Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J.C. van Vliet, J.B. Warmer

Intertable

Computer Science/Department of Software Technology Report CS-R8636 November

---

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

Intertable

J.C. van Vliet, J.B. Warmer
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Intertable is an interactive program for editing tables. It can be used to create and manipulate tables. The present report describes Intertable, with emphasis on the algorithms used to incrementally update the widths of rows and columns of a table.

1980 Mathematics Subject Classification: 68K05
1986 CR Categories: I.7.2 [Text processing]: Document preparation
Note: This report will be submitted for publication elsewhere.

1. Introduction

In the past, a compositor assembled pieces of lead on his composing-frame. Nowadays, compositors are replaced by computers. Numerous systems with varying capabilities are available for the processing of all kinds of documents. For simple documents, one may choose from a large variety of textprocessing systems. These systems are often interactive, are capable of processing relatively simple documents, and output the result on printers with limited capabilities.

At the other end of the spectrum we find systems intended to process complex documents, documents that contain figures, formulae and tables, that are to be printed on high-resolution devices such as phototypesetters. These systems are usually not interactive. Troff [1] and TEX [2] are typical examples from this category. (The above crude distinction gets blurred by developments such as TEX-implementations on micros, and the availability of cheap yet powerful laserprinters.)

A typical way these batch-systems for processing complex documents work is as follows:

1) The user types in his text, interspersed with markup-commands;
2) Some process is started to format the text according to the markup-commands included in the document;
3) The user checks the result to see whether it is as intended;
4) If the result is not satisfactory, the markup is changed and the process is iterated.

If the user is lucky, the output may be produced on some low-cost device for proofreading purposes. If she is real lucky, the result may even be previewed on a screen. In both cases, she has to be rather proficient in the system being used to get things right the first time. If she is not, it may be a time-consuming task to get the intended result. In our environment, where most users are relatively well-acquainted with the system used (Troff), it is not atypical to have several iterations before a table or picture has the intended outlook.

The problem that is addressed in this paper is that of building tables interactively. This is part of ongoing research on interactive document manipulation at the Centre of Mathematics and Computer Science. Similar projects are being carried out for formulae and plain (structured) text.

An important aspect of our system is that the object, such as a table or formula, that is being created or manipulated, is known to the system as being a table or formula. That is, the system does

not manipulate some rectangular collection of characters or pixels that happens to represent a table or formula, but it rather manipulates the table or formula itself, an object with certain well-defined structural properties. This is similar to the way syntax-directed editors for languages like Pascal differ from standard screen-oriented editors [3]. Such syntax-directed editors, or structure-oriented editors, exist for formulae [4] and more general classes of documents (see for example the collection of papers in [5]).

The advantages of such an approach are:

1) The user need not have a precise knowledge of how markup-commands exactly look like. It is much easier to select an entry *paragraph* or $\sum$ from a menu than it is to remember commands like '.PP' or '\ sum'.
2) The system may check the syntactic form of constructs, so that illegal combinations can be reported on immediately, or can even become impossible.
3) If the system displays the object as it would (approximately) be printed, the user has a much more immediate feedback of what she is producing.

Relatively little work in this area has been done for tables. The only article on interactive table makeup we know off is [6]. The emphasis in his research is somewhat different though; it explores an object-oriented architecture for editors of complex structures, such as tables. The repertoire of the editor is similar to ours, and it also generates tbl-output to be further processed to produce hard-copy output. We are more interested in the interactive aspect: how to create and manipulate complex objects such as tables, providing for an easy-to-use yet flexible user interface, and a view of the object that allows the user to accurately judge its appearance if processed by the formatter.

Tables somehow seem to be more difficult to handle than, say, formulae. They have less structure; there is much more freedom in the two-dimensional arrangement of table entries. This has its bearings on the command repertoire and user interface one provides for. There are also fairly few strict typographic rules as to what constitutes a properly laid out table. [7] contains an elaborate discussion of the various issues involved.

The present paper deals with an interactive version of tbl, the preprocessor of Troff that handles tables [8]. Tbl by no means produces beautiful tables in all cases, nor does it allow one to produce any table layout one may desire. This is partly caused by the fairly simple algorithms that are used internally to decide on the widths of columns and the like, as is further discussed in the next sections. A big advantage however is that we may now restrict ourselves to the interactive, screen-directed portion of the problem, and fall back on the standard available software whenever hard-copy output is needed. Also, tables in tbl-format that are produced in a different way, e.g., by just typing in the appropriate markup-commands, can be further processed by our system. Thus it becomes relatively easy to incorporate this tool in an environment that is used to Troff in a batch mode, very much in the way CIP [9] is a useful tool for interactively creating pictures.

After a short description of some relevant features of tbl in section 2, the main discussion in section 3 centers around the internal manipulations needed whenever incremental changes are made to a table. In section 4 some details are given on the present version of our system. Section 5 finally contains some concluding remarks.

2. TBL

For an elaborate discussion of the possibilities of tbl, the reader is referred to [8]. Tbl is a preprocessor of Troff [1], the formatter most heavily used under UNIX. Tbl scans the input text, processes the information contained between commands .TS and .TE, and ignores the rest. A typical table is specified in figure 1; its output is reproduced in figure 2. (This example shows but one trap the user may fall into: At first sight, the table in figure 2 contains 7 rows. The specification in figure 1, however, contains format information for 8 rows (line 3-10). This is needed because the (partial) line separating rows 3 and 4 in the table is counted as a row by tbl too. We fell into this trap, and the result was a slightly disrupted table.)

In specifying a table for tbl, the user has to provide three pieces of information:

```
1   .TS
2   center, box, tab(@);
3   cfB s s
4   cfB s s
5   c | c | c
6   n | n | n
7   n | n | n
8   n | n | n
9   n | n | n
10  n | l s.
11  Average liquor consumption
12  (in liters)
14
15  _@Beer@wine
16  _@_@_
17  1970@20@7.5
18
19  1975@23@9.3
20
21  1980@30@12.7
22
23  1985@data not available
24  .TE
```

Figure 1: Sample tbl-input

Average liquor consumption (in liters)		
	Beer	wine
1970	20	7.5
1975	23	9.3
1980	30	12.7
1985	data not available	

Figure 2: A table after processing by tbl and troff

1) Some information on the global appearance of the table, such as whether it has to be centered on the page, and whether or not a box has to be drawn around the whole table (line 2 in figure 1).
2) Information on how the individual entries have to be handled (on a row-by-row basis): whether text has to be centered or left-aligned, whether the text spans over more than one column or row, etc. (lines 3-10 in figure 1).
3) The contents of the individual entries (line 11-23 in figure 1).

In part 2, the user may also give information on the width of columns. She may for example specify that certain columns should have equal width, or that a certain column should get a given minimal width. Given this a priori information, tbl computes all column widths. Since tbl may scan the whole table, the widths of all entries can be computed in one sweep. If the table is built up interactively, this information is not available, and column widths therefore have to be recomputed quite often. This has direct bearings on the speed and accuracy with which the table is represented on the screen after a user change. The algorithm to compute column widths is discussed in the next section.

3. Calculating column widths

Calculating the width of all columns of a table is relatively easy if no items span over more than one column. In that case, the width CW of a column c is simply given by

$$CW(c) = \max_{r \in Rows} (IW(r,c)) \tag{1.1}$$

where r ranges over *Rows*: the set of all rows of the table. Here, $IW(r,c)$ denotes the width of the contents of the entry in column c at row r. (We ignore the fact that computing the value of $IW(r,c)$ may not be all that easy in practice, since it may involve summing up sizes of characters in quite different fonts. Also, white space in between columns is not taken into account.) Problems arise if items are spanned over more than one column, as in figure 3.

entry1	
entry2	entry3

Figure 3: a simple table with one spanned entry.

There are various ways to display tables containing spanned entries:

1) One may assign equal portions of the spanned item to each of the columns it spans. In figure 3, this amounts to:

$$CW(1) = \max(width(entry\,2),\ width(entry\,1)/2)$$

$$CW(2) = \max(width(entry\,3),\ width(entry\,1)/2)$$

2) One may ignore the possible effect of a spanned item to all but the last column it spans, and assign the remainder to the last column. For figure 3, this amounts to:

$$CW(1) = width(entry\,2)$$

$$CW(2) = \max(width(entry\,3), width(entry\,1) - CW(1))$$

3) One may similarly ignore the possible effect of a spanned item to all but the first column it spans. This leads to a right-to-left computation similar to case 2 above.

4) One may minimise the total width $\sum CW(c)$ of the table, subject to the constraints set forth by the individual entries:

$$CW(1) + CW(2) \geqslant width(entry\,1)$$

$$CW(1) \geqslant width(entry\,2)$$

$$CW(2) \geqslant width(entry\,3)$$

Solution 4 is probably the most pleasant looking. Algorithmically, it amounts to solving a set of lineair inequalities; this is further elaborated upon in [7]. Tbl uses a mixture of solutions 1 and 2. The approach of solution 2 is taken in the remainder of this paper. Note that the tbl-algorithm may result in a layout such as the one given in figure 4.

a rather long text xxxxxxxxxxxxxxxxxx	
one	two

Figure 4

Although tbl allows the user to specify that certain columns have equal width, doing so for the above example leads to the output as given in figure 5, which is probably not what one expected.

a rather long text xxxxxxxxxxxxxxxxx	
one	two

Figure 5

3.1. Calculating widths, non-incremental version

In the subsequent discussion, the following notations and definitions are used:

c	denotes some column $c \in [1,n]$, where n is the number of columns in the table;
r	denotes some row $r \in [1,m]$, where m is the number of rows in the table;
$item(r, c)$	denotes the item in row r and column c. Note that an item may span several rows and/or columns, so that $item(r_1, c_1)$ and $item(r_2, c_2)$ may denote the same item even if $(r_1, c_1) \neq (r_2, c_2)$.
i	denotes some item, i.e., $i \in \{ item(r, c) \mid r \in [1,m] \wedge c \in [1, n] \}$;
$first\_column(i)$	denotes the first column spanned by item i
$last\_column(i)$	denotes the last column spanned by item i
$I(c)$	denotes the set of items occurring in column c, i.e, $I(c) = \{ i \mid first\_column(i) \leqslant c \leqslant last\_column(i) \}$
$Q(c)$	denotes the set of items that end in column c, i.e, $Q(c) = \{ i \mid last\_column(i) = c \}$
$CW(c)$	denotes the width of column c
$IW(i)$	denotes the width of the contents of item i

Using these definitions, equation (1.1) may now be rephrased as:

$$CW(c) = \max_{i \in I(c)} (IW(i)) \tag{1.2}$$

If the table contains spanned items, the situation gets more complicated. In that case, only those items that end in column c, the qualifying set $Q(c)$, may influence its width. The width of column c is then given by

$$CW(c) = \max_{i \in Q(c)} (IW(i) - \sum_{cc = first\_column(i)}^{last\_column(i) - 1} CW(cc)) \tag{1.3}$$

Computing column widths from left to right, the widths of column 1 up to $c-1$ are known when the width of column c is computed, and these values are used in equation (1.3) for spanned items ending in column c. Note that, when there are no spanned items ending in column c, equation (1.3) reduces to (1.2).

For the following discussion it is helpful to define the notion of a *restvalue*. The restvalue R of an item i is defined as that portion of its width that has to fit in the last column that is spanned by i. So,

$$R(i) = IW(i) - \sum_{cc = first\_column(i)}^{last\_column(i) - 1} CW(cc) \tag{1.4}$$

and equation (1.3) reduces to

$$CW(c) = \max_{i \in Q(c)} R(i) \tag{1.5}$$

Given these basic equations it is relatively easy to derive the algorithms to compute column widths.

```
(*
 *  Calculate widts of all columns, no spanned items.
 *)
PROCEDURE CalculateWidth(table);
BEGIN
    FOR column := first_column(table) TO last_column(table) DO
        column_width := minimum_width(column);
        FOR row := first_row(table) TO last_row(table) DO
            item         := item(row, column);
            column_width := max(column_width, width(item) );
        END;
    END;
END CalculateWidth;
```

Figure 6: Algorithm 1, derived from equation 1.2

```
(*
 *  Calculate width of all columns, looks also at spanned items.
 *)
PROCEDURE CalculateWidth(table);
BEGIN
    FOR column := first_column(table) TO last_column(table) DO
        column_width := minimum_width(column);
        FOR row := first_row(table) TO last_row(table) DO
            item  := item (row, column);
            IF column = last_column(item) THEN
                restvalue := 0;
                tmp       := first_column(item);
                WHILE tmp != column DO              (* calculate restvalue *)
                    restvalue += column_width(tmp);
                    tmp       := next_column (tmp);
                END;
                (*
                 *  What's left must fit in this column.
                 *)
                column_width := max(column_width, width(item) - restvalue);
            END;
        END;
        column_width(column) := column_width;
    END;
END CalculateWidth;
```

Figure 7: Algorithm 2, derived from equation 1.3

Figures 6 and 7 contain algorithms for the non-spanned (equation 1.2) and spanned (equation 1.3)

case, respectively. In both cases a certain minimum width is assumed for each column. Figure 7 also assumes that for each column there is at least one entry that ends in that column, i.e., $Q(c) \neq \emptyset, \forall\, c \in [1{:}n]$. This is reasonable, since there is little use for that column otherwise.

Again, if there are no spanned items, algorithm 2 reduces to algorithm 1. Computing the heigth of the rows goes in a similar fashion and is not further elaborated upon here.

If $n \approx m$, algorithm 2 takes $O(n^3)$ steps in the worst case. Since it may be expected that the majority of tables will include few spanned entries, algorithm 2 works fine in a batch environment. However, if used interactively, recomputing all column widths after each change in the table will lead to an excessive computational overload. Most of the changes will have a small effect, if any, on the column widths. It thus becomes crucial to recompute only those widths that are affected by the change (this leads to less computation; moreover, refreshing the screen can be optimised too). In the next section an incremental version for computing column widths will be derived which does less recomputation and makes heavy use of the restvalues $R(i)$.

3.2. Calculating widths, incremental

In an interactive environment a table will be constructed by making many little changes. These changes include changing the contents of an item, changing the format, adding and deleting rows or columns and changing the table options. The goal of our program is to give the user a view of the table as it will eventually look like on paper after each change. This means that after each small change the column widths have to be recomputed. Since this happens very often it is necessary to optimize the corresponding algorithm.

Most of the changes will affect few columns, if any, so calculating the widths using algorithm 2 is not appropriate. We are looking for a method in which only those column widths that are actually changed wil be recomputed. If we take a look at equation (1.5) we see that the width of a column c depends only on the restvalues of the items in $Q(c)$. This means that the column width need only be recomputed if a restvalue of some item in $Q(c)$ changes. We must note, however, that a change in the width of one column may affect the width of other columns. If, for example, one changes the first entry in figure 8a to 'a' or 'appletree', the width of column 2 changes as well, as can be seen from figure 8b.

apple	pear
apple, orange, banana	

Figure 8a: A sample table

appletree	pear
apple, orange, banana	

or

a	pear
apple, orange, banana	

Figure 8b: The same table after changing the first entry into 'a' and 'appletree'

In this way each column width changed may cause a cascade of changes in other columns. One may construct examples in which almost every column width changes because of a change in one item.

From equation (1.5) it follows that a change in column c affects the restvalues of all items that belong to column c, and do not end there. This set is given by $NQ(c) = I(c) \setminus Q(c)$ (NQ is Non Qualifying). Therefore, the widths of columns cc with $cc = last\_column(i)$ for $i \in NQ(c)$ may change too. Changing the width of these columns may lead to further changes, and so on. In figure 9, an algorithm is given which thus marks all possibly affected columns.

This method potentially marks too many columns. In particular, columns need only be marked if their width really changes. As we have seen, such depends on the restvalues R. The actual width of a

```
PROCEDURE Mark(column);
BEGIN
    IF column is marked THEN RETURN; END;

    mark column
    FOR row = first_row(table) TO last_row(table) DO
        IF item(row, column) = spanned THEN
            Mark( last_column(item) )
        END;
    END;
END Mark;
```

Figure 9: Mark all possibly changed columns

column c equals $R(item)$, for some *item* such that

$$R(item) = \max_{i \in Q(c)} R(i)$$

(see Eq. (1.5).) Suppose we keep the list of restvalues for items from $Q(c)$ in decreasing order. Then the first item on that list will determine the column's width. After a change, one entry from this list will get a different value. If the list is now sorted again, the new list may have a different head. Only if the head of the list has changed, the column width changes. Our final algorithm is based on this observation.

Let S_c be the list of items from $Q(c)$, sorted in decreasing order of the corresponding restvalues.

$$\forall\ l,\ k\ with\ 1 \leqslant l \leqslant k \leqslant |Q(c)| :\quad R(\ S_c(l)) \geqslant R(S_c(k))$$

It follows that $CW(c) = R(S_c(1))$. A change of δ in the width of some entry i from $Q(c)$ will result in a corresponding change in $R(i)$. If X is the value of $R(S_c(1))$ prior to the change, and Y is the value of $R(S_c(1))$ after the change and having sorted the list anew, then $Y - X$ is the resulting change in the width of column c.

```
PROCEDURE Update(item, δ);
BEGIN
    c       := last_column(item);
    R(item) := R(item) + δ;
    Sort item in S_c                (* needs only one bubble-sort step *)
    change  := CW(c) - R(S_c(1));
    IF change ≠ 0 THEN
        CW(c) = CW(c) - change;
        FOR i ∈ NQ(c) DO
            Update(i, change);
        END
    END
END Update;
```

Figure 10: Recursive update

If $Y - X = 0$, no column width is changed. If $Y - X \neq 0$, a cascade of further changes may result. In that case a change of $Y - X$ is propagated to all restvalues from the set $NQ(c)$. Figure 10 contains the algorithm which recursively updates all column widths that are changed. This procedure is optimal in the sense that only column widths that actually change are recomputed. However, a changed column width might be recomputed more than once. Such is the case when a column occurs as the last column of a member from $NQ(c)$ more than once. Because of the recursion the number of computations may still grow exponentially.

Since we use a left-to-right evaluation of column widths, we may replace the recursion by iteration. Rather than accomodating for the possible effect of each of these changes immediately, we may first collect all changes, and only then determine whether further columns are indeed affected. Such has been done in the final algorithm that is given in figure 11.

```
PROCEDURE Update(item, δ);
BEGIN
    Touched_columns := {};
    Update_item(item, δ);
    Update_columns();
END

PROCEDURE Update_item(item, δ);
BEGIN
    R(item) := R(item) + δ;
    c       := last_column(item);
    Sort(item, c);
    Add c to Touched_columns;
END Update;

PROCEDURE Update_columns();
BEGIN
    WHILE Touched_columns ≠ ∅ DO
        c := min (Touched_columns)
        change  := CW(c) - R(S_c(1));
        IF change ≠ 0 THEN                (* column-width changed *)
            CW(c) = CW(c) - change;
            FOR i ∈ NQ(c) DO
                Update_item(i, change);
            END
        END
        delete c from Touched_columns
    END
END Update;
```

Figure 11: Iterative version

4. INTERTABLE

Intertable reads and writes tables in tbl-notation. It does have some limits and it is not yet able to read all tbl-specifications. Also, some specifications are read, but not used. It will only recognize plain text as the contents of an item. It will not recognize eqn-input (i.e. one can not incorporate formulae within a table) or troff-commands, except for the leading .TS and the ending .TE. Font- and pointsizes are read, written and can be changed, but they are not used in the layout computations and they are not shown on the screen.

After reading the input, the table is shown on the screen. When no input file is given, a minimal table consisting of one row and one column is created. Subsequently, the user may edit the table. When manipulating the table and/or its contents, three different levels are distinguished: the item-, ruler- and text-level. The program starts at the item-level.

4.1. The item-level

At the item-level, the user's focus is on one complete item, which is shown to the user in reverse video. On this level commands are allowed which relate to the row and column-structure of the table. These commands include creating new rows or columns, deleting rows or columns, changing global table options and traversing the table. There are four commands, **Left**, **Right**, **Up** and **Down**, which allow the user to traverse the table at the item-level, going from one item to another. To create a new row or column the commands **Insert Row** and **Insert Column** are available. These commands create a new row or column just before the current row or column. When a row is created, the format of the current row is copied and used as the format of the new row. When a new column is opened, the formats of all rows are updated and the format information of the current column is used for the newly created column. Similarly, the commands **Append Row** and **Append Column** create a new row or column after the current row or column. The user may change this format afterwards. (See section 4.3 below) The commands **Delete Row** and **Delete Column** are used to delete the current row or column.

At the item-level the user may also change some options global to the table. To do this, she has to perform the command **Change Options**. A menu is then shown with which the various table-options can be turned on or off.

4.2. The text-level

Whenever the user wants to change the text inside an item, she has to zoom in on the item. This will change the focus of attention to the text-level. The cursor focus will now be positioned on the first character position within the item. Again, the user is able to use the four commands **Left**, **Right**, **Up** and **Down** to move from one character to another within the item. If in doing so, the border of the item is hit, the cursor will jump over this border and the focus will be at the first character of the next item. At the text level some simple text-editing commands have been made available. Only the very basic commands are currently supported: delete-character, add-character and insert-character. Whenever the user has finished editing the text, she zooms out to the item-level.

4.3. The ruler-level

The format information for the rows of the table is similar to tbl. To each row a 'ruler' is attached, which contains the format for that row. Normally, this ruler is not visible to the user. To see the ruler the user has to perform the command **Open Ruler**. The ruler of the current row will then appear on the screen, just above the current row. It is visually separated from the table proper. The format consists of the alignment, pointsize, font and for each column and the possible vertical bars seperating adjacent columns. The possible values for the alignment are shown in figure 12.

l	left aligned
r	right aligned
c	centered
s	horizontal spanned
^	vertical spanned
-	horizontal line
=	double horizontal line

Figure 12: The alignments and their shorthands used by Intertable

For an example of a showed ruler, see figure 13. Whenever the ruler is displayed, the user may use the commands **Left** and **Right** to go from one column's format to another's. The commands **Up** and **Down** have no meaning in this context. The format of the current column is fully displayed on the bottom line of the screen, as shown in figure 13. This currently encompasses the alignment, pointsize and font information. The user may zoom in on that format to change it. Note that Intertable does keep a record of the font and pointsize of each item, but that it isn't shown on the screen or used by the layout algorithms. This might at present lead to surprising hard-copy output.

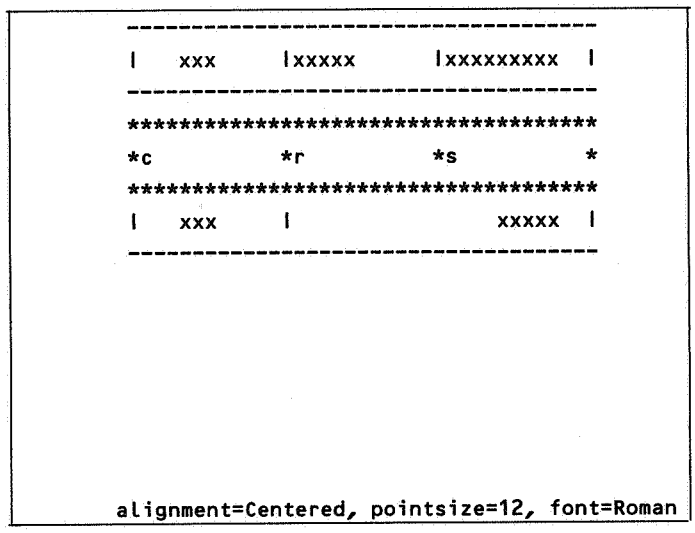

Figure 13: An example of a showed ruler

5. Conclusions

The objectives we had are twofold. The first objective of this research was to create an editor for editing tables, which knows about the structure of tables. It has to be able to recognize and handle all structural parts of a table. For example, the editor is able to add/remove rows or columns. The second objective was to see wether it is possible to do this in a WYSIWYG (What You See Is What You Get) fashion. This means that the user sees , at any time during a session, a picture of the table on the screen as it will eventually look like on paper.

As far as the second motive is concerned, we used ordinary ascii-terminals, since no bit-mapped workstations were available. The major drawback hereof is that only an approximate look of the table can be given. Different fonts and pointsizes cannot be shown. Within these constraints there is another, more basic, problem which is inherent to WYSIWYG table editing. The computation of the

layout of the table (the widths of the columns) takes a lot of time. We have tried to optimize this process along the lines shown in section 3. We also had to decide wether it is necessary and possible to keep the picture of the table up-to-date after each keystroke. Here we made one concession. Whenever the user is editing the text of a single item, the complete table will not be updated until she moves to another item. When the user is just typing in some text inside one item immediate feedback is necessary. Completely updating the table after each keystroke is not feasible is this situation. Also, reorganizing the entire screen after each keystroke might very well be disturbing to the user, even if it can be done instantaneously.

The present version of Intertable works satisfactorily. It is easy to build and change tables using this tool. If the workload of our system is not excessive, refreshing the screen after an item has changed is fast enough to not have an annoying effect on the interaction.

The user interface has not been tested on 'real' users yet. It definitely requires further thought. We plan to do so, together with some enhancements of the functionality, when porting Intertable to a workstation with a bit-mapped display.

REFERENCES

1. B.W. KERNIGHAN ET AL (1978). Document Preparation, *The Bell System Technical Journal*.57.6, 2115-2135.
2. D.E. KNUTH (1984). *The TEXbook*, Addison-Wesley.
3. T. TEITELBAUM and T. REPS (1981). The Cornell Program Synthesizer: A Syntax-Directed Programming Environment, *Communications of the ACM*.24.9, 563-573.
4. V. QUINT (1982). An Interactive System for Editing Mathematical Documents, in *Integrated Interactive Computing Systems*, 153-165, ed. P. Degano and E. Sandewall, North-Holland.
5. J.C. VAN VLIET (ED.) (1986). *Text Processing and Document Manipulation, Proceedings of the International Conference, Nottingham*, The British Computer Society Workshop series, Cambridge Univerity Press.
6. T.J. BIGGERSTAFF ET AL (1984). Table: Object Oriented Editing of Complex Structures, in *Proceedings 7th International Conference on Software Engineering*, 334-345, IEEE.
7. R. BEACH (1985). *Setting Tables and Illustrations with Style,* CSL-85-3, PhD Thesis, Xerox Corporation.
8. M.E. LESK (1979). TBL — A Program to Format Tables, in *UNIX Programmers Manual 7th edition*, 163-180, Bell Laboratories.
9. *5620 Dot-Mapped Display Text/Graphics Guide.*