Lists in \TeX's Mouth

Alan Jeffrey

1 Why lists?

Originally, I wanted lists in \TeX for a paper I was writing which contained a lot of facts.

\textbf{Fact i} Cows have four legs.

\textbf{Fact ii} People have two legs.

\textbf{Fact iii} Lots of facts in a row can be dull.

These are generated with commands like

\begin{verbatim}
\begin{fact}
  \Forward{Facyawn}
  Lots of facts in a row can be dull.
\end{fact}
\end{verbatim}

I can then refer to these facts by saying \By{Fac\-yawn, Fac\-cows, Fac\-people} to get \[i, ii, iii\]. And as if by magic, the facts come out sorted, rather than in the jumbled order I typed them. This is very useful, as I can reorganize my document to my heart's content, and not have to worry about getting my facts straight.

Originally I tried programming this sorting routine in \TeX's list macros, from Appendix D of \textit{The \TeXbook}, but I soon ran into trouble. The problem is that all the Appendix D macros work by assigning values to macros. For example:

\begin{verbatim}
\concatenate{\foo=\bar\baz}
\begin{fact}
  \Forward{Fac\-yawn}
  Lots of facts in a row can be dull.
\end{fact}
\end{verbatim}

I can then refer to these facts by saying \By{Fac\-yawn,Fac\-cows,Fac\-people} to get \[i, ii, iii\]. And as if by magic, the facts come out sorted, rather than in the jumbled order I typed them. This is very useful, as I can reorganize my document to my heart's content, and not have to worry about getting my facts straight.

Originally I tried programming this sorting routine in \TeX's list macros, from Appendix D of \textit{The \TeXbook}, but I soon ran into trouble. The problem is that all the Appendix D macros work by assigning values to macros. For example:

\begin{verbatim}
\concatenate{\foo=\bar\baz}
\begin{fact}
  \Forward{Fac\-yawn}
  Lots of facts in a row can be dull.
\end{fact}
\end{verbatim}

I can then refer to these facts by saying \By{Fac\-yawn,Fac\-cows,Fac\-people} to get \[i, ii, iii\]. And as if by magic, the facts come out sorted, rather than in the jumbled order I typed them. This is very useful, as I can reorganize my document to my heart's content, and not have to worry about getting my facts straight.

Originally I tried programming this sorting routine in \TeX's list macros, from Appendix D of \textit{The \TeXbook}, but I soon ran into trouble. The problem is that all the Appendix D macros work by assigning values to macros. For example:

\begin{verbatim}
\concatenate{\foo=\bar\baz}
\begin{fact}
  \Forward{Fac\-yawn}
  Lots of facts in a row can be dull.
\end{fact}
\end{verbatim}

I can then refer to these facts by saying \By{Fac\-yawn,Fac\-cows,Fac\-people} to get \[i, ii, iii\]. And as if by magic, the facts come out sorted, rather than in the jumbled order I typed them. This is very useful, as I can reorganize my document to my heart's content, and not have to worry about getting my facts straight.

Originally I tried programming this sorting routine in \TeX's list macros, from Appendix D of \textit{The \TeXbook}, but I soon ran into trouble. The problem is that all the Appendix D macros work by assigning values to macros. For example:

\begin{verbatim}
\concatenate{\foo=\bar\baz}
\begin{fact}
  \Forward{Fac\-yawn}
  Lots of facts in a row can be dull.
\end{fact}
\end{verbatim}

I can then refer to these facts by saying \By{Fac\-yawn,Fac\-cows,Fac\-people} to get \[i, ii, iii\]. And as if by magic, the facts come out sorted, rather than in the jumbled order I typed them. This is very useful, as I can reorganize my document to my heart's content, and not have to worry about getting my facts straight.
Once \texttt{Baz} is expanded, it will expect to be given a parameter, but when we are defining things, we can go around partially applying them all we like.

Here, I’m using = without formally defining it, which is rather naughty. If I say \( x = y \), this means “given enough parameters, \( x \) and \( y \) will eventually expand out to the same thing.” For example \( \texttt{Foo} = \texttt{Baz} \), because for any \( x \),

\[
\begin{align*}
\texttt{Foo} x &= \text{True} \\
&= \text{Or True } x \\
&= \text{Baz } x
\end{align*}
\]

Normally, functions have to respect equality which means that:

- if \( x = y \) then \( f x = f y \), and
- if \( x \) respects equality, then \( f x \) respects equality.

However, some \LaTeX{} control sequences don’t obey this. For example, \texttt{\string\Foo} and \texttt{\string\Baz} are different, even though \( \texttt{Foo} = \texttt{Baz} \). Hence \texttt{string} doesn’t respect equality. Unless otherwise stated, we won’t assume functions respect equality, although all the functions defined here do.

All of our functions have capital letters, so that their \LaTeX{} equivalents (\texttt{\Not}, \texttt{\Or} and so on) don’t clash with standard \LaTeX{} or \LaTeXe{} macros.

3.1 Identity

The simplest function is the identity function, called \texttt{Identity} funnily enough, which is defined:

\[
\texttt{Identity } x = x
\]

This, it must be admitted, is a pretty dull function. but it’s a useful basic combinator. It can be implemented in \LaTeX{} quite simply.

\begin{verbatim}
def\Identity#l{#l}
\end{verbatim}

The rules around this definition mean that it is actually part of \texttt{Lambda.sty} and not just another example.

3.2 Error

Whereas \texttt{Identity} does nothing in a fairly pleasant sort of way, \texttt{Error} does nothing in a particularly brutal and harsh fashion. Mathematically, \texttt{Error} is the function that destroys everything else in front of it. It is often written as \( \bot \).

\[
\texttt{Error } x = \text{Error}
\]

In practice, destroying the entire document when we hit one error is a bit much, so we’ll just print out an error message. The user can carry on past an error at their own risk, as the code will no longer be formally verified.

\begin{verbatim}
def\Error
  {\errmessage{Abandon verification all ye who enter here})
\end{verbatim}

Maybe this function ought to return a more useful error message …

3.3 First and Second

Two other basic functions are \texttt{First} and \texttt{Second}, both of which take in two arguments, and do the obvious thing. They are defined:

\[
\begin{align*}
\texttt{First} x y &= x \\
\texttt{Second} x y &= y
\end{align*}
\]

We could, in fact, define \texttt{Second} in terms of \texttt{Identity} and \texttt{First}. For any \( x \) and \( y \),

\[
\begin{align*}
\texttt{First Identity } x y &= \text{Identity } y \\
&= y \\
&= \text{Second } x y
\end{align*}
\]

So \texttt{First Identity} = \texttt{Second}. This means that anywhere in our \LaTeX{} code we have \texttt{\First\Identity} we could replace it by \texttt{\Second}. This is perhaps not the most astonishing \LaTeX{} fact known to humanity, but this sort of proof did enable more complex bits of \LaTeX{} to be verified before they were run.

The \LaTeX{} definitions of \texttt{\First} and \texttt{\Second} are pretty obvious.

\begin{verbatim}
def\First#1#2{#1}
def\Second#1#2{#2}
\end{verbatim}

Note that in \LaTeX{} \texttt{\First\foo\bar} expands out to \texttt{\foo without} expanding out \texttt{\bar}. This is very useful, as we can write macros that would take forever and a day to run if they expanded all their arguments, but which actually terminate quite quickly. This is called lazy evaluation by the functional programming community.

3.4 Compose

Given two functions \( f \) and \( g \) we would like to be able to \texttt{compose} them to produce a function that first applies \( g \) then applies \( f \). Normally, this is written as \( f \circ g \), but unfortunately \LaTeX{} doesn’t have infix functions, so we’ll have to write it \texttt{Compose \( f \circ g \)}.

\[
\texttt{Compose } f \, g \, x = f( g \, x )
\]

From this definition, we can deduce that \texttt{Compose} is associative:

\[
\texttt{Compose } ( \texttt{Compose } f \, g ) \, h = \texttt{Compose } f ( \texttt{Compose } g \, h )
\]
and Identity is the left unit of Compose:

\[ \text{Compose} \ Identity \ f = f \]

The reader may wonder why Identity is called a left unit even though it occurs on the right of the Compose — this is a side-effect of using prefix notations where infix is more normal. The infix version of this equation is:

\[ \text{Identity} \circ f = f \]

so Identity is indeed on the left of the composition.

Compose can be implemented in TeX as

\begin{verbatim}
\def\Compose#1#2#3{#1(#3){#2(#1)}}
\end{verbatim}

3.5 Twiddle

Yet another useful little function is Twiddle, which takes in a function and reverses the order that function takes its (first two) arguments.

\[ \text{Twiddle} f \ x \ y = f \ y \ x \]

Again, there aren't many immediate uses for such a function, but it'll come in handy later on. It satisfies the properties

\[ \text{Twiddle First} = \text{Second} \]
\[ \text{Twiddle Second} = \text{First} \]
\[ \text{Compose} \ \text{Twiddle} \ \text{Twiddle} = \text{Identity} \]

Its TeX equivalent is

\begin{verbatim}
\def\Twiddle#1#2#3{#1(#3){#2}}
\end{verbatim}

This function is called “twiddle” because it is sometimes written \( f \) (and \( \sim \) is pronounced “twiddle”). It also twiddles its arguments around, which is quite nice if your sense of humour runs to appalling puns.

4 Booleans

As we're trying to program a sorting routine, it would be nice to be able to define orderings on things, and to do this we need some representation of boolean variables. Unfortunately TeX doesn't have a type for booleans, so we'll have to invent our own. We'll implement a boolean as a function \( b \) of the form

\[ b \ x \ y = \begin{cases} x & \text{if } b \text{ is true} \\ y & \text{otherwise} \end{cases} \]

More formally, a boolean \( b \) is a function which respects equality, such that for all \( f \), \( g \) and \( z \):

\[ b \ f \ g \ z = b \ (f \ z) \ (g \ z) \]

and for all \( f \) and \( g \) which respect equality,

\[ b \ (f \ b) \ (g \ b) = b \ (f \ \text{First}) \ (g \ \text{Second}) \]

All the functions in this section satisfy these properties. Surprisingly enough, so does Error, which is quite useful, as it allows us to reason about booleans which “go wrong”.

4.1 True, False and Not

Since we are implementing booleans as functions, we already have the definitions of True, False and Not.

\[ \text{True} = \text{First} \]
\[ \text{False} = \text{Second} \]
\[ \text{Not} = \text{Twiddle} \]

So for free we get the following results:

\[ \text{Not True} = \text{False} \]
\[ \text{Not False} = \text{True} \]
\[ \text{Compose} \ \text{Not} \ \text{Not} = \text{Identity} \]

The TeX implementation is not exactly difficult:

\begin{verbatim}
\let\True=\First
\let\False=\Second
\let\Not=\Twiddle
\end{verbatim}

4.2 And and Or

The definitions of And and Or are:

\[ \text{And} \ a \ b = \begin{cases} \ b & \text{if } a \text{ is true} \\ \ False & \text{otherwise} \end{cases} \]
\[ \text{Or} \ a \ b = \begin{cases} \ True & \text{if } a \text{ is true} \\ \ b & \text{otherwise} \end{cases} \]

With our definition of what a boolean is, this is just the same as

\[ \text{And} \ a \ b = a \ b \ \text{False} \]
\[ \text{Or} \ a \ b = a \ True \ b \]

From these conditions, we can show that And is associative, and has left unit True and left zeros False and Error:

\[ \text{And} \ (\text{And} \ a \ b) \ c = \text{And} \ a \ (\text{And} \ b \ c) \]
\[ \text{And} \ True \ b = b \]
\[ \text{And} \ False \ b = \text{False} \]
\[ \text{And} \ \text{Error} \ b = \text{Error} \]

Or is associative, has left unit False and left zeros True and Error:

\[ \text{Or} \ (\text{Or} \ a \ b) \ c = \text{Or} \ a \ (\text{Or} \ b \ c) \]
\[ \text{Or} \ False \ b = b \]
\[ \text{Or} \ True \ b = \text{True} \]
\[ \text{Or} \ \text{Error} \ b = \text{Error} \]

De Morgan’s laws hold:

\[ \text{Not} \ (\text{And} \ a \ b) = \text{Or} \ (\text{Not} \ a) \ (\text{Not} \ b) \]
\[ \text{Not} \ (\text{Or} \ a \ b) = \text{And} \ (\text{Not} \ a) \ (\text{Not} \ b) \]

and And and Or left-distribute through one another:

\[ \text{Or} \ (\text{Or} \ a \ b) \ c = \text{Or} \ (\text{Or} \ a \ b) \ (\text{Or} \ a \ c) \]
\[ \text{And} \ a \ (\text{Or} \ b \ c) = \text{Or} \ (\text{And} \ a \ b) \ (\text{And} \ a \ c) \]
And and Or are not commutative, though. For example,

\[
\begin{align*}
\text{Or True Error} & = \text{True True Error} \\
& = \text{True} \\
\end{align*}
\]

but

\[
\begin{align*}
\text{Or Error True} & = \text{Error True True} \\
& = \text{Error} \\
\end{align*}
\]

This is actually quite useful since there are some booleans that need to return an error occasionally. If \(a\) is \(\text{True}\) when \(b\) is safe (i.e. doesn't become \(\text{Error}\)) and is \(\text{False}\) otherwise, we can say \(\text{Or a b}\) and know we're not going to get an error. This is handy for things like checking for division by zero, or trying to get the first element of an empty list.

Similarly, because of the possibility of \(\text{Error}\), \(\text{And}\) and \(\text{Or}\) don't right-distribute through each other, as

\[
\begin{align*}
\text{Or (And False Error) True} & \neq \text{And (Or False True) (Or Error True)} \\
\end{align*}
\]

As errors shouldn't crop up, this needn't worry us too much.

\[
\begin{align*}
\text{\textbackslash def\&\textbackslash And\&\#1\&\#2\&\textbackslash False}{} \\
\text{\textbackslash def\&\textbackslash Or\&\#1\&\#2\&\textbackslash True\{\#2\}}{} \\
\end{align*}
\]

4.3 Lift

Quite a lot of the time we won’t be dealing with booleans, but with predicates, which are just functions that return a boolean. For example, the predicate \(\text{Lessthan}\) is defined below so that \(\text{Lessthan}\ i\ j\) is true whenever \(i < j\). Given a predicate \(p\) we would like to be able to lift \(p\) to \(\text{Lift}\ p\), defined:

\[
\text{Lift } p f g x = p x f g x
\]

For example, \(\text{Lift (Lessthan 0)} f g \) takes in a number and applies \(f\) to it if it is positive and \(g\) to it otherwise. This is quite useful for defining functions.

\[
\begin{align*}
\text{\textbackslash def\\textbackslash Lift\&\#1\&\#2\&\#3\&\#4\&\textbackslash1\&\{\#4\}\&\textbackslash2\&\{\#2\}\&\{\#3\}\&\{\#4\}}{} \\
\end{align*}
\]

4.4 Lessthan and \TeXif

Finally, we would like to be able to use \TeX's built-in booleans as well as our own. For example, we would like a predicate \(\text{Lessthan}\) such that:

\[
\text{Lessthan } i\ j = \begin{cases} \\
\text{True} & \text{if } i < j \\
\text{False} & \text{if } i \geq j \\
\text{Error} & \text{otherwise}
\end{cases}
\]

The \(\text{Error}\) condition happens if we try applying \(\text{Lessthan}\) to something that isn't a number—

\(\text{Lessthan True False is Error}.\)\(^1\) This is fine as a mathematical definition, but how will we implement it? If we assume we have a macro \(\text{\TeXif}\), which converts \TeX if-statements into booleans, we could just define:

\[
\begin{align*}
\text{\textbackslash def\&\textbackslash Lessthan\&\#1\&\#2\&\textbackslash\{}\textbackslash\{\textbackslash ifnum\&\#1\&\#2\&\}\textbackslash\{}\textbackslash\{\textbackslash\}}{} \\
\end{align*}
\]

So the question is just how to define \TeXif. Unfortunately, the “obvious” code does not work:

\[
\begin{align*}
\text{\textbackslash def\&\textbackslash Lessthan\&\#1\&\#2\&\#3\&\textbackslash1\&\#2\&\textbackslash\{\#1\&\#2\&\textbackslash\}}\textbackslash\textbackslash fi}{} \\
\end{align*}
\]

For example, \(\text{\TeXif\&\textbackslash iftrue\&\textbackslash True\&\textbackslash True}\) doesn't expand out to \(\text{\True}\). Instead, it expands as:

\[
\begin{align*}
\text{\TeXif\&\textbackslash iftrue\&\textbackslash True\&\textbackslash True} = \\text{\iftrue\&\textbackslash True\&\textbackslash else\&\textbackslash True\&\textbackslash fi} = \\text{\True\&\textbackslash else\&\textbackslash fi} = \\text{\else\&\textbackslash fi} = \text{\else} \\
\end{align*}
\]

Another common \TeX technique is to use a macro \(\text{\next}\) to be the expansion text:

\[
\begin{align*}
\text{\textbackslash def\&\textbackslash Lessthan\&\#1\&\#2\&\#3\&\textbackslash%}{} \\
\text{\textbackslash def\&\textbackslash next\&\#2\&\textbackslash%}{} \\
\end{align*}
\]

However, this uses \TeX's stomach to do the \texttt{def}, and we are trying to do this using only the mouth. One (slightly tricky) solution is to use pattern-matching to gobble up the offending \texttt{else} and/or \texttt{fi}.

\[
\begin{align*}
\text{\textbackslash def\&\textbackslash gobblefalse\&\textbackslash else\&\textbackslash gobbletrue\&\textbackslash fi\&\#1\&\#2%}{} \\
\text{\textbackslash def\&\textbackslash gobbletrue\&\textbackslash fi\&\#1\&\#2%}{} \\
\text{\textbackslash def\&\textbackslash Texif\&\#1%}{} \\
\text{\textbackslash def\&\textbackslash next\&\#2\&\textbackslash%}{} \\
\text{\textbackslash def\&\textbackslash Text\&\textbackslash iftrue\&\textbackslash True\&\textbackslash True\&\textbackslash else\&\textbackslash True\&\textbackslash fi}{} \\
\text{\textbackslash def\&\textbackslash Text\&\textbackslash if\&\textbackslash True\&\textbackslash True\&\textbackslash else\&\textbackslash True\&\textbackslash fi}{} \\
\text{\textbackslash def\&\textbackslash Text\&\textbackslash if\&\textbackslash True\&\textbackslash True\&\textbackslash else\&\textbackslash True\&\textbackslash fi}{} \\
\text{\textbackslash def\&\textbackslash Text\&\textbackslash iftrue\&\textbackslash True\&\textbackslash True\&\textbackslash else\&\textbackslash True\&\textbackslash fi}{} \\
\text{\textbackslash def\&\textbackslash Text\&\textbackslash if\&\textbackslash True\&\textbackslash True\&\textbackslash else\&\textbackslash True\&\textbackslash fi}{} \\
\end{align*}
\]

So if the \TeX if-statement is true, \texttt{gobblefalse} gobbles up the false-text, otherwise \texttt{gobbletrue} gobbles up the true-text. For example,

\[
\begin{align*}
\text{\TeXif\&\textbackslash iftrue\&\textbackslash True\&\textbackslash True} = \\text{\iftrue\&\textbackslash True\&\textbackslash else\&\textbackslash True\&\textbackslash fi} = \\text{\True\&\textbackslash else\&\textbackslash fi} = \\text{\else\&\textbackslash fi} = \\text{\else} = \\text{\else} = \\text{\if\&\textbackslash True\&\textbackslash True\&\textbackslash else\&\textbackslash True\&\textbackslash fi} = \\text{\if\&\textbackslash True\&\textbackslash True\&\textbackslash else\&\textbackslash True\&\textbackslash fi} = \\text{\if\&\textbackslash True\&\textbackslash True\&\textbackslash else\&\textbackslash True\&\textbackslash fi} = \\text{\if\&\textbackslash True\&\textbackslash True\&\textbackslash else\&\textbackslash True\&\textbackslash fi} = \\text{\if\&\textbackslash True\&\textbackslash True\&\textbackslash else\&\textbackslash True\&\textbackslash fi} = \text{\iftrue} \\
\end{align*}
\]

Phew. And so we have booleans.

\(^1\) Actually, that's a little white lie—trying to persuade \TeX to do run-time type checking isn't much fun. So the \TeX implementation of this is actually a refinement where the \(\text{Error}\) condition has been replaced by whatever it is \TeX does if you try doing \texttt{ifnum } x < y \text{ when } x \text{ and } y \text{ aren't numbers.}
5 Lists

A list is a (possibly infinite) sequence of values. For example, the list [1, 2, 3] contains three numbers, the list [ ] contains none, and the list [1, 2, 3, ...] contains infinitely many. A list is either empty (written [ ] ) or is comprised of a head x and a tail xs (in which case it's written x : xs). For example, 1 : 2 : 3 : [] is [1, 2, 3].

In a similar fashion to the implementation of booleans, a list xs is implemented as a function of the form

\[ xs : e = \begin{cases} e & \text{if } xs \text{ is empty} \\ f y y s & \text{if } xs \text{ has head } y \text{ and tail } y s \end{cases} \]

Again, we are implementing a datatype as a function, a quite powerful trick, just not one usually seen in TeX. We will assume that whenever a list x : xs is applied to f and e, f x e respects equality. This allows us to assume that if xs = ys then x : xs = x : ys, which is handy.

5.1 Nil, Cons, Stream and Singleton

The simplest list is Nil, the empty list which we have been writing [ ].

\[ \text{Nil} = \text{Second} \]

The other possible list is Cons x xs, which has head x and tail xs.

\[ \text{Cons } x \text{ } x s : e = f x s \]

Every list can be constructed using these functions. The list [1, 2, 3] is Cons 1 (Cons 2 (Cons 3 Nil)), and the list [a, a, a, ...] is Stream a where Stream is defined:

\[ \text{Stream } a = \text{Cons } a \text{ (Stream } a) \]

There's even at least one application for infinite lists, as we'll see in Section 7.

The singleton list [a] is Singleton a, defined as:

\[ \text{Singleton } a = \text{Cons } a \text{ Nil} \]

These all have straightforward TeX definitions.

\begin{verbatim}
\let\Nil=\Second
\def\Cons#1#2#3#4{%3(#1){#2}}
\def\Stream#1{%\Cons{#1}{\{\Stream{#1}\}}} \def\Singleton#1{%\Cons{#1}{\Nil}}
\end{verbatim}

5.2 Head and Tail

So, we can construct any list we like, but we still can't get any information out of it. To begin with, we'd like to be able to get the head and tail of a list.

\[ \text{Head } x s = x s \text{ First Error} \]
\[ \text{Tail } x s = x s \text{ Second Error} \]

For example, the tail of x : xs is

\[ \text{Tail } (\text{Cons } x \text{ xs)} \]
\[ = \text{Cons } x \text{ xs Second Error} \]
\[ = \text{Second } x \text{ xs} \]
\[ = x s \]

The tail of [ ] is, as one would expect,

\[ \text{Tail } \text{Nil} \]
\[ = \text{Nil Second Error} \]
\[ = \text{Error} \]

And the head of Stream a is

\[ \text{Head } (\text{Stream } a) \]
\[ = \text{Stream } a \text{ First Error} \]
\[ = \text{Cons } a \text{ (Stream } a) \text{ First Error} \]
\[ = \text{First } a \text{ (Stream } a) \]
\[ = a \]

So we can get the head of an infinite list in finite time. This is fortunate, as otherwise there wouldn't be much point in allowing infinite objects.

\begin{verbatim}
\def\Head#1{#1\text{First}\text{Error}} \def\Tail#1{#1\text{Second}\text{Error}}
\end{verbatim}

5.3 Foldl and Foldr

Using Head and Tail we can get at the beginning of any non-empty list, but in general we need more information than that. Rather than write a whole bunch of recursive functions on lists, I'll implement two fairly general functions, with which we can implement (almost) everything else.

Foldl and Foldr both take in functions and apply them recursively to a list. Foldl starts at the left of the list, and Foldr starts at the right. For example,

\[ \text{Foldl } f e [1, 2, 3] = f (f (f e 1) 2) 3 \]
\[ \text{Foldr } f e [1, 2, 3] = f 1 (f 2 (f 3 e)) \]

These functions will be used a lot later on. Foldl can be defined:

\[ \text{Foldl } f e x s = x s (\text{Foldl} f e x) \]
\[ \text{Foldl } f e x s = \text{Foldl } f (f e x) x s \]

So Foldl f e [ ] is

\[ \text{Foldl } f e \text{ Nil} \]
\[ = \text{Nil} (\text{Foldl} f e) e \]
\[ = e \]

And Foldl f e (x : xs) is

\[ \text{Foldl } f e \text{ (Cons } x \text{ xs)} \]
\[ = \text{Cons } x \text{ xs (Foldl} f e) e \]
\[ = \text{Foldl} f e x s \]
\[ = \text{Foldl } f (f e x) x s \]
For example, \( \text{Foldl} \ f \ e \ [1, 2, 3] \) is

\[
\text{Foldl} \ f \ e \ [1, 2, 3] = \text{Foldl} \ f \ (f \ e \ 1) \ [2, 3] = \text{Foldl} \ f \ (f \ (f \ e \ 1) \ 2) \ [3] = \text{Foldl} \ f \ (f \ (f \ (f \ e \ 1) \ 2) \ 3) \ [1] = f (f (f e \ 1) \ 2) \ 3
\]

as promised. Similarly, we can define \( \text{Foldr} \) as

\[
\text{Foldr} \ f \ e \ xs = \text{xs} \ (\text{Foldr}' \ f \ e) \ e \\
\text{Foldr}' \ f \ e \ xs = f \ e (\text{Foldr} \ f \ e \ xs)
\]

For \( \text{Foldr} \) to respect equality, \( f \ x \) should respect equality.

When we do the unfolding, we discover that

\[
\text{Foldr} \ f \ e \ [\ ] = e \\
\text{Foldr} \ f \ e \ (x : xs) = f \ e (\text{Foldr} \ f \ e \ xs)
\]

\( \text{Foldr} \) tends to be more efficient than \( \text{Foldl} \), because \( \text{Foldl} \) has to run along the entire list before it can start applying \( f \), whereas \( \text{Foldr} \) can apply \( f \) straight away. If \( f \) is a lazy function, this can make quite a difference. \( \text{Foldl} \) on infinite lists, anyone?

5.4 Cat

Given two lists, we would like to be able to stick them together, which is what \( \text{Cat} \) [short for “concatenate”] does. For example, \( \text{Cat} \ [1, 2] [3, 4] \) is \( [1, 2, 3, 4] \). It can be defined using \( \text{Foldr} \):

\[
\text{Cat} \ xs \ ys = \text{Foldr} \ Cons \ ys \ xs
\]

So

\[
\text{Cat} \ [1, 2] [3, 4] = \text{Foldr} \ Cons [3, 4] \ [1, 2] = \text{Cons} \ 1 \ (\text{Foldr} \ Cons \ [3, 4] \ [2]) = \text{Cons} \ 1 \ (\text{Cons} \ 2 \ (\text{Foldr} \ Cons \ [3, 4] \ [1])) = \text{Cons} \ 1 \ (\text{Cons} \ 2 \ [3, 4]) = [1, 2, 3, 4]
\]

The \TeX\ code for \( \text{\textbackslash Cat} \) is suspiciously similar to its mathematical definition.

5.5 Reverse

We can reverse any list with the function \( \text{Reverse} \), defined using \( \text{Foldl} \):

\[
\text{Reverse} = \text{Foldl} \ (\text{Twiddle} \ \text{Cons}) \ Nil
\]

For example, \( \text{Reverse} \ [1, 2, 3] \) can be calculated:

\[
\text{Reverse} \ [1, 2, 3] = \text{Foldl} \ (\text{Twiddle} \ \text{Cons}) \ Nil \ [1, 2, 3] = \text{Twiddle} \ \text{Cons} \ (\text{Twiddle} \ \text{Cons} \ Nil \ [1, 2, 3]) = \text{Cons} \ 3 \ (\text{Cons} \ 2 \ (\text{Cons} \ 1 \ Nil)) = [3, 2, 1]
\]

The \TeX\ code for \( \text{\textbackslash Reverse} \) doesn't even take in any parameters.

5.6 All, Some and Isempty

Given a predicate \( p \), we can find out if all the elements of a list satisfy \( p \) with \( \text{All} \ p \). Similarly we can find if something in the list satisfies \( p \) with \( \text{Some} \ p \). For example,

\[
\text{All} \ (\text{Less} \ \text{than} \ 1) \ [1, 2, 3] = \text{False} \\
\text{Some} \ (\text{Less} \ \text{than} \ 1) \ [1, 2, 3] = \text{True}
\]

These can be defined

\[
\text{All} \ p = \text{Foldr} \ (\text{Compose} \ \text{And} \ p) \ True \\
\text{Some} \ p = \text{Foldr} \ (\text{Compose} \ \text{Or} \ p) \ False
\]

For example, \( \text{Isempty} \) can be defined

\[
\text{Isempty} = \text{All} \ (\text{First} \ False)
\]

This is probably not the most efficient check in the world, but we hardly ever need it — \( \text{Foldl} \) or \( \text{Foldr} \) will normally do the job.

5.7 Filter

\( \text{Filter} \) takes a predicate \( p \) and a list \( xs \), and returns a list containing only those elements of \( xs \) that satisfy \( p \). For example,

\[
\text{Filter} \ (\text{Less} \ \text{than} \ 1) \ [1, 2, 3] = [2, 3]
\]

\( \text{Filter} \) can be defined as a \( \text{Foldr} \):

\[
\text{Filter} \ p = \text{Foldr} \ (\text{Lift} \ p \ \text{Cons} \ \text{Second}) \ Nil
\]

Another easy bit of \TeX\:

\[
\text{\textbackslash Filter} \ #1 = \{\text{Foldr} \ {\text{\textbackslash Lift} \ #1} \ {\text{Cons} \ \text{Second}} \ \text{\textbackslash Nil}}
\]
5.8 Map

Map takes a function \( f \) and a list \( xs \) and applies \( f \) to every element of \( xs \). For example,
\[
Map f [1,2,3] = [f 1, f 2, f 3]
\]
This is another job for Foldr.

\[
Map f = Foldr (Compose Cons f) \Nil
\]
We shall see Map used later on, to convert from a list of names such as [Fac-yawn, Fac-cows], to a list of labels such as [i, iii].

\[
\def\Map#l\{\Foldr{\Compose\Cons{\#1}}\Nil}
\]

5.9 Insert

The only function we need which isn't easily defined as a reduction is Insert, which inserts an element into a sorted list. For example,

\[
Insert Lessthan 3 [1,2,4,5] = [1,2,3,4,5]
\]
Insert takes in an ordering as its first parameter, so we're not stuck with one particular order. It is defined directly in terms of the definition of lists.

\[
Insert o x xs = xs (Insert' o x) (Singleton x)
\]
\[
Insert' o x y ys = o x y
\]
\[
(\text{Cons} x (\text{Cons} y ys))
\]
\[
(\text{Cons} y (\text{Insert} o x ys))
\]

We can then define the function all this has been leading up to, Insertsort which takes an ordering and a list, and insert-sorts the list according to the ordering. For example,

\[
Insertsort Lessthan [2,3,1,2] = [1,2,3,4]
\]
We can implement this as a fold:

\[
Insertsort o = Foldr (Insert o) \Nil
\]
And so we've got sorted lists.

\[
\def\Insert#1\#2#3\%\
{\#3{\Insert\#1\{\#2}\}{\Singleton\{\#2}\}}
\]
\[
\def\Insert#1\#2#3#4\%
{\#1{\#2}\{\#3}\%
{\Cons\{\#2\}{\Cons\{\#3\}{\Cons\{\#4\}{\Singleton\{\#4\}}}}
{\Cons\{\#3\}{\Insert\{\#1\}{\Singleton\{\#2\}{\Cons\{\#4\}}}}}
\}
\def\Insertsort#1\{\Foldr{\Insert\#1}\Nil}
\]

Interestingly, as we have implemented unbounded lists in \( \text{\TeX} \)'s mouth, this means we can implement a Turing Machine. So, if you believe the Church-Turing thesis, \( \text{\TeX} \)'s mouth is as powerful as any computer anywhere. Isn't that good to know?

6 Sorting reference lists

So, these are the macros I've got to play with — how do we apply them to sorting lists of references? Well, I'm using \LaTeX, which keeps the current reference in a macro called \( @currentlabel \), which is 6 at the moment, as this is Section 6. So I just need to store the value of \( @currentlabel \) somehow.

Fortunately, I'm only ever going to be making references to facts earlier on in the document, in order to make sure I'm not proving any results in terms of themselves. So I don't need to play around with auxiliary files, and can just do everything in terms of macros.

6.1 Number and Label

Each label in the document is given a unique number, in the order the labels were put down. So the number of Fac-cows is \( \Number\{\text{Fac-cows}\} \), which expands out to 1, the number of Fac-people is 2, and so on.

Each number has an associated label with it. For example, the first label is \( \Label\{1\} \), which is i, the second label is ii and so on. So to find the label for Fac-cows, we say \( \Label\{\Number\{\text{Fac-cows}\}\} \) which expands out to i.

These numbers and labels are kept track of in macros. For example, the number of Fac-cows is kept in \( \Number\{\text{Fac-cows}\} \). Similarly, the first label is kept in \( \Label\{1\} \). As these macros have dashes in their names, they aren't likely to be used already.

So the \( \text{\TeX} \) code for \( \Number \) and \( \Label \) is pretty simple.

\[
\def\Number#1\{\csname Number-#1\endcsname\}
\def\Label#1\{\csname Label-#1\endcsname\}
\]

6.2 Lastnum and Forward

The number of the most recent label is kept in \( \Lastnum \).

\[\newcount\Lastnum\]
To put down a label Foo, I type \( \Forward\{\text{Foo}\} \). This increments the counter \( \Lastnum \), and \( \xdef\Number\{\text{Foo}\} \) to be the value of \( \Lastnum \), which is now 4. So \( \Number\{\text{Foo}\} \) now expands to 4. Similarly, it \( \xdef\Label\{\text{Foo}\} \) to be \( @currentlabel \), which is currently 6.2. So \( \Label\{\Number\{\text{Foo}\}\} \) now expands to 6.2.

\[
\def\Forward#1\%
{\global\advance\Lastnum\by\1\}
\csnameafter\xdef\Number-#1\%
{\the\Lastnum\%
\csnameafter\xdef\Label-\the\Lastnum\%
{\@currentlabel\}}
\]
This uses \csname after \foocbar), which expands out to \foocbar.
\def\csnameafter{\expandafter}\csnameafter}{\expandafter\csnameafter}{\csnameafter}{\\endcsname}

6.3 Listize, Unlistize and Show
At the moment, lists have to be built up using \Cons and \Nil, which is rather annoying. Similarly, we can't actually do anything with a list once we've built it. We'd like some way of converting lists in the form \([a, b, c]\) to and from the form \([a, b, c]\).
This is done with \Listize and \Unlistize. So \Listize\([a, b, c]\) expands to
\Cons\([a]\}\Cons\([b]\}\Cons\([c]\}\Nil}}
Similarly, \Unlistize takes the list \([a, b, c]\) and expands out to \([a, b, c]\). \Unlistize is done with a Foldr.
\def\Listize\[#1\]%%\{\Listize\[#1, \relax]\}
\def\Unlistize\[#1, \#2\]%%\{\TeXif{\ifx\relax\#2}%%\{\Singleton\[#1]\}\{\Cons\[#1]\{\Listize\[#2]\}\}

This only works for nonempty lists—\Listize[] produces the singleton list \Singleton[]. It also uses \relax as its end-of-list character, so lists with \relax in them have to be done by hand. You can't win them all. So
\$\Unlistize\{\Listize[a, b, c]\}\$
produces \([a, b, c]\). This is such a common construction that I've defined a macro \Show such that
\Show\fooc\{\Listize[a, b, c]\}\}
For example, the equation
\[
\text{Filter } (\text{Less than } 1) \{1, 2, 3\} = \{2, 3\}
\]
was generated with
\begin{eqnarray*}
\text{Filter } (\text{Less than } 1) \{1, 2, 3\}
&\equiv& \text{Show } \text{Filter } \{\text{Less than } 1\} \{1, 2, 3\}
\end{eqnarray*}
Many of the examples in this article were typeset this way.

6.4 By
Given these macros, we can now sort any list of references with \Bylist, defined
\[
\text{Bylist } xs = \text{Map } \text{Label}
\]
\[
(\text{Insertsort } \text{Less than})
\]
This takes in a list of label names like Fac-yawn, converts it into a list of numbers with \Map Number, sorts the resulting list with \Insertsort, and finally converts all the numbers into labels like iii with \Map Label. For example,
\[
\text{Bylist } \{\text{Fac-yawn, Fac-cows}\}
\]
\[
= \text{Map } \text{Label}
\]
\[
(\text{Insertsort } \text{Less than})
\]
\[
(\text{Map Number } (\text{Fac-yawn, Fac-cows}))
\]
\[
= \text{Map } \text{Label}
\]
\[
(\text{Insertsort } \text{Less than } [3, 1])
\]
\[
= \text{Map } [1, 3]
\]
\[
= [i, iii]
\]
The \TeX code for this is
\def\Bylist\[#1\]%%\{\Map\Label
\def\Map\Label\{\Insertsort\Lessthan
\def\Lessthan\{\Map\Number
\}
So we can now stick all this together, and define the macro \By that prints out lists of references. It is
\def\By\{\Show\Bylist\}
So \By[Fac-yawn,Fac-cows] is \([i, iii]\). Which is quite nice.

7 Other applications
Is all this worth it? Well, I've managed to get my lists of facts in order, but that's not the world's most astonishing application. There are other things that these lists are useful for, though.
For example, Damian Cugley has a macro package under development for laying out magazines. MAG\TeX's output routine needs to be quite smart, as magazines often have gaps where illustrations or photographs are going to live. In general, each block of text needs to be output in a different fashion from every other block of text. This will be handled by keeping an infinite list of output routines. Each time a box is cut off the scroll to be output, the head of the list is chopped off and is used as the output routine for that box. That way, quite complex page shapes can be built up.
Mainly, though, these macros were written just as a challenge. I learned quite a lot about T\TeX and needed some T\TeX techniques I'd never seen before. It was also quite pleasing to see that T\TeX code can be formally verified, albeit in a rather noddy way. Without some sort of abstract view of lists, these T\TeX macros could not have been written.

8 Acknowledgements

Thanks to Jeremy Gibbons for letting me bounce ideas off him and spotting the duff ones, to Damian Cugley for saying "Do you really think T\TeX is meant to do this?", and to the Problem Solving Club for hearing me out. This work was sponsored by the Science and Engineering Research Council and Hewlett Packard.

\begin{quote}
\begin{verbatim}
Alan Jeffrey
Programming Research Group
Oxford University
11 Keble Road
Oxford OX1 3QD
Alan.Jeffrey@uk.ac.oxford.prg
\end{verbatim}
\end{quote}

A Nestable Verbatim Mode

Philip Taylor

A few months ago, Sebastian Rahtz asked me if I could make some changes to the verbatim code which he was currently using, and sent me the source. I found it so opaque that I decided to write my own, and the following evolved over a period of a couple of weeks. I would like to acknowledge my debt to Sebastian, and also to Chris Rowley, without whose helpful comments and criticism the code could never have evolved. Of course, the code is now ten times as opaque as that originally used by Sebastian, but at least I understand it (on a good day, when the moon is the seventh house, and Jupiter \halips with Mars).

The idea is as follows: having said
\verbatim
at the beginning of one's document, one invokes verbatim mode by
\verbatim (char)
What follows can then contain any character, with the single exception of (char), and all such text will be copied verbatim, with leading spaces retained but invisible, and all embedded spaces retained and shewn. If (char) is encountered, T\TeX enters a new inner group (the verbatim environment is itself a group), within which the preceding meaning (i.e. \catcode) of all characters is reinstated. This new inner group continues typesetting in the normal (non-verbatim) manner until a further (char) is encountered, whereupon it reverts to verbatim mode: the inner 'normal' mode can itself be interrupted by a further
\verbatim (char)
where (char) can be the same or a different escape character. There is no theoretical limit on the level of nesting, but T\TeX implementations will invariably run out of space (usually save-stack space) if too many levels are attempted.

To end verbatim mode, one enters inner 'normal' mode through the escape character and then says \mitabrev. Note that this is not a reserved string, but simply a macro which expands to \endgroup \endgroup; any other name can be chosen if one finds "\mitabrev" unappealing. Thus, at the outermost level, the call and end to \verbatim look like:
\verbatim (char)
\mitabrev
\verbatim
Finally, a mechanism is provided for listing arbitrary files in verbatim mode. If, while in inner 'normal' mode, one says \AfterGroup (\verbatim (any balanced text)) \verbatim (note the case of \AfterGroup), the (balanced text) will be re-inserted with its original \catcodes immediately after the closing (char) which terminates inner 'normal' mode. Thus it will not itself be listed \verbatim, but will be elaborated according to T\TeX's normal conventions. Thus if one says
\verbatim (\input (filename))
the contents of the file will be listed in verbatim mode. For example, to list this file itself, one can say
\verbatim
\verbatim
There remains an anomaly at present: "\" cannot form the escape-character as it will automatically form a (control sequence) with the following character(s) when called with
\verbatim
I will endeavour to rectify this deficiency in a future release.