

A System for Typesetting Mathematics

B. W. Kernighan
and
L. L. Cherry

Bell Laboratories, Murray Hill, N. J.

ABSTRACT

This paper describes the design and implementation of a system for typesetting mathematics, currently running on the UNIX operating system. An appendix contains the user's manual for the language.

The language has been designed to be easy to learn and to use by people (for example, secretaries and mathematical typists) who know neither mathematics nor typesetting. Early experience indicates that the language can be learned in an hour or so, for it has few rules and fewer exceptions. For typical expressions, the size and font changes, positioning, line drawing, and the like necessary to print according to mathematical conventions are all done automatically. For example, the input

sum from $i=0$ to infinity $x_{\text{sub } i} = \pi \text{ over } 2$

produces

$$\sum_{i=0}^{\infty} x_i = \frac{\pi}{2}$$

The syntax of the language is specified by a small context-free grammar; a compiler-compiler is used to make a compiler that translates this language into typesetting commands.

Output may be produced on either a phototypesetter or on terminals with forward and reverse half-line motions. The system interfaces directly with text formatting programs, so mixtures of text and mathematics may be handled simply. This technical report is an example of its output.

A System for Typesetting Mathematics

B. W. Kernighan
and
L. L. Cherry

Bell Laboratories, Murray Hill, N. J.

1. Introduction

"Mathematics is known in the trade as *difficult*, or *penalty copy* because it is slower, more difficult, and more expensive to set in type than any other kind of copy normally occurring in books and journals." [1]

One difficulty with mathematical text is the multiplicity of characters, sizes, and fonts. An expression as simple as

$$\lim_{x \rightarrow \pi/2} (\tan x)^{\sin 2x} = 1$$

requires an intimate mixture of Roman, italic and Greek letters, in three sizes, and a special character or two. ("Requires" is perhaps the wrong word, but mathematics has its own typographical conventions which are quite different from those of ordinary text.) A compositor setting such an expression by traditional methods must sit before a large box containing a large number of pieces of lead, choosing them one at a time and fitting them together by hand.

A second area of difficulty is the two dimensional character of mathematics, which the superscript and limits in the preceding example showed in its simplest form. This is carried further by

$$a_0 + \frac{b_1}{a_1 + \frac{b_2}{a_2 + \frac{b_3}{a_3 + \dots}}}$$

and still further by

$$\int \frac{dx}{ae^{mx} - be^{-mx}} = \begin{cases} \frac{1}{2m\sqrt{ab}} \log \frac{\sqrt{a}e^{mx} - \sqrt{b}}{\sqrt{a}e^{mx} + \sqrt{b}} \\ \frac{1}{m\sqrt{ab}} \tanh^{-1} \left(\frac{\sqrt{a}}{\sqrt{b}} e^{mx} \right) \\ \frac{-1}{m\sqrt{ab}} \coth^{-1} \left(\frac{\sqrt{a}}{\sqrt{b}} e^{mx} \right) \end{cases} \quad (a > 0, b > 0)$$

These examples also show line-drawing, built-up characters like braces and radicals, and a spectrum of positioning problems. (A later section shows what one has to type to produce these on our system.)

2 TYPESETTING MATHEMATICS

2. Photocomposition

Photocomposition techniques, which have already had a substantial effect on traditional printing, can also be used to solve some of the problems of setting mathematics.

A phototypesetter is a device which exposes, for example, a piece of photographic paper, placing characters wherever they are wanted. The Graphic Systems phototypesetter[2] on the UNIX [3] operating system works by shining light through a character stencil. The character is made the right size by lenses, and the light beam is directed by fiber optics to the desired place on a piece of photographic paper. The exposed paper is developed and typically used in some form of photo-offset reproduction.

On UNIX, the phototypesetter is driven by a formatting program called TROFF [4]. TROFF is quite acceptable for setting running text, the job it was designed for. It also provides all of the facilities that one needs for doing mathematics — arbitrary horizontal and vertical motions, line-drawing, size changing, and so on — but the syntax for describing these special operations is difficult to learn, and difficult even for experienced users to type correctly.

For this reason we decided to use TROFF as an "assembly language", write a language for describing mathematical expressions, and compile it into TROFF.

3. Language Design

The fundamental principle upon which we based our language design is that the language should be easy to use by people (for example, secretaries) who know neither mathematics nor typesetting.

This principle implies several things. First, "normal" mathematical conventions about operator precedence, parentheses, and so on cannot be used, for to give special meaning to such characters means that the user has to understand what he or she is typing. Thus the language should not assume, for instance, that parentheses are always balanced — consider the half-open interval $(a,b]$ — nor that $\sqrt{a+b}$ can be replaced by $(a+b)^{1/2}$, nor that $1/(1-x)$ is better written as $\frac{1}{1-x}$ (or vice versa).

Second, there should be relatively few rules, keywords, special symbols and operators, and the like, so the language is easy to learn and to remember. Furthermore, there should be few exceptions to the rules that do exist — if something works in one situation, it should generally work everywhere. If a variable can have a subscript, then a subscript can have a subscript, and so on without limit.

Third, standard things should happen automatically, so that common or usual cases require no special treatment. Someone who types $x=y+z$ should get $x=y+z$. Subscripts and superscripts should automatically be printed in an appropriately smaller size, with no special intervention. Fraction bars have to be made the right length and positioned at the right height. And so on. Indeed a mechanism for overriding default actions has to exist, but its application is the exception, not the rule.

We will assume that the typist has a reasonable picture (a two dimensional representation) of the desired final form, as might be written by the author of a paper. We also assume that the input is typed on a computer terminal much like an ordinary typewriter, which implies an input alphabet of perhaps one hundred characters, none of them special. This is fortunate, for we can then resist the temptation to build a language where each special character has a special meaning.

A secondary, but still important, goal in our design was that the system should be easy to implement, since neither of the authors had any desire to make a long-term project of it. Further, since when we began we had a less than precise idea of where we were going, it was also necessary that the program be easy to change at any time.

To make the program easy to build and easy to change, and to guarantee regularity ("it should generally work everywhere"), the language is defined by a context-free grammar (described in a later section). The compiler for the language was built using a compiler-compiler.

A priori, the grammar/compiler-compiler approach seemed the right thing to do. Our subsequent experience leads us to believe that any other course would have been folly. The original language was designed in a few days, and construction of a working system sufficient that we could try things out required perhaps a person-month. Since then, we have spent another part-time three months or so tuning, adding facilities, and regularly changing things as users make criticisms and suggestions.

We also decided quite early that we would let TROFF do our work for us whenever possible, rather than reinventing the wheel. Since TROFF is quite a powerful program, with a macro facility, text and arithmetic variables, numerical computation and testing, and conditional branching, we have been able to avoid writing a lot of mundane but tricky software. For example, we store no text strings, but simply pass them on to TROFF. Thus we avoid having to write a storage management package. Furthermore, we have been able to isolate ourselves from the details of the particular device, character set, and so on currently in use. For example, we let TROFF compute the widths of all strings of characters — we need know nothing about them.

A third design goal is special to our environment. Since our program is only useful for typesetting mathematics, it is necessary that it interface cleanly with the underlying typesetting language for the benefit of those users who want to set intermingled mathematics and text (the usual case). The standard mode of operation is that when a document is typed, mathematical expressions are input as part of the text, but marked by (user settable) delimiters. The math-setter reads this input and treats as comments those things which are not mathematics, simply passing them through untouched. At the same time it converts the mathematical input into the necessary TROFF commands. The resulting output is passed directly to TROFF where the comments and the mathematical parts both become text and/or TROFF commands.

4. The Language

We will not try to describe the language precisely here; interested readers should look at [5] for more details. Throughout this section, we will write expressions exactly as they are handed to the math-setter, except that we won't show the delimiters that mark the beginning and end.

As we said, typing $x=y+z$ should produce $x=y+z$, and indeed it does. Variables are made italic, operators are Roman, and normal spacings between letters and operators are altered slightly to give a more pleasing appearance.

Spaces and newlines in the input are used by the math-setter to separate pieces of the input; they are not used to create space in the output. Thus

$$x = y + z$$

also gives $x=y+z$. This free-form input makes it easier to type and edit the input, for an expression may be typed as many short lines.

Extra white space can be forced into the output by several characters of various sizes — a tilde “~” gives a space equal to the normal word spacing in text; a circumflex “^” gives half this much.

Spaces (or tildes, etc.) are used to delimit pieces of the input. For example, to get something like

$$f(t) = 2\pi \int \sin(\omega t) dt$$

we write

$$f(t)=2\ pi\ int\ sin\ (\ omega\ t)^{\wedge}dt$$

Here spaces are *necessary* in the input to indicate that *sin*, *pi*, *int*, *omega* are special, and potentially worth special treatment. The math-setter looks each such token up in a table, and if appropriate gives it a translation. In this case, *pi* and *omega* become their Greek equivalents, *int* becomes the integral sign (which must be moved down and enlarged so it looks “right”), and *sin* is made Roman, following conventional mathematical practice. Parentheses, digits and operators are

$$x_i$$

And braces can occur within braces if necessary:

$$x \sup (i \sup (\alpha + \beta))$$

is

$$x^{i^{\alpha+\beta}}$$

To print braces, enclose them in double quotes, like "{". This is discussed in a later section.

8. Fractions

To draw a fraction, use the word *over*:

$$a+b \text{ over } 2c = 1$$

gives

$$\frac{a+b}{2c} = 1$$

The line is made the right length and positioned automatically. Braces can be used to make clear what goes over what:

$$\{\alpha + \beta\} \text{ over } \{\sin(x)\}$$

is

$$\frac{\alpha + \beta}{\sin(x)}$$

Over is considered by the equation-setter to be of lower precedence than *sub* and *sup*, so

$$-b \sup 2 \text{ over } \{\pi x\}$$

needs no braces to be unambiguously

$$\frac{-b^2}{\pi x}$$

The precedence rules, which decide which operation is done first, are summarized near the end of the user's guide. When in doubt, however, *use braces* to make clear what goes with what.

9. Square Roots

To draw a square root, use *sqrt*:

$$\text{sqrt } a + \text{sqrt}\{ax \sup 2 + bx + c\}$$

is

$$\sqrt{a} + \sqrt{ax^2 + bx + c}$$

Warning — square roots of tall quantities look lousy, because a root-sign big enough to cover the quantity is too dark and heavy:

$$\text{sqrt}(a \sup 2 \text{ over } b \text{ sub } 2)$$

is

$$\sqrt{\frac{a^2}{b_2}}$$

Good style replaces big square roots by something to the power $\frac{1}{2}$:

$$(a^2/b_2)^{1/2}$$

10. Summation, Integral, Etc.

Summations, integrals, and similar constructions are easy:

sum from $i=0$ to $\{i= \text{inf}\}$ x sup i

produces

$$\sum_{i=0}^{\infty} x^i$$

Notice that we used braces to indicate where the upper limit begins and ends. No braces were necessary for the lower limit, because it contained no blanks. The braces will never hurt, though — *always* use braces around the *from* and *to* limits, if they contain any blanks.

The *from* and *to* parts are both optional, but if both are used, they have to be in that order.

Other useful characters can replace the "sum":

int prod union inter

become, respectively,

$$\int \prod \cup \cap$$

Since the thing before the *from* can be anything, the from-to construction can often be used in unexpected ways:

lim from $\{n \rightarrow \text{inf}\}$ x sub $n = 0$

is

$$\lim_{n \rightarrow \infty} x_n = 0$$

11. Big Brackets, Etc.

To get big brackets [], braces { }, parentheses (), and bars | | around things, use the *left* and *right* commands:

left { a over b + 1 right } = left (c over d right) + left [e right]

is

$$\left\{ \frac{a}{b} + 1 \right\} = \left(\frac{c}{d} \right) + [e]$$

The resulting brackets are made big enough to cover whatever they enclose.

Several warnings about brackets are in order. First, notice that braces are typically bigger than brackets and parentheses, because they are made up of three, five, seven, etc. pieces, while brackets can be made up of two, three, etc. Second, left and right parentheses typically look poor, because the character set is poorly designed. Finally, the "right" part may be omitted: a "left something" need not have a corresponding "right something". If the *right* part is omitted, put braces around the thing you want the left bracket to encompass. Otherwise, the resulting brackets may be too large.

12. Piles

There is a general facility for making vertical piles of things; it comes in several flavors. For example:

A ~ left [pile { a above b above c } ~ pile { x above y above z } right]

will make

$$A = \begin{bmatrix} a & x \\ b & y \\ c & z \end{bmatrix}$$

The elements of the pile (there can be as many as you want) are centered one above another, at the right height for most purposes. The keyword *above* is used to separate the pieces; braces are used around the entire list.

Three other forms of pile exist: *lpile* makes a pile with the elements left-justified; *rpile* makes a right-justified pile; and *cpile* makes a centered pile, just like *pile*. The vertical spacing between the pieces is somewhat larger for l-, r- and cpiles than it is for ordinary piles.

```
sign(x) == left { lpile { 1 above 0 above -1 }
-- lpile { if x > 0 above if x = 0 above if x < 0 }
```

makes

$$\text{sign}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases}$$

Notice the left brace without a matching right one.

13. Size and Font Changes

By default, equations are set in 10 point type, with standard mathematical font conventions. Although the equation-setter makes a valiant attempt to use esthetically pleasing sizes and fonts, it is not perfect. To change sizes and fonts, use *size n* and *roman*, *italic*, and *bold*. Legal sizes which may follow *size* are 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, 36.

```
size 14 bold x = y + size 14 { alpha + beta }
```

gives

$$x = y + \alpha + \beta$$

As always, use braces to delimit what you want affected. For example, you can change an entire equation by

```
size 12 { ... }
```

14. Diacritical Marks

To get funny marks on top of letters:

```
x dot + X dot + y hat + y dotdot + x-y bar + {alpha + beta} bar + x tilde
```

gives

$$\dot{x} + \dot{X} + \hat{y} + \ddot{y} + \overline{x-y} + \overline{\alpha + \beta} + \tilde{x}$$

As well as possible, the mark is placed at the right height. The *bar* is made the right length for the entire construct; other marks are centered. (At present, this works only on italic letters; other fonts are botched.) By the way, there is no "prime" — use "'".

15. Quoted Text

Any input entirely within quotes ("...") is not subject to any of the font changes and spacing adjustments normally done by the equation setter. This provides a way to do your own spacing and adjusting if needed:

The ambiguous grammar approach seems to be quite useful — the grammar we use is small enough to be easily understood, for it contains none of the productions that would be normally used for resolving ambiguity. Instead the supplemental information about precedence and associativity (also small enough to be understood) provides the compiler-compiler with the information it needs to make a fast, deterministic parser for the specific language we want. When the language is supplemented by the disambiguating rules, it is in fact LR(1) and thus easy to parse.

The output code is generated as the input is scanned. In particular, any time a production of the grammar is recognized, (potentially) some TROFF is output. For example, when the lexical analyzer reports that it has found a TEXT (i.e., a string of contiguous characters), we have recognized the production

text : TEXT

The translation of this is simple — we generate a local name for the string, then hand the name and the string to TROFF, and let TROFF worry about the storage management. All we save is the name of the string, its height, and its baseline.

As another example, the translation associated with the production

box : box OVER box

is

Width of output box = slightly more than largest input width
 Height of output box = slightly more than sum of input heights
 String describing output box is
 move down; move right enough to center bottom box;
 draw bottom box (i.e., copy string for bottom box);
 move up; move left enough to center top box;
 draw top box (i.e., copy string for top box);
 move down and left; draw line full width;
 return to proper base line.

Most of the other productions have equally simple semantic actions — picturing the output as a set of properly-placed boxes makes the right sequence of positioning commands quite obvious. The only difficulty is in finding the right numbers to use for esthetically pleasing positioning.

With a grammar, it is usually clear how to extend the language. For instance, one of our users suggested a TENSOR operator, to make constructions like

$$\begin{matrix} & k \\ & / \\ \text{I} & \text{T} \\ m & \backslash \\ & \end{matrix}$$

Grammatically, this is easy; adding the production

box : TENSOR { list }

would be sufficient; semantically, we need only juggle the boxes to the right places.

6. Experience

There are really three aspects of interest — how well the math-setter sets mathematics, how well it satisfies its goal of being “easy to use”, and how easy it was to build.

The first area is easily disposed of. The material in this paper, of course, has all been set by the program. The reader can judge for himself whether it is good enough for the purpose he has in mind. One comment made by a user is that although the output is not as good as the best hand-set material, it is still better than average, and much better than the worst. In any case, who cares? Printed books cannot compete with the birds and flowers of illuminated manuscripts on esthetic grounds, either, but they have some clear economic advantages.

Some of the deficiencies in the output could be cleaned up with more work on our part. For example, we often leave too much space between a Roman letter and an italic one. If we were willing to keep track of the fonts involved, we could do this better more of the time.

8 TYPESETTING MATHEMATICS

Some other weaknesses are inherent in our output device — it is hard, for instance, to draw a line of an arbitrary length without getting a perceptible overstrike at one end.

As to ease of use, at the time of writing, the system has been used by two distinct groups. The main user population right now is about half a dozen members of staff in the Computing Science Research Center, who have collectively produced well over a hundred pages of mathematical text. Their typical reaction has been something like

- (1) It's easy to write, although I make the following mistakes...
- (2) How do I do...?
- (3) It botches the following things.... Why don't you fix them?
- (4) You really need the following features...

The learning time is short — a few minutes gives the general flavor, and typing a page or two of a paper generally uncovers most of the misconceptions about how it works. This group seems very satisfied with the language.

We have much less experience with the second user group, the secretaries and mathematical typists who were the original target of the system. For administrative reasons, most of them had had little chance to try it, so their response is limited to "it looks easy, much easier than what we have to do now." The one math typist who now uses it on a regular basis is an enthusiastic convert.

The language is somewhat prolix, but this doesn't seem excessive considering how much is being done, and it's certainly more compact than the corresponding TROFF commands. For example, here is the source for the continued fraction expression in section 1:

```
a sub 0 + b sub 1 over
{a sub 1 + b sub 2 over
{a sub 2 + b sub 3 over
{a sub 3 + ... }}}}
```

This is the input for the large integral of Section 1; notice the use of definitions.

```
define emx "{e sup mx}"
define mab "{m sqrt ab}"
define sa "{sqrt a}"
define sb "{sqrt b}"
int dx over {a emx - b e sup -mx} "=
left { lpile {
  1 over (2 mab) "log" {sa emx - sb} over {sa emx + sb}
  above
  1 over mab "tanh sup -1 ( sa over sb emx )
  above
  -1 over mab "coth sup -1 ( sa over sb emx )
}
----- (a>0, b>0)
```

As to ease of construction, we have already mentioned that there are really only a few person-months invested. Much of this time has gone into two things — fine-tuning (what is the most esthetically pleasing space to use between the numerator and denominator of a fraction?), and changing things found deficient by our users (shouldn't a tilde be a delimiter?).

The program consists of a number of small, essentially unconnected modules for code generation, a simple lexical analyzer, a canned parser which we did not have to write, and some miscellany associated with input files and the macro facility. There are only about 15 global variables. The program is currently about 1000 lines of C (14); a language reminiscent of BCPL. About 20 percent of these lines are "print" statements, generating the output code. To our everlasting shame, there are two GOTOs in the program.

The semantic routines that generate the actual TROFF commands can be changed to accommodate other output languages or devices. For example, in less than 24 hours, one of us changed the entire semantic package to drive NROFF, a precursor of TROFF, for typesetting mathematics on teletypewriter devices capable of reverse line motions. Since many potential users do not have access to the typesetter, but still have to type mathematics, this provides a way to get a typed version of the final output which is close enough for debugging purposes, and sometimes even for ultimate use.

7. Conclusions

We think we have shown that it is possible to do acceptably good typesetting of mathematics on a photocomposer, with an input language that is easy to use and that satisfies many users' demands, and that such a package can be implemented in short order, given a decent typesetting package underneath.

Defining a language, and building a compiler for it with a compiler-compiler seems like the only sensible way to do business. Our experience with the use of a grammar and a compiler-compiler has been uniformly favorable. If we had written everything into code directly, we would have been locked into whatever design we originally thought of. Furthermore, we would have never been sure where the exceptions and special cases were. But because we have a grammar, we can change our minds readily and still be reasonably sure that if a construction works in one place, it will work everywhere.

8. Acknowledgements

We are deeply indebted to J. F. Ossanna, the author of TROFF, for his willingness to modify TROFF to make our task easier, and for his continuous assistance during the development of our program. We are also grateful to A. V. Aho for help with language theory, to S. C. Johnson for aid with the compiler-compiler, and to our early users A. V. Aho, S. I. Feldman, S. C. Johnson, R. W. Hamming, and M. D. McIlroy for their constructive criticisms.

References

- [1] *A Manual of Style*, 12th Edition. University of Chicago Press, 1969. p 295.
- [2] *Model CIAIT Phototypesetter*. Graphic Systems, Inc. Lowell, Mass.
- [3] *D. M. Ritchie and K. L. Thompson*, The UNIX Time-Sharing System. CACM, July 1974.
- [4] *J. F. Ossanna*, TROFF User's Manual. Bell Laboratories internal memorandum.
- [5] *B. W. Kernighan, L. L. Cherry*, *Typesetting Mathematics — User's Guide*. Appendix to this paper.
- [6] *D. M. Ritchie*, *C Reference Manual*. Bell Laboratories internal memorandum.