the result could be checked in the dvi-file. (I regard this difference as inessential.)

Furthermore, setting the catcode of the model character to 12 could easily be omitted (use some character that is known to be an ‘other character’), but I think it should be included: It makes the test independent of any assumption on the format running. This makes the solution more closed and selfsufficient, and therefore also simpler and more elegant (if I may say so).

End solution

6 Counting arguments

6.1 EXERCISE (HARD)

(Ed: Originally posted on 1991/11/04. Archived as `exercise.006`*.)*

Define a macro `\args` that can be used to fill in the proper number in the following sentence no matter how `\foo` is defined (except you may assume it is not `\outer`).

The macro `\tt\string\foo` has `\args\foo` arguments.

Is it possible to solve this if `\foo` is `\outer` also? Is it possible to make `\args` fully expandable, so that it could be used in a message:

```
\message{The macro \noexpand\foo has \args\foo\space arguments.}
```

6.2 ANSWERS

(Ed: Originally posted on 1991/12/23. Archived as `answer.006`*.)*

This was a tough one. All who sent in answers to this exercise (counting myself) used the approach of applying `\meaning` to `\foo` and analyzing the resulting string. There are some drawbacks to this.

(1) In a `\meaning` string, all characters (other than spaces) have catcode 12. This means that all occurrences in a `\meaning` string of the character # are indistinguishable, regardless of their true significance in the parameter text or replacement text of the macro in question. Consequently, an occurrence of a # character, not category 6, followed by a number, in the parameter text of `\foo` can potentially make `\args` report an incorrect number of arguments. For example, in the following definitions `\foo` has no arguments, only delimiter text, in all three cases, but the `\meaning` string would appear to show that `\foo` has one argument:

```
\def\foo\#1{}
\expandafter\def\expandafter\foo\string #1{}
\catcode`\#=12 \def\foo#1{}
```

(2) The following two examples produce identical `\meaning` strings:

```
\def\foo&1{} % no arguments
\catcode`\&=6 \def\foo&1{} % one argument
```

(The string is `"macro:&1->"`.) I.e., characters other than # can be used to create parameter markers in a macro definition, and such a parameter marker cannot be distinguished in a `\meaning` string from a normal use of the character in question.

(3) There is no completely general way to isolate the parameter text of an arbitrary macro from the replacement text. The best you can do is remove the tail of the `\meaning` string—everything after the last occurrence of `->` in the string—and say 'This is not part of the parameter text'. Likewise, anything preceding the first occurrence of `->` is certainly part of the parameter text. If there are two or more occurrences of `->` in the string, however, you cannot say for sure whether anything between the first and last occurrences is parameter text or replacement text. This raises a slight additional possibility that pseudo 'parameter markers' in the replacement text could cause `\args` to give an incorrrect result. For example:

```
\edef\foo #1{\string#2->}
```

defining `\foo` with one argument, produces a `\meaning` string of

```
macro:#1->#2->
```

which is exactly the same as the `\meaning` string for

```
\def\foo#1->#2{}
```

where `\foo` has two arguments.

Speaking practically, however, rather than theoretically, using `\meaning` to analyze the number of arguments of an arbitrary macro works fine. Donald Arseneau's solution, below, is admirably brief and demonstrates an easy way of handling an outer argument that I had never seen before.

Solution 1 (Donald Arseneau)

> Here is my solution for counting arguments. It is totally expandable, and relies on the fact that the parameter numbers must be in increasing order, that they are only single digits, and that there is no parameter zero. Also important is that `\meaning` of a macro defined by `\def\x#{...}` reports a syntax of `{` rather than #.

```
{\catcode`\*=6 \catcode`\#=12 % use * for macro parameters while # is "other"
%
\gdef\args{\expandafter\Args\noexpand}% get rid of \outerness
%
\long\gdef\Args*1{\expandafter\countargs \meaning*1:->{}\end}%
%  ... \meaning will display the parameter syntax (as "other" characters).
%
\gdef\countargs*1:*2->*3\end{\twoargs#0*2#0}% get just the parameter syntax
% ... in format #0junk#1junk...#njunk#0.  \twoargs processes the list to
% ... give "n", the last number before #0.
```

> Here's what tests the parameter numbers, two at a time. (Thus, the two `#0`'s in `\countargs`, so there are always at least two `#n`'s detected.) When the second number of a comparison isn't zero, `\twoargs` re-executes itself to test the next pair; when the second `n` is 0, the first `n` is the highest parameter number, so it is output.

```
\gdef\twoargs*1#*2*3#*4{\ifnum0=*4 *2\else % note the space to end the number
  \expandafter\twoargs\expandafter#\expandafter*4\fi}
}
```

> Here is my test suite. The character ":" works in a funny way: it confuses how `\countargs` reads its parameter list, and another colon gets into the supposed syntax. But it works because there are no parameters. The primitive `\halign` is reported to have no parameters because it is not a macro. This could be confusing to someone. The same confusion could arise with `\args` itself because it doesn't read the parameter right away.

```
\def\test#1#{nothing}
\def\Test[#1]#2:{\##1,#2##}
\def\#{haha}

\show\test \show\Test
```

> (I condensed this test suite—MJD)

```
\long\def\msg#1{\message{The object \string#1 has \args#1 arguments.}}

\msg\mathpalette \msg\mathhexbox \msg\par \msg\halign \msg\args
\msg\relax \msg # \msg\# \msg\test \msg\Test \msg : \msg\: \msg\csname
```

```
\msg t \msg ~ \msg $ \msg ^
```

(Outer macros—MJD)

```
\message{The object \string\bye\space has \args\bye\space arguments.}
\message{The object \string\newhelp\space has \args\newhelp\space
  arguments.}

\bye  % --  Donald Arseneau
```

End solution

Although the problem statement only mentioned ‘macros’ Arseneau earned some thoroughness points by including primitives `\halign`, `\relax`, and `\csname`, as well as characters `# : t $ ^` in his tests. This is of some interest because of the difference in `\meaning` strings between macros and non-macros.

In my solution for this exercise, I amused myself by trying to pack everything into as few control sequences as possible. Although I got it down to two, that's really only one less than Arseneau's four, because one control sequence in his solution is expended to handle outer macros, something my solution didn't attempt to do.

Solution 2 (mine)

```
%   Use & instead of # temporarily.
\catcode`\&=6 \catcode`\#=12

\long\def\args &1{\expandafter\countargs\meaning &1#\args->\countargs 0}
```

Analysis is restricted to the parameter text by chopping off everything after `->` in the meaning string (this will leave possibly only part of the parameter text).

Then we look in the parameter text for # followed by a number (checking to make sure that the thing after # is a number handles a few extra possibilities, such as `\#` followed by non-number in the parameter text). If we find # plus a number, we pass the number onward to the next invocation of `\countargs`, where it will end up as the returned value (argument #5) if the next `\countargs` determines that the remaining parameter text contains no more parameter markers.

```
\def\countargs &1#&2&3->&4\countargs &5{%
  \ifx\args&2&5%
  \else
    \ifodd0&21 % Then &2 is a number, carry forward.
      \countargs&3#\args->\countargs&2%
    \else % &2 not a number---ignore, carry forward last number instead
      \countargs&3#\args->\countargs&5%
    \fi
  \fi}

\catcode`\#=6

\def\test{\message{The macro \noexpand\foo has \args\foo\space
  arguments (\meaning\foo).}}

%\tracingmacros=2 \tracingcommands=2
% Success:
\def\foo{No args}\test
```

```
\def\foo#1{One arg}\test
\def\foo#1#2{Two args}\test
\def\foo./{No args, delimited}\test
\def\foo#1#2#3#4#5#6#7#8#9{Nine args}\test
\def\foo//#1#2#3#4#5#6#7#8#9//{Nine args, delimited}\test
\def\foo#{Weird}\test
\def\foo#1#{Weird, one arg}\test
\def\foo#1#2#3#4#5#6#7#8#9#{Weird, nine args}\test
\def\foo#1 {One arg, space delimited}\test
\def\foo#1 #2 #3 #4 #5 #6 #7 #8 #9 {Nine args, space delimited}\test
\def\foo/{\def\foo}
\foo/ #1{Interesting}\test

\edef\foo#1#2{\string #3\string #4}\test
\edef\foo{\string #}\test
\expandafter\edef\expandafter\foo
  \csname 0\string #\string #\endcsname#1#2{#1#2}\test

% Failure:
\def\foo->#1->#2->#3->#4->#5->#6->#7->#8->#9->{Nine args, devious
  delimiter}\test
\expandafter\edef\expandafter\foo
  \csname 0\string #1\string #2\endcsname{...}\test
\let\foo=\bye \test % \outer bomb
```

End solution

When I originally posed this problem, I had seen far enough ahead to suspect that the drawbacks of `\meaning` mentioned above would be impossible to overcome. But `\meaning` is the only way to analyze a macro that has a nonsimple parameter text—that is, one containing delimited arguments. Another possibility I had in mind was restricting the analysis to macros with simple parameter texts—empty or having only nondelimited arguments—to see what might be done without `\meaning`. The best that I could manage in my experiments along these lines was a definition of `\args` with an unacceptably cumbersome call syntax. But it does have the virtue of correctly identifying any number of nondelimited arguments, no matter whether `\foo` was originally defined using # (category 6) or some other category 6 character.

Solution 3 (mine)

```
% This solution is not fully expandable, hence cannot be used
% inside a \message.

\def\args{\expandafter\argscontinue}

\def\argscontinue{\begingroup
```

Make all digits have category 2 (= end of group) so that they will serve to end the token register assignment `\global\toks1 ...`

```
  \catcode`\0=2 \catcode`\1=2 \catcode`\2=2 \catcode`\3=2 \catcode`\4=2
  \catcode`\5=2 \catcode`\6=2 \catcode`\7=2 \catcode`\8=2
```

We use `\afterassignment` to put an `\endgroup` after the token register assignment, so that numbers will revert to their ordinary catcodes. And we use `\aftergroup` to put a `\finishup` token after the `\endgroup`. Thus `\finishup` can look ahead to see what numbers are remaining; this information reveals how many arguments were used up by the `\foo` macro call.

```
  \aftergroup\finishup \afterassignment\endgroup
  \global\toks1\bgroup}
```

`\finishup` takes the first digit following it and returns it as the value of `\args`; any following numbers are discarded (note that #2 is delimited by a space).

```
\def\finishup#1#2 {%\showthe\toks1
  #1}

%\tracingmacros=2 \tracingcommands=2 \tracingonline=1
\def\foo{}
The macro {\tt\string\foo} has \args\foo 00123456789 \ arguments.

\def\foo#1{}
The macro {\tt\string\foo} has \args\foo 00123456789 \ arguments.

\edef\foo#1{\string #2\string #3\string #4->\string #4\string #3#1}
The macro {\tt\string\foo} has \args\foo 00123456789 \ arguments.

\def\foo#1#2#3{a#1b#2c#3}
The macro {\tt\string\foo} has \args\foo 00123456789 \ arguments.

\def\foo#1#2#3#4#5#6#7#8#9{#1#2#3#5#8bb#9}
The macro {\tt\string\foo} has \args\foo 00123456789 \ arguments.
```

End solution

The fourth solution for Exercise 6 is by Peter Schmitt; it gets the robustness prize for carrying out a diligent analysis of `\meaning` strings that enables it to correctly handle a greater variety of exotic cases than the other solutions. Schmitt's original method of handling outer macros was effective, but more complicated than Arseneau's method, incorporated here as noted. Even though my approach was rather different from Schmitt's, some of the comments in Schmitt's solution inspired me in turn to improve my solution [2] from its previous much inferior state.

Solution 4 (Peter Schmitt)

```
% \args <token> expands to:  -   if <token> is not a macro
%                           0..9 according to the number of parameters
%                                if the <token> is a macro
% \args is fully expandable and accepts outer macros as well.
%    It assumes, however, that the tested macro has been defined using the
%    standard parameter symbol #,
%    and that the current value of \escapechar is the standard backslash \.
```

The definition of the macros uses the expansion of `\meaning\cs`: It is of the form:

```
[..] macro: [parameter text] -> [replacement text]
```

and consists of 'other characters'.

The macro `\args` checks:

1. if the expansion contains ‘macro’:
   — if not, then `\cs` is not a macro and `\args` yields ‘-’
2. if the expansion contains parameters #1 etc.
   — if `#n` is the first that is not present then `\cs` takes (n-1) arguments and `\args` yields ‘n-1’

The following special characters are chosen to make the definitions as readable as possible. Any characters having catcodes different from 12 will serve the same purpose:

```
\catcode`\:3 \catcode`\/3  % : and / are used as parameter delimiters
\catcode`\^3               % ^ is used to detect empty arguments
\catcode`\?11              % ? is used to make the control sequences private
```

Since the occurrences of # in the expansion of `\meaning\cs` has to be detected, it has to be used as an ‘other character’. To avoid confusion it has been replaced not only where necessary but throughout all the definitions:

```
\catcode`\#12  \catcode`\*6   % * is parameter character
```

- `\?macro` is defined to be ‘macro’ consisting of ‘other characters’ using the expansion of `\meaning\TeX`.
- \DEF? inserts these five characters into some definitions where they are as parameter delimiters:

  ```
  \DEF\cs { <parameter text> } { <replacement text> }
  ```

  where the texts may contain *1 and **1 .. **9 yields

  ```
  \def\cs <parameter text>{<replacement text>}
  ```

  where *1 is replaced by ‘macro’ and **1 yields *1 etc.

```
\def\?macro *1:*2:{*1} \edef\?macro{\expandafter\?macro\meaning\TeX:}
\def\?DEF *1*2{\def*1**1:{\long\def*1*2}\expandafter*1\?macro:}
```

- `\args` passes the ⟨*token*⟩ unexpanded to `args?`
- (taken from the solution by Donald Arseneau) `\args?` takes one argument, expands its `\meaning` to TEXT and passes it to `\macro?` after appending `macro^`:
- `\macro?` checks the first token after the first occurrence of ‘macro’: if this is `^(3)`, then ‘macro’ was not present in TEXT (output: -) otherwise TEXT is further investigated.

```
\def\args{\expandafter\args?\noexpand}
\?DEF \args? {**1{\expandafter\macro?\meaning **1*1^:}}
   \?DEF\macro? {**1*1**2:{\ifx^**2-\else\expandafter\purge? **2:\fi}}
```

The parameters taken by a control sequence all appear (once and in numerical order) in the parameter text — and no other occurrence of a pair `#n` is allowed in it. Moreover, only the same pairs `#n` may occur in the replacement text. It is, however, not possible to simply look for occurrences of these pairs since there are tokens that may — if followed by some number — be (wrongly) interpreted as parameters:

- the token `##` in the replacement text, and
- (as pointed out by Michael Downes) -the control symbol `\#` both in the parameter text and the replacement text.

Since `\\#n` has to be distinguished from `\#n` the control symbol `\\` is also important.

Therefore `\purge?` is used to remove all occurrences of these tokens. After that the search-macro `\head?` is invoked, appending the sequence `#n^(n-1)` for every possible parameter `#n`.

Since `\purge?` has to identify the character `\(12)` it is necessary to change the escapecharacter:

```
\catcode`\!0 !catcode`!\=12   % ! is used as escape character
```

`\purge?` appends `## \#^ and \\^` to the TEXT as a means to stop the search for these tokens, and : as delimiter:

1. `\backslash?` looks for the first occurrence of the character pair `\\` in TEXT (this must be a token `\\`) and replaces it by a space. If it is followed by `^(3)` then the search is completed, otherwise the process is repeated.
2. `\numbersign?` looks for the first occurrence of the character pair `\#` in the (in the meantime modified) TEXT (since all `\\` have been removed this must correspond to a token `\#`) and replaces it by a space. Again the process is stopped when it is followed by `^(3)`.
3. `\parametersign?` truncates TEXT at the first occurrence of the character pair. Note that this pair must correspond to a parameter token `##` in the replacement text and therefore the rest of TEXT is not needed any more.

```
    !def!purge? *1:{!backslash? *1##\#^\\^:}

% \purge? could be avoided - \macro? could call \backslash? directly

    !def!backslash? *1\\*2*3:{!ifx^*2!expandafter!numbersign?
                               !else !expandafter!backslash?
                                 !fi *1 *2*3:}
    !def!numbersign? *1\#*2*3:{!ifx^*2!expandafter!parametersign?
                                 !else !expandafter!numbersign?
                                   !fi *1 *2*3:}

!catcode`!\0 \catcode`\!=12  % return to the normal use of backslash

    \def\parametersign? *1##*2:{%
               \head? *1^#1^0#2^1#3^2#4^3#5^4#6^5#7^6#8^7#9^8#0^9:}
```

For each n from 0 to 9 `\head?` extracts the characters contained in the (appended) TEXT between the first occurrence of `#n` and `#(n+1)` and investigates them with `\used?`.

If `#n` is not present in TEXT, then the first of these characters is `^(3)`, taken from the appended string:
When this happens for the first time `\used?` outputs the second character (the number of parameters) and calls `\skip?` to hide all the remaining parts of the appended TEXT, otherwise `\used?` checks the next item.

Since eleven parameters are necessary to handle the ten cases (0..9) this duty has to be distributed on two macros:
The appearance of the character `/(3)` is used to indicate that the second macro `\tail?` has to be invoked by `\used?`.

```
\def\head? *1#1*2#2*3#3*4#4*5#5*6:{%
      \used? *2..:*3..:*4..:*5..:/.:%
      \expandafter\tail? *6://}
\def\tail? *1#6*2#7*3#8*4#9*5#0*6:{\used? *2..:*3..:*4..:*5..:*6:}
\def\used? *1*2*3:{\ifx^*1*2\expandafter\skip?\else\ifx/*1\else
                   \expandafter\expandafter\expandafter\used?\fi\fi}
\def\skip? *1//{}
```

```
%% Finally, catcodes are turned back to normal:

\catcode`\#6 \catcode`\*12 \catcode`\?12
\catcode`\:12 \catcode`\/12 \catcode`\^12

%%%%%%%%%%%%%%%%%%%%%%

\long\def\test#1{
     The macro {\tt\string#1} has {\args#1} arguments.

     \message{The macro \noexpand#1 has :\args#1:\space arguments.}
}

\def\exc#1\\#2\ #3{\#4\\#1\\\#4\\\\#2two arguments}
\test\exc

\end
```

End solution

Schmitt's solution assumes the use of mine and Arseneau's test suites as well, because they had been shared between us before Schmitt sent in the final version of his solution.

7 Self replication

7.1 EXERCISE (HARD)

(Ed: Originally posted on 1991/11/04. Archived as `exercise.007`*.)*

In the September 1991 issue of Dr. Dobb's Journal, in an article 'Little Languages, Big Questions' (pp. 16–25), Ray Valdés described a 'little language' as a part of a more complex application that is

> partitioned into two (or more) nested components: a core module that provides a primitive set of services for an application area (the "engine"), and a surrounding module that provides programmatic access to these services. The surrounding module is typically a language interpreter for a simple, easily parsed computer language–a "little language".

Since TeX seems to fall into this category, I wonder if any Dr. Dobb's readers who know TeX tried their hand at the challenge given in a sidebar ('How Strong Is Your Little Language')?

> [An] informal benchmark of a language's computational power is the programming exercise that Ken Thompson (coauthor of Unix) used to pass the time in college. ... The goal is to write the shortest self-reproducing program: "More precisely stated ... to write a source program that, when compiled and executed, will produce as output an exact copy of its source."

When I tried it it turned out to be a real challenge for me. In the Unix world, for conventional compiled languages, the problem as originally stated can assume output on the 'standard output' stream; but TeX already clutters up standard output with some of its built-in messages. This leaves three alternatives in refining the statement of the problem to be meaningful for TeX:

1. Write a TeX program that includes the built-in messages in its source in such a way that it exactly fulfills the the original problem statement with standard output as the output stream.
2. Pretend the built-in messages don't exist and write a TeX program that reproduces an exact copy of itself (with no extra garbage) in the middle of the built-in messages.
3. Write on a different output stream.

Take your pick, any or all of the above, and see what you can come up with. I have solutions for 2 and 3 but have not gotten around to really thinking about 1 yet. I believe it will require at least a different algorithm than the other 2, if it is not impossible.

7.2 ANSWERS

(Ed: Originally posted on 1992/01/07. Archived as `answer.007`*.)*

Plenty of good answers for this one.

Solution 1 (mine)

This solution is type 2 (print the copy in the middle of TeX's built-in messages). It assumes `plain.tex` or similar has been loaded to set the catcodes of the left and right curly braces.

The idea is to assign the text to the token register `\errhelp` (used merely because it is a convenient pre-existing token register), and then print out `\the\errhelp` twice. There is a bit of shuffling to ensure that `\errhelp` will swallow the last half of the file and that the last half of the file is equal to the first half, which contains all the preparations necessary to prepare `\errhelp` for that swallowing and the subsequent message-sending.

A space is left after every control word, because this is easier than trying to prevent TeX from printing spaces after control words when the message is eventually printed on screen.

The lines are carefully arranged to break at column 79 (including spaces) since this is the normal value for `max_print_line`, a constant compiled into TeX which controls the length of screen output lines. It would be easy to make the lines work out nicely no matter what the working code required, by varying the length of the macro name `\selfcopy` and using, say, `\everyhbox` or `\everyjob` instead of `\errhelp`.

The total number of tokens in this solution is 54.

```
{\gdef \selfcopy {\message {{\the \errhelp }}\message {{\the \errhelp }}\end }
\aftergroup \errhelp \afterassignment \selfcopy }
{\gdef \selfcopy {\message {{\the \errhelp }}\message {{\the \errhelp }}\end }
\aftergroup \errhelp \afterassignment \selfcopy }
```

End solution

Solution 2 (mine) This variation is Type 3, writing the copy to a disk file instead of to the screen. The total number of tokens in this solution is 126.

```
\immediate \openout 0=\jobname .cpy
{\gdef ˜#112{\errhelp {#112}\immediate \write 0{\the \errhelp
}\immediate \write 0{\the \errhelp }\immediate \closeout 0 \end}}
\newlinechar 13 \catcode ‘\#=3 \afterassignment ˜\catcode 13=12
\immediate \openout 0=\jobname .cpy
{\gdef ˜#112{\errhelp {#112}\immediate \write 0{\the \errhelp
}\immediate \write 0{\the \errhelp }\immediate \closeout 0 \end}}
\newlinechar 13 \catcode ‘\#=3 \afterassignment ˜\catcode 13=12
```

End solution

I learned from Victor Eijkhout that he had submitted a short article to TUGboat discussing this very problem, well before I asked it here in 'Around the bend'. He kindly sent me a copy of the article, which contains a good discussion of the underlying ideas, and a couple of different solutions. To summarize briefly, he gave a Type 2 solution similar in length to mine, and also a solution that involved printing out the source file on PAPER! A 'Type 4' solution, in other words. I'm a little embarrassed that I didn't think of this, given that the whole idea of TeX is to print things on paper.

Solution 2 (Victor Eijkhout) Forthcoming in TUGboat. It appeared as: ‘Self-replicating macros’ by Victor Eijkhout and Ron Sommeling, TUGboat 13 (1992) no 1, p. 84.

End solution

Although I'm giving them all together, as ‘Solution 3’, Peter Schmitt actually sent in six different variations, including a Type 4 solution. His first solution, `log-pl.tex` is Type 2 like my first solution but comes in at 38 tokens, significantly shorter. His third solution is comparable to my second solution but once again significantly shorter (87 tokens).

Solution 3 (Peter Schmitt) The principal structure of the solution is the following:

```
<initial commands>
\def \run { <additional commands>
            \write { <the initial commands>
                     \def \run
                     {
                     <the replacement text extracted from \meaning\run>
                     }
                     \run
                   }
            <final commands>
          }
\run
```

The following TeX-File `out-ini.tex` when processed by INITeX produces a file `out-ini.out` that is identical to
`out-ini.tex` (case (3) below):

(The file consist of a single line, it is broken up to make comments possible - each occurrence of the comment sign % has to be removed together with the rest of the line to produce identical output.)

```
\catcode ‘\{1 \catcode ‘\}2 \catcode ‘\#6 % these \catcodes are required
\def \run {%                       a macro to called at the end of the file
\immediate \openout 1=out-ini.out%   % opens output
\def \select ##1:->##2{##2}% an auxiliary macro to extract the replacement text
\immediate \write 1{%                  write the output file
\catcode ‘\noexpand \{1 \catcode ‘\noexpand \}2 \catcode ‘\noexpand \#6 %
%                                      writes the first ‘line’ of the output
\noexpand \def \noexpand \run %        writes \def \run
{\expandafter \select \meaning \run }% writes the replacement text of \run
\noexpand \run }%                      writes the last ‘line’ of the program
\immediate \closeout 1%                close output file
\end }%                                close input
\run %                                 start the macro
```

Comments:

1. `\immediate` prevents that a dvi-file is produced.
2. the tex-file can be shortened (less characters) by using shorter names, maybe also by using a controlsymbol for `\noexpand`, both possibilities do not reduce the number of tokens. Maybe some `\space` tokens can be removed but most of them are necessary because they are produced by `\meaning`.
   - `\immediate` may be omitted (produces dvi-file)
   - at least with my implementation closing the output file is not necessary
3. The TeX-file can be modified to solve variations of the exercise:
   - If the file is to be processed by plain TeX `\catcodes` need not be set (see (1) below).
   - if the output file is replaced by standard output or the log file `\message` instead of `\write` can be used (see (1) and (2) below). Note that in this case macro names and spaces have to be adjusted so that the line breaks

produced do not prevent processing the file (In the log file line breaks may occur even in control sequence names!)

I have not (not yet?) been able to solve the exercise using more pleasant (predetermined) linebreaks.

- It is possible to produce a log file that is identical to the input file. But since the log file contains the time of processing this will be the case only at a specific date and time (see (4) below). (The time is output before the input file is read. Therefore it is impossible to change this part of output by the input.)
- Of course, the above variation can be modified to produce a screen output identical to the input file.
- It is possible to pass a verbatim copy of the input to TeX and set it in `\tt`

Some of the variations:

(1) plain TeX `-->` section of log file or standard output terminal

```
%%% log-pl.tex:
\def \run {\def \select ##1:->##2{##2} \message {\noexpand \def \noexpand \run
{\expandafter \select \meaning \run } \noexpand \run } \end } \run

%%% log-pl.log
This is TeX, Version 3.1(c)sb34 (preloaded format=plain3sm 91.4.28)
24 NOV 1991 02:15
** &plain  log-pl
(log-pl.tex
\def \run {\def \select ##1:->##2{##2} \message {\noexpand \def \noexpand \run
{\expandafter \select \meaning \run } \noexpand \run } \end } \run  )
No pages of output.
```

(2) INITeX `-->` section of log file or standard output terminal

```
%%% log-ini.tex
\catcode ‘\{=1 \catcode ‘\} =2 \catcode ‘\#=6 \def \run {\def \selectit
##1:->##2{##2} \message {\catcode ‘\noexpand \{=1 \catcode ‘\noexpand \}
 =2 \catcode ‘\noexpand \#=6 \noexpand \def \noexpand \run {\expandafter
 \selectit \meaning \run }\noexpand \run }\end }\run

%%% log-ini.log
This is TeX, Version 3.1(c)sb34 (INITEX)
24 NOV 1991 02:16
** log-ini.tex
(log-ini.tex
\catcode ‘\{=1 \catcode ‘\} =2 \catcode ‘\#=6 \def \run {\def \selectit
##1:->##2{##2} \message {\catcode ‘\noexpand \{=1 \catcode ‘\noexpand \}
 =2 \catcode ‘\noexpand \#=6 \noexpand \def \noexpand \run {\expandafter
 \selectit \meaning \run }\noexpand \run }\end }\run  )
No pages of output.
```

(3) INITeX `-->` output file

```
%%% out-ini.tex (Note: A single line broken at the %’s!)
\catcode ‘\{1 \catcode ‘\}2 \catcode ‘\#6 \def \run {\immediate \openout %
1=out-ini.out\def \select ##1:->##2{##2}\immediate \write 1{\catcode %
`\noexpand \{1 \catcode `\noexpand \}2 \catcode `\noexpand \#6 \noexpand \def %
\noexpand \run {\expandafter \select \meaning \run }\noexpand \run }%
\immediate \closeout 1\end }\run
```

(4) INITeX --> log file

```
%%% flog-ini.tex
This is TeX, Version 3.1(c)sb34 (INITEX)
24 NOV 1991 02:17
** flog-ini.tex
(flog-ini.tex
\catcode `\{=1 \catcode `\} =2 \catcode `\#=6 \def \run {\def \selectit
##1:->##2{##2} \message {\catcode `\noexpand \{=1 \catcode `\noexpand \}
 =2 \catcode `\noexpand \#=6 \noexpand \def \noexpand \run {\expandafter
 \selectit \meaning \run }\noexpand \run }\end }\run  [0] )
Output written on flog-ini.dvi (1 page, 512 bytes).

%%% flog-ini.log
This is TeX, Version 3.1(c)sb34 (INITEX)
24 NOV 1991 02:18
** flog-ini.tex
(flog-ini.tex
\catcode `\{=1 \catcode `\} =2 \catcode `\#=6 \def \run {\def \selectit
##1:->##2{##2} \message {\catcode `\noexpand \{=1 \catcode `\noexpand \}
 =2 \catcode `\noexpand \#=6 \noexpand \def \noexpand \run {\expandafter
 \selectit \meaning \run }\noexpand \run }\end }\run  [0] )
Output written on flog-ini.dvi (1 page, 512 bytes).
```

(5) INI-TeX --> log-file (formatted)

```
%%% fmt-log.tex
This is TeX, Version 3.1(c)sb34 (INITEX)
30 NOV 1991 13:13
** fmt-log
(fmt-log.tex [0
\catcode `\{=1 \catcode `\}=2
\catcode `\#=6
\def \run
{\newlinechar 1 \lccode `\|=1
 \lccode `\[=`\{ \lccode `\]=`\}
 \lowercase {
 \def \format ##1>##2=1##3]##4[##5]##6]{##2=1|##3]|##4[|##5]|##6]|\+}
 \def \+ ]##12]##2]##3]##4]]##5] { ]|##12]|##2]|##3]|##4]]|##5]|}
 }
 \write 0{\catcode `\noexpand \{=1 \catcode `\noexpand \}=2}
 \write 0{\catcode `\noexpand \#=6}
 \write 0{\noexpand \def \noexpand \run }
 \write 0{{\expandafter \format \meaning \run }}
 \write 0{\noexpand \run }
\end }
```

```
\run
] )
Output written on fmt-log.dvi (1 page, 512 bytes).
```

(6) INITeX --> dvi-file

```
%%% dvi-ini.tex
\catcode‘\% = 13
\catcode‘\{ = 1 \catcode ‘\} = 2
\catcode‘\# = 6 \catcode ‘\| = 13
\catcode‘\% = 13
\def \run {
 \lccode ‘\[=‘\{ \lccode ‘\]=‘\} \lccode ‘\/=‘\% \let % = \par %%
 \font\tt=cmtt10 \tt %
 \hsize 15cm \vsize 15cm \parskip 3pt \def |{\par \hskip .5em} %
 \lowercase { %
 \def \fmt ##1>##2//##3/##4/##5/##6/##7/{|##2//|##3/|##4/|##5/|##6/|##7/|\+} %
 \def \+ ##1/##2/##3/##4//##5/##6/##7/{##1/|##2/|##3/|##4//|##5/|##6/|##7/|} %
 } %
 \string \catcode ‘\string \{ = 1 \string \catcode ‘\string \} = 2 %
 \string \catcode ‘\string \# = 6 \string \catcode ‘\string \| = 13 %
 \string \catcode ‘\string \% = 13 %%
 \string \def \string \run \lowercase { [} %
 \expandafter \fmt \meaning \run  \lowercase {]} %
 \string \run %
 \end }
\run
```

End solution

8 `\end` too soon

8.1 EXERCISE (HARD)

(Ed: Originally posted on 1993/06/21. Archived as `exercise.008`*.)*

A few readers of info-tex and comp.text.tex may recall some postings of mine under the name of 'Around the Bend' more than a year ago. This was intended to be a regular quasi-monthly stream of challenging questions about TeX macro writing, but after a few appearances it fell into limbo because of too many other demands on my time. However I continue to encounter hard, interesting problems in my work so herewith wish to announce resumption of the 'Around the Bend' postings on an occasional, slightly less ambitious basis.

For background, here are a couple of excerpts from the first 'Around the Bend' post:

> With the encouragement of George Greenwade (the INFO-TeX list owner), I would like to propose a regular department for INFO-TeX, called 'Around the bend'. It will consist of macro-writing challenges on the level of the dangerous-bend exercises in the *TeXbook*, with interested parties invited to collaborate and/or compete to find the best solution. My motivation for doing this is partly selfish: to get more feedback from other macro writers about some of the interesting macro-writing problems that I run into.
>
> ...
>
> Solutions should be sent to me instead of to INFO-TeX or comp.text.tex, on the premise that people usually won't want to read others' solutions until they've had a chance to try their own hand. A summary of the results would then be posted to the INFO-TeX list after two or three weeks; to those who submit solutions before the deadline, I could forward without delay solutions submitted by other people, for comparison.

And here's number 8.

Under certain conditions, TeX fails to give an error message for a missing closing brace or `\endgroup` or `\fi`; it only gives an unobtrusive warning message after the end of the TeX run, which is easy to overlook:

```
(\end occurred inside a group at level 1)
(\end occurred when \iffalse on line 6 was incomplete)
(\end occurred when \iftrue on line 3 was incomplete)
```

Is there any way to trap these conditions and give a true error message?—if, let's say, you are programming for a major macro package like LaTeX and want to make sure these conditions are brought to the user's attention.

Remark Off-hand one would think that trapping these conditions is impossible, since otherwise Knuth would presumably have built the trapping into TeX; `\iffalse` ... `\end` generates an error message, it's only `\iffalse` ... `\else` ... `\end` or `\iftrue` ... `\end` that leave TeX mumbling instead of shrieking. But in some cursory experiments, I found a not-too-bad solution for the missing end of group condition. I'd be pleased to see someone else come up with a better solution, however, as well as a solution to the missing `\fi` problem.

8.2 ANSWERS

(Ed: Originally posted on 1993/07/22. Archived as `answer.008`*.)*

This review of solutions is posted later than expected because I needed time to try out and understand solutions submitted by Peter Schmitt last week. For clarity's sake, I have split the solutions into two parts, one dealing with groups, the other with conditionals.

8.2.1 Groups

Peter Schmitt remarked that if TeX can give a warning message for a missing endgroup there is nothing to prevent it from giving an error message except the choice of TeX's author. In some cursory perusal of *TeX: the Program*, I wasn't able to find any explanation from Knuth as to why he didn't make it a real error message instead of just a warning. Perhaps someone else can shed some light here?

Now for solutions. The first one was submitted by Peter Schmitt. My commentary: Assume the body of the TeX document is enclosed within start and end commands (here named `\BEGIN` and `\END`), with the starting command contributing a `\begingroup` and the closing command providing the matching `\endgroup`, with some juggling to make a group mismatch trigger an error.

If the document contains any unclosed groups that were opened with `{` or `\bgroup`, the `\endgroup` will trigger TeX's low-level error recovery, which is to insert matching `}`s (`‘Missing } inserted’`). Thus only the case of an unmatched `\begingroup` needs to be handled. Schmitt does this by (essentially) making a local redefinition of `\end` that produces an error message; if all groups are closed properly, the local definition will disappear, restoring the normal definition, which will execute a normal endgame.

Here now Schmitt's submitted solution. I have simplified it slightly by disentangling some other stuff that will be discussed later below.

Solution 1 (Peter Schmitt)

```
\catcode`_11

\let\standard_end\end                  % save original meaning of end
                                       % define modified end
\def\unexpected_end{%
  {\errorcontextlines=0               % minimize errormessage
  \errmessage{Unexpected \string\END\space inside group}% errormessage
  }\standard_end                      % continue with \standard_end
}

\let\End\standard_end

\def\END{\endgroup\End}

\def\BEGIN{\begingroup
           \let\End\unexpected_end}

\BEGIN
```

```
%%% some tests:

% \bgroup\egroup\end                                  % balanced
 \begingroup\end \endgroup                           % unbalanced
% \bgroup\end                                         % unbalanced
% { \end                                              % unbalanced

% } \begingroup \end                            % this is reported
% \endgroup \begingroup \end                    % this is not reported
```

End solution

Solution 2 (mine) This solution uses a rather dirty trick with `\batchmode`. Jonathan Fine also found the same idea, though in his mail to me he did not elaborate it into a fully wrapped solution.

Enclosing the entire document inside a `\begingroup \endgroup` places an extra burden on the save stack (one would presume this is why LaTeX's `\begin{document}` and `\end{document}` take some pains to avoid constructing such a group, although the comments in `latex.tex` don't provide an explicit reason). (Extra credit question: Just how much of a burden would it place on the save stack in, say, an average LaTeX document?) So my solution seeks to trap unmatched `{` or `\begingroup` without enclosing the document body in a group. The reason the `\batchmode` trick is 'dirty' is that it leaves a spurious extra error message in the log file. On screen for the typical interactive user, this error message is hidden by the temporary switch to `\batchmode`, but if for example the user has as part of their TeX system an editor setup that automatically proceeds through the `.log` file to help the user take care of all error messages, then the spurious error message will be somewhat inconvenient.

The following clip shows what a user would typically see on screen if their document contained an unmatched `{`.

```
! Missing } added.
\bgrouperr ...ffalse {\fi \string } added}

\enddocument ...rgroup \bgrouperr \egroup
                                          \if \errorstopping \batchmo...
l.50 \enddocument

? h
There appears to be an unmatched opening brace or \bgroup somewhere
in your document.
?

 )
No pages of output.
```

Here then is the code for the solution. As it stands, only the most recent unmatched open-group is dealt with in the error message. As the on-screen result from the test section marked as 'test 2' will indicate, a recursive definition for `\bgrouperr` would be better for maximum robustness, but I haven't had the spare time to work out the extra details.

```
\def\enddocument{%
%    Go into \batchmode to suppress possible error messages that we
%    don't want to bring to the user's attention.
  \batchmode
%    Set a flag to enable us to handle the \endgroup properly if the
%    \egroup pairs up with an unmatched { or \bgroup.
  \def\errorstopping{TF}%
%    If the following \egroup matches with a preceding unmatched { or
%    \bgroup in the user document, then the aftergroup tokens
%    \errorstopmode \bgrouperr will be executed. Otherwise they will
%    go away into uncharted limbo.
  \aftergroup\errorstopmode\aftergroup\bgrouperr
  \egroup
%    If there was no unmatched { or \bgroup, then the preceding
%    \egroup was discarded by TeX. And \errorstopping is still false.
%    Otherwise we need to insert some new \aftergroup tokens.
  \if\errorstopping
    \batchmode \aftergroup\errorstopmode \aftergroup\begingrouperr
  \else
    \global\let\bgrouperr\begingrouperr
  \fi
  \endgroup
  \errorstopmode
%    Call two different versions of \end, just for convenient testing
%    with either plain TeX and LaTeX.
  \csname\string @\string @end\endcsname
  \end}

\def\bgrouperr{%
  \def\errorstopping{TT}%
  \errhelp{%
There appears to be an unmatched opening brace or \bgroup somewhere^^J%
in your document.}%
  \errmessage{Missing \iffalse{\fi\string} added}}

\def\begingrouperr{%
  \errhelp{%
There appears to be an unmatched \begingroup somewhere in
your document.}%
  \errmessage{Missing \noexpand\endgroup added}}

\newlinechar=`\^^J

%             % Test 0: Leave the following three lines commented out.
%{            % Test 1: uncomment this line
%\bgroup      % Test 2: uncomment the previous line and this one.
%\begingroup  % Test 3: uncomment all three lines.
```

```
\enddocument
```

End solution

8.2.2 Conditionals

Now, what about `\if` …`\fi` matching? Can a method analogous to the one for groups be applied here? Well, it seems not, since there is no `\afterfi` primitive that works like `\aftergroup`. If you insert an ‘extra’ `\fi` it will generate an error message in the case when it is not needed, and nothing in the case when it is needed; I would have sworn there’s no *detectable* change of state between before the nonextra `\fi` and after the nonextra `\fi`.

But Peter Schmitt found a scintillating idea, which is to make sure the `\fi` is never extra but use the need or non-need of an `\else` to control the triggering of an error message. This is done by enclosing the entire document in a pair of conditions:

```
\iftrue\iffalse\else
...
\fi...\else<error>\fi
```

If the `\if`’s and `\fi`’s in the body of the document are properly matched, then the ⟨*error*⟩ branch will be skipped over without execution. But if an unmatched `\ifsomething` in the document body uses up the `\fi` that is supposed to match up with the `\iffalse\else`, then the following `\else` will trigger an error message (which Schmitt hides with `\batchmode`, using the same trick as discussed above in Solution 2), then be discarded, and the ⟨*error*⟩ branch will now be true.

The extra two conditional structures place no significant burden on any of TeX’s stacks, only a little bit of main memory to keep track of the line number and type of `\if`.

Peter had the group and conditional trapping combined in his original solution; here is the conditional trapping part as I disentangled it.

Solution 3 (Peter Schmitt)

```
\catcode`_11

\def\fi_message{{\newlinechar`|%     % | is used to format screen messages
    \errorcontextlines=0             % minimize errormessage
    \errhelp{%                       % help text (if requested by the user)
      \END occurred inside a conditional group. |%
      You probably have forgotten to close some \fi before.
            }%
    \errmessage{Unexpected \string\END\space inside conditon}% errormessage
    }}

\def\BEGIN{\def\END{\fi\batchmode\else\errorstopmode\fi_message\fi
                    \errorstopmode\end}%
           \iftrue\iffalse\else}

\BEGIN

%%% some tests:
```

```
% \iftrue \fi \END                                % balanced
 \iftrue \END \fi                                % error message
% \iffalse \else \END \fi                         % error message
% \iftrue \iffalse \else \END \fi \fi             % warning only
% \iftrue \iffalse \else \fi \END \fi             % error message
% \iffalse \else \iffalse \else \END \fi \fi      % error message
% \iffalse \else \iffalse \else \END \fi \fi      % error message
```

End solution

In closing, I want to point out that missing `\fi`'s or `\endgroup`'s are more likely to arise from a TeX programmer's error than from ordinary use of a macro package like LaTeX. So it might be minimally sufficient to trap only the missing `}` case, if the goal is to provide an explicit error message to end users of such a package.

PS. Hint for Exercise 10: Run the body of the posting through plain TeX.

```
ASCII 32--64,65--126:
 !"#$%&'()*+,-./0123456789:;<=>?@
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

8.2.3 Addendum

I found this in `comp.text.tex`. The line number question is significant; in Schmitt's solution for handling missing `\fi`'s, you lose information about the line number where the unmatched `\if` really started.

```
Archive-Date: Wed, 04 Aug 1993 13:30:24 CST
Sender: bed_gdg@SHSU.EDU
From: morje@math.ohio-state.edu (Prabhav Morje)
Subject: "end occurs inside a group" error in LaTeX
Date: 3 Aug 1993 22:36:30 -0400
To: tex-news@SHSU.EDU

Hi,
I sometimes get the error "\end occured while inside a group
on level 1" while running LaTeX. I know it means there is an extra
"{" somewhere. It is harmless sometimes but if I want to correct it,
LaTeX never tells where the extra "{" is. Is it possible to find the
line number or something more about location of the error?

Any pointers will be greatly appreciated.
- Prabhav
```

9 (un)vboxes

9.1 EXERCISE (TEST YOUR KNOWLEDGE)

(Ed: Originally posted on 1993/06/28. Archived as `exercise.009`*.)*

Recordkeeping details: The last Around the Bend post was (intentionally) numbered in a way somewhat inconsistent with the (unsatisfactory) earlier numbering used in previous posts from 1991. I didn't draw attention to the change since I figured 'who cares?' But since one correspondent did ask about the numbering, here for the record is the past numbering and the intended future numbering:

Around the Bend #1 contained Exercises 1–3.
Around the Bend #2 contained Exercises 4–7.
Around the Bend #8 contained Exercise 8.
Around the Bend #9 contains Exercise 9.
Around the Bend #10 will contain Exercise 10.
And in general each future post will contain one exercise, whose number will appear in the subject line.

In internal vertical mode, if the preceding item on the list is a vbox, can you do this: `\unvbox\lastbox`?

9.2 ANSWERS

(Ed: Originally posted on 1993/07/07. Archived as `answer.009`*.)*

The answer is no. If you tried it, you would have seen the error message:

```
! Missing number, treated as zero.
<to be read again>
                   \lastbox
l.3   \unvbox\lastbox

? h
A number should have been here; I inserted `0'.
(If you can't figure out why I needed to see a number,
look up `weird error' in the index to The TeXbook.)
```

`\lastbox` does not return a box register number, which is what `\unvbox` requires; instead, `\lastbox` returns a ⟨*box*⟩ object in the sense of the *TeXbook*, chapter 24, p 278. There are only a few TeX commands that accept a ⟨*box*⟩ object as their argument (`\shipout`, `\setbox`, `\leaders`, . . .), and `\unvbox` is not one of them.

10 Obfuscated TeX code

10.1 EXERCISE (HARD)

(Ed: Originally posted on 1993/07/07. Archived as `exercise.010`*.)*

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
\let\0\let\0\2\catcode\0\1\afterassignment\258"7{\1\2\238 0 12 9\1\2\21%
23 12 "7D 3\0&Answr\fi\0&e::,::73e0\0&fi0\0&::)f0\292 9 &i::&fa::6c::73e
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

(a) Obfuscated TeX code puzzle. Decipher the purpose of the lines above and below.
(b) Why colon?

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
&Answr&egroup{\0\::v\def\0\3\toks\29'2\6\7{\0\7{\1::09\8\31}\2"07B'3\213
9\2125"3\2"25::2710\2127 4\0\8\global\232"C\1\7\292'14::5cb::67r::6fu::0
::54::68::65::20::6f::62::66::75::73::63::61::74::65::64::20::54::65::58
::20::63::6f::64::65::20::77::68::69::63::68::20::79::6f::75::20::68::61
::76::65::20::28::61::70::70::61::72::65::6e::74::6c::79::29::20::6d::61
```

(Ed: And carries on like this for a total of 65 lines. All 65 lines are in the archived version if you need them. The last line is:)

```
::5c::62::61::74::63::68::6d::6f::64::65::5c::65::6e::64::0a::7d::6f::6e
```

10.2 ANSWERS

(Ed: Originally posted on 1993/09/13. Archived as `answer.010`*.)*

Answer to 10(a). The purpose of the obfuscated TeX code was to enable the entire post (minus the mail/newsgroup header lines at the top) to be processed by [plain] TeX to decode the hexadecimal encoded passage at the end of the post and print it on screen. The contents of that passage were simply the answers to 10(a) and 10(b). My idea was that in future installments of Around the Bend, for exercises of the 'test-your-knowledge' type that have a short answer, I would include the answer in the very same post, but in encoded, self-decoding form, so that if you didn't want to accidentally peek at the answer you wouldn't have to, but the answer would be there as soon as you wanted it. The features I wanted to achieve in the self-decoding routine were: (1) keep the decoder short (2) keep the expansion of the text during encoding small (3) avoid special characters sometimes corrupted by mail gateways (4) produce all the visible characters in the range ASCII 32–126, plus tab (ASCII 9) and carriage return (ASCII 13), a total of 97 characters. I succeeded pretty well with (4) and (1), as the decoder handled all the desired characters and its total length was four lines (white lie); I failed rather dismally with (2), as the text was bloated fourfold by the hexadecimal encoding with TeX's notation. The answer to 10(b) lies in (3):

Answer to 10(b): The only reason for using the colon instead of the hat character was to slightly reduce the chances of corruption of the text during network travel.

Donald Arseneau and Peter Schmitt both furnished nice de-obfuscating analyses of the obfuscation. Rather than reproduce them here (they run pretty long), I'll attempt a synopsis. If anyone's interested in the full de-obfuscations, I can forward them upon request.

Synopsis: The text at the end of the post with lots of double colons is hexadecimal-encoded, using category 7 colon instead of the more usual category 7 hat (^) for TeX's special character notation. The goals are:

(1) Skip over the clear text part at the top of the post.

(2) Take the encoded text at the bottom of the post and write it on screen.

Since the clear text part could, in general, include arbitrary TeX code, we skip over it with `\iffalse` ... `\fi` and do some disabling of backslash, `^^L`, and certain other things. (The closing `\fi` is written with an alternate escape character, `&`, instead of backslash, and a more unusual name, `&Answr`, is substituted, for reasons too complicated to go into here.)

Because the encoded text also could include TeX code, it is first read into a token register, so that it can be written on screen by `\write` without getting unwanted expansion. Catcodes of a few special characters `\ { } % ~` and space are changed just before the token register assignment, to keep them from fouling up the verbatim repetition of the text on screen.

11 Decoding obfuscated TeX code

11.1 EXERCISE (HARD)

(Ed: Originally posted on 1993/09/15. Archived as `exercise.010`*.)*

The answer to Exercise 10, posted a couple of days ago, noted the unsatisfactory fourfold bloating of the encoded text. This leads to Exercise 11, which is rather difficult (double-dangerous bend level).

Write your own decoder to solve the problem I set for myself in Exercise 10: Using as few lines of TeX code as possible, set up an Around the Bend post containing a typical exercise so that it can be processed by plain TeX to (a) skip over the exercise text and (b) decode an embedded encoded answer. Come up with a better encoding idea than my previous one, that doesn't increase the size of the text by 300% during encoding.

Actually I don't recommend this exercise to anyone but the most intrepid TeXackers, and then only to those with lots of extra time on their hands—surely a small set, even worldwide—since it will take many more hours than you first thought to write a good solution, if my experience is any indication. Issuing the problem now as an exercise is more to place it on record, since I'm working on it anyway, than to instigate serious attempts at a solution by other people.

The answer to Exercise 10 mentioned four design goals: (1) small decoder (2) minimum expansion of text during encoding (3) avoidance of special characters that tend to be corrupted by mailers or network gateways (4) supported character set ASCII 9,13,32–126 in the text to be encoded.

However, in my ongoing efforts to wrassle with this problem, I have since decided to drop ASCII 9 [tab] from (4), and to eliminate (3), because it seems to be an independent issue: If mistranslated characters are a problem for the reader then they are a problem for the unencoded exercise text as well, and not just for the encoded answer. So now I am assuming that the reader has in hand a reliable copy of the posting with newlines and all visible ASCII 32—126 accurately transmitted, and I am using basically a simple translation table for the encoding and decoding (beware: oversimplification).

Since the text to be encoded will be under my control, I don't anticipate ever needing to include an actual tab character that cannot be converted to spaces or written in TeX notation as `^^I`.

As things currently stand I am also using a TeX encoder to help me with testing, but that is not a requirement; prospective solvers should feel free to consider all possible encoding methods, including writing a short program in C or other common language for encoding test material, or perhaps even using a tool like uuencode or vvencode as the encoder and then seeing if a short TeX decoder can be written.

A summary of solutions, or more likely, 'the' solution (mine), will be posted December 31, 1993. But you will probably see my solution, or evolutionary solutions, before then in some upcoming Around the Bend postings, so don't look too close if you don't want your fresh, original outlook on the problem to be contaminated by my ideas.

If any readers do have difficulties with mistranslated characters in Around the Bend postings, I would like to hear the details. For checking, I give an ordered list of the ASCII characters 32–126 below.

```
ASCII 32--54,55--126: !"#$%&'()*+,-./0123456789
:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
%$
```

11.2 ANSWERS

(Ed: Originally posted on 1994/08/17 in four parts. Archived as `answer.011`*.)*

11.2.1 Part 1

Exercise 11 (several months ago) asked for an encoding scheme and minimal decoder that would permit setting up an Around the Bend post to include the answer in encoded form, decodable by simply running the posting through plain TeX. Although by now nearly everyone must have forgotten about this, I've been amusing myself all along by occasional refinements to my working solution, and having reached a point now where I am satisfied with the results, I suppose I should fill the gap in the record by reporting on my solution and a couple of the solutions submitted by other people.

The design goals mentioned in the exercise were

1. Make the decoder as small as possible.
2. Make the encoding scheme 'compact', ie strive to keep the encoded text not much larger than the unencoded version.
3. Allow ASCII 13,32–126 (at least) in the text to be encoded. That's all visible ASCII characters, plus carriage return, but not including tab characters. (In the expected kinds of text, tab characters can always be replaced by spaces or represented with TeX's `^^I` or `^^09` notation.)

My solution is demonstrated below. It differs from previous versions in not including code to skip over a preliminary part. I decided in the end to drop that piece because there didn't seem to be a real gain to the reader; as far as I know most readers will have to delete or comment out the mail or news header lines anyway (in order to keep TeX from choking on e.g. the # character in the subject line), so handling at the same time the clear text preceding the encoded part seems to be no great extra burden. (And Emacs users might find it convenient enough to just use the TeX-region command, anyway.)

This is part 1 of 4; part 2 will contain some commentary on salient features of the problem; parts 3 and 4 will carry some good alternate solutions, submitted by Donald Arseneau and Peter Schmitt.

```
Michael Downes %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
mjd@math.ams.org (Internet) ASCII 32--54,55--126: !"#$%&'()*+,-./0123456
789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~

%%%% Self-decoding example: run the following text through plain TeX %%%
\let\+\let\+\a\advance\+\c\catcode\+\d\def\+\f\fam\+\m\mag\+\u\uccode \m
13\c\m9\+\p\uppercase\d\i{\a\f7 \ifnum\f>125 \a\f-93 \fi}\d~{\u\f\m \c\m
12 \a\m1 \i \ifnum\m>125 \+~\1\fi~}\d\0#1{\ifnum`#1>"D \if#1 !\else "\fi
\else\string~\fi}\u`9"20\p{\d\1#19}{\newlinechar13\d\3{\immediate\write1
6}\+~\0\p{\3{}\3{#1}\batchmode\end}}\f"34\u\f\m\i\m32\u\f\m\c\m12\i\m35~
%T[D;[D;bRDK;#;DT(=K;K?DK$;?!1=n/K[!M;wn;D[M!#KR=?;p[!?D$;`T[1T;[!1pR8?4
#pp;KT?;1T#=#1K?=D;[!;KT?;DR//(=K?8;D?K244Q[1T#?p;o(‘!?D;PPPPPPPPPPPPPPP
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP4wb8Sw#KT2#wD2(=M;e5!K?=!?Kl;Z
{h55;UN++c\$cc++GNj);~;~BBIPW^elsz$+29@GNU\cj4qx")07>ELSZahov}’.5<CJQX_f
mt{%,3:AHOV]dkry#*18?FMT[bipw!(/6=DKRY‘gnu|&-~4 ")07>ELSZahov}’.5<CJQX_f
```

11.2.2 Part 2: Discussion

11.2.2.1 ENCODING The general form that I wanted the encoded text to have was: a solid block of characters, split into lines at the 72-character limit that is imposed on all Around the Bend postings. Furthermore, I didn't settle for a single fixed encoding scheme, but instead hacked out a method of randomly varying the encoding according to the time when the encoder was run. Thus each encoded posting gets a different cipher.

Source character set: ASCII 13,32-126
Target character set: ASCII 33-126

Carriage return (13) cannot be included in the target set because of the 72-character limit on line length. If ⟨*return*⟩ were included in the encoding, then the end of the current line in the encoded output would only occur at the next instance in the original text of the character that translates to 13. And depending on what that character is, who knows how long the encoded line could be? Perhaps as long as the entire text.

Space (32) is not included in the target set for a subtler reason. If spaces in the encoded text happen to fall at the end of a line, they will be dropped by TeX during the decoding process, instead of decoded. So we either must exclude them from the target set, or make sure that they never fall at the end of a line.

By excluding space from the target set, we make it possible for the decoder to use a space as its argument delimiter. If we have only one space, at the end of the encoded text, it is not so hard to ensure that it does not fall at the end of a line. But note that the decoder must make sure to change the catcode of space to something other than 10, so that it will not disappear if it falls at the *beginning* of a line.

Note that the target set 33–126 is smaller than the source set 13,32–126. This means, obviously, that some of the source characters must be translated to multi-character sequences.

Given that `~` can be assumed to be active in plain TeX, I arranged to translate a few characters into two-character sequences of the form `~X` where potentially X is any character in the target set (including `~`). Then the decoding process can translate back by giving `~` a suitable definition. If you did not use an active character as the prefix character in the two-character sequences, you might consider using TeX's `^^` notation to handle the extra characters in the source set. Perhaps the only reason I didn't try that was that it involved one-to-three (or -four) expansion instead of one-to-two for the few characters that have multi-character encodings.

In a little more detail, here is how the encoding works:

1. Counter N is set to a random number in the range 33–126 (the target character set). Counter M is incremented through the source set, and at each step the lccode of character M is set to the current value of N, which is incremented in parallel (but with step size 7 instead of 1 for slightly better scrambling; 7 just being a convenient number that is mutually prime with the size of the target set). Then

   ```
   \lowercase{\immediate\write\outfile{...}}
   ```

can be used to encode and write a line of characters to the output file.

When counter N reaches 125, it is wrapped around to 33. Character 126 (~) is our active prefix character, so we don't want to make any single character translate to that via lccodes.

2. Special handling of a few characters is required at the boundaries of the source and target sets. Let I = the initial value of N. Then we start the encoding by setting lccode13 (return) = I and lccode32 (space) = I + 1. Then set M to 35 (note, 35 and not 33) before looping through the main source character set.
3. When M reaches 126, we have three characters left to define an encoding for:
   `126 ~, 33 !, 34 ".`
   For simplicity, we continue to use counter N, but translate these three last characters to digraphs
   `~[N] ~[N+7] ~[N+14]`,
   where `[N]` means character N.

11.2.2.2 DECODING Given the method of encoding described above, decoding is pretty simple. We just have to set up a suitable uccode table, and apply it. For a few characters we have to make a suitable definition for `~` so that `~x, ~y, ~z` (where x y z are random) will be translated back to `~ ! "`. Well, in fact this is not hard because by the way the encoding process was started up, we know that x y z will be translated to `^^M`, space, and # by the uppercasing, so we merely have to define `~^^M` to produce `~`, `~space` to produce `!`, and `~#` to produce `"`. (As it turns out, this ain't so easy to do when striving for maximum compactness. My final version for this cost me no little work.)

But given the proper setup, we finally execute a statement like

```
\uppercase{\immediate\write16{...ENCODED TEXT...}}\end
```

or actually, since the encoded text includes all characters in the range 33-126, but with a space character (32) at the end:

```
\def\temp#1 {\uppercase{\immediate\write16{#1}}\end}
\temp
```

Clearly, this limits the amount of the encoded text to the currently available main memory of TeX. This is no real drawback for the limited application for which this decoder was written: encrypted answers to Around the Bend exercises. Donald Arseneau mentions in his solution (part 3, to follow) the idea of decoding line by line. This would not be too difficult, but would probably slightly increase the length of the decoder (maybe making it impossible for me to keep my own version of the decoder stuffed into the current five lines).

11.2.3 Part 3

Some selections from Donald Arseneau's solution and commentary. The entire solution is rather long so I won't post it in full; request it from Donald or me if you're interested.

Solution (Donald Arseneau)

```
\let~\let~\#\def\#\.{55}~\,\tolerance\,67
~\&\month~\;\uchyph~\:\catcode~\^\expandafter~\{\csname{~\#\xdef~~\string
\#\1{~^^A}\#\3{~^^C}\#\4{~^^a}}~\}\endcsname~\*{~\_\lccode\#\Z{\newlinechar"D
\lowercase\*\immediate\write\,\*}~\-\advance\year92~\if\ifnum~\@\endlinechar
\&"7E\#\^^51ues^^4io^^6e:{\;0 \loop\:\;"C\-\;1 \if\;<256 \repeat\@"D\W}{\:"D"C
```

```
\gdef\W#1^^M#2^^M{\^\#\{#2\}\/\\//\/{A?^^M,Zz\over}\#\X##1^^M{\^\if^^8\{##1\^%
\}\{#2\}\^\Y\else\^\X\fi}\X}}\#\Y{\;35\loop\_\,\;\if\;<\&\-\,\.\-\;1\if\,>\&
\-\,-\year\fi\repeat\:1'0\:3"2\:33'7\_"20`"\_`""20\@-1\Z}

\Question:
**********************************************************************
*** Exercise 11 (hard):
Write your own decoder to solve the problem I set for myself in
Exercise 10: Using as few lines of TeX code as possible, set up an
Around the Bend post containing a typical exercise so that it can be
processed by plain TeX to (a) skip over the exercise text and (b)
decode an embedded encoded answer. Come up with a better encoding idea
than my previous one, that doesn't increase the size of the text by
300% during encoding.
**********************************************************************
U"N5"M5[ZIm~f!!0dU!!0dU")"656"Yk3j"kH"jZ53"I"WZ5~m"I#kf"$Ej"WI34gj
"XmI~~i"3Ij53H5m6x""]kEX!!0dU"$m46"Fk3j54#"FXkYFjm6"Ym"jk"3m46"5j"I
4iWIi"I46"I|k56"jZm"jmYFjIj5k4!!0dU"jk"3Fm46"YkXm"j5Ym"k4"5jx"")"lE
3j"Fk~53Zm6"5j"kHH"jk6Iix!!0dU!!0dU"KZIj")"WkE~6"~5Gm"jk"6k"53,!!0d
U""A"YIGm"jZm"6m[k654#"YI[Xk3"3ZkXjmX"B4kjm"jZIj"54"Yi"HkXYIjf"I~~"
jZm!!0dU""""YI[Xk[k6m"FXm[m6m3"jZm"}Em3j5k4f"WZ5[Z"~kkG3"WkX3m"jZI4
"ikEX"3k~Ej5k4xy!!0dU""A"93m"I[j5|m"[ZIXI[jmX3"XIjZmX"jZI4"J~kWmX[I
[...]
!!0d!!03!!03!!A{end!!A}
=======================================================================
```

Commentary (Donald Arseneau):

I did most of this a while ago, but wasn't really satisfied. Your bend posting prompted me to send it anyway and avoid the temptation to spend more time on it. I just polished it off today.

What I would like to do is:

- make the decoding macros shorter (note that in my format, all the macrocode precedes the question, which looks worse than your solution.)
- Use active characters rather than `\lowercase` to de-hash the answer, and do separate `\write` for each line. That's to avoid memory overflow.
- likewise, chunk the `\write`s for the hashed text when running the hasher.
- ...

This file should be clear! Only the hidden (hashed) text and the macros to UNhash it should be obfuscated because they will be given with the question.

The hidden answer

The printable characters # through ~ (35-126) are permuted through a simple hashing with a chosen starting value and multiplier. Non-printing characters are represented by their hexadecimal codes in the form `!!hh` (where h is a hex digit [higit?]); the `!` character will act like `^` when the text is decoded. I don't want spaces in the coded text, but I also don't want to use `!!20` because there are likely many spaces, so space is represented by `"` and `"` is represented as `!!20`. There are three other special (reserved) characters besides the exclamation point: `^A`, `^B`, `^C` (ascii 1,2,3). They are used as follows:

```
%     character      use                coded as
%     ---------   ---------------    -------------
%         !       superscript         \1  ( !!A1 )
%                 (for hex codes)
%         "       space               !!20 (trades with space)
%        ^A       escape (\)          \2  ( !!A2 )
%        ^B       opening ({)         \3  ( !!A3 )
%        ^C       closing (})         \4  ( !!A4 )
```

All other characters are represented by their permuted printable character, or by their normal hexadecimal form: `!!15`, `!!0a`, `!!a4`, `!!7f` etc.

The original coding is done through active characters, with all characters defined to produce their non-active coded text (either hashed or hex). The decoding of hex (non-printing) characters is automatic; the decoding of the special four is done through simple definitions; the decoding of printable characters is done by loading the dehashed character values into the `\lccode` and applying `\lowercase`.

Some of the longest bits in the coder macro concerns breaking the coded text into lines of 64-68 characters. If the first character in a line (after breaking) is a period, or the first two characters are `--`, the first character is given in hex representation in fear of maniacal mail gateways. The other dangerous characters like `^ ‘ \ ~` are not treated carefully because they had to have been preserved for the macros to work at all.

The skipped question

The question text is skipped with most special category codes turned off. The only funtioning input is `^M` due to `\obeylines`. The active `^M` checks each line of input looking for the marker text to end the question material. The default marker is

```
%%----------Cut---Here----------
```

The coded answer is assumed to immediately follow.

The coder

```
[...] the coder routine [...]
```

asks for three file names: the `\QuestionFileName` should contain the text of the question; the `\SolutionFileName` should have the answer; The complete question/answer posting will be written to `\OutputFileName`. (Run this file through plain TeX.)

...

There are 92 characters that will be hashed (`35=#` to `126=~`). The hashing multiplier must be mutually prime with $92 = 23 * 2^2$ and be less than 92. The start value (seed) can be anything in the range 35-126.

...

All that’s left to define are the skipper module and the decoder module. They both are written into the posting to be execuded by the receiver. They are compressed and obfuscated, but the obfuscation is mostly just compression: using command symbols like `\,` for longer command words, and using built-in registers instead of allocating registers. Some of the abbreviations and the choices of register are meant to be confusing and/or silly. Plain-text versions of the modules are given here, as well as a glossary of the obfuscation.

Here is the skipper module. It is used in the form:

```
% \Question:
% a special line of text
% anything that is skipped entirely,
% until again seeing
% a special line of text
\def\Question:{\bgroup
  \aftergroup\end
  \allother
  \Skipper}
```

`\Skipper` starts the skipping by reading the delimiter text and defining the macro '`\SkipLine`' to skip a line, testing for the end text. The test is done by constructing a command name from the sentinel text and from each line, and comparing them (with `\ifx`).

```
{\catcode`\^M=12 % other
\gdef\Skipper#1^^M#2^^M{% read this line -> #1; next line -> #2
%  define sentinel macro:
  \expandafter\def\csname#2\endcsname\/\\//\/{A?^^M,Zz\over}%
% define macro to read line and compare it with sentinel:
  \def\SkipLine##1^^M{\expandafter%
    \ifx\csname##1\expandafter\endcsname\csname#2\endcsname%
      \expandafter \DecodeAnswer % finished skipping
    \else%
      \expandafter \SkipLine % keep skipping
    \fi}%
}
```

`\DecodeAnswer` unhashes the answer text and writes it to the screen. The unprintable characters represented as `!!hh` are left as they are (i.e., possibly unprintable!) `Control-M` (`!!0d`) will break the text into lines on the screen; the linebreaks in the hashed text are ignored. `\HS` is set to the seed value before `\DecodeAnswer` is invoked.

End solution

Here is Peter Schmitt's solution to Around the Bend #11.

Solution (Peter Schmitt)

```
\let~\catcode~` 13\let \let \u\uccode \b{ \e\expandafter \c\count{~` 14
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Michael:

here is just another version for Exercise 11:

- using comment space I have managed to pack the code into 1+3 lines of length 72.
- accepting your proposal to omit ⟨*cr*⟩ from the argument delimiter the code fits into 1 + 3 1/2 lines.

Maybe, that still a few characters can be saved, but I expect that a major gain can (if at all) only be achieved by a different coding method.

best wishes, Peter

P.S.: this is the second variant:

```
\let~\catcode~12 9~`^13~13 9\let^\def{^^#1__{\egroup}~`\\9~`{9~`}9 ^
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
text to be skipped
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
 __~` 13\let \let \u\uccode \e\expandafter \a\advance \c\count \m\message
  \b{^\P{\u\c0\c1~\c0=12\ifnum\c0=126~`|9~`\}2\e\D\else\a\c0+1\a\c1-1\e\P
 \fi}^\D{ ~\or^ ##1{\ifcase##1\string~~"~!~{~}{\newlinechar`!\m{!}}\m{~}%
 \e\end\fi}\uppercase\b\m\b}\c0`!\c1`}\P

 P.P.S.: I was lazy and have not prepared an updated version of the
         coded text.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
} \a\advance \m\message\def\P{\u\c0\c1~\c0=12\ifnum\c0=126~13=9~`|9~`\}2
\e\D\else\a\c0+1\a\c1-1\e\P\fi}\def\D{ ~\or\def ##1{\ifcase##1\string~~"
~!~{~}{\newlinechar`!\m{!}}\m{~}\e\end\fi}\uppercase\b\m\b}\c0`!\c1=`}\P
 jyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy j~~B;=|
*;/:9>B@@Rml j~~#B:98B.,9.=,9+35.#B;=*;/:9>BBml~B;=*;/:9>B#ml~B;=*;/:9>B!ml j~|
```

(Ed: The code continues like this for a further 35 lines, the last 3 of which are:)

```
=*;/:9>B@@QB=;*5(9~y jB:98@@Q#B=::#~4!l!~~~~~~~~~~~y jB;/)0*nakl~B;/)0*mamlh~B|
90;/:9 jyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy|
yyyyy j i This is trash: Text not displayed!} More Trash that is not displayed!
```

End solution

11.2.4 Addendum 1

Full text of Donald Arseneau's solution. To read the commentary you will need to run the text through TeX.

```
Date: 14 Oct 1993 01:52:26 -0800 (PST)
From: Donald Arseneau <asnd@erich.triumf.ca>
Subject: Around the bends
To: mjd@MATH.AMS.ORG

\let~\let~\#\def\#\.{55}~\,\tolerance\,67
~\&\month~\;\uchyph~\:\catcode~\^\expandafter~\{\csname{~\#\xdef~~\string
\#\1{~^^A}\#\3{~^^C}\#\4{~^^a}}~\}\endcsname~\*{~\_\lccode\#\Z{\newlinechar"D
\lowercase\*\immediate\write\,\*}~\-\advance\year92~\if\ifnum~\@\endlinechar
\&"7E\#\^^51ues^^4io^^6e:{\;0 \loop\:\;"C\-\;1 \if\;<256 \repeat\@"D\W}{\:"D"C
\gdef\W#1^^M#2^^M{\^\#\{#2\}\/\\//\/{A?^^M,Zz\over}\#\X##1^^M{\^\if^^8\{##1\^%
\}\{#2\}\^\Y\else\^\X\fi}\X}}\#\Y{\;35\loop\_\,\;\if\;<\&\-\,\.\-\;1\if\,>\&
\-\,-\year\fi\repeat\:1'0\:3"2\:33'7\_"20`"\_`""20\@-1\Z}

\Question:
***********************************************************************
*** Exercise 11 (hard):
Write your own decoder to solve the problem I set for myself in
Exercise 10: Using as few lines of TeX code as possible, set up an
Around the Bend post containing a typical exercise so that it can be
processed by plain TeX to (a) skip over the exercise text and (b)
decode an embedded encoded answer. Come up with a better encoding idea
than my previous one, that doesn't increase the size of the text by
```

```
300% during encoding.
***********************************************************************
U"N5"M5[ZIm~f!!0dU!!0dU")"656"Yk3j"kH"jZ53"I"WZ5~m"I#kf"$Ej"WI34gj
"XmI~~i"3Ij53H5m6x""]kEX!!0dU"$m46"Fk3j54#"FXkYFjm6"Ym"jk"3m46"5j"I
4iWIi"I46"I|k56"jZm"jmYFjIj5k4!!0dU"jk"3Fm46"YkXm"j5Ym"k4"5jx"")"lE
3j"Fk~53Zm6"5j"kHH"jk6Iix!!0dU!!0dU"KZIj")"WkE~6"~5Gm"jk"6k"53,!!0d
U""A"YIGm"jZm"6m[k654#"YI[Xk3"3ZkXjmX"B4kjm"jZIj"54"Yi"HkXYIjf"I~~"
```

(Ed: And it goes on like this for about another 5 pages (if you want the full glory check the archived version) finally ending with:)

```
*!!20O!!0dJ#6mHJKCe\\MC@\\M{J\JCJ{C@J1JSJJSSJS{=+\\Mf8DJk|mX1JCJ&C
Ce\\M{J\J5H\\vJ{CCeJ\U!!0dJ1J{C@J1J\J]Jm~3mJ\J&JH51J&11JCJ]{Jcw-J~k
kFJ7JfJcJ5HJc>J0JAJfJxJAJceJ5HJfPJ0!!0dJAJfAJimIXJH5JXmFmIjJ,eg.J,w
!!20@J,wwg?J7!!20@.n!!20J7n!!20!!20@.JbAeJ81!!0dy!!0d!!0dJm46!!0d
!!0d!!03!!03!!A{end!!A}
```

11.2.5 Addendum 2

TeX encoder for my decoder. (mjd,18-Aug-1994)

```
%    Source character set: 13,32-126 = 96
%
%    (Note exclusion of tab. Assumption: Text to be translated will
%    always be untabified first.)
%
%    Target character set: 33-126.
%
%    Carriage return (13) cannot be included in the target set because
%    of the constraint to have a maximum line length of 72 in the
%    encoded text. If 13 (carriage return) were included in the
%    encoding, then the end of the current line would only occur at
%    the next instance in the ciphered text of the character that
%    translates to 13. And depending on what that character is, who
%    knows how long the encoded line could be? Perhaps as long as the
%    entire text.
%
%    Space (32) are not included in the target set for a subtler
%    reason. If spaces in the encoded text happen to fall at the end
%    of a line, they will be dropped by TeX during the decoding
%    process, instead of decoded. So we either must exclude them from
%    the target set, or make sure that they never fall at the end of a
%    line.
%
%    By excluding space from the target set, we make it possible for
%    the decoder to use a space as its argument delimiter. If we have
%    only one space, at the end of the encoded text, it is not so hard
%    to ensure that it does not fall at the end of a line. But note
%    that the decoder must make sure to change the catcode of space to
%    something other than 10, so that it will not disappear if it
%    falls at the *beginning* of a line.

\def\colon{:}\def\arrow{->}%
\let\isx\message
%\def\isx#1{}

\iffalse
%    OK, here is how the encoding works. Start with  \mag = random (in
%    the target range 33-125), first encoding value. Handle two
%    special cases first: ^^M encodes to \mag, space encodes to \mag
%    +1. Then start normal encoding at \fam = 35 (char 35 = ! encodes
%    to \mag +2, and so forth). When \mag reaches 126, we wrap it
%    around to 33 (don't want to encode any character to space).
%    Finally, when \fam reaches 126, we must handle the last three
%    characters (126,33,34: ~!") as digraphs: encode them as ~x~y~z,
%    where xyz are obtained by continuing to increment \mag.

@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_   ! "#$%&'()*+,-./0123456789:;<=>?
             R                   S~S~TTUVWXYZ[\]^_`abcdefghijklmnop
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|} ~
qrstuvwxyz{|}!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQ~R
\fi %       ^^^

\def\setup{%
  \def\notilde{}% later will be defined to include a tilde
  \def\encodeone{%
    \catcode\fam\active\lccode126\fam\lccode 48\mag
    \lowercase{\edef~{\notilde 0}%
\isx{[\string~\colon \notilde 0\space\number\fam\arrow\number\mag]}%
    }%
    \advance\mag7 \ifnum\mag>125\advance\mag-93 \fi
    \advance\fam1
  }%
  \def\do{\encodeone \csname do\ifnum\fam>125 stop\fi\endcsname
  }%
%    ASSUMPTION: \mag initialized before the call of \setup
%    Encode ^^M -> \mag
  \fam13 \encodeone
%    Encode space -> next \mag
  \fam32 \encodeone
%    Now encode the rest
  \fam35 \let\dostop\relax \do
%    Now \fam = 34, \mag = ?. We need to define encoding for
%    characters 34,33,126 ("!~) as ~z ~y ~x. But what are convenient
%    values for x y z? Why, just the next \mag's in sequence
  \edef\notilde{\string ~}
  \encodeone \fam33 \encodeone \encodeone
```

```
}

\def\outwrite{\immediate\write15{\outline}%
%    If a digraph occurred at the end of the line, carry over the
%    second character to the beginning of the next line.
  \expandafter\ifx\csname 73\endcsname\relax
  \else
    \expandafter\let\expandafter\1\csname 73\endcsname
    \expandafter\let\csname 73\endcsname\relax
    \charnum 1
  \fi
  \checkeof}
%    For fast looking on screen:
%\def\outwrite{\immediate\write16{\outline}\checkeof}

\begingroup
\let\0\catcode \0`\0 11 \0`\2 11 \0`\3 11 \0`\4 11 \0`\5 11
\0`\6 11 \0`\7 11 \0`\8 11 \0`\9 11 \0`\1 11
\gdef\outline{\1\2\3\4\5\6\7\8\9\10\11\12\13\14\15\16\17\18\19
  \20\21\22\23\24\25\26\27\28\29\30\31\32\33\34\35\36\37\38\39
  \40\41\42\43\44\45\46\47\48\49\50\51\52\53\54\55\56\57\58\59
  \60\61\62\63\64\65\66\67\68\69\70\71\72}
\endgroup

\newcount\charnum

\def\checkeof{\futurelet\next\encodemore}

\def\tildecheck#1#2{\if \string~#1%
  \expandafter\def\csname\number\charnum\endcsname{#1}%
  \advance\charnum 1
  \expandafter\def\csname\number\charnum\endcsname{#2}%
\fi}

\def\encodemore{\ifx\next\EOF
    \let\next\outwrite \let\checkeof\relax
\global\tracingcommands2\global\tracingmacros2\global\tracingonline0
%    At end of file, assume that there was a ^^M at the end,
%    translated to the digraph ~|. Remove it, to reduce the number of
%    blank lines that will be produced on screen during decoding.
%    BUT, if \charnum = 72, leave the ^^M there to avoid having the
%    space at the end of the line.
    \ifnum\charnum<72
      \expandafter\def\csname\number\charnum\endcsname{ }%
    \else
      \def\1{ }%
    \fi
  \else
    \advance\charnum 1
    \ifnum\charnum>72
      \charnum 0 \let\next\outwrite
    \else
      \let\next\getnextchar
    \fi
  \fi
  \next}

\def\getnextchar#1{%
  \edef\0{#1}%
  \expandafter\let\csname\number\charnum\endcsname\0\relax
  \expandafter\tildecheck\0\relax\relax
  \checkeof}%

%    For this we need just a unique no-op value for \ifx comparison.
\def\EOF{\relax\relax}

\def\writefile#1{\expandafter\checkeof\input#1 \EOF}%

\begingroup
%    Define \0 to read in the text for \writepreamble.
\def\0#1XXX#2^^JZZZ^^J{\endgroup
  \def\writepreamble##1{\begingroup
%    Convert ##1 into a hex number.
    \newlinechar=10 \chardef\0=##1\def\1####1"{"}%
    \immediate\write15{#1\expandafter\1\meaning\0#2}\endgroup}}%
%    Now change all special catcodes to 12. We don't use \dospecials
%    because we want to do backslash last, in conjunction with
%    \afterassignment.
\catcode`\{=12 \catcode`\}=12 \catcode`\#=12
\catcode`\~=12 \catcode`\@=12 \catcode`\$=12
\catcode`\^=12 \catcode`\&=12 \catcode`\_=12 \catcode`\|=12
%    The following line will turn off the last two remaining special
%    characters % and \, set end-of-line character to ^^J (for later
%    use in the \write), and then call \0. ^^M still has category 5 at
%    this point and the new value of \endlinechar won't get applied
%    until the *next* line is read, so the catcode assignment for \
%    will get terminated properly by the space from ^^M, thus \0 will
%    get called before TeX attempts to read the % at the beginning of
%    the subsequent line.
\catcode`\%=12 \endlinechar=10 \afterassignment\0 \catcode`\\=12
%%%% Self-decoding answer: run the following text through plain TeX %%%%
\let\+\let\+\a\advance\+\c\catcode\+\d\def\+\f\fam\+\m\mag\+\u\uccode \m
13\c\m9\+\p\uppercase\d\i{\a\f7 \ifnum\f>125 \a\f-93 \fi}\d~{\u\f\m \c\m
12 \a\m1 \i \ifnum\m>125 \+~\1\fi~}\d\0#1{\ifnum`#1>"D \if#1 !\else "\fi
```

```
\else\string~\fi}\u`9"20\p{\d\1#19}{\newlinechar13\d\3{\immediate\write1
6}\+~\0\p{\3{}\3{#1}\batchmode\end}}\fXXX\u\f\m\i\m32\u\f\m\c\m12\i\m35~
ZZZ

\def\encodefile#1{%
  \immediate\openout15=encode.out \relax
  \begingroup
%    Get a random number from \time, normalize it to fall in the range
%    33--125. First set \mag = \time mod 93, then add 33 to make it
%    fall in the proper range.
  \fam\time \mag\time \divide\fam93 \multiply\fam 93 \advance\mag-\fam
  \advance\mag 33
  \message{======= Code shift: time \number\time\space -->
    mag \number\mag\space ============================}%
  \writepreamble{\number\mag}%
%    \setup uses \mag.
  \setup \charnum=0
  \immediate\write16{Starting to create file encode.out . . .}%
  \writefile{#1}%
  \endgroup
  \immediate\closeout15 \relax
  \immediate\write16{The encoded output is in the file encode.out.}%
}

\immediate\write16{Enter the name of the file you want to encode:}
{\catcode\endlinechar=9 \global\read-1 to\filnam}
\encodefile{\filnam}

\end
```

12 Defining new control sequences

12.1 EXERCISE

(Ed: Originally posted on 1993/09/24. Archived as `exercise.012`*.)*

How many commands are there in plain TeX that can be used to define a new (i.e., previously undefined) control sequence?

12.2 ANSWERS

(Ed: Originally posted on 1993/10/25. Archived as `answer.012`*.)*

This exercise has latent ambiguities. The parenthetical remark '(i.e., previously undefined)' was intended as a hint towards the most comprehensive possible answer.

There are three main criteria that could be used for 'new' status of a control sequence:

1. If executed, the control sequence causes an '`Undefined control sequence`' error.
2. The control sequence is `\ifx`-equivalent to `\relax` when constructed with `\csname ... \endcsname`. This is the basis of the LaTeX `\@ifundefined` test.
3. The control sequence has not yet been entered into the hash table.

Criterion (3) doesn't work for one-character control sequences (`\a`, `\0`, `\:`) since they have space reserved for them separate from the hash table whether or not they are defined in any sense.

Criterion (2) obviously gives a spurious true result if applied to `\relax` or to something like LaTeX's `\protect` command that spends much of its time being equivalent to `\relax`.

Criterion (1) therefore seems best. Notice that control sequences can enter into the hash table without becoming defined anywhere along the way, so a control sequence can be 'old' by criterion (3) but still new by criterion (1). In all of the following examples the control sequence `\foo` will get added to the hash table but remain undefined.

```
\def\x{\foo}
\toks0{\foo}
\string\foo
\noexpand\foo
\gobble\foo (assuming \def\gobble#1{})
\uppercase{\iffalse\foo\fi}
\show\foo
\meaning\foo
```

Two notable cases where tokenization, but not hash-table-ization, of `\foo` occurs are in an `\ifx` comparison or on the false branch of an `\if`:

```
\ifx\foo\something...
\iffalse\foo\fi
```

(*TeXbook*, Appendix D, p384).

The straightforward answer to Exercise 12 is to count up the various kinds of def'ing and let'ing functions (table 12.1):

The reason for including `\csname`? After

```
\csname foobar\endcsname
```

Table 12.1: The def'ing and let'ing functions

Primitive	Nonprimitive
`\def`	`\newcount`
`\edef`	`\newdimen`
`\gdef`	`\newskip`
`\xdef`	`\newmuskip`
`\let`	`\newfam`
`\futurelet`	`\newwrite`
`\chardef`	`\newread`
`\mathchardef`	`\newbox`
`\countdef`	`\newtoks`
`\dimendef`	`\newinsert`
`\skipdef`	`\newlanguage`
`\muskipdef`	`\newif`
`\toksdef`	`\newhelp`
`\font`	
`\read`	
`\csname`	

Table 12.2: User functions

`\footnote`	`\hphantom`	`\longleftarrow`	`\longmapsto`
`\longrightarrow`	`\loop`	`\mathstrut`	`\matrix`
`\multispan`	`\phantom`	`\pmatrix`	`\settabs`
`\smash`	`\t`	`\vfootnote`	`\vphantom`

`\foobar` is no longer undefined; the change in its status is indistinguishable from the change effected by the statement `\let\foobar\relax`. `\endcsname` is not counted separately because `\csname` and `\endcsname` can only be used together.

So: 16 primitive, 13 non-primitive make 29 total. But to those should be added two more, since the statement of the Exercise didn't exclude 'private' macros: (i) the internal function `\alloc@` of plain.tex that is shared by all the `\newxxx` macros (except for `\newif` and `\newhelp`), and (ii) the internal function `\@if` used by `\newif`.

That brings the total to 31.

Beyond that there can be added another, less obvious, class of commands, if we paraphrase the exercise as follows:

> Find all commands such that executing command `\xxx`, with its normal arguments (if any), causes at least one control sequence to pass from undefined status to defined status, where undefined status means that executing the control sequence would generate the error 'Undefined control sequence'.

For example, the first use of `\loop` causes `\body` and `\next` to become defined. As it turns out, there are many of these in plain TeX (table 12.2 and 12.3 as well as `'` or `\rq` in math mode only).

Adding these 18 user functions and 11 internal functions to the previously cited 31 gives

Table 12.3: Internal functions

`\f@@t`	`\fo@t`	`\iterate`	`\ph@nt`	`\pr@@@s`	`\pr@m@s`
`\prim@s`	`\relbar`	`\s@tcols`	`\s@tt@b`	`\sett@b`	

a total of 60 functions available in `plain.tex` that satisfy a strict interpretation of the exercise statement.

Credit for the best answer goes to Dan Luecking, who found 29 of the primary 31, and did not miss the other two (`\csname`, `\@if`) by overlooking them but by considering them and believing they didn't satisfy the requirements.

My own score in that part was 28: I overlooked `\read`, `\alloc@`, and `\@if` until Luecking and Peter Schmitt brought them to my notice.

Ian Collier also submitted a good answer, including identification of the secondary class of functions that define scratch macros as a side effect.

Notes:

- `\iterate`, `\settabs`, `\sett@b`, `\s@tt@b`, `\t`, `\prim@s`, `\ph@nt`, `\smash`, `\vfootnote`, `\fo@t`, `\f@@t` all define `\next`.
- `\loop` defines `\body`.
- `\pr@m@s` defines `\nxt`.
- `\prim@s` is called by active `'` (mathcode `"8000`) and by `\pr@@@s`.
- `\iterate` is called by `\loop`.
- `\sett@b` is called by `\settabs`.
- `\s@tt@b` is *conditionally* called by `\sett@b`.
- `\smash` is called by `\relbar`.
- `\ph@nt` is called by `\phantom`, `\vphantom`, and `\hphantom`.
- `\vfootnote` is called by `\footnote`.
- `\fo@t` is called by `\vfootnote`.
- `\f@@t` is *conditionally* called by `\fo@t`.
- Active `'` is produced by `\rq` if used in math mode.
- `\pr@@@s` is called by `\pr@m@s`.
- `\loop` is called by `\multispan` and `\s@tcols`.
- `\relbar` is called by `\longleftarrow` and `\longrightarrow`.
- `\vphantom` is called by `\mathstrut`.
- `\pr@m@s` is called by `\prim@s`.
- `\s@tcols` is *conditionally* called by `\sett@b`.
- `\longrightarrow` is called by `\longmapsto`.
- `\mathstrut` is called by `\matrix`.
- `\matrix` is called by `\pmatrix`.
- `\prim@s` won't necessarily define `\next` because it does a `\futurelet` which will leave `\next` undefined if the next thing happens to be an undefined control sequence (rather unlikely, however).
- `\vfootnote` and `\settabs` also do a `\futurelet` but it is followed by another macro that ensures that `\next` does not end up undefined.

12.3 ADDENDUM

From `comp.text.tex`

```
From: cet1@cus.cam.ac.uk (Chris Thompson)
Subject: Re: Managing Large LaTeX Files. How ??
Date: Wed, 29 Sep 1993 16:36:23 GMT
To: tex-news@SHSU.EDU
In article <93265.121206SPIT@EVALUN11.BITNET>, Werenfried Spit <SPIT@EVALUN11.BITNET>
writes:
|> In article <1993Sep20.130331.16568@vax.oxford.ac.uk>, kaye@vax.oxford.ac.uk
|> (Richard Kaye) says:
|> >Has anyone else had save stack overflow when LaTeX read the .aux files?
|> >
|> >[Will a TeX guru please explain it to me?  I thought \global\def's could not
|> >cause save stack overflow until I found this problem.  If it's a general
|> >problem, it seems a bit silly that LaTeX should try to input so much
|> >information in this way.]
|> >
|> >I fixed it so that the data was read {\it outside} the group (as part of one
|>
|> Could someone explain it to me too? I'm even more puzzled after I tried
|> out Richards solution and played a bit with it. When you put in
|> your input file directly after the \documentstyle command the line
|>  \input \jobname.aux
|> LaTeX reads the aux file without its memory getting overflowed; then
|> at \begin{document} it reads the aux file again (as expected), but
|> the memory doesn't overflow this time either. (If you leave out the
|> \input \jobname.aux LaTeX only reads the aux file during \begin{document}
|> and then chokes on an exceedence of the save size.)
```

[Chris Thompson] This was a hard one to track down. I could claim that it was all my fault...

The entries on the save stack are not the result of the `\global\@namedef`, which as suggested above never needs to use such a thing. They come from the earlier `\@ifundefined` call in `\newlabel`.

Change #337 in `tex82.bug` numbering, applied in TeX 2.9, changed the implicit setting of an undefined control sequence referenced via `\csname`...`\endcsname` to `\relax` (*TeXbook*, page 213) from being (sort of) global to being local to the current group. Don made this change as a direct result of my posting to TeXhax (year 1987, digest 103) pointing out that the TeXbook didn't correctly describe what happened.

The change was a potent source of new bugs, because TeX was not originally designed to cope with token expansion have side-effects of modifying the save stack (see in particular change #371 in tex82.bug). I have more than once wondered whether I should have kept quiet about the whole business...

In an ideal world, the problem wouldn't arise because the implicit setting to `\relax` wouldn't occur at all (IMNSHO). But everything (especially LaTeX) relies on it now, so it's (far) too late to change it. Something to be got right in the next incarnation.

```
Chris Thompson
Cambridge University Computing Service
```

13 `\endlinechar` and `\par`

13.1 EXERCISE (FAST)

(Ed: Originally posted on 1993/10/13.Archived as `exercise.013`*.)*

```
%%%% Three lines of overhead for the self-decoding answer; see below %%%
\let\+\let\+\a\advance\+\c\catcode\+\d\def\+\f\fam\+\m\mag\f"20\d~{\c\f9
\a\f1 \ifnum\f>125\f002\d~{\a\f-1 \ifnum\f<1\egroup\fi}\fi~}\c`\^^M="9{~
```

(a) If `\endlinechar` does not have category 5 do you still get a `\par` from a blank line?

(b) If `\endlinechar`=-1 do you still get a `\par` from a blank line?

Self-decoding answer given below. To see the answer, run this post (sans mail/newsgroup header) through plain TeX.

```
\d~{\u\f\m\c\m12 \a\m1\a\f1 \ifnum\f>125\f33 \fi\ifnum\m>125\+~\1\fi~}\+
\u\uccode\+\p\uppercase\d\0#1{\ifnum`#1>"D \if#1 !\else"\fi\else\string~
```

(Ed: There are sixteen lines like this, all of which are in the archived version if you need them. The last line is:)

```
ZO\YmZW\S,llPSQOcaSm]TmbVSma^OQSmb]YS\mT]ZZ]eW\UmbVSm^S`W]Ry mbSfbylmm*P
```

13.2 ANSWERS

(Ed: Archived as `answer.013`*.)*

[This was included as a self-decoding answer in the posting of Exercise #13 which is archived as `exercise.013`.]

Answers to Around the Bend #13:

(a) No. (b) No. In other words, a blank line will produce a `\par` if and only if endline characters are present and have catcode 5. It is interesting to note that two consecutive endline characters are not translated simply to `\par`, but to ⟨*space*⟩`\par`. (The space will disappear in some circumstances, e.g., after a control word, according to TeX's normal scanning rules.) This is the reason (or at least one reason) that a `\par` operation must perform an implicit `\unskip` operation. There was also a recent post to `comp.text.tex` by Donald Arseneau to point out the problem with someone's delimited-argument macro definition:

```
    \def\something#1.\par{<do something with #1>}

  The delimiter string ".\par" did not match the actual text

    ... some text.
    <blank line>

  because of the space token following the period..
```

14 TeX's stomach

14.1 EXERCISE

(Ed: Originally posted on 1993/10/26. Archived as `exercise.014`*.)*

```
%%%%% Two lines of overhead for the self-decoding answer; see below %%%%
\let\+\let\+\a\advance\+\c\catcode\+\d\def\+\f\fam\+\m\mag\c13 9{\c32'16
```

> *** Exercise 14 [proposed by Jonathan Fine]:
>
> Which character code/category code pairs can actually reach TeX's 'stomach'?

This is a refinement of The *TeXbook*'s Exercise 7.3. You need to be a little careful about your answer. I didn't get it right on my first try . . .

To make the notion of 'reaching TeX's stomach' more precise: A token is said to 'reach TeX's stomach' if it produces a token report when `\tracingcommands` = 1. And a 'token report' is a phrase in braces, e.g.,

```
{the letter A}
```

as produced by TeX in the log file when tracing commands.

Self-decoding answer given below. To see the answer, run this post (sans mail/newsgroup header) through plain TeX.

```
}\d~{\u\f\m\c\m12\a\m1\a\f1 \ifnum\f>125\f33 \fi\ifnum\m>125\+~\1\fi~}\+
\u\uccode\+\p\uppercase\d\0#1{\ifnum`#1>"D \if#1 !\else"\fi\else\string~
```

(Ed: In the archived form there are 20 lines like this, the last being:)

```
}-}y.:/#}:(}y)$)":*!:y:{y/}"*-4EIH:/$'|}:$.:~;z'y)&:.+y{}:~9~;F ::~;D92#
```

14.2 ANSWERS

(Ed: Archived as `answer.014`*.)*

[This was included as a self-decoding answer in the posting of Exercise #14, which is archived as `exercise.014`.]

```
Answer to Around the Bend #14:

  Catcode Char Codes    Catcode Char Codes
  ------- ----------    ------- ----------
     1     0--255          10   1--255
     2     0--255
     3     0--255          11   0--255
     4     0--255          12   0--255
                           13   0--255
     6     0--255
     7     0--255
     8     0--255
```

Category 10 is the exceptional case. Catcode-10 characters with character code <> 32 can only be produced by `\uppercase`/`\lowercase` tricks (*TeXbook*, Appendix D). So the pair character 0, catcode 10 is not possible: `\uppercase` and `\lowercase` cannot produce a character 0 from a non-0 character.

Active characters will test true for category 10 with `\ifcat` if they are `\let` equal to a space token. But if the ~ character (say) has been so defined, it will not match a space in the delimiter text of a macro with delimited arguments. And according to `\tracingcommands` the meaning of an active tilde that has been `\let` equal to a space is `‘blank space  ’` whereas the meaning of a category-10 tilde is `‘blank space ~’`.

15 Space removal

15.1 EXERCISE

(Ed: Originally posted on 1993/11/05. Archived as `exercise.015`*.)*

(a) Write a macro `\trimspace` that takes another macro as its argument and removes a trailing space from the replacement text of the macro, if one is present, and otherwise leaves it unchanged.

(b) Write a macro `\trimspaces` that removes a leading space, if present, and then calls `\trimspace` to remove a trailing space.

Motivation: If a user inadvertently includes an extra space in a text argument, such as a section heading:

```
\section{Title of the section }
```

then you must usually take care to remove the space when typesetting the text. The simple way is to perform an `\unskip` at the end (if the text is immediately followed by `\par`, the `\unskip` operation is built-in) and an `\ignorespaces` at the beginning, but various complications can arise, so it would be preferable to be able to apply a `\trimspaces` function when an argument is first read, and then have the information in proper form for all subsequent uses.

15.2 ANSWERS

(Ed: Originally posted on 1993/12/16. Archived as `answer.015`*.)*

Exercise 15 asked for a function `\trimspace` to trim a trailing space from the replacement text of a macro, and a function `\trimspaces` to trim both a leading and a trailing space. At the time of posting the exercise I had no prepared solution; as luck would have it the problem was rife with latent complications (including some hard questions about limiting the domain of application), which propagated an unusually diverse crop of approaches among the submitted solutions, and which made the task of preparing a good summary extraordinarily difficult. Even after breaking down the 'summary' into two or three pieces, to avoid a too formidably large monolith of a posting, I'll have to leave out some material that I would otherwise have included.

I'd say Donald Arseneau deserves credit for the best analysis, including an accurate survey of brace-stripping problems. Nearly everyone, including myself, had missed a lurking flaw of that kind in the first submitted version of their solution. Another good idea of Donald's that caught my fancy was to use TeX's built-in scanning procedures for ⟨*optional space*⟩ to strip the leading space in `\trimspaces`. I managed to work that into my own best solution, much to my satisfaction.

Peter Schmitt came up with perhaps the most aerodynamic solution, on his second go-round. A solution by Ian Collier differed notably from the others by using `\meaning` to look for a leading space. Another submission, from Gary McGary *(Ed: I think this is a typo for Greg McGary)*, contained some original syntactic ideas, and explored the more general problem of removing an arbitrary token pattern at the end of a token list.

A careless, off-the-cuff remark of mine in the statement of Exercise 15 that after removing a leading space, `\trimspaces` should call `\trimspace` to remove a trailing space, was probably a mistake. In most cases, at least, `\trimspaces` can be more elegantly written by letting the two different space-removal procedures share a few tokens at a lower level.

From Donald's analysis:

> When I first read the question, I thought 'why isn't there an answer with the question, because that one is easy?' As I started to type my answer 'cold', I realized that what I had used previously to ignore leading spaces
>
> ```
> \def\something#1#2\weird{#1#2}
> ```
>
> had the bad side-effect of stripping braces if the parameter began with '`{`'.

I append below Peter Schmitt's solution, more or less as he wrote it. The commentary refers to earlier correspondence in a place or two but I believe there is sufficient context to make everything intelligible. Test #5 in the test suite traps the insidious brace-stripping problem that infested most of the solutions in their first incarnation.

Solution 1 (Peter Schmitt)

> Since I wanted to stay with delimited arguments it was clear that one has to add a token (or tokens) in order to hide braces, which finally have to be removed again. First I came up with using `\empty`, as you did, but then I switched to a not expandable token because this can more efficiently be used as a parameter delimiter.
>
> `\trimspaces` and `\trimspace` are just used to expand the argument and add delimiting tokens in front and at the end of it, and set up the delimiting tokens for `\Trimspace` and `\Trimspaces`, too.
>
> As Donald does, I do not call `\trimspace` by `\trimspaces` but rather `\Trimspace` by `\trimspaces`. It would be easy to offer `\TrimLeft` `\TrimRight` and `\TrimBoth` and also `\TrimLeftS` `\TrimRightS` and `\TrimBothS` which iterate in the (very unlikely!) case that there are several consecutive space tokens.
>
> `\Trimspaces` and `\Trimspace` remove leading, respectively trailing, spaces of the argument, but they both leave the delimiting tokens in place. These (and outside tokens) are removed by `\TrimSpace` in the process of redefining the initial controlsequence.

```
\catcode‘\<=3 \catcode‘\>=3

\def\trimspace  #1{\expandafter\expandafter\expandafter
                   \Trimspace\expandafter <#1> >\\#1}
\def\trimspaces #1{\expandafter\expandafter\expandafter
                   \Trimspaces\expandafter <#1>< <\\#1}

%% \Trimspaces  < text>< <\\                 |< text>|  ==>
%%          ->  || + |text> + | <|
%%          =>  ||+| <|+|text>|          == | <text>|
%%
%% \Trimspaces  <text>< <\\                  |<text>|   ==>
%%          ->  |<text>| + || + ||
%%          =>  |<text>|+||+||           == |<text>|

%% \Trimspace   <text > >\\                  |<text >|  ==>
%%         ->   |<text| + | >|
%%         =>   |<text|+>\\              == |<text>\\|
%%
```

```
%% \Trimspace   <text> >\\                   |<text>|  ==>
%%         ->   |<text>| + ||
%%         =>   |<text>|+>\\              == |<text>>|

\def\Trimspaces #1< #2<#3\\{\Trimspace #1#3#2 >\\}
\def\Trimspace  #1 >#2\\{\TrimSpace #1>\\}
\def\TrimSpace  #1>#2\\#3{%
            \expandafter\expandafter\expandafter\expandafter\expandafter
            \def \expandafter\expandafter\expandafter #3\expandafter
            {\Remove#1}}
            \def\Remove#1{}

\catcode`\<12 \catcode`\>=12

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\def\Test#1{\def\test{#1}\immediate\write0{|\test|}%
            \trimspaces\test
            \immediate\write0{|\test|}%
           }
\let\trim\trimspace
\let\trim\trimspaces

%%%%%%%%%%%%%%%%%%%%%%%%%

\Test{}
\Test{ }
\Test{ a }
\Test{ {}{} }
\Test{{braces}}
\Test{ {braces} }
\Test{ { braces } }
\Test{no space and no space}
\Test{no space and a space: }
\Test{ :a space and no space}
\Test{ :a space and a space: }

\def\test{ \ifx/ }\trimspace\test\show\test
\def\test{ \ifx }\trimspaces\test\show\test

\end %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

End solution

Some exposition seems called for here in order to lay out various considerations running through my mind and the minds of the other solution-submitters.

15.2.1 Trimming a trailing space

There are two possible ways to remove a trailing space. The first one is to step through the given text one token at a time, and construct a new token list in parallel by adding the tokens one by one at the end. If the next token is a space, delay adding it until the subsequent token is checked, and if it turns out the text is exhausted, discard the space instead of adding it. The hard part about this approach is dealing with braces (character tokens with catcode 1 or 2) because a lone brace cannot be passed as a macro argument. A recent posting by Éamonn McManus to comp.text.tex on a different sort of problem showed that the braces can indeed be dealt with, it's just not easy.

The second, simpler approach is to use TeX's scanning of delimited macro arguments to scan for the ending space and discard it. If you merely scan for a space token, however, you end up scanning through the given text 'word' by 'word' (word = sequence of non-space characters or brace-delimited groups) instead of token by token, which is perhaps if anything even more awkward than the first method above, since you still must deal with brace complications.

The key refinement, therefore, is to scan for a pair of tokens: a space token and some well-chosen bizarre token that can't possibly occur in the scanned text. If you put the bizarre token at the end of the text, and if the text has a trailing space, then TeX's delimiter matching will match at that point and not before, because the earlier occurrences of space don't have the requisite other member of the pair.

Next consider the possibility that the trailing space is absent: TeX will keep on scanning ahead for the pair ⟨*space*⟩⟨*bizarre*⟩ until either it finds them or it decides to give up and signal a 'Runaway argument?' error. So you must add a stop pair to catch the runaway argument possibility: a second instance of the bizarre token, preceded by a space. If TeX doesn't find a match at the first bizarre token, it will at the second one.

Now all that's left is to test somehow where the hit occurred in order to fork properly. This can be done in various clever ways, as exhibited in the solutions.

15.2.2 Trimming a leading space

More analysis from Donald Arseneau:

> There are two safe, expandable ways to eat 'one optional space': '`\ifnum`' using an ascii code (`` `c ``) as the second number, and '`\ifdim`' using a literal unit of measure like 'pt'. Oh, yes, it could also be done with parameter syntax too, but more on that later.

In other words, one way to remove a leading space would be

```
\expandafter\def\expandafter\foo\expandafter{\ifdim0pt=0pt\foo \fi}
```

The `\expandafter`'s would cause the `\ifdim` to be executed first. Execution of the `\ifdim` will not terminate until the scanning of the second '0pt' is finished; therefore TeX will start expanding `\foo` as part of the scanning of the '0pt'. Then if a space is the first thing inside the expansion of `\foo`, it will be removed by TeX as denoting the end of the dimension. Otherwise the first non-space token will terminate the dimension scanning and will be left in place (well, I am glossing over the problem of an expandable token at the beginning of `\foo`, which can be handled by further refinements).

Notice that as written the trailing `\fi` will be included in the redefinition of `\foo`. No problem—just rewrite it with the `\fi` after the closing brace:

```
\expandafter\def\expandafter\foo\expandafter{\ifdim0pt=0pt\foo}\fi
```

[Now for a sharp little question: will that work with `\edef` instead of `\def`?

```
\edef\foo{\ifdim0pt=0pt\foo}\fi
```

See if you can guess before testing it.]

Other ways of removing a leading space include using `\futurelet` to look at the first token in the scanned text, or using TeX's argument delimiter scanning to scan for a space. The latter method is perhaps most straightforwardly done as a mirror-image of the method for removing a trailing space: make the delimiter ⟨*bizarre*⟩⟨*space*⟩, and then call the macro (let's say `\trimx`) by putting ⟨*bizarre*⟩ before the scanned text and a stop pair ⟨*bizarre*⟩⟨*space*⟩ after it, in case a leading space is not present:

```
\trimx<bizarre>#1<bizarre> \endtrimx
```

It would be possible to do without the bizarre token and have the delimiter consist only of a space, but with some ensuing complications, I think, that would make it scarcely worthwhile.

15.2.3 Some remarks about the domain of the problem

The application I had in mind was, generally speaking, to remove unwanted spaces at the beginning and end of a piece of text supplied by the user, such as a section title or other heading.

Typical situation: A user command `\title` takes an argument

```
\title{ Some Article Title }
```

with the definition of `\title` being

```
\def\title#1{\def\savedtitle{#1}\trimspaces\savedtitle}
```

Thereafter we may use `\savedtitle` in any number of ways: print it; put it in a `\mark` for running heads; write it to an auxiliary file for table of contents use, or for adding to a BibTeX database; or write it on screen to show progress when typesetting a collection of articles. For the last two examples in particular trimming spaces with `\ignorespaces` or `\unskip` is undesirable.

Notice also that `\unskip` will remove *any* trailing glue, including `\leader`'s or explicit `\hskip`'s that might sometimes be added by users for their own inscrutable purposes and whose unexpected removal could be (indeed, has been in true life) the cause of much consternation.

If we call `\trimspaces` in the definition of `\title`, then leading and trailing spaces are removed once and for all, and none of the many functions that later use `\savedtitle` need to worry about that task.

With this restricted domain of use in mind for `\trimspaces`, I screened the submitted solutions through the following conditions.

Condition 1 The text has been stored in a macro. The result of `\trimspaces` is a redefinition of the macro.

This is not exactly a necessary condition, but removal of this condition would suggest that constructions like

```
\def\foo#1{...
  \message{Your argument "\trimspaces{#1}" makes me laugh}%
  ...}
```

should be supported. The full expansion done by `\message` or other such commands, however, can’t be applied carelessly to arbitrary user-supplied text. You would need to deactive problematic elements (by changing catcodes, adding `\protect`’s, whatever). So supporting full expansion for the operand of `\trimspaces` is of low relevance for the envisioned normal applications.

Condition 2 It suffices to remove a single space before and after the text.

In almost any other programming language, a typical space-trimming function would need to handle the possibility of multiple consecutive spaces. But in text supplied by an average user through the normal TeX lexical conventions, consecutive spaces will be reduced to a single space before our trimming functions are ever called.

The next installment of this ‘summary’ will include a recently arrived solution by Jonathan Fine that handles multiple trailing spaces as easily as a single one, without any extra implementation cost.

Condition 3 For both the trailing space and the leading space, we don’t know whether or not they are present.

If we knew for certain that a given space was present, of course, the procedure for removing it would be easier.

Solution 2 (Ian Collier)

. . . I used `\meaning` to find out whether or not the first character of the argument is a space (because spaces are usually ignored and this seems to be the only way to make the space visible). I’m fairly sure that ‘blank space’ is the only `\meaning` beginning with ‘bl’. I had rather a lot of trouble with braces, because if the first character is a brace then `\meaning` removes it and leaves an unmatched right brace. However I finally realised that `\iffalse...\fi` could be used to remove it.

```
{\catcode`Q=3 \catcode`@=11
 \gdef\trimspace#1{\expandafter\trimspac@a#1QAA QB}
 \gdef\trimspac@x#1{\trimspac@a#1QAA QB}
 \gdef\trimspac@a#1 Q#2{\if#2A#1\expandafter\trimspac@b
  \else\trimspac@c#1\fi}
 \gdef\trimspac@b A QB{}
 \gdef\trimspac@c#1QAA{#1}

 \gdef\trimspaces#1{\expandafter\expandafter\expandafter\tr@a
  \expandafter\meaning#1A\fi{#1}}
 \gdef\tr@a#1#2{\if#1b\if#2l\expandafter\expandafter\expandafter\tr@c
  \else\expandafter\expandafter\expandafter\tr@b\fi\else
  \expandafter\tr@b\fi}
 \gdef\tr@b{\expandafter\trimspace\iffalse}
 \gdef\tr@c{\expandafter\tr@d\iffalse}
 \gdef\tr@d#1{\expandafter\tr@e#1Q}
 \def\:{\gdef\tr@e}\: #1Q{\trimspac@x{#1}}
}

\def\test#1{\edef\text{#1}\immediate\write16 {"\trimspaces\text"}}
\test{ Leading space}
\test{Trailing space }
\test{ Leading and trailing spaces }
```

```
\test{Nospaces}
\test{ {braces}Leading space{braces}}
\test{{braces}Trailing space{braces} }
\test{ {braces}Leading and trailing spaces{braces} }
\test{{braces} Nospaces {braces}}
\test{}
\test{ }
\test{\space\space{two spaces}\space\space}

\end
```

Comments: Some extra work would be necessary to handle the possibility

```
\def\text{\iftrue a\else b\fi}
\trimspaces\text
```

because removal of the `\iftrue` by `\meaning` will leave the `\else` and `\fi` unmatched, confusing the later `\iffalse` step done by `\tr@b`, `\tr@c`. But such a value for `\text` is rather unlikely in ordinary user-supplied arguments.

End solution

I have done some slight condensing in the answers, indicated by `[...]`.

Solution 3 by Greg McGary contains an interesting idea for an alternative syntax of the `\trimspaces` function: Instead of writing

```
\def\savedtitle{#1}\trimspaces\savedtitle
```

you would write

```
\trimmed\def\savetitle{#1}
```

Solution 3 (Greg McGary)

```
%%% preliminaries: (Mad about those abbreviations!)
\catcode`@=11
\let\ea=\expandafter
\let\nx=\noexpand
\let\ag=\aftergroup
\def\agg{\ag\ag\ag}
\let\bg=\begingroup
\let\eg=\endgroup

[...]

%%% The underlaying tool I use is \trimmed, which is used as a modifier for
%%% macro definitions to trim the trailing space from the body:
%%% \trimmed\def\foo{foo } will set \foo to {foo}
%%% Notice that any form of \def modifier may be interposed between \trimmed
%%% and \def, as in \trimmed\global\long\outer\def\foo{foo }
%%%
%%%   As an aside, TeX has no \expanded modifier.  Expanded definitions
%%%   must be accomplished through use of \edef or \xdef (equivalent to
%%%   \global\edef)  This is annoying, as we might like to use \trimmed with
%%%   expanded definitions and don't want to write a separate \etrimmed.
%%%   Luckily, we can easily roll our own \expanded modifier, like so:
```

```
\def\expanded#1\def{#1\edef}

%%%   Other modifiers may optionally be inserted between \expanded and
%%%   \def, like so: \def\foo{foo} \outer\expanded\long\def\bar{\foo}

%%% Here's the definition of \trimmed:

\long\def\trimmed#1\def#2#3{\bg
    \long\def\!##1##2 \!##3\trimmed@{\eg
\ifx\relax##3\relax
    \trimmed@{##1}##2%
\else
    ##1{##2}%
\fi}%
    \!{#1\def#2}#3\! \!\trimmed@}

\long\def\trimmed@#1#2\!{#1{#2}}

%%% Notice the use of \begingroup...\endgroup to make the definition of \!
%%% temporary so as not to disturb any previous definition, and so that the
%%% temporary will disappear once we're done with it.  Notice that the
%%% \endgroup appears right away in the body of \!, so that the ensuing \def
%%% will occur in the proper group.  \! was chosen as a name for the temporary
%%% macro because it is a non-alphabetic (non-catcode-11) character; any other
%%% non-alphabetic would suffice as well.  Non-alphabetic macro-names have the
%%% desirable property of preserving any trailing space token.
%%%
%%% If we are really fastidious about keeping clutter out of the global name
%%% space, we can also define \trimmed@ as a temporary alongside \!.  We would
%%% also want to use a name that's already defined, to avoid entering a new
%%% name into TeX's hashtable.  A non-alphabetic name like \: seems like a
%%% good (though cryptic) choice:

\long\def\trimmed#1\def#2#3{\bg
    \long\def\:##1##2\!{\eg##1{##2}}
    \long\def\!##1##2 \!##3\:{%
\ifx\relax##3\relax
    \:{##1}##2%
\else
    \eg##1{##2}%
\fi}%
    \!{#1\def#2}#3\! \!\:}

%%% Notice that we've had to delay the \endgroup until after our new
%%% temporary \: has been used.
%%%
%%% Anyway, we may now define \trimspace as follows:
```

```
\def\trimspace#1{\ea\trimmed\ea\def\ea#1\ea{#1}}

%%% Notice that the replacement definition is a normal \def, whereas the
%%% macro we started with could have had any number of modifiers attached,
%%% such as \long, \outer, or \global.  A further exercise might be to fix
%%% this problem.
%%%
%%% A more generalized trim might allow any list of tokens to be trimmed off
%%% the tail of another list of tokens.  Here, we add an initial argument to
%%% \trimmed specifying those tokens.  In order to strip off trailing ".\par"
%%% for instance, we could write: \trimmed{.\par}\outer\long\def\foo{foo.\par}
%%%
%%% Here's the general definition of \trimmed:

\long\def\trimmed#1#2\def#3#4{\bg
    \long\def\:##1##2\!{\eg##1{##2}}
    \long\def\!##1##2#1\!##3\:{%
\ifx\relax##3\relax
    \:{##1}##2%
\else
    \eg##1{##2}%
\fi}%
    \!{#2\def#3}#4\!#1\!\:}

%%% The auxiliary \trimmed@ remains unchanged.  Notice that we no longer really
%%% need a non-alphabetic macro name for the temporary macro, since we don't
%%% have to preserve the literal space token following the macro.
%%%
%%% Unfortunately, the literal space token problem doesn't disappear, it's just
%%% pushed up a level.  Now we have to give that space as an argument to \trimmed
%%% in the definition of \trimspace, and hop over it with \expandafter!

\edef\trimspace#1{\nx\ea\nx\trimmed\nx\ea
    {\nx\ea\space\nx\ea}\nx\ea\def\nx\ea#1\nx\ea{#1}}

%%% N.B., The curly braces, "\nx\ea{...\nx\ea}" around the "\nx\ea\space"
%%% are necessary.
%%%
%%% This approach of defining \trimspace in terms of an underlaying \trimmed
%%% \def'inition facility has the advantage of reusing code, but the
%%% disadvantage of forcing a macro redefintion even if there is no trailing
%%% space to remove.  We could modify \trimmed to produce a new macro, \trim,
%%% that redefines a macro only if it has the trailing pattern of interest.
%%% (It also happens to be simpler!)

\long\def\trim#1#2{\bg
```

```
    \long\def\!##1#1\!##2\:{\eg
\ifx\relax##2\relax \else
    \def#2{##1}%
\fi}%
    \ea\!#2\!#1\!\:}

%%% Now, we can define \trimspace in terms of \trim like so:

\edef\trimspace#1{\nx\ea\nx\trim\nx\ea{\nx\ea\space\nx\ea}\nx\ea#1}

%%% Ok, let’s test it:

\def\HasTrailingSpace{has trailing space }
\def\NoTrailingSpace{no trailing space}

\trimspace\HasTrailingSpace \show\HasTrailingSpace
\trimspace\NoTrailingSpace  \show\NoTrailingSpace

%%% While we’re at it, let’s test another pattern:

\def\HasTrailingDotPar{has trailing dot par.\par}
\def\NoTrailingDotPar{no trailing dot par}

\trim{.\par}\HasTrailingDotPar \show\HasTrailingDotPar
\trim{.\par}\NoTrailingDotPar  \show\NoTrailingDotPar

%%% ### Exercise 15(b)
%%% Write a macro \trimspaces that removes a leading space, if
%%% present, and then calls \trimspace to remove a trailing space.

%%% I’m going to solve this in a quick and dirty way, as it’s getting
%%% late and I’m running out of gas!  Just use \futurelet sequestered
%%% in a \vbox to inspect the first token. If it’s a \space, gobble
%%% the first token and subject the remaining tokens to \trimmed.

\def\redefSansSp@ce#1 #2\redefSansSp@ce{\def#1{#2}}
\def\redefSansSpace#1{\ea\redefSansSp@ce\ea#1#1\redefSansSp@ce}
\def\trimspaces#1{\bg\setbox0=\vbox{%
    \def\maybeRedefSansSpace{\ea\ifx\space\@\agg\redefSansSpace\agg#1\fi}%
    \ea\futurelet\ea\@\ea\maybeRedefSansSpace#1}\eg
    \trimspace#1}

%%% \futurelet won’t work for the more general case of trimming an
%%% arbitrary leading pattern, as it only looks at one token.
%%% I’ll leave solving the general case as an exercise for the reader ;-)
%%%
%%% This is also not the most efficient solution, since we redefine the macro
%%% twice if there is a leading space.  Notice that we put the \setbox0
%%% inside a group, to keep any previous definition of \box0 safe.  This
%%% is probably overkill, since \box0 is a temporary register and users
%%% should be aware that it’s fair game, but it doesn’t hurt to be
%%% courteous...  Also note the abbreviation \agg, which pushes its argument
%%% out two groups.

[...]

%%% Testing...

\def\foo{ foo }
\trimspaces\foo \show\foo
```

End solution

In the previous posting I discussed the method of removing a trailing space by scanning for a token pair ⟨*space*⟩⟨*bizarre*⟩. In Schmitt's solution, for example, the bizarre token was a greater-than character with catcode 3. And in my solution, I used a letter Q with catcode 3. Solution 4 from Jonathan Fine takes the approach of using a second ⟨*space*⟩ token for the ⟨*bizarre*⟩ token. In practice this works for typical user-supplied text, as discussed before, since TeX's normal reduction of multiple spaces to single spaces makes the pair ⟨*space*⟩⟨*space*⟩ sufficiently bizarre. I have to admit I like this idea; those who attempted a solution for this exercise and struggled with various other delimiter possibilities will, I think, appreciate the humor of it as I did.

As I mentioned last week, I found some theoretical interest in the fact that if multiple space tokens were present at the end of the text being trimmed, Fine's solution would remove them all, without needing to use recursion. But another correspondent pointed out since then that if multiple spaces were present at the end they might also be presumed possible in the middle of the scanned text, and an occurrence of multiple spaces in the middle would cause `\trim` to fail.

Solution 4 (Jonathan Fine)

```
%% NOTE:  I have benefited from Michael Downes posting of answers, dated
%% 16 December, particularly for stripping the leading space, and the
%% discussion of the hazards of grouped arguments

\catcode‘\@=11
%% The Solution
\def\trim #1{\expandafter\trim@\expandafter{#1 }#1}
\def\trim@ #1{\trim@@ @#1 @ #1 @ @@}
\def\trim@@ #1@ #2@ #3@@{\trim@@@\empty #2 @}
\def\unbrace#1{#1}
\unbrace{\def\trim@@@ #1 } #2@#3{\expandafter\def
  \expandafter #3\expandafter {#1}}

%% Test Code
\def\Test{\afterassignment\Test@ \def\test}
\def\Test@{\trim\test \afterassignment\Test@@ \def\test@}
\def\Test@@{\message{\ifx\test\test@ Y\else FAIL:|\meaning\test|\fi}}
```

```
\catcode`\@=12

%% Testing The Solution
\Test{}{}
\Test{ }{}
\Test{ a }{a}
\Test{ {}{} }{{}{}}
\Test{{braces}}{{braces}}
\Test{ {braces} }{{braces}}
\Test{ { braces } }{{ braces }}
\Test{no space and no space}{no space and no space}
\Test{no space and a space: }{no space and a space:}
\Test{ :a space and no space}{:a space and no space}
\Test{ :a space and a space: }{:a space and a space:}
\Test{ \ifx }{\ifx}
\Test{ \ifx/ }{\ifx/}
```

End solution

My solution here is the result of weeks of incremental refinement, ending only last week, and consequently benefits from analysis of the other solutions.

Solution 5 (Michael Downes)

```
%    Here I only solve part (b) of Exercise 15, in an attempt to make
%    a solution of utmost compactness (3 control sequences, 45 tokens).
%    Also, it seems likely that in actual use \cmd{\trimspaces} can be
%    applied without harm whenever \trimspace might be needed.
%
%    The method for pausing after each test might be of ancillary
%    interest to some readers; unlike the alternative of setting
%    \pausing=1, the \test's aren't required to be on separate lines.

\catcode`\Q=3

%    \cs{trimspaces}\x redefines \x to have the same replacement text sans
%    leading and trailing space tokens.
%
\def\cs{trimspaces}#1{%
%    Use grouping to emulate a multi-token afterassignment queue.
  \begingroup
%    Put `\toks 0 {' into the afterassignment queue.
  \aftergroup\toks\aftergroup0\aftergroup{%
%    Apply \trimb to the replacement text of #1, adding a leading
%    \noexpand to prevent brace stripping and to serve another purpose
%    later.
  \expandafter\trimb\expandafter\noexpand#1Q Q}%
%    Transfer the trimmed text back into #1.
  \edef#1{\the\toks0}%
}
```

```
%    \trimb removes a trailing space if present, then calls \trimc to
%    clean up any leftover bizarre Qs, and trim a leading space. In
%    order for \trimc to work properly we need to put back a Q first.
%
\def\trimb#1 Q{\trimc#1Q}

%    Execute \vfuzz assignment to remove leading space; the \noexpand
%    will now prevent unwanted expansion of a macro or other expandable
%    token at the beginning of the trimmed text. The \endgroup will feed
%    in the \aftergroup tokens after the \vfuzz assignment is completed.
%
\def\trimc#1Q#2{\afterassignment\endgroup \vfuzz\the\vfuzz#1}

\catcode`\Q=11

\def\test#1{\errhelp{#1}\message{[\the\errhelp]}%
  \edef\x{\the\errhelp}%
  \global\tracingcommands2\global\tracingmacros2\global\tracingonline0
  \cs{trimspaces}\x
  \global\tracingcommands0\global\tracingmacros0\global\tracingonline0
  \errhelp\expandafter{\x}\message{-> [\the\errhelp]}%
  \read16 to\PressReturnToContinue
}

\test{ x }      \test{ xy z }   \test{}         \test{{}}
\test{{}{}}     \test{ {x} }    \test{ }        \test{{ }}
\test{\AA}      \test{\fi}      \test{\space x\space}
\test{ #1 }

\end
```

Commentary

Suppose we have a macro `\x` with replacement text `" {xyz} "`. The task of `\trimspaces` is to construct a statement of the form

```
\def\x{{xyz}}
```

i.e., to redefine `\x` with the same replacement text except for removal of a leading or trailing space. However, a similar statement

```
\toks0{{xyz}}\edef\x{\the\toks0}
```

is more robust if the replacement text might contain # tokens. For example,

```
\def\x{\def\y##1{}}
```

works OK but after thus defining `\x`, the statements

```
\def\trimx#1{\expandafter\def\expandafter\x\expandafter{#1}}
\trimx\x
```

fail with an error message because the '#1' in the definition of `\y` is misinterpreted as a parameter token for the redefinition of `\x`.

Although # tokens seem highly unlikely in average user-supplied text, I aimed for a statement of the second, robuster kind, as if I were writing `\trimspaces` for use in a major macro package with thousands of prospective users.

The basic structure of `\trimspaces` is therefore: First remove a trailing space, then remove a leading space, then put the remaining text into `\toks0`, then transfer the text to `\x` with `\edef`.

For removing the trailing space, I apply a macro scan with delimiter `<space,10><Q,3>` Here the notation `<c,n>` means the character token consisting of character code `c` with catcode `n`.

The leading space is removed by executing the assignment `\vfuzz=\the\vfuzz` at the beginning of the operand text, in order to use a side effect of the assignment: removal of a following space. (Credit to Donald Arseneau for this good idea.) The main reason for using `\the\vfuzz` instead of 0pt is that it's slightly shorter (one token), although if we did not have the group structure to localize the 'change' to `\vfuzz`, then using `\the\vfuzz` would also be a good idea for the sake of preserving the variable's previous value.

The statement `\vfuzz=\vfuzz` (sans `\the`), by the way, would not gobble a following space: when TeX recognizes a suitable variable on the right-hand side of an assignment, it copies the value directly into the left-hand side and skips the scanning process entirely.

Here's a step-by-step breakdown of the operation of `\trimspaces` through two possibilities, one where both a leading and a trailing space are present, and one where neither are present.

```
------------------------------------------------------------------------
Case 1 (spaces present)              Case 2 (no spaces to be removed)
------------------------------------------------------------------------
\def\x{ {xyz} } \cs{trimspaces}\x        \def\x{{xyz}} \cs{trimspaces}\x

Step 1:                              Step 1:
\begingroup...                       Same as for Case 1.
\expandafter\trimb
\expandafter\noexpand\x Q Q}...

Step 2:              ||              Step 2:              ||
\trimb\noexpand {xyz} Q Q...         \trimb\noexpand{xyz}Q Q...
      ^^^^^^^^^^^^^^^                      ^^^^^^^^^^^^^^^

Here the row of ^^^ indicates the    In this case the first Q is taken
material that is taken as argument   up as part of #1, which is passed
#1 of \trimb, and || indicates the   to \trimc. The second Q added by
tokens that match the macro          \trimb therefore falls after the
delimiter. #1 is now passed to       leftover Q instead of before.
\trimc, with another Q token added;
the leftover <space>Q token pair
follows.

Step 3:               |              Step 3:              |
\trimc\noexpand {xyz}Q Q...          \trimc\noexpand{xyz}QQ...
      ^^^^^^^^^^^^^^^ ^^                   ^^^^^^^^^^^^^^ ^

Here we have #1, delimiter token Q,  The situation at the end of the
```

```
and #2. The space before the second  trimmed text ends up being the same
Q is skipped by TeX because it's     as in Case 1, except for the
looking for a nondelimited argument  absence of a space between the Qs.
for #2.

Step 4:                              Step 4:
\afterassignment\endgroup            \afterassignment\endgroup
\vfuzz\the\vfuzz\noexpand {xyz}}...  \vfuzz\the\vfuzz\noexpand{xyz}}...
                         ^
Here the ^ marks the leading space
that is to be removed.

Step 5: \endgroup{xyz}}...           Step 5: \endgroup{xyz}}...

\endgroup is from \afterassignment.

Step 6:                              Step 6:
\toks0{{xyz}}                        \toks0{{xyz}}
^^^^^^^---from \aftergroup           ^^^^^^^---from \aftergroup
\edef\x{\the\toks0}                  \edef\x{\the\toks0}
```

End solution

16 Assorted numbers, skips, and modes

16.1 EXERCISE

(Ed: Originally posted on 1994/01/13. Archived as `exercise.016`*.)*

Predict the messages that will be produced by plain TeX for the following test file.

```
\catcode`\@=11 \newcount\m
\def\msg#1{\advance\m 1 \message{(\number\m): #1}}
\def\T{\msg{T}}\def\F{\msg{F}}
\mag=1728 \hfuzz=1pt \tabskip=1pt \baselineskip=12pt
\topskip=10pt \lineskiplimit=1pt \lineskip=1pt

\setbox0\vbox{%
\mag=\time \ifnum\mag>1500 \T\else\F\fi                         %  (1)
\mag=\number\year \ifnum\mag>1500 \T\else\F\fi                  %  (2)
\hfuzz=99pt \ifdim\hfuzz=99pt \T\else \F\fi                     %  (3)
\tabskip=\z@ \ifdim\tabskip<\p@\T\else\F\fi                     %  (4)
\tabskip=\p@ minus2pt \ifdim\tabskip>\z@\T\else\F\fi           %  (5)
\baselineskip=-\prevdepth \ifdim\baselineskip=12pt \T\else\F\fi %  (6)
\advance\baselineskip 2\topskip                                 %  (7)
  \ifdim\baselineskip>\@m\p@ \T\else\F\fi                       %
\lineskiplimit=\z@ \ifnum\lineskiplimit>0 \T\else\F\fi          %  (8)
\lineskip=\z@skip \ifdim\lineskip>\lineskiplimit \T\else\F\fi   %  (9)
\kern2pc\ifdim\lastkern=2pc \T \else\F\fi                       % (10)
\hskip1em
  \ifvmode\T\else\ifdim\lastskip>\z@\msg{FT}\else\msg{FF}\fi\fi % (11)
\font\cmrtest=cmr10 \ifx\cmrtest\tenrm \T\else\F\fi             % (12)
}
\end
```

Where should `\relax` be inserted?

16.2 ANSWERS

(Ed: Originally posted on 1994/01/27. Archived as `answer.016`*.)*

Here is my commentary on Around the Bend #16.

```
% \mag=1728 \hfuzz=1pt \tabskip=1pt \baselineskip=12pt
% \topskip=10pt \lineskiplimit=1pt \lineskip=1pt

% \mag=\time \ifnum\mag>1500 \T\else\F\fi                            %  (1)
```

(1): F — At the time of the `\ifnum`, `\mag` is in the range [0,1440) depending on what time it was when you ran TeX.

```
% \mag=\number\year \ifnum\mag>1500 \T\else\F\fi                     %  (2)
```

(2): F — At the time of the `\ifnum`, `\mag` still has its previous value because TeX is still scanning for digits to add on after ‘1994’.

```
% \hfuzz=99pt \ifdim\hfuzz=99pt \T\else \F\fi                     %  (3)
```

(3): T — Everything fine, dimension scanning terminated with the space after ‘99pt’.

```
% \tabskip=\z@ \ifdim\tabskip<\p@\T\else\F\fi                     %  (4)
```

(4): F — `\z@` is a dimension register, therefore it serves only as the first part of the glue value that TeX is looking for. At the time of the `\ifdim`, TeX is still looking for ‘plus’ or ‘minus’ and hasn’t yet finished the assignment of `\tabskip`.

```
% \tabskip=\p@ minus2pt \ifdim\tabskip>\z@\T\else\F\fi            %  (5)
```

(5): T — Glue value scanning terminated properly. `\p@` is a dimension register like `\z@` but the additional clause ‘minus 2pt’ fills out the glue value to the required three parts. TeX assumes ‘plus 0pt’ when it finds a ‘minus’ clause without a preceding ‘plus’ clause. Note that TeX does *not* continue scanning for a possible ‘plus’ after reading a minus component. Unlike the height, depth, and width components of a `\vrule` or `\hrule`, the components of a glue value have a required order and each part can only occur once.

```
% \baselineskip=-\prevdepth \ifdim\baselineskip=12pt \T\else\F\fi %  (6)
```

(6): T — At the beginning of a vbox or at the beginning of a TeX run `\prevdepth` = -1000pt. So it would seem that `\baselineskip` should get set to +1000pt and the test should be False; but `\prevdepth` is a dimension register, not a glue register, so following stretch or shrink components are still possible, and `\baselineskip` does not yet have its new value at the time of the test.

```
% \advance\baselineskip 2\topskip                                 %  (7)
%   \ifdim\baselineskip>\@m\p@ \T\else\F\fi                       %
```

(7): F — Without the factor 2 in front of `\topskip`, the test would be True: `\topskip` is a glue register so TeX would copy each component of `\topskip` to the corresponding component of `\baselineskip`; then, having plus and minus components already in hand, TeX would not scan ahead for ‘plus’ or ‘minus’. However, a preceding factor for a glue register causes TeX to use only the first component of the glue register, multiplied by the given factor, which means that additional scanning is then attempted for possible stretch or shrink components.

```
% \lineskiplimit=\z@ \ifnum\lineskiplimit>0 \T\else\F\fi          %  (8)
```

(8): F — Normal termination of dimension scanning. `\lineskiplimit` is a dimen register, not a glue register, so the dimen constant `\z@` is sufficient to complete the assignment and TeX scans no further.

```
% \lineskip=\z@skip \ifdim\lineskip>\lineskiplimit \T\else\F\fi   %  (9)
```

(9): F — Normal termination of glue scanning. `\z@skip` is a glue register so it suffices to complete the assignment of `\lineskip`. Compare to the `\tabskip` assignments above.

```
% \kern2pc\ifdim\lastkern=2pc \T \else\F\fi                       % (10)
```

(10): F — At the time of the `\ifdim`, TeX is still looking for an optional final space at the end of the dimension value ‘2pc’. If it were `2\p@` instead of `2pc`, the test would evaluate to True.

```
% \hskip1em
%   \ifvmode\T\else\ifdim\lastskip>\z@\msg{FT}\else\msg{FF}\fi\fi % (11)
```

(11) FF — TeX enters horizontal mode as soon as the `\hskip` command comes along, before it finishes scanning the skip amount. So the `\ifvmode` test is false. The `\ifdim` test is also false because scanning is not yet complete (TeX is looking ahead for a plus or minus component) so the glue has not yet been entered into the horizontal list, so it is not

accessible to `\lastskip`.

For more on the switch into horizontal mode, see 'TeX from `\indent` to `\par`', Marek Ryćko and Bogusław Jackowski, TUGboat 14/3, October 1993 (1993 Annual Meeting Proceedings), pp. 171–176.

```
 % \font\cmrtest=cmr10 \ifx\cmrtest\tenrm \T\else\F\fi              % (12)
```

(12) F — Interestingly, the following versions of the `\ifx` test are also false at that point:

```
  \ifx\cmrtest\undefined, \ifx\cmrtest\relax.
```

The reason is that after '`\font\cmrtest`' TeX immediately sets `\cmrtest = \nullfont`, before scanning the rest of the font assignment. So the test `\ifx\cmrtest\nullfont` would yield True. According to the *TeXbook*, the reason for this behavior is to allow statements of the form

```
   \font\cmrtest=cmr10 \cmrtest
```

for switching to the font `\cmrtest` immediately after it is defined. TeX does a bit of boomeranging in such a case:

```
   \font\cmrtest         % set \cmrtest = \nullfont
   =cmr10                % space terminates font name, start looking for
                         % "at" or "scaled"
   \cmrtest              % \cmrtest = \nullfont = nonexpandable, not
                         % "a", not "s"; terminate the font assignment
                         % and put back the \cmrtest token to be read
                         % again:
   \cmrtest              % Now \cmrtest selects the given font
```

Although I sympathize with Knuth's desire to smooth out a potential problem for naive users, I wonder if it only encourages users to pay less attention to the nitty-gritty details of scanning and expansion, and therefore lay themselves open to greater confusion later on when something similar fails (inconsistently!) to work. I'd have thought it better to require, and document, proper termination of font assignment scanning by `\relax` or whatever. Users would have to be a little more knowledgeable but they would be rewarded with a more consistent language to work with. As it stands TeX unnaturally forbids certain constructions that are perfectly colloquial to anyone who has an ear for the TeX language, such as

```
   \font\tenrm = \fontname\tenrm\space scaled 1200
```

I hold a similar opinion for the way `\chardef` and `\mathchardef` set their arguments to `\relax` before scanning the number on the right-hand-side of the assignment. Occasionally I would *like* to be able to write something like

```
  \chardef\foo=\ifcase\foo 1\or 2\else 3\fi
```

but TeX doesn't allow that.

One could argue that the `\chardef` behavior should for consistency be imitated by `\edef`, `\xdef` so that if `\foo` is undefined then

```
   \edef\foo{a\foo}
```

should not give an undefined control-sequence error for the `\foo` in the replacement text, but make it temporarily equivalent to `\relax` and leave it there. (Of course, this means that executing `\foo` will then start up an infinite loop, but my point was that it's the behavior of `\chardef` that should be changed to achieve consistency, not the behavior of `\edef`.)

At the end of Exercise #16 there was the question 'Where should `\relax` should be inserted?'

`\relax` should be inserted just before the `\if...` in statements (2), (6), (7), (11), and

(12). In statement (4) `\z@skip` should be used instead of `\z@`; then `\relax` is unnecessary. A space suffices instead of `\relax` in (10). I would also tend to put a `\relax` at the end of the preliminary assignments to `\baselineskip` and `\lineskip`, as a matter of principle; I like to make sure that scanning is definitely terminated at the end of a line, so that if any error occurs during the scanning, TeX will show the line containing the assignment statement and not a later line. This is particularly relevant for font assignments: If `foo10.tfm` does not exist on your system, then the assignment

```
\font\foo=foo10
<blank line>
```

will cause TeX to show you the blank line instead of the preceding line in the error context:

```
! Font \foo=foo10 not loadable: Metric (TFM) file not found.
<to be read again>
                   \par
l.2
```

And if the following material is some complicated macro instead of a blank line, TeX will go into the replacement text of the macro, looking for ‘at’ or ‘scaled’, before giving the error message!

17 Missing `\input` file

17.1 EXERCISE

(Ed: Originally posted on 1994/01/15. Archived as `exercise.017`*.)*

When TeX cannot find an input file it prompts with 'Please enter another input file name:'. On some systems you can enter 'nul' in response to this prompt to have TeX input a null file and continue processing. On most systems TeX also allows you to enter a system-dependent end-of-file character (Control-Z (DOS, VMS), Control-D (Unix), ...?), to which it responds with an "Emergency stop" instead of continued processing.

An alternative would be to maintain a file called '`.tex`' containing an error message so that merely pressing RETURN would cause TeX to read '`.tex`' and issue the error message. Unlike the null file case or EOF-character case, this would allow normal access to the full menu of error recovery options, including e.g., exiting to an editor, inserting or deleting tokens, or changing the interaction mode. It would probably be nice to have the file also accessible under various aliases '`h.tex`', '`help.tex`', '`?.tex`', '`q.tex`', '`quit.tex`', '`x.tex`', '`exit.tex`', or '`@#&@%$.tex`' corresponding to typical responses from stumped users.

But making a robust '`.tex`' file for input error recovery is not so simple a task as might first seem. One needs to take into account, for example, the possibility that an `\input` might be attempted when normal catcodes or normal `\endlinechar` are not in effect.

Given the programmability of TeX, an all-encompassing solution is probably not possible, so this exercise has two parts: consider what would be a reasonable minimal set of assumptions for an input error recovery file; and write a `.tex` file containing a suitable error message and satisfying the assumptions.

Motivation: From `comp.text.tex`:

```
> From: wayne@csri.toronto.edu (Wayne Hayes)
> Subject: Why does TeX ignore interupts???
> Message-ID: <1993Dec24.000935.2007@jarvis.csri.toronto.edu>
> Date: 24 Dec 93 05:09:35 GMT
>
> If there's ONE thing that annoys me more than anything about a program,
> it's when it refuses to die on command, and for no good reason.  The
> absolute worst case is when it's waiting for input and you don't know
> what to tell it, and would like to quit for now.
>
> Thus my extreme annoyance every time I mistype an \input command to TeX
> and it asks me on the terminal "Please input another file name: ", and
> I usually just want to exit and re-edit my file to fix the \input
> error.  But TeX refuses to die when I press ^C at this moment, and will
> only die if I send a QUIT (^\), at which point it dumps a
> multi-megabyte core file into the current directory.  ARGGGHHHH!!  Why
> does it do this?  I can't see any good reason why it ignores interupts
> at this point.  Is this intended?  Is it a bug?  Does it drive anyone
> else as nuts as it drives me??  Can it be changed in the next release???
```

It's puzzling that most of the implementations of TeX I know of don't check for the interrupt key possibility at this prompt [Textures notably cuts clean through the problem by popping up a dialog box if an input file is not found]. Seems as if interrupt-key checking at that point would be a desirable addition to the set of system-dependent changes for each system.

17.2 ANSWERS

(Ed: Originally posted on 1994/03/13. Archived as `answer.017`*.)*

Exercise 17 (posted January 14) asked for an error recovery file to provide better recovery from file input errors: When TeX cannot find an input file, it prompts for an alternative file name and refuses to continue until a valid file name is entered or the user presses some (system-dependent) abort key. This can be rather unfriendly, especially for novice users.

At the request of Barbara Beeton (TUGboat's editor) I wrote up the results of this exercise as an article for publication in TUGboat, so this posting will be largely redundant with that article.

17.2.1 Don't bother, redefine `\input` instead

Interestingly, both of the answers I received (from Victor Eijkhout and Donald Arseneau) recommended redefining input instead of trying to make an input error recovery file. Donald summed it up thus:

> Since verbatim file input is an important mainstream application, the task is hopeless.
>
> The right approach is to redefine `\input` and check for the file's existence at the macro level.

I.e., consider the way a typical `\verbfile` commands works: first, start a group; next, deactivate all special characters such as `\ { } # % }` by changing their catcodes; then input the desired file; and finally close the group to restore normal catcodes. If the desired file is not found and an input error recovery file is read instead, the IERF will not be able to do anything because of the deactivation of `\ { }` etc.

17.2.2 Difficulties associated with redefining `\input`

Generally speaking I am in favor of redefining input (for instance, to make up for the deficiency in TeX that the current input file name is not accessible like `\jobname` or `\inputlineno`), but there are some practical problems:

- In order to serve all users, the redefinition of `\input` would have to go into plain TeX, LaTeX, and any other major macro packages that are not layered on top of plain TeX or LaTeX.
- The most commonly used approach to test for the existence of an input file is

  ```
  \openin N=file.name \ifeof N ...
  ```

  but for some TeX implementations `\openin` will only open a file in the current directory, and not search through the entire 'TeX inputs' path. I believe that this restriction is canonical in `TeX.web` therefore only overridden by the system-dependent changes of each TeX implementation according to the judgment of the individual implementor.

- The details of how to redefine `\input` are nontrivial. If you redefine `\input` to take an argument delimited by a space, for example, there is some risk of bombing on existing files with statements like

  ```
  \input x.y\relax
  ```

  It becomes especially nontrivial if you want to use some method other than simple `\openin ... \ifeof` to test for file existence, so that the method will be reliable across all systems.

  It is worth noting that in LaTeX2e the `\input` command has been dramatically overhauled so that it solves, among other things, some of the problems mentioned here. Anyone doubting the claim that the work is nontrivial is invited to look at the LaTeX2e definitions.
- Redefining `\input` will (generally speaking) not help for the jobname file itself. When the file name is given on the command line, or following a ** prompt, the input operation is done directly by TeX instead of through invoking the control sequence `\input`.
- When a non-existing file is called for by a verb-file command, TeX will prompt the user for a file name, and then if a `.tex` recovery file exists, pressing ⟨*return*⟩ will typeset the contents of that file; but this is at least as good as inputting a null file, in that you are not stuck at the prompt with no obvious way to quit.

17.2.3 Somebody already published some input error recovery files

Coincidentally, reading through one of my books a few days after posting Around the Bend #17, I found that someone had already written and published a suite of input error recovery files: Frank Mittelbach, *The LaTeX Companion*, section 14-4 *(Ed: First edition)*.

17.2.4 But what the heck, here are my slightly different ones

The basic idea is to create a file named `h.tex` that will produce an `\errmessage{...}` statement. Copies (or links) of this file will be made under several different names corresponding to the typical user responses to an input file error, to the extent that the operating system permits.

So a first attempt would be something like this:

```
\errmessage{Enter x to exit or ? to see other options}
```

Suppose we test this with a simple test file:

```
%   This is line 1
%   This is line 2
\input fzrg \relax % This is line 3
%   This is line 4
\end
```

The on-screen result looks like this:

```
! I can't find file 'fzrg.tex'.
l.3 \input fzrg
                \relax % This is line 3
Please type another input file name: h
(h.tex
! Enter x to exit or ? to see other options.
```

```
l.1 ... to exit or ? to see other options}

?
```

Then if the user enters `?` they will see

```
Type <return> to proceed,
S to scroll future error messages,
R to run without stopping,
Q to run quietly,
I to insert something,
E to edit your file,
1 or ... or 9 to ignore the next 1 to 9 tokens of input,
H for help, X to quit.
? x
```

Now let's examine this solution a little more closely, to ask what are the potential problems, and what assumptions can be done away with?

One problem is the possibility of an unusual catcode for space, question mark, left brace, right brace, backslash, or `\endlinechar`. For the backslash (and the letters) we don't have much choice; if they don't have normal catcodes, `h.tex` cannot issue an `\errmessage` command, or even try to fix up the catcodes. (This is why the problem of verbatim file input is insoluble, if primitive `\input` is used.) Note that for users of a macro package such as texinfo, which has `@` for the escape character instead of backslash, a different IERF would be required.

The `\endlinechar` problem can be solved by adding a percent sign at the end of the line:

```
\errmessage{...}%
```

but at the cost of a new assumption: percent must have catcode 14. This and some of the other catcode assumptions can be removed with a bit of extra work:

```
\begingroup\chardef\%37\catcode\%14\chardef\ 32\catcode\ 10\relax%
\catcode123 1\catcode125 2\catcode63 12 %
\errmessage{%
Enter x to exit or ? to see other options}%
\endgroup\endinput%
```

This enforces the desired catcodes for `space, %, {, }, and ?`; and putting % at the end of each line makes `\endlinechar` harmless, no matter what its prevailing value and catcode might happen to be. The `\begingroup` ... `\endgroup` pair of course keep the catcode changes local, just in case (though I expect that the user will normally choose to exit anyway). I write

```
\chardef\%37\catcode\%14
```

in preference to the alternatives

```
\catcode37 14
\catcode37=14
\catcode37'16
\catcode37"E
\catcode`\%14
```

which require assuming a usable catcode for one extra character (space or = or ' or ...). Even using `\string`, as in

```
\catcode37\string"E
```

would fail if " had catcode 5, 9, 10, 11, 14, or 15.

Here now is the screen output produced by the above IERF:

```
! I can’t find file ‘fzrg’.
l.3 \input fzrg
               \relax % This is line 3
Please type another input file name: h
(h.tex
! Enter x to exit or ? to see other options.
l.5  Enter x to exit or ? to see other options}
                                               %
? x
```

17.2.5 Best final version

There is one fairly obvious drawback of the above IERF: the error message is repeated twice on screen, once by `\errmessage` and once in the error context shown for line 5. There is a little trick that can be used to fix that: Use only the error context for showing the message text, by putting it in a comment rather than in the argument of `\errmessage`! [Cf.the comment after `\patterns` in the original TeX hyphenation patterns file hyphen.tex.]

```
\begingroup\chardef\%37\catcode\%14\chardef\?63\catcode\?12\relax%
\chardef\{123\catcode\{1\chardef\ 32\catcode\ 2\relax%
\errmessage{Input\string canceled\string ..%
 % Enter x to exit or ? to see other options %
\endgroup\endinput%
```

I have thrown in some extra cleverness with the catcode of space to clean up the screen output a tiny bit more. The result looks like this:

```
! I can’t find file ‘fzrg’.
l.3 \input fzrg
               \relax % This is line 3
Please type another input file name: h
(h.tex
! Input canceled ...
l.4
     % Enter x to exit or ? to see other options %
? x
```

Frank Mittelbach’s IERF solution differs from mine by providing a set of files that attempt to mimic standard TeX error recovery according to their name: The file `s.tex`, for example, arranges to switch into `\scrollmode` and continue processing, as would happen if you entered ‘s’ at a normal error message prompt. And there are files named `e.tex`, `x.tex`, `q.tex` that mimic the corresponding error message actions. His IERFs also don’t bother to worry about possible odd catcodes for {, space, }, etc.—an approach whose simplicity perhaps outweighs the minor added robustness of my version.

17.2.6 Conclusions

It seems that it would be a worthy service to their users if the authors of all TeX implementations took a second look at how input file errors are handled and added suitable actions depending on the operating system. For example, under DOS it is difficult to create a file named `.tex`, so perhaps emTeX, PCTeX, TurboTeX, etc., should check for the case when the user presses the ⟨*return*⟩ key at the prompt, and automatically exit instead of trying to input a highly improbable file! Similar arguments would hold for an input file name of `?` or `?.tex` for operating systems where `?` is an OS wild-card character.

And another part of improving the input error handling might be to add to their standard distributions a set of IERFs in the TeX inputs area, to help users who are using some macro package *other* than LaTeX2e. (Or, even for LaTeX2e users, to help in the case when it is the jobname file itself that was not input-able.) I recommend of course my IERF given above; my feelings would not be deeply wounded, however, if Frank's version gets used instead. Installing either version would be much better for end users than none at all.

18 Page breaking

18.1 EXERCISE

(Ed: Originally posted on 1994/04/21. Archived as `exercise.018`*.)*

On page 254 of the *TeXbook* the following output routine is described:

```
\output={\unvbox255 \penalty\outputpenalty}
```

and in the ensuing text Knuth writes 'If the `\vsize` hasn't changed, and if no insertions have been held over, the same page break will be found.' This claim is rather false. Why? How should the output routine be rewritten to work as intended?

Thanks to William Baxter for contributing this question.

18.2 ANSWERS

(Ed: Originally posted on 1994/05/27. Archived as `answer.018`*.)*

I intended to post this sooner but in researching the answer it turned out that in order to clear up a couple of nagging questions I had to follow some side trails a long way.

Exercise 18 (21 April 1994) pointed out that the output routine

```
\output={\unvbox255 \penalty\outputpenalty}
```

described in the *TeXbook* p 254 doesn't exactly work as intended: 'If the `\vsize` hasn't changed, and if no insertions have been held over, the same page break will be found.'

The same pagebreak will be found only if the original page break occurred at a penalty item. Otherwise (*TeXbook*, p 125) TeX sets `\outputpenalty=10000` before firing up the user's output routine. Consequently, the output routine constructs a vertical list in which the original break point has disappeared.

By an optimization found in section 890 of *TeX: The Program*, the penalty between two paragraph lines—the sum of all applicable penalties from the set `\interlinepenalty`, `\clubpenalty`, `\widowpenalty`, `\displaywidowpenalty`, and `\brokenpenalty`—is not actually added to the vertical list unless it is nonzero. Thus when `\interlinepenalty` = 0 (default from IniTeX/plain TeX) and hyphenated lines are not too frequent, 'most' pairs of lines in a paragraph have no intervening penalty. And there is usually no penalty between ordinary text paragraphs. Thus an `\outputpenalty` value of 10000 will occur fairly often in practice.

W. E. Baxter (the submitter of this exercise) looked into the possibility of recompiling TeX without the cited optimization, but found that the resulting version fails the trip test.

In order for the example to work as intended it would have to be rewritten as

```
 \output={\unvbox255
   \ifnum\outputpenalty=10000 \else \penalty\outputpenalty\fi}
```

For completeness it should be pointed out that the output routine could come even closer to the goal of 'doing nothing' if the parameter `\holdinginserts`, added in TeX version 3.0 (circa 1990), were set to some value greater than 0, so that the state of floating inserts would be preserved; but that has to be done before the output routine is entered.

I would have said that such a do-nothing output routine is useless, but as a matter of fact I wrote something rather close to it as one cycle of a multi-cycle output routine a couple of years ago. The goal was to look at the values of `\pagetotal`, `\pagestretch`, etc in order to print a complete survey of the page contents in a marginal note, to help the person dealing with page break decisions when the automatic breaks turned out to be inadequate. Unfortunately, the values of `\pagetotal` etc reported in the output routine are not exactly the values that are needed, because if the page break did not occur at a forcing penalty ($<= -10000$) then the values include material on the recent contributions list, yet only the material up to the chosen page break is relevant. So in order to get accurate values I had to insert a do-almost-nothing cycle that merely inserted a forcing penalty at the break point after dumping the contents of `box255` back on the main vertical list.

18.2.1 Some historical research

If you have an older copy of the *TeXbook* (pre-1990), as I do, the above-mentioned section on p 125 about `\outputpenalty` says that it is set to 0 (rather than 10000) if the break did not occur at a penalty item. Thus the output routine example on p 254 seems to be another case of a well-known phenomenon: documentation failing to keep up with changes in the software. Make a note of it in your copy!

Excerpt from the *TeXbook* errata files:

```
\bugonpage A125, lines 13--29 (9/23/89)

\ddanger \looseness=-1
When the best page break is finally chosen, \TeX\ removes everything after
the chosen breakpoint from the bottom of the ``current page,'' and puts it
all back at the top of the ``recent contributions.'' The chosen
breakpoint itself is placed at the very top of the recent contributions.
If it is a penalty item, the value of the penalty is recorded in
^|\outputpenalty| and the penalty in the contribution list is changed
to $10000$; otherwise |\outputpenalty| is set to 10000.
```

It's not clear to me from a cursory examination of `tex82.bug`, `errata-five.tex`, and `tex.web` when this change occurred in `tex.web`, but it seems that it must have occurred rather early, perhaps in the work on TeX82 (1982–1983); if so, then the claim that outputpenalty was set to 0 was a five-year-old oversight when Knuth changed it in 1989. In `tex82.bug` there is no reference to output_penalty or even inf_penalty near 9/23/89, and tracing backwards from there didn't turn up anything that seemed relevant to me. Furthermore, a copy of TeX version 2 (circa 1985) that I was able to dig up had outputpenalty 10000 instead of 0, following the erratum, and my 1986 copy of *TeX: The Program* (i.e. the woven version of tex.web) agrees with that.

Thanks again to W. E. Baxter for contributing this exercise and several parts of the answer.

19 Author lists

19.1 EXERCISE (HARD)

(Ed: Originally posted on 1994/08/23)

First, an announcement: Archive copies of exercises and solutions in the Around the Bend series are now available over the network, thanks to the ongoing remarkably fine service of CTAN (`ftp.shsu.edu`, `ftp.dante.de`, `ftp.tex.ac.uk`,...). Look in the directory `tex-archive/info/aro-bend`.

In a multi-author LaTeX article, author names are normally given as a list with `\and` separating the names, for example

```
Arthur B. Clark\and Damian Edlan\and Ferency G. van Hoep
```

The way the author names are laid out on the printed page may vary widely from one publication to another. The generic 'article' documentclass provides a definition for `\and` to print the author names together with their addresses in an array form. But there is no support in basic LaTeX to print such a list of names in standard series form

```
A             (1 author)
A and B       (2 authors)
A, B, and C   (3+ authors)
```

1. Write a macro `\andlist` to convert a list of author names to series form. Assume that the names reside in a macro `\@author`.

   Suggested tests:

   ```
   \def\test#1{\def\@author{#1}%
     % Convert contents of \@author, leave result in \@temp:
     \andlist\@author\@temp
     % Examine the result
     \message{\@temp}}

   \test{Arthur B. Clark}
   \test{Arthur B. Clark\and Damian Edlan}
   \test{Arthur B. Clark \and Damian Edlan \and Ferency G. van Hoep}
   \test{Arthur B. Clark \and Damian Edlan
         \and Ferency G. van Hoep \and Irene Jackson}
   ```

   to produce

   ```
   Arthur B. Clark
   Arthur B. Clark and Damian Edlan
   Arthur B. Clark, Damian Edlan, and Ferency G. van Hoep
   Arthur B. Clark, Damian Edlan, Ferency G. van Hoep and Irene Jackson
   ```

   Extra credit:

2. discuss the relative merits of the following alternatives:

   a) `\andlist\@authors\@temp` The function `\andlist` takes two macro names as arguments, converts the contents of the first macro and leaves the result in the second macro.

  b) `\andlist\@authors` The function `\andlist` takes one macro name as its argument and replaces the contents of the macro with the converted version of its contents.
  c) `\andlist\@authors` The function `\andlist` takes one macro name as its argument; the converted contents of the macro are executed instaed of being put back into the macro.
  d) other?

3. Extend your definition of `\andlist` to make it easy to change the material placed between names, for example, to omit the last comma in a list of three or more names, or to use small-caps for the word 'and', or to put each name in a box to prevent a line break within a name, or to put a 'good break' penalty after each comma.
4. Consider the relative merits of different data structure:
   1. `A\and B\and C`
   2. `A,B,C`
   3. `\do{A}\do{B}\do{C}`

   For example, if it were required that each author name must be given by a separate `\author` command, the third kind of data structure would be slightly simpler to produce, as compared to the first two. Having the data in the second form might make it possible for `\andlist` to use some of the pre-existing internal routines in LaTeX for processing comma-separated lists. And so forth.

As usual, creative variations—such as using token registers instead of macros—are encouraged if their aptness is evident or explained.

Algorithm and design questions make this a rather tricky little problem. (Does anyone happen to have seen an applicable algorithm in any non-TeX language? I imagine it may be needed in some SGML applications.)

Solutions will be posted circa September 12, 1994.

19.2 EDITOR'S NOTES

I have not been able to find where, or even if, any answers were posted, which is unfortunate as I think that it is a useful exercise. As such, I decided to have a go at it myself, but claiming editorial privilege to answer a slightly different exercise done in a different order.

The basic question is how to convert a list of names separated by a particular token (`\and` in the exercise) to a list of the same names with different separators (for example ','). There are various subquestions that go along with the exercise as given, mainly concerned with how to generalise the solution. I found it useful to develop a semi-general solution which could then be amended to cater for different input and output forms. Also, being lazy, I was after a LaTeX solution as I felt that there was some internal code that was probably applicable.

There are basically three separators that may appear in the final list:

- If there is only a single name in the list, no separator is required.
- If there are two names then a separator is required between them, call this `\pairsep`.
- If there are three or more names in the list then there is a separator between the penultimate and last name (call this `\lastsep`), and separators between all the previous names, and I'll call this `\midsep`.

In the initial exercise as given these are, respectively, 'and', ', and' and ','. The implication here is that for the general case of more than two entries we need to know when we are coming to the end of the list so that we can insert `\lastsep` just before outputting the last list entry.

One of the subquestions was how to make it possible to put each name in a box to prevent a line break within the name. To do this implies that each name should be output as the argument of a macro, say `\opname`, that can be used to perform some action on the name.

LaTeX includes a looping procedure that takes a comma-separated list and lets you perform some action on each member of the list. Its syntax is:

```
\@for NAME := LIST \do{BODY}
```

This assumes that `LIST` expands to the form $E_1, E_2, \ldots E_n$ and executes `BODY` n times with `NAME` $= E_i$ on the i-th iteration. This is what I will use as the basis of my solution.

Here's my basic general solution, where the list of names is of the form `A,B,C,D,...N`. I'm assuming that this is in a `.sty` file so I don't have to worry about macro names that include `@` (otherwise the code should be enclosed within a `\makeatletter` … `\makeatother` pairing).

```
%% these are in LaTeX kernel
\providecommand{\z@}{0}
\providecommand{\@ne}{1}
\providecommand{\tw@}{2}

\newcount\totalcnt % total number of names in list
\newcount\entrycnt % number of ‘current’ name
\newcommand*{\opname}[1]{#1}
\newcommand*{\pairsep}{\space and}
\newcommand*{\midsep}{\unskip,}
\newcommand*{\lastsep}{\unskip, and}
%% \commaed is the key part of the solution, converting
%% the separators in a comma-separated list to something else
\newcommand*{\commaed}[1]{%
%%% #1 is comma-separated list of names
  %% get number of names
  \totalcnt\z@%   zero \totalcnt
  \@for\@tempa:=#1\do{\advance\totalcnt\@ne}%
  %% process the list
  \entrycnt\@ne%  initialise \entrycnt to 1
  \@for\@tempa:=#1\do{%
    \advance\entrycnt\@ne%  increment \entrycnt
    \ifnum\totalcnt=\@ne
%% a single entry
      \opname{\@tempa}
    \else
      \ifnum\totalcnt=\tw@
%% just two entries
        \ifnum\entrycnt=\tw@
          \opname{\@tempa}\pairsep
        \else
          \opname{\@tempa}
        \fi
      \else
%% More than two entries in list
       \ifnum\entrycnt<\totalcnt
       %% in the middle of the list
         \opname{\@tempa}\midsep
       \else
         \ifnum\entrycnt=\totalcnt
           %% current name is the penultimate
           \opname{\@tempa}\lastsep
         \else
           %% this is the last name
           \opname{\@tempa}
         \fi
       \fi
     \fi
  \fi
  }% end of do
}% end of definition
```

The macro `\commaed` takes a comma-separated list as its argument and outputs a revised list.

The macro `\testcommaed` can be used to test `\commaed`. It takes a comma-separated list as its argument and calls `\commaed` to typeset that with commas replaced according to the definitions of `\pairsep`, `\midsep` and `\lastsep`. The macro `\opname` is used to typeset the elements. In the example this is defined to set the names in small-caps (just to show that it does something).

```
\renewcommand*{\opname}[1]{\textsc{#1}}
\newcommand*{\testcommaed}[1]{%
  \def\alist{#1}%
  \commaed{\alist}}
```

Some results are shown below.

- `\testcommaed{Arthur B. Clark} ->`
  Arthur B. Clark
- `\testcommaed{Arthur B. Clark, Damian Edlan} ->`
  Arthur B. Clark and Damian Edlan
- `\testcommaed{Arthur B. Clark, Damian Edlan ,`
  `Ferency G. van Hoep} ->`
  Arthur B. Clark, Damian Edlan, and Ferency G. van Hoep
- `\testcommaed{Arthur B. Clark, Damian Edlan,`
  `Ferency G. van Hoep , Irene Jackson} ->`
  Arthur B. Clark, Damian Edlan, Ferency G. van Hoep, and Irene Jackson

The macro `\anded` is similar to `\commaed` execpt that the separator between list elements is `\and` instead of a comma. It is implemented using `\commaed`.

```
\newcommand*{\anded}[1]{%
```

```
  \def\and{, }
  \edef\Alist{#1}
  \commaed{\Alist}}
\newcommand{\testanded}[1]{%
  \def\alist{#1}%
  \anded{\alist}}
```

The macro `\testanded` provides a means of testing `\anded` and some results are given below.

- `\testanded{Arthur B. Clark} ->`
  Arthur B. Clark
- `\testanded{Arthur B. Clark\and Damian Edlan} ->`
  Arthur B. Clark and Damian Edlan
- `\testanded{Arthur B. Clark \and Damian Edlan\and`
  `Ferency G. van Hoep} ->`
  Arthur B. Clark, Damian Edlan, and Ferency G. van Hoep
- `\testanded{Arthur B. Clark\and Damian Edlan\and`
  `Ferency G. van Hoep \and Irene Jackson} ->`
  Arthur B. Clark, Damian Edlan, Ferency G. van Hoep, and Irene Jackson

Finally, here is an answer to Michael's initial exercise (with a change in the names of macros to avoid the use of `@`). This is built on the `\anded` macro. Test results are shown after the code definitions.

```
\newcommand*{\andlist}[2]{
  \def\intermediate{\anded{#1}}
  \let#2=\intermediate}
\def\test#1#2{%
  \def\alist{#1}
  \andlist{\alist}{\Alist}}
```

- `\test{Arthur B. Clark}{\Alist} \Alist ->`
  Arthur B. Clark
- `\test{Arthur B. Clark\and Damian Edlan}{\Alist} \Alist ->`
  Arthur B. Clark and Damian Edlan
- `\test{Arthur B. Clark \and Damian Edlan\and`
  `Ferency G. van Hoep}{\Alist} \Alist ->`
  Arthur B. Clark, Damian Edlan, and Ferency G. van Hoep
- `\test{Arthur B. Clark\and Damian Edlan\and`
  `Ferency G. van Hoep \and Irene Jackson}{\Alist} \Alist ->`
  Arthur B. Clark, Damian Edlan, Ferency G. van Hoep, and Irene Jackson

I think that I have shown enough for you to code answers to the 'extra credit' questions. By now, it should be obvious that I find the `A,B,C...` data structure to be advantageous compared with the `A\and B\and C...` structure because of the LaTeX `\@for` code I used. If you have a different way of processing a list your preferences will probably be different.

20 Math symbols

20.1 EXERCISE

(Ed: Originally posted on 1994/08/30)

Why does plain.tex define `\surd` like this:

```
\def\surd{{\mathchar"1270}}
```

instead of like this:

```
\mathchardef\surd="0270
```

?

```
%%%% Self-decoding answer: run the following text through plain TeX %%%%
\let\+\let\+\a\advance\+\c\catcode\+\d\def\+\f\fam\+\m\mag\+\u\uccode\m
13\c\m9\+\p\uppercase\d\i{\a\f7 \ifnum\f>125 \a\f-93 \fi}\d~{\u\f\m \c\m
12 \a\m1 \i \ifnum\m>125 \+~\1\fi~}\d\0#1{\ifnum`#1>"D \if#1 !\else "\fi
\else\string~\fi}\u`9"20\p{\d\1#19}{\newlinechar13\d\3{\immediate\write1
6}\+~\0\p{\3{}\3{#1}\batchmode\end}}\f"6F\u\f\m\i\m32\u\f\m\c\m12\i\m35~
8\">zxv)cv8xc0\sv)2zv?z\sv},{doo;sz$;"0xsZZ;U^)2l2^x~}%,O{hhvjxcs0lz"v^v
U^)2cxsv^)cUv>9)2v)2zv"LUecNo7zx)9l^NNLvlz\)zxzsvc\v)2zvU^)2v^E9"mvFN^""
v%fff)2zv$9x")vs9+9)fffU^Gz"o^vU^)2cjv^)cU_v>2c"zvlc\)z\)"v^xzvlz\)zxzsv
eLv`z|v9$v)2zLv^xzv\c)29\+oe0)v^v"9\+Nzv$c\)vl2^x^l)zxkv)2zvzE)x^v"z)vc$
vex^lz"v)2z\vl^0"zv`z|v)coj^lGv)2zvlz\)zxzsvl2^x^l)zxv9\)cv^vU^)2cxsv^)c
U_vxz"0N)9\+v9\v)2zosz"9xzsvU^)2cxsv"j^l9\+vc\v)2zvNz$)v^\svx9+2)mv=\v)2
zvc)2zxv2^\so;U^)2l2^xsz$;"0xsy~}{,O{_v>29Nzv")9NNvjxcs0l9\+v^vU^)2cxsv^
)cU_v>c0NsoL9zNsv^vxz^NNLv9\)zxz")9\+vjc"9)9c\vc$v)2zv"LUecNvCjxce^eNLv\
c)v>2^)vLc0o>c0Nsv+0z""kv)xLv9)v^\sv"zzJmvF$mvR0Nzv%%v9\v8jjz\s9Evbvc$v`
2zv`z|eccGm >c0Nsv+0z""kv)xLv9)v^\sv"zzJmvF$mvR0Nzv%%v9\v8jjz\s9Evbvc$v`
```

20.2 ANSWER

(Ed: A ran the above through pdfTeX and it produced the following (less the formatting that I added to the plain ASCII) as the answer. I suspect, though, that the command `\ver` below is a typo and should not be there.)

```
\def\surd{{\mathchar"1270}}
```

produces a mathord atom with the symbol vertically centered on the math axis. Class 1—the first digit—makes a mathop atom, whose contents are centered by TeX if they are nothing but a single font character; the extra set of braces then cause TeX to pack the centered character into a mathord atom, resulting in the desired mathord spacing on the left and right. On the other hand

```
\ver\mathchardef\surd="0270
```

while still producing a mathord atom, would yield a really interesting position of the symbol (probably not what you would guess; try it and see). Cf. Rule 11 in Appendix G of *The TeXbook.*

21 Variable number of arguments

21.1 REMARKS

(Ed: Originally posted on 2002/09/13)

Back in the days when there existed an INFO-TeX mail list whose postings were automatically piped (by suitable arrangements) into `comp.text.tex`, I launched a thing called 'Around the Bend' with the following explanation:

> [Date: Thu 10 Oct 91]
>
> I would like to propose a regular department for INFO-TeX, called 'Around the Bend'. It will consist of macro-writing challenges on the level of the dangerous-bend exercises in the *TeXbook*, with interested parties invited to collaborate and/or compete to find the best solution. My motivation for doing this is partly selfish: to get more feedback from other macro writers about some of the interesting macro-writing problems that I run into.

There was never any attempt to establish a regular schedule for Around the Bend postings, I simply would do another one whenever I ran across an interesting problem, if I was able to spare some time to do so. The series is archived at `CTAN:pub/tex/info/aro-bend` for anyone who has an interest in looking at it. I also noticed that the exercises and answers are available in `comp.text.tex` archives through `groups.google.com`.

In response to a question on July 24, 2002 from Antoine Chambert-Loir (with apologies for the delay in answering):

> ...why did 'Around the Bend' stop? There were nice challenges proposed there.

I am tempted to say 'Well, actually they didn't stop, there was just an unusually large gap in the aperiodic schedule'.

But what I also wanted to say is that there are others quite as capable as I am of devising good Around the Bend exercises—I am thinking of a recent post by David Kastrup about a completely expandable string comparison macro—and it occurred to me it might be better to invite interested parties to sign up for an informal 'editorial board' to issue further exercises, so that other demands on my time do not have such a dampening effect on the rate of output. I don't have any desire to put restrictions on what goes out in continuation of the series apart from a (fairly crucial) one of striving for high quality and creativity. Send e-mail if you are interested, to the address below. There are only some obvious questions of coordination to address, such as trying (I think) to avoid two different people posting different exercises at the same time.

Turning now to the next exercise, prompted by a recent `comp.text.tex` question from David Reitter:

21.2 EXERCISE

Define a macro that takes a variable number of arguments. Do it in the best way possible. For the sake of concreteness, consider this somewhat contrived example as a test case that your solution should be able to handle, though possibly using a different syntax:

```
\printdate                        -> today's date in preferred form
\printdate[Tuesday]               -> Tuesday
```

```
\printdate[Tuesday][17]            -> Tuesday the 17th
\printdate[Tuesday][17][9]         -> Tuesday, September 17th
\printdate[Tuesday][17][9][2002]   -> and so on
\printdate[Tuesday][17][9][2002][Gregorian calendar] -> and so forth
```

The lines above illustrate six different ways of calling the `\printdate` macro. The macro should print something appropriate in each case, but the exact form of the output is a matter of taste, it need not follow exactly what I have given here.

Part of a good solution will be a good analysis of why one way might be better than another. The solution that I came up with is based on the question from David Reitter that originally inspired this exercise, thus it assumes the context is LaTeX and tries to solve the problem in a way that is natural for LaTeX.

A straightforward solution based on existing examples of multiple-option commands in the LaTeX kernel would qualify as natural, but definitely not elegant since that would require defining a separate macro to handle each stage of the multiple option scanning. Non-LaTeX solutions are also considered to be of interest.

I suggest posting your answers directly to comp.text.tex instead of mailing them to me (as was done in the past), though depending on how late you stayed up working on this entertaining exercise instead of writing your thesis or balancing your checkbook as you *ought* to have been doing, you might want to beware of posting in haste and wait until you have had some sleep and a chance to reread what you wrote, to avoid embarrassing oversights [. . . said he, speaking from experience].

Please e-mail a copy in addition (or instead, if you like) to the Around the Bend Editorial Board ... hmm, that gives me an idea . . . [pausing to consult the dictionary] make that the Supremely Honorable, Ingenious and, in Special Honor of Knuth, Around the Bend Editorial Board—whose size will not long remain one I dare say, especially after the establishment of this glamorous name—at `<seeacronym>@pobox.com`

21.3 ANSWERS

Solution 1 (David Kastrup)

(Ed: Originally posted on 2002/09/14)

```
\def\printdate{\count@\z@\toks@{}\printdate@a}
\def\printdate@a{\@ifnextchar[{\printdate@b}{\printdate@c}}
\def\printdate@b[#1]{\toks@\expandafter{\the\toks@{#1}}%
  \advance\count@\@ne\printdate@a}
\def\printdate@c{\csname printdate@@\romannumeral\count@
  \expandafter\endcsname\the\toks@}
```

You can now define the one-argument macro `\printdate@@i`, the 5-argument macro `\printdate@@v` and so on.

`\printdate@c` might also contain other stuff. For testing, we just define it as

```
\def\printdate@c{\message{\number\count@\space arguments: \the\toks@}}
```

This needs the LaTeX macro `\@ifnextchar`, of course.

If you want to have various defaults in sequence and just want to call `\printdate@@v`, you could write something like

```
\def\printdate@c{\let\gobble@x\relax\expandafter\newcommand
  \expandafter\gobble@x\expandafter[\number\count@]{}%
  \edef\next{{Tuesday}{17}{9}{2002}{Gregorian calendar}%
  \the\toks@}\expandafter\expandafter\expandafter
  \printdate@@v\expandafter\gobble@x\next}
```

Ok, this latter proposal is ugly. Better ideas?

End solution

Solution 2 (mine)

(Ed: Originally posted on 2002/09/20)

Define a macro that takes a variable number of arguments. and gave the following example application:

```
\printdate                        -> today's date in preferred form
\printdate[Tuesday]               -> Tuesday
\printdate[Tuesday][17]           -> Tuesday the 17th
\printdate[Tuesday][17][9]        -> Tuesday, September 17th
\printdate[Tuesday][17][9][2002] -> and so on
```

My solution (see below), written with LaTeX in mind, has the following characteristics:

- The kernel of the solution is not specific to a particular user-level command; for each user-level command, only two command-specific macros are needed: the top-level one invoked by the user, and the internal one that handles all the arguments. By contrast, the standard LaTeX method of handling multiple options requires a separate command-specific macro for each step of the argument scanning.
- The number of optional arguments is quasi-limited. The number of default values that you give in a command's definition becomes an upper limit on the number of arguments that will be scanned for. And if you supply twenty default values, the code that ends up handling them will have to be more than a simple TeX macro since macro arguments only go up to 9.
- Commands defined with this method can be nested, because the delimiters for the optional arguments are regular curly braces `{ }`, not square brackets [ ].

The choice of square brackets in LaTeX for optional arguments is OK for arguments whose values are suitably restricted, but when used for arguments that may contain arbitrary text—in particular, other commands with optional arguments—it becomes a pitfall that many users have fallen into over the years, and generally costing them an amount of lost time in inverse proportion to their understanding of catcodes. (I.e., its worst effects are on the kind of users that LaTeX was intended to serve in the first place.) The most common examples in practice are perhaps `\twocolumn[...]` and `\begin{thm}[...]`, but it could also happen in the optional arguments of `\section`, `\caption`, or `\cite`.

The chief argument against using braces for optional arguments came out coincidentally in another thread only a couple of days ago, as stated by Heiko Oberdiek on `comp.text.tex`

> How do you want to distinguish between a parameter and a group, both enclosed in "`{}`" Example:
>
> `\foo{bar}{\bfseries bla}`

But in practice it seems to me that this is not a significant drawback. Savvy users would normally use the `\textbf{...}` form anyway (I hope).

In fact the "`{\whatever ...}`" form (called a *declaration* in the LaTeX book) is, in a certain sense, quite unnatural for a linear language like TeX where the macro expansion works by simple left-to-right substitution. At least, if used at document level such a syntax makes it unnecessarily difficult to remap the functions involved and therefore is a stumbling block in many special applications. For example, it becomes feasible to add italic corrections automatically only when we use the `\emph{...}` form rather than the `{\em...}` form. (There is an `\aftergroup` trick that would sort of do the job but only by placing some assumptions on the usage that do not hold in the real world.)

```
\documentclass{article}
\usepackage{ifmtarg}
\makeatletter

% If \cmd{\MyCmd} is defined as
%    \VariableArgs{\MyCode ...}{{Default1}{Default2}}
% then
%     \MyCmd        -> \MyCode...{Default1}{Default2}
%     \MyCmd{aaa}   -> \MyCode...{aaa}{Default2}
%     \MyCmd{a}{bc} -> \MyCode...{a}{bc}
% In other words, \VariableArgs takes two arguments <code> and <defaults>
% and if the invocation via \MyCmd finds $n$ actual arguments, the first
% $n$ default values are replaced by the actual arguments.
%
% In principle the number of optional arguments is "whatever \MyCode is
% able to handle" but if the number of defaults is $d$ then scanning
% will stop as soon as $d$ arguments have been read, if not before.
% In practice things will begin to get unwieldy after a dozen or so
% arguments, because the process of scanning one more
% actual argument involves rescanning the whole list of arguments
% each time (actual arguments read previously plus any remaining defaults).

\newcommand{\VariableArgs}[2]{%
  \toks@{#1}%
  \@ifnextchar\bgroup{\AddArg #2{}@}{#1#2}}

\def\AddArg#1#2@#3{%
  \toks@\expandafter{\the\toks@{#3}}%
  \edef\RunIt{\the\toks@}%
  \@ifnextchar\bgroup{%
    \ifx @#2@%
      \begingroup
      \def\AddArg{\endgroup \expandafter\RunIt\@gobble}%
    \fi
    \AddArg #2@%
   }{%
     \RunIt #2%
   }%
  }

\newcommand{\printdate}{%
```

```
  % If zero args, use \today.
  \VariableArgs{\PrintDateFive}{{\today}{}{}{}{}}}

% This example is slightly more complicated than necessary because it
% behaves differently depending on the number of arguments.
\newcommand{\PrintDateFive}[5]{%
  % Always print #1, which might be \today (from the default value).
  #1%
  \@ifnotmtarg{#2#3#4#5}{%
    % If only #1 & #2 are given, use a slightly different form.
    \@ifmtarg{#3#4#5}{ the}{,}%
    % Args 2,3,4,5: Print each one if nonempty, but rearranging the
    % order slightly.
    \@ifnotmtarg{#3}{ \MonthName{#3}}%
    \@ifnotmtarg{#2}{ \OrdinalDay{#2}}%
    \@ifnotmtarg{#4}{, #4}%
    \@ifnotmtarg{#5}{ (#5)}%
   }}

\def\MonthName#1{%
  \ifcase 0#1 \number\month\or
    January\or February\or March\or April\or May\or June\or
    July\or August\or September\or October\or November\or December%
    \else Thirteen's Month\fi}

% If #2 is not a digit, use #1
\def\LastDigit#1#2{%
  \ifodd 0#21 \else #1\expandafter\@gobbletwo\fi\LastDigit #2}

\def\OrdinalDay#1{#1%
  \ifcase\LastDigit #1\space th\or st\or nd\or rd\else th\fi}

\begin{document}
\noindent Testing:
\begin{enumerate}\setcounter{enumi}{-1}
\item \printdate
\item \printdate{Tuesday}
\item \printdate{Tuesday}{17}
\item \printdate{Tuesday}{17}{9}
\item \printdate{Tuesday}{17}{9}{2002}
\item \printdate{Tuesday}{17}{9}{2002}{Gregorian calendar}
\end{enumerate}
\end{document}
```

End solution

Solution 3 (Donald Arseneau)

(Ed: Originally posted on 2002/09/24)

*** Exercise 21:
Define a macro that takes a variable number of arguments.

```
\printdate[Tuesday][17][9][2002][Gregorian calendar] -> and so forth
```

I did it (acually before MD posed the challenge) using `{ }`, not `[ ]`, and this answer does not match the challenge in other ways. But I haven't got around to working it in the last week or so.

Two features notably missing are: error checking for a bad number when specifying the number of arguments, and provision of default values for omitted arguments (they are all null here). (I also think I could make do with one fewer `\MultiArgCollect` macros.)

I think `{}` delimiters really are the 'best way' in regards to nesting macros. The one problem is confusion with non-explicit `{`, and so I handle the most common case of `\bgroup`.

```
\makeatletter
\let\MultiArgBgroup={

\def\MultiArg#1#2{\begingroup
  \let\bgroup\begingroup \let\egroup\endgroup
  \expandafter\MultiArgCollect\romannumeral\number#1001\delimiter{#2}}

\def\MultiArgCollect#1{\csname MultiArgCollect#1\endcsname}
\def\MultiArgCollectm#1\delimiter#2{%
  \@ifnextchar\MultiArgBgroup
    {\MultiArgCollectA#1\delimiter{#2}}%
    {\MultiArgCollect#1\delimiter{#2{}}}}

\def\MultiArgCollectA#1\delimiter#2#3{%
  \MultiArgCollect#1\delimiter{#2{#3}}}}

\def\MultiArgCollecti#1\delimiter#2{\endgroup#2}

\newcommand\DeclareMultiArgCommand[2]{\expandafter
  \Declare@MultiArg@ \csname MA\string_\string#1\endcsname{#1}{#2}}
\def\Declare@MultiArg@#1#2#3{%
  \DeclareRobustCommand{#2}{\MultiArg{#3}{#1}}
  \newcommand{#1}[#3]}

\DeclareMultiArgCommand {\printdate}{6}{...}
```

End solution

Index