Interactive Editing of MathML Markup Using TEX Syntax*

Luca Padovani
Department of Computer Science, University of Bologna
Mura Anteo Zamboni, 7
40127 Bologna
Italy
lpadovan@cs.unibo.it
http://www.cs.unibo.it/~lpadovan/

Abstract

We describe the architecture of a syntax-directed editor for authoring structured mathematical documents which can be used for the generation of MathML markup [4]. The author interacts with the editor by typing TEX markup as he would do in a normal text editor, with the difference that the typed markup is parsed and displayed on-the-fly. We discuss issues regarding both the parsing and presentation phases and we propose implementations for them. In contrast with existing similar tools, the architecture we propose offers better compatibility with TEX syntax, a pervasive use of standard technologies and a clearer separation of content and presentation aspects of the information.

1 Introduction

MathML [4] is an XML [2] application for the representation of mathematical expressions. Like most XML applications, MathML is unsuitable to be written directly because of its verbosity except in the simplest cases. Hence the editing of MathML documents needs the assistance of dedicated tools. As of today, such tools can be classified into two main categories:

1. WYSIWYG (What You See Is What You Get) editors that allow the author to see the formatted document on the screen while it is being composed. The editor usually provides some "export mechanism" that creates XML with embedded MathML from the internal representation of the document;
2. conversion tools that generate MathML markup from different sources, typically other markup languages for scientific documents, such as TEX [5].

Tools in the first category are appealing, but they suffer from at least two limitations: a) editing is typically *presentation oriented* — the author is primarily concerned about the "look" of the document and tends to forget about its content. b) They may slow down the editing process because they often involve the use of menus, palettes of symbols, and, in general, the pointing device for completing most operations.

In this paper we describe the architecture of a tool that tries to synthesize the "best of both worlds". The basic idea is to create a WYSIWYG editor in which editing is achieved by typing concrete markup as the author would do in an actual plain text editor. The markup is then tokenized and parsed on-the-fly, a corresponding presentation is created by means of suitable transformations, and finally displayed. The editor is meant not only as an authoring tool, but more generally as an interface for math applications.

Although in the paper we assume that the concrete markup typed by the user is TEX (more precisely the subset of TEX concerned about mathematics) and that presentation markup is MathML, the system we are presenting is by no means tied to these languages and can be targeted to other contexts as well. One question that could arise is: "why TEX syntax?" We can see at least three motivations: first of all because of TEX popularity in many communities. Second, because macros, which are a fundamental concept in TEX, are also the key to editing at a more content-oriented level, which is a primary requirement for many applications handling mathematics. Finally, because, as we will see, TEX markup has good *locality* properties which make it suitable in the interactive environment of our concern.

* This work has been supported by the European Project IST-2001-33562 MoWGLI.

The body of the paper is structured into four main sections: in Section 2 we overview the architecture of the tool while in Sections 3, 4, 5 we describe in more detail the main phases of the editing process (lexing, parsing, and transformation). Familiarity with TEX syntax and XML-related technologies is assumed.

2 Architecture

Several tools for the conversion of TEX markup suffer from two major drawbacks that we are not willing to tolerate in our design: (1) they rely on the TEX system itself for parsing the markup. While guaranteeing perfect compatibility with TEX, this implies the installation of the whole system. Moreover, the original TEX parser does not meet the incremental requirements that we need; (2) the lack of flexibility in the generation of the target document representation, which is either fixed by the conversion tool or it is only slightly customizable by the user.

To cope with problem (1) we need to write our own parser for TEX markup. This is well known to be a non-trivial task, because of some fancy aspects regarding the very nature of TEX syntax and the lack of a proper "TEX grammar". We will commit ourselves with a subset of TEX syntax which appears to be just what an average author needs when writing a document. As we will see, the loss in the range of syntactic expression is compensated by a cleaner and more general transformation phase. As for the lack of a TEX grammar, we perceive this as a feature rather than a weakness: after all TEX is built around the fact that authors are free to define their own macros. Macros are the fundamental entities giving structure to the document.

Let us now turn our attention to problem (2): recall that the general form of a TEX macro definition (see *The TEXbook*, [5]) is

\def⟨control sequence⟩⟨parameter text⟩
{⟨replacement text⟩}

where the ⟨parameter text⟩ gives the syntax for invoking the macro and its parameters whereas the ⟨replacement text⟩ defines somehow the "semantics" of the macro (typically a presentational semantics). Thus the ultimate semantic load of a macro is invariably associated with the configuration of the macro at the point of definition.

We solve problem (2) by splitting up macro definitions so that structure and semantics can be treated independently. A well-formed TEX document can be represented as a tree whose leaves are either literals (strings of characters) or macros with no parameters, and each internal node represents a macro and the node's children are the macro's parameters. Entities like delimiters, square brackets surrounding optional parameters or literals occurring in the ⟨parameter text⟩ of macro definitions are purely syntactic and need not be represented in the tree if our main concern is capturing the structure of the document. Fig. 1(a) shows the tree structure of a simple mathematical formula.

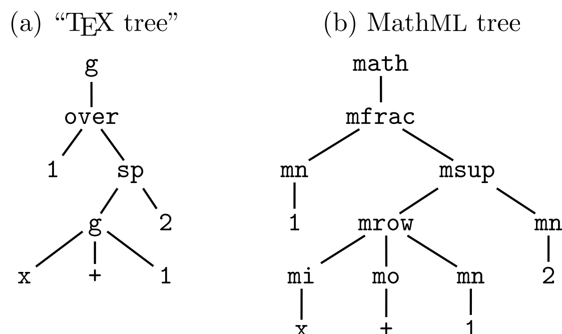

Figure 1: Tree representation for `{1\over{x+1}^2}` and corresponding MathML markup.

Once the document is represented as a tree, the process of macro expansion — that is, *interpretation* — can be defined as a recursive transformation on the nodes of the tree. As we will represent trees using XML, transformations can be very naturally implemented by means of XSLT stylesheets [3]. Fig. 1(b) shows the MathML tree corresponding to the TEX tree on the left hand side. The two trees are basically isomorphic except for the name of the nodes and the presence of explicit token nodes for literals in the MathML tree. This is to say that the MathML tree can be generated from the TEX tree by simple transformations. However, once the interpretation phase is independent of parsing (something which does not happen in TEX) it is natural to define much more general transformations that are not just node-by-node rewritings.

The following are the main components of an interactive, syntax-based editor for structured documents:

Input Buffer: the sequence of concrete characters typed by the author;

Lexical Analyzer: responsible for the tokenization of the characters in the input buffer;

Dictionary: a map from ⟨control sequence⟩ to ⟨parameter text⟩ which is used to know the syntax of macros;

Parser: for the creation of the internal tree structure representing the document;

TRANSFORMATION ENGINE: for mapping the internal tree into the desired format.

No doubt these entities are common to all tools converting TeX markup into a different format, but the degree of mutual interdependence and the way they are implemented may differ considerably, especially when interactivity is a main concern. The added value of our approach is that it allows the author to independently customize both the dictionary and the transformation engine, and the advanced user of the editor the possibility of adapting the lexical analyzer to languages other than TeX (we will spend a few more words on this topic in the conclusions).

Notation We will use the following conventions regarding lists. Lists are uniformly typed, that is elements of a list are all of the same type. We use α^* to denote the type of a list whose elements have type α. $[]$ is the empty list; $n :: x$ is the list with head element n and tail x; $x@y$ is the concatenation of two lists x and y; $[n_1; \ldots; n_k]$ is a short form for $n_1 :: \cdots :: n_k :: []$.

3 Lexical Analysis

The purpose of this phase is to tokenize the input buffer. As we are talking about an interactive tool, the presence of an input buffer may look surprising. Implementations for the input buffer range from *virtual* buffers (there is no buffer at all, characters are collected by the lexical analyzer which outputs tokens as they are completed) to *flat* buffers (just a string of characters as in a text editor) to *structured* buffers. For efficiency, we do not investigate in detail all the possibilities in this paper, but early experiments have shown that working with virtual buffers can be extremely difficult. As long as insert operations are performed at the right end of the buffer the restructuring operations on the parsing tree are fairly easy, but when it comes to deletion or to modifications in arbitrary positions, the complexity of restructuring operations rises rapidly to an unmanageable level. Hence, from now on we will assume that a flat input buffer is available. Whether the buffer should be visible or not is a subjective matter, and may also depend on the kind of visual feedback given by the editor on incomplete and/or incorrect typed markup.

The outcome of the lexer is a stream (list) of tokens. Each token may have one of three forms: a *literal*, that is a single character to be treated "as is", a *space*, that is a sequence of one or more space-like characters, or a *control sequence*, that is the name of a macro.

Since the token stream is the only interface between the lexer and the parser, the lexer has the freedom to perform arbitrary mappings from the characters in the input buffer to tokens in the stream. In particular, some TeX commands like `\alpha` or `\rightarrow` are just placeholders for Unicode characters. There is no point in communicating these entities as control sequences as the internal tree representation (XML) is able to accommodate Unicode characters naturally; also, treating them as literals simplifies the subsequent transformation phase.

On the other hand, there are characters, such as curly braces `{` and `}` or scripting operators `_` and `^`, that have a special meaning. Logically these are just short names for macros that obey their own rules regarding parameters. What we propose is a general classification of parameter types which, in addition to parameters in normal TeX definitions, allows us

- to deal with optional parameters as LaTeX [6] does;
- to treat `{` as just an abbreviation for `\bgroup` and make `\bgroup` a macro with one parameter delimited by `\egroup`, which we treat as the expansion for `}`. In order for this "trick" to work we have to design the parser carefully, as we will see in Sect. 4;
- to treat scripting operators `_` and `^` as the two macros `\sb` and `\sp` both accepting a so-called pre-parameter (a parameter that occurs *before* the macro in the input buffer) and a so-called post-parameter (a parameter that occurs *after* the macro in the input buffer);
- to deal with macros that have "open" parameters. For instance `\rm` affects the markup following it until the first delimiter coming from an outermost macro is met. We treat `\rm` as a macro with an open post-parameter that extends as far as possible to the right. Similarly, `\over` can be seen as a macro with open pre- and post-parameters.

In order to describe parameter types we need to define the concept of *term*. A *term* is either a literal or a macro along with all its parameters (equivalently, a term is a subtree in the parsing tree). A *simple* parameter consists of one sole term. A *compound* parameter consists of one or more terms extending as far as possible to the left or to the right of the macro depending on whether the parameter is "pre-" or "post-". A *delimited* parameter consists of one or more terms extending as far as possible to the right up to but not including a given token

Table 1: Examples of TEX and LATEX macros along with their signature.

Macro	Parameters	
	pre	**post**
`overline`		[simple]
`sqrt`		[simple] (TEX)
		[optional; simple] (LATEX)
`root`		[delimited(control(`of`)); simple]
`over`, `choose`	[compound]	[compound]
`frac`		[simple; simple]
`rm`, `bf`, `tt`, `it`		[compound]
`left`		[simple; delimited(control(`right`)); simple]
`sb`, `sp`	[simple]	[simple]
`bgroup`		[delimited(control(`egroup`))]
`begin`		[simple; optional; delimited(control(`end`)); simple]
`proclaim`		[token(space); delimited(literal(.)); token(space); delimited(control(`par`))]

t. An *optional* parameter is either empty or it consists of one or more terms enclosed within a pair or square brackets `[` and `]`. The absence of the opening bracket means that the optional parameter is not given. A *token* parameter is a given token t representing pure syntactic sugar. It does not properly qualify as a parameter and does not appear in the parsing tree.

Formally tokens and parameter types are defined as follows:

$$\begin{array}{rcl} \mathit{token} & ::= & \mathsf{literal}(v) \mid \mathsf{space} \mid \mathsf{control}_{\langle p_1, p_2\rangle}(v) \\ \mathit{type} & ::= & \mathsf{simple} \mid \mathsf{compound} \mid \mathsf{delimited}(t) \\ & & \mid \mathsf{optional} \mid \mathsf{token}(t) \end{array}$$

where $t \in \mathit{token}$, $v \in \mathit{string}$ is an arbitrary string of Unicode characters, $p_1 \in \{\mathsf{simple}, \mathsf{compound}\}^*$ and $p_2 \in \mathit{type}^*$ are lists of parameter types for the pre- and post-parameters respectively. Note that pre-parameters can be of type simple or compound only.

The dictionary is a total map

$$\mathit{dictionary} : \mathit{string} \mapsto \mathit{token}$$

such that for each unknown control sequence v we have $\mathit{dictionary}(v) = \mathsf{control}_{\langle [], [] \rangle}(v)$. Table 1 shows part of a possible dictionary for some TEX and LATEX commands (mostly for mathematics). Note how it is possible to encode the signature for the `\begin` control sequence, although it is not possible to enforce the constraint that the first and the last parameters must have equal value in order for the construct to be balanced.

4 Parsing

We now come to the problem of building the TEX parsing tree starting the stream of tokens produced by the lexical analyzer. As we have already pointed out there is no fixed grammar that we can use to generate the parser automatically: authors are free to introduce new macros and hence new ways of structuring the parse tree. Thus we will build the parser "by hand". More reasons for writing an ad-hoc parser, namely error recovery and incrementality, will be discussed later in this section.

The following grammar captures formally the structure of a TEX parsing tree, which is the outcome of the parser:

$$\begin{array}{rcll} \mathit{node} & ::= & \mathsf{empty} & \\ & \mid & \mathsf{literal}(v) & v \in \mathit{string} \\ & \mid & \mathsf{macro}(v, x) & v \in \mathit{string}, x \in \mathit{param}^* \\ \mathit{param} & ::= & \{a\} & a \in \mathit{node}^* \end{array}$$

Note that a parameter is made of a list of nodes and that literals are strings instead of single characters. The empty node is used to denote a missing term when one was expected; its role will be clarified later in this section.

The appendix contains the Document Type Definition for the XML representation of TEX parsing trees. It is simpler than the TEXML DTD [7] and we are providing it as mere reference.

4.1 Parsing Functions

Table 2 gives the operational semantics of the parser. In this table only, for each $a \in \mathit{node}^*$ we define $a! = [\mathsf{empty}]$ if $a = []$ and $a! = a$ otherwise. There are four parsing functions: $\mathcal{T}$ for terms, $\mathcal{A}$ for pre-parameters, $\mathcal{B}$ for post-parameters, and $\mathcal{C}$ for delimited sequences of terms. Each parsing function is defined by induction on the structure of its arguments. Axioms (rules with no horizontal line) denote base cases, while inference rules define the value

Table 2: Parsing functions for the simplified TEX markup.

$$
\begin{array}{rccl}
\forall d \in \mathit{token}^* & \xrightarrow{\mathcal{T}(d)} & : & \mathit{node}^* \times \mathit{token}^* \to \mathit{node}^* \times \mathit{token}^* \\
 & \xrightarrow{\mathcal{A}} & : & \mathit{node}^* \times \mathit{type}^* \to \mathit{node}^* \times \mathit{param}^* \\
\forall d \in \mathit{token}^* & \xrightarrow{\mathcal{B}(d)} & : & \mathit{type}^* \times \mathit{token}^* \to \mathit{param}^* \times \mathit{token}^* \\
\forall d \in \mathit{token}^*, \forall b \in \mathit{bool} & \xrightarrow{\mathcal{C}(d,b)} & : & \mathit{node}^* \times \mathit{token}^* \to \mathit{node}^* \times \mathit{token}^*
\end{array}
$$

$$(T.1)\; a, [] \xrightarrow{\mathcal{T}(d)} a, [] \qquad (T.2)\; a, t :: l \xrightarrow{\mathcal{T}(d)} a, t :: l \quad (t \text{ occurs in } d)$$

$$(T.3)\; a, \mathsf{literal}(v) :: l \xrightarrow{\mathcal{T}(d)} a@[\mathsf{literal}(v)], l \qquad (T.4)\; \frac{a, l \xrightarrow{\mathcal{T}(d)} a', l'}{a, \mathsf{space} :: l \xrightarrow{\mathcal{T}(d)} a', l'}$$

$$(T.5)\; \frac{a, p_1 \xrightarrow{\mathcal{A}} a', x \quad p_2, l \xrightarrow{\mathcal{B}(d)} y, l'}{a, \mathsf{control}_{\langle p_1, p_2 \rangle}(v) :: l \xrightarrow{\mathcal{T}(d)} a'@[\mathsf{macro}(v, x@y)], l'}$$

$$(A.1)\; a, [] \xrightarrow{\mathcal{A}} a, [] \qquad (A.2)\; \frac{[], p \xrightarrow{\mathcal{A}} a, x}{[], s :: p \xrightarrow{\mathcal{A}} a, x@[\{[\mathsf{empty}]\}]}$$

$$(A.3)\; \frac{a, p \xrightarrow{\mathcal{A}} a', x}{a@[n], \mathsf{simple} :: p \xrightarrow{\mathcal{A}} a', x@[\{[n]\}]} \qquad (A.4)\; \frac{[], p \xrightarrow{\mathcal{A}} a', x}{a, \mathsf{compound} :: p \xrightarrow{\mathcal{A}} a', x@[\{a\}]}$$

$$(B.1)\; [], l \xrightarrow{\mathcal{B}(d)} [], l \qquad (B.2)\; \frac{p, t' :: l \xrightarrow{\mathcal{B}(d)} x, a}{\mathsf{token}(t) :: p, t' :: l \xrightarrow{\mathcal{B}(d)} x, a}\,\dagger \quad (t \neq t')$$

$$(B.3)\; \frac{p, [] \xrightarrow{\mathcal{B}(d)} x, l}{\mathsf{token}(t) :: p, [] \xrightarrow{\mathcal{B}(d)} x, l}\,\dagger \qquad (B.4)\; \frac{p, l \xrightarrow{\mathcal{B}(d)} x, l'}{\mathsf{token}(t) :: p, t :: l \xrightarrow{\mathcal{B}(d)} x, l'}$$

$$(B.5)\; \frac{[], l \xrightarrow{\mathcal{T}(d)} a, l' \quad p, l' \xrightarrow{\mathcal{B}(d)} x, l''}{\mathsf{simple} :: p, l \xrightarrow{\mathcal{B}(d)} \{a!\} :: x, l''} \qquad (B.6)\; \frac{[], l \xrightarrow{\mathcal{C}(d,\mathsf{false})} a, l' \quad p, l' \xrightarrow{\mathcal{B}(d)} x, l''}{\mathsf{compound} :: p, l \xrightarrow{\mathcal{B}(d)} \{a!\} :: x, l''}$$

$$(B.7)\; \frac{p, [] \xrightarrow{\mathcal{B}(d)} x, l}{\mathsf{optional} :: p, [] \xrightarrow{\mathcal{B}(d)} \{[]\} :: x, l} \qquad (B.8)\; \frac{[], l \xrightarrow{\mathcal{C}(\mathsf{literal}(]) :: d, \mathsf{true})} a, l' \quad p, l' \xrightarrow{\mathcal{B}(d)} x, l''}{\mathsf{optional} :: p, \mathsf{literal}([) :: l \xrightarrow{\mathcal{B}(d)} \{a\} :: x, l''}$$

$$(B.9)\; \frac{p, t :: l \xrightarrow{\mathcal{B}(d)} x, l'}{\mathsf{optional} :: p, t :: l \xrightarrow{\mathcal{B}(d)} \{[]\} :: x, l'} \quad (t \neq \mathsf{literal}([))$$

$$(B.10)\; \frac{[], l \xrightarrow{\mathcal{C}(t :: d, \mathsf{true})} a, l' \quad p, l' \xrightarrow{\mathcal{B}(d)} x, l''}{\mathsf{delimited}(t) :: p, l \xrightarrow{\mathcal{B}(d)} \{a!\} :: x, l''}$$

$$(C.1)\; a, [] \xrightarrow{\mathcal{C}(d,b)} a, []$$

$$(C.2)\; a, t :: l \xrightarrow{\mathcal{C}(t :: d, \mathsf{true})} a, l \qquad (C.3)\; a, t :: l \xrightarrow{\mathcal{C}(d,b)} a, t :: l \quad (t \text{ occurs in } d)$$

$$(C.4)\; \frac{a, t :: l \xrightarrow{\mathcal{T}(d)} a', l' \quad a', l' \xrightarrow{\mathcal{C}(d,b)} a'', l''}{a, t :: l \xrightarrow{\mathcal{C}(d,b)} a'', l''} \quad (t \notin d)$$

of a parsing function (the conclusion, below the line) in terms of the value of one or more recursive calls to other functions (the premises, above the line). Right arrows denote the action of parsing. Arrows are decorated with a label that identifies the parser along with its parameters, if any. The $\mathcal{T}$, $\mathcal{B}$, and $\mathcal{C}$ parsers have a parameter representing the list of delimiters in the order they are expected, with the head of the list being the first expected delimiter. The $\mathcal{C}$ parser also has a boolean parameter indicating whether the parser should or should not "eat" the delimiter when it is eventually met.

The root parsing function is $\mathcal{T}$. Given a delimiter $t \in \mathit{token}$ and a token stream $l \in \mathit{token}^*$ we have

$$[], l \xrightarrow{\mathcal{T}([t])} [n], l'$$

where $n \in \mathit{node}$ is the parsed term and $l' \in \mathit{token}^*$ is the part of the token stream that has not been consumed. Spaces are ignored when parsing terms and pre-parameters (rule $T.4$), but not when parsing post-parameters (rule $B.4$). The $\mathcal{A}$ function differs from the other parsing functions because by the time a macro with pre-parameters is encountered, pre-parameters have already been parsed. The lists $a \in \mathit{node}^*$ in the $\mathcal{T}$, $\mathcal{A}$, and $\mathcal{C}$ parsers represent the terms accumulated before the term being parsed. Note that pre-parameters are inserted at the end of the parameter list (rules $A.2$ to $A.4$) and that post-parameters are inserted at the beginning of the parameter list (rules $B.5$ to $B.10$). This way parameter nodes appear in the parse tree in the same order as in the original token stream (rule $T.5$).

4.1.1 Example

Given that the input buffer contains the TEX source shown in Fig. 1, the lexical analyzer would produce the following stream of tokens:

$$\begin{aligned} l_0 \stackrel{\text{def}}{=} [&\mathsf{control}_{\langle[],[\mathsf{delimited}(\mathsf{control}(\mathsf{egroup}))]\rangle}(\mathtt{bgroup}); \\ &\mathsf{literal}(\mathtt{1}); \mathsf{control}_{\langle[\mathsf{compound}],[\mathsf{compound}]\rangle}(\mathtt{over}); \\ &\mathsf{control}_{\langle[],[\mathsf{delimited}(\mathsf{control}(\mathsf{egroup}))]\rangle}(\mathtt{bgroup}); \\ &\mathsf{literal}(\mathtt{x}); \mathsf{literal}(\mathtt{+}); \mathsf{literal}(\mathtt{1}); \\ &\mathsf{control}(\mathtt{egroup}); \mathsf{control}_{\langle[\mathsf{simple}],[\mathsf{simple}]\rangle}(\mathtt{sp}); \\ &\mathsf{literal}(\mathtt{2}); \mathsf{control}(\mathtt{egroup})] \end{aligned}$$

By the application of the parsing rules given in Table 2 it can be shown that

$$[], l_0 @ [\mathsf{control}(\mathtt{eoi})] \xrightarrow{\mathcal{T}([\mathsf{control}(\mathtt{eoi})])} [n], [\mathsf{control}(\mathtt{eoi})]$$

where $n \in \mathit{node}$ is the same tree shown in Fig. 1 except that the g nodes are labeled with `bgroup`.

4.2 Error Recovery

Parsing functions are all total functions, they always produce a result, even when the input token stream is malformed. Unlike parsers of batch TEX converters or the TEX parser itself, there will often be moments during the editing process when the input buffer contains incorrect or incomplete markup, for example because not all the required parameters of a macro have been entered yet. The parser must recover from such situations in a tolerant and hopefully sensible way. We distinguish three kinds of situations: *missing parameters*, *pattern mismatch*, and *ambiguity*, which we examine in the rest of this section.

4.2.1 Missing Parameters

Consider an input token stream representing the sole `\over` macro with no arguments provided:

$$\begin{aligned} l_1 \stackrel{\text{def}}{=} [&\mathsf{control}_{\langle[\mathsf{compound}],[\mathsf{compound}]\rangle}(\mathtt{over}); \\ &\mathsf{control}(\mathtt{eoi})] \end{aligned}$$

It is easy to check that

$$[], l_1 \xrightarrow{\mathcal{T}([\mathsf{control}(\mathtt{eoi})])} \begin{array}{l} [\mathsf{macro}(\mathtt{over}, [\mathsf{empty}; \mathsf{empty}])], \\ [\mathsf{control}(\mathtt{eoi})] \end{array}$$

More generally the parser inserts empty nodes in the parsing tree wherever an expected parameter is not found in the token stream. This behavior can be seen in rule $A.2$ and also in rules $B.5$, $B.6$, and $B.10$ where the ! operator is used. For optional parameters an empty node list is admitted (rules $B.7$ and $B.8$).

The presence of empty nodes guarantees that the generated tree is structurally well-formed, which is crucial for the subsequent transformation phase. It also allows the application to give the user feedback indicating the absence of required parameters. In the example above, for instance, the application may display something like $\frac{\square}{\square}$ suggesting that a fraction was entered, but neither the numerator nor the denominator have been.

4.2.2 Pattern Mismatch

Rules $B.2$ and $B.3$ have been marked with a † to indicate that the parser expects a token which is not found in the token stream. In both cases the parser will typically notify the user with a warning message.

4.2.3 Ambiguities

In TEX one cannot pass a macro with parameters as the parameter of another macro, unless the parameter is enclosed within a group. For example, it is an error to write `\sqrt\sqrt{x}`, the correct form is `\sqrt{\sqrt{x}}`. Because we treat the left curly brace like any other macro, grouping would not help our parser in resolving ambiguities. However, the

parser knows how many parameters a macro needs, because the token representing the control sequence has been annotated with such information by the lexer. When processing a macro with arguments the parser behaves "recursively", it does not let an incomplete macro to be "captured" if it was passed as parameter of an outer macro. A consequence of this extension is that any well-formed fragment of TeX markup is accepted by our parser resulting in the same structure, but there are some strings accepted by our parser that cause the TeX parser to fail.

4.3 Incremental Parsing

Parsing must be efficient because it is performed in real-time, in principle at every modification of the input buffer, no matter how simple the modification is. Fortunately TeX markup exhibits good *locality*, that is small modifications in the document cause small modifications in the parsing tree. Consequently we can avoid re-parsing the whole source document, we just need to re-parse a small interval of the input buffer around the point where the modification has occurred, and adjust the parsing tree accordingly. Let us consider again the example of Fig. 1 and suppose that a change is made in the markup

`{1\over{1+x}^2}` $\Rightarrow$ `{1\over{1+x+y}^2}`

(a `+y` is added to the denominator of the fraction). To be conservative we can re-parse the smallest term within braces that includes the modified part (the underlined fragments). Once the term has been re-parsed it has to be substituted in place of the old term in the parsing tree.

In order to compute the interval of the input buffer to be re-parsed we annotate the nodes of the parsing tree with information about the first and the last characters of the buffer which were scanned while building the node and all of its children. A simple visit of the tree can locate the smaller interval affected by the modification.

Curly braces occur frequently enough in the markup to give good granularity for re-parsing. At the same time limiting re-parsing to braced terms helps control the costs related to the visit to the parsing tree and to the implementation of the incremental parsing and transformation machinery.

5 Transformation

The transformation phase recognizes structured patterns in the parsing tree and generates corresponding fragments of the result document. We have already anticipated that XSLT is a very natural choice for the implementation of this phase. Besides, XSLT stylesheets can be extended very easily, by providing new *templates* that recognize and properly handle new macros that an author has introduced.

We can see in Fig. 2 two sample templates taken from an XSLT stylesheet for converting the internal parsing tree into a MathML tree. Both templates have a preamble made of an `xsl:if` construct which we will discuss later in this section. Since the TeX tree and the MathML tree are almost isomorphic (Fig. 1) the transformation is generally very simple and in many cases it amounts at just renaming the node labels. Template (a) is one such case: it matches any node in the parsing tree with label `macro` and having the `name` attribute set to `over`. The node for the `\over` macro corresponds naturally to the `mfrac` element in MathML. The two parameters of `\over` are transformed recursively by applying the stylesheet templates to the first and second child nodes (`p[1]` means "the first `p` child of this node", similarly `p[2]` refers to the second `p` child).

Template (b) is slightly more complicated and shows one case where there is some change in the structure. For combined sub/super scripts TeX accepts a sequence of `_` and `^` no matter in what order they occur, but MathML has a specific element for such expressions, namely `msubsup`. The template matches an `sb` node whose first parameter contains an `sp` node, thus detecting a `...^..._...` fragment of markup, then the corresponding `msubsup` element is created and its three children accessed in the proper position of the parsing tree. A symmetric template will handle the case where the subscript occurs before the superscript.

5.1 Incremental Transformation

As we have done for parsing, for transformations we also need to account for their cost. In a batch, one-shot conversion from TeX this is not generally an issue, but in an interactive authoring tool a transformation is required at every modification of the parsing tree in order to update the view of the document.

Intuitively, we can reason that if only a fragment of the parsing tree has changed, we need re-transform only that fragment and substitute the result in the final document. This technique makes two assumptions: (1) that transformations are context-free; that is, the transformation of a fragment in the parsing tree is not affected by the context in which the fragment occurs; (2) that we are able to relate corresponding fragments between the parsing and the result trees.

```
<xsl:template
 match="macro[@name='over']">
 <m:mfrac>
  <xsl:if test="@id">
   <xsl:attribute name="xref">
    <xsl:value-of select="@id"/>
   </xsl:attribute>
  </xsl:if>
  <xsl:apply-templates select="p[1]"/>
  <xsl:apply-templates select="p[2]"/>
 </m:mfrac>
</xsl:template>
```

(a)

```
<xsl:template
 match="macro[@name='sb']
        [p[1]/*[1][self::macro[@name='sp']]]">
 <m:msubsup>
  <xsl:if test="@id">
   <xsl:attribute name="xref">
    <xsl:value-of select="@id"/>
   </xsl:attribute>
  </xsl:if>
  <xsl:apply-templates select="p[1]/*/p[1]"/>
  <xsl:apply-templates select="p[2]"/>
  <xsl:apply-templates select="p[1]/*/p[2]"/>
 </m:msubsup>
</xsl:template>
```

(b)

Figure 2: Example of XSLT templates for the transformation of the internal parsing tree into a MathML tree. MathML elements can be distinguished because of the `m:` prefix.

Template (b) in Fig. 2 shows one case where the transformation is not context free: the deeper `sp` node is not processed as if it would occur alone, but it is "merged" together with its parent. More generally we can imagine that transformations can make almost arbitrary re-arrangements of the structure. This problem cannot be solved unless we make some assumptions, and the one we have already committed to in Sect. 4 is that braces define "black-box" fragments which can be transformed in isolation, without context dependencies.

As for the matter of relating corresponding fragments of the two documents, we use identifiers and references. Each node in the parsing tree is annotated with a unique identifier (in our sample templates we are assuming that the identifier is a string in the `id` attribute). Templates create corresponding `xref` attributes in the result document "pointing" to the fragment with the same identifier in the parsing tree. This way, whenever a fragment of the parsing tree is re-transformed, it replaces the fragment in the result document with the same identifier.

More generally, back-pointers provide a mechanism for relating the view of the document with the source markup. This way it is possible to perform operations like selection or cut-and-paste that, while having a visual effect in the view, act indirectly at the content/markup level.

6 Conclusion

We have presented architectural and implementation issues of an interactive editor based on TEX syntax which allows flexible customization and content-oriented authoring. TEXmacs[2] is probably the existing application that most closely adopts such architecture, with the difference that TEXmacs does not stick to TEX syntax as closely as we do and that, apart from being a complete (and cumbersome) editing tool and not just an interface, it uses encoding and transformation technologies not based on standard languages (XML [2] and XSLT [3]).

Among batch conversion tools we observe a tendency to move towards the processing of content. The TEX to MathML converter by Igor Rodionov and Stephen Watt at the University of Western Ontario [8, 9] is one such tool, and the recent Hermes converter by Romeo Anghelache [10] is another. These represent significant steps forwards when compared to converters such as LATEX2HTML.[3]

A prototype tool called EdiTEX, based on the architecture described in this paper, has been developed and is freely available along with its source code.[4] No mention of MathML is made in the name of the tool to remark the fact that the architecture is very general and can be adapted to other kinds of markup. The prototype is currently being used as interface for a proof-assistant application where editing of complex mathematical formulas and proofs is required. In this respect we should remark that TEX syntax is natural for "real" mathematics, but it quickly becomes clumsy when used for writing terms of programming languages or λ-calculus. This is mainly due to the conventions regarding spaces (for instance, spaces in the λ-calculus denote function application) and identifiers (the rule "one character is one identifier" is fine for mathematics, but not for many other languages). Note however that, since the lexical analyzer is completely separate from the rest of the architecture, the token stream being its interface, it can be easily targeted to a language with different conventions than those of TEX.

[2] http://www.texmacs.org/

[3] http://www.latex2html.org/

[4] http://helm.cs.unibo.it/software/editex/

The idea of using some sort of restricted TeX syntax for representing mathematical expressions is not new. For example, John Forkosh's MimeTeX[5] generates bitmap images of expressions to be embedded in Web pages. However, to the best of our knowledge the formal specification of the parser for simplified TeX markup presented in Sect. 4 is unique of its kind. A straightforward implementation based directly on the rules given in Table 2 amounts at only just 70 lines of functional code (in an ML dialect), which can be considered something of an achievement given that parsing TeX is normally regarded as a hard task. By comparison, the parsing code in MimeTeX amounts to nearly 350 lines of C code after stripping away the comments.

One may argue that the simplified TeX markup is too restrictive, but in our view this is just the sensible fragment of TeX syntax that the average user should be concerned about. In fact the remaining syntactic expressiveness provided by TeX is mainly required for the implementation of complex macros and of system internals, which should never surface at the document level. By separating the transformation phase we shift the mechanics of macro expansion to a different level which can approached with different (more appropriate) languages. Since this mode of operation makes the system more flexible we believe that our design is a valuable contribution which may provide an architecture for other implementers to adopt.

References

[1] The Unicode Consortium: The Unicode Standard, Version 4.0, Boston, MA, Addison-Wesley (2003). http://www.unicode.org/

[2] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler (editors): Extensible Markup Language (XML) 1.0 (2nd Edition), W3C Recommendation (2000). http://www.w3.org/TR/2000/REC-xml-20001006

[3] James Clark (editor): XML Transformations (XSLT) Version 1.0, W3C Recommendation (1999). http://www.w3.org/TR/1999/REC-xslt-19991116

[4] Ron Ausbrooks, Stephen Buswell, Stéphane Dalmas, Stan Devitt, Angel Diaz, et al.: Mathematical Markup Language (MathML) Version 2.0 (2nd Edition) W3C Recommendation, (2003). http://www.w3.org/TR/2003/REC-MathML2-20031021/

[5] http://www.ctan.org/tex-archive/support/mimetex/

[5] Donald E. Knuth: The TeXbook, Addison-Wesley, Reading, MA, USA (1994).

[6] Leslie Lamport: A Document Preparation System: LaTeX, Addison-Wesley, Reading, MA, USA (1986).

[7] Douglas Lovell: TeXML: Typesetting XML with TeX, *TUGboat*, 20(3), pp. 176–183 (September 1999).

[8] Sandy Huerter, Igor Rodionov, Stephen M. Watt: Content-Faithful Transformations for MathML, Proc. International Conference on MathML and Math on the Web (MathML 2002), Chicago, USA (2002). http://www.mathmlconference.org/2002/presentations/huerter/

[9] Stephen M. Watt: Conserving implicit mathematical semantics in conversion between TeX and MathML, *TUGboat*, 23(1), pp. 108–108 (2002).

[10] Romeo Anghelache: LaTeX-based authoring tool, Deliverable D4.d, MoWGLI Project (2003). http://relativity.livingreviews.org/Info/AboutLR/mowgli/index.html

Appendix: The TML DTD

```
<!ENTITY % TML.node "
  empty|space|literal|macro">
<!ENTITY % TML.common.attrib "
  id         CDATA #IMPLIED
  xref       CDATA #IMPLIED
  start      NMTOKEN #IMPLIED
  end        NMTOKEN #IMPLIED">
<!ELEMENT empty EMPTY>
<!ATTLIST empty %TML.common.attrib;>
<!ELEMENT space EMPTY>
<!ATTLIST space
  %TML.common.attrib
  name    NMTOKEN #IMPLIED
  literal CDATA #IMPLIED>
<!ELEMENT literal #PCDATA>
<!ATTLIST literal
  %TML.common.attrib;
  name    NMTOKEN #IMPLIED>
<!ELEMENT macro (p)*>
<!ATTLIST macro
  %TML.common.attrib;
  name    NMTOKEN #REQUIRED
  literal CDATA #IMPLIED>
<!ELEMENT p (%TML.node;)*>
<!ATTLIST p %TML.common.attrib;>
```