Galleys, Space, and Automata

Jonathan Fine
203 Coldhams Lane
Cambridge
CB1 3HY
England
J.Fine@pmms.cam.ac.uk

Abstract

The thread which runs through this article is the concept of a Finite State Automaton (FSA). It is introduced as a solution to the problem of how to specify and code the amount of vertical space to be left between the various type of item which can be placed upon the page. Several methods of coding FSA are described and compared. One solution is to store the transition table and other data in the ligature table of a custom font. The best use of this method requires sofware tools which cannot readily be programmed in TEX, and also some extensions to TEX. These are discussed, and the article concludes.

Introduction

TEX forms pages by adding paragraphs and other vertical material such as skips, penalties, titles and displayed boxes to a galley, which is broken and presented to the output routine once sufficient material has been gathered. Hand setting of movable type proceeds in the same way. This article is focussed on how TEX should be instructed to insert appropriate vertical space and so forth between the paragraphs and other textual items. The proper use of space is essential to good typography.

Here are some spacing rules. Add extra space around a displayed quotation. Add extra space before a new section. Just after a section title is a bad place to break the page, so insert a penalty. Just before a subsection is a good place to break, so insert a reward — i.e., a negative penalty.

These rules do not tell us what should be done if a displayed quotation is followed by a new section. Should one use both extra spaces, or just one, or something else? It is common practice to specify 'before' and 'after' space for each element, and to take the larger of the applicable values when one element follows another. Of course, there will be exceptions. Similar considerations apply to lists. When a section title is immediately followed by a subsection, is this a good place to break the page, or a bad place? It is important to get these details right, for they can make or break the document (at the wrong place).

We shall assume that when a paragraph or other item X is added to the galley, the vertical space that should be added to the galley before X is placed upon it depends only on the sort of item that X is, together with the sort of item last placed on the galley. Thus, vertical space rules belong not to the vertical matter type itself, perhaps in *before* and *after* variants, but to combinations of vertical matter types, applying when type W is followed by type X. One can think of this as a relational approach.

If there are five types of paragraph, A to E, then there are 25 different possibilities AA, AB, AC, AD, AE, BA, BB, BC, BD, BE, CA, CB, CC, CD, CE, DA, DB, DC, DD, DE, EA, EB, EC, ED, EE, and for each of these a rule is required. Ten types of paragraph will give 100 possibilities. A large part of this article is devoted the problem of how one might specify and implement such a large collection of rules.

When TEX is typesetting the document, it needs to record the type of the last item on the galley, which we shall call the *state* of the galley. Each time an item is present for addition to the galley, we have an *event*, which may then change the state of the galley. Also, each event will cause a possibly empty *action* to take place. The action chosen will depend on the current state and on the event. In this example it is the addition of vertical space. It may be more compliciated. For example, a format may allow a section to begin on the current page if it is at most half full.

Finite State Automata

Here is a simpler example of a machine that has states, events, and actions. It is a coin-operated

turnstile, such as is used by the New York subway system. One approaches the turnstile, places a coin in the slot, pushes against the bar, which then gives and allows admission. The next person will have to use another coin to gain admission. Without a coin, the bar will not move.

Here is a more formal description of the operation of the turnstile.

```
\FSA \turnstile \barred
{
  * \barred
    @ \push  \barred
    @ \coin  \open

  * \open
    @ \push  \barred      \admit_one
    @ \coin  \open
}
```

The first line tells us that `\turnstile` is a *Finite State Automaton*, whose initial state is `\barred`. (This is something we forgot to specify). The line

```
@ \push  \barred
```

beneath `* \barred` tells us that should the `\push` occur event when the `\turnstile` is `\barred` then the state is unchanged, and the action is empty. The line

```
@ \coin  \open
```

says that placing a coin in a `\barred` turnstile changes its state to `\open` and again has no action.

The interesting line is

```
@ \push  \barred      \admit_one
```

beneath `* \open`, which tells us that when the turnstile is in the `\open` state (as a result of the `\coin` event), a `\push` will result in the `\admit_one` action. In specifying the operation of the turnstile, we forget to say what should happen when a coin is placed in a turnstile that is already open. In the above description, nothing happens.

The operation of the `\turnstile` has been described by the *transition lines*. These begin with an `@`, and are followed by an event, and then the new state, and then, optionally, the action for the event. To find the transition line for a given existing state X and event Y, first look for `*` followed by X in the description. This is the *label* for the state X. Now read on until you reach `@` followed by Y. This is the transition line to be followed.

The most general FSA with n states and m events will require $n \times m$ transition lines. (The acronym FSA stands for Finite State Automaton or Automata as is appropriate). In real applications this number can be reduced, by careful use of two further properties of the `\FSA` construction. The first is that the one or more labels for other events can intervene between the label for the current state, and the line for the message. In our case, the `\turnstile` will be `\open` after the `\coin` event, whatever the current state. Here is a description of `\turnstile` with only 3 transition lines.

```
\FSA \turnstile \barred
{
  * \barred
    @ \push  \barred

  * \open
    @ \push  \barred      \admit_one
    @ \coin  \open
}
```

The second property is a little more complicated. Within the rules for `\turnstile`, `#1` can be used to stand for the current state, and `#2` for the event. The FSA `\echo` described here

```
\FSA \echo \default
{
  * #1
    @ #2 #1  \message
    {
      state=\string#1,
      event=\string#2
    }
}
```

starts in the `\default` state. Whatever the event, the state is unchanged. The action is to `\message` the current state, and the event that occurred.

Here is a more sophisticated version of `\echo`. It is to have two states, `\on` and `\off`. The event `\on` is to turn change state to `\on`, the event `\off` to change state to `\off`. All other events are to leave the state unchanged, and if the state is `\on` there should be a state and event `\message` as before.

```
1. \FSA \echo \on
2. {
3.   * \off
4.     @ \on    \on
5.     @ #2     \off
6.
7.   * \on
8.     @ \off   \off
9.     @ #2     \on    \message
10.    {
11.      state=\string#1,
12.      event=\string#2
13.    }
14. }
```

Lines 3-5 can be read as follows. If the state is `\off` and the event is `\on` then the state is changed to on, otherwise do nothing. Lines 7-10 say that if the state is `\on` and event is `\off`, then state is changed to `\off`, otherwise the state is `\on` and the `\message` is executed.

The order in which the labels and transition lines appear is very important, and may require

careful thought to get the best formulation of the operation of the FSA. By way of an example, consider coding this enhancement of the `\turnstile`. It is required that the person should pass through the turnstile before some fixed period of time has elapsed since the `\coin` event. This could be requested if the turnstile is in fact a security door, and the `\coin` is the entry of an admission code.

This is an exercise, whose answer appears towards the end of the article. Please assume that there is a `\set_timer` action, which will send the `\time_out` event to the `\turnstile` at the end of the fixed time period. Please also think as to how the timer should behave if `\set_timer` is called a second time, before the `\time_out` has occurred.

A note on White Space

The macros above, and all other macros in this article, are to be read and understood in a context when all white space characters (these are space, tab, end of line, and form feed) are ignored. To allow access to space tokens, the category code of ~ is changed to 10. Knuth has programmed TEX so that such characters when read have their character code changed to that of an ordinary space (*The TEXbook*, page 47). In the code below, a ~ will produce a space character.

As usual `@` will be a letter in macro files. In addition, _ will also be a letter. Math subscript, if required, can be accessed via `\sb`, which is provided in `plain` for those whose keyboards do not allow access to _.

The code fragment

```
\catcode`\ =9       \catcode`\^^I=9
\catcode`\^^M=9     \catcode`\^^L=9
\catcode`\~=10
\catcode`\@=11      \catcode`\_=11~
```

will establish this change of category codes.

Comparison with Existing Solutions

The problem of programming the vertical space between elements is scarcely discussed in *The TEXbook*.

LATEX has the very good idea of having all requests for vertical space in the document processed by special macros. This allows some resolution of conflicting rules, such as mentioned in the introduction. According to `latex.tex`, "Extra vertical space is added by the command `\addvspace{SKIP}`, which adds a vertical skip of `SKIP` to the document." For example, the sequence

```
\addvspace{S1} \addvspace{S2}
```

is equivalent to

```
\addvspace{maximum of S1, S2}
```

and so successive `\addvspace` commands will result in only the largest space requested being added to the page. The complicated question, as to whether `2pc` is larger or smaller than `1pc plus 2pc`, is resolved by an `\ifdim` comparison. The former is larger.

There is also a command `\addpenalty` which functions in a similar manner. The `\@startsection` command, which is the generic sectioning command for LATEX, uses these two commands to adjust the penalty and vertical space before a section, subsection, etc.

Another solution has been provided by Paul Anagostopoulos's ZzTEX. One of the most difficult tasks in creating this format, he says (1992, page 502) "was to ensure consistent vertical space between elements." His solution was to define six commands which produce vertical space. As with LATEX, all requests for vertical space should pass through these special commands.

To support these vertical spacing commands, a stack of structures is maintained. Each level of the stack records the type and amount of the previous vertical space request, penalties requested, and other data, or in other words, *the state of the galley*. Because a floating figure, for example, will add items to a galley other than the main vertical list, the state of several galleys must be recorded.

Both LATEX and ZzTEX adopt what may be called an item-based approach. Before and after a text item is added to the galley, vertical space and penalties are added or adjusted. The galley is an essentially passive object on which items are placed, and from which they are removed. This approach places restrictions on the use of the galley by the text items, for whatever is done must be capable of being undone.

In the FSA approach, responsibility is divided between the commands which form the text items, and the FSA which controls vertical space on the galley. Each text item, when it begins, passes a message to the galley. The galley then does what it will, depending particularly on the last item it processed, as recorded in its state.

For example, here is a fragment from a galley space automaton which has states `\quote`, `\text`, and `\section`.

```
* \quote
  @ \text \text \vskip 3pt~
  @ \section \section \vskip 1pc~
```

This code says that should a \quote be followed by \text, then 3 points of space are required. Should a \section follow, then 1 pica is required. Similarly

```
* \title
  @ \quote \quote  \nobreak \vskip 3pt~
```

will inhibit a page break between a \title and a \quote, and also provide some extra vertical space.

The author's experience is that it is natural to express galley spacing rules in terms of an FSA. The \FSA construction produces code that is simple to understand, and it gathers all spacing activity into one location. When errors of implementation or deficiencies of specification arise, it is not hard to change the code to embody the improved understanding.

Those who understand SMALLTALK will realize that the galley is being modelled as an object which responds to the messages sent to it by the text item objects. The state of the galley is memory that is private to the galley, and not accessed by the text item objects except through the messages they pass to the galley. It will also be possible for the galley to send messages to or set flags for the text items, for example to suppress indentation on a paragraph which immediately follows a section title.

Performance

It is important, when writing macros that will appear frequently in the code of other programmers, or which will often be called as TEX typesets a document, that these macros be written with an eye to performance.

There are several aspects to performance. Perhaps most important is the human side. Are the macros easy to use, and do they produce code that is easy to maintain? Do they produce programs that are robust and simple to debug? Are there any traps and pitfalls for the unwary? These are the human questions.

Also a human question is speed of execution. Do the macros act quickly enough? Important here is to have an idea as to how often the macros will be called. This is not the same as how often the macros appear in the source code. Because TEX does not provide access to the current time (as opposed to the time at the start of the job), this quantity will be measured with a stopwatch.

The third measure of performance is the quantity of memory used for the storage and execution of macros. This can be crucial. Often, an enlarged version of TEX the program is required to process a document which uses both LATEX and PICTEX. The capacity of TEX is described by 14 quantities, which are listed on page 300 of *The TEXbook*.

Most important is main memory, which is used for storing boxes, glue, breakpoints, token lists, macros etc. How much space does a macro require? This question is asked and answered on page 383 of *The TEXbook*. Next most important is the hash size, which limits the number of distinct control sequence names.

TEX has a virtual memory system, by which it stores the less often used data on disc, if not enough machine memory is available. Even though TEX's memory may not be exhausted, if the total demands on the token memory exceed the actual machine memory then the resulting use of virtual memory will slow operations.

These three aspects of performance — ease of use, speed of execution, and conservation of resources — may pull in different directions. The best single goal is probably simplicity. It can bring benefit all round.

To indicate the benefits of the FSA approach, here is the \turnstile recoded using \if... to control selection of code to be executed. The state will be stored as a number by \turnstile@. Zero and one will represent \barred and \open respectively. The parameter #1 will be zero or one for \push and \open respectively. Here is one version of the code for \turnstile.

```
\def\turnstile #1
{
  \ifcase #1 ~
    \ifnum \turnstile@ = 1~
      \global \chardef \turnstile@ 0~
      \admit_one
    \fi
  \else
      \global \chardef \turnstile@ 1~
  \fi
}
```

When used, it will generate an error, because we have yet to initialise the private macro \turnstile@.

The reader may complain that although the FSA version is easier than the one just given, the use of the four new control sequences \open, \push, \barred and \coin is a cost not worth bearing. There is some merit in this. However, \open etc. are being used only as labels or delimiters, not macros. Their meaning is irrelevant, if indeed they have a meaning. Thus, existing control words could be used in their place, or the same labels shared by several FSA.

To achieve the ultimate parsimony in use of the hash table, characters can be used as delimiters. There are many character tokens available, such as

x with category code 8 (subscript), which do not appear in the normal use of TEX. They can be used as delimiters. There are problems in this. The programmer would like to write \on, and have some other piece of software to consistently translate it into an unusual character token. In other areas of computing, such a program is called a preprocessor or compiler. Also required is a means of loading macros containing such unusual characters into the memory of TEX.

There is much to gain by writing TEX macros in such a manner, and the author is developing such tools. When available this problem of consumption of the hash table will disappear.

The next concern is main memory. According to the definition of the \FSA constructor which appears later, the second definition of \turnstile will produce macros equivalent to

```
\def \turnstile
  { \FSA@ \turnstile \turnstile@ \off }
```

and

```
\def \turnstile@ #1 #2
{
  * \barred
    @ \push  \barred

  * \open
    @ \push  \barred     \admit_one
    @ \coin  \open
}
```

which, according to the *The TEXbook*, page 383, will occupy 6 and 18 tokens of main memory respectively. The overhead of \FSA@ we will ignore, assuming that is shared between many FSA. By contrast, the \if... version as coded occupies 25. This could be reduced to 19 by replacing explicit numbers with the constants \z@ and \@ne. The \FSA approach seems to be superior when the specifications are more complex.

Some Other Approaches

To investigate speed of execution, an ideal problem will be coded in several ways, and then timed. The problem is that of an n-state n-event FSA, with no actions. The control sequence \state is to hold a number between 0 and $n-1$. The goal is a macro which takes a single parameter, which we shall assume is a single digit, and on the basis of this digit and the existing value of \state assign a new value to \state.

Here is the solution the author believes will be the quickest.

```
\newcount \state        \state 0

\def \quickest #1
{
  \state \csname
           \number \state #1
         \endcsname
}
```

where lines such as

```
\expandafter
\chardef \csname 00 \endcsname 3
```

define control sequences \00, \01, ... which contain the transition data. Clearly, n^2 distinct control sequences will be required to hold this table. Actions can also be supplied. With the definition

```
\expandafter
\def \csname 12 \endcsname
{
  3~
  \myaction
}
```

event 2 applied to state 1 will change the state to 3 and call \myaction.

Although quick, this approach does take a large bite out of the hash table, and so is probably not appropriate for coding the change of state as items are added to the galley. During a normal document this code will be executed perhaps 12 times each page, whereas font changes and accents will be called more often. This approach has been presented as to show how quickly TEX can do the calculation, if resources are no limitation.

There is another context in which TEX keeps a record of the state, and adjusts the action in terms of what follows. As it typesets a word, one letter after another, it consults the information that is stored in the .tfm file, to produce kerns and ligatures. TEX has been carefully programmed to store this information compactly, and to access it at high speed while processing the characters. This is the famous ligature and kerning, which has become yet more powerful with version 3 of TEX.

Suppose that in a font one wants 2 units of extra space between the characters whenever a 0 is followed by a 4. The property list .pl file below contains the script for the character 0. Line 7 says that when a 0 is followed by a 4, a kern of 2 design units should be inserted. This process is similar to the operation of the \FSA construction. It is clear that the transition table of an actionless automaton can be stored in the ligature table of such a font. More details of ligatures and kerning may be found in *The* METAFONT*book*, and Knuth (1989, 1990).

```
1. (LIGTABLE
2.    (LABEL C 0)
3.    (KRN C 0 R 1)
```

```
4.    (KRN C 1 R 3)
5.    (KRN C 2 R 1)
6.    (KRN C 3 R 4)
7.    (KRN C 4 R 2)
8.    (STOP)
9.)
```

A font needs characters as well as ligatures and kerns. Normally, the characters of a font are defined, and then the kerns and ligatures are something of an afterthought. For fonts that encode FSA, the characters are the afterthought. The lines

```
(CHARACTER C 0)
(CHARACTER C 1)
(CHARACTER C 2)
(CHARACTER C 3)
(CHARACTER C 4)
```

will supply 0, 1, 2, 3 and 4 as characters for this font, all with zero height, depth, and width. Now encode the remaining transitions into the ligature table, and call the font fsa5. This font encodes a 5 by 5 FSA.

The problem now is to access this data from within TEX. We shall assume that the design size and design unit of the font are both 1pt. This makes the examples easier. First load the font fsa5

```
\font \fsa fsa5~at~1sp
```

at one scaled point. The width of the box

```
\hbox { \fsa 04 }
```

in scaled points will be transition table entry for state 0 and event 4. As before #1 will be a digit. The \state will be a \chardef. The macro

```
\def \quicker #1
{
  \setbox \zero \hbox
    { \fsa \number \state #1 }
  \chardef \state \wd \zero
}
```

uses the transition table to determine the new \state.

Finally, here is a variant of the \FSA construction, specially adapted to this ideal problem. The transition data is stored in the table \slow@, where the x's indicate where the values should be placed.

```
\def \slow@
{
  *0 @0x @1x @2x @3x @4x
  *1 @0x @1x @2x @3x @4x
  *2 @0x @1x @2x @3x @4x
  *3 @0x @1x @2x @3x @4x
  *4 @0x @1x @2x @3x @4x
}
```

The FSA itself will use the numerical value of \state as a delimiter. This is the reason for the \expandafters.

```
\def \slow #1
{
  \expandafter \def
  \expandafter \next
  \expandafter ##  \expandafter 1
  \expandafter *
               \number \state
               ##2
               @ #1
               ##3
               ##4
               ;
  { \chardef \state ##3 ~ }

  \expandafter \next \state@ ;
}
```

Finally, for those who are not in a hurry, here is the same FSA coded using the \FSA constructor

```
\FSA \slowest 0
{
  *0 @0x @1x @2x @3x @4x
  *1 @0x @1x @2x @3x @4x
  *2 @0x @1x @2x @3x @4x
  *3 @0x @1x @2x @3x @4x
  *4 @0x @1x @2x @3x @4x
}
```

where as before the x's indicate where the transition table should be entered.

Speed of Execution

The macros \quickest, \quicker, \slow, and \slowest will now be timed.

Even the slowest macro executes in a fraction of a second. The stopwatch will be applied not to one application of a macro, but hundreds or thousands. Rather than use a \loop, which will introduce considerable overheads of its own into the elapsed time, the lines

```
\let  \0 \relax
\def  \1 {\0\0\0\0\0\0\0\0\0\0}
\edef \2 {\1\1\1\1\1\1\1\1\1\1}
\edef \3 {\2\2\2\2\2\2\2\2\2\2}
\def  \4 {\3\3\3\3\3\3\3\3\3\3}
\def  \5 {\4\4\4\4\4\4\4\4\4\4}
```

will be read, resulting in macros \n which expand \0 exactly 10^n times, for $n = 0, 1, 2, 3, 4$ and 5. Timing tests can now be done by setting \0 to an appropriate value, typing \3, \4 or even \5 at the console, and starting the stopwatch.

Here is a table of results for an old MS-DOS personal computer, with a 286 processor running at 10Mhz.

A	B	C	D	E
\let \0 \relax	\5	4.3	43	-
\def \0 {}	\5	14.9	149	-
\def \0 { \relax\relax }	\5	22.8	228	28
\def \0 { \quickest 0 }	\4	16.3	1630	1430
\def \0 { \quicker 0 }	\4	33.2	3320	3120
\def \0 { \quicker 4 }	\4	33.5	3350	3150
\def \0 { \slow 0 }	\3	10.6	10600	10400
\def \0 { \slowest 0 }	\3	12.0	12000	11800
\def \0 { \bigslow 0 }	\3	27.1	27100	26900
\def \0 { \bigquicker 0 }	\4	33.6	3360	3160
\def \0 { \bigquicker 9 }	\4	34.3	3430	3330
\let \0 \extract	\4	39.7	3970	3770
\let \0 \quickextract	\4	13.4	1340	1140

The first column provides a meaning for the control sequence \0 which is repeatedly called by \1, \2, etc. Column **B** is the number of iterations performed, and column **C** the time in seconds taken for this number of iterations. The raw time taken for each iteration, the quotient of **C** by **B**, is in column **D**. Finally, column **E** is **D** adjusted to account for the overheads involved in the timing tests.

The first three lines of this table are there to help establish a baseline. It may be surprising that the empty macro executes at about one third of the speed of the \relax command. A certain amount of time will be spent in expanding \0, \1, \2, \3 and \4. This time should be discounted from the raw figures, to obtain the time actually executing the macro being timed. To produce round numbers, the base line correction has been taken as −200. This is close enough, given the accuracy on the raw data, and the purpose of the table.

According to *The* METAFONT*book* (page 317) TEX will stop reading a ligature table once it has found a "hit", and thus the entries appearing earlier in the script for a given label will be found quicker than those that are later. For the font fsa5 as used in the test, the new state is the same digit as the event, but it is coded as a 5 by 5 table, with smaller digits first. Thus, \quicker 0 will execute just a bit quicker than \quicker 4. The timing tests show that the difference in time, while significant, is small in relation to the whole.

It may again be surprising that \slow and \slowest are relatively so close to each other (although twice the *difference* is approximately the time taken by \quicker). If performance is an issue, it seems to be better to move to the \quicker or \quickest style rather than produce a custom FSA which works through macro expansion alone.

The previous tests relate to a 5 by 5 transition table. It should be clear that for a 10 by 10 table, the \quickest approach will be just as rapid as before. The \slow macro rewritten for 10 by 10 is \bigslow which runs at well under half the speed. This is because there is a quadratic element in the running time. (Encoding the data requires a replacement text with over $300 = 3 \times 10 \times 10$ tokens, each of which must be read as the helper macro searches for its delimiters.) The \bigquicker macro uses the font fsa10, which encodes the general 10 by 10 transition table. The decrease in performance is slight, compared to the 5 by 5 \quicker macro. Why is this? When TEX consults a ligature table, it needs to find the location of the label for the first of the two characters. It can find this data immediately, because this location is stored as part of the information for the character. There is no quadratic element in the running time.

It should be noted that adding actions to the body of a \FSA will further increase the execution time, even if they are not selected, because they too constitute tokens which the helper macro has to read.

The last two lines of the table give the times for utility macros \extract and \quickextract, which will be described later.

The \FSA approach is probably the easiest to write code for. The \csname ... \endcsname approach runs very quickly, but will consume the hash size. The font ligature table approach gives code that runs quite quickly, without wasting the hash size. However, it will not be so easy to write code for this approach, not least because symbolic names will not be available. Property list files are not a preferred programming language. There will be more on this later.

Finally, when it comes to size, the font approach is a clear winner. The font fsa5, which encodes the general 5 by 5 transition table, occupies 47 words of font information, while the replacement text for the \FSA approach and also the slightly quicker variant \slow both consist of 85 tokens. The font fsa10 encoding the general 10 by 10 table occupies 132 words of font information.

The next section will show how the action as well as the new state can be recorded by and recovered from the kern.

Exploring Fonts

Here are some of the important facts, concerning TEX's capacity for handling fonts. A font can contain up to 256 characters. The maximum number of fonts that may be loaded depends on the implementation, and is commonly 127 or 255.

The total amount of font information that can be stored might be 65,000 pieces. Each ligature or kern occupies one such piece. Before TEX 3 and METAFONT 2.7, there was an effective limit of 256 on the number of ligatures or kerns in a single font, but now more than 32,000 are possible. In addition, TEX 3 has smarter ligatures, which in particular allow the ligature tables for several characters to share common code.

A kern can be positive or negative, and must be smaller in magnitude than 2048pt. It can be specified with a precision of 1sp (scaled point), of which there are $65536 = 2^{16}$ in a single point. Thus there are $268435455 = 2^{28} - 1$ possible different values for a kern. Previously, the transition table of a FSA has been encoded by setting the kern for a character pair to be the new state, as a digit. Instead, the ASCII code for the new state could be stored in the kern, along with a lot more information.

In addition, each character has a height, a depth, a width, and an italic correction. Although each character in a font can have its own width, there can be at most only 15 different nonzero heights, nonzero depths, and nonzero italic corrections in a single font. Each font has at least seven dimensions, accessed in TEX through the `\fontdimen` primitive. A font can have many more such dimensions; at least 32,000 are possible.

The reader may wonder how to use the wealth of digital information in a font. Here is one method. It assumes that the kern is to have positive width, and to be a nine-digit number when writen in units of sp. This kern gives the width to `\box\zero`.

```
\wd\zero 123456789 sp  % sample value
\def \extract
{
  \expandafter
  \extract@ \number \wd \zero
}

\def \extract@ #1#2#3#4#5#6#7#8#9
   { \chardef\state #2#3#4~ }
```

Here is another. The control sequence `\count@` is a count register dedicated to scratch purposes. Note the one million is too large to be stored as a `\mathchar`.

```
\newcount \million
\million 1000000

\def \quickextract
{
  \count@ \wd \zero
  \divide \count@ \million
  \chardef \state \count@
}
```

Speed of execution for these macros is respectable, and indicates that even when the extraction time is figured in, the font approach will run quicker than the `\FSA` approach, except perhaps for the very smallest examples. For the moment it is enough to know that digital extraction is practical. What will be best will depend on the demands of the applications.

A more substantial problem is this. There may be only 127 fonts available, or perhaps up to 255 if a really large version of TEX is used. The fonts which encode FSA will never get into the final `.dvi` file (unless the macros are not working properly) and so will not trouble the `.dvi` device driver. However, to use 10 or 15 of the precious allocations of fonts for the coding of FSA may be too much. Fortunately, the same font can be used to store several FSA. In fact, as long as no FSA has more than 255 distinct events, the limit is on the total number of states, across the various FSA being packed into the font. It seems that in practice most FSA will have rather more events than states.

Implicit in this discussion is the assumption that TEX 3 is being used. Earlier versions of TEX are limited to 255 kerns and ligatures for each font. This may be enough for particular applications but even those that do may grow beyond this limit over the course of time. While compatibility with TEX 2 is desirable, it should not be required.

Is It Practical?

This article began by defining the concept of a finite state automaton. The one implementation, via the `\FSA` macro is easy to code, moderate in its use of resources, and relatively slow to run. The other, via font ligature and kerning tables, is quick to run, impressively economical in resources, and difficult to program without special tools.

The state of the galley, as we have defined it, must be recalculated with every new paragraph, heading, etc. Although not a rare event, it is not so ubiquitious that the very best performance is demanded.

Use of the font method will require special tools that translates code, perhaps written with a TEX like syntax, into a property list file from which the `.tfm` file can be produced using the program `pltotf`, together with some TEX code for handling the special actions which cannot be encoded in the font information.

Such a tool would not be tremendously difficult to write, but to ensure that it is available to all TEX users on all platforms is another matter. The only

programming language a TEX user can be sure to have is TEX itself! (one can also hope that the user has BIBTEX and makeindex available.) This seems to force one to write the tool in TEX. But then it will be interpreted, not compiled, and so perhaps slow to run. There have been complaints about TEX as a programming language, some perhaps well founded. It is possible to write a compiler/preprocessor in TEX itself, but it is not as it now stands a tool well adapted to this task.

This line of thought, which is relevant to discussion of the future of TEX, will be investigated further in a separate article.

Solution to Exercise

Here is the solution to the \turnstile problem with timer. The following rules govern the behaviour. Whenever \coin occurs, the state becomes \open and action is \set_timer, irregardless of the existing state. Any other event, including \time_out, should result in the \barred state and no action, with one exception. If the state is \open then a \push will still \admit_one and result in a \barred state.

```
\FSA \turnstile \barred
{
  * \open
    @ \push  \barred      \admit_one

  * #1
    @ \coin  \open        \set_timer
    @ #2     \barred
}
\endverbatim
%%
```

The \set_timer macro requires a small amount of numerical information. It must record the number of clock ticks since the turnstile last became \open. The \timer FSA will have two states, \off and \on. It will respond to events \on, \off, and \tick. The \off event is for internal use only, to turn the \timer off when time has run out. Also for internal use is \counter, which counts the \tick events since the \timer was turned \on. The \set_timer macro in \turnstile should send the \on message to the \timer automaton.

```
\FSA \timer \off
{
  * \on
    @ \tick \on
        \ifnum \counter < 100~
          \global \advance \counter 1~
        \else
          \timer \off
          \turnstile \time_out
        \fi
  * \off
    @ \on    \on     \global \counter 0~
    @ #2     \off
}
\endverbatim
%%
```

Finally, it should be noted that the timer will still receive the \tick event from the clock, whether it is \on or \off. The last two transition line say that when \off the timer responds only to the \on event.

Coding the \FSA Macros

The \FSA macro depends on some helper macros \FSA@ and \FSA@@ which are similar to the \CASE and \FIND macros which the author has defined elsewhere (Fine, 1993).

The definition of \FSA is a little complicated. The command

```
\FSA \mymacro \initial_state { ... }
```

will first of all define the \mymacro to have expansion

```
\FSA@ \mymacro \FSA\mymacro
              \initial_state
```

where \FSA\mymacro *is a single control word.* Next, \FSA\mymacro is defined to be a two parameter macro (this allows #1 and #2 to appear in the transition table of the FSA) whose expansion is indeed the transition table.

These goals are accomplished by the following definition.

```
\def\FSA #1 % name of FSA
         #2 % initial state
         #3 % transitions
{
  % define the FSA
  \edef #1
  {
    \noexpand \FSA@
    \noexpand #1  % name of FSA
    \expandafter\noexpand
    \csname FSA\string #1 \endcsname
    \noexpand #2  % initial state
  }

  % define the transitions store
  \expandafter\gdef
  \csname FSA\string #1 \endcsname
      ##1 % <state>
      ##2 % <event>
  { #3 }
}
```

With these values, the result of expanding \mymacro \event is

```
\FSA@ \mymacro \FSA\mymacro
      \state \event
```

and so a start can be made on the coding of the helper macro \FSA@. It must expand the transitions

store \FSA\mymacro, passing \state and \event as parameters, and then look within it for the current state and the following transition line. The scratch helper macro \next will search the expansion of \FSA\mymacro. This macro can readily find the \state and the \event, and from this the new state. To find the action is more difficult, because the action portion is not delimited by a fixed token. The next token after the action may be a @, or it may be a *. Or it may be that the transition line selected is at the very end of the transition store. If not, then the rest of the transition store must be discarded. To help take care of these possibilities, helpful delimiters are placed after \FSA\mymacro.

Here is the code for \FSA@. Like \CASE and \FIND, it is a selector macro.

```
\def\FSA@ #1 % name
          #2 % transitions store
          #3 % <state>
          #4 % <event>
{
  \def\next  ##1 % discard
             * #3 % find <state>
              ##2 % discard
             @ #4 % find <event>
              ##3 % new state
              ##4 % <action> + rubbish
               @  % next transition line
  {
    % redefine the FSA
    \gdef #1 % name
    {
      \FSA@
         #1 % name
         #2 % transitions store
        ##3 % new state
    }

    % prepare to extract the action
    \FSA@@ ##4 *  % may need this *
  }

  \expandafter
  \next #2 % transitions store
        #3 % <state>
        #4 % <event>
         @ % may need a trailing @
    \FSA@ % final delimiter
}
```

Careful thought is required to follow the execution of \FSA@. First \next is defined, with the existing state and the event as delimiters. This allows \next to extract information from the transition store. In fact, \next must find the current state, and then the event which occured, and then extract the action, and then discard the rest of the transition store, and then execute the action.

The transition store should be expanded, with the state and event as parameters, before \next is called. The part of the transition store which lies *before* the new event and action should be discarded, as should the part which lies *after* the event and action. The delimiters supplied to the definition of \next will discard the *before* portion. The new state is found immediately after the old state, in the expansion of \next, but the action is not so easy. It may be delimited by * or by @, depending on whether what follows is another line for the same state, or the script for another state. If the action is the last line of the transition store, the action will not have a delimiter at all. This range of possibilities is a consequence of the flexible syntax allowed in the \FSA command. The trailing tokens @ \FSA@ at the end of the expansion of \FSA@ are there to allow the action to be extracted, and the rest of the transition store to be discarded.

The execution of \next will result in \mymacro being defined. The new value will be the same as the old, except that \state will have been replaced by the new state. Now to extract the action, which lies in the tokens formed by the expansion of the transition store, which were not absorbed by \next. Suppose that the action had been delimited by an * rather than an @ in the transition store, or even by nothing at all. In either of these cases, the action — which is picked up and copied by \next — will now be delimited by a *. So, all that remains is to pick up the action, throw away the rubbish, and perform the action. This is done by a final helper macro.

```
\def\FSA@@  #1    % <action>
             *    % delimiter
            #2    % <discard>
         \FSA@    % delimiter
       { #1 }     % at last, the action
```

The very last token(s) in the expansion of \mymacro come out to be the action for the curent state and event, as determined by the transitions store. The action is not called until the FSA has finished *its* activities. Thus, the action can take parameters, if need be. This may be helpful. In SMALLTALK, messages are allowed to have parameters.

Bibliography

Anagnostopoulos, Paul. "ZzTEX: A macro package for books", *TUGboat*, **13** (4), pages 497 - 505, 1992.

Fine, Jonathan. "The \CASE and \FIND macros", *TUGboat*, **14** (1), pages 35 - 39, 1993.

Knuth, Donald E. "The new versions of TEX and METAFONT", *TUGboat*, **10** (3), pages 325 - 328, 1989.

Knuth, Donald E. "Virtual Fonts: More Fun for Grand Wizards", *TUGboat*, **11** (1), pages 13 - 23, 1990.