ELECTRONIC PUBLISHING, VOL. 2(2), 101–115 (DECEMBER 1989)

Drawing trees nicely with TEX

A. BRÜGGEMANN-KLEIN
Institut für Informatik
Universität Freiburg, Rheinstr. 10–12
7800 Freiburg, West Germany

D. WOOD
Data Structuring Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario
N2L 3G1, Canada

SUMMARY

We present a new solution to the tree drawing problem that integrates an excellent tree drawing algorithm into one of the best text processing systems available. More precisely, we present a TEX macro package called TreeTEX that produces drawings of trees from a purely logical description. Our approach has three advantages: labels for nodes can be handled in a reasonable way; porting TreeTEX to any site running TEX is a trivial operation; and modularity in the description of a tree and TEX's macro capabilities allow for libraries of subtrees and tree classes.

In addition, TreeTEX has an option that produces drawings that make the *structure* of the trees more obvious to the human eye, even though they may not be as aesthetically pleasing.

KEY WORDS Trees Graphics Drawing algorithms TEX

1 INTRODUCTION

The problem of successfully integrating pictures and text in a document processing environment is tantalizing and difficult. Although there are systems available that allow such integration, they fall short in many ways, usually in document quality. Furthermore, most authors using document preparation systems are neither book designers nor graphic artists. Just as modern document preparation systems do not expect an author to be a book designer, so we would prefer that they do not expect an author to be a graphic artist. The second author, Wood, needed to draw many trees in a series of papers and in a projected book on trees. This problem enabled us to tackle the integration issue for one sub-area of graphics, namely, tree drawing. We had the decided advantage that there already existed good algorithms to draw trees *without any author intervention*. Previous experience of the integration of pictures and text had been uninspiring; the systems expected the author to prepare each picture in total. For example, a tree could be built up from smaller sub-trees but the relative placement of them was left to the author. This situation continues to hold today with the drawing facilities available on most personal computers, and, because of this, the resulting figures still appear to be 'hand-drawn.' Additionally, they are of inferior quality when compared with the quality of the surrounding text.

In this paper we present an entirely new solution that integrates a tree drawing algorithm into one of the best text processing systems available. More precisely, we describe TreeTEX, a TEX macro package that produces an aesthetically pleasing drawing of a tree from a purely logical description. We made two fundamental design decisions that

0894–3982/89/020101–15$7.50

Received 5 October 1988
Revised 1 May 1989

heavily influenced the method of implementation. First, we wanted to allow an author to label the nodes of a tree. This decision means that the tree drawing package must be able to typeset labels exactly as they would be typeset by the typesetting program. There are two reasons for this. Text should be typeset consistently, wherever it appears in a document, and the tree drawing program needs to know the dimensions of the typeset labels. Second, we wanted to ensure that the program could be ported easily to other installations and sites, so that other, putative users would be able to use it easily. Indeed, TreeTEX has been used successfully to typeset trees in [1], [2], and [3].

By basing our package on TEX, which more for subjective reasons we preferred over other typesetting systems such as *troff*, we could ensure wide interest in the package. By implementing it as a TEX macro package instead of a preprocessor we made porting trivial and, furthermore, ensured consistency of typeset text within a document. The adverse side of this decision is that we had to program with TEX macros, not an experience to be recommended, and we had to live with the inherent register limitations of TEX.

This paper consists of a further nine sections. In Sections 2, 3 and 4, we discuss the aesthetics of tree drawing and the algorithm of Reingold and Tilford [4]. In Sections 5, 6, and 7, we describe our method of incorporating tree drawing into TEX. Then, in the last three short sections, we consider the expected number of registers TEX needs to draw a tree, the user interface (and three TreeTEX examples), and discussion of, among other things, the performance of TreeTEX.

2 AESTHETICAL CRITERIA FOR DRAWING TREES

In this paper, we are dealing with ordered trees in the sense of [5], specifically binary and unary–binary trees. A *binary tree* is a finite set of nodes that is either empty, or consists of a root and two disjoint binary trees called the left and right sub-trees of the root. A *unary–binary tree* is a finite set of nodes that is either empty, or consists of a root and two disjoint unary–binary trees, or consists of a root and one non-empty unary–binary tree. An *extended binary tree* is a binary tree in which each node has either two non-empty sub-trees or two empty sub-trees.

There are some basic agreements on how such trees should be drawn, reflecting the up–down and left–right ordering of nodes in a tree. In [4] and [6] these basic agreements were formalized as the following axioms.

1. Trees impose a distance on the nodes; no node should be closer to the root than any of its ancestors.
2. Nodes on the same level should lie on a straight line, and the straight lines defining the levels should be parallel.
3. The relative order of nodes on any level should be the same as in the level order traversal of the tree.

These axioms guarantee that trees are drawn as planar graphs: edges do not intersect except at nodes. Two further axioms improve the aesthetical appearance of trees.

4. In a unary–binary tree, each left child should be positioned to the left of its parent, each right child to the right of its parent, and each unary child should be positioned below its parent.

5. A parent should be centred over its children.

An additional axiom deals with the problem of tree drawings becoming too wide and therefore exceeding the physical limit of the output medium:

6. Tree drawings should occupy as little width as possible without violating the other axioms.

In [6], Wetherell and Shannon introduce two algorithms for tree drawings, the first of which fulfills axioms 1–5, and the second 1–6. However, as Reingold and Tilford in [16] point out, there is a lack of symmetry in the algorithms of Wetherell and Shannon which may lead to unpleasant results; therefore, Reingold and Tilford introduce a new structured axiom.

7. A sub-tree of a given tree should be drawn the same way regardless of where it occurs in the tree.

Axiom 7 allows the same tree to be drawn differently only when it occurs as a sub-tree in different trees. Reingold and Tilford give an algorithm which fulfills axioms 1–5 and 7. Although this algorithm does not fulfill axiom 6, the aesthetical improvements are well worth the additional space. Figure 1 illustrates the benefits of axiom 7, and Figure 2 shows that the algorithm of Reingold and Tilford violates axiom 6.

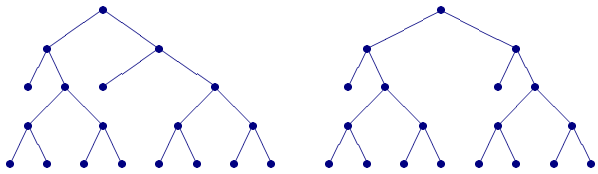


Figure 1. The left tree is drawn by the algorithm of Wetherell and Shannon [6], and the tidier right one is drawn by the algorithm of Reingold and Tilford [4]

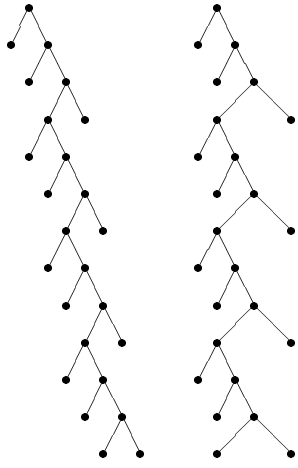


Figure 2. The left tree is drawn by the algorithm of Reingold and Tilford [4], but the right tree shows that narrower drawings fulfilling all aesthetic axioms are possible

3 THE ALGORITHM OF REINGOLD AND TILFORD[4]

The algorithm of Reingold and Tilford (hereafter called 'the RT algorithm') takes a modular approach to the positioning of nodes. The relative positions of the nodes in a sub-tree are calculated independently of the rest of the tree. After the relative positions of two sub-trees have been calculated, they can be joined as siblings in a larger tree by placing them as close together as possible and centring the parent node above them. Incidentally, this modular approach is the reason that the algorithm fails to fulfill axiom 6; see [7]. Two sibling sub-trees are placed as close together as possible, during a post-order traversal, as follows. Imagine that the two sub-trees of a binary node have been drawn and cut out of paper along their contours. Then, starting with the two sub-trees superimposed at their roots, move them apart until a minimal agreed-upon distance between the trees is obtained at each level. This can be done gradually. Initially, their roots are separated by some agreed-upon minimum distance; then, at the next level, they are pushed apart until the minimum separation is established there. This process is continued at successively lower levels until the last level of the shorter sub-tree is reached. At some levels no movement may be necessary, but at no level are the two sub-trees moved closer together. When the process is complete, the position of the sub-trees is fixed relative to their parent, which is centred over them. Assured that the sub-trees will never be placed closer together, the post-order traversal is continued.

A non-trivial implementation of this algorithm has been obtained by Reingold and Tilford in [4] that runs in time O(N), where N is the number of nodes of the tree to be drawn. Their crucial idea is to keep track of the contour of the sub-trees by special pointers, called threads, such that whenever two sub-trees are joined, only the top part of the trees down to the lowest level of the smaller tree need to be taken into account.

The nodes are positioned on a fixed grid and are considered to have zero width; labelling is not provided. Although the algorithm only draws binary trees, it is easily extended to multiway trees.

4 IMPROVING HUMAN PERCEPTION OF TREES

It is common understanding in book design that aesthetics and readability do not necessarily coincide, and—as Lamport [8] puts it—'documents are meant to be read, not hung in museums.' Therefore, readability is more important than aesthetics.

When it comes to tree drawings, readability means that the structure of a tree must be easily recognizable. This criterion is not always met by the RT algorithm. As an example, there are trees whose structure is different even though they have the same number of nodes on each level. The RT algorithm might assign identical positions to these nodes making it very hard to perceive the structural differences. Hence, we have modified the RT algorithm such that additional white space is inserted between sub-trees of *significant* nodes. Here a binary node is called significant if the minimum distance between its two sub-trees is achieved *below* their root level. Setting the amount of additional white space to zero retains the original RT placement. The effect of having non-zero additional white space between the sub-trees of significant nodes is illustrated in Figure 3.

Another feature we have added to the RT algorithm is the possibility to draw an unextended binary tree with the same placement of nodes as its associated extended version; this makes the structure of a tree more prominent; see Figure 4. We define the

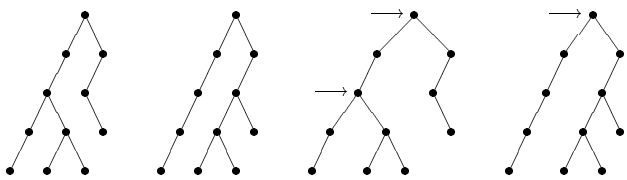


Figure 3. The nodes of the first two trees are placed in the same positions by the RT algorithm, although the structure of the two trees is different. The alternative drawings highlight the structural differences of the trees by adding additional white space between the sub-trees of (⟶) significant nodes

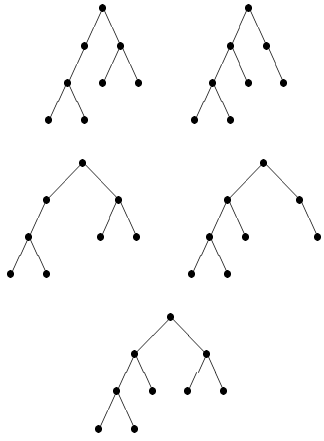


Figure 4. As in Figure 3, the nodes of the first two trees are placed in the same position by the RT algorithm, although their structure is different. The modified RT algorithm highlights the structural differences of the trees by drawing them like their identical extended version (given in the third row), but suppressing the additional nodes

associated extended version of a binary tree to be the binary tree obtained by replacing each empty sub-tree having a non-empty sibling with a sub-tree consisting of one node.

5 TREES IN A DOCUMENT PREPARATION ENVIRONMENT

Drawings of trees do not usually appear by themselves, but are included in some text that is itself typeset by a text processing system. Therefore, a typical scenario is a pipe of three stages. First, we have a tree drawing program that calculates the positioning of the nodes of the tree to be drawn and outputs a description of the tree drawing in some graphics language; this is followed by a graphics system that transforms this description into an intermediate language that can be interpreted by the output device; and, finally, we have the text processing system that integrates the output of the graphics system into the text.

This scenario loses its linear structure once nodes have to be labelled, since the labelling influences the positioning of the nodes. Labels usually occur inside, to the left of, to the right of, or beneath nodes (the latter only for external nodes). Their widths

should certainly be taken into account by the tree drawing algorithm. But the labels have to be typeset first to determine their extent, preferably by the typesetting program that is used for the regular text, because this ensures uniformity in the textual parts of the document and provides the author with the full power of a text processing system for composing the labels. Hence, a more complex communication scheme than a simple pipe is required.

Although a system of two processes running simultaneously might be the most elegant solution, we wanted a system that is easily portable to widely different machines at our sites including personal computers with single process operating systems. Therefore, we decided to use a text processing system having programming facilities powerful enough to program a tree drawing algorithm and graphics facilities powerful enough to draw a tree. One text processing system rendering outstanding typographic quality and satisfactory programming facilities is TEX, developed by Knuth at Stanford University; see [9]. The TEX system includes the following programming facilities.

1. Datatypes:
   integers (256), dimensions[1] (512), boxes (256), tokenlists (256), and boolean variables (unrestricted).
2. Elementary statements:
   a := const, $a := b$ (all types);
   $a := a + b$, $a := a * b$, $a := a/b$ (integers and dimensions); and
   horizontal and vertical nesting of boxes.
3. Control constructs:
   if-then-else statements testing relations between integers, dimensions, boxes, or boolean variables.
4. Modularization constructs:
   macros with up to 9 parameters (can be viewed as procedures without the concept of local variables).

Although the programming facilities of TEX hardly exceed the abilities of a Turing machine, they are sufficient to handle small programs. How about the graphics facilities? Although TEX has no built-in graphics facilities, it allows the placement of characters in arbitrary positions on the page. Therefore, complex pictures can be synthesized from elementary picture elements treated as characters. Lamport has included such a picture drawing environment in his macro package LATEX, using quarter circles of different sizes and line segments (with and without arrow heads) of different slopes as basic elements; see [8]. These elements are sufficient for drawing trees.

This survey of TEX's capabilities implies that TEX may be a suitable text processing system to implement a tree drawing algorithm directly. We base our algorithm on the RT algorithm, because this algorithm gives, aesthetically, the most pleasing results. In the first version presented here, we restrict ourselves to unary–binary trees, although our method is applicable to arbitrary multiway trees. To take advantage of the text processing environment, we expand the algorithm to allow labelled nodes.

In contrast to previous tree drawing programs, we feel no necessity to position the

[1] The term *dimension* is used in TEX to describe physical measurements of typographical objects; for example, the length of a word.

nodes of a tree on a fixed grid. While this may be reasonable for a plotter with a coarse resolution, it is certainly not necessary for TEX, a system that is capable of handling arbitrary dimensions and producing device *independent* output.

6 A REPRESENTATION METHOD FOR TEXTREES

The first problem to be solved in implementing our tree drawing algorithm is how to choose a good internal representation for trees. A straightforward adaptation of the implementation by Reingold and Tilford requires, for each node, at least:

1. two pointers to the children of the node,
2. two dimensions for the offset to the left and the right child (these may be different once there are labels of different widths to the left and right of the nodes),
3. two dimensions for the x- and y-coordinates of the final position of the nodes,
4. three or four labels, and
5. one token to store the geometric shape (circle, square, framed text, etc.) of the node.

Because these data are used frequently in calculations, they should be stored in registers (that is what variables are called in TEX) rather than being recomputed, to obtain reasonably fast performance. This gives a total of $10N$ registers for a tree with N nodes, which quickly exceeds TEX's limited supply of registers. Therefore, we present a modified algorithm hand-tailored to the abilities of TEX. We start with the following observation. Suppose a unary–binary tree is built bottom-up, using a post-order traversal. This can be done by repeating the following three steps in an order determined by the tree to be built.

1. Create a new sub-tree consisting of one external node.
2. Create a new sub-tree by appending the two sub-trees last created to a new binary node; see Figure 5.
3. Create a new sub-tree by appending the sub-tree created last as a left, right, or unary sub-tree of a new node; see Figure 5.

(A pointer to) each sub-tree that has been created in steps 1–3 is pushed onto a stack, and steps 2 and 3 remove two trees or one tree, respectively, from the stack before the push operation is carried out. The tree to be built is the tree remaining on the stack.

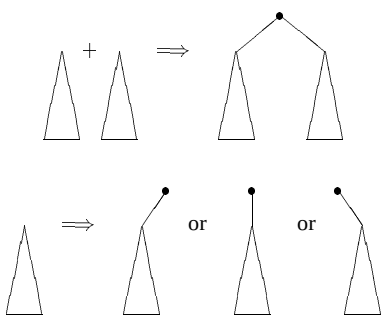


Figure 5. Construction steps 2 and 3

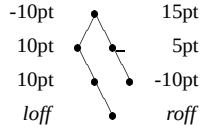
This tree traversal is performed twice in the RT algorithm. During the first pass, at each execution of step 2 or 3, the relative positions of the sub-tree(s) and of the new node are computed. A closer examination of the RT algorithm reveals that information about the sub-tree's coordinates is not needed during this pass; the contour information alone is sufficient. Complete information is only needed in the second traversal, when the tree is really drawn. This is where we can use a special feature of TEX that allows us to save registers. Unlike Pascal, TEX has the capability of storing a drawing in a single box register that can be positioned freely in later drawings. This means that in our implementation the two passes of the original RT algorithm can be woven into a single pass, storing the contour and drawing of each sub-tree on the stack. Although the latter is a complex object, it takes only one of TEX's precious registers.

7 THE INTERNAL REPRESENTATION

Given a tree, the corresponding TEXtree is a box containing the 'drawing' of the tree, together with some additional information about the contour of the tree. The reference point of a TEXtree-box is always in the root of the tree. The height, depth and width of the box of a TEXtree are of no importance in this context.

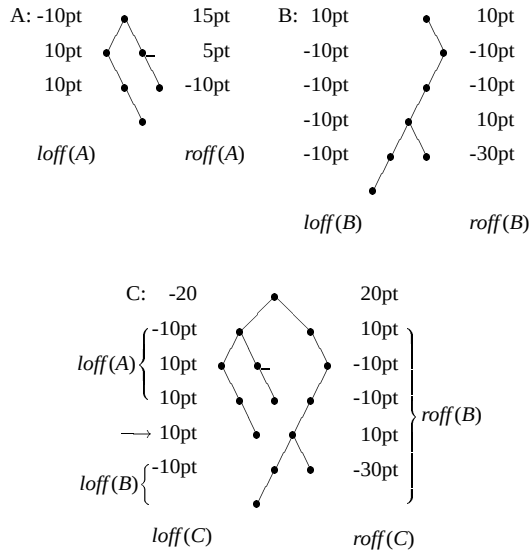
The additional information about the contour of the tree is stored in registers for numbers and dimensions and is needed in order to put sub-trees together to form a larger tree. An array *loff* of dimensions contains for each level of the tree the horizontal offset between the left end of the leftmost node at the current level and the left end of the leftmost node at the next level. The horizontal offset between the root and the leftmost node of the whole tree is held in *lmoff*, and the horizontal offset between the root and the leftmost node at the bottom level of the tree is held in *lboff*. Finally, *ltop* holds the distance between the reference point of the tree and the leftmost end of the root. We use *roff*, *rmoff*, *rboff*, and *rtop* as the corresponding variables for 'left' replaced by 'right.' Finally, *height* holds the height of the tree, and *type* holds the geometric shape of the root of the tree. Figure 6 shows an example TEXtree, which is a tree drawing and corresponding additional information.

Given two TEXtrees *A* and *B*, how can a new TEXtree *C* be built that consists of a new root and has *A* and *B* as sub-trees? An example is given in Figure 7. First we determine



height: 3, type: dot, ltop: 2pt, rtop: 2pt, lmoff: -10pt, rmoff: 20pt, lboff: 10pt, rboff: 10pt

Figure 6. A TEXtree consists of the drawing of the tree and the additional information. The width of the dots is 4pt, the minimal separation between adjacent nodes is 16pt, making a distance of 20pt centre to centre. The length of the small rule labelling one of the nodes is 5pt. The column left (right) of the tree drawing is the array loff (roff), describing the left (right) contour of the tree. At each level, the dimension given is the horizontal offset between the border at the current and at the next level. The offset between the left border of the root node and the leftmost node at level 1 is −10pt, the offset between the right border of the root node and the rightmost node at level 1 is 15pt, etc.



	A	*B*	*C*
height	3	5	6
type	dot	dot	dot
ltop	2pt	2pt	2pt
rtop	2pt	2pt	2pt
lmoff	-10pt	-30pt	-30pt
rmoff	20pt	10pt	30pt
lboff	10pt	-30pt	-10pt
rboff	10pt	-30pt	-10pt

level	*totsep*	*currsep*
0	20pt	0/16pt
1	25pt	11/16pt
2	40pt	1/16pt
3	40pt	16pt

Figure 7. The TEXtrees A and B are combined to form the larger TEXtree C. The first table gives the additional information of the three TEXtrees, and the second table gives the history of the computation for totsep and currsep

which tree is higher; this is *B* in the example. Then we have to compute the minimum distance between the roots of *A* and *B*, such that at all levels of the trees there is free space of at least *minsep* between the trees when they are drawn side by side. For this purpose we keep track of two values, *totsep* and *currsep*. The variables *totsep* and *currsep* hold the total distance between the roots and the distance between the rightmost node of *A* and the leftmost node of *B* at the current level. To calculate *totsep* and *currsep*, we start at level 0 and visit each level of the trees until we reach the bottommost level of the smaller tree; this is *A* in our example.

At level 0, the distance between the roots of *A* and *B* should be at least *minsep*. Therefore, we set *totsep* := *minsep* + *rtop*(*A*) + *ltop*(*B*) and *currsep* := *minsep*. Using *roff*(*A*) and *loff*(*B*), we can calculate *currsep* for the next level. If *currsep* < *minsep*, we have to increase *totsep* by the difference and update *currsep*. This process is repeated until we reach the lowest level of *A* at which point *totsep* holds the final distance between the nodes of *A* and *B*, as calculated by the RT algorithm. If the root of *C* is a significant node, then the additional space, which is 0pt by default, is added to *totsep*. However, the approach of synthesizing drawings from simple graphics characters allows only a finite

number of orientations for the tree edges; therefore, *totsep* must be increased slightly to fit the next orientation available.

Now we are ready to build the box of TEXtree *C*. Simply put *A* and *B* side by side, with the reference points *totsep* units apart, insert a new node above them, and connect the parent and children by edges. Next, we compute the additional information for *C*. This can be done by using the additional information for *A* and *B*. Note that most components of *roff*(*C*) and *lroff*(*C*) are the same as in the higher tree, which is *B* in our case. So, if we can avoid moving this information around, the number of counters we have to access to update the additional information for *C* is within a small constant of the height of *A*. Hence, we can apply the same argument as in [4], which gives us a running time of O(*N*) for drawing a tree with *N* nodes.

We must design the allocation of storage registers for the additional information of TEXtrees carefully to fulfill the following requirement. If a new tree is built from two sub-trees, the additional information of the new tree shares storage with its larger sub-tree. Organizational overhead, that is, pointers that keep track of the locations of different parts of additional information, must be avoided. This means that the additional information for one TEXtree should be stored in a sequence of consecutive dimension registers such that only one pointer for access to the first element in this sequence is needed. On the other hand, each parent tree is higher and, therefore, needs more storage than its sub-trees. So we must ensure that there is always enough space in the sequence for more information.

The obvious way to fulfill these requirements is to use a stack and to allow only the topmost TEXtrees of this stack to be combined into a larger tree at any time. This leads to the following allocation of registers: a contiguous sequence of box registers contains the tree-boxes of the sub-trees in the stack. A contiguous sequence of token registers contains the type information for the nodes of the sub-trees in the stack. For each sub-tree in the stack, a contiguous sequence of dimension registers contains the contour information of the sub-tree. The ordering of these groups of dimension registers reflects the ordering of the sub-trees in the stack. Finally, a contiguous sequence of counter registers contains the height and the address of the first dimension register for each sub-tree in the stack. Four address counters store the addresses of the last tree-box, type information, height, and address of contour information. A sketch of the register organization for a stack of TEXtrees is provided in Figure 8.

When a new node is pushed onto the stack, the tree-box, type information, height, address of contour information, and contour information are stored in the next free registers of the appropriate type, and the four address counters are updated accordingly.

When a new tree is formed from the topmost sub-trees on the stack, the tree-box, type information, height, and address of contour information of the new tree are stored in the registers formerly used by the bottommost sub-tree that has occurred in the construction step, and the four address registers are updated accordingly. This means that this information for the sub-trees is no longer accessible. The contour information of the new sub-tree is stored in the same registers as the contour information of the larger sub-tree used in the construction, apart from the left and right offset of the root to the left and right child, which are stored in the following dimension registers. This means that gaps can occur between the contour information of sub-trees in the stack, namely when the right sub-tree, which is in a higher position in the stack, is higher than the left one. To avoid these gaps, the user can specify an option `\lefttop` when entering a binary node, which makes the topmost tree in the stack the left sub-tree of the node.

Dimension registers
lmoff(1) *rmoff*(1) *lboff*(1) *rboff*(1) *ltop*(1) *rtop*(1)
loff(h_1) *roff*(h_1) ... *loff*(1) *roff*(1)
...
lmoff(*n*) *rmoff*(*n*) *lboff*(*n*) *rboff*(*n*) *ltop*(*n*) *rtop*(*n*)
loff(h_n) *roff*(h_n) ... *loff*(1) *roff*(1)

Counter registers
lasttreebox lasttreeheight lasttreeinfo lasttreetype
treeheight(1) *diminfo*(1) ... *treeheight*(*n*) *diminfo*(*n*)

Box registers
treebox(1) ... *treebox*(*n*)

Token registers
type(1) ... *type*(*n*)

Figure 8. lasttreebox, lasttreeheight, lasttreeinfo, lasttreetype contain pointers to treebox(n) treeheight(n), lmoff(n), type(n), diminfo(i) contains a pointer to lmoff(i). Unused dimension registers are allowed between the dimension registers of subsequent trees. The counter registers lasttreebox,..., diminfo(n) serve as a directory mechanism to access the TEXtrees on the stack

This stack concept also has consequences for the design of the user interface that is discussed in Section 9.

8 SPACE COST ANALYSIS

Suppose we want to draw a unary–binary tree T of height h having N nodes.[2] According to our internal representation, for each sub-tree in the stack we need:

1. one box register to store the box of the TEXtree;
2. one token register to store the type of the root of the sub-tree;
3. $2h'+6$ dimension registers to store the additional information, where h' is the height of the sub-tree; and
4. three counter registers to store the register numbers of the box register, the token register, and the first dimension register above.

Lemma 8.1

Let T be a unary–binary tree of height h and size N; then:

1. *at any time, there are at most $h+1$ sub-trees of T on the stack; and*
2. *for each set $\mathcal{T}$ of sub-trees of T that are on the stack simultaneously we have*

$$\sum_{T' \in \mathcal{T}} (\mathrm{ht}(T') + 1) \leq N$$

[2] The height h and the number of nodes N refer to the drawing of the tree. N is the number of circles, squares, etc., actually drawn, and h is the number of levels in the drawing minus 1.

The lemma implies that our implementation uses at most $9h+2N$ registers. To compare this with the $10N$ registers used in the straightforward implementation, an estimation of the average height of a tree with N nodes is needed. Several results, depending on the type of trees and of the randomization model, are cited in Figure 9, which compares the number of registers used in a straightforward implementation with the average number of registers used in our implementation. This table shows clearly the advantage of our implementation.

nodes N	registers (straight-forward)	average registers: binary $(2\sqrt{\pi N})$ [6]	average registers: unary–binary trees $(\sqrt{3\pi N})$ [6]	average registers: binary search trees $(4.311 \log N)$ [5]
10	100	120.89	107.37	109.34
20	200	182.68	163.56	156.23
30	300	234.75	211.33	191.96
40	400	281.78	254.75	223.12
50	500	325.60	295.37	251.78
60	600	367.13	334.02	278.86
70	700	406.93	371.17	304.84
80	800	445.36	407.13	330.02
90	900	482.67	442.12	354.59
100	1000	519.04	476.30	378.68

Figure 9. The numbers of registers used by a straightforward implementation (second column) and by our modified implementation (third to fifth column) of the RT algorithm are given for different types of trees and randomization models. The formulas in parentheses indicate the average height of the respective classes of trees

9 THE USER INTERFACE

The user interface of TreeTEX has been designed in the spirit of the thorough separation of the logical description of document components and their layout; see [10,11]. This concept ensures both uniformity and flexibility of document layout and frees authors from layout problems that have nothing to do with the substance of their work. For some powerful implementations and projects see [8,12–15].

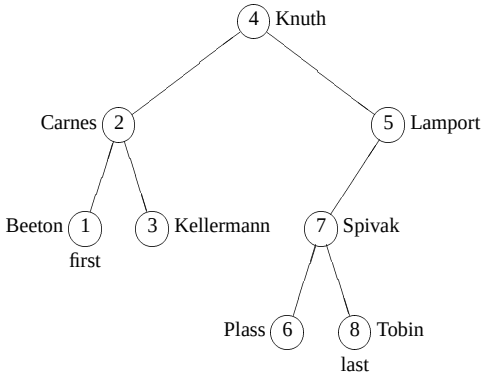
The description of a tree consists of a description of its nodes in post-order. Each description of a node, in turn, has to specify the out-degree, the geometric shape and the labels of the node. Defaults are provided for all specifications, thereby allowing the user to omit many definitions if the defaults match what he or she wants.

A separate style command defines layout parameters for tree drawings that are valid for all trees of a document. Layout parameters include the font to be used for labels, the diameter of circle nodes, the vertical distance between two subsequent levels of the tree, and the minimal horizontal distance between nodes.

Standard versions of TEX provide only a limited number of font and circle sizes. Hence,

the user of the style command must make sure that the specified sizes can be realized. This is especially cumbersome when everything has to be magnified for later reproduction with reduction. But the style variables can be made parametric for installations that provide scalable fonts and replace LaTeX's circle- and line-drawing commands with routines that provide arbitrary diameters and slopes.

Three examples of tree descriptions are given in Figures 10–12. A more detailed description of the user interface is given in [16].




```
\begin{Tree}
\node{\external\bnth{first}\cntr{1}\lft{Beeton}}
\node{\external\cntr{3}\rght{Kellermann}}
\node{\cntr{2}\lft{Carnes}}
\node{\external\cntr{6}\lft{Plass}}
\node{\external\bnth{last}\cntr{8}\rght{Tobin}}
\node{\cntr{7}\rght{Spivak}}
\node{\leftonly\cntr{5}\rght{Lamport}}
\node{\cntr{4}\rght{Knuth}}
\end{Tree}

\hspace{\leftdist}\usebox{\TeXTree}\hspace{\rightdist}
```

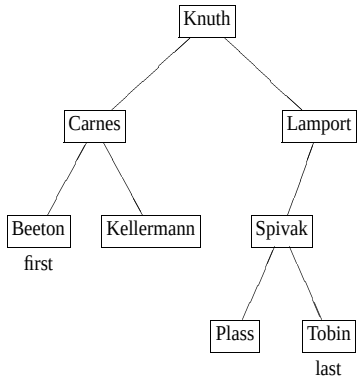
Figure 10. This is an example of a tree that includes labels

10 CONCLUSIONS

We hope that, by now, we have convinced the reader of the main advantages of TreeTEX: It integrates graphics and text; it is portable to all sites running TEX; and it is easy to use for the author, because it derives the drawing of a tree from a purely structural description. But our decision to implement TreeTEX as a TEX macro package has also some drawbacks, both for the programmer and for the user of the system.

From the programmer's point of view, TEX's macro language is a low level programming language. Hence, maintaining and extending the package is a more tedious task than it would be if we had used a higher level language with better support for modularization.

From the author's point of view, TreeTEX's limitations lie in speed, size of trees, and graphical primitives. Typesetting all the trees in this article takes about two minutes on a VAX 750, and typesetting a complete binary tree with 63 internal and 64 external nodes




```
\begin{Tree}
\node{\external\type{frame}\bnth{first}\cntr{Beeton}}
\node{\external\type{frame}\cntr{Kellermann}}
\node{\type{frame}\cntr{Carnes}}
\node{\external\type{frame}\cntr{Plass}}
\node{\external\type{frame}\bnth{last}\cntr{Tobin}}
\node{\type{frame}\cntr{Spivak}}
\node{\leftonly\type{frame}\cntr{Lamport}}
\node{\type{frame}\cntr{Knuth}}
\end{Tree}

\hspace{\leftdist}\usebox{\TeXTree}\hspace{\rightdist}
```

Figure 11. This is an example of a tree with framed centre labels

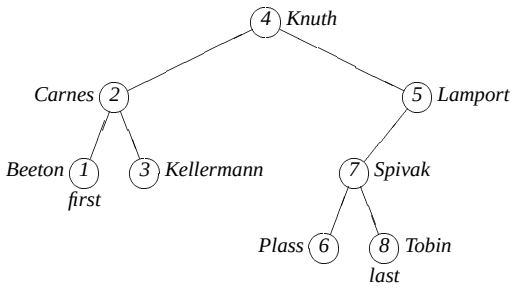


Figure 12. This tree was produced from the same logical description as in Figure 10, but with different style parameters

takes about one minute on the same machine. The size of the trees is limited by three factors, namely, the number of registers, the complexity of the nested boxes that contain the drawing of a tree, and the limited number of slopes that are available for the edges, the latter being the most severe problem at present. Hence, the main area of application for TreeTEX is modest use such as in textbooks; displaying large amounts of statistical data, for example, is out of the question.

Currently edges and circular nodes are drawn from LATEX's set of predefined graphical characters. Hence, TreeTEX cannot draw arbitrarily wide trees or large circular nodes.

We consider this restriction, however, to be a temporary one, since a committee inside the TEX Users Group is working on standard graphic extensions to TEX that will remove these limitations.

As to further developments of TreeTEX, it would be desirable to draw larger classes of trees, for example multiway trees, and to allow labels not only for nodes, but also for edges and whole sub-trees.

ACKNOWLEDGEMENTS

This work was supported by a Natural Sciences and Engineering Research Council of Canada Grant A-5692, a Deutsche Forschungsgemeinschaft Grant Sto167/1-1, and a grant from the Information Technology Research Centre. It was begun during the first author's stay with the Data Structuring Group in Waterloo.

REFERENCES

1. R. A. Baeza-Yates, 'On embedding a binary tree on a hypercube', submitted for publication, November 1988.
2. R. Klein and D. Wood, 'On binary trees', in *Proceedings of the 11th World Computer Congress*, ed. G. Ritter, San Francisco, USA, 1989, to appear.
3. Th. Ottmann and P. Widmayer, *Algorithmen und Datenstrukturen*. Bibliographisches Institut, Mannheim, 1989, to appear.
4. E. M. Reingold and J. S. Tilford, 'Tidier drawings of trees', *IEEE Transactions on Software Engineering*, **7**(2), 223–228 (1981).
5. D. E. Knuth, *Fundamental Algorithms, volume 1 of The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, 1973.
6. Ch. Wetherell and A. Shannon, 'Tidy drawings of trees', *IEEE Transactions on Software Engineering*, **5**(5), 514–520 (1979).
7. K. J. Supowit and E. M. Reingold, 'The complexity of drawing trees nicely', *Acta Informatica*, **18**(4), 377–392 (1983).
8. L. Lamport, LATEX, *User's Guide & Reference Manual*. Addison-Wesley, Reading, Massachusetts, 1986.
9. D. E. Knuth, *The TEXbook, volume A of Computers & Typesetting*. Addison-Wesley, Reading, Massachusetts, 1986.
10. R. Furuta, J. Scofield and A. Shaw, 'Document formatting systems: surveys, concepts, issues', *Computing Surveys*, **14**(3), 417–472 (1982).
11. Ch. F. Goldfarb, 'A generalized approach to document markup', *SIGPLAN Notices of the ACM*, **16**(6), 68–73 (1981).
12. R. J. Beach, *Setting Tables and Illustrations with Style*. PhD thesis, University of Waterloo, 1985.
13. A. Brüggemann-Klein, P. Dolland and A. Heinz, 'How to please authors and publishers: a versatile document preparation system at Karlsruhe, in *TEX for Scientific Documentation*, ed. J. Désarménien, Springer-Verlag Lecture Notes in Computer Science 236, Strasbourg, France, June 1986, pp.8–31.
14. V. Quint, I. Vatton, and H. Bedor, 'Grif: an interactive environment for TEX', in *TEX for Scientific Documentation*, ed. J. Désarménien, Springer-Verlag Lecture Notes in Computer Science 236, Strasbourg, France, June 1986, pp.145–158.
15. B. K. Reid, *Scribe: A Document Specification Language and its Compiler*. PhD thesis, Carnegie Mellon University, 1980.
16. A Brüggemann-Klein and D. Wood, 'Drawing trees nicely with TEX', in *Proceedings of the Third European TEX-Conference*, ed. M. Clark, Exeter, England, July, 1988, to appear.
17. L. Devroye, 'A note on the height of binary search trees', *Journal of the ACM*, **33**(3), 489–498 (1986).
18. Ph. Flajolet and A. Odlyzko, 'The average height of binary trees and other simple trees', *Journal of Computer and System Sciences*, **25**, 171–213 (1982).