Typesetting REDUCE output with TEX — A REDUCE-TEX-Interface —

Werner Antweiler
Andreas Strotmann
Volker Winkelmann
University of Cologne Computer Center, West Germany[1]
January 12, 2025

Abstract: REDUCE is a well known computer algebra system invented by Anthony C. Hearn. Although a pretty-printer is already incorporated in REDUCE, the output is produced only in line-printer quality. The simple idea to produce high quality output from REDUCE is to link REDUCE with Donald E. Knuth's famous TEX typesetting language. This draft reviews our efforts in this direction. We introduce a program written in REDUCE-Lisp which is able to typeset REDUCE formulas using TEX. Our REDUCE-TEX-Interface incorporates three levels of TEX output: without line breaking, with line breaking, and with line breaking plus indentation. This paper deals with some of the ideas we have put into LISP-code and it summarizes some of our experiments we have made with it yet. Furthermore, we compile a small user's manual introducing to the use of our REDUCE-TEX-Interface.

Keywords: Line-Breaking Algorithm, LISP, Prefix-to-Infix Conversion, REDUCE, TEX, Typesetting

1 Introduction

REDUCE is a well known computer algebra system invented by Anthony C. Hearn. While every effort was made to improve the system's algebraic capabilities, the readability of the output remained poor by modern typesetting standards. Although a pretty-printer is already incorporated in REDUCE, the output is produced only in line-printer quality. The simple idea to produce high quality output from REDUCE is to link REDUCE with Donald E. Knuth's famous TEX typesetting language. This draft reviews our efforts in this direction. We introduce a program written in REDUCE-Lisp to typeset REDUCE formulas with TEX. Our REDUCE-TEX-Interface incorporates three levels of TEX output: without line breaking, with line breaking, and with line breaking plus indentation. While speed without line breaking is comparable to that achieved

[1] The authors are with: Rechenzentrum der Universität zu Köln (University of Cologne Computer Center), Abt. Anwendungssoftware (Application Software Department), Robert-Koch-Straße 10, 5000 Köln 41, West Germany.

with REDUCE's pretty-printer, line breaking consumes much more CPU time. Nevertheless, we reckon with a cost increase due to line breaking which is almost linear in the length of the expression to be broken. This paper deals with some of the ideas and algorithms we have programmed and it summarizes some of the experiments we have made with our program. Furthermore, at the end of this paper we provide a small user's manual which gives a short introduction to the use of our REDUCE-TEX-Interface. For simplicity's sake the name "REDUCE-TEX-Interface" will be abbreviated to "TRI" in this paper.[2] At this point we should mention major goals we pursue with TRI:

- We want to produce REDUCE-output in typesetting quality.
- The intermediate files (TEX-input files) should be easy to edit. The reason is that it is likely that the proposed line-breaks are sub-optimal from the user's point of view.
- We apply a TEX-like algorithm which "optimizes" the line-breaking over the whole expression. This differs fundamentally from the standard left-to-right, one-line look-ahead pretty-printers of REDUCE, LISP and the like.

2 From REDUCE to TEX: concepts

REDUCE uses the function `varpri` to decide how to output a REDUCE expression. The function gets three arguments: the expression to be printed, a list of variables to each of which the expression to be printed gets assigned, and a flag which determines if the expression to be printed is the first, last or only expression in the line. `varpri` may be called consecutively for preparing a line for output. So, our task is to assemble all expressions before finally printing them. When !*TeX is true, `varpri` redirects output to our function `TeXvarpri`, which receives a REDUCE expression, translates it into TEX and pushes it onto a variable called `TeXStack*` before eventually printing it once the line is completed.

The `TeXvarpri` function first calls a function named `makeprefix`. Its job is to change a REDUCE algebraic expression to a standard prefix list while retaining the tree structure of the whole expression. Generally, this is done using a call `prepsq*(simp(expression))`, but lists and matrices need some special treatment. After this has been done the new intermediate expression is passed to the most important module of the TRI: the `mktag/makefunc`-family.[3] These functions recursively expand the structured list (or operator tree) into a flat list, translating each REDUCE symbol or expression into so-called TEX-items on passing by. For that reason, this list is called the TEX-item list. If the simple TEX-mode (without line breaking) was chosen this list is then printed immediately without further considerations. Translation and printing this way is almost as fast as with the standard REDUCE pretty printer.

When line-breaking has been enabled things get a bit more complicated. The greatest effort with TRI was to implement the line-breaking algorithm. More than half of the entire TRI code deals with this task. The ultimate goal is to add some "break items", i.e. `\nl`-TEX-commands[4], marking — in a certain way — optimal line-breaks. Additionally, these break items can be followed immediately by "indentation items", i.e. `\OFF{...}` TEX-commands[5], specifying the amount of indentation applicable for the next new line.

[2] The reason why it was called TRI and not RTI is simply due to the fact that TRI corresponds better to the three-level ("tri-level") mode.

[3] The whole family currently has five members. The "parents" are the mktag and makefunc functions which do the most burdensome job. The "children" are `makearg, makemat and makeDF` which handle special cases such as list construction or differentiation operators. We do not review the minor functions since they are easily understandable.

[4] This is not a TEX-primitive but a TRI-specific TEX-macro command which expands into a lot of stuff.

[5] see previous footnote

The problem is to choose the right points where to insert these special TEX-items. Therefore, the TEX-item list undergoes three further transformation steps.

First, the TEX-item list gets enlarged by so-called "glue items". Glue items are two-element lists, where the first element is a width info and the second element is a penalty info. The "penalty" is a value in the range $-10000 \ldots +10000$ indicating a mark-up on a potential break-point, thus determining if this point is a fairly good (if negative) or bad (if positive) choice. The amount of penalty depends (a) on the kind of TEX-items surrounding the glue item, (b) on the bracket nesting, and finally (c) on special characteristics.[6] The function handling this job is named `insertglue` which implicitly calls the function `interglue`. The latter determines the glue item to insert between a left and a right TEX-item.

During the second level, the TEX-item list becomes transformed into a so-called breaklist consisting of active and passive nodes. A passive node is simply a width info giving the total width of TEX-items not interspersed by glue items. On the other hand, active nodes are glue items enlarged by a third element, the offset info, indicating an indentation level which is used later for computing the actual amount of indentation. Active nodes are used as potential breakpoints. Moreover, while creating the breaklist, the TEX-item list will be modified if necessary according to the length of fractions and square roots which cannot be broken if retained in their "classical" form. Hence fractions look like `(...)/(...)` if they don't fit into a single line, especially in the case of large polynomial fractions. The major function for this job is named `breaklist` which calls `resolve` if necessary.

The third and most important level is the line-breaking algorithm itself. This algorithm embedded in the function `trybreak` will be described below. The idea how to break lines is based on the article by Knuth/Plass(1981). Line-breaking can occur at active nodes only. So, you can loop through the breaklist considering all potential break-points. But in order to find a suitable way in a reasonable amount of time you have to limit the number of potential breakpoints considered. This is performed by associating a "badness" with each potential breakpoint, describing how good looking or bad looking a line turns out. If the badness is less than a given amount of "tolerance" — as set by the user — then an active node is considered to be feasible and becomes a delta node. A delta node is simply an active node enlarged by four further infos: an identification number for this node, a pointer to the best feasible break-point (i.e. delta-node) to come from[7] the total amount of demerits (i.e. a compound value derived from badness and penalty) accumulated so far, and a value indicating the amount of indentation applied to a new line beginning at this node. When `trybreak` has stepped through the list, the breakpoints will have been determined. Afterwards all glue items (i.e. active nodes) are deleted from the TEX-item list while break- and indentation-items for those nodes marked as break-points are inserted.

Finally the TEX-item list is printed with regular ASCII-characters. We didn't put much emphasis on the question on how to format the intermediate output since it will be input directly into TEX. The best way to characterize the routine `texout` is to call it quick and dirty. The readabiltiy of the output is low, but it may be quite good enough for users to do some final editing work. Nevertheless, `texout` keeps the nesting structure of the term visible when printed, so it will be easy to distinguish between parenthesis levels simply by considering the amount of indentation.

---

[6] For example, the plus- and the difference operator have special impact on the amount of penalty.

[7] If one were to break the formula at this delta-node, the best place to start this line is given by this pointer.

3 Creating a TEX-item list

The first TEX-specific step in preparing a typesettable equivalent of a REDUCE expression is to expand the operator tree generated by REDUCE into a so-called TEX-item list. The operator tree is preprocessed by `makeprefix` in order to receive an operator tree in standard prefix notation. A TEX-item is either a character (letter or digit or special character) or a TEX-primitive or -macro (i.e. a LISP symbol), with properties `'CLASS`, `'TEXTAG`, `'TEXNAME` and (only if the item represents an operator) `'TEXPREC`, `'TEXPATT` and `'TEXUBY` bound to them, depending on what kind of TEX-item it actually is. The latter three properties are used for operators only. `'TEXPREC` is the precedence of the operator, a number between 0 and 999. Here the value itself is less important than the position with respect to other operators' precedences. The remaining properties will be described later.

First let's have a look at how a REDUCE expression arriving at TRI's main entry — the function `TeXvarpri` — is transformed through several levels of TRI-processing. For instance, let us consider the expression $(x-y)^{12}$, which expands into a polynomial $x^{12} - 12 \cdot x^{11} \cdot y + \cdots - 12 \cdot x \cdot y^{11} + y^{12}$ when evaluated. REDUCE uses a special form to store an expression. This form is called "standard quotient" because it in fact represents a quotient of two polynomials. The contents of the following figure 1 shows the "standard quotient" form of our example.

```
(*SQ ((((X . 12)           .    1)
       ((X . 11) ((Y . 1)  .  -12))
       ((X . 10) ((Y . 2)  .   66))
       ((X . 9)  ((Y . 3)  . -220))
       ((X . 8)  ((Y . 4)  .  495))
       ((X . 7)  ((Y . 5)  . -792))
       ((X . 6)  ((Y . 6)  .  924))
       ((X . 5)  ((Y . 7)  . -792))
       ((X . 4)  ((Y . 8)  .  495))
       ((X . 3)  ((Y . 9)  . -220))
       ((X . 2)  ((Y . 10) .   66))
       ((X . 1)  ((Y . 11) .  -12))
       ((Y . 12)           .    1)
) . 1) T)
```

Fig. 1: Standard Quotient Notation. This form is the way REDUCE represents terms.

The term has been indented by hand to retain the structure of this expression. Actually the denominator is 1 as you easily find out from the last line.[8] Obviously the standard-quotient form is a bit complicated for further manipulations. It can be changed to a real prefix notation as displayed in figure 2. Here, too, the term was edited by hand to make it a bit more readable and comparable to the other forms.[9] Note that `PLUS` is not a binary but a n-ary operator, i.e. it takes an arbitrary number of arguments, while `MINUS` is always a unary operator.[10] This causes a bit of trouble because real binary operators are much easier to handle. To tackle this problem we have introduced the `'TEXUBY` property, which is used to change a unary into a binary form if possible.

[8] The "T" in the last line is an "already-simplified"-flag indicating that the term doesn't need to undergo any more processing.

[9] "Edit" only means we have provided some additional indentation. We have changed neither the expression nor its structure.

[10] The same problem arises with the RECIP-operator which is the unary form of the binary QUOTIENT-operator.

```
(PLUS                   (EXPT X 12)
      (MINUS (TIMES  12 (EXPT X 11)       Y   ))
             (TIMES  66 (EXPT X 10) (EXPT Y  2))
      (MINUS (TIMES 220 (EXPT X  9) (EXPT Y  3)))
             (TIMES 495 (EXPT X  8) (EXPT Y  4))
      (MINUS (TIMES 792 (EXPT X  7) (EXPT Y  5)))
             (TIMES 924 (EXPT X  6) (EXPT Y  6))
      (MINUS (TIMES 792 (EXPT X  5) (EXPT Y  7)))
             (TIMES 495 (EXPT X  4) (EXPT Y  8))
      (MINUS (TIMES 220 (EXPT X  3) (EXPT Y  9)))
             (TIMES  66 (EXPT X  2) (EXPT Y 10))
      (MINUS (TIMES  12       X     (EXPT Y 11)))
                                    (EXPT Y 12))
```

Fig. 2: Prefix Notation. This list represents the state after application of `makeprefix`

A REDUCE expression is expanded using the two functions `mktag` and `makefunc`. Function `mktag` identifies the operator and is able to put some brackets around the expression if necessary. `makefunc` is a pattern oriented "unification"[11] function, which matches arguments of a REDUCE expression in order of appearance with so-called "unification tags", as explained below. Thus, `mktag` and `makefunc` are mutually dependent and highly recursive functions.

A "unification tag list" is a list (or a pattern, if you like) which consists of single "unfication tags". Each REDUCE operator is associated with a unification pattern. While expanding the expression, each tag is replaced by the appropiate TEX-item or partial TEX-item list created subsequently. A tag is defined as either an atom declared as a TEX-item or one of the following:

(F)	insert operator
(X)	insert non-associative argument
(Y)	insert a left- or right-associative argument
(Z)	insert superscript/subscript argument
(R)	use tail recursion to unify remaining arguments (necessary with operators having more than two arguments, e.g. the plus operator; associativity depends on previous (X) or (Y) tag)
(L *hs*)	insert a list of arguments (eat up all arguments on passing by); put *hs* as a horizontal separator between the arguments (e.g., a separator could be a comma for simple argument lists.)
(M *vs hs*)	insert a matrix (and eat up all arguments on passing by); put *vs* as a vertical separator and *hs* as a horizontal separator between the rows and columns
(APPLY *fun*)	apply function *fun* to remaining argument list

These "tags" are assembled to a tag-list or pattern, respectively. For each functor (i.e. the head of a prefix list, e.g. `PLUS`, `MINUS` or `SQRT`) such a list is bound to its property `'TEXPATT`. For instance, the functor `PLUS` has got the pattern `((X) (F) (R))` bound to it, and the functor `EXPT` possesses the pattern `((X) ^{ (Z) })`. The following two boxes with pseudo-code (figures 3 and 4) survey the two major functions performing the expansion of a prefix REDUCE-expression into a TeX-item list.

At this point, the way we use our LISP pseudo-code should be explained. Words typeset in boldface are reserved words, e.g. **begin** and **end**. We use a PASCAL-like syntax which is actually used by REDUCE-Lisp, too, but with a few differences: we use

[11] in the terminology of the programming language Prolog

```
function mktag(tag,outer-precedence,associative);
begin
    if ⟨ tag is empty ⟩ then return nil
    else if ⟨ tag is an atom ⟩ then return ⟨ get the TEX-item for tag ⟩
    else begin { the tag is a list }
        precedence←⟨ precedence of this tag or 999 ⟩
        { now expand the expression, the first element is the }
        { functor, the following elements are the arguments }
        term←makefunc(car tag,cdr tag,precedence);
        { check for parentheses: term is surrounded by parentheses in order }
        { to prevent it from overruling by precedence }
        if (associative and (precedence = outer-precedence))
            or (precedence < outer-precedence)
        then term←⟨ put a pair of brackets around term ⟩;
        return term
    end
end;
```

Fig. 3: The function `mktag`. This function deals with the transformation from prefix notation to TEX notation. One important task of it is to decide whether or not brackets should be placed around the term.

the word **function** to indicate that a value is returned[12], and we use **return** to return the value of the function and therefore to exit the function. This is in contrast to the use of `return` in REDUCE-Lisp, where `return` is used only to return the value of a begin-end-block. Identifiers are printed in italics. Where identifiers are used as logical values, e.g. in conditions, they are either false if their value is **nil** or true otherwise, regardless of their exact value. Pseudo-operations are printed in roman and are put in angle brackets. Comments, too, are printed in roman but they are put in curly brackets. Assignments are typeset by the assignment operator ←, thus indicating the direction of assignment. Semicolons are used (as in PASCAL and REDUCE) as separators. In order to improve readability, mathematical expressions are given in mathematical form instead of real code. Finally, the operator `::=` is used to identify a pseudo-code-operation with its real code. We do not provide proper data type declarations for variables since this seems to be superfluous in LISP where you only deal with atoms and lists.

You can bind a TEX-item to any REDUCE atom (except the operators) you like. This is supported by binding the TEX-item to the specific atom by its property `'TEXNAME`. You can choose to have some default `'TEXNAME` properties for your variables. Function `makeset` defines a set of such default names. At the moment, two sets are provided for greek and for lowercase letters. Refer to the User's Guide for how you can use them.

But now turn back to the state of modifications our example term has undergone. With our set of functions we have expanded the prefix form into a TEX-item list consisting of single TEX-items such as numbers, letters, TEX-macros, TEX-primitives and other TEX symbols. The result is shown in figure 5. The `\cdot` command is the multiplication sign, whereas `^{` indicates the beginning of a superscript. (The term has been edited by hand to provide for proper indentation.)

The last box in this chapter (i.e. figure 6) is a verbatim copy of the output from TRI for our example. Because our example will be used to demonstrate line-breaking, too, some additional commands appear which won't occur in normal TEX-mode. These

[12] REDUCE-Lisp uses the phrase "symbolic procedure" here.

```
function makefunc(functor,argument-list,precedence);
begin
    term←nil;
    pattern←⟨ pattern of this functor or default pattern ⟩;
    while pattern do { as long as pattern isn't empty }
    begin
        tag←car pattern;
        pattern←cdr pattern;
        if ⟨ tag is an atom ⟩ then aux←nil
        else if ⟨ tag is (F) ⟩ then aux←⟨ get the TeX-item for functor ⟩
        else if ⟨ argument-list is empty ⟩ then aux←nil
        else if ⟨ tag is (X) ⟩ then
        begin
            aux←mktag(car argument-list,precedence,nil);
            argument-list←cdr argument-list
        end
        else if ⟨ tag is (Y) ⟩ then
        begin
            aux←mktag(car argument-list,precedence,T);
            argument-list←cdr argument-list
        end
        else if ⟨ tag is (R) ⟩ then { tail recursive pattern }
            if cdr argument-list { more than one argument remaining? }
            then begin
                pattern←⟨ pattern for functor ⟩;
                argument-list←nil
            end
            else begin
                aux←mktag(car argument-list,precedence,nil);
                argument-list←cdr argument-list
            end
        else if ⟨ tag is (L hs), (M vs hs) or (APPLY xxx) ⟩ then
        begin
            aux←⟨ result from call to a special routine ⟩;
            argument-list←nil
        end
        else aux←nil;
        if aux then ⟨ concatenate it to the end of term ⟩
    end;
    return term
end;
```

Fig. 4: The function `makefunc`. As well as the function `mktag` this function performs the prefix-to-TeX notation. Its major task is to "unify" operators and their arguments with predefined patterns in order to build up lists of TeX-items.

additional commands you find at the beginning and ending of the output and as `\nl`-commands within the output. Nevertheless, the structure of the output would be much the same with our normal TeX-mode.

The actual printing of TRI output in this example is easily readable since the expres-

```
(              x ^{ 1 2 }
-   1 2 \cdot  x ^{ 1 1 } \cdot  y
+   6 6 \cdot  x ^{ 1 0 } \cdot  y ^{   2 }
- 2 2 0 \cdot  x ^{   9 } \cdot  y ^{   3 }
+ 4 9 5 \cdot  x ^{   8 } \cdot  y ^{   4 }
- 7 9 2 \cdot  x ^{   7 } \cdot  y ^{   5 }
+ 9 2 4 \cdot  x ^{   6 } \cdot  y ^{   6 }
- 7 9 2 \cdot  x ^{   5 } \cdot  y ^{   7 }
+ 4 9 5 \cdot  x ^{   4 } \cdot  y ^{   8 }
- 2 2 0 \cdot  x ^{   3 } \cdot  y ^{   9 }
+   6 6 \cdot  x ^{   2 } \cdot  y ^{ 1 0 }
-   1 2 \cdot  x           \cdot  y ^{ 1 1 }
+                                 y ^{ 1 2 } )
```

Fig. 5: A TEX-item list. A TEX-item is either a letter, a digit or another plain character, or it is a TEX-command. Every TEX-item belongs to one out of eight TEX-item-classes.

```
$$\displaylines{\qdd
x^{12}
-12\cdot x^{11}\cdot y
+66\cdot x^{10}\cdot y^{2}
-220\cdot x^{9}\cdot y^{3}
+495\cdot x^{8}\cdot y^{4}
-792\cdot x^{7}\cdot y^{5}\nl
+924\cdot x^{6}\cdot y^{6}
-792\cdot x^{5}\cdot y^{7}
+495\cdot x^{4}\cdot y^{8}
-220\cdot x^{3}\cdot y^{9}
+66\cdot x^{2}\cdot y^{10}
-12\cdot x\cdot y^{11}
+y^{12}
\Nl}$$
```

Fig. 6: Output produced by the TRI. This TEX-code has to be postprocessed by TEX. This example includes commands for line-breaking as produced with the second level of TRI.

sion is not deeply nested. Complications arise if expressions to be printed are deeply nested, use many subscripts and superscripts, have fractions and large operators and the like. Then output structure is worsened, especially if the whole expression extends over several lines. We provide a "cheap" way of indentation to retain some of the structure, but our solution is far from perfect. As the need for post-TRI-editing rises the output from TRI should be made better. However, our quick-and-dirty solution should suffice.

4 Breaking REDUCE expressions into lines

As mentioned earlier, there are a few properties bound to each TEX-item, two of them dealing with line-breaking. The following list gives you a survey of these two properties and the values they can take:

`'CLASS` one of the following class specifiers
- `'ORD` ordinary symbols
- `'LOP` large operators, such as integrals
- `'BIN` binary operators
- `'REL` relational operators
- `'OPN` opening symbols (left parentheses)
- `'CLO` closing symbols (right parentheses)

`'PCT` punctuation symbols
`'INN` inner TEX group delimiters

`'TEXTAG` this is either an atom describing an `'INN` class or a list of widths defining the width of a TEX-item, where succeeding elements of the list will be used depending on the TEX inner group level (i.e. the nesting of subscripts or superscripts)

Glue items are to be inserted between consecutive TEX-items (similar to what TEX does with its items). The following table specifies for each left and right class of a TEX-item the corresponding glue measure. The glue item values have following meanings: 0 = no space, 1 = thin space, 2 = medium space, and 3 = thick space. An asterisk means that this case never arises, and values put in brackets indicate no space in the case of sub- or superscripts.

Left Class	Right Class							
	ORD	LOP	BIN	REL	OPN	CLO	PCT	INN
ORD	0	1	(2)	(3)	0	0	0	0
LOP	1	1	*	(3)	0	0	0	(1)
BIN	(2)	(2)	*	*	(2)	*	*	(2)
REL	(3)	(3)	*	0	(3)	0	0	(3)
OPN	0	0	*	0	0	0	0	0
CLO	0	1	(2)	(3)	0	0	0	0
PCT	(1)	(1)	*	(1)	(1)	(1)	(1)	(1)
INN	0	1	(2)	(3)	(1)	0	(1)	0

Actually, a glue item is a list consisting of two elements — a width info characterizing the width of this item (in scaled points) and a "penalty" to be used if a line should be broken at this point. The algorithm for inserting glue items is easily described: for every consecutive pair of TEX-items, get their classes and create a glue item according to the value found in the glue item table. For some special TEX-items use special penalties instead of the default values. That's all.

Let us return to our example from the last chapter. When the functions `insertglue` and `interglue` have finished, the TEX-item list will be left (temporarily) extended with glue items. You can find them as the two-element lists in the example. All glue items there have (by chance) the same width 163840. But they have different penalties 0, 50 and -390. The latter therefore indicates a very good breaking point because it is a negative penalty, i.e. a bonus. See the following figure 7 for the changes made to the TEX-item list.

Setting break points requires the creation of a "breaklist". A breaklist is a sequence of passive and active nodes, where each active node is followed by a passive node and vice versa. Active nodes represent glue items. Passive nodes are integer atoms which represent the width of a sequence of ordinary TEX-items which must not be interspersed with glue items. Each breaklist consists of (at least one) passive nodes surrounded by delta nodes representing the beginning and ending of the list.

breaklist ::= (*delta-node inner-list delta-node*)
inner-list ::= *passive-node active-node ... passive-node*
active-node ::= (*width penalty offset*)
passive-node ::= *width*

```
(                          x ^{ 1 2 } (163840 0)
-   1 2 \cdot  (163840 50) x ^{ 1 1 } \cdot  (163840 50) y          %
(163840 -390)
+   6 6 \cdot  (163840 50) x ^{ 1 0 } \cdot  (163840 50) y ^{   2 } %
(163840    0)
- 2 2 0 \cdot  (163840 50) x ^{   9 } \cdot  (163840 50) y ^{   3 } %
(163840 -390)
+ 4 9 5 \cdot  (163840 50) x ^{   8 } \cdot  (163840 50) y ^{   4 } %
(163840    0)
- 7 9 2 \cdot  (163840 50) x ^{   7 } \cdot  (163840 50) y ^{   5 } %
(163840 -390)
+ 9 2 4 \cdot  (163840 50) x ^{   6 } \cdot  (163840 50) y ^{   6 } %
(163840    0)
- 7 9 2 \cdot  (163840 50) x ^{   5 } \cdot  (163840 50) y ^{   7 } %
(163840 -390)
+ 4 9 5 \cdot  (163840 50) x ^{   4 } \cdot  (163840 50) y ^{   8 } %
(163840    0)
- 2 2 0 \cdot  (163840 50) x ^{   3 } \cdot  (163840 50) y ^{   9 } %
(163840 -390)
+   6 6 \cdot  (163840 50) x ^{   2 } \cdot  (163840 50) y ^{ 1 0 } %
(163840    0)
-   1 2 \cdot  (163840 50) x          \cdot  (163840 50) y ^{ 1 1 } %
(163840 -390)
+                                                    y ^{ 1 2 } )
```

Fig. 7: A TEX-item list extended with glue items.

delta-node ::= *active-node* + *appendix*
appendix ::= (*id-number ptr demerits indentation*)

The breaklist will be created using the function `breaklist`. line breaking is performed with this list only; the TEX-item list becomes modified only indirectly since the active nodes are shared. That means that the active nodes aren't copied while creating the breaklist. Instead, their memory addresses are put into the breaklist as a reference. This is both memory saving and necessary, since later we deal with the TEX-item list itself again in order to insert `\nl`-commands. So remember there exist two lists sharing all the active nodes (and hence all the delta nodes). Figure 8 contains the breaklist from our $(x-y)^{12}$ example. Bear in mind that passive nodes are sums of widths. The first line and the last line contain the beginning and ending delta nodes, respectively. By default, their *id-numbers* are 0 and -1, respectively.

The task of setting the break points (i.e. break items) in the breaklist is up to the function `trybreak`. During this phase, some active nodes are selected as "feasible" break points. Thus, they will be extended and called "delta nodes" furtheron. By default, the first and last node in a breaklist are delta nodes. When trybreak has finished, the *ptr*'s of the delta nodes point back to the best preceding delta node in terms of minimal total demerits. So, by stepping through this pointer list, it is easy to find the best path for breaking the whole breaklist apart. We use some terminology we'd like to explain:

width	width of this item (both active and passive nodes)
penalty	a numeric value which prohibits line breaking (if negative, line breaking will be merited)
offset	distance to the most recent opening bracket
id-number	the identification number of this delta node (1,2,3,...)
ptr	pointer to the best delta node to come from with respect to the minimal demerits path. (Note: a zero pointer indicates the very beginning

```
((0 0 0 0 0 0 0)
 915227 (163840    0 0) 1347128 (163840 50 0) 1097271 (163840 50 0)
 321308 (163840 -390 0) 1347128 (163840 50 0) 1097271 (163840 50 0)
 598015 (163840    0 0) 1674808 (163840 50 0)  820564 (163840 50 0)
 598015 (163840 -390 0) 1674808 (163840 50 0)  820564 (163840 50 0)
 598015 (163840    0 0) 1674808 (163840 50 0)  820564 (163840 50 0)
 598015 (163840 -390 0) 1674808 (163840 50 0)  820564 (163840 50 0)
 598015 (163840    0 0) 1674808 (163840 50 0)  820564 (163840 50 0)
 598015 (163840 -390 0) 1674808 (163840 50 0)  820564 (163840 50 0)
 598015 (163840    0 0) 1674808 (163840 50 0)  820564 (163840 50 0)
 598015 (163840 -390 0) 1347128 (163840 50 0)  820564 (163840 50 0)
 874722 (163840    0 0) 1347128 (163840 50 0)  543857 (163840 50 0)
 874722 (163840 -390 0) 1384446
 (0 0 41140184 -1 0 2147483647 0))
```

Fig. 8: A breaklist. Three types of objects are included in a breaklist. Active nodes are the lists with three elements. Delta nodes contain exactly seven elements. Passive nodes are integer atoms representing a width.

	of the breaklist)
demerits	total demerits accumulated so far
indentation	amount of indentation when breaking at this point

The algorithm itself will be described now. To determine the "quality" of a line we introduce a value called "badness". It simply is a heuristic describing how good-looking a line comes out. This concept is due to Knuth/Plass(1981) and is a major concept of TEX. We use a slightly different heuristic here. We do not measure badness in terms of "stretchability" and "shrinkability". Instead we measure how "full" a line is, where "full" means that three quarters of the page width are optimal. Furthermore we add a surplus badness for the indentation: the less indentation the better. The badness is a value between 0 and 10000 and is calculated with the following code (displayed in figure 9). Surprisingly, we got a higher speed with floating point arithmetic here than with integer arithmetic.

function *badnessof*(*length*,*indentation*);
begin
  temp←**abs**(*length*$-\frac{3}{4}\cdot$*pagewidth*)/($\frac{1}{6}\cdot$*pagewidth*);
  return min(10000, $100 \cdot temp^3 + 2500\cdot$*indentation*/*pagewidth*)
end;

Fig. 9: The badness function. "Badness" is just a heuristic to compute a numerical value describing how "good-looking" a line comes out. A correction term is applied to provide for indentation.

Figure 10 summarizes the line breaking algorithm. The code is part of the function `trybreak` and describes the "heart" of our algorithm. Basically, it consists of two loops: the outer loop steps through the breaklist considering each delta node as a potential start of a new line while the inner loop looks ahead exactly one line (bounded by the *page-width* or by the rightmost delta-node) checking each active node if it is a feasible breakpoint, and if so, saving it as the best path of breaking. Or to put it in another way, simply imagine a window as wide as a page which moves over the unbroken expression from the very left to the very right. The left end of the window is put on every feasible breakpoint determined earlier. The right end of the window just defines the border of the search for feasible breakpoints within the window.

```
bottom←breaklist;  pagewidth← ⟨ user defined page width ⟩;
⟨ set all variables not mentioned explicitly to zero or nil ⟩;
while bottom do
begin { try a new line starting at this delta node }
    base←car bottom; top←cdr bottom;
    baseid←idof base; baseptr←ptrof base;
    basedemerits←demeritsof base;
    baseoffset←offsetof base;
    baseindent←length←indentof base;
    total←total+widthof base;
    while top and length<pagewidth do
    begin { consider this node for end of the line }
        node←car top;
        penalty←penaltyof node;
        if ⟨ node is a passive node ⟩
        then len←len+node
        else begin { node is an active node }
            badness← ⟨ compute current badness from length and baseindent ⟩
            penalty←penaltyof node; offset←offsetof node;
            if badness < tolerance
            or badness < 1−penalty
            or ⟨ this is the rightmost delta node ⟩
            then begin { we have a feasible breakpoint }
                demerits←basedemerits+badness^2+penalty·abs(penalty);
                if ⟨ node is a delta node ⟩
                then begin
                    if demerits<demeritsof node { better path found? }
                    then begin
                        ⟨ save current demerits and baseid to node ⟩;
                        ⟨ compute amount of indentation ⟩
                    end
                end
                else begin
                    feasible←feasible+1;
                    ⟨ create new delta node with feasible, baseid, demerits ⟩;
                    ⟨ compute amount of indentation ⟩
                end;
                if penalty= −10000 then top←nil{ must break here }
            end;
            length←length+wdithof node
        end;
        if top then top←cdr top
    end;
    ⟨ save the total length so far to the delta node ⟩;
    ⟨ step ahead to next delta node and count total length ⟩
end;
```

Fig. 10: The code segment from `trybreak` dealing with line-breaking.

Earlier we introduced the concept of "badness" derived from TEX. But actually this

is not the only measure answering the question whether a certain point in the breaklist is feasible or not. There are three conditions which decide whether a breakpoint is feasible or not. The first condition requires the badness to be smaller than the value of *tolerance* as specified by the user. This condition can be overridden if the active node under consideration has a negative penalty whose (absolute, i.e. positive) amount is greater than the badness. That means you can buy a breakpoint if you've got enough money (i.e. a bonus = negative penalty) to pay the price (i.e. the badness). The third condition just forces the rightmost delta node to be considered feasible anyway.[13] At this point we introduce a second measure called "demerits" which is defined as the sum of the demerits so far (i.e. up to the beginning of the line currently under consideration), the squared badness and the sign-propagated square of the penalty.[14] Now we have a measure which not only refers to the current line but to the previous lines, too. Therefore, our modified KNUTH-algorithm "optimizes" not only over *one* line but over *all* lines.

Figure 11 should make clear what is happening to the breaklist and the TEX-item list. For readability purposes we display the latter, but really it is the breaklist under consideration within `trybreak`. As in the previous display of our example, you can identify the active-nodes. These are the three element number lists. The third element which is zero here is used as an offset width to the last opening bracket. This information is used for indentation. The delta-nodes are remarkable. These are the number lists with seven elements. Count them by their fourth element. They run 0, 1, 2 through 8, 9, -1. The fifth element gives you the way back through the list. Start at the last delta-node. There the best way to come from is 1. So go to delta-node 1 where you find 0 as the best way to come from. Delta-node 0 is the beginning, so you're finished. The sixth element stands for the total demerits so far. The seventh element stands for the amount of indentation. Here it is a zero because the term isn't nested in our example.

We haven't mentioned how we generate indentation yet. Generally speaking, indentation is entirely directed by brackets, either round, curly, or dummy brackets. But how do we compute the amount of indentation? First let's turn to the code segment which deals with this problem within the big line-braking algorithm shown before. The contents of figure 12 explains what happens to the amount of indentation.

```
⟨ compute amount of indentation ⟩ ::=
begin
    if offset>total
    then indent←offset−total+baseindent    { opening bracket case }
    else if offset<baseoffset
        then indent←findindent()           { closing bracket case }
        else indent←baseindent;            { no change case }
    ⟨ save indent to delta-node ⟩
end;
```

Fig. 12: The code segment from `trybreak` dealing with indentation.

The logic of this code segment is easily summarized. The *offset* is a measure of how distant the last opening bracket is from the beginning of the whole expression. So in the

[13] This is necessary since the end of the expression is naturally a breakpoint even if it is a bad one, and the last delta node is needed for accounting purposes as well as for storing the pointer to the preceding breakpoints.

[14] The latter is evaluated as the product of the value (which can be positive or negative) and the absolute value (which can be positive only).

```
(                                          (     0        0 0 0 0       0 0)
                              x ^{ 1 2 } (163840    0 0)
- 1 2   \cdot  (163840 50 0) x ^{ 1 1 }
        \cdot  (163840 50 0) y          (163840 -390 0)
+ 6 6   \cdot  (163840 50 0) x ^{ 1 0 }
        \cdot  (163840 50 0) y ^{   2 } (163840    0 0)
- 2 2 0 \cdot  (163840 50 0) x ^{   9 }
        \cdot  (163840 50 0) y ^{   3 } (163840 -390 0)
+ 4 9 5 \cdot  (163840 50 0) x ^{   8 }
        \cdot  (163840 50 0) y ^{   4 } (163840    0 0)
- 7 9 2 \cdot  (163840 50 0) x ^{   7 }
        \cdot  (163840 50 0) y ^{   5 } (163840 18624949 0 1 0 -151875 0)
+ 9 2 4 \cdot                           (163840 20463597 0 2 0    2500 0)
                              x ^{   6 }
        \cdot                           (163840 21448001 0 3 0    2500 0)
                              y ^{   6 } (163840 22209856 0 4 0       1 0)
- 7 9 2 \cdot  (163840 50 0) x ^{   5 }
        \cdot  (163840 50 0) y ^{   7 } (163840 25794763 0 5 0 -142299 0)
+ 4 9 5 \cdot  (163840 50 0) x ^{   4 }
        \cdot  (163840 50 0) y ^{   8 } (163840 0 0)
- 2 2 0 \cdot  (163840 50 0) x ^{   3 }
        \cdot  (163840 50 0) y ^{   9 } (163840 32964577 0 6 1 -207875 0)
+ 6 6   \cdot  (163840 50 0) x ^{   2 }
        \cdot  (163840 50 0) y ^{ 1 0 } (163840 0 0)
- 1 2   \cdot                           (163840 38009479 0 7 1 -149350 0)
                              x
        \cdot                           (163840 38717176 0 8 1 -149374 0)
                              y ^{ 1 1 } (163840 39755738 0 9 1 -303975 0)
+ y ^{ 1 2 }                            (     0 41140184 ?-1 1 -151866 ?)
```

Fig. 11: A TEX-item list extended with delta-nodes. From this list the line-breaking way can be derived. Start with node –1, go to node 1 and from there to node 0. That makes one break point at node 1.

case where *offset* is greater than the total width accumulated until the very beginning of the line, the indentation is just the difference between *total* and the sum of *offset* and the amount of indentation *baseindent* for the currently line. That'll work if the line currently under consideration contains at least one opening bracket which hasn't become closed in the same line. This case may be labelled the "opening bracket case". But what shall we do in the other cases? We have to decide if we've got a "closing bracket case", i.e. if we have at least one more closing bracket than we have opening brackets, or if we've got a "no change case", i.e. the number of opening brackets in the current line matches the number of closing brackets in this line. This decision is made by comparing *offset*<*baseoffset*. If the *baseoffset*, i.e. the offset at the beginning of the line, is greater than *offset*, i.e. the offset at the current point, then we've got the "closing bracket case", otherwise we've got the "no change case". In the latter, the amount of indentation for the next line is just the amount of indentation for the current line. But the "closing bracket case" causes us much trouble. So we need an extra function dealing with this case, as displayed in figure 13.

This macro[15] searches the list of delta-nodes previously created until it reaches a delta-node (at least the very first delta-node in the breaklist) where the total width accumulated so far, i.e. the variable *local*, is less than *offset*, i.e. the offset at the

[15] Macros become expanded where they are called and thus share all variable names defined in the code block where they are called. So, there is no need for argument passing if certain variable names in the code block don't differ from call to call.

```
macro function findindent;
{ macro functions share all variables of outer code blocks }
begin { first check if we can save search time for equal destination }
    if offset=lastoffset and baseptr=lastbaseptr
    then return lastindent
    else begin { search the delta-node-stack for previous indentation }
        stack←deltastack; lastoffset←offset;
        p←lastbaseptr←baseptr;
        while stack do { as long as we have a delta-node ahead }
        begin
            node←car stack; { current delta-node }
            if p=idof node then begin
                p←ptrof node; local←totalof node;
                if local<offset then stack←nil
            end;
            if stack then stack←cdr stack
        end;
        lastindent←offset−local+indentof node;
        return lastindent
    end
end;
```

Fig. 13: The macro function findindent. This macro is used to compute the amount of indentation in the "closing bracket case". It causes some trouble since we have to travel back to previous delta nodes.

end of the line under consideration, and computes the amount of indentation from the difference of *offset* and *local* plus the amount of indentation so far. Plainly speaking, we go back the lines until we find a line where we find the opening parenthesis matching the closing parenthesis in the current line. When we've found it, we compute the amount of indentation as described in the "opening bracket case", but with *local* instead of *total*.

5 Postprocessing with the TEX module "tridefs.tex"

When a TEX-output-file has been created with the TRI it has to be processed by TEX itself. But before you run TEX you should make sure the file looks the way you want it. Sometimes you will find it necessary to add some TEX-code of your own or delete some other TEX-code. This job is up to you before you finally run TEX.

During the TEX-run the sizes of brackets are determined. This task is not done by the TRI. In order to produce proper sized brackets we put some `\left(` and `\right)` TEX-commands where brackets are opened or closed. A new problem arises when an expression has been broken up into several lines. Since, for every line, the number of `\left(` and `\right)` TEX-commands must match but bracketed expressions may start in one line and end in another, we have to insert the required number of "dummy" parentheses (i.e. `\right.` and `\left.` TEX-commands) at the end of the current line and the beginning of the following line. Therefore, we have to keep track of the depth of bracketing. See the following figure 14 for the TEX-code actually applied.

There is a caveat against this method. Since opening and closing brackets needn't lie in the same line, it is possible that the height of the brackets can differ although they should correspond in height. That will happen if the height of the text in the opening line has a height different from the text in the closing line. We haven't found a way of

tackling this problem yet, but we think it is possible to program a little TEX-macro for this task. Furthermore, some macros deal with tricks we had to use in order to provide for indentation, fraction handling and the like.

```
\def\qdd{\quad\quad}          % simply a double quad
\def\frac#1#2{{#1\over#2}}    % fractions from prefix notation
\newcount\parenthesis         % nesting of brackets
\parenthesis=0                % intialize
\newcount\n                   % a temporary variable
% ---- round and curly brackets ----
\def\({\global\advance\parenthesis by1\left(}
\def\){\global\advance\parenthesis by-1\right)}
\def\{{\global\advance\parenthesis by1\left\lbrace}
\def\}{\global\advance\parenthesis by-1\right\rbrace}
\def\[{\relax} % dummy parenthesis
\def\]{\relax} % dummy parenthesis
% ---- provide for looping using tail recursion ----
%      \loop ...what... \repeat
\def\loop#1\repeat{\global\n=0\global\let\body=#1\iterate}
\def\iterate{\body\let\next=\iterate\else\let\next=\relax\fi\next}
% ---- conditions and statements for loop interior
\def\ldd{\ifnum\n<\parenthesis\global\advance\n by1
\left.\nulldelimiterspace=0pt\mathsurround=0pt}
\def\rdd{\ifnum\n<\parenthesis\global\advance\n by1
\right.\nulldelimiterspace=0pt\mathsurround=0pt}
% ---- newline statement as issued by TRI ----
\def\nl{\loop\rdd\repeat\hfill\cr\quad\quad\loop\ldd\repeat{}}
% ---- indentation statement as issued by TRI ----
\def\OFF#1{\hskip#1sp\relax}
% ---- last newline statement before end of math group ----
\def\Nl{\hfill\cr}
```

Fig. 14: The file "tridefs.tex". This is the code you have to use on the TEX side in order to typeset output produced by our TRI.

There is at least one more line of TEX-code you have to insert by hand into the TEX-input-file produced by TRI. This line runs

```
\input tridefs
```

and inputs the module `tridefs.tex` into the file. This is necessary because otherwise TEX won't know how to deal with our macro calls. If you use the TEX-input-file as a "stand-alone" file, don't forget a final `\bye` at the end of the text. If you use code produced by TRI as part of a larger text then simply put the input-line just once at the beginning of your text.

6 Experiments

We measured performance using the TIME-facility of REDUCE, which can be switched on and off with the two commands

```
ON TIME;
OFF TIME;
```

We have tested our TRI on a μVAX-II operating under VAX/VMS, with no other users operating during this phase in order to minimize interference with other processes, e.g. caused by paging. The TRI code has been compiled with the PSL 3.2a-Compiler. The following table presents results obtained with a small number of different terms. All data were measured in CPU-seconds as displayed by LISP's TIME-facility. For expressions where special packages such as `solve` and `int` were involved we have taken only effective output-time, i.e. the time consumption caused by producing the output and not by evaluating the algebraic result.[16]

REDUCE-Expression	nor-mal	TeX	TeX-Break	TeX-Indent
$(x+y)^{12}$	0.82	0.75	3.42	3.47
$(x+y)^{24}$	2.00	2.22	12.52	12.41
$(x+y)^{36}$	4.40	4.83	21.48	21.44
$(x+y)^{16}/(v-w)^{16}$	2.27	2.38	12.18	12.19
$solve((1+\xi)x^2-2\xi x+\xi,x)$	0.41	0.62	0.89	0.87
$solve(x^3+x^2\mu+\nu,x)$	4.21	20.84	31.82	40.43

This short table should give you an impression of the performance of TRI. It goes without saying that on other machines results may turn out which are quite different from our results. But our intention is to show the relative and not the absolute performance. Note that printing times are a function of expression complexity, as shown by rows three and six.

7 User's Guide to the REDUCE-TEX-Interface

If you intend to use the TRI you are required to load the compiled code. This can be performed with the command

```
load!-package 'tri;
```

During the load, some default initializations are performed. The default page width is set to 15 centimeters, the tolerance for page breaking is set to 20 by default. Moreover, TRI is enabled to translate greek names, e.g. TAU or PSI, into equivalent TEX symbols, e.g. τ or ψ, respectively. Letters are printed lowercase as defined through assertion of the set LOWERCASE. The whole operation produces the following lines of output

```
***Function TEXVARPRI redefined
% set GREEK asserted
% set LOWERCASE asserted
% \hsize=150mm
% \tolerance 20
```

Now you can switch on and off the three TRI modes as you like. You can use the switches alternatively and incrementally. That means you have to switch on TeX for receiving standard TEX-output, or TeXBreak to receive broken TEX-output, or TeXIndent to

[16] That means we assigned the result of an evaluation to an itermediate variable, and then we printed this intermediate variable. Thus we could eliminate the time overhead produced by "pure" evaluation. Nevertheless, in terms of effective interactive answering time, the sum of evaluation and printing time might be much more interesting than the "pure" printing time. In such a context the percentage overhead caused by printing is the critical point. But since we talk about printing we decided to document the "pure" printing time.

receive broken TEX-output plus indentation. Thus, the three levels of TRI are enabled or disabled according to:

```
On TeX; % switch TeX is on
On TeXBreak; % switches TeX and TeXBreak are on
On TeXIndent; % switches TeX, TeXBreak and TeXIndent are on
Off TeXIndent; % switch TeXIndent is off
Off TeXBreak; % switches TeXBreak and TeXIndent are off
Off TeX; % all three switches are off
```

More specifically, if you switch off `TeXBreak` you implicitly quit `TeXIndent`, too, or, if you switch off `TeX` you implicitly quit `TeXBreak` and, consequently, `TeXIndent`.

The most crucial point in defining how TRI breaks multiple lines of TEX-code is your choice of the page width and the tolerance. As mentioned earlier, "tolerance" is related to TEX's famous line-breaking algorithm. The value of "tolerance" determines which potential breakpoints are considered feasible and which not. The higher the tolerance, the more breakpoints become feasible as determined by the value of "badness" associated with each breakpoint. Breakpoints are considered feasible if the badness is less than the tolerance. You can easily set values for page width and tolerance using

`TeXsetbreak(`*page-width*`,`*tolerance*`);`

where *page-width* is measured in millimeters[17] and the *tolerance* is a positive integer in the closed interval $[0 \ldots 10000]$. You should choose a page width according to your purposes, but allow a few centimeters for errors in TRI's metric. For example, specify 140 millimeters for an effective 150 or 160 millimeter wide page. That way you have a certain safety-margin to the borders of the page. Now let's turn to the tolerance. A tolerance of 0 means that actually no breakpoint will be considered feasible (except those carrying a negative penalty), while a value of 10000 allows any breakpoint to be considered feasible. Obviously, the choice of a tolerance has a great impact on the time consumption of our line-breaking algorithm since time consumption increases in proportion to the number of feasible breakpoints. So, the question is what values to choose. For line-breaking without indentation, suitable values for the tolerance lie between 10 and 100. As a rule of thumb, you should use higher values the deeper the term is nested — if you can estimate. If you use indentation, you have to use much higher tolerance values. This is necessary because badness is worsened by indentation. So, TRI has to try harder to find suitable places where to break. Reasonable values for tolerance here lie between 700 and 1500. A value of 1000 should be your first guess. That'll work for most expressions in a reasonable amount of time.

Sometimes you want to add your own REDUCE-symbol-to-TEX-item translations. For such a task, TRI provides a function named `TeXlet` which binds any REDUCE-symbol to one of the predefined TEX-items. A call to this function has the following syntax:

`TeXlet(`*REDUCE-symbol*`,`*TEX-item*`)`

Three examples show how to do it right:

```
TeXlet('velocity,'!v);
TeXlet('gamma,'!\!G!a!m!m!a! );
TeXlet('acceleration,'!\!v!a!r!t!h!e!t!a! );
```

[17] You can also specify page width in scaled points (sp). Note: 1 pt = 65536 sp = 1/72.27 inch. The function automatically chooses the appropiate dimension according to the size: all values greater than 400 are considered to be scaled points.

Besides this method of single assertions you can choose to assert one of (currently) two standard sets providing substitutions for lowercase and greek letters. These sets are loaded by default. You can switch these sets on or off using the functions

```
TeXassertset setname;
TeXretractset setname;
```

where the setnames currently defined are `'GREEK` and `'LOWERCASE`. So far you have learned only how to connect REDUCE-atoms with predefined TEX-items but not how to create new TEX-items itself. We provide a way for adding standard TEX-items of any class `'ORD, 'BIN, 'REL, 'OPN, 'CLO, 'PCT` and `LOP` except for class `'INN` which is reserved for internal use by TRI only. You can call the function

```
TeXitem(item,class,list-of-widths)
```

e.g. together with a binding

```
TeXitem('!\!n!a!b!l!a! ,'ORD,'(655360 327680 163840))
TeXlet('NABLA,'!\!n!a!b!l!a! );
```

where *item* is a legal TEX-code name[18], *class* is one of seven classes (see above) and *list-of-width* is a non-empty list of elements each one representing the width of the item in successive super-/subscript depth levels. That means that the first entry is the breadth in display mode, the second stands for scriptstyle and the third stands for scriptscriptstyle in TEX-terminology. But how can you retrieve the width information required? For this purpose we provide the following small interactive TEX facility called `redwidth.tex` documented in figure 15. Simply call

```
tex redwidth
```

on your local machine. Then you are prompted for the TEX-item you want the width information for. Type "end" when you want to finish the session.

```
\newbox\testbox\newcount\xxx\newif\ifOK\def\endloop{end }
\def\widthof#1{\message{width is: }\wwidthof{$\displaystyle#1$}
\wwidthof{$\scriptstyle#1$}\wwidthof{$\scriptscriptstyle#1$}}
\def\wwidthof#1{\setbox\testbox=\hbox{#1}\xxx=\wd\testbox
\message{[\the\wd\testbox=\the\xxx sp]}}
\loop\message{Type in TeX item or say 'end': }\read-1 to\answer
  \ifx\answer\endloop\OKfalse\else\OKtrue\fi
  \ifOK\widthof{\answer}
\repeat
\end
```

Fig. 15: The file "redwidth.tex". This TEX code you can use to determine the width of specific TEX-items.

Finally let us discuss how you can compile the TRI into a binary. (We refer to PSL, but other LISP versions work quite similar.) First of all start REDUCE. Than type in

```
on comp;
symbolic;
```

[18] Please note that any TEX-name ending with a letter must be followed by a blank to prevent from interference with letters of following TEX-items. Note also that you can legalize a name by defining it as a TEX-macro.

```
faslout "tri";
in "tri.red";
faslend;
bye;
```

We stress the fact that this procedure is definitely LISP dependent. Ask your local REDUCE or LISP wizards how to adapt to it.

8 Examples

Some examples — which we think might be representative — shall demonstrate the capabilities of our TRI. For each example we state (a) the REDUCE command (i.e. the input), (b) the tolerance if it differs from the default, and (c) the output as produced in a TEX run.

1	Standard	MODE: `TeXindent`	TOLERANCE: `250`
`(x+y)**16/(v-w)**16;`			

$$\begin{aligned}&\left(x^{16}+16\cdot x^{15}\cdot y+120\cdot x^{14}\cdot y^2+560\cdot x^{13}\cdot y^3\right.\\&\quad+1820\cdot x^{12}\cdot y^4+4368\cdot x^{11}\cdot y^5+8008\cdot x^{10}\cdot y^6\\&\quad+11440\cdot x^9\cdot y^7+12870\cdot x^8\cdot y^8+11440\cdot x^7\cdot y^9\\&\quad+8008\cdot x^6\cdot y^{10}+4368\cdot x^5\cdot y^{11}+1820\cdot x^4\cdot y^{12}\\&\quad\left.+560\cdot x^3\cdot y^{13}+120\cdot x^2\cdot y^{14}+16\cdot x\cdot y^{15}+y^{16}\right)/\\&\left(v^{16}-16\cdot v^{15}\cdot w+120\cdot v^{14}\cdot w^2-560\cdot v^{13}\cdot w^3\right.\\&\quad+1820\cdot v^{12}\cdot w^4-4368\cdot v^{11}\cdot w^5+8008\cdot v^{10}\cdot w^6-11440\cdot v^9\cdot w^7\\&\quad+12870\cdot v^8\cdot w^8-11440\cdot v^7\cdot w^9+8008\cdot v^6\cdot w^{10}-4368\cdot v^5\cdot w^{11}\\&\quad\left.+1820\cdot v^4\cdot w^{12}-560\cdot v^3\cdot w^{13}+120\cdot v^2\cdot w^{14}-16\cdot v\cdot w^{15}+w^{16}\right)\end{aligned}$$

2	Integration	MODE: `TeX`	TOLERANCE: `-;`
`int(1/(x**3+2),x)`			

$$-\left(\frac{2^{\frac{1}{3}}\cdot\left(2\cdot\sqrt{3}\cdot\arctan\left(\frac{2^{\frac{1}{3}}-2\cdot x}{2^{\frac{1}{3}}\cdot\sqrt{3}}\right)+\ln\left(2^{\frac{2}{3}}-2^{\frac{1}{3}}\cdot x+x^2\right)-2\cdot\ln\left(2^{\frac{1}{3}}+x\right)\right)}{12}\right)$$

3	Integration	MODE: `TeXindent`	TOLERANCE: `1000`
`int(1/(x**4+3*x**2-1,x);`			

$$\begin{aligned}&\left(\sqrt{2}\cdot\left(3\cdot\sqrt{\sqrt{13}-3}\cdot\sqrt{13}\cdot\ln\left(-\left(\sqrt{\sqrt{13}-3}\cdot\sqrt{2}\right)+2\cdot x\right)\right.\right.\\&\quad-3\cdot\sqrt{\sqrt{13}-3}\cdot\sqrt{13}\cdot\ln\left(\sqrt{\sqrt{13}-3}\cdot\sqrt{2}+2\cdot x\right)\\&\quad\left.+13\cdot\sqrt{\sqrt{13}-3}\cdot\ln\left(-\left(\sqrt{\sqrt{13}-3}\cdot\sqrt{2}\right)+2\cdot x\right)\right.\end{aligned}$$

$$
\begin{aligned}
&- 13 \cdot \sqrt{\sqrt{13}-3} \cdot \ln\left(\sqrt{\sqrt{13}-3} \cdot \sqrt{2} + 2 \cdot x\right) \\
&+ 6 \cdot \sqrt{\sqrt{13}+3} \cdot \sqrt{13} \cdot \arctan\left(\frac{2 \cdot x}{\sqrt{\sqrt{13}+3} \cdot \sqrt{2}}\right) \\
&- 26 \cdot \sqrt{\sqrt{13}+3} \cdot \arctan\left(\frac{2 \cdot x}{\sqrt{\sqrt{13}+3} \cdot \sqrt{2}}\right)\Bigg)\Bigg) /104
\end{aligned}
$$

4	Solving Equations	Mode: `TeXindent`	Tolerance: `1000`
`solve(x**3+x**2*mu+nu=0,x);`			

$$
\begin{aligned}
\Bigg\{ x = -\Bigg(\Bigg(\Big(&9 \cdot \sqrt{4 \cdot \mu^3 \cdot \nu + 27 \cdot \nu^2} - 2 \cdot \sqrt{3} \cdot \mu^3 - 27 \cdot \sqrt{3} \cdot \nu\Big)^{\frac{2}{3}} \cdot \sqrt{3} \cdot i \\
&+ \Big(9 \cdot \sqrt{4 \cdot \mu^3 \cdot \nu + 27 \cdot \nu^2} - 2 \cdot \sqrt{3} \cdot \mu^3 - 27 \cdot \sqrt{3} \cdot \nu\Big)^{\frac{2}{3}} \\
&+ 2 \cdot \Big(9 \cdot \sqrt{4 \cdot \mu^3 \cdot \nu + 27 \cdot \nu^2} - 2 \cdot \sqrt{3} \cdot \mu^3 - 27 \cdot \sqrt{3} \cdot \nu\Big)^{\frac{1}{3}} \cdot \\
&2^{\frac{1}{3}} \cdot 3^{\frac{1}{6}} \cdot \mu - 2^{\frac{2}{3}} \cdot \sqrt{3} \cdot 3^{\frac{1}{3}} \cdot i \cdot \mu^2 + 2^{\frac{2}{3}} \cdot 3^{\frac{1}{3}} \cdot \mu^2\Big) / \\
&\Big(6 \cdot \Big(9 \cdot \sqrt{4 \cdot \mu^3 \cdot \nu + 27 \cdot \nu^2} - 2 \cdot \sqrt{3} \cdot \mu^3 - 27 \cdot \sqrt{3} \cdot \nu\Big)^{\frac{1}{3}} \cdot 2^{\frac{1}{3}} \cdot 3^{\frac{1}{6}}\Big)\Bigg) \\
x = \Bigg(\Big(&9 \cdot \sqrt{4 \cdot \mu^3 \cdot \nu + 27 \cdot \nu^2} - 2 \cdot \sqrt{3} \cdot \mu^3 - 27 \cdot \sqrt{3} \cdot \nu\Big)^{\frac{2}{3}} \cdot \sqrt{3} \cdot i \\
&- \Big(9 \cdot \sqrt{4 \cdot \mu^3 \cdot \nu + 27 \cdot \nu^2} - 2 \cdot \sqrt{3} \cdot \mu^3 - 27 \cdot \sqrt{3} \cdot \nu\Big)^{\frac{2}{3}} \\
&- 2 \cdot \Big(9 \cdot \sqrt{4 \cdot \mu^3 \cdot \nu + 27 \cdot \nu^2} - 2 \cdot \sqrt{3} \cdot \mu^3 - 27 \cdot \sqrt{3} \cdot \nu\Big)^{\frac{1}{3}} \cdot \\
&2^{\frac{1}{3}} \cdot 3^{\frac{1}{6}} \cdot \mu - 2^{\frac{2}{3}} \cdot \sqrt{3} \cdot 3^{\frac{1}{3}} \cdot i \cdot \mu^2 - 2^{\frac{2}{3}} \cdot 3^{\frac{1}{3}} \cdot \mu^2\Big) / \\
&\Big(6 \cdot \Big(9 \cdot \sqrt{4 \cdot \mu^3 \cdot \nu + 27 \cdot \nu^2} - 2 \cdot \sqrt{3} \cdot \mu^3 - 27 \cdot \sqrt{3} \cdot \nu\Big)^{\frac{1}{3}} \cdot 2^{\frac{1}{3}} \cdot 3^{\frac{1}{6}}\Big) \\
x = \Bigg(\Big(&9 \cdot \sqrt{4 \cdot \mu^3 \cdot \nu + 27 \cdot \nu^2} - 2 \cdot \sqrt{3} \cdot \mu^3 - 27 \cdot \sqrt{3} \cdot \nu\Big)^{\frac{2}{3}} \\
&- \Big(9 \cdot \sqrt{4 \cdot \mu^3 \cdot \nu + 27 \cdot \nu^2} - 2 \cdot \sqrt{3} \cdot \mu^3 - 27 \cdot \\
&\sqrt{3} \cdot \nu\Big)^{\frac{1}{3}} \cdot 2^{\frac{1}{3}} \cdot 3^{\frac{1}{6}} \cdot \mu + 2^{\frac{2}{3}} \cdot 3^{\frac{1}{3}} \cdot \mu^2\Big) / \\
&\Big(3 \cdot \Big(9 \cdot \sqrt{4 \cdot \mu^3 \cdot \nu + 27 \cdot \nu^2} - 2 \cdot \sqrt{3} \cdot \mu^3 - 27 \cdot \sqrt{3} \cdot \nu\Big)^{\frac{1}{3}} \cdot 2^{\frac{1}{3}} \cdot 3^{\frac{1}{6}}\Big)\Bigg\}
\end{aligned}
$$

5	Matrix Printing	Mode: `TeX`	Tolerance: `--`
`mat((1,a-b,1/(c-d)),(a**2-b**2,1,sqrt(c)),((a+b)/(c-d),sqrt(d),1));`			

$$
\begin{pmatrix}
1 & a-b & \frac{1}{c-d} \\
a^2-b^2 & 1 & \sqrt{c} \\
\frac{a+b}{c-d} & \sqrt{d} & 1
\end{pmatrix}
$$

9 Caveats

Techniques for printing mathematical expressions are available everywhere. This TRI adds only a highly specialized version for most REDUCE output. The emphasis is on the word *most*. One major caveat is that we cannot print SYMBOLIC-mode output from REDUCE. This could be done best in a WEB-like programming-plus-documentation style. Nevertheless, as Knuth's WEB is already available for PASCAL and C, hopefully someone will write a LISP-WEB or a REDUCE-WEB as well.

Whenever you discover a bug in our program please let us know. Send us a short report accompanied by an output listing.[19] We'll try to fix the error. The whole TRI package consists of following files:

`tri.tex`	This text as a TEX-input file.
`tri.red`	The REDUCE-LISP source code for the TRI.
`tridefs.tex`	The TEX-input file to be used together with output from the TRI.
`redwidth.tex`	The TEX-input file for interactive determination of TEX-item widths.
`tritest.red`	Run this REDUCE file to check if TRI works correctly.
`tritest.tex`	When you have run `tritest.red` just make a TEX run with this file to see all the nice things TRI is able to produce.

10 References

Antweiler, W.; Strotmann, A.; Pfenning, Th.; Winkelmann, V. (1986) Zwischenbericht über den Status der Arbeiten am REDUCE-TEX-Anschluß. Internal Paper, Rechenzentrum der Universität zu Köln, November 1986.

Fateman, Richard J. (1987) TEX Output from MACSYMA-like Systems. ACM SIGSAM Bulletin, Vol. 21, No. 4, Issue #82, November 1987, pp. 1–5.

Knuth, Donald E.; Plass, Michael F. (1981) Breaking Paragraphs into Lines. Software—Practice and Experience, Vol. 11, 1981, pp. 1119–1184.

Knuth, Donald E. (1986) The TEXbook. Addison-Wesley, Readings/Ma., sixth printing, 1986.

Hearn, Anthony C. (1987) REDUCE User's Manual, Version 3.3. The RAND Corporation, Santa Monica, Ca., July 1987.

[19] You can reach us with e-mail at the following addresses: Werner Antweiler: antweil@epas.utoronto.ca, Andreas Strotmann: strotmann rrz.uni-koeln.de and Volker Winkelmann: winkelmann rrz.uni-koeln.de.