Cahiers GUTenberg

DRAWING TREE STRUCTURES WITH GWEZ

Bernard LEGUY

Cahiers GUTenberg, nº 10-11 (1991), p. 135-146.

<http://cahiers.gutenberg.eu.org/fitem?id=CG_1991___10-11_135_0>

Drawing tree structures with GWEZ

Bernard LEGUY

Université de Lille Flandres Artois, laboratoire d'informatique fondamentale de Lille, bâtiment M3, 59655 Villeneuve d'Ascq Cedex
leguyb@lifl.lifl.fr

Abstract. GWEZ is a set of macros able to build tree structures and to draw them; these macros are written with TEX; they use only plain TEX commands and fonts and can as well be used with LATEX.

Résumé. *GWEZ est un ensemble de macros permettant de construire des structures arborescentes et de les tracer; ces macros sont écrites en TEX; elles n'utilisent que des commandes et des polices de plain TEX et peuvent aussi bien être utilisées avec* LATEX.

Key words: tree structures, program design, macros.

1. What kind of trees can we draw?

Tree structures are often used for describing a lot of hierarchical organizations: a firm, a table of contents, genealogies, directories of disk systems... Let us have a look at some examples shown in figure 1.

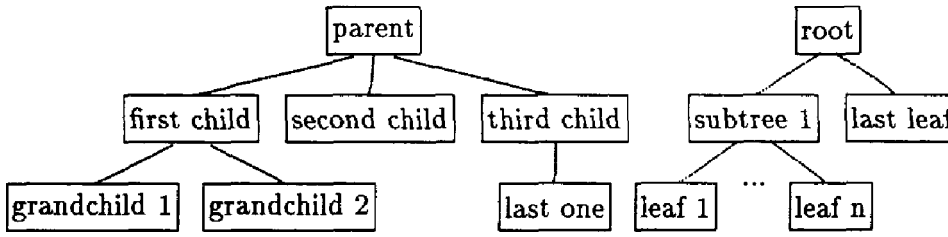

Figure 1. two trees.

When a tree is made of several nodes, one of them, usually above the others, is called the root, or parent, or main node. That main node is linked directly by straight lines to some other nodes which are put in a row under it and called the children of the main node. The children are in fact the roots of subtrees. These subtrees are said to depend on the main node: they

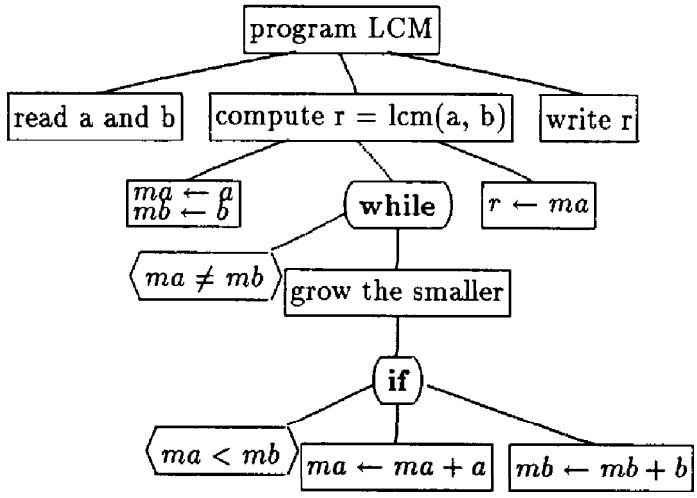

Figure 2. a tiny program.

represent groups under the authority of the main node, offspring of the parent, components of the root...

In LILLE, we also use trees for the design of algorithms by stepwise top down refinements. The figure 2 is an example of such a design for a tiny problem in arithmetic. The main idea is that the children of any node are a series of statements that achieves the action stated in the node. Some nodes in round boxes like **if** and **while** represent the control structures we find in most of the programming languages; the angle boxes contain the conditions associated with them.

```
% read a and b
   \message{a=?} \read-1 to\answer \newcount\a \a\answer
   \message{b=?} \read-1 to\answer \newcount\b \b\answer
% compute r = lcm(a,b)
   \newcount\ma \ma\a \newcount\mb \mb\b
   \loop\if\ifnum\ma=\mb01\else00\fi% while a <> b
      \ifnum\ma<\mb \advance\ma\a \else \advance\mb\b \fi \repeat
   \newcount\r \r\ma
% write r
   \message{lcm(\the\a,\the\b)=\the\r}\bye
```

Figure 3. example of lcm program.

The figure 3 is an example of translation of this algorithm[1] in your favorite programming language.

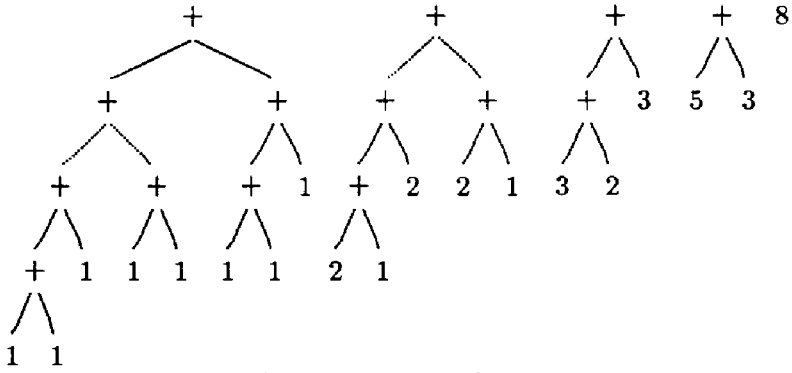

Figure 4. a series of trees.

As a last example, figure 4 is a series of trees representing arithmetical expressions. The series begins with a Fibonacci tree; the other trees represent different steps of the computation; the last step is a single Fibonacci number.

You can draw all these trees by using any wysiwyg system. But these systems, very efficient at drawing simple geometric patterns, are unable to understand the structure you are designing. So you have the tedious task of drawing (sometimes by copying) every node and every link. It is just a little more tedious if you want to have the nodes precisely aligned. Moreover, if you want a style for your trees (I mean they will look the same and they will be balanced in the same way), you have to be disciplined and trained at this kind of drawing.

Unfortunately, things are often even worse. Imagine you have already wasted a lot of time coping with the drawing, and you change your mind: one node gets wider and you must make room for it! When this happens, it is generally much more efficient to return to true wysiwyg: draw the new node somewhere, print the whole thing on true paper, take true scissors and glue and do the job in the old way. It will save a lot of time and keep you from running out of patience!

[1] This algorithm is not very efficient, it would be much better to compute first the gcd of a and b, and then multiply a by the quotient of b by the gcd.

2. Using TEX for drawing trees

TEX, which is able to cope with a lot of structured documents, does not provide any tool for trees. Fortunately, it is open to new ideas and the main part of its power is its ability to learn. So, after some weeks (I had to learn a lot about TEX before trying to control it), I was able to draw these trees by simply describing their structures with a small set of commands which are very close to the programs we write in usual programming languages. The main idea is that the subtrees under a node are viewed as components embedded in the node such as blocks in block structured languages. So the tree of figure 2 is defined by the commands given in figure 5.

```
\begintree
   \block program LCM!!                        % a block has children
      \leafblock read a and b!!                % a leaf has no children
      \block compute r = lcm(a, b)!!
          \leafblock $ma\leftarrow a$!$mb\leftarrow b$!!
              % two lines in one node!
          \while $ma\ne mb$!!                  % while has a condition
             \block grow the smaller!!         % and a single child
                \ifblock $ma < mb$!!           % if may have two children
                   \leafblock $ma\leftarrow ma + a$!! % then part
                   \leafblock $mb\leftarrow mb + b$!! % else part
                \endif
            \endblock
          \endwhile
          \leafblock $r\leftarrow ma$!!
      \endblock
      \leafblock write r!!
   \endblock    % endblock is needed to know where is the last child
\endtree
```

Figure 5. Example of commands to draw a program tree.

The text we want to put in a node is ended with a double exclamation mark[2]. If the text is to be put on several lines, these lines are separated by a single exclamation mark as in the second leaf in figure 5.

A single `\begintree`[3] command is needed to draw several trees in a row as shown in figure 1. In fact, these trees are orphan subtrees; ie, they are

[2] exclamation marks are not often used in trees, so it was a convenient marker.

[3] A better name for the command would be `begintrees`, but most of the time we use it to draw a single tree. Perhaps it would be better to use the breton word gwez which is the collective name for tree or trees. But the word tree proved to be understood by a few more people.

drawn as if they were under some parent node which had disappeared. The node introduced by **\comments** has no box around its text and no link to its parent (see figure 6).

```
\begintree
   \block parent!!
...
   \endblock
   \block root!!
      \block subtree 1!!
         \leafblock leaf 1!!
         \comments ...!!
         \leafblock leaf n!!
      \endblock
      \leafblock last leaf!!
   \endblock}
\endtree
```

Figure 6. Example of commands to draw two trees.

```
\node $+$!!
   \node $+$!!
      \leaf 3!!
      \leaf 2!!
   \endnode
   \leaf 3!!
\endnode
```

Figure 7. Commands for nodes without boxes.

The nodes in figure 4 have links but no box, since they are defined by other commands. The third tree of figure 4 is defined as shown in figure 7. The commands **\node**, **\endnode** and **\leaf** act exactly like **\block**, **\endblock** and **\leafblock** but no box is visible around them.

3. The way GWEZ stores trees.

The command **\begintree** opens a new scope by the mean of a **\begingroup** command; then it initializes some variables. Every command like **\block** builds an hbox which contains the text of the node, and then the node is inserted in a complex structure.

The first part of that structure defines the tree structure. Each node has seven fields defined by TEX registers:

- a box register **txt** which contains the text of the node
- a dimen register **pos** which is the distance to the left margin
- two dimen registers **dx** and **dy** that define the link to the parent
- a count register **frr** which is a pointer to the right sibling
- a count register **nbf** which is the number of children
- a token register **typ** which is the type of node.

So, when we have a pointer to a node in a given structure, we can know everything about it and its siblings to the right. Pointers are integers. The first child of any node is introduced immediately after its parent, and its pointer is one more than the pointer of its parent. So, from any node we can determine its children without the need of a first child pointer.

Actually, the `frr` register is a pointer to the next right sibling when the node is not the last child of its parent. Otherwise, if the node has a right cousin or grand cousin... at the same level (for example, in figure 1 'leaf 1' is the right cousin[4] of 'last one') then `frr` is a pointer to this cousin with a minus sign. In any row, only the last node has `frr` equal to 0. In this way, we are able to visit a whole row if we know the pointer to the node of that row farthest to the left.

The second part of the structure is related to the rows of nodes. These rows are numbered from one. For every row or level there are six fields:

- a count register `ptn` which points to the node which was the last visited at that level. The stack of these registers is used as a tree cursor; when we want to return to a node after having finished visiting its children, we find a pointer to that node in `ptn`.
- a count register `cpn` used to count the children of the current node.
- a count register `egn` which points to the first node of that level.
- a count register `edn` which points to the current node of that level farthest to the right.
- a dimen register `pon` which is the current width of the row.
- a dimen register `epn` which is the depth of that level; the nodes have no height but only a depth. `epn` is useful to get the nodes aligned in a row.

Only `egn` and `epn` are useful throughout the entire process. The other four fields are useful only when the structure is being built. TEX does not provide as many registers as we could hope, so we have to set a limit to the complexity of the trees. By now, the number of nodes is limited to 50 and the number of levels to 10.

Here is how pointers are used. Let the counter variable `\e` be the number of the current node. When the field `pos` of this node is to be used, it is

[4] Actually they are grand 'grand cousins' if we remember that their grandparents are orphans and supposed to have had the same disappeared parent.

```
\def\mkptr#1{\c=#1\advance\c} % \c is a counter with many uses
\def\sptrpos{\mkptr\e50\dimendef\pos=\c}
```

Figure 8. example of macro to use pointers

convenient just to say \pos as if it were a dimension variable. In order to get that, it is just necessary to call the macro \sptrpos defined in figure 8. The value 50 is there because the pos fields are (very arbitrarily) supposed to be the dimension registers 51, 52,... (for the node i, the pos field is the dimension register whose number is $i+50$). Other macros are defined in the same way for the other fields.

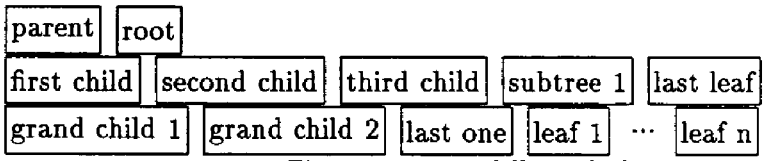

Figure 9. two awfully packed trees.

When the commands describing the trees are finished, almost everything is settled in these structures. But, the link dimensions dx and dy remain undefined and the pos field which is intended to define the position of each node in its row is just a left boundary of that position: the nodes are put one after another in their row like hboxes in a line. You can think that the nodes of the trees of figure 1 are packed to the left as shown in figure 9, when the \endtree command is reached. This macro is finished by an \endgroup command.

4. The way GWEZ computes the positions of the nodes.

At first sight, we could proceed recursively: draw the subtrees of a node in hboxes, put the row of hboxes in a single hbox, then put the node in the middle of a line over that hbox, perhaps by using a vbox. But it would waste a lot of space. For instance, the first tree of figure 1 would look like the one in figure 10 and there would not be enough place for the second tree.

The algorithm presently used is not so easy. We begin with the next to the last level and go upward, level after level until the first level is reached. For every node at the current level, we try to put its subtrees at what we consider to be their best place relative to their parent.

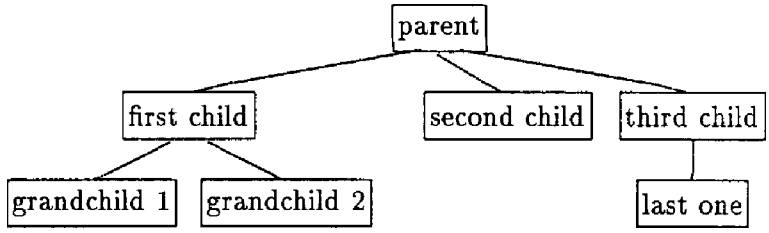

Figure 10. one tree with wasted space.

The choice of the best place depends on the types of nodes. Let us consider only plain nodes like those built by `\block`, `\leafblock`, `\node` or `\leaf`. We choose to have the subtrees of a node as tight as possible without overlapping. The best place for the middle of the node is at a point at an equal distance from the middle of the first child to the middle of the last one. This gives well balanced trees even when subtrees have very different widths. (Convince yourself by looking at figure 4.) When we begin this process, the nodes are packed to the left with a standard small space between them. We can say that they are not too far towards the right, and we will proceed as if they were never too far towards the right. This is almost always true and it avoids left-shifting.

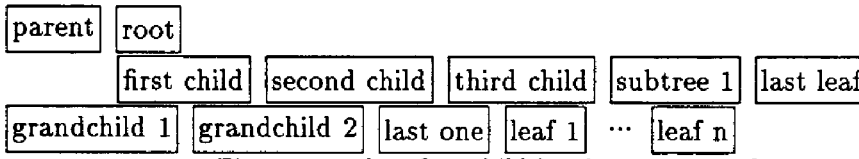

Figure 11. when first child has been processed.

If the node is too far to the left, we just have to shift it to the right until it reaches its best place. But doing so, it will overlap its right siblings or cousins, and we will have to shift them by the same amount. We need not be aware of the subtrees of these siblings or cousins. In any case they are not already at their best places and will be visited later. The figure 11 shows what we get after applying this process to the 'first child'. The 'second child' needs no processing because it has no children. Then either the 'third child' is too far to the right, or its child is too far to the left; so the child is shifted to the right as shown in figure 12. The 'subtree 1' is processed in the same way as 'first child' and, 'last leaf' having no child, the processing of the second level is finished as shown in figure 13.

Unfortunately, when the node is too far to the right, the shifting of its children is quite time consuming because we have to shift to the right all the

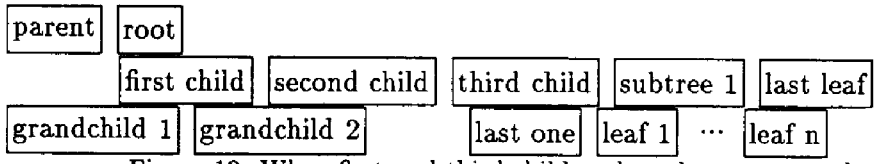

Figure 12. When first and third children have been processed.

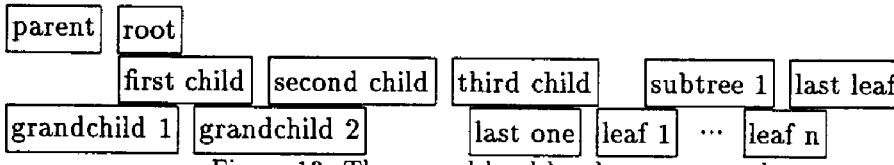

Figure 13. The second level has been processed.

nodes that are to the right of the first child, and recursively, the first grand child and all the nodes to its right, and so on...

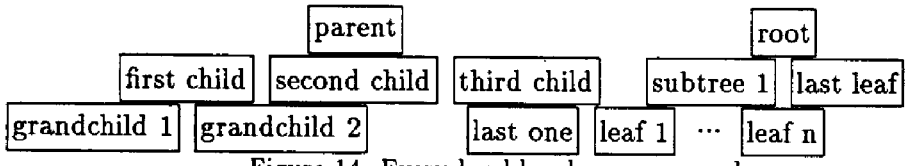

Figure 14. Every level has been processed

When every node of every level has been processed up to the last node of the first level, the nodes are at their best places and the trees are (almost) as narrow as possible.

At this point we can compute the links (ie `dx` and `dy`). The process ends with the output of the result, which is done one row at a time by using the `txt` fields, which are hboxes, and by putting kerns between them. The links are also drawn between two rows of nodes.

5. When the result is not what we expected.

When a node has more than two children, nothing is stated about the positions of the subtrees that are between the first child and the last one. Actually, they are close to the first one. Sometimes it does not look as good as we would like. We can improve the tree by introducing void comments that contain only a kern or an hskip. We can also make use of the `\espace` command which takes a one dimension parameter (the width of the blank space inserted). It would be better if GWEZ could manage `\hfil`, but it would probably be much more time consuming.

```
\begintree
   \block program LCM!!
      \leafblock read a and b!!
      \block compute r = lcm(a, b)!!
          \espace{6mm}
          \leafblock $ma\leftarrow a$!$mb\leftarrow b$!!
          \while $ma\ne mb$!! ...
```

Figure 15. change in a tiny program.

After a close look at the tree in figure 2, we notice that there is a gap at the left of the 'while' node. This gap occurs because the first time that the position of that node is computed, the condition '$ma \neq mb$' as well as the left sibling of 'while' are against the left margin. The 'while' node has to be shifted to the right until its center is vertically aligned with the center of the node representing the action of the 'while'; then a gap is made between the 'while' node and its left sibling. When the parent of the 'while' node is processed, its subtrees are shifted all together to the right and the gap is never removed. We can avoid or reduce that gap by inserting a blank space to the right of the row (see the change in commands figure 15 and the new tree figure 16). Some of these adjustments could be avoided by revisiting

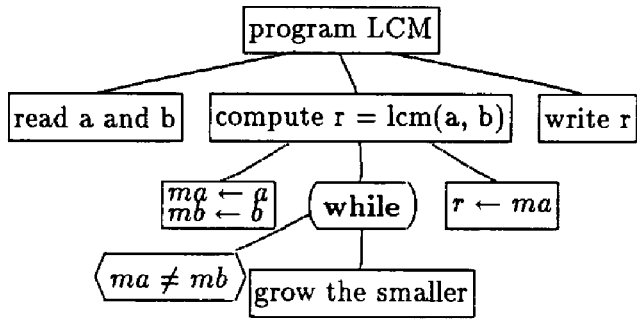

Figure 16. a better looking tiny program.

the tree. But it is not sure that it will always work. Moreover, there are the points that it would be quite time consuming, that adjustments are easy to make, and that the trees are often quite acceptable without any adjustment.

6. Graphics drawing with TEX

The drawing of rectangular boxes is easy and well known. The round boxes are built with hrules and parentheses. The boxes with angles which are used for the conditions of 'while' and 'if' are more difficult to draw because left and right angles cannot be very high with plain TEX; such angles are built by the repeated use of slashes and back slashes.

```
\def\linetopoints{\kern\dtx
   \vrule width\dempt height\ph depth\pb % one point
   \advance\ph\dty \advance\pb by -\dty \advance\c by -1
   \ifnum\c>0 \let\next\linetopoints \else \let\next\relax \fi \next}
```

Figure 17. macro to draw a line.

Another difficulty arises in line drawing. In a tree we can find a lot of links. We can say nothing about their slopes, which inhibits the use of the picture environment of LATEX. Moreover, the lines have to fit with the rules used for boxes. The figure 17 is the main part of the line drawing macro used by GWEZ. It makes use of the following variables.

- **dempt** is the dimension of the points used to draw lines (0.5pt).
- **dtx** is the space to put between points (may be negative).
- **dty** is the vertical shift from one point to another.
- **c** is the number of points needed for the line.

Before calling **\linetopoints** the registers have to be set to values depending on the line we want to draw. This is done by direct computation. Actually, the result of **\linetopoints** is put into an hbox whose width, height and depth are zero. This line drawing is a little slow and space consuming, but I was unable to find any better way.

7. Conclusion

The present version of GWEZ was built piece after piece; the design was overly imposed by our peculiar program trees. So the structure of the whole is quite odd. But it works, we use it and, by using it, we get new ideas about the next version[5]. Actually, GWEZ can be run with plainTEX and LATEX on a SUN without any problem. On a PC, it works with TEX; but with LATEX,

[5]If you wish to try it, I can send it to you by email.

the TEX capacity is often exceeded because GWEZ uses a lot of registers and runs out of save size. For instance, the present article is proceeded easily on a SUN but not on a PC. However, if I have time to learn more about TEX, I think the next version will be better and less space gready.

One could be doubtful about the usefulness of writing everything in TEX. I tried to import drawings from other systems. But it does not work on every computer (the command `\special` has different uses when it works). In the trees, there is a lot of text and it does not look nice when the fonts are not TEX fonts. Moreover, the number of operations needed is greater: (draw the tree, put the special command, make room in the document...). In case of any change in trees or text, we have to reconsider the whole process (more room may be needed). In any case, other drawing systems do not give any efficient help in drawing trees. GWEZ is completely portable; it needs only plain TEX and you can use screen drivers and dot printers.

What is lacking most in TEX is simple drawing capabilities. First of all, lines of any slope and any thickness: almost anything else could be made easily with them. It seems that the idea of using a post-processor on dvi files could be a good one. Another way could be to build a METAFONT file that would define a new font with characters having size zero; each new line drawing could add a new character definition in that file and put that character in the text processed by TEX. After the end of the process, we could ask to METAFONT[6] to process the new created file and build the font which could be used by the output drivers.

References

[Hendrickson90] Amy HENDRICKSON, "Getting TEXnical: Insights into TEX Macro Writing Techniques", *TUGboat*, Volume 11 (1990), No. 3 - Proceedings of the 1990 Annual Meeting.

[Olejniczak-Burhert89] Rolf OLEJNICZAK-BURHERT, "*texpic* — Design and Implementation of a Picture Graphics Language in TEX à la pic", *TUGboat*, Volume 10 (1989), No. 4 — 1989 Conference Proceedings.

[Rogers89] David F. ROGERS, "Computer Graphics and TEX — A Challenge", *TUGboat*, Volume 10 (1989), No. 1, pp. 39–44.

[Wilcox89] Patricia P. WILCOX, "METAPLOT, Machine Independent Line Graphics for TEX", *TUGboat*, Volume 10 (1989), No. 2, pp. 179–187.

[6] or perhaps some specific program, because drawing simple lines does not requires the whole power of METAFONT.