S. van Egmond, F.C. Heeman, J.C. van Vliet

INFORM: an interactive syntax-directed formulae-editor

Computer Science/Department of Software Technology Report CS-R8702 January

69K82

INFORM: an Interactive Syntax-Directed Formulæ-Editor

S. van Egmond, F. C. Heeman, J. C. van Vliet
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

An article or book does not simply exist of a lot of characters, but it is structured. A book consists of chapters, which in return consist of subchapters, paragraphs, pictures, etc. The difference in approach between text-editing and document-editing is, that the former regards the text as a string of characters while the latter takes the structure of the text into account. As a subproject of the development of an interactive document-editor we have designed and implemented a prototype for an interactive grammar-driven editor for mathematical formulæ.

1986 CR Categories: I.7.2 [**Text Processing**]: Document Preparation - *format and notation*;

1986 Mathematics Subject Classification: 68B99 [**Software**]

Key Words & Phrases: Syntax-Directed Editing, Typesetting, Mathematical Formulæ, Interactive.

Note: this paper will be submitted for publication elsewhere.

1. Introduction

Most formulæ-processing systems (EQN [1], T$_E$X [2]) are batch systems. That is, the user gives a linear description of a formula, runs it through a program that gives a result on paper, and, seeing that it is not what he intended it to be, he corrects the description and runs the program again. This usually has to be repeated more than once. Take for instance the following description in EQN:

in from 0 to 1 xdx = 1 over 2

This yields as output:

$$\overset{1}{\underset{0}{in}} xdx = \frac{1}{2}$$

This may not be what the user really wants, so he changes the description to:

int from 0 to 1 xdx = 1 over 2

The output then becomes:

$$\overset{1}{\underset{0}{\int}} xdx = \frac{1}{2}$$

EQN has but a superficial knowledge of the structure of mathematical formulæ. The input 'int from 0 to 1 xdx' does not really say that the integral has two limits. It only states that '0' must be put below and '1' must be put above the preceding symbol, which happens to be the integral-sign in this

Report CS-R8702
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

case. If some other character is taken instead of the integral-sign, as shown in the example, EQN prints the '0' and '1' below and above that character respectively.

The batch approach has several disadvantages: the user has to learn how a formula is described (e.g. in the EQN or $\TeX$ language), the user has to know how to work with an editor to input this description, and the result on paper may be surprisingly different from the intention the user has.

As a contrast, an interactive system displays the formula on the screen in its two-dimensional form. The user then manipulates this two-dimensional formula instead of editing some linear representation.

Take for instance the following formula:

$$\frac{a+b}{c+d\times e}=f$$

To enter this formula in INFORM, the user gives the fraction command, and a horizontal bar appears on the screen with the cursor positioned above the bar, at the position of the nominator (see FIGURE 1, (1)). In the figure, the cursor on the screen is indicated by a '•'. Next the user enters '$a+b$' (2) and gives an END-command (3) to indicate the end of the nominator. Now the cursor appears centered below the bar, and the denominator can be entered (3). The user enters '$c+d\times e$', and after the END command (4), the last part is entered ('$=f$'). The two parts of the fraction always remain centered with respect to the horizontal bar. The bar itself will be as long as the maximum width of the nominator and denominator.

command	screen	
[FRAC]	$\frac{\bullet}{}$	(1)
a + b	$\frac{a+b\bullet}{}$	(2)
[END]	$\frac{a+b}{\bullet}$	(3)
c + d × e [END]	$\frac{a+b}{c+d\times e}\bullet$	(4)
= f	$\frac{a+b}{c+d\times e}=f\bullet$	(5)

FIGURE 1

When the user selects the whole formula (the selected part is displayed in reverse video), he may "zoom in" on the nominator and then go to the "next" subformula (denominator). Now it is possible for the user to "zoom in" on 'c'. A "next" movement will put the cursor on '$+$', and another "next" movement will put the cursor on '$d\times e$'. So the system has knowledge of the tree-like mathematical structure of the formula, including the precedence of '$\times$' over '$+$'.

In an interactive system the user sees the result immediately and manipulates the two-dimensional representation directly. The user does not need to know anything about lay-out rules (like variables being set in italic, the nominator and denominator being centered, etc.). The system will take care of all this.

This behaviour is similar to that of a syntax-directed editor for a programming language (see for instance Teitelbaum [3]). In such an editor, the user enters the command for, say, a while-statement and a template is then shown on the screen for the while-construct of that language. The template has two "holes" (placeholders) for the condition and the body, which the user can fill in subsequently. The equivalent of automatically generating the proper lay-out is called pretty printing in syntax-directed editors.

Interactive formulæ-editors using this approach have already been designed and implemented. An early example is Levison ([4]). In his system, formulæ are printed on an alphanumeric display. This system is not grammar-driven. Also, certain commands are not entered in reading order. For example, the superscript command must be entered before the ground-expression. When the user first enters the ground-expression, it is not easy to add the superscript: the already typed-in ground-expression must be clipped and then pasted into the template of the superscript.

Quint describes a system called Edimath [5]; it uses a bit-map display. In Edimath the formulæ are not always well structured in the mathematical sense. A superscript for example is not linked to the corresponding ground-expression. Edimath as well as Levison's system do not structure expressions, but rather treat them as text. This is further discussed in section 2.3.

Originally, Edimath has not been grammar-driven. However, this has changed and Edimath is now part of a grammar-driven document editor called Grif [6].

1.1. Design Goals

Our most important design goal in constructing the formulæ-editor has been to use a grammar to describe the structure and lay-out of formulæ, instead of directly incorporating this knowledge into the program. The reasons for this are threefold:

- The system may then be used for other classes of objects. Instead of a grammar for mathematical formulæ, one may also use "grammars" for chemical formulæ, documents, etc.
- Different parts of the system need the same information. For example, information about the structure of formulæ is needed for editing as well as for printing. It seems sensible to locate this information in one place.
- Using a grammar, it is relatively easy to add or change information about formulæ: only the grammar needs to be changed, not the program.

A second design goal has been to make a system that is easy to use for both mathematicians and non-mathematicians. This means that the commands must be easy to understand and use. To that end, a formula can best be entered in reading order (from left to right). So, operators like '$+$' are entered in infix-order, while constructs like fraction are entered in prefix-order.

A third design goal is, that the formulæ are automatically typeset according to commonly used typographical rules. The user is not required to specify any lay-out information. Also, the picture on the screen should look like the version printed on paper.

1.2. Overview

The major obstacle in designing our formulæ-editor has been to find a formalism with which infix operators can be entered in infix-order. Most syntax-directed editors for programming languages have two 'modes'. In 'template-mode' prefix-type constructs, like if- or while-statements, are entered by first selecting the construct, and subsequently filling in the holes of the template. For infix-type constructs such as arithmetic expressions, a 'text-mode' is often used, whereby the user enters the whole construct on a character-by-character basis. After the construct has been entered, it is parsed and the parse tree is built up.

Building up the parse tree for infix-type constructs in an incremental way is not easy, since the parse tree may have to be changed rather drastically when new symbols are input. For instance, after inputting '$a+b$' there is a node '$+$' with two siblings, 'a' and 'b'. A subsequent '$\times$' then causes the second sibling of '$+$' to become '$\times$', while 'b' becomes the first sibling of '$\times$'. The problems of finding a proper grammatical notation and an incremental parsing algorithm for the description of such constructs is elaborated upon in the next section. Section 3 describes the current status of the system. Some conclusions and directions for further work are discussed in section 4.

2. The grammar

In batch systems the input is often regarded as a sentence according to some grammar, and some well-known parsing algorithm can be used to build an internal representation (e.g. a parse tree).

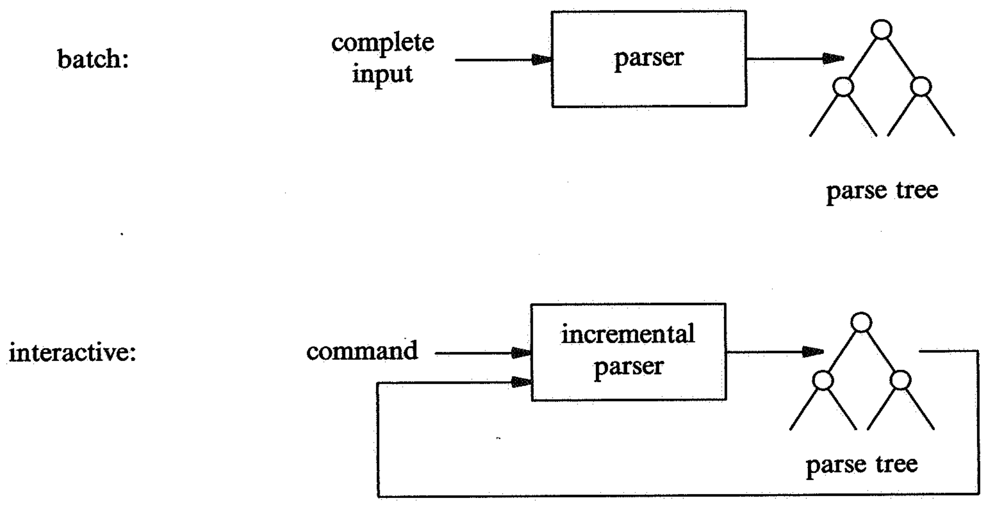


Figure 2

In interactive syntax-directed systems, an input-command is used to choose some grammar-rule, which is then used to augment the internal representation (Figure 2). The commands entered by the user follow the grammar used by the system. For instance, the name on the lefthand side of a production-rule can be used as the name of the command to invoke this particular rule. If we apply this method to our interactive formulæ editor, we need a grammar from which commands can be derived that correspond to the user's model in a natural way.

Further, we would like the structure that is imposed on the formula to agree with the idea the user has of a formula, because then a single user-action (like a cursor-movement) corresponds to a single action in the internal representation. But since the internal representation is derived from the grammar, this means that the grammar must be able to express (our model of) the user's idea of a formula.

2.1. BNF-grammar

The grammar for a programming language is usually written in BNF-format, so we also tried to write a grammar for formulæ in BNF. It is, however, not easy to derive appropriate commands: even with a simple grammar 'unnatural' commands arise. Figure 3 contains a simple grammar in which only addition is used. In the second part of the figure, the user-commands for the sample expression '$a+b$' are given. These commands result from the direct use of this grammar for an incremental parsing algorithm. For each command, the resulting parse tree and screen representation is given. In all following figures '*' indicates the position of the cursor in the internal representation and the box (□) indicates the cursor on the screen.

The commands like [id] and [term] in Figure 3 are pointless to a user, but have to be given if the system is directly derived from the grammar. To express a structure for expressions in BNF, it is necessary to make one production-rule for each precedence-level. It is also necessary to make a production-rule left recursive to achieve left associativity. In this way extra levels are introduced in the parse tree. The user can be made unaware of these extra levels but it makes the system far more complicated.

Figure 3 shows that the user has to know quite a lot about the grammar. When the cursor indicates the whole formula, and the user "zooms in" on the 'a' (a single step in the user's point of view),

expr	= expr + term | term
term	= id
id	= a | b | c | ... | z

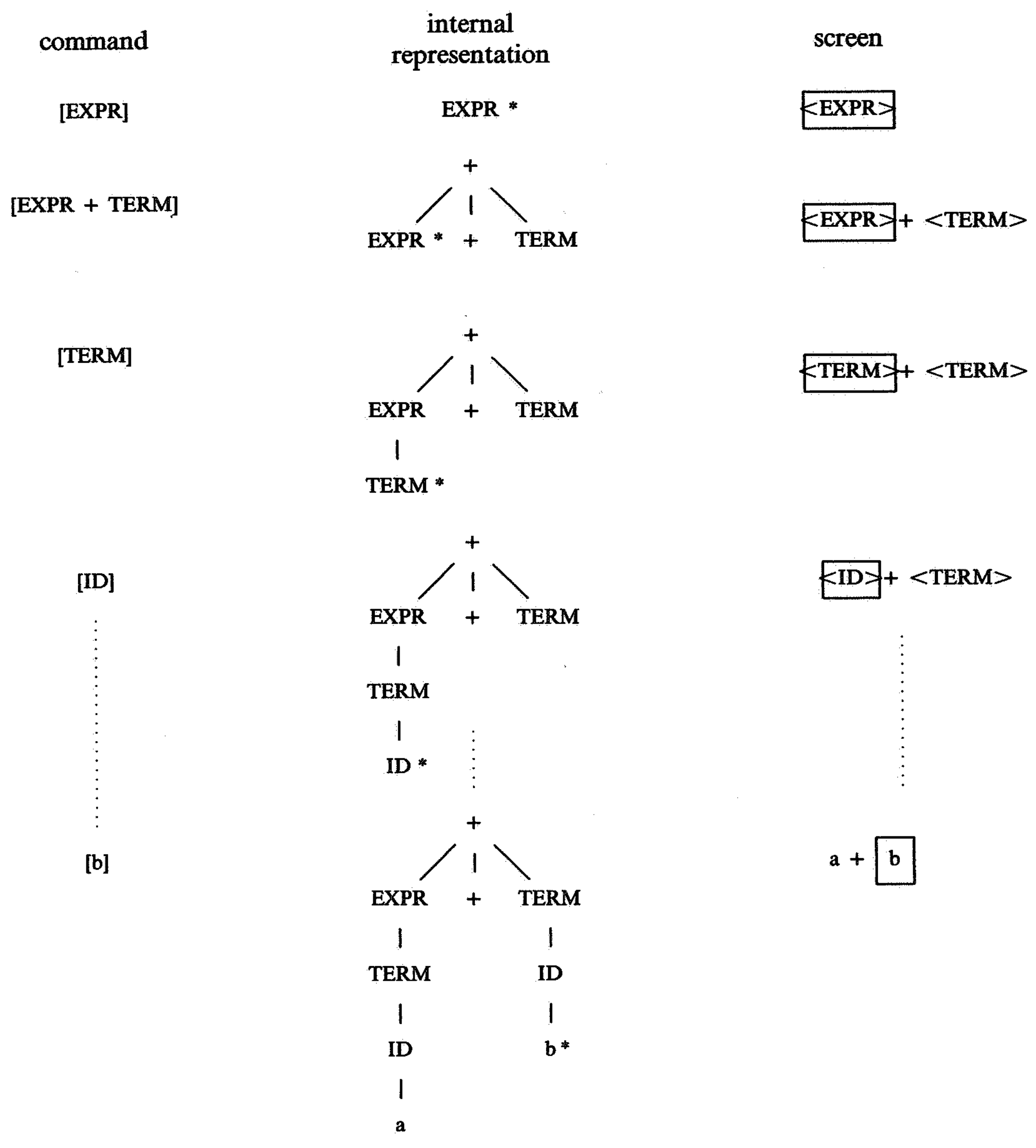


FIGURE 3

four internal "moves" are necessary to pass the various redundant nodes in the parse tree. However, to descend to '*b*' three internal "moves" are required. There is no simple correspondence between the user's model of the structure of a formula and the corresponding internal representation.

The problems of 'unnatural' commands arise because BNF is a concrete syntax, which gives a concrete syntax tree (CST), rather than an abstract syntax tree (AST). An AST displays the logical structure, rather than the concrete structure. In an AST the operators are internal nodes with the operands as children of the node while in a CST, the operator is also a leaf. Therefore, we looked for a grammar from which it is not too difficult to derive an AST, and from which a friendly user-interface can be generated directly.

2.2. Abstract syntax

Syntax-directed editors, such as MENTOR [7] and ALOE [8] use an abstract syntax. FIGURE 4 shows an example of the abstract syntax of ALOE (MENTOR is similar). The startsymbol is indicated by 'root'. All non-terminals (i.e. operators) have a list of classes (i.e. operands) associated with them. The non-terminal 'plus' has two classes, EXP **and** EXP. These classes uniquely determine the number and kind of operands of this non-terminal: this is called the AND-structure (see Tichy [9]).

All classes have a list of (non-)terminals: one of the members of that list can be substituted for that class. In the grammar of FIGURE 4, the class EXP can be replaced by the non-terminal 'plus' **or** by the terminal 'id'. Since only one member of the list can be chosen to replace a particular instance of the class, this is called the OR-structure. So a class has the function of a non-terminal in BNF, but a class is **replaced** (expanded) by a member of its set of (non-)terminals. FIGURE 4 also shows an example of a session and the internal representation that is built when this grammar is used to derive a syntax-directed editor. The language produced by this grammar is the same as the language produced by the grammar of FIGURE 3.

Compared with the BNF-grammar, the commands are more natural. There are no extra commands for intermediate levels. The internal representation is simple to build and use, because there are no redundant nodes. There now is a 1-1 correspondence between steps in the internal representation and moves on the screen.

However, the user still has to enter an expression in prefix-order, i.e. the operator is entered before the operands. When the operator is entered, the number of operands is known, so the corresponding number of holes (i.e. unexpanded classes) can be generated and subsequently filled in by the user. In infix notation, the user first enters an operand, and then the operator, but since the hole for the first operand is only available after the operator is entered, it is impossible to enter the expression in infix-order when this grammar is used.

Prefix notation is difficult to use, especially for a non-mathematician (for instance, '$a+b\times c\times d+e$' is entered as '$++a\times\times bcde$'). A possible solution is to enter and structure expressions as text, as in the Cornell Program Synthesizer [3] and Edimath [5]. In this way, however, the correct lay-out of a formula can not be derived from its structure. For example, it is custom practice to print variables in italic and names of functions like 'sin' in roman. Moreover, the structure of the formula might be different from what the user expects according to mathematical rules (e.g. in Edimath 'xyz^2' is structured as '$(xyz)^2$' and not as '$xy(z^2)$').

2.3. The Grammar of INFORM

Our goal then is to be able to input expressions "from left to right", **and** build an AST that represents the mathematical structure of the formula.

In BNF, operator precedence and associativity can be expressed, but this has some disadvantages as shown in section 2.1.

In ALOE, precedence-values are specified as attributes of (non-)terminals in the grammar. However, these values are not used in building the AST: the AST is built up by successively replacing some class by a specific (non-)terminal. The precedence-values are only used to provide automatic parenthesization while unparsing (prettyprinting) the AST into infix notation (the same holds for associativity values). For example, to input '$a\times b+c$' the user types '$+\times abc$'; to input '$a\times(b+c)$' the user types '$\times a+bc$'.

The grammar-mechanism in INFORM includes also precedence and associativity information for each (non-)terminal. But rather than using classes which are replaced by some specific (non-)terminal, we use **items**. Items are replaced too, but this replacement is dynamic and depends on the precedence- and associativity-values involved. We have defined two different items, EXPR and

root	: formula;
terminals	: id = a b c ... z;
non-terminals	: formula = EXP;
	plus = EXP EXP;
classes	: EXP = id plus;

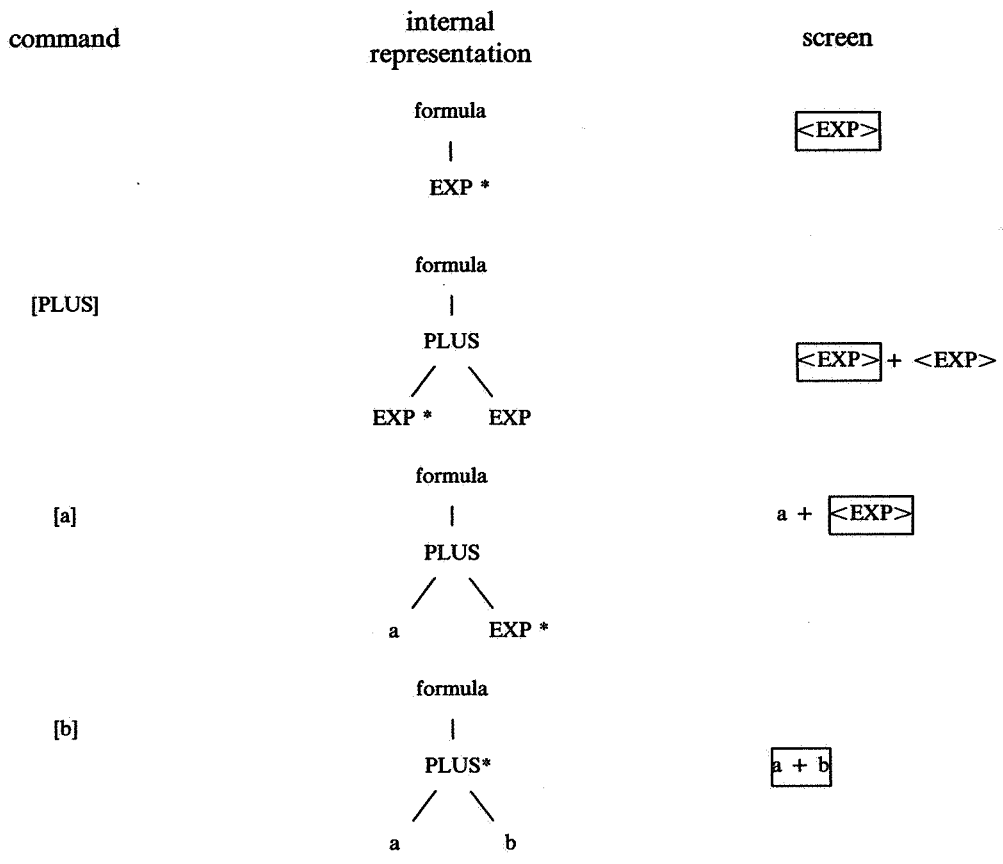


FIGURE 4

START, which are explained below.

An EXPR-item represents an operand. What the operand exactly is, is determined by the precedence-values of the operator and operand. The precedence-value of an operator is smaller than or equal to the precedence-value of its operands. As an example, take the following grammar in INFORM:

terminals	:		
		a, b, c	: prec = ∞
non-terminals	:		
		=	: EXPR EXPR, prec = 1
		+	: EXPR EXPR, prec = 2
		$\times$	: EXPR EXPR, prec = 3

The user types '$a + b$' (see FIGURE 5.1). The first item of '+' is made to be 'a' and the second item is made to be 'b'. Next, the user continues with '×' (FIGURE 5.2): now the second item of '+' is made to be '×' (since '×' has precedence over '+'), and '*b*' made to be the first item of '×'. On the other hand, if the user had continued with '=' (FIGURE 5.3) instead of '×', the '+' would have been made to be the first item of '=', while the operands of '+' would not have changed.

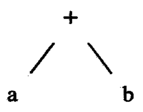


FIGURE 5.1

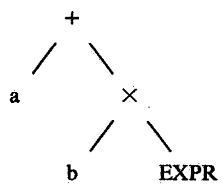


FIGURE 5.2

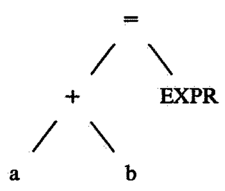


FIGURE 5.3

The second type of item is called START-item. This item represents a separate subformula. In the following example '×' now has a START-item as its second item:

terminals	:	
	a, b, c	: prec = ∞
non-terminals	:	
	=	: EXPR EXPR, prec = 1
	+	: EXPR EXPR, prec = 2
	×	: EXPR START, prec = 3

Suppose the user types '$a \times b$' (FIGURE 6.1). Now, if the user types '+' the AST is as in FIGURE 6.2.

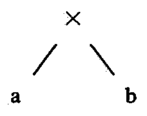


FIGURE 6.1

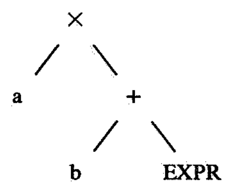


FIGURE 6.2

Although the '×' has precedence over '+', the '+' becomes an operand of '×', because the second operand of '×' is specified as a START-item. A START-item acts as if it has infinite low precedence: it is a threshold when restructuring the parse tree.

With these two items we can describe the mathematical structure of formulæ in the grammar. In our grammar a fraction has two START-items: both the nominator and denominator are separate subformulæ. Addition has two EXPR-items separated by the '+'-sign: the contents of both operands depends on the precedence of all previous and coming input. Superscript has one EXPR-item and one START-item: the EXPR-item represents the ground-expression, its contents depends on all previous input, and the START-item represents the exponent. In this way, the superscript command can be entered after the ground-expression has already been input, since the EXPR-item "automatically" matches with the correct ground-expression.

2.4. Incorporating Lay-out Information in the Grammar

So far, we only talked about building the internal representation by using the precedence of each (non-)terminal. To display the formula on the screen, we need a mapping from the internal representation to a two-dimensional representation. Since we decided to make the system completely grammar-driven, this information is specified in the grammar too.

To determine the lay-out of formulæ, we use the idea of boxes [2]. Each box has a reference point. This point, at the left side of the box, is placed on the so-called baseline. The position of the reference point determines the position of the box (and thus its contents). By placing the boxes next to one another with their reference points on the baseline, their contents are placed next to one another in the proper way (see FIGURE 7).

Furthermore, a box has three attributes: height (the amount of space occupied above the baseline), depth (the amount of space occupied below the baseline) and width. In FIGURE 7, the box with contents 'h' has zero depth, and the box with contents '—' has zero height.

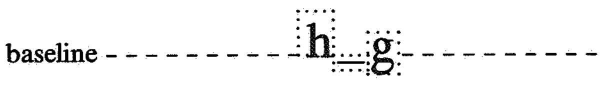


FIGURE 7

The contents of a box is irrelevant. In FIGURE 7, the boxes contain characters, but a box might as well contain a complete word or formulæ, and is then composed of other boxes.

To use boxes in INFORM the grammar is extended with attributes. We have three attributes for the values of height (h), depth (d) and width (w), respectively. Those attributes are specified for every (non-)terminal and indicate the size of the enclosing box containing the formula represented by the (non-)terminal and its descendants. The size of the enclosing box is calculated using the sizes of the subformulæ.

Two other attributes, for the x-coordinate and y-coordinate of the reference point, are used for the position of the formula on the screen. The position of subformulæ (siblings) is calculated with respect to the cartesian coordinates (0,0) of the enclosing box (parent).

It is rather cumbersome to specify the (x, y) position for each subformula: for instance, centering two objects with respect to each other is a frequently occuring operation. We therefore use a few convenient procedures for specifying positions, like center_above(a, b) (which means: center a above b).

FIGURE 8 shows an example of a part of our grammar together with the corresponding AST and an illustration of the way boxes are used. For legibility, we have inserted some extra white space between adjacent boxes. Appendix I contains a larger example with several (non-)terminals. The grammar in both examples is part of the grammar as used by INFORM. In the grammar:

$0 refers to the lefthand side of a production-rule;
$i, $i \geqslant 1$, refers to an item;
@i, $i \geqslant 1$, refers to a symbol (optional).

A symbol is something that is part of the concrete syntax but not of the abstract syntax (e.g. a fraction bar). The user can not select such a symbol, so it can not be changed or deleted on its own.

The precedence and associativity are not specified in this example because for FRACTION only one value is permitted, which is the default value (precedence = ∞; associativity = NONE). A '%' starts a comment, that extends to the end of the line.

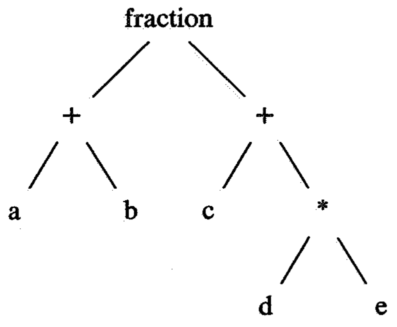
In the grammar-part of FIGURE 8, the symbol '@1' represents the horizontal bar (HORBAR). Usually, the fraction bar is slightly longer than the wider of nominator and denominator, so in calculating the width of the bar a constant (2) is added. The position of the bar is determined using the procedure 'print'. The line 'print(@1, 0, 2 * @1.d)' means: put the bar (@1) on x-coordinate '0' and on y-coordinate '2 * @1.d' (this way the bar is printed a bit above the baseline, which looks nicer). The nominator ($1) is centered above the bar (@1) by 'center_above($1, @1)', etc. The size of the fraction ($0) is calculated using the sizes of the nominator ($1), denominator ($2) and the fraction-bar (@1).

```
FRACTION:   % structure information:
            $0.e = START START;                 % two START-items

            % symbol information:
            @1.f = "S";                         % font of symbol
            @1.i = HORBAR;                      % symbol is horizontal bar
            @1.w = max($1.w, $2.w)+2;           % width of symbol

            % lay-out information:
            print(@1, 0, 2 * @1.d);             % position of symbol
            center_above($1, @1);               % and subformulæ
            center_under($2, @1);

            % size of this formula:
            $0.h = @1.h+2*@1.d+$1.h+$1.d;       % height
            $0.d = $2.d+$2.h-@1.d;              % depth
            $0.w = @1.w                         % width
#           % end
```



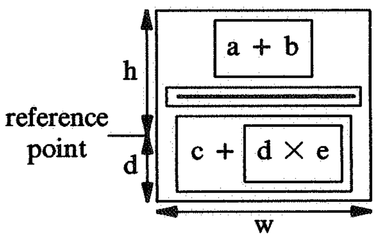


FIGURE 8

The lay-out of a complete formula is determined by:
- first calculating the sizes of the subformulæ recursively, and then using these sizes to calculate the size of the whole formula. Sizes are synthezised attributes: these are determined bottom-up.
- determining the position of the subformulæ recursively with respect to the reference-point of the whole formula, using the sizes of the subformulæ. The position is an inherited attribute: these are determined top-down.

3. CURRENT STATE

We have implemented our system according to the design goals stated earlier. The current system works on a Sun-3/75 workstation. The system is implemented in C++ [10] and uses the SunView window package.

When the program is started, a window appears on the screen. In this window the formula will be displayed. It is possible to read a file that contains EQN-code: the formula it represents is then displayed on the screen. However, EQN-input may not contain EQN lay-out commands (like 'mark', 'lineup', 'up', 'gsize', etc). Further, since our system has knowledge about structure of formulæ, the EQN-input must represent a well-structured formula. It is also possible to generate EQN-code so that a hard-copy result can be obtained by using the EQN-system.

The formula is entered by using a mouse and the keyboard. Pressing the middle and right button of the mouse will display a menu. In the menu, commands like 'integral', 'root', 'fraction', etc. are displayed and can be selected. After selection (releasing the button) the chosen construct appears on the screen at the current cursor-position. For constructs like brackets, greek characters, etc. additional information must be entered (the specific bracket or greek character that is wanted): in this case a second-level menu is displayed with the available options. The contents of all the menu's to be displayed is derived from the grammar. It is also possible to enter input from the keyboard. All the characters (a-z, A-Z), digits (0-9) and operators ($+$, $-$, $\times$, /) are entered this way.

Before the formula can be drawn, the size and position of every symbol must be known. To calculate all sizes each time makes the system too slow, so it has been optimized in order to recalculate changed sizes only. Printing on the screen, however, must be done each time. Because the most time-consuming operation in displaying the formula is the opening of the necessary font tables, all symbols whose size is not changed are copied from the screen-memory to a background-memory on their (possibly changed) positions while the changed symbols are drawn in the background-memory using information from the font tables. The copy-operation is a bit-operation, which takes far less time than drawing the characters anew. Then the background-memory is copied to the screen-memory (also a bit-operation), to update the formula on the screen. This goes sufficiently fast, so that no unpleasant flashing of the screen occurs.

All this makes the system work fast enough to be really interactive. Only with a large matrix (say $> 10 \times 10$) one has to wait to see the change (albeit less than a second), because all columns and rows must be recalculated.

The system presently contains only a few elementary edit-commands and a few cursor-commands. The edit-commands are *delete* (remove the formula the cursor indicates), *insert* (add before the cursor) and *append* (add after the cursor). The cursor-movements are:

- *widen:* go to the surrounding formula
- *narrow:* go to the first subformula
- *right:* go to the next formula
- *left:* go to the previous formula
- *extend right:* the cursor is extended so that the next formula is indicated too
- *extend left:* analogue to *extend right*

It is also possible to click with the left mouse-button somewhere in the formula. The formula in the smallest box in which the mouse-click took place, now becomes indicated by the cursor.

The following table gives an indication of the size in lines of C++-code (including comments) of several parts of our system:

reading input	800
generating output	550
determining lay-out	1000
building the internal representation	2300
generating code from the grammar	2000
Total	6650
grammar	370

The source code is 140K, the object code (including the window-system) is 512K.

4. Conclusions

We have made a working prototype which is, however, not yet complete. We plan to extend the formula-editor so that it can handle several lay-out methods per formula. For example, the limits of an integral might be typeset differently when this formula is between two words on a line (called an inline formula) from when it is between two blocks of text (called a displayed formula). Further, we may add the possibility of giving lay-out information explicitly, like: put this 'x' in bold, with point size 18.

In the future the user-interface will probably change too: instead of the name of the command, an icon can be shown. Also the number of editing-functions will increase. We have not spent much time on the user-interface and edit-commands yet, since this system will become part of an integrated document-editor. The purpose is to make the edit-commands and user-interface of the document-editor uniform; there should not be a different command for inserting text and inserting a formula.

The prototype fulfills the third design goal (to produce a typographically well-formed formula). Variables are printed in italic, names of standard functions in roman, super/subscripts in a smaller point size, etc.

We can not judge very well whether we have succeeded with the second design goal (to make the system easy to use), since the current user-interface is a rudimentary one. But we think that the way formulæ are entered and structured is a good approach. When the user wants to enter some mathematical construct, the system is able to guide the user since it has knowledge of the objects the user enters. Expressions are entered in infix-order and an incremental parser builds the AST. In [11] an incremental parsing algorithm for expressions entered in infix-order is described. This algorithm uses about twelve distinct tree-transformation functions to update the internal representation, while we use only two. Also, parsing parentheses is built into their algorithm, and is not derived from the specific grammar used to build the system. In that sense their system is not completely grammar-driven.

Our first design goal (to make the system completely grammar-driven) is fully met. Code is generated directly from the grammar. This code is compiled with the rest of the program and yields an interactive formulæ-editor for a specific grammar. Code is generated for:

- determining sizes and positions
- part of the user-interface
- filling a table with information about precedence-values, associativity-values, etc.

The external grammar describes the structure and lay-out of various mathematical constructs, like addition, fraction, super/subscript, integral, piles of formulæ, matrices, etc. The system uses the lay-out and structure description in the external grammar to determine the lay-out and structure of a formula. So changing the lay-out description in the grammar results in a formula typeset in a different way, and by adding new constructs to the grammar the system can be extended.

References

1. B. W. Kernighan and L. L. Cherry (March 1975). A System for Typesetting Mathematics, *Communications of the ACM*, 18.3, 151-157.
2. D. E. Knuth (1979). *TEX and METAFONT: New Directions in Typesetting*, Digital Press.
3. T. Teitelbaum and T. Reps (September 1981). The Cornell Program Synthesizer: a syntax-directed programming environment, *Communications of the ACM*, 24.9, 563-573.
4. M. Levison (February 1983). Editing Mathematical Formulae, *Software - Practice and Experience*, 13.2, 189-195.
5. V. Quint (March 1983). An Interactive System for Mathematical Text Processing, *Technology and Science of Informatics*, 2.3, 169-179.
6. V. Quint and I. Vatton (April 1986). Grif: An Interactive System for Structured Document Manipulation, in *Proceeding of the Conference on Text Processing and Document Manipulation*, ed. J.C. van Vliet, Nottingham, England.
7. B. Lang (1985). Mentor - Design and Implementation of the Kernel of a Program Manipulation

System, in *Integrated Project Support Environments*, 175-188, ed. J. McDermid, Peregrinus Ltd., London.

8. R. Medina-Mora (March 1982). *Syntax-Directed Editing: Towards Integrated Programming Environments,* Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh.
9. W. F. Tichy (1982). A Data Model for Programming Support Environments and its Application, in *Automated Tools for Information Systems Design*, 31-48, ed. A.I. Wasserman, North-Holland Publishing Company.
10. B. Stroustrup (March 1986). *The C++ programming language*, Addison-Wesley Publishing Company, Murray Hill, New Jersey.
11. G. E. Kaiser and E. Kant (May 1985). Incremental Parsing without a Parser, *The Journal of Systems and Software*, 5.2, 121-144.

Appendix I

Grammar of INFORM

TABLE I.1 shows the possible attributes for the lefthand side ($0) of a production-rule. TABLE I.2 shows the possible attributes for the items ($i, i $\geqslant$ 1). TABLE I.3 shows the possible attributes for a symbol (@i, i $\geqslant$ 1).

attribute name	explanation	default values	possible values
h	height of enclosing box	-	$\mathbb{N} \cup \{0\}$
d	depth of enclosing box	-	$\mathbb{N} \cup \{0\}$
w	width of enclosing box	-	$\mathbb{N} \cup \{0\}$
t	refers to type of lefthand side	-	grammar dependent
a	associativity	-	LEVEL | LEFT | RIGHT | NONE
p	priority	∞	$\mathbb{N} \cup \{0\} \cup \{\infty\}$
e	items of a production rule, see section 2.3	-	(START | EXPR | $0.t)+

TABLE I.1: attributes for $0

attribute name	explanation	default values	possible values
h	height of item	-	$\mathbb{N} \cup \{0\}$
d	depth of item	-	$\mathbb{N} \cup \{0\}$
w	width of item	-	$\mathbb{N} \cup \{0\}$
t	type of contents of item	-	grammar dependent
s	multiplicative factor to change point size	1.0	$\mathbb{R}^+ \cup \{0\}$
f	font in which contents of item must be drawn	italic	roman, special, italic

TABLE I.2: attributes for $i, i $\geqslant$ 1

attribute name	explanation	default values	possible values
h	height of a symbol	size from font table	$\mathbb{N} \cup \{0\}$
d	depth of a symbol	using 'f' and 'i'	$\mathbb{N} \cup \{0\}$
w	width of a symbol	and the pointsize	$\mathbb{N} \cup \{0\}$
s	multiplicative factor to change point size	1.0	$\mathbb{R}^+ \cup \{0\}$
f	font in which symbol must be drawn	italic	roman, special, italic
i	index in font table	-	character codes

TABLE I.3: attributes for @i, i $\geqslant$ 1

The directory which contains the fonts used and the initial point size are specified first (in this order) in the grammar.

```
fontdir = "/usr/lib/vfont"
pointsize = 10
```

Next, all the terminals with their symbols must be specified. This list is preceded by the word TERMINAL in capital letters. Each grammar rule for a terminal is terminated by '#'. A terminal is allowed to have one symbol only. The index can be a list though (as in FSTANDARD). In this case, the program generates a submenu with the possible options which is displayed after selection of the FSTANDARD-option in the main-menu (see FIGURE I.4).

```
TERMINAL
\FSTANDARD :
        @1.f = "R";
        @1.i = { "sin", "cos", "tan", "exp", "Re", "Im", "ln"};
        #
```

The non-terminals are specified after the terminals. This list is preceded by the word NTERMINAL. After each non-terminal, a list of values for attributes, symbols, and lay-out information belonging to that lefthand side is specified, terminated by '#'. The lefthand side of may consist of several non-terminals. The grammar rule applies to each of these. In the example, the strings '\063', '\011' and '\012' represent the character-codes for '≠', '⩽' and '⩾' respectively.

```
NTERMINAL
\063, \011, \012 :
        $0.p = 0;
        $0.a = LEVEL;
        $0.e = EXPR $0.t EXPR;
        % width of enclosing box is sum of width of all items
        $0.w = sum(w);
        % height of enclosing box is maximum of height of all items
        $0.h = max_all(h);
        $0.d = max_all(d);
        % font of second item is 'special font'
        $2.f = "S";
        % print 1st item at position (0, 0)
        print($1, 0, 0);
        % print 2nd item next to 1st item and 3rd item next to 2nd item
        print_next($2, $1);
        print_next($3, $2);
        #
```

```
\+, \-:
        $0.p = 1;
        $0.a = LEVEL;
        $0.e = EXPR $0.t EXPR;
        $0.w = sum(w);
        $0.h = max_all(h);
        $0.d = max_all(d);
        % font of second item is 'roman font'
        $2.f = "R";
        print($1, 0, 0);
        print_next($2, $1);
        print_next($3, $2);
        #

\SUP :
        $0.p = 9;
        $0.e = EXPR START;
        % width of enclosing box becomes width of 1st item
        % when contents of 1st item is not a subscript,
        % otherwise it becomes the width of the groundexpression
        % of the subscript
        $0.w = if_then($1.t = \SUB, ground(\SUB, w), $1.w) + $2.w;
        $0.h = sum(h);
        $0.d = $1.d;
        $2.s = 4.0/5.0;
        print($1, 0, 0);
        print($2, if_then($1.t = \SUB, ground(\SUB, w), $1.w), $1.h);
        #

\BRAC :
        $0.e = START;
        @1.f = "R";
        @1.i = { "(", "[", "{", "|" };
        @2.f = "R";
        @2.i = { ")", "]", "}", "|", EMPTY };
        $0.h = max_all(h);
        $0.d = max_all(d);
        $0.w = sum(w);
        print(@1, 0, 0);
        print_next($1, @1);
        print_next(@2, $1);
        #
```

```
\PILE :
        $0.e = < START >;
        $0.w = max_all(w);
        $0.h = sum(h)*2.0/3.0 + sum(d)*2.0/3.0;
        $0.d = $0.h/2.0;
        % move the reference point to position (0, $0.h)
        move(0, $0.h);
        % repeat for all items ($i)
        all(i, 1,
            move(0, -$i.h);
            print($i, 0, 0);
            move(0, -$i.d);
        );
        #
```

The non-terminal 'BRAC' specifies lists for its symbols. When the option for bracket is selected in the main-menu, a submenu appears with the possible values for the opening bracket (see Figure I.4). After selecting one of the brackets a second submenu is displayed with the possible values for the closing bracket. For example, the pair '(]' is possible. The lists of choices are determined by the list of characters for '@1' and '@2', respectively.

In the construct 'PILE' there is a list (indicated by '<' and '>') of START-items. When the user has selected this construct the system asks how many items are needed. It is possible to have nested list of items.

Figure I.4 shows the main-menu and some generated submenu's. In the main-menu, the option for GREEK and the bottom half of the menu (∞ up to REDISPLAY) are provided by default. Characters are also part of INFORM by default, and thus are not specified in the grammar.

PILE	BRAC
SUP	$\neq$
$\leqslant$	$\geqslant$
FSTANDARD	GREEK
∞	EQN_INPUT
EQN_OUPUT	NEWFORMULA
STOP	SHOWHOLES
REDISPLAY	

main-menu

sin cos tan exp Re Im ln

submenu for FSTANDARD

([{ |

)] } | EMPTY

submenus for BRAC

Figure I.4