HAL open science

Towards semantic mathematical editing

Joris van der Hoeven

To cite this version:

Joris van der Hoeven. Towards semantic mathematical editing. 2011. hal-00569351v2

HAL Id: hal-00569351

https://hal.science/hal-00569351v2

Preprint submitted on 9 Sep 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Towards semantic mathematical editing*

Joris van der Hoeven

LIX, CNRS
École polytechnique
91128 Palaiseau Cedex
France

Email: vdhoeven@lix.polytechnique.fr
Web: http://lix.polytechnique.fr/~vdhoeven

September 9, 2011

Currently, there exists a big gap between formal computer-understandable mathematics and informal mathematics, as written by humans. When looking more closely, there are two important subproblems: making documents written by humans at least syntactically understandable for computers, and the formal verification of the actual mathematics in the documents. In this paper, we will focus on the first problem.

For the time being, most authors use TEX, LATEX, or one of its graphical front-ends in order to write documents with many mathematical formulas. In the past decade, we have developed an alternative wysiwyg system GNU TEX_{MACS}, which is not based on TEX. All these systems are only adequate for visual typesetting and do not carry much semantics. Stated in the MathML jargon, they concentrate on presentation markup, not content markup.

In recent versions of TEX_{MACS}, we have started to integrate facilities for the semantic editing of formulas. In this paper, we will describe these facilities and expand on the underlying motivation and design choices.

To go short, we continue to allow the user to enter formulas in a visually oriented way. In the background, we continuously run a packrat parser, which attempts to convert (potentially incomplete) formulas into content markup. As long as all formulas remain sufficiently correct, the editor can then both operate on a visual or semantic level, independently of the low-level representation being used.

An important related topic, which will also be discussed at length, is the automatic correction of syntax errors in existing mathematical documents. In particular, the syntax corrector that we have implemented enables us to upgrade existing documents and test our parsing grammar on various books and papers from different sources. We will provide a detailed analysis of these experiments.

Keywords: Mathematical editing, content markup, packrat parsing, syntax checker, syntax corrector, TEX_{MACS}

A.M.S. subject classification: 68U15, 68U35, 68N99

1. Introduction

One major challenge for the design of mathematical text editors is the possibility to give mathematical formulas more semantics. There are many potential applications of mathematical texts with a richer semantics: it would be easier and more robust to copy and paste formulas between a text and a computer algebra system, one might search for formulas on websites such as Wikipedia, various kinds of "typos" in formulas can be detected automatically while entering formulas, etc.

*. This work has been supported by the ANR-09-JCJC-0098-01 MaGiX project, as well as a Digiteo 2009-36HD grant and Région Ile-de-France.

Currently, most mathematicians write their documents in TEX, LATEX, or one of its graphical front-ends [15, 17, 5, 29, 21]. Such documents usually focus on presentation and not on mathematical correctness, not even syntactic correctness. In the past decade, we have developed GNU TEXMACS [32, 33] as an alternative structured wysiwyg text editor. TEXMACS does not rely on LATEX, and can be freely downloaded from `http://www.texmacs.org`. The main aims of TEXMACS are user-friendliness, high quality typesetting, and its use as an interface for external systems [12, 2, 13, 18]. However, until recently, mathematical formulas inside TEXMACS only carried presentation semantics.

Giving mathematical formulas more semantics can be regarded as a gradual process. Starting with formulas which only carry presentation semantics, we ultimately would like to reach the stage where all formulas can be checked by a formal proof checker [28, 30, 20, 19]. In between, the formulas might at least be correct from the syntactic point of view, or only involve non ambiguous symbols.

In this paper, we focus on techniques for making it easy for authors to create papers in which all formulas are correct from the syntactic point of view. In the MATHML [26] jargon, this means that we can produce both *presentation markup* and *content markup* for all formulas. From now on, when we will speak about *semantics*, then we will always mean syntactical semantics. For instance, in the formula $x + y$ we merely want to detect or enforce that the operator $+$ applies to x and y; we are not interested in the actual mathematical meaning of addition.

If we are allowed to constraint the user to enter all texts in a way which is comfortable for the designer of the editor, then syntactic correctness can easily be achieved: we might constrain the user to directly enter content markup. Similarly, if the author is only allowed to write text in a specific mathematical sub-language (e.g. all formulas which can be parsed by some computer algebra system), then it might be possible to develop *ad hoc* tools which fulfill our requirements.

Therefore, the real challenge is to develop a general purpose mathematical editor, which imposes minimal extra constraints on the user with respect to a presentation-oriented editor, yet allows for the production of syntactically correct documents. As often in the area of user interfaces, there is a psychological factor in whether a particular solution is perceived as satisfactory: certain hackers might happily enter MATHML content markup in VI. Instead of proposing a single best solution, we will therefore present a collection of ideas and techniques for simplifying the task of writing syntactically correct formulas; time will learn us which of them will be most useful for the typical end-user.

One of the central ideas behind our approach is to stick as much as possible to an existing user friendly and presentation-oriented editor, while using a "syntax checker/corrector" in the background. This syntax checker will ensure that it always possible to formally parse the formulas in the document, while these formulas are represented using presentation markup. We will see that this is possible after a few minor modifications of the editor and the development of a suitable series of tools for the manipulation of formal languages. All our implementations were done in TEXMACS, but our techniques might be implemented in other editors.

An interesting related question is whether the production of syntactically correct documents is indeed best achieved during the editing phase: there might exist some magical algorithm for giving more semantics to most existing LATEX documents. In this paper, we will also investigate this idea and single out some of the most important patterns which cause problems. Of course, from the perspective of a software with a non trivial user base, it is desirable that the provided syntax corrector can also be used in order to upgrade existing documents.

The paper is organized into two main parts: the first part (sections 2, 4 and 3) does not depend on the packrat-based formal language tools, whereas the second part (sections 5, 6 and 7) crucially depends on these tools. The last section 8 contains some applications and ideas for future developments.

In section 2, we first review the specificities of various formats for mathematical formulas. In section 3, we discuss elementary techniques for giving existing documents more semantics and the main obstacles encountered in this process. In section 4, we present a few minimal changes which were made inside TEXMACS in order to remedy some of these obstacles.

In section 5, we recall the concept of a *packrat parser* [9, 10]. This kind of grammars are both natural to specify, quick to parse and convenient for the specification of on-the-fly grammars, which can be locally modified inside a text. We have implemented a formal language facility inside TEXMACS, which is based on packrat parsing, but also contains a few additional features. In section 6, we will discuss the development of a standard grammar for mathematics, and different approaches for customizating this grammar. In section 7, we present several grammar-assisted editing tools which have been implemented inside TEXMACS.

In order to analyze the adequacy of the proposals in this paper, we have made a selection of a few books and other documents from various sources, and tested our algorithms on this material (1303 pages with 59504 formulas in total). These experiments, which are discussed in detail in sections 3.4 and 6.4, have given us a better understanding of the most common syntactical errors, how to correct them, the quality of our parsing grammar and remaining challenges.

We have one our best to make our experiments as reproducible as possible. Our results were obtained using the SVN development revision 4088 of TEXMACS, which should be more or less equivalent to the unstable release 1.0.7.10. For detailed information on how to use the semantic editing features, please consult the integrated documentation by clicking on the menu entry Help→Manual→Mathematical formulas. The present document is also intended as a source for complementary information for TEXMACS users who are interested in semantic mathematical editing. With the kind permission of their authors, the test documents are publicly available from

`http://www.texmacs.org/Data/semedit-data.tar.gz`
`http://perso.univ-rennes1.fr/marie-francoise.roy/bpr-ed2-posted2-29122010.tar.gz`

As a disclaimer, we would like to add that we take responsibility for any bad usage that the authors may have made of TEXMACS. Except for a few genuine typos, the semantic errors which were detected in their work are mostly due to current imperfections in TEXMACS.

Taking into account the numerous existing formula editors, it is a subtle task to compare every single idea in this paper to previous work. Indeed, many of the individual ideas and remarks can probably be traced back to earlier implementations. However, such earlier implementations are often experiments in a more restricted context. We recall that the aim of the current paper is to provide a general purpose tool with the average "working mathematician" (or scientist) as its prototype user. The main challenge is thus to fit the various building bricks together such that this goal is achieved.

Nevertheless, we would like to mention a few people who directly influenced this paper. First of all, the decision to make the user interface of TEXMACS purely graphically oriented was inspired by a talk of O. ARSAC on the EMATH system [1]. This system was merely intended as a general purpose interface for computer algebra systems. It also inte-

grated a parser for making the editor's behaviour depend on the semantics, although the decoupling between the editor and the parser was not as clean as in our present implementation. We are also grateful to H. LESOURD for pointing us to the concept of a packrat grammar. Some other related, but unpublished work on semantic editing was done jointly with T. NEUMANN and S. AUTEXIER. However, we follow a different approach in the present paper.

We finally notice that some semantic editing features are integrated into the proprietary computer algebra systems MAPLE and MATHEMATICA [11, 27]. However, these features are not that much intended for general purpose mathematical editing, but rather for the creation of documented worksheets. Since scientists do not receive free copies of these systems, we were unable to evaluate the features in detail.

Remark 1. After prepublication of the first version of this paper in januari 2011, we have successfully been using the new semantic features ourselves. The following adjustments, which were made in version 1.0.7.11 of TEXMACS, address some of the remaining problems:

1. It is better to treat big operators as prefix symbols than to introduce explicit markup for skoping such operators. This required the universal grammar in section 6 to be tweaked accordingly.
2. We have modified the rendering of bounding boxes for the environments around the cursor, by putting a higher emphasis on the innermost environment, the current focus (see section 7.3).
3. We have implemented a mode in which hints for invisible multiplications are rendered on the screen (e.g. $a \cdot b$ instead of $a\,b$).

It also seems that semantic selection (see section 7.4) is more often a harm than a good (especially now that the presentation markup enforces bracket matching for selections).

2. SURVEY OF FORMATS FOR MATHEMATICAL FORMULAS

2.1. LATEX

The current standard for mathematical documents is TEX/LATEX [15, 17]. There are three main features which make TEX convenient for typing mathematical documents:

1. The use of a cryptic, yet suggestive pseudo-language for ASCII source code.
2. The possibility for the user to extend the language with new macros.
3. A high typographic quality.

In addition, LATEX provides the user with some rudimentary support for structuring documents.

One major drawback of TEX/LATEX is that it is not really a data format, but rather a programming language. This language is very unconventional in the sense that it does not admit a formal syntax. Indeed, the syntax of TEX can be "hacked" on the fly, and may for instance depend on the parity of the current page. This is actually one important reason for the "success" of the system: the ill-defined syntax makes it very hard to reliably convert existing documents into other formats. In particular, the only reliable parser for TEX is TEX itself.

If, for convenience, we want to consider TEX/LATEX as a format, then we have to restrict ourselves to a sublanguage, on which we impose severe limits to the definability of new macros and syntax extensions. Unfortunately, in practice, few existing documents conform to such more rigourous versions of LATEX, so the conversion of TEX/LATEX to other formats necessarily involves a certain amount of heuristics. This is even true for papers which conform to a journal style with dedicated macros, since the user is not forced to use only a restricted subset of the available primitives.

Even if we ignore the inconsistent syntax of TEX, another drawback of TEX/LATEX is its lack of formal semantics. Consider for instance the TEX formulas `$a(b+c)$` and `$f(x+y)$`. Are `a` and `f` variables or functions? In the formulas `$\sum_i a_i+C$` and `$\sum_i a_i+b_i$`, what is the scope of the big summation? Should we consider `$[a,b[$` to be incorrect, or did we intend to write a french-style interval? Some heuristics for answering these questions will be discussed below.

There are more problems with TEX/LATEX, such as inconsistencies in semantics of the many existing style packages, but these problems will be of lesser importance for what follows.

2.2. MathML

The MATHML format [26] was designed from scratch, with two main objectives:

1. Having a standard format for mathematics on the web, which assumes compatibility with existing XML technology.
2. Defining a DTD with a precise semantics, which is sufficient for the representation of mathematical formulas.

In particular, the MATHML format provides a remedy to most of the major drawbacks of TEX/LATEX that we mentioned above.

In fact, MATHML is divided into two main parts. The first part concerns *presentation markup* and provides constructs for describing the presentation of mathematical formulas. The second part concerns *content markup* and describes the actual syntactic meaning of a mathematical expression.

For instance, returning to the TEX formula `$a(b+c)$`, the typical presentation markup would be as follows

```
<mrow>
  <mi>a</mi>
  <mo>&#x2062;<!-- &InvisibleTimes; --></mo>
  <mrow>
    <mo>(</mo>
    <mi>b</mi>
    <mo>+</mo>
    <mi>c</mi>
    <mo>)</mo>
  </mrow>
</mrow>
```

We observe two interesting things here: first of all, the invisible multiplication makes it clear that we intended to multiply a with $b+c$ in the formula $a\,(b+c)$. Secondly, the (optional) inner `<mrow>` and `</mrow>` suggest the scope and intended meaning of the brackets.

In principle, the above piece of presentation markup therefore already carries enough semantics so as to produce the corresponding content markup:

```
<apply>
  <times/>
  <ci>a<ci>
  <apply>
    <plus/>
    <ci>b</ci>
    <ci>c</ci>
  </apply>
</apply>
```

This fragment of code can be regarded as a verbose variant of the SCHEME [22] expression

```
(* a (+ b c))
```

More generally, as illustrated by the above examples, MATHML tends to be very precise and fairly complete, but also very verbose. For this reason, MATHML has not yet succeeded to impose itself as the new standard for the working scientist. Browser support has also been quite poor, although this situation is slowly improving.

2.3. TEXMACS

The three main objects of the original TEXMACS project were the following:

1. Provide a free and user friendly wysiwyg *editor* for mathematical formulas.
2. Provide a typesetting quality which is as least as good as the quality of TEX.
3. Make it possible to use TEXMACS as an interface for other software, such as computer algebra systems, while keeping a good typesetting quality for large formulas.

One consequence of the first point is that we require an internal format for formulas which is not read-only, but also suitable for modifications. For a wysiwyg editor, it is also more convenient to work with tree representations of documents, rather than ASCII strings. TEXMACS provides three ways to "serialize" internal trees as strings (native human readable, XML and SCHEME). Nevertheless, the important issue is not that much the syntax of the format, but rather a sufficiently rich semantics which makes it possible to *convert* documents into other formats. In this respect, modern MATHML would have been perfectly suitable as a format for TEXMACS.

In order to be as compatible with TEX/LATEX as possible, the original internal TEXMACS format very much corresponded to a "clean" tree representation for TEX/LATEX documents. Some drawbacks of TEX, such as lacking scopes of brackets and big operators, were therefore also present in TEXMACS. On the other hand, TEXMACS incites users to make a distinction between multiplication and function application, which is important for the use of TEXMACS as an interface to computer algebra systems. For instance, $a\ (b + c)$ is entered by typing `a * ( b + c )`, and internally represented as a string leaf

```
a*(b+c)
```

in all TEXMACS versions prior to 1.0.7.6. More generally, TEXMACS provides non ambiguous symbols for various mathematical constants (e, π, i, etc.) and operators. However, as a general rule, traditional TEXMACS documents remain presentation oriented.

2.4. Other formats

There are many other formats for the machine representation of mathematical texts. SCHEME or LISP expressions have traditionally been used as a precursor for MATHML content markup, but without a similar effort towards standardization. In this paper, we will use SCHEME notation for content markup, since it is less verbose than MATHML. We notice that SCHEME is also used inside TEXMACS as an extension language (similar to EMACS-LISP in the case of EMACS).

For some applications, such as automatic theorem proving [28, 30, 20, 19] and communication between computer algebra systems, it is important to develop mathematical data formats with even more semantics. Some initiatives in this direction are OPENMATH [24] and OMDOC [16]. However, these topics are beyond the scope of this paper.

3. BASIC SYNTAX CORRECTION

3.1. Common ambiguities

The multiplication versus function application ambiguity mentioned in section 2.1 is probably the most important obstacle to the automatic association of a semantics to mathematical formulas. There are several other annoying notational ambiguities, which are mostly due to the use of the same glyph for different purposes. In this section, we will list the most frequent ambiguities, which were encountered in our own documents and in a collection of third party documents to be described in section 3.4 below.

Invisible operators. Besides multiplication and function application, there are several other invisible operators and symbols:

- Invisible separators, as in matrix or tensor notation $A = (a_{ij})$.
- Invisible addition, as in $17\,{}^{3}\!/_{8}$.
- Invisible "wildcards", as in the increment operator $+1$ (also denoted by $\cdot + 1$)
- Invisible brackets, for forced matching if we want to omit a bracket.
- Invisible ellipses as in the center of the matrix $\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix}$.
- Invisible zeros as in $\begin{pmatrix} a_{11} & & \\ & \ddots & \\ & & a_{nn} \end{pmatrix}$.

Invisible operators have been incorporated in the UNICODE standard [25], even though invisible wildcards, brackets, ellipses and zeros are not yet supported. We also notice that function application is implicit in formulas such as $f(x)$ and explicit in formulas such as $\sin x$ or Kf.

Vertical bars. Vertical bars are used in many circumstances:

- As brackets in absolute values $|x|$ or "ket" notation $\langle A|$.
- As "such that" separators in sets $\{a \in X \,|\, a \geqslant 0\}$ or lists.
- As the "divides" predicate $11 \,|\, 1001$ (both in binary and decimal notation).

- As separators $\langle a_1|\cdots|a_n\rangle$.
- For restricting domains or images of applications: $\varphi|_D$, $\varphi|^I$.

The possibility to use bars as brackets is particularly problematic for parsers, since it is not known whether we are dealing with an opening bracket, a closing bracket, or no bracket at all.

Ponctuation. The comma may either be a separator, as in $f(x,\ y)$, or a decimal comma, as in 3,14159$\cdots$, or a grouping symbol, as in 1,000,000. The period symbol "." can be used instead of a comma in numbers, but also as a data access operator $a.b$, or as a connector in lambda expressions $\lambda x.x^2$. Moreover, in the formula

$$a^2+b^2=c^2,$$

ponctuation is used in the traditional, non-mathematical way. The semicolon ":" is sometimes used as an alternative for such that (example: $\{x\in X: x\geqslant 0\}$), but might also indicate division or the binary infix "of type", as in $a:\mathsf{Int}$.

Miscellaneous homoglyphs. In what follows, *homoglyphs* will refer to two semantically different symbols which are graphically depicted by the same glyph (and a potentially different spacing). In particular, the various invisible operators mentioned above are homoglyphs. Some other common homoglyphs are as follows:

- The backslash $\setminus$ is often used for "subtraction" of sets $X\setminus Y$.
- The dot $\cdot$ can be used as a symbol for multiplication (example: $\boldsymbol{a}\cdot\boldsymbol{b}$) or as a wildcard (example: the norm $|\cdot|_p$).
- The wedge $\wedge$ can be used as the logical and operator or as the wedge product.

It should be noticed that UNICODE support for homoglyphs is quite poor; see section 7.6 for a discussion.

3.2. Common errors

Authors who are not aware of the ambiguities described in the previous section are inclined to produce documents with syntax errors. Following TEX/LATEX habits, some authors consistently omit multiplications and function applications. Others systematically replace multiplications by function applications. Besides errors due to ambiguities, the following kinds of errors are also quite common:

Superfluous spaces. Inside TEXMACS, where we recall that spaces correspond to function application, users often leave superfluous spaces at the end of formulas (after changing their mind on completing a formula) or around operators (following habits for typing ASCII text).

Misplaced invisible markup. In wysiwyg editors, context changes (such as font changes, or switches between text mode and math mode) are invisible. Even though TEXMACS provides visual hints to indicate the current context, misplaced context changes are a common mistake. Using LATEX notation, the most common erroneous or inappropriate patterns are as follows:

- Misplaced parts: `\begin{math}f(x\end{math})`.
- Breaks: `\begin{math}a+\end{math}\begin{math}b\end{math}`.
- Redundancies I: `\begin{math}a+\begin{math}b\end{math}\end{math}`.

- Redundancies II: `\begin{math}\text{hi}\end{math}`.

Notice that some of these patterns are introduced in a natural way through certain operations, such as copy and paste, if no special routines are implemented in order to avoid this kind of nuisance.

Bracket mismatches. Both in ASCII-based and wysiwyg presentation-oriented editors, there is no reliable way to detect matching brackets, if no special markup is introduced to distinguish between opening and closing brackets. A reasonable heuristic is that opening and closing brackets should be of the same "type", such as (and) or [and]. However, this heuristic is violated for several common notations:

- Interval notation $[a, b)$ or $[a, b[$.
- Ket notation $\langle A|$.

Absolute values are also particularly difficult to handle, since the opening and closing brackets coincide.

Unknown scope of big operators. If no special markup is present to indicate the scope of a big operator, then it is non trivial to determine appropriate scopes in formulas such as `\sum_i a_i + K` and `\sum_i a_i + b_i`. Although old versions of TEXMACS provided an invisible closing bracket, users (including ourselves) tended to ignore or misuse it (of course, this really means that the introduction of invisible closing brackets was not the right solution to the scoping problem).

One common aspect of all these errors is that authors who only care about presentation will usually not notice them at all: the printed versions of erroneous and correct texts are generally the same, or only differ in their spacings.

3.3. Syntax correction

TEXMACS implements several heuristics to detect and correct syntax errors. Some of these heuristics are "conservative" in the sense that they will only perform corrections on incorrect texts and when we are sure or pretty sure that the corrections are right. Other heuristics are more "agressive" and may for instance replace spaces by multiplications or *vice versa* whenever this seems reasonable. However, in unlucky cases, the agressive heuristics might replace a correct symbol by an unwanted one. Notice that none of the syntactic corrections alters the presentation of the document, except for some differences in the spacing.

One major technique which is used in many of the heuristics is to associate a symbol type to each symbol in a mathematical expression. For instance, we associate the type "infix" to $\wedge$, "opening bracket" to (, and special types to subscripts, superscripts and primes. Using these types, we can detect immediately the incorrectness of an expression such as $(\wedge b) \Rightarrow (b \wedge a)$. Some of our algorithms also rely on the binding forces of mathematical symbols. For instance, the binding force of multiplication is larger than the binding force of addition.

It is important to apply the different algorithms for syntax correction in an appropriate order, since certain heuristics may become more efficient on partially corrected text. For instance, it is recommended to replace `$f($)` by `$f(x)$` before launching the heuristic for determining matching brackets. We will now list, in the appropriate order, the most important heuristics for syntax correction which are implemented in TEXMACS.

Correction of invisible markup. It is fairly easy to correct misplaced invisible markup, *via* detection and correction of the various patterns that we have described in the previous section.

Bracket matching. There are several heuristics for the determination of matching brackets, starting with the most conservative ones and ending with the most agressive ones if no appropriate matches were found. The action of the routine for bracket matching can be thought of as the insertion of appropriate `mrow` tags in MATHML or `{}` pairs in LaTeX. For instance, `I=[a,b[` might be transformed into `I={[a,b[}`.

Each of the heuristics proposes an "opening", "closing", "middle" or "unknown" status for each of the brackets in the formula and then launches a straightforward matching algorithm. The first most conservative heuristic first checks whether there are any possibly incorrect brackets in the formula, such as the closing [bracket in $[a, b[$, and turns their status into "unknown". The status of the remaining brackets is the default one: opening for (, [, {, etc. and closing for },],).

We next launch a series of more and more agressive heuristics for the detection of particular patters, such as french intervals $[a, b[$, absolute values $|a|$, ket notation $\langle a|$, etc. If, at the end of all these heuristics, some brackets still do not match, then we (optionally) match them with empty brackets. For instance, `$\left(a$` will be turned into `$\left(a\right.$`.

We notice that the heuristics may benefit from matching brackets which are detected by earlier heuristics. For instance, in the formula $f(|x|) = g(|x| + |y|)$, the brackets () are matched at an early stage, after which we only need to correct the subformulas $|x|$ and $|x| + |y|$ instead of the formula as a whole.

Scope of big operators. The determination of scopes of big operators is intertwined with bracket matching. The main heuristic we use is based on the binding forces of infix operations inside the formula and the binding forces of the big operators themselves. For instance, in the formula

$$\sum_{i=1}^{\infty} a_i = \sum_{i=1}^{\infty} b_i \sum_{j=1}^{\infty} c_{i,j}, \tag{1}$$

the binding force of the equality is weaker than the binding force of a summation, whereas the binding force of the invisible multiplication is larger. In correct formulas, this heuristic corresponds to interpreting the summation as a prefix operator with a suitable binding force (see section 6.2.3).

This heuristic turns out to be extremely reliable, even though it would be "incorrect" on the formula

$$\sum_{i} a_i + b_i, \tag{2}$$

where the big summation is taken to have a slightly larger binding force than the addition. The heuristic might be further improved by determining which summation variables occur in which summands. For instance, in the formula (1), the occurrence of i inside $c_{i,j}$ gives a second indication that we are dealing with a nested summation. Notice however that it is not clear whether it pays off to further improve the heuristic, since notations as in (2) are somewhat clumsy anyway.

Removing superfluous invisible symbols. Trailing or other superfluous invisible symbols are easily detected and removed from formulas.

Homoglyph replacement. Another easy task is the detection of errors which can be repaired through the replacement of a symbol by a homoglyph. For instance, if the backslash symbol (with type "basic symbol") occurs between two letters (also with types "basic symbol"), as in $X \backslash Y$, then we may replace it by the "setminus" infix.

Insertion of missing multiplications and function applications. The insertion of missing invisible symbols (or replacement of incorrect ones) is one of the hardest tasks for syntax correction. Two main situations need to be distinguished:

- The text to be corrected was written with TEXMACS and we may assume that most of the multiply/apply ambiguities were already resolved by the author.
- The text to be corrected was imported from TEX/LATEX, so that no multiplications or function applications were present in the source document.

In the first case, most multiplications and function applications are already correct, so the correction algorithm should be very conservative. In the second case, the insertions should be very agressive, at the risk of being incorrect in many cases.

At a first stage, it is important to determine situations in which it is more or less clear whether to insert a multiplication or a function application. For instance, between a number and a letter, it is pretty sure that we have to insert a multiplication. Similarly, after operators such as sin, it is pretty sure that we have to insert a space.

At a second stage, we determine how all letter symbols occurring in the document are "used", i.e. whether they occur as lefthand or righthand arguments of multiplications or function applications. It is likely that they need to be used in a similar way throughout the document, thereby providing useful hints on what to do.

At the end, we insert the remaining missing symbols, while pondering our decisions as a function of the hints and whether we need to be conservative or not.

3.4. Experimental results

In this section, we report on the performance of the TEXMACS syntax corrector. The results were obtained by activating the debugging tool inside TEXMACS as well as the Debug→Correct option. This allows the user to display detailed status reports on the number of errors in all loaded files using Debug→Mathematics→Error status report. We have tested our corrector on various types of documents from various sources:

BPR. This book on algorithms in real algebraic geometry was originally written in LATEX [4] and then converted to TEXMACS and transformed into a book with extra interactive features [3].

BPR$_k$. The k-th chapter of BPR.

COL. This corresponds to a collection [6] of six courses for students on various topics in mathematics.

COL$_3$. The third paper in COL on information theory.

MT. This Master's Thesis [23] was written by an early TEXMACS user and physicist.

LN. Our own book on transseries, which was started quite long ago with an early version of TEXMACS and completed during the last decade [34].

HAB. This habilitation thesis [35] was written more recently.

HAB$_k$. The k-th chapter of HAB.

In the introduction, we have mentioned URLs where these documents can be downloaded. We also repeat that TEXMACS (and not the authors) should be held responsible for most of the semantic errors which were found in these documents.

In table 1, we have compiled the main results of the syntax corrector. Concerning the total number of formulas, we notice that each non empty entry of an equation array is counted as a separate formula. For the books, we only ran the corrector on the main chapters, not on the glossaries, indexes, etc. The concept of "error" will be described in more detail in section 7.1 below. Roughly speaking, a correct formula is a formula that can be parsed in suitable (tolerant) grammar, which will be described in section 6. In section 6.4, we will discuss in more detail to which extent our notion of "error" corresponds to the intuitive notion of a syntax error or "typo".

The corrector tends to reduce the number of errors by a factor between 3 and 15. The typical percentage of remaining errors varies between 0.5% and 5%. This error rate tends to be slightly higher for documents with many complex formulas, such as MT; in ordinary mathematical documents, most formulas are very short, whence trivially correct. Furthermore, many errors tend to be concentrated at certain portions of a document. For instance, the error rate of the individual document COL_3 is as about twice as low as the error rate of the collection COL. Similarly, the bulk of the errors in LN are concentrated in the first chapters. Finally, errors tend to be repeated according to patterns, induced by systematic erroneous notation.

Document	BPR	BPR_1	BPR_2	COL	COL_3	MT	LN	HAB
Total number of formulas	30394	883	2693	13048	2092	1793	12626	1643
Initial number of errors	2821	63	221	4158	607	308	629	61
Errors after syntax correction	705	35	53	543	37	86	98	6
Number of pages	585	16	48	357	56	85	233	43

Table 1. Global performance of the $\mathrm{TEX}_{\mathrm{MACS}}$ syntax corrector on various documents.

It is interesting to study the relative correction rates of the techniques described in the previous section. More precisely, we have implemented the following algorithms:

Invisible tag correction. Correction of invisible structured markup.

Bracket matching. Since all documents were written with older versions of $\mathrm{TEX}_{\mathrm{MACS}}$ in which bracket matching was not enforced (see also section 4.2 below), this algorithm implements a collection of heuristics to make all brackets match. In the case of absolute values, which cannot be parsed before this step, this typically decreases the number of errors. The algorithm also determines the scopes of big operators.

Bracket motion. This algorithm detects and moves incorrectly placed brackets with respect to invisible structured markup. For instance, the LaTeX formula `Let $y=f(x$).` would be corrected into `Let $y=f(x)$.`

Superfluous invisible removal. Removal of superfluous invisible symbols.

Missing invisible insertion. Heuristic insertion of missing invisible symbols, either function applications or multiplications.

Homoglyph substitution. Replacing symbols by appropriate homoglyphs.

Miscellaneous corrections. Various other useful corrections.

The results are shown in table 2.

In a sense, among the original errors, one should not really count the errors which are corrected by the bracket matching algorithm. Indeed, the versions of $\mathrm{TEX}_{\mathrm{MACS}}$ which were used for authoring the documents contained no standard support for, say, absolute values or big operator scopes. Hence, the author had no real possibility to avoid parsing errors caused by such constructs. Fortunately, our bracket matching algorithm is highly effective at finding the correct matches. One rare example where the algorithm "fails" is the formula $(X', |\cdot|)$ from COL, where $\cdot$ was tagged to be a multiplication.

The overwhelming source of remaining error corrections is due to missing, wrong or superfluous invisible multiplications and function applications. This fact is not surprising for documents such as BPR, which were partially imported from LaTeX. Curiously, the situation seems not much better for MT, COL and IT, which were all written using TeXmacs. Even in our own documents LN and HAB, there are still a considerable number of "invisible errors". This fact illustrates that TeXmacs provides poor incitation to correctly enter meaningful invisible content. In COL, the author actually took the habit to systematically replace multiplications by function applications. Consequently, many of the "correct" formulas do not have the intended meaning.

Document	BPR	BPR_1	BPR_2	COL	COL_3	MT	LN	HAB
Invisible tag correction	92	0	3	10	5	16	56	3
Bracket matching	676	14	67	1236	94	49	282	29
Bracket motion	16	0	4	6	0	1	4	0
Superfluous invisible removal	382	1	25	1046	305	115	149	11
Missing invisible insertion	873	11	56	1271	164	40	33	6
Homoglyph substitution	44	2	7	45	2	1	6	0
Miscellaneous corrections	16	0	6	1	0	0	1	6

Table 2. Numbers of corrections due to individual algorithms.

4. Basic semantic editing

4.1. Introduction

In section 3, we have described many common ambiguities and syntactical errors, and provided heuristics for correcting them. Of course, a more satisfactory solution would be to develop mathematical editors in such a way that these errors are ruled out right from the start, or at least, that it is harder for the user to enter obviously incorrect formulas.

In the process of making mathematical editors more sophisticated, one should nevertheless keep in mind that most authors are only interested in the presentation of the final document, and not in its correctness. For most users, presentation oriented interfaces are therefore the most intuitive ones. In particular, every time that we deviate from being presentation oriented, it should be clear for the user that it buys him more than he looses.

There are three basic techniques for inciting the user to write syntactically correct documents:

- Enforcing correctness *via* the introduction of suitable markup.
- Providing visual hints about the semantics of a formula.
- Writing documentation.

Of course, the main problem with the first method is that it may violate the above principle of insisting on presentation oriented editing. The problem with the second method is that hints are either irritating or easily ignored. The third option is well-intentioned, but who reads and follows the documentation? One may also wish to introduce one or more user preferences for the desired degree of syntactic correctness.

In some cases, correctness can also be ensured by running appropriate syntax correction methods from section 3.3. For instance, the algorithms for the correction of invisible markup can be applied by default. Alternatively, one may redesign the most basic editing routines so as to ensure correctness all the way along.

4.2. Enforcing matching brackets

Matching brackets and well-scoped big operators are particularly important for structuring mathematical formulas. It is therefore reasonable to adapt markup so as to enforce these properties. In a formula with many brackets, this is usually helpful for the user as well, since it can be tedious to count opening and closing brackets. Highlighting matching brackets consists an alternative solution, which is used in many editors for programming languages.

In TEXMACS 1.0.7.7, we have introduced a special ternary markup element `around` (similar to `fenced` in MATHML), which takes two brackets and a body as its arguments. A possible future extension is to allow for additional pairs of middle brackets and extra arguments. The typical behaviour of a "matching bracket mode" is as follows:

- When typing an opening bracket, such as (, the corresponding closing bracket) is inserted automatically and the cursor is placed in between (|).
- The brackets are removed on pressing `backspace` or `delete` inside a pair of matching brackets with nothing in between.
- The `around` tag has "no border", in the sense that, in the formula $f(x) + y$, there is only one accessible cursor position between f and (resp.) and $+$.

In addition, one has to decide on ways to replace the opening and closing brackets, if necessary, thereby enabling the user to type intervals such as $[a, b)$ or $[a, b[$, or other kinds of "matching brackets". In TEXMACS, this is currently done as follows:

- When removing an opening or closing bracket, it is actually replaced by an invisible bracket. In the special case that there was nothing between the brackets, or when both brackets become invisible, we remove the brackets. Notice that the cursor can be positioned before or after an invisible bracket.
- If we type an opening or closing bracket just before or after an invisible bracket, then we replace the invisible bracket by the new bracket.

For instance, we may enter $[a,\ b)$ by typing `[ a , b delete )`. In addition, we use the following convenient rule:

- When typing a closing bracket just before the closing bracket of an `around` tag, we replace the existing closing bracket by the new one.

An alternative way for entering $[a, b)$ is therefore to type `[ a , b )`. Notice that the last rule also makes the "matching bracket mode" more compatible with the naive presentation oriented editing mode. Indeed, the following table shows in detail what happens when typing $f(x)$ in both modes:

	presentation oriented mode	matching bracket mode
`f`	$f\vert$	$f\vert$
`f (`	$f(\vert$	$f(\vert)$
`f ( x`	$f(x\vert$	$f(x\vert)$
`f ( x )`	$f(x)\vert$	$f(x)\vert$

Notice that the shortcut f (x → is equivalent to f (x) in this example.

The above rules also apply for bar-like brackets, except that the bar is either regarded as an opening or closing bracket, depending on the context. For instance, when typing | in an empty formula, we start an absolute value $|\,|$. On the other hand, when the cursor is just before a closing bracket, as in $\langle A|\rangle$, then typing | results in the replacement of the closing bracket by a bar: $\langle A||$. As will be explained below, other bar-like symbols can be obtained using the tab key.

Another question is whether big operators should be scoped in the matching bracket mode. We have examinated this kind of markup based scoping in version 1.0.7.10 of TEXMACS. As in the case of an invisible bracket, there are two cursor positions next to the invisible righthand border of a scoped big operator: inside and outside the scope of the operator. We use light cyan boxes in order to indicate the *current focus* (see section 7.3), i.e. precise position of the cursor:

$$\boxed{\sum_{i=1}^{\infty} a_i|}=\infty \qquad\qquad \sum_{i=1}^{\infty} a_i|=\infty$$

Inside the scope Outside the scope

However, it is easy for the user to ignore these visual hints. For instance, the user might incorrectly enter an equality sign at the end inside the scope of a big sum. Indeed, in the printed version of a document, there is no way to notice such incorrect scopes. In fact, our heuristic for the determination of scopes of big operators works "so well" that it turns out to be better to consider big operators as prefix symbols (see section 6.2.3), as we do in version 1.0.7.11 of TEXMACS. In the case of a formula such as $\sum_i a_i + b_i$, for which our heuristic fails, the formula would actually become more readable if we put brackets around $a_i + b_i$.

A similar problem is to determine the bases of subscripts and superscripts in expressions such as $a_1 + a_2$ and $(a+b)^3 + c$. Again, this might be done *via* the introduction of explicit markup. In a formula such as $a + b^2$, this would lead to two accessible cursor positions between $+$ and b, depending on whether we are outside or inside the base of the script. However, this would create an invisible border which is even less natural than in the case of big operators. Indeed, scripts are postfixes, so we need to tag the base of the script *a priori*. For this reason, TEXMACS only allows for the determination of bases of scripts using grammar based techniques, as described in section 6.

4.3. Homoglyphs and the "variant" mechanism

TEXMACS provides a simple "variant" mechanism, based on using the tab key. For instance, Greek letters α, β, Λ, etc. can be entered by typing a tab, b tab, L tab, etc. Similarly, the variants $\leq$, $\leqq$ and $\Leftarrow$ of $\leqslant$ can be obtained by typing < = and pressing the tab key several times. Using the variant mechanism, there are also simple shortcuts e tab tab, p tab tab, i tab tab and g tab tab for the most frequent mathematical constants e, π, i and γ. Similarly, d tab tab can be used for typing the operator d in $\int f(x)\,\mathrm{d}x$.

It is natural to use the variant mechanism for disambiguating homoglyphs as well. In table 3, we have displayed the current keyboard shortcuts for some of the most common homoglyphs.

Shortcut	Example	Semantics
`*`	$a\,b$	Invisible multiplication
`space`	$\sin x$	Invisible function application
`+` `tab` `tab`	$2\,{}^1\!/_2$	Invisible addition
`,` `tab` `tab`	a_{ij}	Invisible separator
`.` `tab` `tab` `tab`	$+x$	Invisible symbol
`,`	$f(x, y)$	Comma separator
`,` `tab`	3,14159$\cdots$	Decimal comma
`:`	$\{x \in R\colon x \geqslant 0\}$	Such that separator
`:` `tab`	$x\colon X$	Of type infix
`|`	$\|x\|$	Absolute value
`|` `tab`	$\{x \in R \| x \geqslant 0\}$	Such that separator
`|` `tab` `tab` `tab`	$11 \mid 1001$	Divides

Table 3. TEXMACS shortcuts for common homoglyphs.

Regarding invisible operators, there are a few additional rules. First of all, letters which are not separated by invisible operators are considered as operators. For instance, typing `a` `b` yields ab and not $a\,b$, as in LATEX. This provides some incitation for the user to explicitly enter invisible multiplications or function applications. Also, the editor forbids entering sequences of more than one invisible multiplication or function application. In other words, typing `a` `*` `*` `b` has the same effect as typing `a` `*` `b`.

Besides making it easy for the user to disambiguate homoglyphs, the editor also provides visual hints. For instance, the difference between the vertical bar as a separator or a division is made clear through the added spacing in the case of divisions. The status bar also provides information about the last symbol which has been entered.

Despite the above efforts, many users don't care about syntactic correctness and disambiguating homoglyphs, or are not even aware of the possibilities offered by the editor. Unfortunately, it is hard to improve this situation: a very prominent balloon with relevant hints would quickly irritate authors. Yet, for users who do not notice spacing subtleties or discrete messages on the status bar, more agressive messages are necessary in order to draw their attention.

One possible remedy would be to display more agressive help balloons only if the user's negligence leads to genuine errors. For instance, whenever the user types `a` `space` `+`, the `space` is clearly superfluous, thereby providing us a good occasion for pointing the user to the documentation.

In the case of invisible operators, we might also display the corresponding visible operator in a somewhat dimmed color, while reducing its size, so as to leave the spacing invariant. Optionally, these hints might only be displayed if the cursor is inside the formula.

5. PACKRAT GRAMMARS AND PARSERS

5.1. Survey of packrat parsing

The grammars of most traditional computer languages are LALR-1 [7], which makes it possible to generate parsers using standard tools such as YACC or BISON [14, 8]. However, the design and customization of LALR-1 grammars is usually quite non trivial. Recently, packrat grammars were introduced as a remedy to these drawbacks [9, 10]. A packrat grammar consists of:

- A finite alphabet Σ of *non-terminal symbols*.

- A finite alphabet T of *terminal symbols* (disjoint from Σ).
- One *parsing rule* $\rho_\sigma \in \mathrm{P}_{\Sigma,\mathrm{T}}$ for each non-terminal symbol σ in Σ.

The set $\mathrm{P}_{\Sigma,\mathrm{T}}$ stands for the set of *packrat parsing expressions*. There are various possibilities for the precise definition of $\mathrm{P}_{\Sigma,\mathrm{T}}$. In what follows, each parsing expression is either of one of the following forms:

- A non-terminal or terminal symbol in $\Sigma \amalg \mathrm{T}$.
- A (possibly empty) concatenation $\rho_1 \cdots \rho_n$ of parsing expressions $\rho_1, \ldots, \rho_n$ in $\mathrm{P}_{\Sigma,\mathrm{T}}$.
- A (possibly empty) *sequential* disjunction $\rho_1 / \cdots / \rho_n$ of $\rho_1, \ldots, \rho_n \in \mathrm{P}_{\Sigma,\mathrm{T}}$.
- A repetition ρ^+ or possibly empty repetition ρ^* of $\rho \in \mathrm{P}_{\Sigma,\mathrm{T}}$.
- And-predicate $\&\rho$, with $\rho \in \mathrm{P}_{\Sigma,\mathrm{T}}$.
- Not-predicate $!\rho$, with $\rho \in \mathrm{P}_{\Sigma,\mathrm{T}}$.

The meanings of the and-predicate and not-predicate will be made clear below.

The semantics of a packrat grammar and the implementation of a parser are very simple. Given an input string s, an input position i and a parsing expression ρ, a packrat parser will attempt to parse s at i according to ρ as far as possible. The result is either "false" or a new position j. More precisely, the parser works as follows:

- If $\rho \in \mathrm{T}$, then we try to read ρ from the string s at position i.
- If $\rho \in \Sigma$, then we parse ρ_ρ at i.
- If $\rho = \rho_1 \cdots \rho_n$, then we parse $\rho_1, \ldots, \rho_n$ in sequence while updating i with the new returned positions. If one of the ρ_k fails, then so does ρ.
- If $\rho = \rho_1 / \cdots / \rho_n$, then we try to parse $\rho_1, \ldots, \rho_n$ at i in sequence. As soon as one of the ρ_k succeeds at i (and yields a position j), then so does ρ (and we return j).
- In case of repetitions, say $\rho = \pi^*$, we keep parsing π at i until failure, while updating i with the new positions.
- If $\rho = \&\pi$, then we try to parse ρ at i. In case of success, then we return the original i (i.e. we do not consume any input). In case of failure, we return "false".
- If $\rho = !\pi$, then we try to parse ρ at i. In case of success, we return "false". In case of failure, we return the original position i.

For efficiency reasons, parse results are systematically cached. Consequently, the the running time is always bounded by the length of string s times the number of parsing expressions occurring in the grammar.

5.2. Advantages and disadvantages of packrat grammars

The main difference between packrat grammars and LALR-1 grammars is due to the introduction of sequential disjunction. In the case of LALR-1 grammars, disjunctions may lead to ambiguous grammars, in which case the generation of a parser aborts with an error message. Packrat grammars are never ambiguous, since all disjunctions are always parsed in a "first fit" manner. Of course, this also implies that disjunction is non commutative. For instance, for terminal symbols x and y, the parsing expression $x\,y / x^*$ parses the string $x\,y$, unlike the expression $x^* / x\,y$. More precisely, parsing the string $x\,y$ at position 0 according to the expression $x^* / x\,y$ succeeds, but returns 1 as the end position.

In practice, writing a packrat grammar is usually easier than writing an LALR-1 grammar. First of all, thanks to sequential disjunctions, we do not need to spend any effort in making the grammars non ambiguous (of course, the disjunctions have to be ordered with more care). Secondly, there is no limitation in the size of the "look-ahead" for packrat grammars. In particular, there is no need for a separate "lexer".

Another important advantage is the ability to parse a string on the fly according to a given grammar. In contrast, LALR-1 grammars first apply formal language techniques in order to transform the grammar into an automaton. Although parsers based on such automata are usually an order of magnitude faster than packrat parsers, the generation of the automaton is a rather expensive precomputation. Consequently, packrat grammars are more suitable if we want to locally modify grammars inside documents.

On the negative side, packrat grammars do not support left recursion in a direct way. For instance, the grammar

$$\begin{aligned} X &\longrightarrow X + I / \\ & X - I / \\ & I \\ I &\longrightarrow (a/b/c/d/e/f/g/h/i/j/k/l/m/n/o/p/q/r/s/t/u/v/w/x/y/z)^+ \end{aligned}$$

leads to an infinite recursion when parsing the string $a+b$. Fortunately, this is not a very serious drawback, because most left-recursive rules are "directly" left-recursive in the sense that the left-recursion is internal to a single rule. Directly left-recursive grammars can easily be rewritten into right-recursive grammars. For instance, the above grammar can be rewritten as

$$\begin{aligned} X &\longrightarrow X^H (X^T)^* \\ X^H &\longrightarrow I \\ X^T &\longrightarrow +I / \\ & -I \\ I &\longrightarrow (a/b/c/d/e/f/g/h/i/j/k/l/m/n/o/p/q/r/s/t/u/v/w/x/y/z)^+ \end{aligned}$$

Furthermore, it is easy to avoid infinite loops as follows. Before parsing an expression ρ at the position i, we first put "false" in the cache table for the pair (ρ, i). Any recursive attempt to parse ρ at the same position will therefore fail.

A negative side effect of the elimination of lexers is that whitespace is no longer ignored. The treatment of whitespace can therefore be more problematic for packrat parsers. For various reasons, it is probably best to manually parse whitespace. For instance, every infix operator such as + is replaced by a non-terminal symbol Plus→Spc +Spc, which automatically "eats" the whitespace around the operator. An alternative solution, which provides less control, is to implement a special construct for removing all whitespace. A more elegant solution is to introduce a second packrat grammar with associated productions, whose only task is to eliminate whitespace, comments, etc.

In fact, TEXMACS documents are really trees. As we will see in section 5.4 below, we will serialize these trees into strings before applying a packrat parser. This serialization step can be used as an analogue for the lexer and an occasion to remove whitespace.

5.3. Implementation inside TEXMACS

An efficient generic parser for packrat grammars, which operators on strings of 32-bit integers, has been implemented in the C++ part of TEXMACS. Moreover, we provide a SCHEME interface, which makes it easy to define grammars and use the parser. For instance, the grammar for a simple pocket calculator would be specified as follows:

```
(define-language pocket-calculator
  (define Sum
    (Sum "+" Product)
    (Sum "-" Product)
    Product)

  (define Product
    (Product "*" Number)
    (Product "/" Number)
    Number)

  (define Number
    ((+ (- "0" "9")) (or ("." (+ (- "0" "9"))) ""))))
```

For top-level definitions of non-terminal symbols, we notice the presence of an implicit `or` for sequential disjunctions. The notation `(- "0" "9")` is a convenient abbreviation for the grammar 0/1/2/3/4/5/6/7/8/9.

TEXMACS also allows grammars to inherit from other grammars. This makes it for instance possible to put the counterpart of a lexer in a separate grammar:

```
(define-language pocket-calculator-lexer
  (define Space (* " "))
  (define Plus (Space "+" Space))
  (define Minus (Space "-" Space))
  (define Times (Space "*" Space))
  (define Over (Space "/" Space)))

(define-language pocket-calculator
  (inherit pocket-calculator-lexer)
  (define Sum
    (Sum Plus Product)
    (Sum Minus Product)
    Product)
  ...)
```

In a similar manner, common definitions can be shared between several grammars.

In traditional parser generators, such as YACC and BISON, it is also possible to specify productions for grammar rules. For the moment, this is not yet implemented in TEXMACS, but a straightforward adaptation of this idea to our context would be to specify productions for translation into SCHEME expressions. For instance, a specification of `Sum` with productions might be

```
(define Sum
  (-> (Sum Plus Product) ("(" 2 " " 1 " " 3 ")"))
  (-> (Sum Minus Product) ("(" 2 " " 1 " " 3 ")"))
  Product)
```

For the last case `Product`, we understand that the default production is identity. In the case of left-recursive grammars, we also have to adapt the productions, which can be achieved *via* the use of suitable lambda expressions [10].

Notice that the productions actually define a second packrat grammar. In principle, we might therefore translate into other languages than SCHEME. *A priori*, the rules could also be reverted, which provides a way to pretty print expressions (see also section 7.1). In this context, we observe one advantage of our manual control of whitespace: we might tweak the converter so as to pretty print `Space` as a single space. Reversion may also be used for syntax correction. For instance, the direct translation could accept the notation $+1$ and the reverse translation might insert an invisible wildcard. Finally, we might compose "packrat converters" so as to define new converters or separate certain tasks which are traditionally carried out by a lexer from the main parsing task. We intend to return to these possible extensions in a forthcoming paper.

Apart from the not yet implemented productions, there are various other kinds of annotations which have been implemented in TEXMACS. First of all, we may provide short descriptions of grammars using the `:synopsis` keyword:

```
(define-language pocket-calculator
  (:synopsis "grammar demo for a simple pocket calculator")
  ...)
```

Secondly, one may specify a physical or logical "color" for syntax highlighting:

```
(define Number
  (:highlight constant_number)
  ((+ (- "0" "9")) (or "" ("." (+ (- "0" "9"))))))
```

We may also associate types and typesetting properties to symbols:

```
(define And-symbol
  (:type infix)
  (:penalty 10)
  (:spacing default default)
  "<wedge>" "<curlywedge>")
```

These types were for instance used in our heuristics for syntax correction (see section 3.3). Finally, we implemented the additional properties `:operator` and `:selectable`, which will be detailed in section 7 on grammar assisted editing. More kinds of annotations will probably be introduced in the future.

5.4. Parsing trees

We recall that TEXMACS documents are internally represented by trees. Therefore, we have two options for applying packrat parsing techniques to TEXMACS documents:

- Generalizing packrat grammars to "tree grammars".
- Flattening trees as strings before parsing them.

For the moment, we have opted for the second solution.

More precisely, a TEXMACS document snippet is either a string leaf or a compound tree which consists of a string label and an arbitrary number of children, which are again trees. String leafs are represented in "enriched ASCII", using the convention that special characters can be represented by alphanumeric strings between the special symbols `<` and `>`. For instance, `<alpha>` represents α, whereas `<` and `>` are represented by `<less>` and `<gtr>`. Although this is currently not exactly the case, one may think of enriched ASCII strings as unicode strings.

A straightforward way to serialize a compound tree t with a given `label` and children $t_1, ..., t_n$ is as follows. Assuming that $t_1, ..., t_n$ are recursively serialized as `u1`, ..., `un`, we serialize t as `<\label>u1<|>...<|>un</>`. For instance, the expression

$$\alpha_k + \frac{x}{y}$$

would be serialized as

```
<alpha><\rsub>k</>+<\frac>x<|>y</>
```

An interesting possibility is to serialize special kinds of trees in alternative ways. For instance, whitespace may be ignored and a document with several paragraphs may be serialized by inserting newline characters instead of the special "characters" `<\document>`, `<|>` and `</>`. In this respect, the serialization step is somewhat analogous to the lexing step for traditional parsers. For user defined macros, it is also convenient to make the serialization customizable, as will be discussed in section 6.3 below.

Recall that our generic packrat parser operates on 32 bit integer strings. Internally, part of the 32 bit integer range is reserved for characters in our enriched alphabet, including the additional "characters" `<\label>`, `<|>` and `</>`. Another part is reserved for non terminal symbols. The remaining part is reserved for constructs of parsing expressions.

In order to simplify the task of writing Scheme grammars for structured documents, we introduced a few additional patterns for building parsing expressions:

- The Scheme expressions `:<label`, `:/` and `:>` stand for the strings `<\label>`, `<|>` and `</>`. Furthermore, `:<` stands for any character of the form `:<label`.
- `:any`, for any well formed TeXmacs tree.
- `:args` as a shorthand for `(:arg (* (:/ :arg)))`.
- `:leaf`, for any well formed leaf.
- `:char`, for any enriched ASCII character, such as `a` or `<alpha>`.

For instance, the `Product` rule in our pocket calculator might be replaced by

```
(define Product
  (Product Times Number)
  (Product Over Number)
  Fraction)

(define Fraction
  (:<frac Sum :/ Sum :>)
  Number)
```

6. Towards a universal grammar for mathematics

6.1. Major design issues

Having implemented a good mechanism for the construction of parsers, we are confronted to the next question: which grammar should we use for general purpose mathematical formulas? In fact, we first have to decide whether we should provide a universal well-designed grammar or make it easy for users to define their own customized grammars.

One important argument in favour of the first option is that a standard well-designed grammar makes communication either, since the reader or coauthors do not have to learn different notational conventions for each document. In the same way as presentation MATHML attempts to capture the presentation aspects of any (non exotic) mathematical document, we hope that a well-designed grammar could capture the syntactical semantics of mathematical formulas.

Another argument in favour of a universal grammar is the fact that the design of new grammars might not be so easy for non experts. Consequently, there is a high risk that customized grammars will be ill-designed and lead to errors which are completely unintelligible for other users. Stated differently: if we choose the second option, then customizations should be made really easy. For instance, we might only allow users to introduce new operators of standard types (prefix, postfix, infix) with a given priority.

The main argument in favour of customizable grammars is that we might not be able to anticipate the kind of notation an author wishes to use, so we prefer to keep a door open for customizing the default settings. External systems, such as proof assistants or computer algebra systems, may also have built-in ways to define new notations, which we may wish to reflect inside TEXMACS.

At any rate, before adding facilities for customization, it is an interesting challenge to design a grammar which recognizes as much of the traditional mathematical notation as possible. For the moment, we have concentrated ourselves on this task, while validating the proposed grammars on a wide variety of existing documents. Besides, as we will see in section 6.3, although we do not provide easy ways for the user to customize the grammar, we do allow users to customize the tree serialization procedure. This is both a very natural extension of the existing TEXMACS macro system and probably sufficient for most situations when users need customized notations.

Having opted for a universal grammar, a second major design issue concerns potential ambiguities. Consider the expression $a \vee b \wedge c$. In analogy with $a + b\,c$, this expression is traditionally parsed as $a \vee (b \wedge c)$. However, this may be non obvious for some readers. Should we punish authors who enter such expressions, or re-lax and leave it as a task to the reader to look up the notational conventions?

In other words, since language of a matter of communication, we have to decide whether the burden of disambiguating formulas should rather incomb to the author or to the reader. For an authoring tool such as TEXMACS, it is natural to privilege the author in this respect. We also notice that the reader might have access to the document in electronic or TEXMACS form. In that case, we will discuss in section 7 how the reading tool may incorporate facilities for providing details on the notational conventions to the reader.

6.2. Overview of the current TEXMACS mathematical grammar

The current TEXMACS grammar for mathematics is subdivided into three subgrammars:

- A lowest level grammar `std-symbols` which mainly contains all supported mathematical symbols, grouped by category.
- An intermediate grammar `std-math-operators` which mainly describes the mathematical operators occurring in the grammar.
- The actual grammar `std-math-grammar`.

6.2.1. The symbol grammar `std-symbols`

The purpose of the lowest level grammar is to group all supported mathematical symbols by category and describe the purpose and typesetting properties of each category. A typical non terminal symbol defined in this grammar is the following:

```
(define And-symbol
  (:type infix)
  (:penalty 10)
  (:spacing default default)
  "<wedge>" "<curlywedge>")
```

The category contains two actual symbols `<wedge>` ($\wedge$) and `<curlywedge>` ($\curlywedge$). The `:type` annotation specifies the purpose of the symbol, i.e. that it will be used as an infix operator. The `:penalty` and `:spacing` annotations specify a penalty for the symbols during line-breaking and the usual spacing around the symbols. For some symbols and operators, subscripts and superscripts are placed below resp. above:

```
(define N-ary-operator-symbol
  (:type n-ary)
  (:penalty panic)
  (:spacing none default)
  (:limits display)
  "inf" "lim" "liminf" "limsup" "max" "min" "sup")
```

The symbol grammar also describes those TEXMACS tags which play a special role in the higher level grammars:

```
(define Reserved-symbol
  :<frac :<sqrt :<wide ...)
```

6.2.2. The operator grammar `std-math-operators`

The purpose of `std-math-operators` is to describe all mathematical operators which will be used in the top-level grammar `std-math-grammar`. Roughly speaking, the operators correspond to the symbols defined in `std-symbols`, with this difference that they may be "decorated", typically by primes or scripts. For instance, the counterpart of `And-symbol` is given by

```
(define And-infix
  (:operator)
  (And-infix Post)
  (Pre And-infix)
  And-symbol)
```

where `Pre` and `Post` are given by

```
(define Pre
  (:operator)
  (:<lsub Script :>)
  (:<lsup Script :>)
  (:<lprime (* Prime-symbol) :>))

(define Post
  (:operator)
  (:<rsub Script :>)
  (:<rsup Script :>)
  (:<rprime (* Prime-symbol) :>))
```

The `:operator` annotation states that `And-infix` should be considered as an operator during selections and other structured editing operations (see section 7).

6.2.3. The main grammar `std-math-grammar`

Roughly speaking, the main grammar for mathematics `std-math-grammar` defines the following kinds of non terminal symbols:

- The main symbol `Expression` for mathematical expressions.
- A relaxed variant `Relaxed-expression` of `Expression`, for which operators + and formulas such as >0 are valid expressions. Relaxed expressions typically occur inside scripts: $a = a^+ - a^-$, $x \in \mathbb{R}^{>0}$, etc.
- The symbol `Main` for the "public interface", which is a relaxed expression with possible trailing ponctuation symbols and whitespace.
- Symbols corresponding to each of the operator types, ordered by priority.
- Some special mathematical notations, such as quantifier notation.
- Prefix-expressions postfix-expressions and radicals.

For instance, the "arithmetic part" of the grammar is given by

```
(define Sum
  (Sum Plus-infix Product)
  (Sum Minus-infix Product)
  Sum-prefix)

(define Sum-prefix
  (Plus-prefix Sum-prefix)
  (Minus-prefix Sum-prefix)
  Product)

(define Product
  (Product Times-infix Power)
  (Product Over-infix Power)
  Power)
```

The grammatical specification of relations $=$, $\neq$, $\leqslant$, $\subsetneq$, etc. is similar to the specification of infix operators, but relations carry the following special semantics:

$$a_0 \, \mathcal{R}_1 \, a_1 \, \mathcal{R}_2 \cdots a_{n-1} \, \mathcal{R}_n \, a_n \quad \Longleftrightarrow \quad a_0 \, \mathcal{R}_1 \, a_1 \wedge \cdots \wedge a_{n-1} \, \mathcal{R}_n \, a_n.$$

Quantified expressions are defined by

```
(define Quantified
  (Quantifier-prefixes Ponctuation-infix Quantified)
  (Quantifier-fenced Quantified)
  (Quantifier-fenced Space-infix Quantified)
  Implication)
```

where

```
(define Quantifier-prefix
  (Quantifier-prefix-symbol Relation))

(define Quantifier-prefixes
  (+ Quantifier-prefix))

(define Quantifier-fenced
  (Open Quantifier-prefixes Close)
  (:<around :any :/ Quantifier-prefixes :/ :any :>))
```

Hence, the most common forms of quantifier notation are all supported:

$$\forall x \in A, \forall y \in B, x \leqslant y$$
$$\forall x \in A \forall y \in B \colon x \leqslant y$$
$$(\forall x \in A)(\exists y \in B)\varphi(x) = \psi(y).$$

In some cases, such as the formula,

$$\underset{\varepsilon>0}{\forall} \underset{\delta>0}{\exists} |x-y| < \delta \Rightarrow |f(x)-f(y)| < \varepsilon,$$

it is also natural to consider quantifiers $\forall$ and $\exists$ as big operators. Unfortunately these big symbols are not supported by UNICODE.

Big operators are treated as special kinds of prefix operators, in which the argument has an appropriate binding strength:

```
(define Prefixed
  ...
  (Big-sum Sum-prefix)
  (Big-product Power)
  (Prefix-prefix Prefixed)
  (Pre Prefixed)
  (Postfixed Space-infix Prefixed)
  Postfixed)
```

In the treatment of prefixed and postfixed expressions

```
(define Postfixed
  (Postfixed Postfix-postfix)
  (Postfixed Post)
  (Postfixed Fenced)
  (Postfixed Restrict)
  Radical)
```

with

```
(define Fenced
  (Open Close)
  (Open Expressions Close)
  (:<around :any :/ (* Post) (* Pre) :/ :any :>)
  (:<around :any :/ (* Post) Expressions (* Pre) :/ :any :>))
```

we consider the two forms $\sin x$ and $f(x)$ of function application as special kinds of prefixed and postfixed expressions.

Radical expressions are either identifiers, numbers, special symbols, fenced expressions, or special TEXMACS markup:

```
(define Radical
  Fenced
  Identifier
  Number
  (:<sqrt Expression :>)
  (:<frac Expression :/ Expression :>)
  (:<wide Expression :/ :args :>)
  ((except :< Reserved-symbol) :args :>)
  ...)
```

6.3. Customization via TEXMACS macros

Although we have argued that the design or modification of packrat grammars may be difficult for the average user, it still remains desirable to provide a way to introduce new kinds of notation.

One traditional approach is to allow for the introduction of new mathematical operators of a limited number of types (prefix, postfix, infix, bracket, etc.), but with arbitrary binding forces. Such operator grammars are still easy to unterstand and modify for most users. Instead of using numerical binding forces, a nice variant of this idea is to allow for arbitrary partial orders specified by rules such as `Product > Sum`.

Remark 2. As suggested in a discussion with M. KOHLHASE, one might even allow for ambiguous grammars, in which case the user would be invited to manually disambiguate expressions, whenever appropriate. This can be achieved by introducing a non sequential "or" primitive into the packrat grammar building constructs. Alternatively, one could make the grammar more strict (e.g. disallow for expressions such as $a \wedge b \vee c$) and provide hints on how to release the grammar. In practice, these complications might not be worth it, since the current focus (see section 7.3) already gives a lot of visual feedback to the author on how formulas are interpreted.

Inside TEXMACS, a more straightforward alternative approach is to capitalize on the existing macro system. On the one hand (and before the recent introduction of semantic editing facilities), this system was already used for the introduction of new notations and abbreviations. On the other hand, the system can be extended with a user friendly "behaves as" construct, after which it should be powerful enough for the definition of most practically useful notations.

We recall that TEXMACS macros behave in a quite similar way as LATEX macros, except that clean names can be used for the arguments. For instance, the macro

$$\langle \mathsf{assign} | \mathit{cbrt} | \langle \mathsf{macro} | x | \langle \mathsf{sqrt} | x | 3 \rangle \rangle \rangle$$

may be used for defining a macro for producing cubic roots $\sqrt[3]{x}$. Our serialization procedure from section 5.3 has been designed in such a way that all user defined macros are simply expanded by default. In particular, from the semantic point of view, there is no noticeable difference between the newly introduced cubic roots and the built-in n-th roots with $n = 3$, except that the 3 in the cubic root is not editable.

The next idea is to allow users to override the default serialization of a macro. For instance, assume that we defined a macro

```
⟨assign|twopii|⟨macro|2*<mathpi>*<mathi>⟩⟩
```

By default, the line $1/2\,\pi\,\mathrm{i}\oint f(z)\,\mathrm{d}z$ is interpreted as $(1/2)\,\pi\,\mathrm{i}\oint f(z)\,\mathrm{d}z$. By specifying a serialization macro for twopii

```
⟨drd-props|twopii|syntax|⟨macro|(2*<mathpi>*<mathi>)⟩⟩
```

and after entering $2\,\pi\,\mathrm{i}$ using our macro, the formula $1/2\,\pi\,\mathrm{i}\oint f(z)\,\mathrm{d}z$ will be interpreted in the intended way $1/(2\,\pi\,\mathrm{i})\oint f(z)\,\mathrm{d}z$.

This very simple idea is both very user friendly (we basically specify how the macro behaves from a semantic point of view) and quite powerful. For instance, if we define a $+$ operator with the binding force of the logical or $\vee$, we simply invent a suitable name, such as "logical-plus", use a $+$ for the rendering and a $\vee$ for the serialization. Of course, this scheme is limited to reduce to those operators which are already present in our predefined universal grammar. Nevertheless, after some years of maturing, we expect that the universal grammar will be sufficiently rich for covering most notational patterns used in practice.

As a side note, it is interesting to observe once more that the serialization procedure plays the role of a "structured" lexer, except that we are rather destroying or transforming tree structure in a customizable way than detecting lexical structure inside strings.

6.4. Experimental results

In section 3.4, we have described the performance of the TEXMACS syntax corrector on several large sample documents. In this section, we will analyze in more detail the causes of the remaining "errors". We will also investigate up to which extent "correct" formulas are interpreted in the way the author probably had in mind.

Throughout the section, one should bear in mind that our test documents were not written using the newly incorporated semantic editing features. On the one hand, our analysis will therefore put the finger on some of the more involved difficulties if we want to make a visually authored document semantically correct. On the other hand, we will get indications on what can be further improved in the syntax corrector and our universal grammar. Of course, our study is limited in the sense that our document base is relatively small. Nevertheless, we think that the most general trends can already be detected.

6.4.1. Analysis of the remaining errors

We have analyzed in detail the remaining errors in BPR_1, BPR_2, CH_3 and HAB, which can be classified according to several criteria, such as severeness, eligibility for automatic correction, and the extent to which the author was to blame for the error. The results of our analysis have been summarized in table 4, where we have distinguished the following few main categories for sorting out the errors:

Eligible for automatic correction. Modulo additional efforts, our syntax corrector might be further improved, so as to take into account some additional common notational patterns. Some of these patterns are the following:

Meaningful whitespace. Sometimes, large whitespaces are used in a semantically meaningful way. For instance, in BPR_1 and BPR_2, the notation

$$\exists x \quad P(x) \tag{3}$$

is frequently used as a substitute for $\exists x,\, P(x)$. Although potentially correctable, it should be noticed that the (currently supported) notation $(\exists x)P(x)$ is also used in BPR. Hence, we detected "at least" an inconsistency in the notational system of BPR.

Non-annotated operators. We have already mentioned the fact that the universal grammar should be sufficiently friendly so as to recognize the formula $\mathbb{R}^+$, where the operator + occurs in a script. More generally, in the formula $\mathrm{Qu}_i \in \{\forall,\, \exists\}$ from BPR_1, the operators $\forall$ and $\exists$ are sometimes used "as such". Although it does not seem a good idea to extend the grammar indefinitely so as to incorporate such notations, we might extend the corrector so as to detect these cases and automatically insert "invisible operator" annotations.

Manual hyphenation in tables. In tables or equation arrays, large formulas are sometimes manually split across cells. One example from BPR_1 is given by

$$\begin{aligned} \mathrm{posgcd}(\emptyset) &= \{(0, 1 \neq 0)\}, \\ \mathrm{posgcd}(\mathcal{P} \cup \{P\}) &= \{(\mathrm{Pol}(p(L)), \mathcal{C} \wedge \mathcal{C}_L) \mid (Q, \mathcal{C}) \in \mathrm{posgcd}(\mathcal{P}) \\ &\qquad \text{and } L \text{ leaf of } \mathrm{TRems}(P, Q)\}. \end{aligned} \tag{4}$$

In this example, the curly opening bracket in the right-hand cell of the second line is only closed on the third line. Another (truncated) example from BPR_2 is

$$\begin{gathered} (a < 0 \wedge s > 0) \\ \vee\, (a < 0 \wedge s < 0 \wedge \delta < 0) \\ \vee\, (a > 0 \wedge s < 0 \wedge \delta < 0) \\ \cdots \end{gathered}$$

The problem here is that formulas of the form $\vee x$ are incorrect.

The first example raises the most severe problem, which might be solved by automatically trying to concatenate certain cells of tables before launching the parser. The second example can be solved more simply by relaxing the grammar for formulas inside cells. This was already done for formulas of the form $x +$.

Unsuccessful automatic correction. Since the process of syntax correction is very heuristic by nature, it sometimes fails. Failure typically occurs when one formula contains at least two difficulties, so that none of the corresponding heuristic criteria holds. The following kinds of failures occurred in our test documents:

Uncorrected invisible markup. This problem is by far the most frequent one. Especially in CH_3 the job is made hard for the corrector by the author's habit to replace multiplications by function applications. As a consequence, many symbols which would usually be detected as being variables might also be functions; this confusion pushes the corrector not to touch any of such formulas. Indeed, we consider it better to leave an incorrect formula than to make a wrong "correction".

Non recognized invisible tag simplifications. Although the juxtaposition of two formulas `{$a+$}{$b$}` is simplified into `$a+b$`, such simplifications can be confused by the presence of whitespace, as in `$a+$ $b$`.

Non upgraded brackets. One example from BPR_2 is the formula

$$\mathcal{B} = \left\{a + i\, b \mid |b| \leqslant -\sqrt{3}\, a\right\},$$

in which the absolute value $|b|$ was not recognized as such. We already mentioned another similar example from COL before: $(X', |\cdot|)$.

Informal visual notation. There are various situations in which the user resorts to informal, purely visual notations with no obvious semantics. Sometimes, the user is just lazy or sloppy, but sometimes it is hard to "do things right". Here are some typical examples from our test documents:

Non standard operator precedence. Sometimes, authors may use apparently standard notation in a non standard way, by making non conventional assumptions on operator precedence. Consider for instance the following fragment from BPR_1:

> Given a formula $\Theta(Y) = (\exists X)\ \mathcal{B}(X, Y)$,where $\mathcal{B}$ is...

In this case, the binding force of $=$ is higher than the binding force of $\exists$, which makes the formula incorrect. Such formulas are on the limit of being typos: it would be better to insert (possibly invisible) brackets. This kind of error is rare though, since non standard operator precedences usually do not prevent formulas from being parsed, but rather result in non intended interpretations of the formulas.

Abusive structured markup for visual twiddling. Consider the formula

$$\prod_{\substack{i<j,\,k<\ell \\ (i,j)<(k,\ell)}} (\alpha_{i,j,k,\ell} + Z\,\beta_{i,j,k,\ell})^2$$

from BPR_1. The authors intended to typeset the subscript of the big product using an extra small font size, and inserted a double subscript to achieve this visual effect. In $\mathrm{TEX}_{\mathrm{MACS}}$, they should have used the Format→Index level→Script script size menu entry instead.

Using text in mathematical formulas. It often occurs that one needs to insert some accompanying text inside formulas, as in the formula

$$a_n \to a \iff \forall \varepsilon, \exists N\ s.t. \text{for}\, n > N, |a_n - a| \leq \varepsilon$$

occurring in COL_3. The problem here is that the text "s.t. for" was entered in math mode, so it will not be ignored by the parser. Other similar examples from BPR are

> There are indeed 4 real roots: $1, -1, 2, \text{and} -2$.
> It is obvious since the $(i, j) - \text{th}$ entry of $\mathrm{Mat}(A, \Sigma)$ is $\sigma_j^{\alpha_i}$.

In each of these examples, we notice that the obtained visual result is actually incorrect: the errors are easily committed by users who do not pay attention to the font slant, the shapes of characters and the spacing between words. For instance, the corrected versions of the last examples read

> There are indeed 4 real roots: $1, -1, 2$, and -2.
> It is obvious since the (i, j)-th entry of $\mathrm{Mat}(A, \Sigma)$ is $\sigma_j^{\alpha_i}$.

Notice also that correctly entered text inside mathematical formulas is ignored by the parser. Consequently, the use of meaningful text inside formulas may raise the same problems as the before-mentioned use of meaningful whitespace.

Using mathematics as a replacement for text. In BPR, there is a general tendency of typesetting implications between substatements in math-mode, as in the example "$a) \Rightarrow b)$". Clearly, this is an abuse of notation, although it is true that convenient ways to achieve the same result in text mode do not necessarily exist.

Visual twiddling with no obvious alternative. In some less frequent cases, it is convenient to use certain kinds of suggestive notations with *ad hoc* semantics. Three examples from BPR_1, COL_3 and HAB respectively are

1. The signs of the polynomials in the Sturm sequence are $+-+-+$
2. No other code word is of the form $0.z_1 \ldots z_{l(x)} * \ldots$
3. $g = e^{\sqrt{x} + e^{\sqrt{\log x} + e^{\sqrt{\log \log x} + \iddots + \log \log \log x} + \log \log x} + \log x}$

Clearly, there is no hope to conceive a "universal grammar" which automatically recognizes this kind of *ad hoc* notations. The only reasonable way to incorporate such constructs is to provide a few basic annotation tags which allow the user to manually specify the semantics.

Genuine typos. All analyzed texts contained at least one syntax error for which the author has no excuse. Roughly speaking, half of them were missing brackets. Another typical typo in BPR which was detected by our grammar is

$$R(P, E, x, Y) = \varepsilon^{-\beta} P(\varepsilon^{\xi}(x + Y)) = \bar{b}_0 + \bar{b}_1 Y + \cdots \bar{b}_p Y^p.$$

Indeed, there is a + missing at the end of the formula.

Other errors. The document COL_3 contained one other strange error, which is very specific to TEXMACS: the personal style package of the author contained a semantically incorrect macro, which induced an error each time this macro was used.

Document	BPR_1	BPR_2	COL_3	HAB
Total remaining errors	35	53	37	6
Eligible for automatic correction	25	13	0	0
Unsuccessful automatic correction	6	2	16	2
Informal visual notation	2	31	19	3
Genuine typos	2	7	1	1
Other errors	0	0	1	0

Table 4. Sources of the remaining errors in our sample documents, sorted out by category.

6.4.2. Analysis of misinterpretations

The above analysis of the remaining errors takes advantage of the possibility to highlight all syntax errors in a TEXMACS document. Unfortunately, there is no automatic way to decide whether a correct formula is interpreted in the way intended by the author. Therefore, it requires far more work to give a complete qualitative and quantitative analysis of the "parse results". Nevertheless, it is possible to manually check the interpretations of the formulas, by careful examination of the semantic focus (see 7.3 below for a definition and more explanations on this concept) at the accessible cursor positions in a formula.

We have carefully examined the documents BPR_1, COL_3, HAB_1 and HAB_5 in this way. The number of misinterpretations in BPR_1 and HAB are of the same order of magnitude as the number of remaining errors. The number of misinterpretations in COL_3 is very high, due to the fact that most multiplications were replaced by function applications. This paper also makes quite extensive use of semantically confusing text inside formulas. When ignoring these two major sources of misinterpretation, the universal grammar correctly parses most of the formulas.

The results of our analysis have been summarized in table 5. Let us now describe in more details the various sources of misinterpretation:

Incorrect invisible operators. Again, the multiply/apply ambiguity constitutes a major source of confusion. Several kinds of misinterpretation are frequent:

1. Replacing a multiplication by a function application or *vice versa* rarely modifies the parsability of a formula, but it completely alters its sense. For instance $a\,b\,c$ might either be interpreted as $a \cdot b \cdot c$ or $a(b(c))$. Fortunately, this kind of misinterpretations only occurs if the author explicitly entered the wrong operators.

2. A more subtle problem occurs when the author forgets to enter certain multiplications (or if the syntax corrector failed to insert them). One example is the subformula $b(12\ c\ +\ a^2)$, occurring in BPR_1. In this formula, b is interpreted as a function applied to $12\,c + a^2$.

3. An even more subtle variant of the above misinterpretation occurs at the end of HAB_5 in the formula $\sim \mathsf{R}(n)\ s$. In order to avoid trailing spaces, the $\sim$ was explicitly marked to be an operator and an explicit space was inserted manually after it. Consequently, the formula is interpreted as $\sim(\mathsf{R}(n)) \cdot s$ instead of the intended $\sim\ (\mathsf{R}(n)\ \cdot\ s)$. In recent versions of $\mathrm{TEX}_{\mathrm{MACS}}$ the precaution to mark $\sim$ as an operator has become superfluous.

Informal list notation. The informal, but widely accepted notation $a, b \in X$ is used quite often in order to signify $a \in X \wedge b \in X$. However, the "natural" interpretation of this formula is $a, (b \in X)$. This is an interesting case where it might be useful to insert an additional rule into the grammar (the general form being $a_1, \ldots, a_n\, \mathcal{R}\, b$, for a relation infix $\mathcal{R}$). This rule should only apply for top-level formulas and not for subformulas (e.g. $f(a, b \in X)$). Furthermore, it should be tested before the default interpretation $a_1, \ldots, (a_n\, \mathcal{R}\, b)$. Here we point out that this is easy to do for packrat grammars, but not for more conventional LALR-1 grammars, where such a test would introduce various kinds of ambiguities.

In BPR_1, one may also find variant of the above notation: "we define the signed remainder sequence of P and Q,

$$\mathrm{SRemS}(P, Q) = \mathrm{SRemS}_0(P, Q), \mathrm{SRemS}_1(P, Q), \ldots, \mathrm{SRemS}_k(P, Q)\text{''}.$$

In this case, it would have been appropriate to insert invisible brackets around the sequence. Yet another blend of list notation occurs frequently in [31]:

> The subsets of affine spaces $\mathbb{R}^n$ for $n = 0, 1, 2, \ldots$ that are ...

Here one might have used the more verbose notation $n \in \{0,\ 1,\ 2,\ \ldots\}$ instead. However, it seems preferable to investigate whether this example justifies another exceptional rule in the universal grammar. Let us finally consider

$$H(Y_i | Y^{i-1}, X^n),$$

which occurs as a subformula in COL_3. Currently, the separators | and , have the same precedence in our universal grammar. In view of the above example, we might want to give , a higher binding force.

Notice that "informal list notation" is a special instance of a potentially more general situation in which certain precedence rules need to be overridden under "certain circumstances".

Non marked text inside formulas. If informal text is not being marked as such inside formulas, then it gives rise to a wide variety of errors and potential misinterpretations. Consider for instance the formula

$$p(\hat{x}\,|x) = 1 \text{ if } \hat{x} = 0$$

from COL_3. Since the word "if" was entered in math mode, the formula is parsed as

$$p(\hat{x}\,|x) = 1(\mathrm{if}(\hat{x})) = 0.$$

Non marked formulas inside text. *Vice versa*, it sometimes happens that a mathematical formula (or part of it) was entered in text mode. Indeed, consider the following fragment from BPR_1:

> In the above, each SRemS_i(P,Q) is the negative of the remainder in the euclidean division of SRemS_{i-2}(P,Q) by $\mathrm{SRemS}_{i-1}(P,\ Q)$ for $2 \le i \le k+1$, ...

At two occasions, the subformula (P, Q) was entered in text mode. This might have been noticed by the authors through the use of an upright font shape.

Misinterpretation of meaningful whitespace. At several places in HAB_1 we have used the following notation:

$$P(f) = 0 \qquad (f \prec \mathfrak{V}).$$

Contrary to (3), this formula can be parsed, but its meaning

$$P(f) = [0(f \prec \mathfrak{V})]$$

is not what the author intended. This example provides yet more rationale for the design of a good criterion for when a large whitespace really stands for a comma.

Miscellaneous misinterpretations. One final misinterpreted formula from the document COL_3 is

$$\max_{p,q,r} -1 - p\log p - q\log q - r\log r - (1-p-q-r)\log(1-p-q-r).$$

Indeed, the subformula $\max_{p,q,r}$ is interpreted as a summand instead of an n-ary operator.

Despite the existence of the above misinterpretations, we have to conclude that our universal grammar is surprisingly good at interpreting formulas in the intended way. On the one hand (and contrary to the remaining errors from the previous subsection), most misinterpretations tend to fall in a small number of categories, which are relatively well understood. A large number of misinterpretations can probably be eliminated by further tweaking of the universal grammar and the syntax corrector. Most of the remaining misinterpretation can easily be eliminated by the author, if two basic rules are being followed:

1. Correct manual resolution of the multiply/apply ambiguity.
2. Correct annotations of all exceptional non mathematical markup inside formulas, such as text and meaningful whitespace.

However, $\mathrm{TEX}_{\mathrm{MACS}}$ can probably do a better job at inciting authors to follow these rules. We will come back to this point in section 8.

Of course, our conclusions are drawn on the basis of an ever shrinking number of documents. Rapid investigation of a larger number of documents shows that certain areas, such as λ-calculus or quantum physics, use various specialized notations. Nevertheless, instead of designing a completely new grammar, it seems sufficient to add a few rules to the universal grammar, which can be further customized using macros in style files dedicated to such areas. For instance, certain notational extensions involving $+$ and $-$ signs may both be useful in real algebraic geometry and quantum physics. Such notations might be activated via certain macros and keybindings defined in dedicated style packages.

Document	BPR_1	COL_3	HAB_1	HAB_5
Invisible operator confusion	some	many		some
Informal list notation	several		several	some
Non marked text inside formulas	several	many		
Non marked formulas inside text	some			
Misinterpreted meaningful whitespace			several	
Miscellaneous misinterpretations		some		

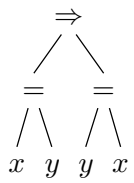
Table 5. Manual determination of common sources of misinterpretation.

7. GRAMMAR ASSISTED EDITING

7.1. On syntactic correctness

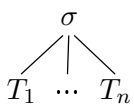
Traditionally, a formal introduction of mathematical formulas starts with the selection of a finite or countable alphabet Σ. The alphabet is partitioned into a subset of logical symbols $\forall, \exists, \wedge, \vee, \ldots$, a subset of relation symbols $=, \leqslant, \ldots$, a subset of function symbols $+$, $-, \cdot, \ldots$ and a subset of constant symbols. Each non-constant symbol $\sigma \in \Sigma$ also carries a set $N_\sigma \in \mathbb{N}$ of possible arities. An *abstract syntax tree* a finite Σ-labeled tree, such that the leafs are labeled by constant symbols and such that the number of children of each inner σ-labeled node is in N_σ.

From now on, a *syntactically correct formula* is understood to be a formula which *corresponds* to an abstract syntax tree. For instance, the abstract syntax tree



corresponds to the formula $x = y \Rightarrow y = x$. For concrete document formats, our definition of syntactic correctness assumes a way to make this correspondance more precise.

The simplest content oriented formats are those for which the correspondence is straightforward. For instance, in SCHEME, a constant symbol would be represented by itself and a compound tree



by `(`σ `T1` ... `Tn)`, where `T1`, ..., `Tn` are the representations of T_1, ..., T_n. Content MATHML is another format in which the representation of abstract syntax trees is rather straightforward.

Of course, content oriented formats usually do not specify how to render formulas in a human readable form. This makes them less convenient for the purpose of wysiwyg editing. Instead of selecting a format for which the correspondence between abstract syntax trees and syntactically correct formulas is straightforward, we therefore propose to focus on convenient ways to specify non trivial correspondances, using the theory developed in sections 5 and 6. In principle, as soon as a reliable correspondance between content and presentation markup is established, the precise internal document format is not that important: at any stage, a reliable correspondance would mean that we can switch at will between the abstract syntax tree and a more visual representation.

Unfortunately, a perfectly bijective correspondence is hard to achieve. Technically speaking, this would require an invertible grammar with productions, which cannot only parse and transform presentation markup into content markup, but also pretty print content markup as presentation markup. Obviously, this is impossible as soon as certain formulas, such as x and (x), are parsed in the same way. Apart from this trivial obstacle, presentation markup usually also allows for various annotations with undefined mathematical semantics. For instance: what would be the abstract syntax tree for a blinking formula, or a formula in which the positions of some subformulas were adjusted so as to "look nice"?

These examples show that presentation markup is usually richer than content markup. Consequently, we have opted for presentation markup as our native format and concentrated ourselves on the one-way correspondence from presentation markup to content markup, using a suitable packrat grammar. In fact, for the semantic editing purposes in the remainder of this section, the grammar does not even need to admit production for the generation of the actual abstract syntax tree. Of course, productions should be implemented if we want actual conversions into MATHML or SCHEME.

As we will detail below, it should also be noticed that we do not only need a correspondance between presentation and content markup, but also between several editor-related derivatives, such as cursor positions and selections. We finally have to decide what kind of action to undertake in cases of error. For instance, do we accept documents to be temporarily incorrect, or do we require parsability at all stages?

Remark 3. It remains an interesting challenge to design grammars which are "as invertible" as possible. For instance, which grammar-based transformations $\Phi\colon L_1 \to L_2$ admit a "Galois"-inverse $\Psi\colon L_2 \to L_1$ with the property that $\Phi = \Phi \circ \Psi \circ \Phi$ and $\Psi = \Psi \circ \Phi \circ \Psi$? Also, how to define a mathematical semantics for annotated formulas? In particular, how would they behave during computations? For instance, we might have a data type "annotated number" which joins the annotation for any operation. When annotating a subset of the input data, this semantics might be used to visualize the part of the output whose computation depends on this annotated subset.

7.2. Maintaining correctness

In order to benefit from semantic editing facilities, we first have to ensure that the formulas in our document are correct. For similar reasons as in section 4.1, we have two main options:

1. Enforce correctness at every stage.
2. Incite the user to type correct formulas by providing suitable visual hints.

It depends very much on personal taste of the user which option should be preferred. For the moment, we have opted for mere incitation in our TEXMACS implementation, but we will now address the issues which arise for both points of view.

If we decide to enforce correctness at every stage, then we are immediately confronted to following problem: what happens if we type + in the example

$$x|, \tag{5}$$

while taking into account that the formula $x+$ is incorrect? Obviously, we will have to introduce some kind of markup which will replace $x+$ by a correct formula, such as

$$x+\boxed{?}|.$$

The automatically inserted dummy box $\boxed{?}$ is parsed as an ordinary variable, but *transient* in the sense that it will be removed as soon as we type something new at its position. We will call such boxes *slots* in what follows. Several existing mathematical editors [29, 21] use slots in order to keep formulas correct and indicate missing subformulas.

Of course, we need an automated way to decide when to insert and remove slots. This can be done by writing wrappers around the event handlers. On any document modification event, the following actions should be undertaken:

1. Pass the event to the usual, non-wrapped handler.
2. For each modified formula, apply a post-correction routine, which inserts or removes slots when appropriate.
3. If the post-correction routine does not succeed to make the formula correct, then undo the operation.

These wrappers have to be designed with care: for instance, the usual undo handling has to be done outside the wrapper, in order to keep the tab-variant system working (see section 4.3).

The post-correction routine should start by trying to make local changes at the old and the new cursor positions. For instance, if a slot is still present around the old cursor position, then attempt to remove it; after that, if the formula is not correct, attempt to make the formula correct by inserting a slot at the new cursor position. Only when the local corrections failed to make the entire formula correct, we launch a more global and expensive correction routine (which corrects outwards, starting at the cursor position). Such more expensive correction routines might for instance insert slots in the numerator and denominator of a newly created fraction.

Remark 4. For editors which are designed from scratch, an alternative point of view is to design all editing operations to correctly handle slots at the first place. For instance, when creating a fraction, one might automatically insert the slots for the numerator and the denominator. However, this strategy puts additional burden on the programmer for the implementation of all fundamental editing operations. In order to *enforce* correctness, our decoupled approach therefore seems more robust and we recommend the implementation of an independent syntax checker/corrector at least as a complement to any other approach.

Assume now that we no longer insist on correctness at all stages. In that case, an interesting question is how to provide maximal correctness *without* making *a posteriori* modifications such as the insertion and removal of slots. Following this requirement, the idea in example (5) would be to consider the formula

$$x+|$$

as being correct, *provided that* the cursor is positioned after the +. In other words, if the current formula is incorrect, then we check whether the current formula with a dummy variable substituted for the cursor is correct. If this check again fails, then we notify the user that he made a mistake, for instance by highlighting the formula in red.

The second approach usually requires a grammar which is more tolerant towards empty subformulas in special constructs such as fractions, matrices, etc. For instance, it is reasonable to consider the empty fraction $\cdot$ as being correct. Again, it really comes down to personal taste whether one is willing to accept this: the fraction $\frac{\boxed{?}}{\boxed{?}}$ definitely makes it clear that some information is missing, but the document also gets clobbered, at least momentaneously by dummy boxes, which also give rise to "flickering" when editing formulas. Moreover, in the case of a matrix

$$\begin{pmatrix} a_{11} & & \\ & \ddots & \\ & & a_{nn} \end{pmatrix},$$

it is customary to leave certain entries blank. Semantically speaking, such blank entries correspond to invisible zeros or ellipses.

Another thing should be noticed at this point. Recall that we defined correctness in terms of our ability to maintain a correspondance between our presentation markup and the intended abstract syntax tree. Another view on the above approach is to consider it as a way to build some degree of auto-correction right into the correspondance. For instance, the formula $x+|$ can be considered to be "correct", because we still have a way to *make* it correct (e.g. by inserting a slot at the cursor position). The correction might even involve the history of the formula (so that the formula remains correct when moving the cursor). Of course, the same visual behaviour could be achieved by inserting invisible slots, or, if we insist on not modifying the document, by annotating the formula by its current correction.

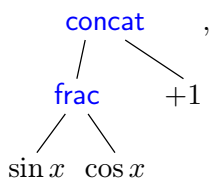
7.3. Focus

In order to describe the first grammar assisted editing feature inside TEXMACS, we first have to explain the concept of *current focus*.

Recall that a TEXMACS document is represented by a tree. Cursor positions are represented by a list $[i_1, \ldots, i_n]$ of integers: the integers $i_1, \ldots, i_{n-1}$ encode a path to a subtree in the document. If this subtree is a string leaf, then i_n encodes a position inside the string. Otherwise i_n is either 0 or 1, depending whether the cursor is before or behind the subtree. For instance, in the formula

$$\frac{\sin x}{\cos x} + 1$$

which is represented by the tree



,

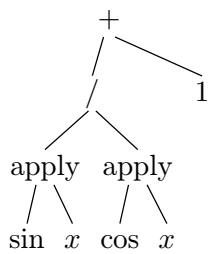
the list $[0, 0, 3]$ encodes the cursor position right after sin and $[0, 1]$ the cursor position right after the fraction and before the $+$. Notice that we prefer the leftmost position $[0, 1]$ over the equivalent position $[1, 0]$ in the second example.

At a certain cursor position, we may consider all subtrees of the document which contain the cursor. Inside TEXMACS, each of these subtrees is visualized by a light grey cyan bounding box. For instance, at the cursor position $[0, 0, 3]$, the above formula would typically be displayed as

$$\boxed{\frac{\sin|x}{\cos x}} + 1$$

The innermost subtree (in this case the fraction) is called the *current focus* and highlighted more prominently. In absence of selections, several so called *structured editing operations* operate on the current focus.

It is natural to generalize the above behaviour to semantic subformulas according to the current grammar. For instance, from a semantic point of view, the above formula corresponds to another tree:



(6)

For the cursor position just after sin, it would thus be natural to consider $\sin x$ as the *semantic current focus* and also surround this subexpression by a cyan box.

Let us describe how this can be done. The tree serializer for our packrat parser, which has been described in section 5.3, also maintains a correspondance between the cursor positions. For instance, the cursor position $[0, 0, 3]$ is automatically transformed into the cursor position 10 inside

```
<\frac>sin x<|>cos x</f>+1
```

Now a successful parse of a formula naturally induces a parse tree, which is not necessarily the same as the intended abstract syntax tree, but usually very close. More precisely, consider the set $\mathcal{S}$ of all successful parses of substrings (represented by pairs of start and end positions) involved in the complete parse. Ordering this set by inclusion defines a tree, which is called the parse tree.

In our example, the parse tree coincides with the abstract syntax tree. Each of the substrings in $\mathcal{S}$ which contains the cursor position can be converted back into a selection (determined by a start and an end cursor position) inside the original TEXMACS tree. In order to find the current focus, we first determine the innermost substring `sin x` in $\mathcal{S}$ (represented by the pair $(7, 12)$) which contains the cursor. This substring is next converted back into the selection $([0, 0, 0], [0, 0, 5])$.

In general, the parse tree does not necessarily coincide with the abstract syntax tree. In this case, some additional tweaking may be necessary. One source of potential discrepancies is the high granularity of packrat parsers, which are also used for the lexical analysis. For instance, floating point numbers might be defined using the following grammar:

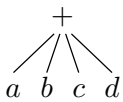
```
(define Integer (+ (- "0" "9")))
(define Number
  Integer
  (Integer "." Integer)
  (Integer "." Integer "e" Integer)
  (Integer "." Integer "e-" Integer))
```

However, in the example

123|45.67890

the substring 12345 would be regarded as the current focus. We thus have to add an annotation to the packrat grammar and (at least for the purpose of determining the current focus) either specify that `Integer` expressions should not be considered as valid subexpressions, or specify that `Number` expressions should be considered as atomic (i.e. without any subexpressions).

Other expressions for which parse trees and abstract syntax trees may differ are formulas which involve an associative operator. For instance, we might want to associate the abstract syntax tree



to the expression

$$a+b+c+d.$$

This would make the parse-subexpression $a+b$ "ineligible" for use as the current focus. In our current implementation, this behaviour is the default.

From the graphical point of view, it should noticed that many mathematical formulas do not require any two-dimensional layout:

$$f(x, g(a\,b)) = 3.$$

In such examples, the usual convention of surrounding *all* subexpressions containing the current cursor may lead to an excessive number of boxes:

$$f(x, g(a\,b)) = 3.$$

For this reason, we rather decided to surround only the current semantic focus, in addition to the subexpression which correspond to native TEXMACS markup. Notice that the level of detail for the visual hints can be further customized in the user preferences.

The semantic focus is a key ingredient for semantic mathematical editors, since it provides instant visual feedback on the meaning of formulas while they are being entered. For instance, when putting the cursor at the right hand side of an operator, the semantic focus contains all arguments of the operator. In particular, at the current cursor position, operator precedence is made transparent to the user *via* the cyan bounding boxes. On the one hand this makes it possible to check the semantics of a formula by traversing all accessible cursor positions (as we did in section 6.4.2).

On the other hand, in cases of doubt on the precedence of operators, the semantic focus informs us about the default interpretation. Although the default interpretation of a formula such as $1/2\,\pi\,\mathrm{i}$ might be wrong, the editor's interpretation is at least made transparent, and it could be overridden through the insertion of invisible brackets. In other words, we advocate an *a posteriori* approach to the resolution of common mathematical precedence ambiguities, rather than the *a priori* approach mentioned in remark 2.

7.4. Selections

Semantic selections can be implemented along similar lines as the semantic focus, but it is worth it to look a bit closer at the details. We recall that a TEXMACS selection is represented by a pair with a cursor starting position and a cursor end position.

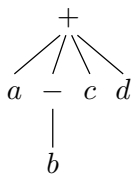
The natural way to implement semantic selections is as follows. We first convert the selection into a substring (encoded by a starting and end position) of the string serialization of the TEXMACS tree. We next extend the substring to the smallest "eligible" substring in $\mathcal{S}$, for a suitable customizable definition of "eligibility". We finally convert the larger substring back into a selection inside the original TEXMACS document, which is returned as the semantic selection.

In the above algorithm, the definition of eligible substring for selection may differ from the one which was used for the current focus. For instance, for an associative operator $+$, it is natural to consider both $a+b+c$ and $b+c+d$ as eligible substrings of $a+b+c+d$. Actually, we might even want to select $b+c$ as a substring of $a+b+c+d$.

In the presence of subtractions, things get even more involved, since we need an adequate somewhat *ad hoc* notion of abstract syntax tree. For instance,

$$a-b+c+d$$

would typically be encoded as



after which $-b+c$ becomes an eligible substring for selection, but not $b+c$.

Furthermore, it is sometimes useful to select the subexpression $+c+d$ of $a+b+c+d$. This occurs for instance if we want to replace $a+b+c+d$ by $a+b+c+d+c+d$. However, when adopting a naive point of view on semantic selections, which might even integrate the above remarks, $+c+d$ remains non eligible as a substring of $a+b+c+d$. Nevertheless, such substrings *do* admit operator semantics. More precisely, we may consider the selection $+c+d$ as an operator

$$\cdot+c+d\colon x \longmapsto x+c+d,$$

which will be applied to a suitable subexpression at the position where we perform a "paste". A less trivial example would be the "selection operator"

$$\frac{1}{1+\cdot}\colon x \longmapsto \frac{1}{1+x}.$$

Although it is harder to see how we would select such an operator, this generalization might be useful for rapidly typing the continued fraction

$$\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\ddots}}}}$$

Notice that this kind of semantics naturally complements the idea to use the paste operation for substitutions of expressions (usually variables) by other expressions. This idea was heard in a conference talk by O. ARSAC, but we ignore to who it was originally due.

It should be noticed that there are other important situations in which non semantic selections are useful. For instance, the default interpretation of the formula $a \wedge b \vee c$ is $(a \wedge b) \vee c$. Imagine a context in which the desired meaning is rather $a \wedge (b \vee c)$. In that case, the user might want to select the formula $b \vee c$ and put (possibly invisible) brackets around it. The same problem occurs if, for some reason, we need to transform a formula such as $a\,b+c$ into $a\,(b+c)$. Without the ability to perform non semantic selections, there is no easy way perform such transformations, which are actually needed quite frequently. For this reason, we have opted for non semantic selections by default, although TEXMACS provides a user preference for enabling semantic selections. We might also allow the user to switch between both behaviours by holding a modifier key while making the selection.

Remark 5. In order to complete the discussion on current focus and selections, we should mention the fact that, in presence of a selection, the current focus is rather defined as being the smallest subexpression which contains the current selection (instead of the cursor). In particular, structured editing operations which are defined for the cursor focus (e.g., inside a fraction, swapping the numerator and denominator, or move to the next fraction) naturally extend to (semantic) selections. Moreover, repeated mouse clicks or repeatedly applying the `Ctrl-space` keyboard shortcut allows for quick *outward selection* (starting with the current focus, select the next larger subexpression at each subsequent iteration).

7.5. Grammar assisted typesetting

Mathematical grammars can also be useful at the typesetting phase in several ways.

First of all, the spacing around operators may depend on the way the operator is used. Consider for instance the case of the subtraction operator $-$. There should be some space around it in a usual subtraction $x - y$, but not in a unary negation $-x$. We also should omit the space when using the subtraction as a pure operation: $\square \in \{+, -, \cdot\}$. Usually, no parsing is necessary in order to determine the right amount of spacing: only the general symbol types of the operator itself and its neighbours should be known for this task. We have seen in section 5.3 how to annotate packrat grammars (using `:type` and `:spacing`) in order to provide this information. Notice that this kind of refined spacing algorithms are also present in TEX [15].

In a similar vein, we may provide penalties for line breaking algorithms. In fact, grammar based line breaking algorithms might go much further than that. For instance, we may decide to replace fractions $\frac{a}{b}$ with very wide numerators or denominators by a/b, thereby making them easier to hyphenate. Such replacements can only be done in a reliable way when semantic information is available about when to use brackets. For instance, $\frac{a+b}{c}$ should be transformed into $(a + b)/c$, but $\frac{a\,b}{c}$ into $a\ b/c$. In order to avoid expensive parsing during the typesetting phase, we may store this bracketing information in the document or suitable annotations during the editing phase.

Notice that semantic hyphenation strategies should be applied with some caution: if one decides to transform a fraction occurring in a long power series with rational coefficients, then it is usually better to transform all fractions: a mixture of transformed and untransformed fractions will look a bit messy to the human eye. For similar reasons, one should systematically transform fractions when a certain width is exceeded, rather than trying to mix the usual hyphenation algorithm with semantic transformations.

Finally, in formulas with more nesting levels, one should carefully choose the level at which transformations are done. For instance, if a large polynomial matrix

$$\begin{pmatrix} a^2\,b^2 + a\,b^2 + 7\,b^2 + a^3\,b + 3\,a\,b - 8 & a^7 - a^6 + a^5 - a^4 + a^3 + a^2 - a + 1 \\ a^7 - a^6 + a^5 - a^4 + a^3 + a^2 - a + 1 & a^2\,b^2 + a\,b^2 + 7\,b^2 + a^3\,b + 3\,a\,b - 8 \end{pmatrix}$$

does not fit on a line, then we may transform the matrix into a double list:

$$[[a^2\ b^2 + a\ b^2 + 7\ b^2 + a^3\ b + 3\ a\ b - 8,\ a^7 - a^6 + a^5 - a^4 + a^3 + a^2 - a + 1],$$
$$[a^7 - a^6 + a^5 - a^4 + a^3 + a^2 - a + 1, a^2\,b^2 + a\,b^2 + 7\,b^2 + a^3\,b + 3\,a\,b - 8]].$$

Nevertheless, a more elegant solution might be to break the polynomials inside the matrix:

$$\begin{pmatrix} a^2\,b^2 + a\,b^2 + 7\,b^2 + & a^7 - a^6 + a^5 - a^4 + \\ a^3\,b + 3\,a\,b - 8 & a^3 + a^2 - a + 1 \\ \\ a^7 - a^6 + a^5 - a^4 + & a^2\,b^2 + a\,b^2 + 7\,b^2 + \\ a^3 + a^2 - a + 1 & a^3\,b + 3\,a\,b - 8 \end{pmatrix}$$

For the moment, our implementation does not reach this level of sophistication; it would already be nice if the user could easily perform this kind of transformations by hand.

7.6. Editing homoglyphs

In section 3.1, we have seen that homoglyphs are a major source of ambiguity and one of the bottlenecks for the design of semantic editors. This means that we should carefully design input methods which incite users to enter the appropriate symbols, without being overly pedantic. Furthermore, there should be more visual feedback on the current interpretation of symbols, while avoiding unnecessary clobbering of the screen. We are investigating the following ideas for implementation in TEXMACS:

- An *a priori* way to disambiguate homoglyphs, while they are being entered, is to introduce the following new keyboard shortcut convention: if we want to override the default rendering of a symbol or operator, then type the appropriate key for the intended rendering just afterwards. For instance, we might use `+ space` to enter an invisible space or `* &` to enter the wedge product. This convention benefits from the fact that many homoglyphs are infix operators, so that the above kind of shortcuts have no other natural meaning.

- An alternative *a posteriori* approach would be to introduce a new shortcut for cycling through "homoglyph variants". The usual variant key might also be used for this purpose, although the list of possible variants would become a bit too long for certain symbols.

- Slightly visible graphical hints might be provided to users in order to inform on the current interpretation of a symbol. For instance, multiplication might be rendered as xy, function application as $\sin x$, and the wedge product as $x \wedge y$, without changing any of the spacing properties. The rendering might also be altered more locally, when the cursor is inside the formula, or next to the operator itself.

Besides, we are in the process of dressing up a list with the most common homoglyphs.

Unfortunately, at the time of writing, the UNICODE standard does not provide a lot of support on this issue, which we regard as a design flaw. For non mathematical scripts, homoglyphs are usually treated in a clean way. For instance, UNICODE distinguishes between the latin, the cyrillic and the greek capital O. However, this convention is dropped when it comes to mathematics:

> « **Semantics.** Mathematical operators often have more than one meaning in different subdisciplines or different contexts. For example, the "+" symbol normally denotes addition in a mathematical context, but might refer to concatenation in a computer science context dealing with strings, or incrementation, or have any number of other functions in given contexts. Therefore the Unicode Standard only encodes a single character for a single symbolic form. There are numerous other instances in which several semantic values can be attributed to the same Unicode value. For example, U+2218 ∘ RING OPERATOR may be the equivalent of white small circle or composite function or apl jot. The Unicode Standard does not attempt to distinguish all possible semantic values that may be applied to mathematical operators or relational symbols. It is up to the application or user to distinguish such meanings according to the appropriate context. Where information is available about the usage (or usages) of particular symbols, it has been indicated in the character annotations in the code charts printed in [Unicode] and in the online code charts [Charts]. »

It is interesting to examine the reason which is advanced here: as a binary operator, the + might both be used for addition and string concatenation. In a similar way, the latin O might be pronounced differently in english, hungarian or dutch. From the *syntactical* point of view, these distinctions are completely irrelevant: both when used as an addition or string concatenation, the + remains an associative infix operator with a fixed precedence.

For genuine homoglyphs, such as the logical or and the wedge product, the syntactical status usually changes as a function of the desired interpretation. For instance, the binding force of a wedge product is the same as the binding force of a product. Furthermore, such operators would usually be placed in different categories. For instance, the logical or can be found in the category "isotech" of general technical symbols. It would have been natural to add the wedge product in the category "isoamsb" of binary operators.

In fact, UNICODE breaks its own rules in various subtle ways. For instance, invisible mathematical operators can also be considered as homoglyphs. For this special case, UNICODE does provide an incomplete set of unambiguous operators U+2061 until U+2064. Another, more perverse example is the suggested use of double stroke letters for frequent mathematical constants. For instance, a suggested symbol for the mathematical constant $\mathrm{e} = 2.71828\cdots$ is the ugly-looking ⅇ (U+2147).

Yet another example concerns the added code charts for sans serif and monospaced characters. For newly developed mathematical fonts, such as the proprietary STIX fonts, these characters tend to be misused as a replacement for typesetting formulas or text in special fonts. For instance, monospaced characters are suggested for typesetting identifiers in computer programs, which would thereby become invisible for standard search tools. We don't know of any classical mathematical notation which uses monospaced symbols and would have justified the introduction of such curiousities.

When taking a look at the existing mathematical UNICODE charts, we may also wonder whether there are really *that* many common mathematical homoglyphs. Indeed an abundance of such homoglyphs might have justified the lack of support for "all possible semantic values" of symbols. However, this hypothetic situation would have been very confusing for mathematicians at a first place. Mathematicians rather tend make suggestive use of existing symbols: they will use a + symbol for operations which behave like additions in many respects; why would they use it to denote, say, existential quantification?

8. CONCLUSION AND FUTURE EXTENSIONS

Currently, the mainstream mathematical editing tools are presentation oriented and unsuitable for the production of documents in which all formulas are at least syntactically correct. In the present paper, we have described a series of techniques which might ultimately make the production of such documents as user friendly as the existing presentation oriented programs.

The general philosophy behind our approach is that the behaviour of the editor should remain as close as possible to the behaviour of a presentation oriented editor, except, of course, for those editing actions which may benefit from the added semantics. One major consequence is that the editor naturally adapts itself to the editing habits of the user, rather than the converse. Indeed, it is tempting to "teach" the user how to write content markup, but the extra learning efforts should be rewarded by a proportional number of immediate benefits for the user. This holds especially for those users whose primary interest is a paper which looks nice, and who do not necessarily care about the "added value" of content markup.

Technically speaking, our approach relies on decoupling the visual editing facilities from the semantic checking and correction part. This is achieved through the development of a suitable collection of packrat grammar tools which allow us to maintain a robust correspondance between presentation and content markup. This correspondance comprises cursor positions, selections, and anything else needed for editing purposes. So far, our collection of grammar tools allowed us to naturally extend the most basic editing features of TEXMACS to the semantic context. Although we found it convenient to use packrat grammars, it should be noticed that other types of grammars might be used instead, modulo an additional implementation effort.

One main difficulty for the development of a general purpose semantic editor is the wide variety of users. Although we should avoid the pitfall of introducing a large number of user preferences (which would quickly become unintelligible to most people), we will probably need to introduce at least a few of them, so as to address the wide range of potential usage patterns. It will take a few years of user feedback before we will be able to propose a sufficiently stable selection of user preferences. In this paper, we have contented ourselves to investigate various possible solutions to problems in which personal taste plays an important role.

Another aspect of customization concerns the underlying mathematical grammar. We have argued that the design of grammars should be left to experts. Indeed, designing grammars is non trivial, and non standard grammars will mostly confuse potential readers. We are thus left with two main questions: the design of a "universal" grammar which is suitable for most mathematical texts, and an alternative way for the user to introduce customized notation. In section 6, we both introduced such a grammar and described our approach of using TEXMACS macros for introducing new notations. Such macros can be grouped in style packages dedicated to specific areas. Of course, our proposal for a "universal" grammar will need to be tested and worked out further.

In some cases though, it is important to allow for customizable grammars. One good example is the use of TEXMACS as an interface to an automatic theorem prover. Such systems often come with their own grammar tools for customizing notations. In such cases, it should not be hard to convert the underlying grammars into our packrat format. Moreover, packrat grammars are very well adapted to this situation, since they are incremental by nature, so it easy and efficient to use different grammars at different places inside the document. An alternative approach, which might actually be even more user friendly, would be to automatically convert standard notation into the notation used by the prover.

In sections 3.4 and 6.4, we have tested the newly implemented semantic tools on some large existing TEXMACS documents from various sources. These experiments made as reasonably optimistic about our ultimate goal of building a user friendly semantic editor for the working mathematician. We have identified a few main bottlenecks, which suggest several improvements and enhancements for the current editor:

- Following section 7.6, we should provide better visual feedback and editing tools for homoglyphs. In version 1.0.7.11, some feedback is given for invisible multiplication by clicking on the menu item Document→View→Informative flags→Short.
- TEXMACS traditionally does not use slots for displaying missing subformulas. Along the lines described in section 7.2, we plan to experiment with an editing mode which will use them in a systematic way so as to keep formulas correct at all stages.
- The "universal grammar" should be tested on a wider range of documents and carefully enhanced to cover most standard notations.
- We should work out a comprehensible set of annotation primitives, which users will find natural when they need to put text inside their formulas or resort to *ad hoc* notations.

- Additional non grammar based hacks might be implemented for parsing certain types of special markup. For instance, inside tables such as (4), we might concatenate certain cells before launching the parser.
- Since the current implementation is very recent, many additional tweaking will be necessary in response to user feedback.

We also have various other plans for future extensions:

- We plan to add productions to our packrat grammars, which will enable us to automatically generate, say, MATHML content markup for all formulas. It should also be possible to translate between distinct grammars.
- We intend to implement at least some rudimentary support for grammar based typesetting, starting with line breaking, as outlined in section 7.5.
- Systematic generalization of all currently supported editing operations (such as search and replace, tab-completion, structured navigation, etc.) to the semantic context.
- Using the packrat grammar tools for programming languages as well, so as to support syntax highlighting, automatic indentation, etc. After dealing with purely textual programs, we will next investigate the possibility to insert mathematical formulas directly into the source code.
- Exploit the extra semantics in our interfaces with external systems.

BIBLIOGRAPHY

[1] Olivier Arsac, Stéphane Dalmas et Marc Gaëtano. The design of a customizable component to display and edit formulas. In *ACM Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation, July 28 - 31*, pages 283–290. 1999.

[2] Philippe Audebaud et Laurence Rideau. Texmacs as authoring tool for formal developments. *Electr. Notes Theor. Comput. Sci.Volume*, 103:27–48, 2004.

[3] S. Basu, R. Pollack et M.-F. Roy. *Algorithms in real algebraic geometry*, volume 10 of *Algorithms and Computation in Mathematics*. Springer-Verlag, 2006. 2-nd edition to appear.

[4] S. Basu, R. Pollack et M.F. Roy. *Algorithms in real algebraic geometry*, volume 10 of *Algorithms and computation in mathematics*. Springer-Verlag, Berlin, Heidelberg, New York, 2003.

[5] T. Braun, H. Danielsson, M. Ludwig, J. Pechta et F. Zenith. Kile: an integrated latex environment. Http://kile.sourceforge.net/, 2003.

[6] E. Chou. Collected course material. `http://math.nyu.edu/~chou/`, 2010.

[7] F. DeRemer. *Practical LR(k) Translators*. PhD thesis, Massachusetts Institute of Technology, 1969.

[8] FSF. Gnu bison. `http://www.gnu.org/software/bison/`, 1998.

[9] B. Ford. Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology, September 2002.

[10] Bryan Ford. Packrat parsing: simple, powerful, lazy, linear time. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, ICFP '02, pages 36–47. New York, NY, USA, 2002. ACM.

[11] K. Geddes, G. Gonnet et Maplesoft. Maple. `http://www.maplesoft.com/products/maple/`, 1980.

[12] A.G. Grozin. TeXmacs interfaces to Maxima, MuPAD and Reduce. In V.P. Gerdt, editor, *Proc. Int. Workshop Computer algebra and its application to physics*, volume JINR E5, page 149. Dubna, June 2001. Arxiv cs.SC/0107036.

[13] A.G. Grozin. TeXmacs-Maxima interface. Arxiv cs.SC/0506226, June 2005.

[14] S.C. Johnson. Yacc: yet another compiler-compiler. 1979.

[15] D.E. Knuth. *The TeXbook*. Addison Wesley, 1984.

[16] M. Kohlhase. OMDoc. http://www.omdoc.org/, 2000.

[17] L. Lamport. *LaTeX, a document preparation system*. Addison Wesley, 1994.

[18] L. Mamane et H. Geuvers. A document-oriented Coq plugin for TeXmacs. In *Mathematical User-Interfaces Workshop 2006*. St Anne's Manor, Workingham, UK, 2006.

[19] T. Nipkow, L. Paulson et M. Wenzel. Isabelle/Hol. http://www.cl.cam.ac.uk/research/hvg/Isabelle/, 1993.

[20] S. Owre, N. Shankar et J. Rushby. Pvs specification and verification system. http://pvs.csl.sri.com/, 1992.

[21] MacKichan Software. Scientific workplace. http://www.mackichan.com/index.html?products/swp.html~mainFrame, 1998.

[22] G.J. Sussman et G.L. Steele Jr. Scheme. http://www.schemers.org/, 1975.

[23] A. Tejero-Cantero. Interference and entanglement of two massive particles. Master's thesis, Ludwig-Maximilians Universität München, September 2005. http://alv.tiddlyspace.com/I2P.

[24] The OpenMath Society. OpenMath. http://www.openmath.org/, 2003.

[25] Unicode Consortium. Unicode. http://www.unicode.org/, 1991.

[26] W3C. MathML. http://www.w3.org/Math/, 1999.

[27] Wolfram Research. Mathematica. http://www.wolfram.com/mathematica/, 1988.

[28] N.G. de Bruijn. The mathematical language automath, its usage, and some of its extensions. In *Symposium on Automatic Demonstration*, volume 125 of *Lect. Notes in Math.*, pages 29–61. Versailles, December 1968, 1970. Springer Verlag.

[29] M. Ettrich et al. The LyX document processor. http://www.lyx.org, 1995.

[30] T. Coquand et al. The Coq proof assistant. http://coq.inria.fr/, 1984.

[31] L. van den Dries. *Tame topology and o-minimal structures*, volume 248 of *London Math. Soc. Lect. Note*. Cambridge university press, 1998.

[32] J. van der Hoeven. GNU. http://www.texmacs.org, 1998.

[33] J. van der Hoeven. GNU TeXmacs: a free, structured, wysiwyg and technical text editor. In Daniel Flipo, editor, *Le document au XXI-ième siècle*, volume 39–40, pages 39–50. Metz, 14–17 mai 2001. Actes du congrès GUTenberg.

[34] J. van der Hoeven. *Transseries and real differential algebra*, volume 1888 of *Lecture Notes in Mathematics*. Springer-Verlag, 2006.

[35] J. van der Hoeven. *Transséries et analyse complexe effective*. PhD thesis, Univ. d'Orsay, 2007. Mémoire d'habilitation.