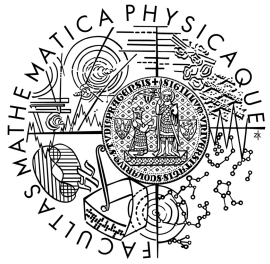
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



David Holaň

Editor of mathematical formulas

Department of Applied Mathematics

Bachelor thesis supervisor: Mgr. Vít Jelínek, Ph.D.
Study field: Computer science, Programming

2009

I would like to thank the supervisor of this thesis, Mgr. Vít Jelínek, Ph.D., especially for proposal of the subject of the thesis and for pointing out many awkward details of the user interface of the program.

Prohlašuji, že jsem svou bakalářskou práci napsal(a) samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 28. května 2009 David Holaň

Contents

Název práce: Editor matematických vzorců
Autor: David Holaň
Katedra (ústav): Katedra aplikované matematiky
Vedoucí bakalářské práce: Mgr. Vít Jelínek, Ph.D.
e-mail vedoucího: jelinek@kam.mff.cuni.cz

Abstrakt: V předložené práci studujeme návrh a implementaci "MaEd for LaTeX", přenositelného programu s grafickým uživatelským rozhraním. Program je určen na vytváření a upravu LaTeXových vzorců. Program je navržen tak aby začátečník mohl vytvořit i složité vzorce bez znalosti LaTeXového zdrojového kódu. Uživatel také může importovat vlastní zdrojový kód, přičemž program zbytečně nemění importovaný kód, například odstraněním komentářů, nebo odražení. Návrh popisuje, jak bylo navrženo uživatelské rozhraní programu. LaTeXové příkazy dostupné k vytváření matematických vzorců jsou popsány i s jejich syntaxí a úrovní podpory v programu. Práce dále analyzuje strukturu souboru LaTeXového zdrojového kódu.

Klíčová slova: LaTeX, editor, WYSIWYG, vzorce, rovnice

Title: Editor of mathematical formulas
Author: David Holaň
Department: Department of Applied Mathematics
Supervisor: Mgr. Vít Jelínek, Ph.D.
Supervisor's e-mail address: jelinek@kam.mff.cuni.cz

Abstract: In the present work we study the concept and implementation of "MaEd for LaTeX", a portable graphical user interface program for creating and editing of LaTeX formulae. The program is designed to allow a beginner to create even complex formulae without any knowledge of underlying LaTeX source code. The user can also import his own source code and the program doesn't make unnecessary changes to the imported source code, like removing comments and indentation. The concept describes how was the program's GUI designed. The LaTeX commands available for creation of math formulae are described along with their syntax and level of support in the program. The work also analyzes structure of LaTeX source code.

Keywords: LaTeX, editor, WYSIWYG, formulae, equations

Chapter 1

Introduction

LaTeX ('lay-tech' or 'lah-tech') is a system of macros for more abstract and easier use of the TeX ('tech') typesetting program.TeX is intended and succeeds as a platform for writing documents containing a lot of mathematic expressions, but the available commands are often too low-level. LaTeX was created to address that, it introduces many macros that allow user to type good looking and well-structured documents. But it still lacks built-in graphical user interface, as does TeX . The user has to learn a lot of commands to create a typical document. Many of these commands are used solely to create mathematical formulae. A formula of average complexity often leads to cumbersome LaTeX code containing many nested parentheses and even an experienced user then finds such code hard to comprehend, although it is well structured.

The aim of this thesis is to document the concept and implementation of "MaEd for LaTeX ", a portable GUI program for creating and editing of LaTeX formulae. The program is designed to satisfy three key objectives. First, a beginner can create even complex formulae without any knowledge of underlying LaTeX source code. Second, user can import his own source code. Third, the program doesn't make unnecessary changes to imported source code, like removing comments and indentation. Existing software often satisfies the first and second objectives, but not the third. The program supports LaTeX 2_εversion of LaTeX.

LaTeX supports definition of user commands, these can be bundled into packages. Parsing and understanding of previously unknown package would be very difficult, if not impossible, so MaEd can support only a limited set of commands. It has been decided that this set is to contain majority of

math commands documented in Oetiker, Partl, Hyna, Schlegl (2008) and a few additions, which mainly improve compatibility with older code.

1.1 Related works

MathType is an equation editor, which can export LaTeX code. Generated code optionally contains commentary with MathType's internal representation, which allows MathType to import such code. A code without MathType's internal representation can't be imported.

TeXmacs is a word processor based on its own TeXmacs format. TeXmacs supports embedded equations. LaTeX code can be imported and exported. TeXmacs doesn't have graphical representation of some LaTeX commands such as `\left` and user indentation is removed.

LyX is a word processor similar to TeXmacs and shares both limitations of not having graphical representation of some LaTeX commands (for example superscript of superscript) and removing user indentation.

Scientific Word and Scientific WorkPlace are word processors with equation support. Both are commercial products. Scientific Word is a subset of Scientific WorkPlace offered at a reduced yet still high price. Both are capable of editing whole LaTeX document including equations. The GUI is similar to the one of LyX and it seems to support LaTeX much better than previous programs, though it ignores double script escape sequences.

Chapter 2

Analysis and Concept

2.1 LaTeX

In order to use LaTeX , one must type a LaTeX source code file and compile it using the LaTeX program into a DVI file. The DVI file can be then converted into postscript file and printed. Alternatively, one can use pdflatex program instead of latex program to create a PDF document. LaTeX source code file is a plain text file and describes the typesetting of the whole document. During compilation, LaTeX is either in text mode or math mode. There are many differences between math and text mode; for example, some commands work only in one mode. Text mode is used to typeset text such as paragraphs and math mode is used to typeset math such as equations or formulae. By default, the source code is compiled in text mode. When the user wants to insert an equation or other math, he starts by typing a math mode entering code, which is a math-shift character (usually `$`) or a command such as `\begin{equation}` or `\[` . This causes LaTeX to enter the math mode. When the user is done typing the math he terminates the math mode by corresponding termination code. This causes LaTeX to shift back into text mode. The code between math mode opening and terminating code, including the bounds will be referred in this document as math block. There is another type of block, a building block. A building block is a single entity such as symbol or matrix. For example $\frac{a}{b} = c$ is a sequence of $\frac{a}{b}$, $=$ and c. Building blocks are used to form the content of a math block. Most building blocks can have a superscript and a subscript. Any building block with exception of parametric commands can be set as an index. Because math is typeset exclusively in math mode, MaEd doesn't need to know anything

about text mode.

2.2 Program's GUI

MaEd's GUI features a main menu bar, and two editor windows. The lower editor is the code editor, as it displays a LaTeX source code. The upper editor is the math editor, as it displays editable WYSIWYM ("what you see is what you mean") representation of a math block.

The code editor is the more important one. When the user decides to open a file, the file's content is copied into the code editor. When the user decides to save changes to a file, it's the content of the code editor that is saved. The math editor assumes a support role. The code editor can contain a complex source code file containing text and multiple math blocks. The "jump to next" button causes the first math block following the current text cursor position to be selected. The "apply" button causes the selected math block to be parsed and shown in the math editor. The code editor also features standard history, clipboard and current line/character counter implementation.

The math editor allows the user to take advantage of GUI while editing a math block. Changes to the math block can be reflected in the code editor either on-demand by clicking the "apply" button or automatically by checking the "auto apply" check button. There are three output code flavors available in the option list near the "apply" button: The "user" flavor with user comments and indentation, the "structured" flavor with machine-generated hierarchic indentation and the "compressed" flavor without any indentation and comments. The math block display has a cursor, which can be navigated by directional keys or put near the mouse cursor by a mouse click. The math cursor can be also used to select a sequence of building blocks by holding the shift key while pressing a directional key. Apart from the math block display, the math editor has a common toolbar and a inserter toolbar, which is dedicated to inserting of building blocks. When the user clicks on a block in the inserter toolbar, it is inserted at the current math editor cursor position. Some blocks are containers, for example the fraction. Containers have one or more contents, for example, the fraction has the numerator and denominator contents. Building blocks can be inserted into any content of a container and the math editor cursor can be moved inside. When a container is inserted and a sequence of blocks is selected, then the selected sequence is moved into the primary content of the inserted container. The math editor

represents indices as contents. When code is generated, each index content is encapsulated by curly braces as needed. (Curly braces are a container building block.) Clicking on "sup" or "sub" toolbar button or pressing keyboard shortcut control-up or control-down moves math cursor into superscript or subscript of the block on the left side of cursor if applicable. Some building blocks have a context menu that can be accessed by right clicking on them. The math block also has a context menu. Pressing an alphanumeric or a punctuation key inserts the pressed symbol at the position of math cursor. Deletion by pressing the backspace and the delete key is also available. The math editor also features standard history and clipboard implementation.

The main menu bar has the "file", "options" and "help" submenus. The "file" submenu offers the "open", "save" and "exit" actions. The "options" submenu allows user to configure the program by several dialogs. The modified configuration is saved into configuration files upon confirmation. The user can configure symbols, function names, glyph aliases and fonts. The "symbol options" configuration dialog allows user to define a symbol that the program doesn't know, see section 2.3 on page 13 for description of symbols. The "function options" configuration dialog does the same for function names, see section 2.3 on page 14 for description of function names. The "font configuration" dialog allows user to specify path to type 1 font files ("pfb" extension) used for displaying of the math block. The "glyph options" configuration dialog allows assigning a physical glyph from some font to a glyph alias. Glyph aliases are used to draw all glyphs, instead of hard-coding font and index.

2.3 Supported commands

This subchapter explains the commands documented by Oetiker, Partl, Hyna, Schlegl (2008) - how and if they are supported. Note that commands that are only for text mode are not mentioned, as they are outside of the scope of MaEd. Commands supported but not mentioned by Oetiker, Partl, Hyna, Schlegl (2008) are explained too.

MaEd can support commands in four levels. The first level is parseability; the program does parse the command as a valid code. The second level is internal representation; some commands are parsed but converted into a different command with equal meaning. Third level is GUI representation; the command's effect is visible in the math editor. Fourth level is GUI reachability; the GUI representation of the command can be created by GUI controls

and subsequently generated source code contains the code of the command. Unspecified level of support implicitly means the fourth level.

All code enclosed by `\lmeignb` and `\lmeigne` is treated as a comment, this is extra feature known only to the program. It allows an expert LaTeX user to import a code containing unsupported commands. The preamble of the LaTeX source code can include a simple definition to make LaTeX ignore both `\lmeignb` and `\lmeigne` but not ignore the code between: `\newcommand{\lmeignb}{}\newcommand{\lmeigne}{}`

Math blocks

The most important commands are those creating math blocks. Oetiker, Partl, Hyna, Schlegl (2008) document `$` and environments "equation", "equation*", "align" and "align*". Environment means that syntax
`\begin{<env. name>}<content>\end{<env. name>}`
is used. Additional math environments are mentioned but not documented so they are not supported. Legacy plain TeX `$$` and environments "eqnarray", "eqnarray*" have support of level one for backward compatibility. Math blocks can be distinguished by several properties. The style influences how are the math block and its content typeset: A text style math block is typeset as part of the text paragraph and a display style math block is typeset apart from the rest of the paragraph and centered. Some symbols are also larger in display style. The `$` math block is the only text style math block. Some display style math blocks enumerate equations. Those are the "equation", "eqnarray" and "align". Starred versions don't enumerate equations. The "eqnarray" and "align" environments can contain multiple equations. All supported math blocks, corresponding codes and properties are in the math block table.

Math blocks not supporting multiple equations contain a sequence of building blocks. Math blocks supporting multiple equations contain a tabular code. For detailed explanation of tabular code see section 2.3 on page 17 (Common code constructs). The tabular code of math blocks can't use `\multicolumn` , `\hline` and `\cline` . Each equation of "eqnarray" consists of three fields, first is right justified, second is centered and last is left justified, fields can be omitted but extra field leads to an invalid code. Each equation of "align" consist of arbitrary number of fields, odd fields are right justified, even fields are left justified, fields can be omitted too. This looks like multiple equations in one row but they are enumerated as one. A row

Table 2.1: Math blocks

Opening code	Closing code	Style	Eqn. enum.	Mult. eqn.
$	$	text	no	no
\begin{equation}	\end{equation}	display	yes	no
\begin{equation*} or \]	\end{equation*} or \]	display	no	no
\begin{align}	\end{align}	display	yes	yes
\begin{align*}	\end{align*}	display	no	yes
$$	$$	display	no	no
\begin{eqnarray}	\end{eqnarray}	display	yes	yes
\begin{eqnarray*}	\end{eqnarray*}	display	no	yes

code can also contain \nonumber , \tag{...} and \label{...} commands, these commands can be placed anywhere, even nested, in the row code. The \nonumber command disables equation enumeration. The \tag command causes the equation to be tagged by the argument instead of by the enumeration. The \label command defines an alias, which can be referenced later, even outside of the math block. Support for these commands is limited only to the beginning of the row code and to support level two.

The "split" environment allows multiple equations to behave as a single equation. It is not documented by Oetiker, Partl, Hyna, Schlegl (2008), although it is mentioned. The program supports it only to allow user to create multiple equations with one shared number. This is done by inserting split into the "equation" math environment. The program understands resulting code but other LaTeX valid usages of "split" are not recognized.

Building blocks

As mentioned previously, most building blocks can have a superscript and a subscript. From source code point of view, superscript character ^ and subscript character _ cause the following building block to be typeset as the respective index of previous building block. "a^b_c" or "a_c^b" would typeset as a_c^b. Commands with at least one argument can't be an index due to a LaTeX limitation; also one building block can't have multiple indices. These limitations are easily circumvented by enclosing the desired content by curly braces. Curly braces are the most essential container building block.

Table 2.2: Space commands

Space command	Width in proportion to the width of \quad
\,	3/18
\:	4/18
\;	5/18
\ "	N/A, width is comparable to the interword spacing
\quad	1
\qquad	2

They allow to typeset constructions like a^{bc} (a^{bc}), a^{b^c} (a^{b^c}) or $a^{\frac{a}{b}}$ a^{\frac bc} . When building blocks from the math editor are converted into source code, they automatically add code for subscript character, superscript character and curly braces if and only if needed. Note that command argument is considered to be content of command, source code \frac ab^c would be typeset as $\frac{a}{b}^c$.

Non-container building blocks

Symbols are the most essential building blocks. Many ASCII characters, including all alphanumeric characters, are typeset as symbols unless they are part of a command. Other symbols are typeset by symbol commands. All symbols in tables in Oetiker, Partl, Hyna, Schlegl (2008) without tag "ams" are supported. Some symbols, such as big operators, use different glyph in text and display mode. The program also offers a symbol options dialog, where the user can add support of a new symbol commands by typing command name and selecting a glyph from a font file.

Some symbols are delimiters. They are listed in the delimiter table in Oetiker, Partl, Hyna, Schlegl (2008), unfortunately the large delimiter group is not supported. Delimiters can be typeset with a \big command and its size variations. The \big command has one argument, the delimiter. Following size variations exist, in ascending order: \big , \Big , \bigg , \Bigg .

There are three groups of spaces, spaces proportional to the current font size, spaces with explicit size and spaces with size of their content. The first is group is described in the space commands table.

The \hskip and \hspace commands define a space with an explicit size described by a number and a unit. The \hskip command originates

from TEX and the only difference is that the \hspace command expects size argument to be enclosed by curly braces; this was probably added for increased safety over the TEX legacy \hskip . The \phantom command typesets an empty space of the size of its argument. The \hphantom and \vphatom variations typeset the empty space with zero height and width respectively. The \hskip , \hspace , \phantom , \hphantom and \vphantom commands have support of level three. There are also the \vskip and \vspace vertical only variations but they behave erratically, so they are not supported. Since the supported variations are most of the time used for fine-tuning of LATEX output, a task not suited for a WYSIWYM editor, users shouldn't mind GUI unreachability.

Function names are commands that typeset a function-like name using a proper font. All function names documented by Oetiker, Partl, Hyna, Schlegl (2008) are supported and the "function options" dialog can add support of previously unknown names. This can be also used to hot-fix the missing support of a building block command without parameters.

Style commands change the style of the following building blocks. The effect is limited to the current building block sequence. There are four styles. Display and text style were already mentioned, the other two are script and scriptscript style. Style commands don't influence the math block itself, only its content. Display style typesets some operators larger and some symbols and function names have indices positioned above and below. Script style uses smaller fonts and scriptscript style uses even smaller fonts.

The \text command typesets its content in text mode. Since apart this command, the program doesn't need to know anything about text mode, supporting it would mean implementing a text mode parser, which would be inefficiently spent development time. The program rather skips its argument by taking advantage of proper nesting of curly braces in both modes, resulting in a support of level three.

Container building blocks

Some building blocks may contain a sequence of building blocks. Such building blocks are called 'containers'. The argument, which specifies content, is often required to be one building block and not a command with argument(s). This limitation can be circumvented by curly braces. Some containers have multiple contents.

Single content containers

Curly braces are the most essential container. Their content is a sequence of building blocks both in GUI representation and source code representation. They don't do anything special apart from grouping their content into a single building block.

The common property of many following single content containers is that their source code is composition of command code and an argument, where the argument describes the content and is in a form of a building block.

Typeface commands make letter and digit symbols to be typeset using the specified typeface. Note the difference between typesetting character and character of a symbol; letters typeset by function name command don't use the specified typeface but function name typeface. A table in Oetiker, Partl, Hyna, Schlegl (2008) lists all available typeface commands.

The `\boldsymbol` command typesets its content bold. Unlike typeface commands, this command does affect all glyphs. Implementation would require a significant amount of time for one minor feature, so this container's GUI representation only indicates boldness of its content by bold lines on sides.

For the `\phantom` command and its variations see section 2.3 on page 14, as they were described along other space commands.

Accent commands display an accent above their content. A table in Oetiker, Partl, Hyna, Schlegl (2008) lists all available accent commands.

Decoration commands display a decoration above their content. They are very similar to accent commands, the difference is that the decoration can match its size to the size of the content.

The `\not` command typesets its content crossed out to negate it. The content usually is a single operator symbol.

The `\pmod` command typesets $(\mathrm{mod}\ x)$, where `x` is the content. The parentheses don't match size of the content unlike the `\left` command below.

The `\left` command typesets specified delimiters around its content. Its syntax is `\left<delimiter><content>\right<delimiter>` . Since the content is delimited by `\right` , it can be an ungrouped sequence of building blocks. The delimiters are typeset so that their size matches the size of the content. Many delimiters have a telescopic version that can grow to any size, when the size of the content is too big for available fixed size glyphs of the delimiter. A full stop can be can be typed instead of a delimiter to typeset no

delimiter. The list of available delimiters is in the delimiter table in Oetiker, Partl, Hyna, Schlegl (2008), note that the big delimiters are not supported.

Multiple content containers

Containers with multiple contents can be divided into two distinct groups, simple containers, which specify contents with arguments, and tabular containers, which use some form of tabular code and can contain arbitrary number of contents.

simple containers

The `\sqrt` command typesets radical sign together with a horizontal bar extending from its top to the end of the radicand. The index is typeset in usual position. The syntax is `\sqrt[<index>]<radicand>` . The `[<index>]` argument is optional, if omitted, no index is typeset. The program also supports a plain TEX `\root` command that has syntax:
`\root<index>\of<radicand>`
The `\root` command is automatically converted into `\sqrt` , so the support is of level one.

The `\frac` command typesets a fraction. The syntax is:
`\frac<numerator><denominator>`
The `\tfrac` and `\dfrac` variations exist with subtle differences in sizing. These differences are not shown in the program and variations are not reachable from GUI.

The `\stackrel` command typesets the first argument above the second argument. The result looks like a `\frac` without the fraction bar, although it is positioned on the line differently.

The `\binom` typesets a binomial. Its parentheses match the size of the content.

tabular containers

See section 2.3 on page 17 (Common code constructs) for explanation of tabular code, table spec and `\multicolumn` .

The first described tabular container is the `\substack` command. Its tabular code is limited to one column. In other words only one field per line and no `\multicolumn` are allowed. The tabular code is the only argument.

Matrix environments typeset a matrix. All columns are centered. The column count and the line count are determined from the tabular code. The syntax is: `\begin{<name>}<tabular code>\end{<name>}` . The name can be "matrix", "pmatrix", "bmatrix", "Bmatrix", "vmatrix" or "Vma-

trix". The variations with extra letter typeset delimiters around the matrix, the delimiter type corresponds to the letter. Matrix commands allow \multicolumn .

The \array environment typesets a more customizable array than matrix commands. The syntax is:
\begin{array}<table spec><tabular code>\end{array}
The \multicolumn command is supported. The column count and justification is specified by the table spec.

Common code constructs

The table spec is a sequence of | , r , c , and l characters. The letters specify a column with right/center/left justification. The pipe character specifies a line between adjacent columns; multiple lines between same adjacent columns can be used. Tabular code is a sequence of row codes separated by \\ . Each row code is a sequence of outline code followed by a sequence of fields separated by & . Field code is a sequence of building blocks or the \multicolumn command. The row separator has an optional argument [<n><unit>] that specifies how much space should be between rows. The optional argument has support of level three. Note that sequence of building blocks is a valid single row single field tabular code.

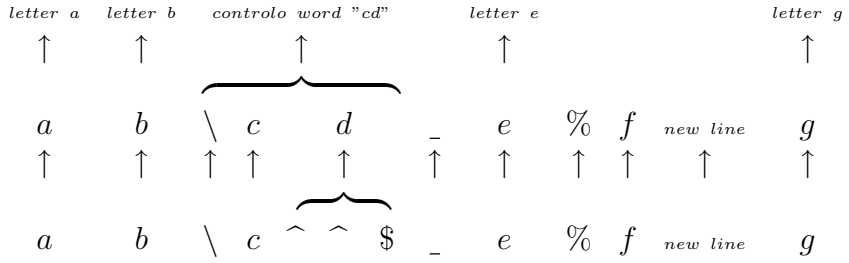
The \multicolumn command specifies a field that spans over one or more columns. It has syntax:
\multicolumn{<column span>}{<table spec>}{<argument>}
Where column span is a natural number. The table spec is limited to one column for obvious reasons, note that | characters are allowed.

Outline code is a sequence of \hline and \cline commands. The \cline command has syntax \cline{<x>-<y>} a and typesets an outline between previous row and current row spanning from column number x to column number y. The \hline command is the same as \cline , except it spans entire row. Multiple \hline commands cause multiple outlines to be typeset.

2.4 Configuration files

Program's configuration is loaded from plain text configuration files. The main configuration file (default "main.cfg") contains paths to other configuration files. Symbol table configuration file (default "symboltable.cfg")

Figure 2.1: TEX parser layers



contains definitions of symbols known to the program. The file also defines groups of symbols displayed in the inserter toolbar of the math editor. Font map configuration file (default "fontmap.cfg") assigns built-in and custom font ids to a font file. Character map configuration file (default "charmap.cfg") assigns built-in and custom glyph aliases to a pair of a font id and glyph index. Function list configuration file (default "functionlist.cfg") defines all known function names.

2.5 In-depth latex

In previous subchapters it has been described how do various commands influence latex output, but to truly understand the mechanics of latex program, one must start on the lowest level - sequence of ASCII characters. LATEX is essentially a system of TEX macros, latex program simply adds these macros to the beginning of the input file and the result is sent to tex program. Therefore the lowest level is tex program.

TEX

As explained by Knuth (1984). TEX reads input file in a few steps. First step translates input character codes to TEX internal representation, which is effectively ASCII. Second step replaces escape sequences. Third step translates characters to tokens. TEX then finds out the meaning of tokens. First step is trivial because MaEd reads input as ASCII characters too.

Escape Sequences

Table 2.3: TEXcharacter categories

Category	Meaning	Character list in default LATEX
0	Escape Character[1]	\
1	Beginning of group	{
2	End of group	}
3	Math shift	$
4	Alignment tab	&
5	End of line	New line character
6	Parameter	#
7	Superscript	^
8	Subscript	Underscore character
9	Ignored character	
10	Space	Space, Tab character
11	Letter	Letters
12	Other character	Characters not assigned to other category
13	Active character	
14	Comment character	%
15	Invalid character	Delete character

[1] *Escape character is not related to escape sequences discussed before.*

Escape sequences allow a user to type characters not present on his keyboard. Nowadays, they are not so needed anymore. Escape sequences are replaced by a single character. They consist of two ^ characters and either two hexadecimal digits xy or one arbitrary character z.

The first escape sequence type is replaced by a character with internal code of value of hexadecimal number xy.

The second escape sequence is more complex. Let val(z) be TEX internal code of z. If val(z) is between 0 and 63 then the internal code of replacement character is equal to val(z) + 64. If val(z) is between 64 and 127, then the internal code of replacement character is equal to val(z)-64.

Translation to tokens

A token is a character with attached category or a control sequence. TEX classifies input characters into 16 categories:

Character's category is subject to change by the TEX primitive \catcode .

However, it is not common practice inside math blocks so MaEd uses standard LaTeX category codes. TeX is always in one of three input states: N, M or S. N means beginning of a new line. M means middle of a line and S means skipping blanks. The following text will explain each category and input state possibility.

A character of category 0 (escape) followed by a character of category 11 (letter) causes TeX to read entire word and create a control word token. Escape character followed by non-letter causes TeX to read the non-letter and create a control character. If the second character was a space or a letter, TeX sets input state to S, otherwise to M.

A character of category 1,2,3,4,6,7,8,11,12 or 13 causes to TeX to create a character token with attached category and set input state M.

A character of category 5 (end of line) translation: If encountered in input state N, TeX creates control word "\par", which means end of paragraph. If encountered in input state M, TeX to creates a character token with internal character code 32 and category space. If encountered in input state S, no token is created. Resulting input state is N for all cases.

A character of category 9 (ignored) is ignored.

A character of category 10 (space) is ignored in input states N and S. In input state M, TeX creates a character token with internal code 32 and category space, resulting state is S.

A character of category 14 (comment) is ignored with the rest of input line, including end of line character.

A character of category 15 (invalid) is ignored and a warning is printed.

Parsing tokens

After obtaining sequence of tokens, TeX proceeds with macro expansion. MaEd does not perform any macro expansion, because it would lose high-level information necessary for GUI editing.

Chapter 3

Implementation

The program is written in C++, the author's favorite language. The program doesn't use any system specific resources so the portability is limited only by availability of used libraries. MaEd has been tested under Windows XP and Debian operating systems.

The program uses a naming convence, which could be usefull to the reader: Names of all classes begin with an uppercase letter, while names of all class instances, functions and methods begin with a lowercasse letter.

3.1 Libraries

Several libraries were needed in order to implement required features. A toolkit was needed for program's GUI, LaTeX fonts had to be somehow drawn and a portable way of representing file path was needed.

GTK+ is a GUI toolkit, it's the only portable and free toolkit which author had some experience with. It is documentation by GTK+ team, the contributors (2009). GTK+ consists of several component libraries, such as GDK, GLib or GObject.

FreeType is a free and portable library for rendering of various fonts. It is used to render fonts used by MaEd. It allowed MaEd to use fonts accompanying LaTeX installation and thus to render glyphs with the same fonts as in the LaTeX output.

Boost library is used to portably represent file paths.

GHADev is author's library. It is most notably used for implementation of a smart pointer *GHAPointer::AutoPointer*, a list of smart pointers *GHAPointer::AutoPointerList* and improved enumerator *GHAEnumerator::ExtendableEnum*.

3.2 Main architecture

The program is using an object model. An overview of the object hierarchy is available in appendix B. A more in-depth description follows.

LatexMathEditor

The root of the object model is the LatexMathEditor class representing the whole program. LatexMathEditor contains following properties:

version property is the version of the program.

file property is the object describing the opened file.

config property is the object containing the current configuration.

callData property is the list of *LMECallData* used to connect signal handlers to gtk signals.

temporaryPopUpData property is a smart pointer containing the pop-up context menu when needed.

editors property is the object containing both the math editor and the code editor data.

The rest are various pointers to the relevant program's GUI elements.

LMECallData

LMECallData is a specialization of *CallData* class. This class is used to wrap GTK+ *g_signal_connect* function that uses a void pointer as a pointer to custom data for the handler. A void pointer can be easily initialized with wrong data, so it is better hidden inside the *CallData*. *CallData* contains a pointer to an object, a pointer to object's method and a pointer to method's parameter. These three properties make up a method call, which is executed

by *CallData*'s method *call*. *CallData* also has method to connect it to a most common signal signature.

config

The *LatexMathEditor*'s *config* property contains:

symbolTable property is the class containing definitions of currently known symbols loaded from the symbol table configuration file.

functionNameList property is the class containing the list of currently known function names loaded from the function name list configuration file.

characterMap property is the class containing the map from glyph alias to a pair of font id and glyph index, loaded from the character map configuration file.

rootFontManager property is the class containing a map from font id to a font file path or name along with a font buffer containing already loaded fonts.

records property is describing location of secondary configuration files.

All classes loaded from a configuration file include data necessary to modify the configuration and save the modified configuration to the file. More detailed description can be found in section 3.4 on page 30.

editors

The *LatexMathEditor*'s *editors* property contains two properties: *math* property and *code* property.

math

The *editors*'s *math* property represents the math editor. It has following properties:

tools property contains pointers to the common toolbar and its buttons.

inserters property contains pointers to the GTK+ widgets of the inserter toolbar and separate LMECallData lists for function names and symbols, because these can be changed during run-time.

cursor property contains current location, color and current timer id of the cursor of the math editor. The color is alternating between visible and invisible. When visible, the cursor inverts the color below. Upon movement, the cursor is visible and the blink interval is reset. The reset is done by creating a new timer and updating the current timer id. When cursor timer's interval of time has passed, it calls a handler, if the timer's id isn't the same as current timer id, the handler terminates the timer, otherwise, the handler flips cursor color and redraws cursor.

document property is the root of the expression system and contains a *TopLevelEnvironment* class representing the edited math block, see the *IExpression* interface below.

history property represents the history of changes. Each step of history contains a full copy of the document and the location of math editor cursor at that time. The backstepCount property indicates how many steps back have been done and thus how many forward steps are available.

isActive switch can turn off drawing of the math block in case of a drawing error.

shallRedrawCanvas switch can indicate that the math block has changed. If it has, the math block is painted first on the buffer, else the buffer is not repainted, saving a lot of computations. The buffer is rendered on the math block display and the math cursor is added.

code

The *editors*'s *code* property represents the code editor. It has following properties:

tools property contains pointers to GTK+ widgets relevant to the toolbar.

history property is similar to the math editor's history. The history steps contain ASCII text instead of document and the editedEnvironment property keeps bounds of the selected math block.

lock stops adding history steps during compound changes.

IExpression

The math editor creates a tree. The root of the tree is the *LatexMathEditor.editors.math.document*. The document has one child, the math block. The document's purpose is to be the link between the *LatexMathEditor* and the math block. The rest of the nodes of the tree are building blocks. The building blocks inside the content of a math block node or container building block node are the children of the node. A node of the tree will be referred as an "expression". All expressions must implement the *IExpression* interface. Most expressions are derived from class *Expression*, this class contains default implementations of many *IExpression*'s virtual methods. *IExpression* contains these subinterfaces:

ISon describes methods used to access the parent of the expression in the expression tree.

IRenderer describes the methods needed to draw the expression (*draw*) and to query the screen size (*getSize*). These methods are used to display math block in the math editor. Containers need to query the sizes of their children before they can draw themselves or compute their own size. When the whole expression is rendered, it is necessary to compute size of all the expressions from bottom towards the top, and then position all the blocks, from the top towards the bottom. To avoid unnecessary recomputations, the method *enableSizeBuffering* initiates the buffering, and the method *disableSizeBuffering* disables it at the end of the drawing operation. The size object returned by the *getSize* method contains usual height and width properties and the distance between the top edge and baseline of the block.

INameToSymbol and IFunctionNameListing describe methods used to access the respective configuration dependant data stored in the *LatexMathEditor.config* object. The implementation forwards queries to parent, until program's *document* node is reached, which then asks program's *config*. This mechanism could be optimized by accessing the configuration data directly. But currently, the performance is not an issue.

IFontManager describe methods used to obtain a font. The *getFont* method call is forwarded up to the document node, which asks the *LatexMathEditor.confg.rootFontManager*. The forwarding mechanism could skipped by accessing the configuration directly. The *rootFontManager* object has a storage of previously requested fonts, if the currently requested font is not in the storage, the font is loaded from a file determined by the configuration and stored in the storage. The reference to the font in the storage is returned. The *specifyFont* method is used to determine the local typeface typesetting letter symbols, the local typeface typesetting digit symbols and what is the local typesetting style (display/line/script/scriptscript). The local typefaces and style depend on the expression's ancestors. The style is further complicated by style changing commands (see section 2.3 on page 14).

The style commands change the style of following blocks in the sequence. A sequence of building blocks is always represented by the an object of the *GeneralContent* class. This class's *specifyFont* implementation allows style commands to change the style of subsequent blocks. This is done by identifying the blocks that affect the style of subsequent blocks, by calling their *doesSpecifyFontForFollowing*. *GeneralContent* then iterates over the identified blocks backwards, beginning on the first identified block preceding the calling block. The *GeneralContent* class remembers identified blocks as long as size buffering is enabled, see the *IRenderer* subinterface for details about size buffering.

IEditationSupport describes services associated with math cursor, mouse cursor and parent containers. In order to display a context menu or move cursor by clicking, the program has to determine which expression lies under the cursor. Virtual methods *getPointedExpression*, *getPointedCursorLocation* and *getPointedCursorLocationInside* help with that. To implement them, the expression needs to know where it was rendered. Whenever the expression is drawn it, should update its internally stored window coordinates. When the math cursor's former location is somehow invalidated, *getDefaultCursorLocation* can be used to find a new one. The routine that caused invalidation of the previous cursor location is responsible to determine a new one. The *navigateCursor* virtual method is used for directional math cursor movement. When *LatexMathEditor* wants to render the math cursor, it asks *getCursorCoords* and *getUnderlineCoords* where and how

to do that. To insert a new building block at the position of cursor, the parent container needs to be determined. The *getHorizontalContentInterface* method does that. To display the context pop-up menu, the menu has to be assembled first by the *getContextMenu* virtual method. The created entries are accompanied by a class derived from *OnContextMenuData* to identify the entry later. Once the user selects an option in context menu, it has to be processed by the *onContextMenu* virtual method. The *onContextMenu* method receives the *OnContextMenu* derivative that accompanied the entry. The *swapExpression* method is used to swap one expression for another.

ICoder is used to generate the expression's source code. The return value is of type *CompoundCode* and includes all variations such as user code or structured code. More details about *CompoundCode* are in section 3.3 on page 28. Additionally, the *hasAtomicCode* method should tell whether it is safe to use returned code as a command argument or index argument without enclosing it by curly braces.

IExpression itself describes the ability to clone (*dynamicCopy*), serialize and deserialize the expression. The virtual method *getPolymorphicSerializationTag* is used to tag serialized data of *IExpression* to determine later to which derived class the data belong to. Serialization/deserialization is required for copy/paste between two program instances. A special feature is the *DeletionWatcher* and *CopyWatcher* class. The first class allows to determine if a certain instance has been deleted, it adds pointer to the list that is checked every time an instance is deleted. This feature is required to determine whether the cursor's location has been invalidated. The copy watcher determines if a certain instance has been copied and what is the address of the copy. This is required for math editor history steps to be able to save cursor's location. The copy watcher is implemented by the same technique as the deletion watcher.

3.3 Other interfaces/classes

IHorizontalContent is an interface of the content of a container, apart from virtual methods similar to standard template library's containers, it supports operations with selection, like querying selection's begin and end, selecting

nothing or something and moving selection's bound.

IExpressionWithContent is an interface of a container. Its *getContentInterface* method is used to retrieve reference to the primary content of the container.

TopLevelEnvironment is an interface of a math block. Since the program allows customization of the edited math block, which often results in a need to change math block's type, virtual methods for copying contents of a math block into another one are required. A method for running common appearance dialog is available.

Scripter is a class implementing the ability of having attached a superscript and/or subscript. The class has methods to help with implementation of *IExpression* interface parts influenced by the presence of scripts. These methods usually require some sort of implementation of the core of the building block. The core is what remains from the expression once indices are removed.

ICode and code generation

The code of the math block is generated in two steps. The first step generates a sequence of elementary code pieces. The generated sequence contains all code types joined together. The second step applies the filter of the requested code variation. The elementary code pieces are then joined together to create the string representation of the generated code variation.

The first step is implemented by the *ICoder*'s *getCode* virtual method. The method returns a *CompoundCode*, which contains a sequence of elementary code pieces in the form of instances of the *ICode* interface. The second step is implemented by the *ICode*'s virtual method *getCode*. This method has a filter argument and returns a string. Depending on the filter, *getCode* either includes or omits *ICode*'s internally stored code string. However, the method also has to handle two complicated situations. The first such situation occurs when an *AutoIndentCode* virtual class is encountered. This virtual class is used to generate indentation appropriate to the nesting level. The nesting level is tracked by the *getCode* method. When *getCode* encounters an *AutoIndentCode*, the virtual method *AutoIndentCode::autoIndentAction* is called and reference to the nesting level is passed. The *autoIndentAction* method of the *BlockBegin* and *BlockEnd* classes modifies the reference. The *autoIndentAction* method of the *Indent* class returns an appropriate amount of whitespace depending on the nesting level. The

second situation is related to a *SignificantCode* representing a control word. When the non-letter character terminating the control word is somehow removed, the following character can be a letter and thus join the control word and change its meaning. Whenever *CompoundCode*'s *getCode* encounters *SignificantCode* marked as a control word code, the first character of following code is checked and if it is a letter, extra space is added.

Notable expressions

List of all classes derived from IExpression can be found in appendix D. In the following subchapters, we present some of the most relevant classes derived from IExpression.

GeneralContent

The *GeneralContent* expression exclusively represents all building block sequences of a container building block or of a math block. It is the only class implementing the *IHorizontalContent* interface. It has the important ability of parsing a building block sequence until the supplied function, a terminator check, returns true. This is done by a loop that first skips space tokens, because they are ignored in math mode, then uses terminator check and conditionally breaks the loop, otherwise it calls *TokenParser* to parse next building block. Unless the block has associated a flag of being an index, it is added to the content. If the block is an index, the routine attempts to give it to the previous building block. If the previous block's appropriate index slot is already occupied or if the previous block can't have an index, an exception is thrown. If there is no previous block, a curly braces container block is constructed to provide one. Once is the parsed building block processed, the loop continues with next iteration.

Symbol

The *Symbol* class represents a symbol building block. The *SymbolSpecs* class property of *Symbol* stores all data required to correctly render the symbol: the glyph alias, the type of index positions (right side of symbol or above and below symbol) and corrections. Corrections allow to modify bounding box metrics supplied by FreeType. Because some symbols such as big operators use different glyph for display style, the SymbolSpecs contains one set of data for display style and one set for non-display style.

LookChanger

The *LookChanger* class is the base of the *StyleChanger* and *FontChanger* classes. It contains a set of properties to adjust behavior of its *specifyFont* method variations.

Overdrawn

The *Overdrawn* class is the base of all decoration and accent commands. It owns an *IFigure* instance supplied to the constructor. The figure is the decoration or accent rendered above or below the content. A boolean property controls whether the figure is above or below the content. Some figures are implemented using *ConstructedGlyph* as base class.

Showcase

The *Showcase* expression has a building block property, this property is rendered and is used as the core of the expression for *Scripter*. *Showcase* blocks the user's access to the property. The *Showcase* class is used to implement commands looking like a simple glyph but typeset using several glyphs, such as `\hookleftarrow` $\hookleftarrow$ (typeset by a left arrow and a right hook), or shortcut commands like `\iint` $\iint$. These command can be found in the inserters toolbar under "Built-in" $\rightarrow$ "Built-in symbols".

3.4 Configuration and configuration file operations

The program uses these configuration files:

The main configuration file assigns path to the other configuration files.

The font map configuration file assigns font filename or path to a font id.

The character map configuration file maps glyph ids to physical glyphs.

The symbol table configuration file defines symbol command groups in the inserter tool bar of the math editor. The symbol group definition includes definition of the symbols in the group.

Table 3.1: syntax of the configuration files

configuration file	record syntax
main	<configuration file id><space><file path>
font map	<font id><size><space><file path>
character map	<glyph id><size><space><font id or generic font id>
symbol table	<symbol command name><size><glyph id><optional parameters>
function list	<function name>

The function list configuration file defines the list of known function name commands.

All configuration files are in plain text and use the layout of one record per line, empty lines are ignored. The symbol table configuration file is an exception, its records are separated into groups by group name lines beginning with "*".

Since they have similar layout, they are similarly processed, modified and saved. The classes representing the configuration files always contain three key structures, an empty line list, a record storage and a line counter. When an empty line is read, it is added to the empty line list and line counter is incremented. When a line with a record is read, the record is parsed and stored in the record storage and the line counter is incremented. The source line code and line index is stored inside record too. When the need to write the structure to a file arises, a loop is executed. The loop has a line counter and an empty line iterator. Each iteration searches the record structure for a record with line index equal to loop's line counter. If one is found, it is written, otherwise an empty line iterator's value is written and the iterator is incremented. At the end of each iteration the loop's line counter is incremented. The loop is terminated once the loop's line counter reaches the value of the structure's line counter.

The symbol table deviates a from the standard, since it has only a list of empty lines in the beginning and list of groups. When a line with a group name is read, an object representing that group is inserted into the group list. Although parse loop is controlled by the symbol table itself, all actions are carried over the last inserted group. A group has internally the standard solution of an empty line list, line counter and a record storage. Writing

of a symbol table is a combination of writing lead-in empty lines and then groups.

Dialogs allow to modify the configuration via GUI. With the exception of main configuration file, which is too simple, the dialogs share the same layout. The dialog has a copy of the programs configuration file representation, cfg class for short. One part of the dialog allows selecting of a record. When a record is selected, its value is displayed in the other part of the dialog. When another record is selected, the values in GUI elements are written to the original record. When a record is inserted or removed, the line indices of all the records, as well as the line counter have to be updated to keep consistency. If/when the user finishes the modifications by choosing "accept", the dialog's cfg class is compared with *LatexMathEditor*'s cfg class and if the meaning is different, *LatexMathEditor*'s cfg class is overwritten by the dialog's cfg class, and then written to the file.

3.5 Parsing

Parsing is the conversion from imported source code to the math block's internal representation. The process is divided into three levels. The first level replaces double superscript escape sequences using the *DoubleSuperscriptReplacer* class. The second level translates characters into tokens using the *CharacterParser* class. The third level translates tokens into expressions using the token *TokenParser*. All parser classes have an interface similar to forward iterator over sequence of their output objects. Parsed expression can create own *TokenParser* instance to parse it's content or can create own *CharacterParser* to look on next tokens and be able to return.

DoubleSuperscriptReplacer

DoubleSuperscriptReplacer is constructed from a *boundWatchingIterator* over a string. The *boundWatchingIterator*'s sole purpose is to throw the *IteratorOutOfBounds* exception when it is dereferenced while pointing beyond the last element of the sequence. This exception is caught by the *CharacterParser* and *Token::EoFile* is returned in response. The original source code of the output current character can be queried by the *getCode* method.

CharacterParser

A *CharacterParser* translates characters into tokens represented by the *Token* class. It uses the same decision logic that was described in section 2.5 on page 18. Above that it has two special behavior flags. The *seekEnvironment* flag ignores all tokens that cannot begin a math block. This is done to accelerate the jump to the next math block. The *isInsideIgnored* flag is an internal flag indicating that the parser is inside of `\lmeignb` and `\lmeigne` pair. When one of the flags is on, the parser uses standard decision logic, only the output is suppressed and the loop continues until a non-ignored token is found. The output token has the form of the *Token* class or one of its derivations. *Token* contains a *TokenId*, a *CompoundCode* and a flag indicating if token's code contained the cursor of the code editor. (Index of character containing the cursor of the code editor can be supplied to *CharacterParser*.)

TokenParser

TokenParser expects to be at a beginning of a math block or of a building block. If a container wants to parse its content, it's up to the container to check for content's terminator. It's a rather simple task, since only space token has to be skipped, any other token is either part of next building block or a part of terminating sequence or a token causing an error. The output expression is in the form of a *IExpression* instance. The *getFlag* method returns circumstances of the current output expression. The circumstances are described by an enumerator. The possible values are:

EF_superscript **or** *EF_subscript* indicating that the output expression is the respective class resulting from superscript or subscript token.

EF_minisuperscript indicating the output expression represents `\prime` created in response to an apostrophe mark. LaTeX codes `a^\prime` and `a'` give the same LaTeX output. Apostrophe behaves as shorthand superscript, for example code `a'_2^2` generates double superscript error. To avoid complex code dealing with this mini-superscript, the apostrophe is converted into "`^\prime`".

EF_normal indicating standard building block.

All expression-representing classes whose code consist of more than one token have a parse constructor, although this requirement can't be stated in the *IExpression* interface. The parse constructor parses all tokens forming

the expression's code and, if successful, moves *CharacterParser* on the following token, otherwise throws a parse exception. The parse constructor of a container block can use the *TokenParser* internally to parse the sequence of building blocks that forms the content of the container.

The *TokenParser* also has the *isTopLevel* behavior flag. If the flag is false and a math block is encountered, a parse exception is thrown. The token parsing table summarizes how *TokenParser* reacts to various tokens.

When *TokenParser* encounters a control word token, it tries to match the token to a *CommandParserOnMatch* instance. The *CommandParserOnMatch* virtual class either knows the control word token and proceeds with parsing or returns *NULL*. A list of usable *CommandParserOnMatch*s is passed to the *TokenParser* during construction in a *ParseContext* object. *TokenParser* simply tries each *CommandParserOnMatch* until one recognizes the control word. If no *CommandParserOnMatch* recognizes the control word, a parse exception is thrown. For details of parsing a building block sequence see section 3.3 on page 29 (GeneralContent).

3.6 Rendering

The section 3.2 on page 25 (IExpression,IRenderer) described the general process of rendering of the math block. This subchapter will deal with the rendering tools available to expressions.

Font

The program uses FreeType library to render symbols. The *Font* class wraps the fonts opened by FreeType. Querying a glyph size is trivial; Font's method *getIndexSize* simply calls FreeType and converts the returned values to a suitable representation. To render a glyph, the Font class's *renderIndex* method internally calls FreeType to paint the glyph on a bitmap compatible with FreeType, the bitmap's content is then copied on a GDK compatible bitmap, which is in turn rendered on a GDK surface. The *renderIndex* method allows to set the rendering color to both black and white, which is used to render the selection in the math editor. Both methods have to account for failure and underscore. When FreeType cannot paint or query glyph it is usually caused by a wrong index, perhaps the glyph is not available in the font. The *Font* class then responds by rendering or returning the size of the error glyph. The underscore doesn't actually have a glyph

Table 3.2: Token parsing table

Token type	Action	Flag
Error	Throw a parse exception about unexpected end of file (the only possible cause of the error token)	N/A
Control word	Call the command's parser, return the result. If no parser matches command, throw parse exception about unknown command. See below for details.	Normal
Letter	Construct a symbol expression directly and return the result.	Normal
Other	If the character is apostrophe construct a prime symbol, otherwise construct a Symbol expression directly. Return the result.	Minisuperscript for apostrophe, else Normal
Begin of group	Call curly braces expression parse constructor, return the result.	Normal
Superscript	Call superscript expression parse constructor, return the result	Superscript
Subscript	Call subscript expression parse constructor, return the result.	Subscript
Math shift	If *sTopLevel* is false, throw a parse exception, otherwise check next token, Call dollar or double dollar math block constructor appropriately and return the result.	Normal
None of above	Throw a parse exception about unexpected token type.	N/A

representation in fonts, so the *Font* class has to render it by itself and return the appropriate size. "Index" in the name of both methods means, that the index argument is the font's internal index. FreeType has a translation mechanism. There are "Char" variations of the "Index" methods, which use that mechanism to obtain the internal index and call the "Index" variation. The *Font* class also offers "String" and "IndexString" versions of the previous methods for convenient rendering of strings.

An expression has to use its parent's *IFontManager* interface to obtain a reference to a *Font* class. If the expression knows exactly which font is required, it calls *getFont* method. If an expression wants to determine what typeface should be used to typeset letters or digits, or what is the local typesetting style, it has to use it has to use the *specifyFont* method of its parent.

FlowLayoutManager

This class is used by *GeneralContent* to typeset its content. The algorithm first has to find B, the maximum $b(x)$ where x is any block in the content and $b(x)$ is distance between the top edge and the baseline of x. For digital typography details see Turner, The FreeType Development Team (2000). The *FlowLayoutManager* then typesets blocks vertically at height$(x) - B$. The height of content is $B + B'$ where B' is maximum height$(x) - b(x)$, where x is any block in the content.

TableLayoutManager

This class is used by expressions to typeset a tabular content. The algorithm has to first determine maximum width of each column and maximum height and maximum $b(x)$ of each row. The situation is complicated by horizontal and vertical outlines, which must be added to heights and widths respectively. The *multicolumn* command complicates the task even further. LaTeX typesets multicolumn so that if it needs more space, it enlarges the rightmost column of its span. To mimic that, the algorithm iterates over columns and a nested loop iterates over lines. Multicolumn can only influence the width of the rightmost column of its span so the other columns of its span process it as a zero width field. The width required from the rightmost column is computed as the width of the multicolumn minus widths of previous columns of the multicolumn's span. Once those are known, the rest of the

algorithm is quite trivial.

The class also contains methods to create and execute context menu entries modifying the tabular content.

ConstructedGlyph

In the analysis chapter, we mentioned the `\left` command, which typesets a delimiter matching the size of the content. This delimiter is often a single glyph but once the content grows sufficiently tall, the telescopic version of the delimiter is used, if one is available. The telescopic version is a combination of several glyphs and a repeating glyph. The repeating glyph always forms a line or double line, so the program renders a line directly. The *ConstructedGlyph* virtual class was developed to ease the implementation of (telescopic) constructions from glyphs. The *ConstructedGlyph* class has the *render* method, the *getPaintQueue* virtual method and the *getSize* virtual method. The *render* method renders the construction by obtaining and executing generalized instructions from the *getPaintQueue*. The *getPaintQueue* method has to return a list of *LineOrders* and a list of *GlyphOrders*. A *LineOrder* tells how to render a line and a *GlyphOrder* tells how to render a glyph. The virtual method *getSize* has to return the size of the represented construction. The constructors of telescopic constructions are often parameterized with the requested size.

3.7 Implementing a new building block

All classes representing a building block must implement the *IExpression* interface. Containers have to also implement the *IExpressionWithContent* interface. Blocks capable of having a superscript and subscript have to be derived from the *Scripter* class. Suppose a class named "TestExp" is to be implemented. Let *TestExp* represent a scriptable container building block, which draws a full or an empty circle on the left side of the content. Let the syntax be `\testexp<star><content>` , where `<star>` is an optional argument "*". Let the circle be filled if and only if the optional star is included. The resulting code from implementing this *TestExp* is included in the source codes of the program for further reference. All the code is skipped unless "_DEBUG" macro is defined. The steps are labeled in the code, each step fragment is tagged by `x` and `/x` , where x is the number of the step.

The implementation should be written in a new .cpp and .h file with the

same name, so create "TestExp.cpp" and "TestExp.h". "TestExp.cpp" has to be added to the source file list in the makefile to allow use of the make tool. The source file list begins with "LMESRCS = ".

1. The source code file should contain all the method definitions and the header file should contain the class declaration, thus the source file has to first include the header file.

2. The header should be protected from double inclusion.

3. Include "Expression.h" to the header to gain access to both *IExpression* interface and its partial default implementation of *Expression*. For container building blocks include "GeneralContent.h", for scriptable building blocks include "Scripter.h".

4. Declare the *TestExp* class, derive it from *Expression* to take advantage of its default implementations. Derive it also from *IExpressionWithContent* if the building block is a container and from *Scripter*, if the building block can have indices.

5. Declare data properties of *TestExp*. It is recommended to declare a *CompoundCode* for each loose source code part. Building block classes capable of having indices usually declare *RendererMapper* property, to implement *IRenderer* of the core, and *ScriptedRenderer* property, to join *Scripter* and *RendererMapper*, resulting in the rendering of the whole building block.

6. Declare basic class capabilities - direct constructor, copy constructor, copy operator, deserialize constructor and *dynamicCopy* method. If the source code is more than one token long, declare parse constructor and *parse* method. All these methods are public except the *parse* method. Declare public methods *serializeTo* and *getSerializedSize*, and protected method *deserialize*. Include "CharacterParser.h" for parse constructor and method arguments declaration.

7. If building block can have indices, declare the core interface methods. Use protected accessibility.

8. Define everything declared in step 6.

(a) The *parse* method should be called by parse constructor which should be called by *ParserOnMatch*, so the CharacterParser should be pointing on the next token after command's control word. In this case, this is either the building block or the optional star. Conditionally parse the star and then parse content using *GeneralContent::parseGroupOrSingle*. Don't ignore *CursorLocation* returned by parsing functions and methods, use | = operator to conditionally assign non-null *CursorLocation*. Caller should detect and solve the case of the cursor belonging to the parsed building block itself.

(b) The *operator=*, *deserialize*, *serialize* and *getSerializedSize* methods should traverse all value defining properties in the same order and carry out respective operations. In the case of scriptable building block, skip *ScriptedRenderer* and *RendererMapper*, but don't forget *Scripter*, since it holds indices.

(c) Non-direct constructors should call respective methods, for example, deserialize constructor calls *deserialize*. If implementing scriptable building block, all constructors should link *RendererMapper* with *this* pointer and the core interface methods, and they also should link *ScriptedRenderer* with the *RendererMapper* and *this* pointer (the base *Scripter*). Don't forget to set parent of properties derived from *IExpression*, like *GeneralContent*.

(d) The *getPolymorphicSerializationTag* method should return an integer unique to the class. To do that, one must be added to the *ExpressionTag* enumerator in "Expression.h" file. This tag is used when deserializing a building block representing an expression, so the tag must be added to the switch in the deserialization routine *deserializePolymorphic* in "Expression.cpp" file. The routine has to call deserialization constructor, so include "TestExp.h".

(e) The *dynamicCopy* method should report the copy by calling *reportDynamicCopy* on the returned pointer.

9. (a) In case of a container, implement *IExpressionWithContent* by using the *IMPLEMENT_IExpressionWithContent* macro. The argument has to evaluate to the primary content of the building block.

(b) Consider overriding default implementation of the method *hasAtomicContent* for blocks made up from multiple tokens.

(c) In case of a command, declare *ParserOnMatch* nested class. Derive *ParserOnMatch* from *CommandParserOnMatch*. Include "Command.h" to gain access to *CommandParserOnMatch*. Implement *ParserOnMatch*. Add the implemented parser-on-match to building block commands parsers in the "TopLevelEnvironment.cpp", include "TestExp.h" in that file.

(d) Consider implementing a context menu. Do it by declaring *getContextMenuEntry* and *onContextMenu* overrides. Implement these overrides. A class derived from *OnContextMenuData* will be probably needed to identify the selected option. Use *ContextMenuEntry* and *ContextMenuOption* classes to create the context menu. It is safe to expect that argument of *onContextMenuOption* originates from the *getContextMenuEntry*, so it can be cast appropriately. Use the *lme* property to query current math cursor location, use *IExpression::DeletionWatcher* to detect invalidated math cursor location. Use the *lme* property to redraw the math block if something visible is changed.

10. Declare the rest of the *IExpression* interface. Define it:

(a) Implement *getCode*. Call *getCodeWithCursorMark* on returned value, to account for cursor on sides. If the block can have indices, use *Scripter::getScriptCode*.

(b) If the block can have indices, define *draw* and *getSize* by calling the respective method of *ScriptedRenderer*, otherwise implement them the same way as *drawScriptless* and *getScriptlessSize* would be implemented. To implement *drawScriptless* you can use especially GDK drawing functions and rendering methods of the *Font* class. Calculate the size of the drawn expression for *getSize* call. Complex blocks might benefit from a shared typesetting method. Remember to *setBufferedOffset* in the *draw* method.

(c) Implement *enableSizeBuffering* and *disableSizeBuffering* in a similar fashion to previous substep. Both methods should reach all contained expressions.

(d) Implement *getPointedExpression* and *getPointedCursorLocation*. The idea is to try all contents and if all return null, return self.

(e) Implement *getCursorCoords*, use *Expression::getCursorCoords* and *Scripter::getCursorCoordsScriptedSpecial*.

(f) Implement *getDefaultCursorLocation*, containers should return *getDefaultCursorLocation* of their primary content.

(g) Implement *navigateCursor*. This method doesn't navigate cursor inside *GeneralContent* properties, only into and out of them.

Chapter 4

Conclusion

4.1 Summary

The program successfully satisfied all the key objectives. Importing and subsequently exporting a source code reveals only minimal changes, mostly when legacy TEX source code is used. The process of supporting a new building block has been documented so the set of supported blocks can be extended and eventually become broad enough to allow wide user base to take advantage of the graphical user interface. However, by not implementing text mode, the program lost the ability to support text mode embedded in math block. Such embedded text mode can be at least skipped.

The implementation of GUI elements was considerably more time-consuming than expected, which can be attributed to the lack of experience with GUI programs. The development was also slowed by necessity to precisely implement the LATEXcommands. A program that uses proprietary syntax can modify the syntax to its needs, MaEd can't.

4.2 Future work

The program's value to the user is dependent on the supported commands, the more, the better. The program will never support all commands because of the extendibility of LATEX. But it could support so many commands so 99% users don't encounter an unsupported one. The program could be faster in many areas, there certainly is a space for optimizations. For example the parser copies the source code as it traverses through its layers. The rendering

system could be more aware of not being changed and return buffered values more often instead of expensively computing them again.

Appendix A

User documentation

Welcome to MaEd for LaTeX , a portable GUI program for creating and editing of LaTeX formulae. You can use this program to ease the task of creating formulae whether you are beginner or experienced LaTeX user. Unlike other programs, this program does not unnecessarily alter imported code, yet it allows you to use full features of the graphical user interface. Be aware that you need a working LaTeX installation in order to reasonably use the created source code file. Since this program is only a specialized tool, you'll need to know how to work with LaTeX. A good tutorial is "The Not So Short Introduction To LaTeX 2_ε".

A.1 Installation

Windows

Copy the entire program's windows version folder into the target directory of your choice.
Continue to the shared part of the installation.

Unix

Ensure the following prerequisites or newer versions are installed on your system:

- g++ 4.1.1

- libboost-dev 1.33.1
- pkg-config 0.21
- libgtk2.0-dev 2.8.29
- libfreetype6-dev 2.2.1

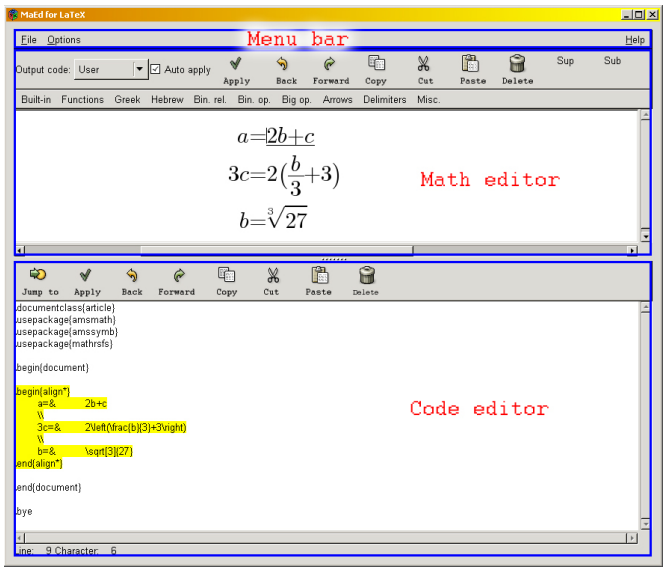
Copy the entire program's unix version folder into the target directory of your choice.
Execute the Makefile in the target directory. If nothing goes wrong the program's executable file "maed" is now created.
Continue to the shared part of installation.

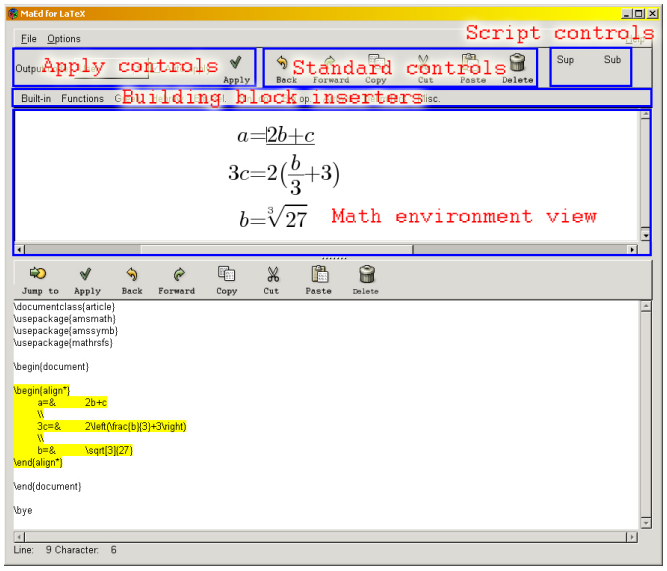
Shared

The program can use the fonts present in your TEX installation. If you don't have one or don't want use its fonts, you can still use the fonts distributed together with the program. To use the LATEX installation fonts, configure the program using the "font options" dialog described in the "preferences" subsection, it might suffice to set the font working directory to type 1 font directory inside font directory of your LATEX installation. To use fonts distributed together with the program, copy the fonts folder into your local MaEd folder; the licenses of the fonts are in their subfolders. The installation is now complete.

A.2 Getting started

User interface overview





The program is divided into 3 parts: the menu bar, the math editor and the source editor. The menu bar contains the file submenu with usual open/save operations, the options submenu and the help submenu. The source editor contains the text that was read from the opened file and that will be saved by "save"/"save as" operations. One math block can be selected by positioning the cursor before its beginning and clicking "jump to". From the source code point of view, a math block is the code causing LaTeX to enter the math mode, combined with the corresponding code causing LaTeX to leave the math mode and all code in between. From the LaTeX output point of view, it is a unit of math. By clicking "apply", the selected math block is parsed and shown in the math editor window.

The math editor has an "apply" button too. By clicking it, the code of the selected math block in the code editor is replaced by the code of the math block shown in the math editor. The math "Apply" button is only one of the apply controls. Checking the "Auto-apply" box causes the code to be updated after each change in the math editor.

The code generated by the button depends on the output code option box. The default value is the user code. This type of output doesn't do any formatting changes to the originally imported code. The second option is the structured code. This type of output formats the code to make the nesting maximally visible. The third option is the compressed code. This type formats the code to save as a much space as possible.

The math editor view shows the math block and the math editor cursor. The cursor can be navigated by keyboard or mouse. To get the cursor into an empty superscript or subscript, navigate the cursor to the right side of the building block in question, then press control-up or control-down respectively or click "sup" or "sub" button respectively. The expression in the math editor window is made of basic elements, which we call "building blocks". A building block is for example a symbol, a matrix, a number, or a fraction. Now that you know how to navigate the cursor, you can use it to insert a building block. To do that, either click on some entry in the building block inserters menu or press any number, letter or punctuation key on your keyboard. Shift key + navigation keys allow to select sequence of building blocks, for copying/deleting and moving into the content of inserted container.

Some building blocks and all math blocks have a context menu. To access it, right-click on the block. Context menu allows you to alter the block.

How to create a not so simple equation system.

$$5x = \left\lfloor \lim_{\substack{n\to\infty \\ k=n^2}} \overbrace{\sum_{i=1}^{\sqrt{25k}} \frac{n \cdot n}{k}}^{5} \right\rfloor^2 y \tag{1.}$$

$$x = \frac{1}{5} \tag{2.}$$

1. Launch the program.
2. Right-click on the math block in the math editor, select "Math block appearance...".
3. Select display, multiple and numbered, confirm.
4. Scroll the window to see the modified block.
5. Right-click again on the math block and select "Add line".
6. Move cursor to the top-left content, type "5x=".
7. Move cursor to the bottom-left content, type "x=".
8. Move cursor to the bottom-right content, click on building block inserters bar: "Built-in" → "Vertical content" → fraction image.
9. Use skills from previous steps to fill the fraction.
10. Move cursor to the top-right content, type "y".
11. Move the cursor back to the left side of "y", click on building block inserters bar: "Built-in" → "Other" → parentheses image.
12. Right-click on the parentheses and select appearance. Select a floor on both sides.
13. Move the cursor right of the parentheses and click "sup" tool button or press control-up arrow, then type "2".

14. Move the cursor into the parentheses and click on building block inserters bar: "Functions" → "lim".

15. Click "sub" tool button or press control-down and click on building block inserters: "Built-in" → "Vertical content" → "substack".

16. Use the skills from previous steps to fill the substack. The infinity symbol is in the "misc." group and right arrow in the "arrows" group of building block inserters bar.

17. Insert the sum from the "big op." group.

18. Insert the fraction including the numerator and denominator, you'll find the dot operator in the "bin op." group.

19. Insert sum's subscript.

20. Move into the sum's superscript, click on building block inserters bar: "Built-in" → "Other" → "Root". Fill the root. Note that content of the index will remain empty and its bounding box drawn, but the LaTeX installation doesn't draw these bounding boxes.

21. Select the sum and the fraction using the shift key, then click on building block inserters bar: "Built-in" → "Decorations" → "Brace over".

22. Fill superscript of the last inserted container.

23. Click apply in the math editor to retrieve LaTeX source code of the math block.

Preferences

The programs preferences, are used to teach the program about LaTeX commands the program doesn't know. You can define a new symbol or a new function name. If you want to do that, you are probably at least an intermediate LaTeX user, so you shouldn't be surprised by LaTeX terminology in the following text.

function names

Function names are the commands typesetting a function name such as `\lim` (lim), `\sup` (sup) or `\max` (max). Open the "options" on the main

menu bar and select "function options". Now you can click add to create a new function name, the entered name's source code is automatically `\name` . The only available option is whether indices are typeset on side or above and below when the display style is active. When you confirm your changes, the functions group on the building block inserter bar is automatically updated.

symbols

Defining a new symbol has one caveat, you have to know which glyph index in which font the symbol uses, unless you are incredibly lucky to have the glyph already defined as one of existing glyph aliases. Some symbols have separate glyph for display and non-display style, so you need to find two glyphs.

Open "options" on the main menu bar and click on the "symbol options". This launches a dialog. On the left side you can modify page list and select a page. Pages are the options groups to the right from "functions" on the building block inserter bar. Once you select a page, you can modify the list of symbols in the page and select a symbol. On the right side of the dialog, you see the settings of the selected symbol. These setting are divided between display style and non-display style. You have various tools to tweak the display of the symbol but the most important is the selection of the glyph. Select a glyph alias. If none of the existing glyph aliases satisfies you (probable), select an unused custom glyph alias and click the edit glyph button and setup the glyph alias using the glyph alias dialog. The glyph alias dialog is explained in the next section. Now that you have selected a valid glyph alias, you should see the preview. Use the available controls to tweak the symbol's appearance. Don't forget to configure both styles. Once you confirm your changes, the building block inserter will be automatically updated and the symbol's command will be recognized. Note that math blocks in the math editor and its history will not get updated, but newly parsed math blocks will be.

glyph aliases

To edit glyph aliases, open the "options" on the main menu bar and select "glyph options". Select the glyph alias you want to edit. Click "use" check box to enable this particular glyph alias. Now you can select the font and the glyph index. You can select a custom font. If you haven't configured the selected font yet, use the "edit font" button and configure it, the font dialog

is explained in the next section. Now you can select the glyph index, note that you should see a preview of the selected index by now. Complete the configuration by selecting an appropriate font size; context is usually the right option for glyphs used in symbols. Note that math blocks in the math editor and its history will not get updated, but newly parsed math blocks will be.

fonts

To edit fonts, open the "options" on the main menu bar and select "font options". This dialog allows assigning a font file to a font alias. The assigned file needs to be a type 1 font ("pfb" extension). The entered paths are relative to the font working directory shown in the top part of the dialog. If you enter just a file name, the entire font working directory including subdirectories will be searched for the name when the font is needed. The middle part of the dialog allows the selection of the font alias you want to edit. Select one. Click "use" check box to enable the selected font alias definition. Note that built-in font aliases use a default file name if the check box is unchecked. Use the entry and/or the browse button to specify the font file. Once you confirm your changes, the program will update used fonts using the specified font files.

The font file assigned to the font can have a different size. In case that you don't have some required font in the right size (probable), use the one closest to the right size. The size of its glyphs will be scaled and the result will be fine.

How to find out which glyph a symbol uses

There is a dirty trick to find the font and glyph index of a symbol using your LaTeX installation. Create following source file:

```
\documentclass{minimal}
\pagestyle{empty}
X
\begin{document}
\[
Y
\]
\end{document}
\bye
```

Replace the "X" with list of `\usepackage{...}` required for the symbol to be typeset. Replace the "Y" with the command of the symbol. Process the file with the pdflatex program. Open the output in text editor/viewer. Find consecutive lines beginning with

```
/BaseFont
/FontDescriptor
/FirstChar
/LastChar
```

The base font line should end by the short name of the font. The first char and last char should contain the same number - the index of the glyph. "Should" means that even if you typed one command, more than one glyph can be used to typeset it. Two glyphs could come from one font and thus the first char isn't equal to the last char and/or the file contains more than one occurrence of the lines, each for one font.

A.3 Troubleshooting

Problem: My source code contains unsupported code, which I want to keep. Solution: If the code in question is a parameter-less command, define this command as a function name. You'll have to remember, what the name means. If the code is a command with parameter, enclose the code by {`\lmeignb` and `\lmeigne`} . You'll have to remember, that it isn't an ordinary group. If it isn't a building block, enclose it by `\lmeignb` and `\lmeigne` . The code will be treated as comment, so be careful with changes near it. To make your LaTeX ignore `\lmeignb` and `\lmeigne` , use this definition:
`\newcommand{\lmeignb}{}\newcommand{\lmeigne}{}`

A.4 Legal

Appendix B

Implementation Objects hierarchy overview

LatexMathEditor

- version
- file
- config
  - symbolTable
  - functionNameList
  - rootFontManager
  - characterMap
  - data describing structure of main configuration file
- callData
- temporaryPopUpData
- editors
  - math
  - code
  - pointers to relevant program's GUI elements
- pointers to relevant program's GUI elements

LatexMathEditor.config.symbolTable

- pageVector
  - name
  - nameCode
  - lineCount
  - mapping
    - lineIndex
    - source
    - visual
  - names
  - emptyLines
- emptyLines

LatexMathEditor.config.functionNameList

- records
  - source
  - lineIndex
  - info
- mapping
- emptyLines
- lineCount

LatexMathEditor.config.rootFontManager

- fonts
- emptyLines
- service

- workingDir
- lineCount
- mapping
  - sourceLine
  - file
  - lineIndex

LatexMathEditor.config.characterMap

- mapping
  - lineIndex
  - source
  - loc
    - fontId
    - index
  - emptyLines
  - lineCount

LatexMathEditor.editors.math

- shallRedrawCanvas
- isActive
- document
- history
  - steps a list of Step
    - document
    - cursorLocation
  - backstepCount

- cursor
  - location
  - color
  - currentId
- tools
  - pointers to relevant program‘s GUI elements
- inserters
  - symbolButtons a list of GtkMenuItem pointer
  - symbolCallData a list of LMECallData
  - functionNameButton
  - functionNameCallData a list of LMECallData

LatexMathEditor.editors.code

- history
  - step a list of Steps
    - code
    - cursor
    - editedEnvironment
  - backstepCount
  - lock
- tools
  - pointers to relevant program‘s GUI elements
- pointers to relevant program‘s GUI elements

Appendix C

Source code file list

Source code files (".cpp" extension) define what is declared in header files ("h" extension). Therefore the description of a source code file is equal to description of the header file with the same name.

Header file	**Content description**
Align.h	The implementation of the "align" math block environment.
Array.h	The implementation of the "array" container building block environment.
BadInputFile.h	The exception to throw when input file has a bad format.
Begin.h	The implementation of the `\begin` , a command parser, the parser of all environments.
Big.h	The implementation of the `\big` , a building block command and its variations.
Binom.h	The implementation of the `\binom` , a container building block command.
BoldSymbol.h	The implementation of the `\boldsymbol` , a container building block command.
CallBack.h	*CallBack*, a wrapper of *gtk_signal_connect*.
CharacterId.h	*CharacterId*, the glyph alias enumerator and utility functions.
CharacterMap.h	*CharacterMap*, a map from a glyph alias to a pair of font and glyph index.
CharacterParser.h	*CharacterParser*, the class translating ASCII characters into tokens.
CodeAssist.h	Implementation of various common code constructs, such as tabular code.

Header file	**Content description**
CodeBlocks.h	Various *ICode* derivatives.
Coder.h	The *ICode* interface and CompoundCode class.
Command.h	The *CommandParserOnMatch* interface.
ConstructedGlyph.h	*ConstructedGlyph*, a glyph-like object made up from several glyphs.
CurlyBraces.h	Curly braces, a container building block for grouping a building block sequence into a single building block.
CursorLocation.h	The *CursorLocation* class, that describes location of math cursor in the math editor.
DebugExp.h	*DebugExp*, a class usable for debugging building blocks.
DebugImage.h	*DebugImage*, a class usable for comparison of MaEd output to LaTeX output.
Document.h	*Document*, the link between the math editor's math block and *LatexMathEditor*.
EditationSupport.h	*IEditationSupport*, the interface used for the implementation of math editor cursor, context menu etc.
EParse.h	The exception to throw when LaTeX source code can't be parsed.
Expression_FwdDecl.h	Forward declaration of IExpression interface.
Expression.h	IExpression, the interface of a building block.
FlowLayoutManager.h	*FlowLayoutManager*, the class that typesets a sequence of building blocks.
Font.h	*Font*, the wrapper of a FreeType font.
FontId.h	*FontId*, the enumerator of fonts and utility functions.
FontManager.h	*IFontManager*, the interface used to allow a building block to look up a font.
FontService.h	*FontService*, the wrapper of FreeType main handle.
Frac.h	\frac , a container building block command.
FunctionName.h	*FunctionName*, a building block command.
FunctionNameList.h	*FunctionNameList*, the class managing function names configuration.
GeneralContent.h	GeneralContent, the class implementing the contents of all container building blocks.
GtkIncluder.h	Includes gtk header.
LatexMathEditor_Deputy.h	a handle to a several properties of *LatexMathEditor*, used by math editor context menu handlers.
LatexMathEditor_FwdDecl.h	a forward declaration of *LatexMathEditor*.

Header file	**Content description**
LatexMathEditor.h	*LatexMathEditor*, the class representing entire program.
Left.h	`\left` , a container building block command, also contains a database of delimiters.
LookChanger.h	*LookChanger* the common base of the *FontChanger* and *StyleChanger* classes, *FontChanger* implements typeface commands and *StyleChanger* implements style commands.
Matrix.h	*Matrix*, a class implementing matrix container building block environments.
Not.h	The implementation of `\not` , the container building block command for negating an operator.
Overdrawn.h	*Overdrawn*, the common base of classes implementing decorations and accents.
Phantom.h	*Phatom*, the class implementing `\phantom` , `\vphatnom` and fixed size space building blocs.
Pmod.h	The implementation of `\pmod` , a container building block.
Renderer.h	*IRenderer*, the interface used to display building blocks.
RendererUtils.h	Utility classes useful for rendering.
RootFontManager.h	*RootFontManager*, the class buffering fonts and managing font configuration.
Root.h	*Root*, the class implementing `\root` and `\sqrt` commands.
Scripter_FwdDecl.h	Forward declaration of *Scripter*.
Scripter.h	*Scripter*, the base class of all building blocks capable of having indices.
Script.h	*Superscript* and *Subscript*, properties of *Scripter* containing the respective index.
Showcase.h	*Showcase*, the class showing its content but disabling access to it.
SingleLineTopLevelEnvironment.h	*SingleLineTopLevelEnvironment*, the common implementation of all math blocks unable of showing multiple equations .
Son.h	The *ISon* interface.
Space.h	*Space*, the common implementation of argument-size spaces.

Header file	**Content description**
Split.h	The "split" environment, supported inside the "equation" environment.
Substack.h	The implementation of the \substack container building block command.
Symbol.h	*Symbol*, a building block.
SymbolSpecs.h	*SymbolSpecs*, the class describing typesetting of a *Symbol* in display and non-display style.
SymbolTable.h	*SymbolTable*, the class managing the symbol configuration.
TableLayoutManager.h	*TableLayoutManager*, the class used to typeset a tabular code.
TestExp.h	*TestExp*, the documented example of implementation of a container building block.
TextExp.h	The implementation of \text , a building block command.
TokenParser.h	*TokenParser*, the class translating tokens into expressions.
TopLevelEnvironment.h	*TopLevelEnvironment*, the base class of a math block.
utils.h	Assorted utility functions and classes.
Zoom.h	*zoom*, the constant modifying the display size of the math block in the math editor.

Appendix D

List of subclasses of IExpression

Derivatives marked by * have further derivatives, which are not listed. Unless specified otherwise, the header file of the class has the same name as the class.

Derivative	**Type/Description**
Split	Implements "split" environment inside "equation" environment
TopLevelEnvironment	Math block interface
Align	Math block
*SingleLineTopLevelEnvironment**	Math block, derivatives represent each non-multiline math block.

Array	Tabular container building block
Big	Building block
Binom	Container building block
BoldSymbol	Container building block
CurlyBraces	Container building block
DebugExp	Debugging block
Document	The link between *LatexMathEditor* and the math editor's math block
Frac	Container building block
FunctionName	Building block
*GeneralContent**	Used to hold containers' content, derivatives represent specialized contents such as field of tabular content.
Left	Container building block
LookChanger	Common base of *StyleChanger* and *FontChanger*
StyleChanger	Building block, style command, declared in "LookChanger.h".
FontChanger	Container building block, typeface command, declared in "LookChanger.h".
Matrix	Tabular container building block
Not	Container building block
*Overdrawn**	Container building block, common base class of all accents and decorations.
Phantom	In LaTeX, a container building block; In the math editor, a building block.
Pmod	Container building block
Root	Container building block
Script	Container, common base of *Superscript* and *Subscript*.
Superscript	*Scripter*'s property holding the superscript, declared in "Script.h".
Subscript	*Scripter*'s property holding the subscript, declared in "Script.h".
Showcase	Building block
Space	Building block
Substack	Tabular container building block
Symbol	Building block
TestExp	Container building block, example of IExpression implementation
TextExp	Container building block

List of Figures

List of Tables

Bibliography

[1] GTK+ team and the contributors (2009): *GTK+ Reference Manual*, available at http://www.gtk.org/documentation.html

[2] Knuth D.E. (1984): *The TeXbook*, Addison-Wesley, Reading, 36-49.

[3] Oetiker T., Partl H., Hyna I., Schlegl E. (2008): *The not so short introduction to LaTeX2e*, available at http://www.latex-project.org/guides/, 49-70.

[4] Turner D., The FreeType Development Team (2000): *FreeType Glyph Conventions - Glyph metrics*, available at http://www.freetype.org/freetype2/docs/glyphs/index.html.