Drawing Trees Nicely with TEX

Anne Brüggemann-Klein
Derick Wood
Data Structuring Group

Research Report
CS-87-05

February, 1987

Drawing Trees Nicely with TEX[*]

Anne Brüggemann-Klein[†] Derick Wood[‡]

February 10, 1987

Abstract

Various algorithms have been proposed for the difficult problem of producing aesthetically pleasing drawings of trees, see [14,16] but implementations only exist as "special purpose software", designed for special environments. Therefore, many users resort to the drawing facilities available on most personal computers, but the figures obtained in this way still look "hand-drawn"; their quality is inferior to the quality of the surrounding text that can be realized by today's high quality text processing systems.

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 present a TEX macro package TreeTEX that produces a drawing of a tree from a purely logical description. Our approach has three advantages. First, *labels* for nodes can be handled in a reasonable way. On the one hand, the tree drawing algorithm can compute the widths of the labels and take them into account for the positioning of the nodes; on the other hand, all the textual parts of the document can be treated uniformly. Second, TreeTEX can be trivially ported to any site running TEX. Finally, modularity in the description of a tree and TEX's macro capabilities allow for libraries of subtrees and tree classes.

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

[*]This work was supported by a Natural Sciences and Engineering Research Council of Canada Grant A-5692. It was done during the first author's stay with the Data Structuring Group in Waterloo in 1986/87

[†]Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Universität Karlsruhe, Postfach 6980, 7500 Karlsruhe, West Germany

[‡]Data Structuring Group, Department of Computer Science, University of Waterloo, Waterloo, Ontario, N2L 3G1, Canada

1 Aesthetical criteria for drawing trees

One of the most commonly used data structures in computer science is the tree. As many people are using trees in their research or just as illustration tools, they are usually struggling with the problem of *drawing* trees. We are concerned primarily with ordered trees in the sense of [9], especially binary and unary-binary trees. A binary tree is a finite set of nodes which either is empty, or consists of a root and two disjoint binary trees called the left and right subtrees of the root. A unary-binary tree is a finite set of nodes which either is empty, or consists of a root and two disjoint unary-binary trees, or consists of a root and one nonempty unary-binary tree. An extended binary tree is a binary tree in which each node has either two nonempty subtrees or two empty subtrees.

For these trees there are some basic agreements on how they should be drawn, reflecting the top-down and left-right ordering of nodes in a tree; see [14] and [16].

1. Trees impose a distance on the nodes; no node should be closer to the root than any of its ancestors.

2. Nodes of a tree at the same height 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 centered 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 [16], 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 [14] point out, there is a lack of symmetry in the algorithms of Wetherell and Shannon which may lead to unpleasant results; see Figure 1. Therefore, Reingold and Tilford introduce a new structured axiom:

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

Axiom 7 allows the same tree to be drawn differently when it occurs as a subtree in different trees. Reingold and Tilford give an algorithm which fulfills axioms 1–5 and 7. Although this algorithm is far from fulfilling also axiom 6, see [15], the aesthetical improvements are well worth the additional space; see Figure 2.

2 The algorithm of Reingold and Tilford

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 subtree are calculated independently from the rest of the tree. After the relative positions of two subtrees have been calculated, they can be joined as siblings in a larger tree by placing them as close together as possible and centering the parent node above them. Incidentally, the modularity principle is the reason that the algorithm fails to fulfill axiom 6; see [15]. Two sibling subtrees are placed as close as possible, during a postorder traversal, as follows. At each node T, imagine that its two subtrees have been drawn and cut out of paper along their contours. Then, starting with the two subtrees 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 lower level, they are pushed apart until the minimum separation is established there. This process is continued at successively lower levels until the bottom of the shorter subtree is reached. At some levels no movement may be necessary; but at no level are the two subtrees moved closer together. When the process is complete, the position of the subtrees is fixed relative to their parent, which is centered over them. Assured that the subtrees will never be placed closer together, the postorder traversal is continued.

A nontrivial implementation of this algorithm has been obtained by Reingold and Tilford 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 subtrees by special pointers, called threads, such that whenever two subtrees are joined, only the top part of the trees down to the lowest level of the *smaller* tree need to be taken into account.

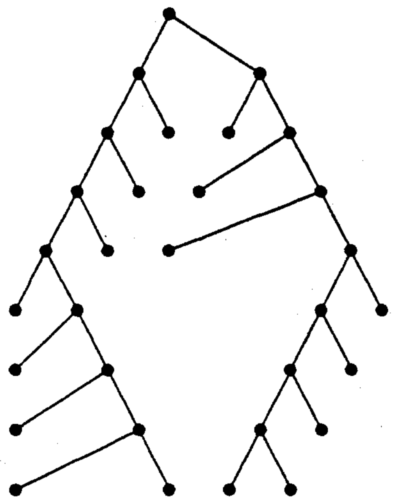


Figure 1: A tree as drawn by the algorithm of Wetherell and Shannon

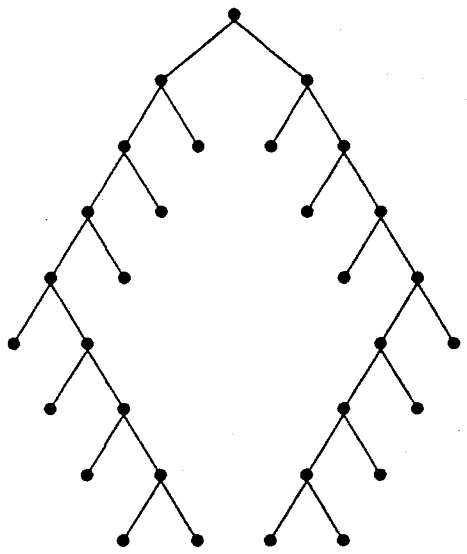


Figure 2: The same tree as drawn by the algorithm of Reingold and Tilford

The RT algorithm is given by a PASCAL procedure, using recursion for the positioning of nodes in a subtree. The trees themselves are given as a pointer structure. The nodes are positioned on a fixed grid and are considered to have zero width. No labelling is provided. The algorithm only draws binary trees, but is easily extendable to multiway trees.

3 Improving human perception of trees

It is a common understanding in book design that aesthetics and readability don't necessarily coincide. A good example for this is [11]. At first glimpse, some pages look rather ugly with their large margins and occasional examples sticking out into them; see pages 66 and 67, for example. One of us remembers well the feeling of dissappointment when thumbing through it for the first time: such a strange result by such a beautiful system like TEX! But the design of these examples proved to be very useful when actually reading the book, and after a time, the irregular shapes were of no further concern.

There is a similar phenomenon with tree drawings. Although the RT algorithm renders aesthetically very pleasant results, in some examples the *structure* of the trees cannot be recognized easily; see Figure 3, for example. In this example, the origin of the long parallel branches is easily lost.

Therefore, we add an optional feature to the tree drawing algorithm which, by widening the drawing at significant nodes, makes it easier to perceive the structure, that is, extra horizontal distance can be inserted between two subtree siblings if their closest level is *not* their root level; see Figure 3 and Figure 4. On the other hand, an unextended binary tree can be drawn with the same placement of nodes as its associated extended version. We define the *associated extended version* of a binary tree to be the binary tree obtained by replacing each empty subtree having a nonempty sibling with a subtree consisting of one node. Note that this is not the usual definition; see [9], for example. This makes it easier to recognize the history of a tree; see Figure 5 and Figure 6.

4 Trees in a document preparation environment

Drawings of trees usually don't come alone, but are included in some text which is itself produced by a text processing system. Therefore, a typical scenario is a pipe of three processes. First comes the tree drawing program which calculates the positioning of the nodes of the tree to be drawn and outputs a description of the tree drawing in some graphics language; next comes a graphics system which transforms this description into an interme-

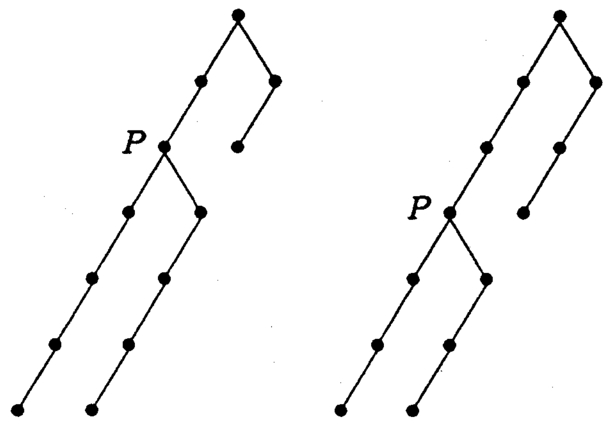


Figure 3: Trees as drawn by the RT algorithm

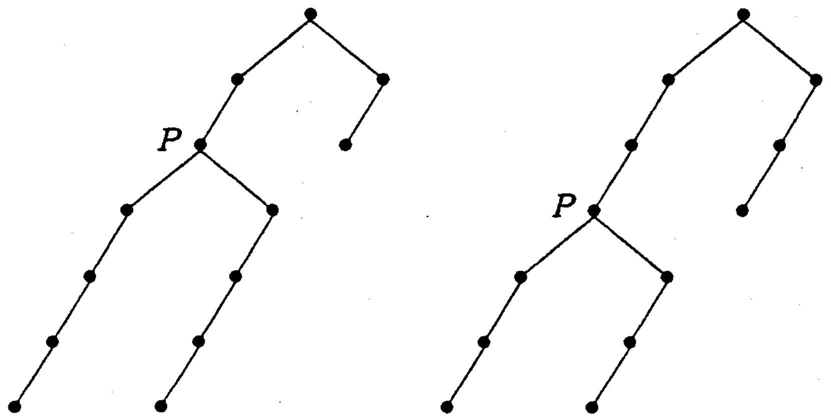


Figure 4: Trees drawn with additional space between the subtrees of node P.

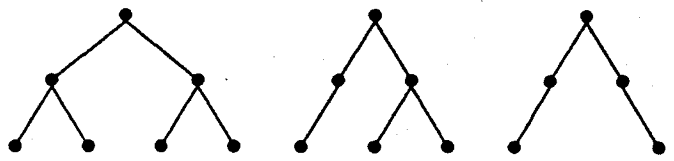


Figure 5: Trees as drawn by the RT algorithm

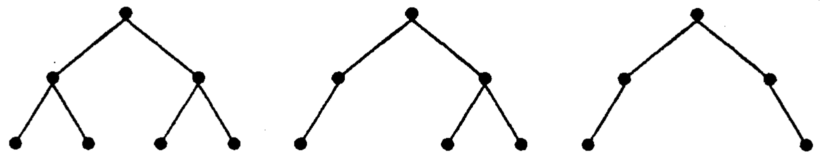


Figure 6: Trees drawn as extended binary trees

diate language which can be interpreted by the output device; and finally comes the text processing system which integrates the output of the graphics system into the text; see Figure 7.

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), and their extensions certainly should be taken into account by the tree drawing algorithm. As soon as the widths of the labels are known, it is straightforward to extend the tree drawing algorithms for labelled nodes of nonzero width. But the widths of the labels are usually calculated either by the graphics system or by the text processing system, which provides these data for the graphics system. This leads to mutual dependencies in our system; see Figure 8 and Figure 9. The latter method is preferred since it makes for uniformity in the textual parts of the document. However, in practice, this is often not the case; for example, with PIC and TROFF.

Therefore, our task would be easier if we had a text processing system having powerful enough programming facilities to program a tree drawing algorithm and powerful enough graphics facilities to draw a tree. One text processing system rendering outstanding typographic quality and good enough programming facilities is TEX, developed by Knuth at Stanford University; see [10]. The TEX system includes the following programming facilities:

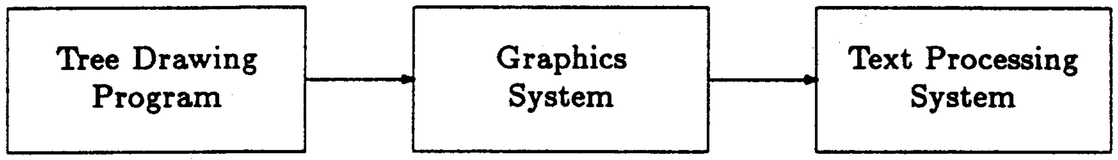


Figure 7: Interaction of a tree drawing and text processing program (1)

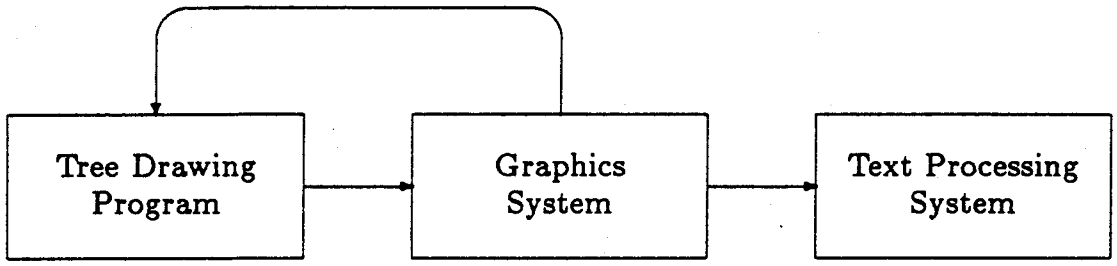


Figure 8: Interaction of a tree drawing and text processing program (2)

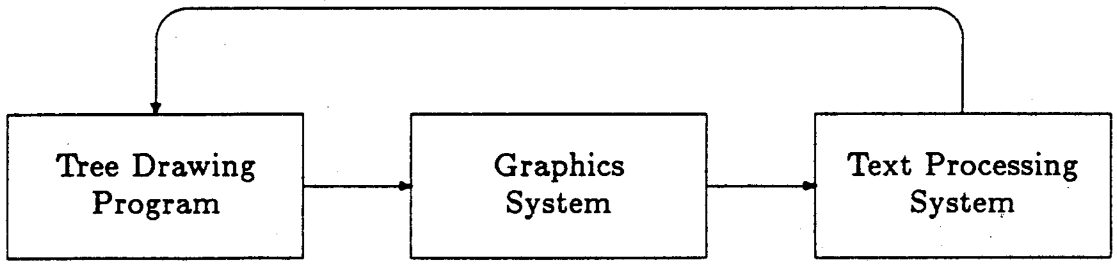


Figure 9: Interaction of a tree drawing and text processing program (3)

1. datatypes:
   integers (256), dimensions[1] (512), boxes (256), tokenlists (256), boolean variables (unrestricted)

2. elementary statements:
   $a :=$ const, $a := b$ (all types);
   $a := a + b$, $a := a * b$, $a := a/b$ (integers and dimensions);
   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 concept of local variables).

Therefore, the programming facilities of TEX seem to be powerful enough to handle a tree drawing algorithm. 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 [11]. 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 the aesthetically most pleasing results. In the first version, we restrict ourselves to unary-binary trees, although our method is applicable to arbitrary multiway trees. But in order 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 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 which produces device *independent* output.

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

5 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 the following fields:

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
5. one token to store the geometric shape of the node.

Because these data are used very frequently in calculations, they should be stored in registers, rather than being recomputed, in order to obtain reasonably fast performance. This gives a total of $10N$ registers for a tree with N nodes. Because of TEX's limited supply of registers, we prefer a slightly different implementation, based on the following observation. Suppose a unary-binary tree is constructed bottom-up, in a postorder traversal. This is done by iterating the following three steps in an order determined by the tree to be constructed.

1. Create a new subtree consisting of one external node.
2. Create a new subtree by appending the last two created subtrees to a new binary node; see Figure 10.
3. Create a new subtree by appending the last created subtree as a left, right, or unary subtree of a new node; see Figure 11.

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

The same tree traversal is used in the RT algorithm, and at each execution of step 2 or step 3 the relative positions of the subtree(s) and the new node are computed. However, this calculation doesn't need complete information about the subtrees, but only their contour, that is, the offsets

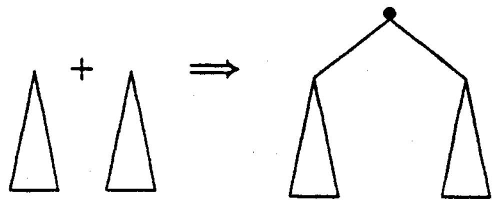


Figure 10: Construction step 2

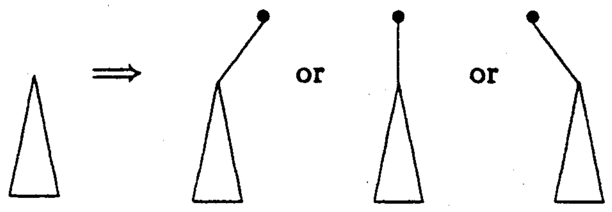


Figure 11: Construction step 3

of the outermost nodes of each level. Complete information is only needed in a second traversal, when the tree is actually drawn, using the coordinates which have been computed before. But, unlike Pascal, TEX provides the capability of storing a drawing in a single box register and positioning it freely in later drawings. Therefore, we only have to keep track of the contours of the subtrees on the stack and their drawings.

6 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; see Figure 12 and Figure 13. 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 some registers for numbers and dimensions and is needed in order to put subtrees together to form a larger tree. *loff* is an array of dimensions which 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. *lmoff* holds the horizontal offset between the root and the leftmost node of the whole tree. *lboff* holds the horizontal offset between the root and the leftmost node at the bottom level of the tree. Finally, *ltop* holds the distance between the reference point of the tree and the leftmost end of the root. The same is true for *roff*, *rmoff*, *rboff*, and *rtop*; just replace "left" by "right". Finally, *height* holds the height of the tree, and *type* holds the geometric shape of the node of the tree.

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 subtrees? An example is given in Figure 12 and Figure 13.

First we determine which tree is higher; this is B in the example. Then we have to compute the minimal 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. In order to calculate *totsep* and *currsep*, we start at level 0 and visit each level of the trees until we reach the bottom 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 proceed to calculate *currsep* for the next level. If $currsep < minsep$, we have to increase *totsep* by the difference and update *currsep*. This process is iterated until we reach the lowest level of A. Then *totsep* holds the final distance between the nodes of A and B, as calculated by the RT algorithm. 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 construct the box of TEXtree C. Simply put A and B side by side, the reference points *topsep* units apart, insert a new node above them, and connect the parent and children by edges.

Next, we update 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, we only have to access $height(A) + const$ many counters in order to update the additional information for C. This implies that we can apply the same argument as in [14], which gives us a running time of $O(N)$ for drawing a tree with N nodes.

Therefore, we must carefully design the storage allocation for the additional information of TEXtrees in order to fulfill the following requirements: If a new tree is built from two subtrees, the additional information of the new tree should share storage with its larger subtree. Organizational overhead, that is, pointers which keep track of the locations of different parts of additional information, must be avoided. This means that all the additional

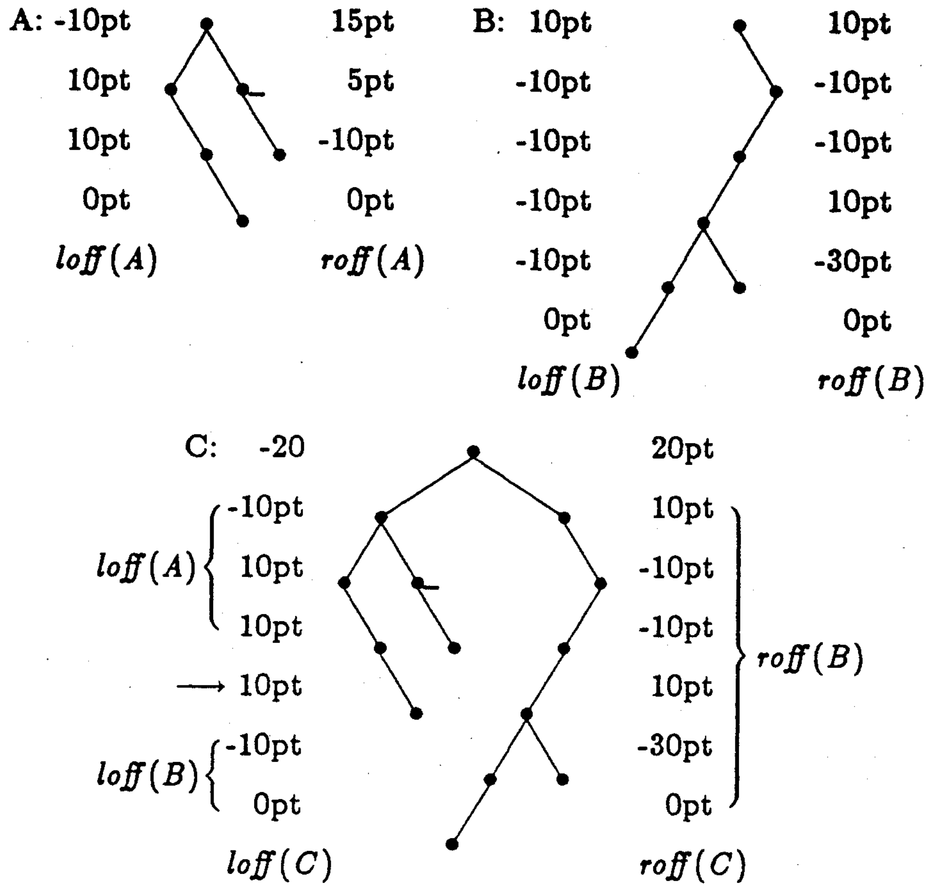


Figure 12: Tree construction

	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	24pt	0/20pt
1	29pt	15/20pt
2	34pt	15/20pt
3	34pt	20/20pt

Figure 13: Additional Information for trees *A*, *B*, and *C* and the history of computation for *totsep* and *currsep*

Dimension registers
lmoff(1) *rmoff*(1) *lboff*(1) *rboff*(1) *ltop*(1) *rtop*(1)
loff(h_1) *roff*(h_1) ... *loff*(0) *roff*(0)
...
lmoff(n) *rmoff*(n) *lboff*(n) *rboff*(n) *ltop*(n) *rtop*(n)
loff(h_n) *roff*(h_n) ... *loff*(0) *roff*(0)

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)

lasttreebox, *lasttreeheight*, *lasttreeinfo*, *lasttreetype* contain pointer 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.

Figure 14: Register organization for TEXtrees

information for one TEXtree should be stored in a row of consecutive counters such that only one pointer to point to the first element in this row is needed. On the other hand, each parent tree is higher and therefore needs more storage than its subtrees. So we must ensure that there is always enough space in the row for more information.

The obvious way to fulfill these requirements is to use a stack and only allow the topmost TEXtrees of this stack to be combined into a larger tree at any time; see Figure 14. This leads to the following register allocation: A subsequent number of box registers contains the treeboxes of the subtrees in the stack. A subsequent number of token registers contains the type information for the nodes of the subtrees in the stack. For each subtree in the stack, a subsequent number of dimension registers contains the contour information of the subtree. The ordering of these groups of dimension registers reflects the ordering of the subtrees in the stack. Finally, a subsequent number of counter registers contains the height and the address of the first dimension register for each subtree in the stack. Four address counters store the addresses of the last treebox, type information, height, and address of contour information.

When a new node is pushed onto the stack, the treebox, 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 subtrees on the stack, the treebox, type information, height, and address of contour information of the new tree are sorted in the registers formerly used by the bottommost subtree used in the construction step, and the four address registers are updated accordingly. This means that these informations for the subtrees are no longer accessible. The contour information of the new subtree is stored in the same registers as the contour information of the larger subtree 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. That means that gaps can occur between the contour informatin of subsequent subtrees in the stack, namely when the right subtree, which is on a higher position on the stack, is higher than the left one. In order 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 subtree of the node.

This stack concept also has consequences for the design of the user interface that is discussed in Section 8.

7 Space cost analysis

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

1. one box register to store the box of the TEXtree
2. one token register to store the type of the root of the subtree
3. $2h' + 8 = 2(h' + 1) + 6$ dimension registers to store the additional information, where h' is the height of the subtree
4. three counter registers to store the register numbers for the box register, token register, and first dimension register above.

The following lemma relates the number of subtrees of T which are on the stack simultaneously and their heights to h and N.

Lemma 7.1

1. *At any time, there are at most $h+1$ subtrees of T on the stack.*
2. *For each set $\mathcal{T}$ of subtrees of T which are on the stack simultaneously we have*

$$\sum_{T' \in \mathcal{T}} (\mathrm{ht}(T')+1) \leq \min(N, \frac{(h+1)(h+2)}{2}+1).$$

Proof

1. By induction.
2. The trees in $\mathcal{T}$ are pairwise disjoint, and each tree of height h' has at least $h'+1$ nodes. This implies

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

The second part is shown by induction over h. The basis $h=0$ is clear. Assume the assumption holds for all trees of height less than h. If $\mathcal{T}$ contains only subtrees of either the left or the right subtree of T, we have

$$\sum_{T' \in \mathcal{T}} (\mathrm{ht}(T')+1) \leq \frac{h(h-1)}{2}+1 \leq \frac{h(h+1)}{2}+1.$$

Otherwise, $\mathcal{T}$ contains the left or the right subtree T_s of T. Then all elements of $\mathcal{T} - \{T_s\}$ belong to the other subtree. This implies

$$\begin{aligned}\sum_{T' \in \mathcal{T}} (\mathrm{ht}(T')+1) &\leq \mathrm{ht}(T_s)+1+\sum_{T' \in \mathcal{T}-\{T_s\}} (\mathrm{ht}(T')+1)\\ &\leq h+\frac{h(h-1)}{2}+1=\frac{h(h+1)}{2}+1. \qquad \square\end{aligned}$$

Therefore, our implementation uses at most $11h+2\min(N,(h+1)(h+2)/2)$ registers. In order to compare this with the $10N$ registers used in the straightforward implementation, an estimation of the average height of a

nodes	registers (straightforward implementation)	average registers		
		extended binary trees	unary-binary trees	binary search trees
5	50	53.60	85.51	120.11
6	60	59.76	94.72	134.58
17	170	114.39	173.23	227.83
18	180	118.72	179.27	233.75
28	280	159.17	234.69	283.97
29	290	162.99	239.85	288.37
40	400	203.31	293.58	332.38
50	500	237.86	338.78	367.64
60	600	271.02	381.58	400.12

Figure 15: Number of registers used for drawing trees with N nodes and height h

tree with N nodes is needed. Some results, depending on the type of trees and randomization model, are cited from the literature.

Lemma 7.2

1. *The average height of a random extended binary tree with n nodes is asymptotically equal to $\sqrt{\pi n}$.*
2. *The average height of a random unary-binary tree with n nodes is asymptotically equal to $\sqrt{3\pi n}$.*
3. *The average height of a random binary search tree with n nodes is asymptotically equal to $4.31107\ldots\log n$.*

Proof

1. By de Brujin, Knuth, and Rice [4].
2. By Flajolet and Odlyzko [6].
3. By Devroye [5]. □

Figure 15 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 advantages of our implementation.

8 The user interface

8.1 General design considerations

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 [7,8]. This concept ensures both uniformity and flexibility of document layout and frees authors from layout problems which have nothing to do with the substance of their work. For some powerful implementations and projects see [1,2,11,12,13].

In this context, the description of a tree is given in a purely logical form, and layout variations are defined by a separate style command which is valid for all trees of a document.

A second design principle is to provide defaults for all specifications, thereby allowing the user to omit many definitions matching these defaults.

The node descriptions of a tree must be entered in postorder. This best fits the internal representation of TEXtrees. Although this is a natural method of describing a tree, users might prefer more flexible description methods. However, note that instances of well defined tree classes can be described easily by TEX macros. We give examples of macros for complete binary trees and Fibonacci trees.

TreeTEX uses the picture making macros of LATEX. If TreeTEX is used with any other macro package or format, the picture macros of LATEX are included automatically.

8.2 The description of a tree

The description of a tree is started by the command `\beginTree` and closed by `\endTree` (or `\begin{Tree}` and `\end{Tree}` in LATEX). The description of a tree can be started in any mode and defines a box and two dimensions. The box is stored in the box register `\TeXTree` and contains the drawing of the tree. The box has zero height and width, and its depth is the height of the drawing. The reference point is in the center of the node of the tree. The dimensions are stored in the registers `\leftdist` and `\rightdist` and describe the distance between the reference point and the left and right margin of the drawing. These data can be used to position the drawing of the tree.

Note that the TreeTEX macros don't contribute anything to the current page but only store their results in the registers `\TeXTree`, `\leftdist`, and `\rightdist`. It is the user's job to put the drawing onto the page, using the commands `\copy` or `\box` (or `\usebox` in LATEX).

Each matching pair of `\beginTree` and `\endTree` must contain the description for only *one* tree. Descriptions of trees cannot be nested and new registers cannot be allocated inside a matching pair of `\beginTree` and `\endTree`.

Each tree description describes the nodes of the tree in postorder, that is, a tree description is a particular sequence of node descriptions.

A node description, in turn, consists of the macro `\node`, followed by a list of node options, included in braces. The list of node options may be empty. The node options describe the labels, the geometric shape (type), and the outdegree of the node. Default values are provided for all options which are not explicitly specified. The following node options are available:

1. `\lft{<label>}`, `\rght{<label>}`, `\cntr{<label>}`, `\bnth{<label>}`:
   These options describe the labels which are put to the left of, to the right of, in the center of, or beneath the node (the latter only makes sense for external nodes). The arguments of these macros are processed in internal horizontal mode (LR-mode in LATEX), but can consist of arbitrary nested boxes for more sophisticated labels. For each of these options, the default is an empty label.

2. `\external`, `\unary`, `\leftonly`, `\rightonly`:
   These options describe the outdegree of the node using selfexplanatory names. The default is binary (no outdegree option is specified).

3. `\type{<type>}`:
   This option describes the type or geometric shape of the node. `<type>` can have the values `square` or `dot`. The default value is `circle` (no type is specified).

4. `\lefttop`:
   The node option `\lefttop` in a binary node makes the last entered subtree the left child of the node (the right child is the default). This option helps to cut down the number of dimension registers used during the construction of a tree. As a rule of thumb, this option is recommended when the left subtree has a smaller height than the right subtree, that is, in this case the right subtree should be entered before the left one and their parent should be given the option `\lefttop`.

8.3 Macros for classes of trees

Tree descriptions can be produced by macros. This is especially useful for trees which belong to a larger class of trees and which can be specified by some simple parameters. A small library of such macros is provided in the file `TreeClasses.tex`.

1. **`\treesymbol{<node options>}`:**
   This macro produces a triangular tree symbol which can be included in a tree description instead of an external node. Labels for these tree symbols are described as for ordinary nodes. In addition, the options `\lvls{<number>}` and `\slnt{<number>}` are provided. `\lvls` defines the number of levels in the tree over which the triangle extends, and `\slnt` gives the slant of the sides of the triangle, ranging from 1 (minimal) to 24 (maximal). On the other hand, `\treesymbol` does not expand to a tree description, because a tree symbol cannot be built from subtrees, and, on the other hand, it is not a node, because it is allowed to extend over several tree levels and therefore has a longer contour than an ordinary node. `\treesymbol` is an example of how a TreeTEX guru, who knows the internal representation of TEXTrees very well, can implement such a hybrid.

2. `\binary{<bin specification>}`:
   This macro truly expands to a tree description. It produces a complete binary tree, that is, an extended binary tree, where, for a given h, all external nodes appear at level h or $h-1$, and all external nodes at level h lie left of those at level $h-1$. `<bin specification>` consists of the following options: `\no{<number>}` defines the number of internal nodes, with `<number>` greater than 0, and `\squareleaves` produces leaves of type square. Defaults are `\no{1}` and leaves of type `circle`.

3. `fibonacci{<fib specification>}`:
   This macro produces a Fibonacci tree of the specified height. `<fib specification>` allows for the three options `\hght{<number>}`, `\unarynodes`, and `\verb.`. Normally, a Fibonacci tree of height $h+2$ is a binary tree with Fibonacci trees of height h and $h+1$ as left and right subtrees. The option `\unarynodes` means that the Fibonacci tree is augmented by unary nodes such that each two subtree siblings have the same height. Defaults are `\hght{0}`, the unaugmented version of a Fibonacci tree, and external nodes of type `circle`.

8.4 Style options for trees

The TreeTEX macro package includes a style command `\Treestyle{<style option>}` where `<style option>` contains all the parameter settings the user might want to change. Normally, the command `\Treestyle` appears only once at the beginning of the document and the style options are valid for all trees of the document. Think twice before changing the tree style inside a document!

The changes in the style options are global. A `\Treestyle` command changes only the specified style options; non specified options retain the last specified or the default value, respectively. The following style options are available:

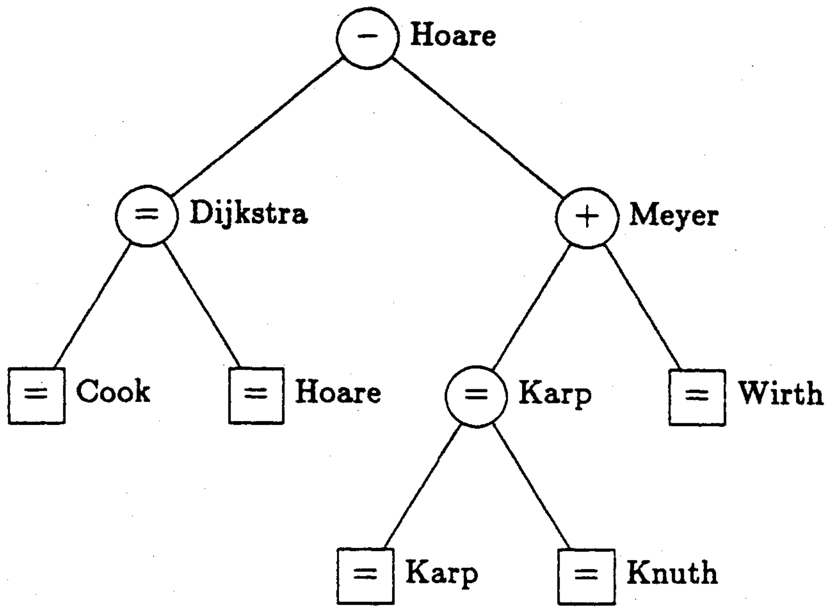
1. `\treefonts{<font options>}`:
   `\treefonts` is invoked by `\beginTree`, and it simply executes whatever is specified in `<font options>`. Defaults are `\treefonts{\tenrm}` (or `\treefonts{\normalsize\rm}` in LATEX).

2. `\nodesize{<size>}`:
   `\nodesize` defines the size of the nodes. `<size>` is a dimension and specifies the diameter of circle nodes. The width of square nodes is adjusted accordingly to be slightly smaller than the diameter of circle nodes in order to balance their appearance. Furthermore, `\nodesize` adjusts the amount of space by which the baseline of the labels is placed beneath the center of the node. The default value of `\nodesize` suits the default of `\treefonts` (taking into account the size option of LATEX's document style).

3. `\vdist{<dimen>}`, `\minsep{<dimen>}`, `\addsep{<dimen>}`:
   `vdist` specifies the vertical distance between two subsequent levels of the tree. Default is `\vdist{60pt}`. `\minsep` specifies the minimal horizontal distance between two adjacent nodes. Default is `\minsep{20pt}`. `\addsep` specifies the additional amount of horizontal space by which two subtree siblings are pushed apart farther than calculated by the RT algorithm, if the level at which they are closest is beneath their root level. Default is `\addsep{0pt}`

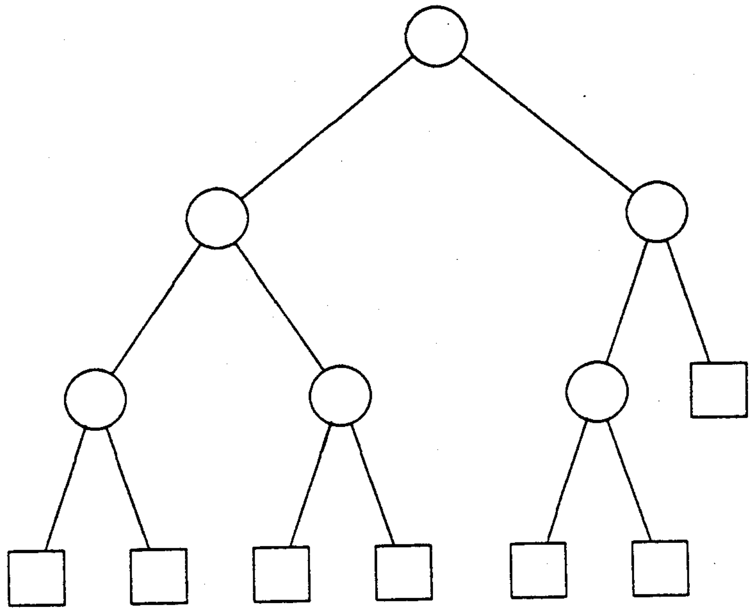
4. `\extended`, `\nonextended`:
   With the option `\extended` in effect, the nodes of a binary tree are placed in exactly the same way as they would be in the associated extended version of the tree (the missing nodes are assumed to have no labels). The default is `\nonextended`, that is the usual layout.

Some examples of tree descriptions are given in Figure 16, Figure 17, and Figure 18. A detailed description of the TreeTEX macros is given in [3].



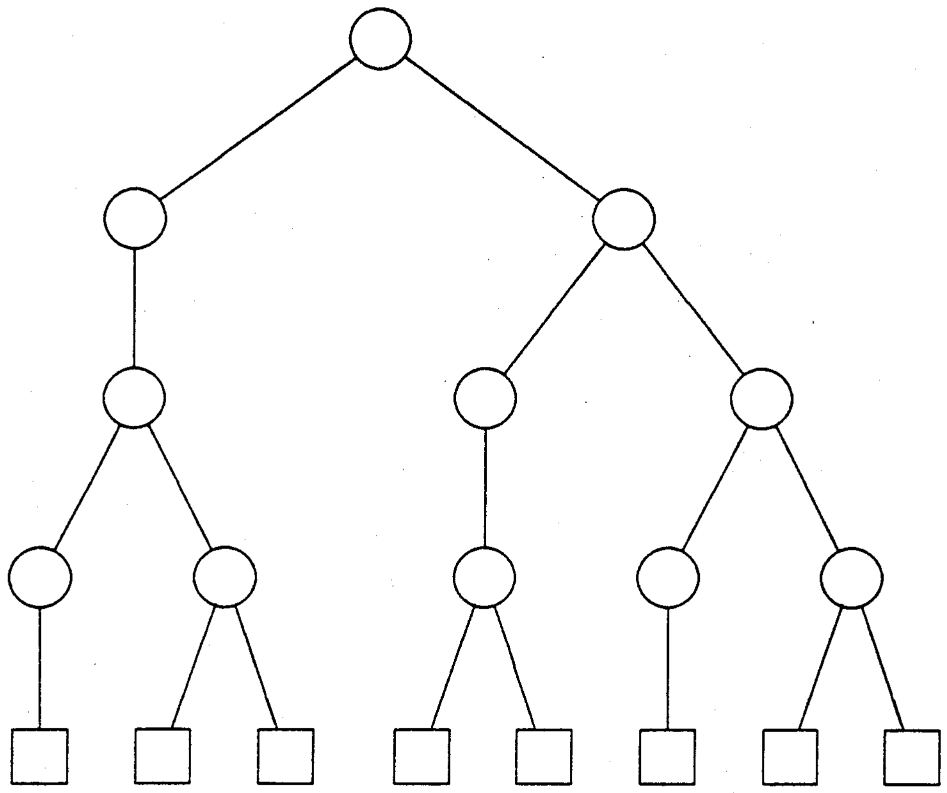
```
\begin{Tree}
\node{\external\type{square}\cntr{$=$}\rght{Cook}}
\node{\external\type{square}\cntr{$=$}\rght{Hoare}}
\node{\cntr{$=$}\rght{Dijkstra}}
\node{\external\type{square}\cntr{$=$}\rght{Karp}}
\node{\external\type{square}\cntr{$=$}\rght{Knuth}}
\node{\cntr{$=$}\rght{Karp}}
\node{\external\type{square}\cntr{$=$}\rght{Wirth}}
\node{\cntr{$+$}\rght{Meyer}}
\node{\cntr{$-$}\rght{Hoare}}
\end{Tree}
\hspace{\leftdist}\usebox{\TeXTree}\hspace{\rightdist}
```

Figure 16: Example of a tree with labels



```
\begin{Tree}
\binary{\no{6}\squareleaves}
\end{Tree}
\usebox{\TeXTree}
```

Figure 17: Example of a complete binary tree



```
\begin{Tree}
\fibonacci{\hght{4}\unarynodes\squareleaves}
\end{Tree}
\usebox{\TeXTree}
```

Figure 18: Example of a Fibonacci tree

References

[1] R. J. Beach. ***Setting Tables and Illustrations with Style.*** PhD thesis, University of Waterloo, 1985.

[2] A. Brüggemann-Klein, P. Dolland, and A. Heinz. How to please authors and publishers: a versatile document preparation system at Karlsruhe. In J. Désarménien, editor, *TeX for Scientific Documentation*, Strasbourg, France, June 1986. LNCS 236.

[3] A. Brüggemann-Klein and D. Wood. *TreeTeX: Documentation and User Handbook.* Technical Report, University of Waterloo, 1987.

[4] N.G. de Bruijn, D. Knuth, and S.O. Rice. The average height of planted plane trees. In R.C. Read, editor, *Graph Theory and Computing*, 1972.

[5] L. Devroye. A note on the height of binary search trees. *Journal of the ACM*, 33(3), July 1986.

[6] Ph. Flajolet and A. Odlyzko. The average height of binary trees and other simple trees. *Journal of Computer and System Sciences*, 25, 1982.

[7] R. Furuta, J. Scofield, and A. Shaw. Document formatting systems: surveys, concepts, issues. *Computing Surveys*, 14(3), 1982.

[8] Ch. F. Goldfarb. A generalized approach to document markup. *SIGPLAN Notices of the ACM*, June 1981.

[9] D. E. Knuth. *Fundamental Algorithms.* Volume 1 of *The Art of Computer Programming*, Addison-Wesley, Reading, Massachusetts, 1973.

[10] D. E. Knuth. *The TeXbook.* Volume A of *Computers & Typesetting*, Addison-Wesley, Reading, Massachusetts, 1986.

[11] L. Lamport. LaTeX, *User's Guide & Reference Manual.* Addison-Wesley, Reading, Massachusetts, 1986.

[12] V. Quint, I. Vatton, and H. Bedor. Grif: an interactive environment for TeX. In J. Désarménien, editor, *TeX for Scientific Documentation*, Strasbourg, France, June 1986. LNCS 236.

[13] B. K. Reid. *Scribe: A Document Specification Language and its Compiler.* PhD thesis, Carnegie Mellon University, 1980.

[14] E. M. Reingold and J. S. Tilford. Tidier drawings of tree. *IEEE Transactions on Software Engineering*, 7(2), March 1981.

[15] K. J. Supowit and E. M. Reingold. The complexity of drawing trees nicely. *Acta Informatica*, 18, 1983.

[16] Ch. Wetherell and A. Shannon. Tidy drawings of trees. *IEEE Transactions on Software Engineering*, 5(5), September 1979.