The design of the MathSpad editor

The Design of the Mathʃpad Editor

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Verhoeven, Richard

The Design of the MathSpad Editor / by Richard Verhoeven – Eindhoven :
Eindhoven University of Technology, 2000
Proefontwerp. – ISBN 90–386-0781-4
NUGI 852
Subject headings: text processing ; mathematics /
mathematics ; notation /
software systems
CR Subject Classification (1998): I.7.2, I.7.1, G.4, H.5.2, D.2.11, D.2.13, K.8.1

Print: UniversiteitsDrukkerij, Eindhoven

This research was financially supported by the Eindhoven University of Technology and the Netherlands Organisation for Scientific Research (NWO). The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

The Design of the MathSpad Editor

Proefontwerp

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. M. Rem, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op woensdag 21 juni 2000 om 16.00 uur

door

Richard Verhoeven

geboren te Veghel

Dit proefontwerp is goedgekeurd door de promotoren:

prof.dr. R.C. Backhouse

en

prof.dr. P.M.E. de Bra

Acknowledgements

There are many people who were supportive while the research described in this thesis took place. They all provided useful information on how to improve the system or were helpful in some sense.

- My supervisor Roland Backhouse, for providing me with this challenging masters project, which became the long lasting research project. I want to thank him for his patience, his suggestions and the effort he put in finding research funds. Furthermore, I want to thank his family for testing the system and for the Friday evening meals.
- The former Ph.D. students Henk Doornbos, Paul Hoogendijk, Laurens de Vries and Ed Voermans, for sharing the room and having fruitful discussions about the different research topics. Furthermore, I want to thank them for testing the system and making suggestions.
- My fellow student Olaf Weber, for writing part of the system during our combined masters project.
- The members and former members of the research group in Eindhoven: Eerke Boiten, Lex Bijlsma, Joop van den Eijnde, Wim Feijen, Netty van Gasteren, Rik van Geldrop, Kees Hemerik, Jaap van der Woude en Gerard Zwaan.
- The other members of the Ph.D. committee: Professors Paul de Bra, Andries Brouwer en Lambert Meertens, for reviewing this thesis and giving detailed and useful comments.
- Several members of the Computing Science department in Utrecht, especially Lambert Meertens, for pointing out several interesting features which should be incorporated into the system.
- The School of Computer Science & IT at the University of Nottingham for funding the final $4\frac{1}{2}$ months of this project.
- The members of the Languages and Programming group in Nottingham: Paul Blampied, Graham Hutton, Claus Reinke and Natasha Alechina, for the discussions on several research topics.
- My family, for their support and for the regularly scheduled weekend jobs.

And everyone I forgot to mention.

To my parents.

Contents

List of Figures

Chapter 1

Introduction

Technical documents, and mathematical documents in particular, are difficult to prepare[12]. The need, however, for systems that assist authors in preparing technical documents is very great; such documents are prepared all the time by all sorts of scientists and technicians and the documents often have several revisions before they are finally ready for publication.

In the past, the preparation of a technical document underwent several stages. First, the author prepared a manuscript which was then typed to produce a typescript (sometimes by a professional typist but sometimes by the author). This process was then often repeated many times before a satisfactory final version was ready. At this stage the document was sent to a professional copy-editor who marked it up ready for handing to a printer whose task was to produce the final printed copy. Several rounds were then necessary for the author to check that no errors had been introduced by the printer and that the layout was to the author's satisfaction.

At the present time the majority of technical documents are prepared using a computerised system, often TEX or LATEX. This has the advantage that the process of revising a document is much easier and quicker, and the author has better control over the production process, in particular avoiding the introduction of technical errors. It has the disadvantage that the author is burdened with many details of layout and appearance that previously were the responsibility of a professional copy-editor. The effect is that, even to this day, few authors compose mathematical material at a computer; they still use paper and pencil to develop their ideas and the computer systems are only used when these have been worked out in some depth. Computerised document preparation systems are thus best-suited for producing high-quality finished products and are of limited assistance during the development of the mathematics itself.

The MathSpad system was conceived with the idea of doing away with the use of paper and pencil during the preparation of mathematical documents. The system would be used for composing mathematics at a computer screen. It would thus support the preparation of mathematical documents from beginning to end, focusing on the

creative activity of doing and writing about mathematics but also assisting with the preparation of the final publication.

This explains the name of the system: an integration ($\int$) between doing mathematics ("math") and writing about it ("pad").

1.1 The situation

The original target users for the Math∫pad system were the members of the Mathematics of Programming Construction group at the Eindhoven University of Technology. This group produced a lot of calculations and publications in non-standard mathematics. At the time, symbolic algebra systems used ASCII characters for input and output of mathematical expressions, which wasn't suitable for the group, as their notational conventions used many symbols not available among the ASCII characters and the mathematical theory was not supported by those systems. There were also some WYSIWYG document preparation systems available, but they didn't support the non-standard mathematical content properly and were not suited for doing mathematics. For technical support, a proof editor was used which had been designed by Paul Chisholm[13] to construct proofs and expressions and import them into a LaTeX document. Although that editor worked as expected, there were several problems with the system. It was built using a graphical toolkit which was not portable and not supported anymore, and the interface was based on a desktop with many small notes lying around, which quickly becomes difficult to use.

1.2 An example task

A typical task for the initial group of users is writing a technical report or a paper for a journal. Such a report usually contains some prose to explain which theorem is proved in that report and when the theorem is applicable. After the initial explanation, a technical discussion follows which leads to a proof of the suggested theorem. Before the proof can be constructed, a collection of lemmas, corollaries, definitions and other theorems are needed, all of which contain mathematical expressions and references to earlier parts of the document.

In the beginning, the author writes a report with some mathematical content. Since there is no special application that can check whether the mathematical content is valid, the author is responsible for the correctness. For the author, it is easier to see if something is correct, if it is displayed as well as possible. The layout on screen might not be perfect, but it should be very readable and clear what the author has written. If corrections are needed, the author should be able to apply them instantly, without switching between different applications or edit modes. Since mathematical calculation implies that the same expressions occur very often[1], usually with small

[1] A challenge from K-Talk to promote MathEdit contained a half page formula, which, after a closer look, consisted of 5 small formulæ, repeatedly used with small adjustments.

changes, the author should be able to select, copy and adjust expressions as easily as possible.

Once the report is written, it is distributed among the members of the group for feedback, after which the report will be revised. At this point, it is important that the user can see where he is editing his document. Since the document contains a lot of expressions which often occur at different locations with small modifications, a clear view of the document is necessary. Often the changes are small and local, but sometimes they are large or global, like replacing a proof or changing the notational convention.

The group's work involves experimenting with different, not necessary mature, formalisms. For this reason, notational conventions might change by adding new notations or replacing existing notations and built-in mathematical knowledge is less important than ease of use. Certain expressions might not make sense mathematically, but if it makes editing easier, it should be possible to use such expressions as temporary results.

1.3 The goal

The initial goal was to develop a system for doing mathematical calculations on screen. The following priorities were identified, each demonstrated with a small example.

- **Readability**. The mathematical expressions have to be readable on screen and easy to manipulate.

$$
\begin{array}{ll}
 & \eta.f \ \in \ F.x \leftarrow G.y \\
\Leftarrow & \quad \{ \quad G.f \in G.x \leftarrow G.y \\
 & \qquad \bullet \quad \eta.f \ = \ \alpha \circ G.f \quad \} \\
 & \alpha \ \in \ F.x \leftarrow G.x \\
\Leftarrow & \quad \{ \quad \text{natural transformation} \quad \} \\
 & \alpha \ = \ \eta_x \ ,
\end{array}
$$

  It is not necessary that the screen view is a perfect copy of the printed output, but the layout of the expressions has to be close enough to see when something goes wrong. The layout of those expressions is domain specific and you probably cannot produce such expressions efficiently with the standard document processors.

- **Flexibility**. The system should not impose any particular notational conventions but should be sufficiently flexible that notational conventions can be introduced or modified on the fly.

$((p \Rightarrow q) \Rightarrow r) \Rightarrow (p \Rightarrow q \Rightarrow r)$ versus $\dfrac{\dfrac{p \vdash q}{r}}{\dfrac{p}{q \vdash r}}$

Most existing mathematical editors are based on well established mathematical theories with a well defined layout and grammar. For such theories an optimised system with built-in properties can be built, which is dedicated to that particular theory. For a young and developing theory, such as the one used by the target user group, the layout and grammar is likely to change over time. Building a fixed system based on the current theory would have resulted in an outdated system by the time it was finished. By making the system flexible with respect to layout and grammar, the users are allowed to develop the theory without interference of the system being used. Since a system for editing mathematical documents might be of interest to others, the flexibility of the system can also be used for other editing tasks, with different mathematical notations or structures.

- **Writability**. The complete document should be edited within one interface and no special mathematical editing mode should be used.

$$
\begin{array}{ll}
 & (\star y \mid R[x := f.y] : P[x := f.y]) \\
= & \quad \langle \text{One point rule (8.14) — Quantification over } x \text{ has} \\
 & \quad \text{to be introduced. The One-point rule is the } \textit{only} \text{ rule} \\
 & \quad \text{that can be applied at first.} \rangle \\
 & (\star y \mid R[x := f.y] : (\star x \mid x = f.y : P)) \\
= & \quad \langle \text{Nesting (8.20) —Moving dummy } x \text{ to the outside} \\
 & \quad \text{gets us closer to the final form.} \rangle \\
 & (\star x, y \mid R[x := f.y] \wedge x = f.y : P)
\end{array}
$$

The density of mathematical expressions in the envisaged documents is very high. Expressions can be both inline and displayed, and a large expression, like a proof, can contain comments in the form of text (which can of course contain expressions again). In general, the user has to be able to combine text and expressions freely.

- **High quality**. The final output had to be of high quality, for use at conferences and in journals.

$\sum_{i=0}^{N} x^i$ versus `sum(x^i,i=0..N)`

TEX is one of the best typesetting systems for mathematical or technical documents and is often required or preferred by technical journals or publishers. To

be able to communicate with others and to publish articles, it has to be possible to generate correct LaTeX output, which has to be standard and readable. Publishers usually have their own restrictions on what LaTeX documents are allowed, since they want to format a collection of documents according to one specific style. This is only possible if the documents use standard constructions and don't require special LaTeX or TeX features.

1.4 Additional goals

During the development of the system, additional goals were stated.

- **Different output formats**. The developments regarding the Internet and its markup language HTML made it clear that the editor would also be useful for editing documents in other markup languages. In order to support the generation of multiple output formats, the system has to be extendable in a direction that wasn't foreseen at the start of the project.
- **Connections to systems**. As the initial users had no system support for verifying their calculations, making connections to other systems was not considered to be a high priority. However, several systems might benefit from an additional, document-based interface. Furthermore, some users had system support for their calculations and would benefit if there was an easy-to-use connection with their system.

1.5 The outline of this thesis

This project started as a masters thesis for a period of 6 months, in co-operation with Olaf Weber. The months became years and the system was extended and redesigned several times, both internally and externally. The final result is described in this thesis.

In the second chapter, an overview is given of existing systems for document preparation. As the design and redesign process took several years, the overview includes several systems that emerged during the development of Mathſpad. As the Mathſpad editor is specialised for complex mathematical document preparation, the mathematical capabilities of the document preparation systems are examined more thoroughly.

The functional properties are described in the third chapter and an explanation is given on how these properties were selected. As these properties determine how the user has to work with the system, they are quite important.

The next chapter describes how the system is implemented and which decisions were made. As the user of the system is not interested in the implementation details, the functional properties are more important than the implementation issues. That is, the functional properties should not change in order to make the implementation easier.

During the development of Mathʃƥadʃ, an additional priority arose in the form of connections to other systems, in particular to PVS. In chapter 5, several techniques for connecting different systems are discussed. After this discussion, one of the methods is selected and implemented in order to connect the Mathʃƥadʃ system to another software system. Again, the keyword is flexibility, as it is not known in advance which other systems might have to be connected to the Mathʃƥadʃ system in the future.

The final chapter gives the status of the current system and the areas where it can be used. As technology has advanced over the years, a short overview is given on which technologies could be selected today if a similar system has to be built again.

Chapter 2

Document preparation

MathʃpadMathʃpad should be a document preparation system for mathematically oriented documents, but should be general enough to allow document preparation of many other technical documents, without too many adjustments. This chapter describes common designs for document preparation systems in general and mathematical document preparation in particular.

2.1 Editing models

There are already a lot of document preparation or text processing tools, all with their own advantages and disadvantages. The collection of tools can be divided into several groups according to some design decision. Some of these decisions influence the rest of the tool so much that it is not possible to change it without rewriting the complete tool.

2.1.1 WYSIWYG vs. markup languages

A what-you-see-is-what-you-get (WYSIWYG) editor is based on the principle of direct-manipulation, where the user sees what the final result will be while the document is edited. With markup languages, the user creates a plain text file with special formatting commands and uses a special application to convert the text file into a formatted document, much like the document produced by a WYSIWYG editor. Both approaches to document processing have their advantages and disadvantages. Both have strong believers that their approach is the right approach. Both have users that would like to switch if it was possible and feasible.

WYSIWYG editor

The WYSIWYG editors have become standard at the same time that the graphical user interface became standard[66]. Without the graphical interface, the user had to switch between two versions of the document, one that was used to edit the document in a textual interface and one that represented the final output, either on paper or on screen.

With the graphical interface, the editor is able to combine the two versions of the document into one. The document can be displayed and manipulated without the need to make a hard copy to check the final output. This is possible due to the following principles:

- The fonts used to draw the content of the document are available for both the screen display and the printer output. By using the same fonts and same algorithms, both output formats will show the same result.
- Scalable fonts enable the use of large headers and small footnotes. Furthermore, the user is able to zoom in on the content of the document without the need to print it and inspect it with a magnifying glass.
- The editor can position the characters with a better precision due to the larger grid, that is, a pixel grid of 1024 by 768 versus a character grid of 80 by 25.
- Images can be displayed on screen, which allows adding all sorts of graphical information to a document.

Since the resolution of a screen is much lower than the resolution of the printer, true WYSIWYG is usually not possible, but if the differences are small enough, the average user will not notice them. However, if the differences are too large, users will be annoyed. So, if an editor wants to claim to be WYSIWYG, it should not allow any noticeable differences between the different output media.

Most WYSIWYG systems have a user interface similar to the one displayed in figure 2.1. At the top, there is a menu bar where all the menus can be accessed. Usually, there is a File menu for accessing, printing and converting documents, an Edit menu for actions related to changing the document, an Insert menu for adding new elements to the document, a Format menu for changing document properties, a Tools menu for advanced features such as spell or grammar checking and a Window menu for managing the different windows. Under the menu bar, there is a collection of buttons with images on them, which provide easy access to the more often used options of the menus. Usually, there are also some pull-down menus that allow the selection of properties like font style or font size. The pull-down menus often display the current properties of the cursor position. At the bottom, there is a status bar with additional information about the document, the cursor position and the system in general. Whatever is left unoccupied will be used to display the document itself.

The following editors could be regarded as WYSIWYG editors or are presented as such, although some of them are only partially WYSIWYG. As all these systems

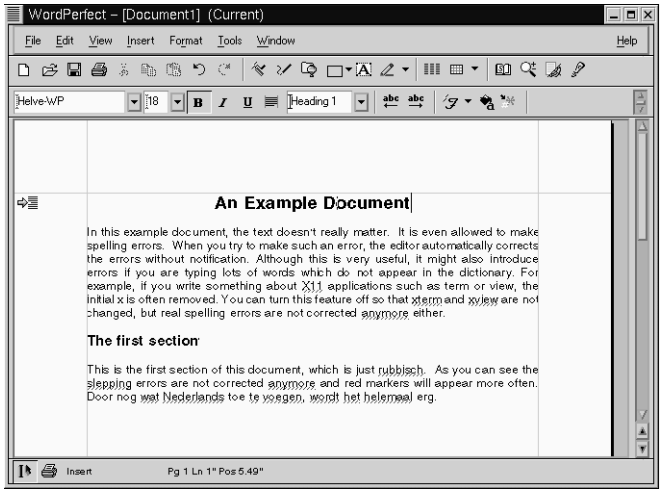


Figure 2.1: The interface of a WYSIWYG document preparation system.

work similarly and some of them are in a constant feature battle, the descriptions are rather short.

WordPerfect The desktop publisher made by Corel. It provides a feature to look at the internal document structure, in order to fix possible layout problems.

Word The desktop publisher made by Microsoft.

FrameMaker The desktop publisher made by Adobe, for long or technical documents.

Netscape Composer The editor integrated in the Netscape Browser, for editing documents in the HTML markup language. As HTML does not provide enough features to force a certain layout, the editor only shows what you will get, not what other people might get.

Adept Editor The editor made by ArborText for editing documents in the SGML and XML markup languages. Although SGML and XML are designed for specifying the document structure, it is possible to build a WYSIWYG editor for it when style sheets are applied. As SGML documents contain higher-level structures, the Adept editor provides different views on the system.

Scientific Word The editor make by MacKichan, for editing scientific documents. As the documents are still processed by the TEX system, it is not a true WYSIWYG system, but it makes life easier for novice LATEX users.

StarOffice The desktop publisher made by StarOffice/Sun, as a multi-platform alternative for the Windows-based desktop publishers.

This list is certainly not complete and many others are available, such as Thot, Grif, Amaya, LyX, KOffice and AbiSuite. However, those listed are supported by companies and have a reasonable user base.

Markup languages

A markup language is a language which uses special character sequences to instruct an application to perform certain operations on the plain text. By adding the correct sequences to the text, it can be formatted beautifully. By adding incorrect sequences, the result will look like a mess. Many markup languages are very powerful, but for the average user, it is very difficult to access all this power. This is probably due to the increasing complexity of the markup sequences when more complex features are used: making certain words in the text bold is easy, adding the rotated, upper-left part of a picture to a table is difficult.

Since users make mistakes in the form of typing errors, markup sequences are likely to contain errors, especially if an awkward syntax is used. As a result, the markup processor, designed to interpret and format a correct markup document, is used to remove these markup errors. This results in an edit-check loop to remove all the markup errors from the document, while the formatted output is usually ignored at this stage.

After all the markup errors are removed, the document is formatted according to the sequences added to the document and the formatted output can be checked for errors. If the user wants to do something complex, the trial and error approach is often used to get what is needed. This results in an edit-format-inspect loop, where the formatted output is inspected after small changes are applied to the document, until the formatted output is correct.

Due to the awkward syntax, documents with a very complex content, such as technical documents with many formulæ, become almost unreadable in the markup version. The only way to make sure that everything is correct, is by reading it again after it is formatted, with the possibility that the user reads what is expected instead of what is written. Therefore, a proof reader is needed to prevent such errors.

Since markup documents are formatted according to their markup and the markup interpreter often uses tables to store properties of the text, global changes to the document can be applied easily by adding the correct markup at the beginning of the document. Furthermore, these tables can also be used to calculate additional properties, such as the table of contents, the index or the references.

The major advantage of markup languages is the fact that markup documents are written in plain text files, which implies that any text-based tool can be used to manipulate the content of the document, such as the favorite text editor, a spell checker and many UNIX text tools such as `grep`, `tr`, `awk`, `sed`, `perl` and `sort`. These tools allow the user to perform global manipulations with the best tool for the job, without having to depend on the built-in tools of the selected editor.

Figure 2.2 shows some typical markup sequences for several markup languages. The

Item	LaTeX	Troff	texinfo
Chapter	`\chapter{...}`	`.sh 1 ...`	`@chapter ...`
Section	`\section{...}`	`.sh 2 ...`	`@section ...`
Paragraph	empty line	`.PP`	empty line
Bold	`\textbf{...}`	`.B ...`	`@b{...}`
Italic	`\textit{...}`	`.I ...`	`@i{...}`
Number Item	`\item`	`.IP n`	`@item`
Bullet Item	`\item`	`.IP \(bu`	`@item`
Table	`\begin{tabular}`	`.TS`	`@multitable`
Row separation	`\\`	newline	`@item`
Column separation	`&`	tab	`@tab`

Image	`\includegraphics{...}`	`.F+`
		`figure ...`
		`.F-`	

Item	**Lout**	**SDF**	**HTML**
Chapter	`@Chapter` `@Title{...}`	`H1: ...`	`<H1>...</H1>`
Section	`@Section` `@Title{...}`	`H2: ...`	`<H2>...</H2>`
Paragraph	`@PP`	empty line	`<P>`
Bold	`@B{...}`	`{{B:...}}`	`<B>...</B>`
Italic	`@I{...}`	`{{I:...}}`	`<I>...</I>`
Number Item	`@ListItem`	`^` or `+`	`<LI>`
Bullet Item	`@ListItem`	`*`	`<LI>`
Table	`@Tab`	`!block table`	`<TABLE>`
Row separation	`@Rowa`	newline	`<TR>`
Column separation	named columns	tab	`<TD>`
Image	`@IncludeGraphic{...}`	`{{IMPORT:...}}`	`<IMG src="...">`

Figure 2.2: Markup languages for common textual elements

markup languages are:

TEX A powerful markup language designed by D.E. Knuth[36], which is well-suited for scientific documents of high quality. The TEX system is mainly used in combination with macro packages.

LATEX A macro package for TEX, constructed by L. Lamport[39], which removes the burden to use all the low-level constructs from TEX. LATEX is the markup language preferred by many publishers of technical journals.

Troff A markup language based on roff and runoff[22], which dates back as far as the early 1960's. It was the main markup language on UNIX systems during the 1970's and early 1980's. It is still used for the online manual pages of many UNIX systems.

texinfo A markup language used for providing documentation on software[11]. It is mainly used by GNU software and provides hyperlink capabilities. The files are used to generate the online and printed version of the documentation. The TEX system is used to produce the printed version.

SGML The standard general markup language defined by ISO[68], defined for standardisation of document exchange. The structure of documents is defined with a document type definition (DTD), while the appearance is defined with a style sheet written in the document style sheet specification language (DSSSL,[71]).

HTML The hypertext markup language defined by the WWW Consortium[81], used for most of the documents on the Internet. The HTML language is defined by a DTD and is therefore a specific application of SGML. As the layout requirements differ per user and system, a standard style sheet is not available for HTML and only recommendations are given on how the layout could be. However, the browser wars and user requirements have led to new versions of HTML with more layout oriented features, such as frames and cascading style sheets (CSS, [6]).

XML The extensible markup language defined by the WWW Consortium, which is the platform for future versions of HTML. XML can be seen as a simplified version of SGML, based on the experience with HTML. With XML, it is possible to construct a modular markup language, as opposed to the monolithic HTML specification. Such modularisation is required in order to serve all the future applications of providing information over the Internet, such as database access, content labeling, mathematical content and vector drawings. XML has its own versions of style sheets (XSL), document definitions (Schemas) and scripting facilities (XFA) [8].

RTF The rich text file format defined by Microsoft[76], used to exchange documents with other applications. As the format can and will be adjusted by Microsoft when required, it is not very useful. Furthermore, the RTF format is extremely verbose and the markup is very layout specific. Therefore, RTF is mainly used

for exchanging Word documents with applications that are unable to handle the latest proprietary Word format.

SDF The simple document format, designed by Clatworthy[14]. SDF can be used to generate documents in multiple formats by using existing filters.

Lout A high-level language for document formatting, designed by Kingston[33]. With the Lout system, a document can be converted to PostScript, PDF or plain text. Lout is influenced by the eqn and Scribe systems [31, 61].

To exchange markup documents, they are usually converted to a document format which is more portable, as the markup processor might not be available to the receiver of the markup document. Common distribution formats are PostScript[2] and Portable Document Format (PDF), which contain the final layout of the document and which can be used to make hard copies.

2.1.2 Structured vs. unstructured.

In a structured editor, the content of the document is stored in some kind of structure, such as a table or tree. This structure is used to determine properties of the contents or process the content in some way. In an unstructured editor, everything is stored as one sequence of characters with a special tag structure to add the properties.

All common text editors work with unstructured text, although the editor might perform syntax highlighting to indicate the intended structure based on the underlying markup language. Many WYSIWYG editors also work with unstructured text, where special features are added by superimposing attributes on top of the text.

As SGML is a rather verbose markup language which imposes a structure on the text, many SGML editors are structure editors, where the DTD is used as a guidance for the possible structure. However, as SGML-based documents usually consist of text, the structure is often hidden from the author by providing the same functionality as in unstructured editors or by automatically adding the structure when possible. As the number of structural elements is quite small, such structure editors are still usable.

For technical documents, such as program files or research papers, the structure of the document and the rules to manipulate that structure are more complex. As a result, it is difficult to write such an editor by hand, especially if support for multiple languages is required. Therefore, several systems exist to generate structure editors for a given language definition:

Centaur A generic interactive environment generator, that produces a language-specific environment from the formal specification of the programming language. Centaur is mainly used in universities and research centers for experimenting with languages.[29]

Synthesizer Generator A tool for creating syntax directed editors and interfaces.

This generator is originally developed at Cornell University and is now maintained by Grammatech.[63]

ASF+SDF Meta-environment A system for generating language-specific environments from algebraic specifications of (programming) languages. This environment is related to the Centaur system.[34]

As a definition of the language is used to generate the syntax directed editors, all the editors will contain the same features, although they are adjusted to the given language. When the language is defined with sufficient details, the editor could support term rewriting, type checking, code optimisation and code generation.

Structure editors, especially syntax directed editors, are often quite difficult to use, as the document has to obey to a certain structure. If the structure elements are difficult to access or the operations are restricted by the structural integrity, the editor is useless, as the use of a normal text editor in combination with existing compiler tools would be just as easy. Therefore, many structure editors provide a text editing mode[42], where the author can manipulate the structure as if it were text. Afterwards, the text is parsed and converted to a structure.

Although the use of a structure editor is more difficult, there are several advantages of working with structured documents.

- As all objects in the document have some structure, you can quickly find all the objects with the same structure, such as theorems, figures or section headers.
- The structure can be used to fold the content of the document, where large structures are shown as a comment indicating the content of that structure. For large documents with many sections, each section can be folded such that only the header is displayed, as is done in a Mathematica notebook[83]. For programming languages, a fold operation can hide the content of a function definition, thereby making the document more easily accessible, as is done in the Oberon system[62].
- The structure is easy to select, which reduces errors when the structure is copied. In a text editor, the correct range of characters has to be selected, which might go wrong due to missing certain characters or due to incorrect indentation.
- The information generated with the structures is less likely to contain errors, as the structure is correct by construction.

2.1.3 Modal vs. modeless

Most keyboards have about 100 keys, which is usually not enough for a text editor. The text editor needs keys to insert characters and symbols and keys to perform edit actions. Since the number of edit actions increases with the number of features, the editor has to reuse the keys in a clever way. Some editors do this by using different

modes: in insert-mode, you can add characters to the document, in command-mode you can perform edit commands. Other editors use combining keys or prefix keys: after pressing a special key, the next key is interpreted differently. Almost every editor uses one of the two methods.

As an alternative to the keyboard, an editor can create graphical input methods like symbol palettes, popup menus and preference windows. The difficulty with these graphical input methods is the need to leave the home row[1] of the keyboard and reach for the mouse or function keys, which will reduce the editing speed.

The Emacs editor mainly uses combining keys or prefix keys for the editing commands, although it also uses modes for specific features, such as incremental search, where normal keys will be inserted in the search string, instead of the plain text. As Emacs places the cursor in a reserved part on the screen while in such a mode, it is not confusing to the user. Emacs also supports major modes and minor modes, where two modes are active at the same time: the major mode defines the standard commands, the minor mode defines document specific commands.

The VI editor uses three main operating modes: command mode, input mode and external mode. In command mode, the keyboard is used to move the cursor, to search for something or to delete something. In input mode, characters are inserted into the text. In external mode, `ex` commands can be executed. The user switches from command mode to external mode with the `:` key, from input mode to command mode with the `escape` key and from command mode to input mode with one of the text entering commands, such as insert (`i`), append (`a`) or change (`c`). From within command mode, the `!` key can be used to run UNIX filter programs, such as `sort`, `awk`, `sed` and `tr`.

For entering plain English text, these methods don't differ very much as all the characters are available on the keyboard. However, if you want to enter more complex languages or technical text in some markup language, the keyboard interface has to be more sophisticated in the form of input methods[40, 72]. Some techniques for input methods are:

one-to-one Each key is bound to one character. This technique is sufficient for languages with a limited collection of characters, but it can easily be combined with other techniques.

dead-keys Certain keys change the behaviour of the next sequence of keys, in order to construct a single character. During the construction of the character no or limited feedback is given to the user. This technique is often used for entering characters which can easily be constructed by combining simpler characters, such as `dead-^ a` to get â or `compose 1 2` to get $\frac{1}{2}$.

trailing modifiers Certain modifier keys are used to change the character that is entered with the previous key. Usually, modifier keys represent accents such as `^`, `"` or `'`. As these keys are also used as normal keys to enter those characters,

[1] The home row of the keyboard is the default row where you place your fingers in the 10 finger system

the `space` key is used to prohibit the modifier key to change the previous character. For example, the sequence `"Bu"ro "` can be used to enter "Büro".

set selection While keys are pressed, the set of characters to select from is reduced. At any time, the user can select a character from the set and start again. This technique is used for languages that require large collections of characters, such as Chinese or Japanese. The method of reducing the sets is based on some ordering on the characters, such as stroke count or transliteration.

complex analysis The text is analysed with complex algorithms in order to solve ambiguous input. The Japanese input method is a good example: the text is first entered in a phonetic form, such as Hiragana or Katakana, with either a special Japanese keyboard or by using some transliteration like Romaji. In the second stage, the phonetic form is analysed to convert it to the final form. As Japanese is context dependent, this analysis might be wrong and the user might have to select one of the possible final forms. To improve the analysis, the selection of the user is used by the analyser to improve itself, for example by suggesting the final forms in a different order.

encoding position The encoding position is used to enter the character. After a special prefix key, the octal, decimal or hexadecimal position of the character is entered with the normal keys. As it is rather difficult to remember these positions, it should only be used as fall-back technique for very few characters. For example, the sequence `Alt 0163` constructs £ under Windows95 with the Latin1 encoding.

As each language can have its own input method, multiple modes have to be used when a document contains multiple languages. As mathematics can be regarded as a language, a mathematical input mode might be required.

2.2 Mathematical documents

Mathematical documents are rather difficult to edit[12]. In fact, any document with something which is not plain text, a table or an image, is difficult in most document processing systems. For WYSIWYG systems it is almost impossible to add large amounts of mathematical expressions to a document and be able to work with it properly. For markup languages, the situation is usually much better, provided that the language provides constructs for mathematical content and is compact enough to remain readable.

2.2.1 WYSIWYG systems

Many WYSIWYG systems use a special editor for entering mathematical content. However, in a mathematical document, there are so many formulæ that it is impossible to use a separate editor. As each formula is a separate object within the

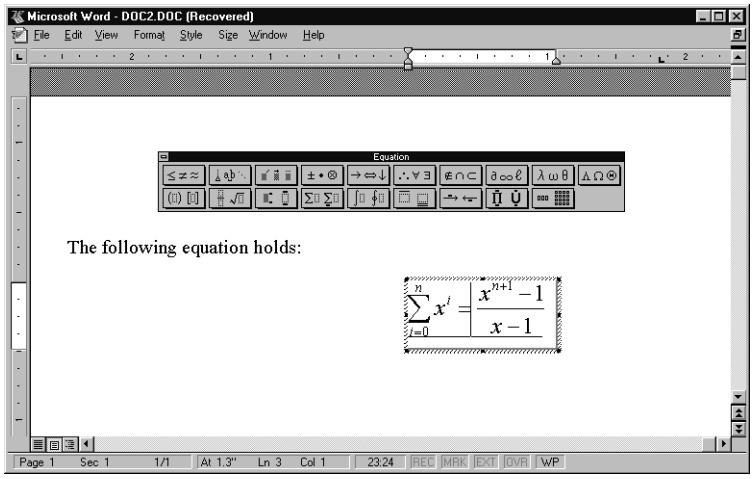
document, the special editor has to be started in order to change the formula. In most mathematical documents, formulæ are often very similar and a decent copy facility is needed in order to exploit that. However, many systems use a clipboard when something is copied and require that the source is first selected and copied to the clipboard, after which the target is selected and the clipboard content is pasted. As each selection requires that the special editor is started, mathematical editing is tedious.

In the Microsoft Windows environment, OLE objects[46] are a common way to enter special content into documents and a special editor allows the user to edit these objects in-place, giving the impression that the editor handles the objects natively. When the OLE object is selected, the special editor is started and the interface is modified to provide access to the special features for those OLE objects. For documents with many mathematical objects, editing becomes tedious due to the constantly changing interface and the difficulty of copying formulæ. Furthermore, mathematical objects can be used inline or displayed, where the inline versions require proper alignment, while the OLE system is more suited for displayed material, such as images or tables. Finally, the OLE system becomes unstable when too many OLE objects are used within a document. Therefore, the OLE system is not well suited for documents with many formulæ or non-standard presentations. For example, the Eindhoven style of proof presentation mixes formulæ and prose in a complex way. This style originated from Feijen and is extensively used in the book by Dijkstra and Scholten[18]. Examples of this presentation style are given in section 1.3 and in figure 5.10 on page 110.

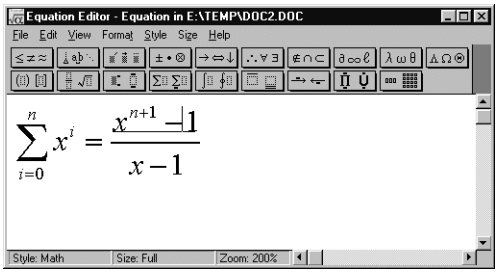
A common editor for mathematical OLE objects is MathType[44], which allows in-place editing as well as editing in a separate window. In the MathType editor, common mathematical symbols and notations are available through menus and keyboard shortcuts. With these menus an expression can be constructed and it is displayed in a WYSIWYG fashion on the screen. After the expression is finished, it is entered into the document by either closing the editor or changing the focus.

WordPerfect version 6.1 provides its own mathematical editor, which cannot be used in-place. This editor gives the user two views on the same expression, one markup view and one WYSIWYG view. Only the markup view can be edited and the markup is based on troffs eqn extension. The available markup constructions are listed in several menus and can be entered by either selecting them from the menu or by typing them in. At any time, the user can request to update the WYSIWYG view, which requires the markup to be parsed and interpreted. After the formula is finished, it can be added to the document by closing the formula editor. Within the document, formulæ are handled as OLE objects.

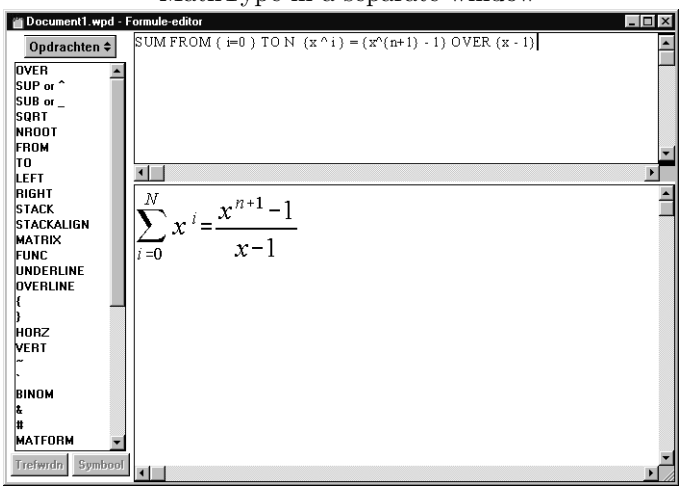
StarOffice is intended to be compatible with Microsoft Office and handles formulæ similar to OLE objects. A formula is edited in a separate window by giving the correct markup, which is similar to the markup used by the WordPerfect mathematical editor. The symbols and notations are selected from menus or palettes, where the menus are ASCII based while the palettes display the resulting symbols. While the markup is manipulated, the in-place WYSIWYG version is constantly updated. As



Microsoft Word with in-place editing



MathType in a separate window



WordPerfect with two views

Figure 2.3: Several systems for editing mathematics

the formulæ are handled as OLE objects, similar problems occur.

In FrameMaker and Scientific Word, the formulæ are handled in-place and no special editor is needed to enter them. A formula is constructed by using menus with common mathematical symbols and notations, similar to the menus available in MathType. As no special editor is needed, both systems are more useful for editing mathematical documents than most WYSIWYG systems. However, for non-standard layout, they are still difficult to use and there are also some problems related to switching between inline and displayed expressions.

To summarise, the following problems are common in WYSIWYG system when mathematical content is concerned.

- Due to the clipboard copy operations, it is difficult to reuse existing formulæ.
- As formulæ are handled as special objects, several textual facilities are missing for formulæ, such as searching or changing the font size.
- When a formula is changed from inline to displayed, its position is changed incorrectly.
- Non-standard mathematical layout is difficult to accomplish.

2.2.2 Markup languages

For markup languages, the support for mathematical content depends on whether it is possible to add special fonts, whether the layout algorithm is powerful enough to construct mathematical notations and whether macro definitions are allowed. When these basic requirements are available, the mathematical content can be added to a document using a plain text editor. Since all the content of the document is written in plain text, all the functionality available for text is also available for the mathematical content, such as cut-and-paste, search and find-and-replace.

As the mathematical content is all entered with markup sequences, the formulæ quickly become unreadable, especially in documents with many similar formulæ, which is very common. Therefore, the markup language should not be too verbose in order to keep the document at least manageable.

The table in figure 2.4 shows some typical markup sequences for mathematical notations. The LaTeX markup language is listed because LaTeX is used very often in mathematical document preparation. The Troff/eqn markup language is listed because it is often used as a basis in other systems, such as WordPerfect, StarOffice and Lout.

HTML version 3.0[59] allows mathematical markup as well. However, that version of HTML has never been accepted by the major browser manufactures and the mathematical markup disappeared in the accepted HTML version 3.2. A working group of W3C has since defined the MathML markup language[28], which can be used in combination with XML. In MathML, it is possible to use presentation markup or

Item	LaTeX	Troff (eqn)
x^n	`x^n`	`x sup n`
a_i	`a_i`	`a sub i`
$\frac{x}{y}$	`\frac{x}{y}`	`x over y`
$\sqrt{x}$	`\sqrt{x}`	`sqrt x`
$\sqrt[3]{x}$	`\sqrt[3]{x}`	not possible
π	`\pi`	`pi`
$\sum_{i=0}^{n} x^i$	`\sum_{i=1}^n x^i`	`sum from i=1 to n x sup i`
$\lim_{i\to\infty} i$	`\lim_{i\rightarrow\infty} i`	`lim from {i -> inf} i`
$\vec{a}$	`\vec{a}`	`a vec`
$\overline{x+y}$	`\overline{x+y}`	`x+y bar`
$R \oplus S$	`R \oplus S`	not possible
$a \xrightarrow{*} b$	`a\stackrel{*}{\rightarrow}b`	`a -> from ^ to * b`
$\left(\frac{x}{y}\right)$	`\left(\frac{x}{y}\right)`	`left ( x over y right )`

$\begin{matrix} a & b \\ c & d \end{matrix}$	`\begin{array}{cc}`
	`a & b \\`
	`c & d`
	`\end{array}`
		`matrix {`
		`ccol { a above c }`
		`ccol { b above d }`
		`}`

Figure 2.4: Markup languages for common mathematical elements

content markup language, both of which are extremely verbose and not intended to be written or read by humans in their ASCII form.

2.3 Mathematical systems and theorem provers

Mathematical systems and theorem provers used to have teletype interfaces and often still do. In these interfaces, mathematical formulæ are entered using ASCII characters, even if the system uses a graphical user interface. After a formula has been entered, it can be evaluated and the result will return either in the same notation as the entered formula or in a nicely formatted graphical layout. Due to the use of ASCII, there is a large difference between conventional mathematical notation and the notation required by the system. Furthermore, each system uses its own notation, which makes it very difficult to exchange between systems.

Most of these systems are specialised in calculational methods and the document preparation part is less well developed. When the system does not support documentation facilities, it is often required to enter the formulæ in a document preparation system again, which is usually a markup language like LaTeX. Some systems allow automatic generation of such LaTeX markup sequences, which reduces the chance of errors in the conversion. Furthermore, it relieves the user of learning an additional markup language. However, the method of combining these generated markup sequences into a document is tedious and error sensitive, especially if the sequences are changed or regenerated.

Some of the common mathematical systems are:

Maple An advanced computational system for handling symbolic, numeric and modelling problems[60]. More recent versions also allow some document preparation. In general, a document consists of three different items: comments, Maple input and Maple output, where the comments are used for documentation purposes. With Maple, it is possible to generate LaTeX and HTML output to allow incorporation of Maple results into LaTeX documents or for publishing online. Maple can be used with either a graphical user interface or a command line interface.

Mathematica A tool for scientific research, analysis and modelling[83], with more facilities for document processing in the form of Mathematica Notebooks. A notebook is a structured document with input and output elements. Folding operations can be used to hide certain information and it is possible to add interactive elements such as buttons and animations. Mathematica provides a graphical interface and allows other applications to use the Mathematica kernel through the MathLink library.

MatLab An environment for doing technical computing, combining numeric computation, advanced graphics and visualisation[45]. The MatLab system also includes tools for building graphical user interfaces. Document preparation is not the strongest point of MatLab, but a separate program exists for generating

reports. MatLab uses a command line interface, but it allows the construction of graphical user interfaces.

MuPAD A computer algebra system for symbolic and numerical computations[23]. MuPAD has no support for document preparation and uses an ASCII based input and output format.

For extending the system, they all provide a tailor-made programming language for handling the mathematical objects that are used within the system.

Some of the common theorem provers are:

PVS A specification and verification system, intended for formal verification of software and model checking[65]. PVS is a large and complex system which uses Emacs as user interface. It uses an ASCII based input and output language and is able to represent proof trees in a graphical format. For document preparation purposes, PVS expressions can be translated to LaTeX markup.

Isabelle A generic theorem proving environment, which is extended by using object logics[55]. Isabelle uses a command line interface, but it can be used within Emacs through the Proof General interface. For document preparation purposes, Isabelle is able to generate LaTeX markup.

Coq A proof assistant based on the calculus of inductive constructions[27]. The Coq system uses a command line interface and uses ASCII-based notations. There are several interfaces built around the Coq system, such as CtCoq, which allow mathematical notations. The CtCoq system is generated by the Centaur system and is a structure editor. Coq provides tools for converting Coq listings into LaTeX documents and for processing LaTeX document containing Coq phrases.

Jape An interactive tool designed to help with learning, teaching and using formal reasoning[73]. It is one of the few theorem provers which requires mathematical symbols and does not have a command line interface. The authors have recognised that using the correct notation is a vital requirement in order to make it useful in education. Jape does not support document preparation itself, but it provides several ways to generate LaTeX markup or encapsulated PostScript. It is also possible to extract Jape files from LaTeX documents.

These systems provide their own configuration or programming language as well.

Both the mathematical systems and theorem provers use their own notation, which makes it difficult, if not impossible, to switch to a different system, to verify obtained results or to combine the strength of a mathematical system with a theorem prover. The MathML[28] and OpenMath[9] markup languages are recent attempts to make the exchange between these systems easier, as each system would only have to provide an interface to the common markup language in order to make information exchanges possible.

2.4 Software documentation

One problem with maintaining software is poor documentation, which is caused by the fact that the documentation and the program are usually two separate documents. Combining those two documents into one document resembles combining mathematical calculations and documents. In both cases, the document consists of two interwoven parts, the documentation and the program text or mathematical content. With special applications, those parts can be separated into nice documentation and executable programs or interpreted mathematics.

There are several systems for combining documentation and programs:

Web The literate programming system developed by D.E. Knuth[35]. According to Knuth, the system should be used for explaining to a human being what the program should do and that description should not necessarily have to follow the restricted program text. It is therefore possible to describe the program by using tags to indicate that certain parts will be defined later. After the interwoven document is separated, the documentation is formatted by TEX and the program is processed by a compiler. Due to the use of these tags, the resulting documentation contains lots of references to used variables and program parts and a useful index is created. However, the resulting program text will be unreadable due to the way the code is distributed in the interwoven document and the way software tools generate feedback messages. Of course, the source should not be edited directly, as it would go against the purpose of using the Web system. One major problem with the Web system is that the user has to learn three languages in order to use it successfully: the TEX language for the document preparation part, the programming language for the program part and the Web language for instructing the programs that extract the two parts.

POD The documentation format used by the Perl system[82]. The POD format allows easy addition of plain old documentation to Perl scripts by using simple markup sequences. When the Perl interpreter reads a Perl script, it ignores the text following POD markup, up to the `=cut` sequence, which indicates that Perl code is entered. As a result it is easy to combine the documentation and the program, as no special languages have to be learned. Although the systems allows the documentation and Perl code to interweave without problems, it is mostly used for appending the documentation to the end of a Perl script or module.

Javadoc The documentation format for Java programs[38]. Javadoc uses documentation comments in Java to generate the documentation for a Java program. By providing a set of guidelines on how to add comments to Java class definitions, sensible documentation can be generated in the form of HTML files. In the comments, plain HTML can be used to improve the layout and special tags are available to refer to other parts of the document or Java class.

These three systems are specially constructed to improve the documentation of software. Of course, it is always possible to add the documentation in the form of comments, which is a common way to do it, but the additional utilities would be missing, such as indexing and cross referencing.

There are several Web-based systems which allow combining documentation and program text. However, these system use a less revolutionary approach, as they usually don't disturb the order in which the program is written.

Revision control systems, such as SCCS[5] or CVS[10] are useful tools to improve the documentation of software when it is maintained by a larger group of people. When the program is changed by several people, it is important that the documentation remains up to date and revision control systems enforce that changes to the system are documented.

2.5 Conclusion

In this chapter, several available systems are reviewed which are related to documentation preparation or processing mathematical material. The WYSIWYG document preparation systems are very useful for plain documents without too much technical content, such as mathematical formulæ. The markup languages are very flexible and allow large quantities of technical content. However, the technical content quickly becomes unreadable due to the use of all the markup sequences. The mathematical systems have powerful facilities for doing calculations, but they provide limited facilities for document preparation, especially if the presentation style is not supported by the system, such as the Eindhoven proof presentation style, shown in figure 5.10 on page 110.

As the survey has pointed out, sufficient support for WYSIWYG document preparation of technical documents is missing, especially if it concerns non-standard presentation styles. Although it is possible to prepare such documents with a markup language, the readability of a markup language is too low to allow easy preparation. In the following chapters, a document preparation system is designed and implemented which supports semi-WYSIWYG document preparation of technical documents with non-standard presentation styles.

Chapter 3

The functional design

In this chapter, the functional design of the MathJpad system is given, which determines how the user will see the system, how the system is operated and thereby what kind of authors could be using the system. As each design decision influences the usability of the system, the decisions have to be taken with care and with respect to the intended user group, authors of technical documents with mathematical content. As each user group is special in some respect, the resulting system might be so specialised that only the intended user group can use it. However, it would be nice if the result can be used by a larger group.

Certain decisions were already fixed at the start of the project, as choosing differently would be against the requirements.

- The LaTeX system will be used for producing the printed document, as LaTeX documents are often required by publishers of technical journals.
- A graphical user interface is required, as the mathematical content should be readable on screen. A teletype interface would restrict the number of available symbols and the means of positioning them.
- A UNIX operating system is used as implementation platform, as the intended users, like many researchers, use UNIX.

In short, the main design is an almost WYSIWYG document processing system for mathematical LaTeX documents running under UNIX.

3.1 Program versus user in control

For a document editor, it is very common that it has control over some part of the document since the document has to obey certain properties, such as correct page numbers, correct references or fitting within a page. When the editor thinks

something is wrong, it mentions it to the user who has to take notice of it and solve the problem. Most users, however, don't like it when a machine tells them what they can and cannot do and want to take control when they see fit. If an editor hyphenates a word incorrectly, the user needs to take control and adjust it. In general, if an application does something automatically, the user has to be able to overrule it.

Examples of programs that take control are most structure editors. Adjusting the structure freely is nice, but it's wrong. So, the user is only allowed to adjust the structure according to strict rules. If the user knows a shortcut, it is not possible to take it. It seems inevitable that such structure editors are not widely used.

Since most users know what they are doing, the editor should not obstruct the user in doing something strange. The editor might protest at first, but it has to allow the user to take control and perform the strange operation, after which the user is responsible for anything that might be incorrect as a result.

Decision 1
The user should be in control of the system, not the other way around.

3.2 Text editing versus structured editing

For document preparation, there are two editing models that are often used, the text model and the structured model. Both models have their advantages and disadvantages.

3.2.1 Text editing

Almost all users are familiar with editing plain text in the text model. If you want to add some text at a certain location, you just have to position the cursor and enter the text with the keyboard or by some other method. To the user, the text seems to be either one large list of characters or a list of lines, each containing characters. Since the text model is easy to understand and to implement, most text editors and document preparation systems are based on this model.

Markup languages, including programming languages, make use of this easy to understand model to add structure to the unstructured text. A special application interprets the markup to create the actual structure and use that. The advantage of this approach is that any text editor can be used to enter the markup text and no special tool has to be provided for that. Since there are dozens of text editors, a user is likely to find one that fits the needs. One problem with the use of markup languages is that it is very easy to make typing errors, which usually results in multiple runs of the interpreter application to remove them.

Since markup languages add structure to the text, some text editors, like Emacs, interpret the markup to highlight certain parts of the text with different colours or styles. The advantage is that the user has visual feedback on whether the markup is

correct or not. This does not mean that the editing model is changed: it is still plain text and the markup is still visible, except that the editor continuously determines what the structure would be.

Some advantages of the text model:

- It is easy to understand.
- It is easy to implement.
- It can contain anything by using correct markup.

Some disadvantages of the text model:

- Complex operations are difficult.
- Markup languages make the text unreadable and error sensitive.
- Mathematical content is difficult to add.

3.2.2 Structured editing

In the structured editing model, the content of a document has to obey a specific structure and the editor helps the user in creating a correct document. Most structure editors work with a fixed set of structure elements and rules on how these elements are combined. During an edit session, the user is allowed to enter only those elements that are correct at the focussed position, to ensure that the entered structure is syntactically or semantically correct.

Structured editing is often difficult, since most users are neither familiar with the structure nor with the rules. To overcome this problem, many structure editors have a free editing mode, which is based on the text model. After a free edit session, the constructed text is converted to a structure and checked for correctness.

If the editor is optimised for a specific set of rules, it can perform extra checks to make sure that other properties hold, such as semantical correctness. Whenever there is something wrong, the user will be notified to correct the problem. Typical examples of such editors are the ones generated from a language specification.

After a user has created a correct structure, the editor often allows operations on these structures, which are easier to implement due to the available structure. For programming languages, it might include code generation, optimisation or interpretation.

Some advantages of the structure model:

- The structure makes certain operations, such as selecting, very easy.
- The structure is always correct.
- Errors are difficult to make.

- Complex operations are easier.

Some disadvantages of the structure model:

- The user is often restricted too much.
- The editor is in control.
- The elements and rules are often fixed to allow optimised operations.

Decision 2
Since MathPad has to handle documents with many mathematical expressions, a combination of the two models seems to be an obvious choice: the text model for the document and the structure model for the expressions. Such a combination has the advantages of both models and reduces the disadvantages. The user can choose between the best model for a specific job. For mathematical documents, one might assume that the user has some knowledge about the structure.

3.3 Fixed grammar versus flexible grammar

For structure editing, a grammar is needed, that is a collection of items and rules to combine them. If the grammar is known in advance, the editor can take advantage of it by providing operations specific for these items and rules. If the grammar is not known in advance, the editor has to provide a method to define the grammar or allow the user to change the grammar interactively.

3.3.1 Fixed grammar

Most structure editors use a fixed grammar, which makes it very specialised for that particular grammar. Since it takes a lot of work to construct such a structure editor from scratch, tools have been constructed to generate a structure editor from a given grammar [63, 29, 34, 57]. Since the grammar is fixed, the structure editor can be optimised for that particular grammar by providing extra functionality which is not available in other editors, such as structure rewriting, optimisation and incremental compilation. For example, a structure editor for Pascal can provide rewrite and optimisation functions specifically for Pascal.

The major advantage of the editor generator is that you only have to maintain the generator and a collection of grammars in order to maintain a collection of structure editors. One disadvantage is that it does not allow the user to modify the grammar, since everything is built in and optimised. For example, the user cannot define rewrite rules for user-defined functions. Furthermore, the grammar for a language might be so complex or large, that the generated structure editor is not very usable.

Some generated structure editors provide incremental attribute evaluation and an attribute can be a semantic check, the generated code or a layout feature. In an incremental evaluation, the attributes are only calculated for the structure that has changed and any structure that depends on the changed structure. The continuous evaluation of the attributes ensures that errors are noticed at the moment they are entered and not in the final processing step. For a user, this might be handy, since the errors directly indicate where adjustments are needed. However, certain errors can generate lots of errors, such as missing variable declarations. If the error messages are displayed in the same window as the structure that generated them, the document will become less readable due to the additional information and the changes in layout.

3.3.2 Flexible grammar

Some structure editors are based on grammars that are loaded when the editor is started or might even allow the user to change the grammar interactively. Since the grammar is not known in advance, the editor cannot be optimised for one specific grammar and might not be able to provide the special functions mentioned in section 3.3.1. However, since the grammar is not fixed, the user is able to adjust the grammar where needed without restarting the editor.

If the grammar of the language is not known in advance, a structure editor that allows a flexible grammar is the only useful solution. Often, if the grammar is not fixed, tools such as incremental evaluation, optimisation or rewriting are not very useful. Markup languages which allow macro definitions are examples of grammars which are not fixed.

Decision 3
Since there is not a fixed grammar for the mathematical theory of the target users and elements are added on a regular basis, the grammar of Mathʃpad cannot be fixed. The grammar specific operations that are common in generated structure editors do not yet apply to this mathematical theory, so there is not much lost in not supplying those operations. Furthermore, generated editors are often not very strong in supplying text editing operations or layout features (such as special symbols or different sizes), which is a must if the mathematical structures have to be combined with plain text.

3.4 Clipboard versus multiple selections

During editing, the user often has to copy or move certain parts of the document to some other location. There are different methods to perform these operations.

3.4.1 The clipboard

Many systems make use of a clipboard. A clipboard is temporary storage to which something, usually text, can be copied. With the operations Cut and Copy, the user is able to move or copy the selected part to the clipboard. With the Paste operation, the clipboard is copied to the selected position. This seems to work fine in general, but there are some caveats.

First of all, some systems have multiple clipboards. For example, under the X11 window system, you have 3 clipboards:

- The primary selection, used by many X11 applications and usually automatically set.
- The clipboard, used by OpenLook applications, where a Cut, Copy and Paste are needed.
- The cut buffers, used by Emacs and automatically set.

For copying or moving the selection to the clipboard, this is not a problem, since it can be copied to all the clipboards. However, for pasting the clipboard into a document, it depends on the application that has set the clipboard which clipboard should be used.

Second, copying information from one location to another location involves making two selections and performing two clipboard operations. If these focus changes involve complicated operations, such as starting an application in an OLE environment, it might be frustrating to the user if it occurs too often.

Third, the item in the clipboard might not always contain what you expect. In the X11 windows environment, the clipboard can also contain the identity of a window. In order to switch to a different window before pasting the clipboard, the target window is often selected first, which might have set the clipboard to contain the identity of the window. In general, it is not clear what the clipboard contains or which clipboard is used if multiple clipboards are available.

3.4.2 Multiple selections

A different approach is to allow multiple selections and copy the content of one selection to another selection. Since the two selections are simultaneously visible, the user can check what is being copied before the operation is performed. This reduces the cognitive overhead.

For mathematical editing, copy and paste operations are very often used. If these operations take time or are unreliable, the constructed mathematical expressions are likely to contain errors. Therefore, the multiple selection approach seems to be better.

Most copy operations will be performed within Mathʃpad, that is, both the source and the target are Mathʃpad expressions. Copying to and from the general clipboard

is therefore not needed. For those situations where the general clipboard is needed, a special copy or paste operation can be used. Since most applications are not able to handle the Mathʃpad structures correctly, the clipboard is only used to export or import textual representations.

Decision 4
Since copy operations occur very often in mathematical editing, the multiple selection approach is used. To allow interaction with other applications, the clipboard approach is also supported.

3.5 WYSIWYG versus markup codes

In a WYSIWYG system[19, 66], the user sees the document exactly as it is published, at least in theory. In a system based on markup codes, the user manipulates a plain text file and describes how it should look like and a markup compiler converts the plain text into the final document. Both versions have their pros and cons.

3.5.1 WYSIWYG

Most users are familiar with WYSIWYG systems, since they either use them or have strong objections against them. Since the system directly shows how the content of the document will look, it is a nice system for a novice user. Just enter the text and add all the features. Once it looks like what you want, you can print it and have a perfect copy of what you see on your screen. In practice, many WYSIWYG systems[84, 16] have some problems with that. If different engines or fonts are used for creating the output to the screen and the printer, it is very likely that the WYSIWYG metaphor disappears, leaving the user with the task of solving a difficult problem: if the screen shows what you want, but the printer does not print what you want, you have to find a way to let the screen show exactly the same, but with the correct printed version.

In a WYSIWYG system, users are often more concerned with the actual layout than with the content of the document[39]. One of the problems that occurs due to this layout centred method is that changing the document afterwards will be difficult and time consuming, since all the effort that was put in the correct layout will be destroyed. Problems often occur when the following methods are used to achieve the correct layout.

- Using hard page breaks to move to the next page. The system automatically divides the document into pages. If such a soft page break seems to be incorrect, the user can insert a hard page break to get the correct result.
- Using tabs and spaces to align text vertically. Some systems do not have an obvious way to achieve vertical alignment, so users apply this method, that usually works in a WYSIWYG system.

- Using newlines to get correct line breaks. For most systems, a newline indicates the end of a paragraph and it influences the standard line breaking algorithms. By adding a newline, a paragraph is split into two paragraphs.

- Adding fixed references to document elements, such as sections, pages or figures. When document elements are added, reordered or removed, all fixed references have to be checked for correctness.

- Using font and size changes to create special headers. Often these changes are not applied consistently and changing them afterwards is laborious.

If one of the above methods is used, all changes to the document require that the complete document is checked for errors. For some systems, trying to print it on a different printer means reviewing the document, since the typeface or paper size might have changed.

For mathematical or technical documents, WYSIWYG systems are often very limited and do not provide the features needed for correct mathematical typesetting, such as inline expressions, line breaks within expressions and references to other expressions. Furthermore, mathematical typesetting is very laborious due to missing features like adjusted copy and paste operations. In many systems, mathematical content is handled as a special image, which disregards the fact that, within a document, mathematical objects are closely related to each other, something which is not common for images.

The following methods are currently available to allow mathematical content in a WYSIWYG editor:

- The mathematical content is entered with an equation editor using a markup language. The user is presented with a window with two views on the expression: a markup view and a WYSIWYG view. The user is only able to edit the markup view, while the WYSIWYG view is updated on demand. After the user is finished, the content is inserted in the document and handled as a special image. The equation editor of WordPerfect is an example of this approach.

- The mathematical content is entered with a WYSIWYG equation editor. The equation is converted to some image format and inserted in the document. This method works independent of the document editor, since most of them support the use of images. The equation editor from K-talk is an example of this approach.

- The mathematical content is entered as an OLE object. The document editor allows general OLE objects as part of a document and a special equation editor is able to handle mathematical objects. From the document editors point of view, the mathematical object is just an OLE object, which usually implies that the object is handled as an image. An example of an OLE editor for mathematics is the Equation Editor from Microsoft.

- The mathematical content is an integrated part of the document and no special equation editor is needed. Due to the integration, the user never has to change between different applications and different command syntaxes. Furthermore, the editor can handle the equations as mathematical objects instead of images. An example of an integrated environment is FrameMaker 5.0.

All the methods that need a special equation editor are very difficult to use if the document contains many expressions, since copying existing expressions from the document into the equation editor is cumbersome due to all the switching between different programs.

3.5.2 Markup codes

In a markup language, the user edits a plain text with special codes to tell the markup interpreter how the text should be typeset. With these markup codes, the user has full control over the final layout, where the interpreter does a decent basic job and the user has to fix a few details. The interpreter makes sure that the output will be correct according the provided markup, which might adjust the default font or the text size to obey a certain style. After the document is finished, the user converts the document with the references to specific fonts, paper and printers. If a different printer or font is selected, the interpreter can be used to reformat the entire document according to those changes, giving warnings where problems might have occurred. After running the interpreter, inspecting the output and adjusting the markup a couple of times, the final output is considered correct.

The markup codes can also be used to automatically generate content where needed, such as references to pages, sections and figures, a table of contents, a list of figures, an index, a glossary or a bibliography. The interpreter calculates those elements and adds them to the documents at the appropriate places. If the document would change, these elements are recalculated to make sure that they are always correct.

Most markup languages are extendable by defining new markup codes. With these new codes, the user is able to adjust the language to local needs, making sure that the document is consistent.

A markup language seems to be a very good alternative to a WYSIWYG system. However, to enter the text with the markup codes is often difficult and error sensitive. Each user has to learn the markup language, which can be very complex and markup errors occur very often, especially since the markup text is less readable compared to the created output or a WYSIWYG system.

Decision 5
The advantages of WYSIWYG systems are combined with the power of markup languages to decrease the number of cycles in the edit-process-check loop. In the hybrid combination, the user can work in a quasi-WYSIWYG document and add markup code in situations where the system does not support the intended features. Since the document can contain markup codes, the WYSIWYG parts of the document are converted to markup codes and the markup interpreter is used to get the final document. The use of structure editing can reduce the number of syntax errors in the markup codes, since the structure can be used to generate correct markup.

3.6 Keyboard handling

For text editing, the keyboard interface is very important, as it is the main interface to manipulate the text and perform operations on it. Since different text editors use different keyboard interfaces and users are familiar with some text editor, Mathʃpad has to allow the user to adjust the keyboard interface, such that a specific keyboard interface can be constructed or partially simulated.

For mathematical editing, the accessibility of mathematical symbols is important and they should be available through the keyboard for easy access. However, the collection of mathematical symbols is very large and not everyone uses the same symbols. Therefore, a user has to be able to adjust the keyboard interface to those symbols, such that the most often used symbols are most easiest to access. Furthermore, some users might prefer a mathematical mode for entering expressions rather than prefix keys.

Some users prefer the keyboard rather than the mouse, so it has to be possible to perform mouse operations through the keyboard as well. Especially the selection of special symbols or mathematical notations should be possible through the keyboard interface, as these will occur very often for certain subsets of symbols and notations.

Decision 6
The keyboard interface has to be highly adjustable to the preferences of specific users. In fact, most of the input methods mentioned in section 2.1.3 should be supported by the keyboard handler.

As all the target users are familiar with the Emacs keyboard interface, that interface is used as a start. Although Emacs is much too complex to integrate all its functionality into Mathʃpad, the most common operations from Emacs have to be available and work similarly.

Chapter 4

The technical design

After the functional design decisions are made to determine how the user will interact with the Mathʃpad system, a technical design is needed to determine how the Mathʃpad system works internally. As the technical design is of no interest to the user, it should not influence the functional design, unless the functional design cannot be implemented without certain small sacrifices.

4.1 Text structures

The plain text of a document has to be stored in a way that it can easily be handled and processed. Common approaches to storing editable text in memory are the buffer-gap method and the list of lines method, as shown in figure 4.1

The buffer-gap method In the buffer-gap method, the complete text is stored in a buffer as one large sequence of characters, with a gap at the location where the edit actions take place. If the edit location changes, the gap is moved by moving the text that lies between the original and the new location of the gap. Since edit actions are usually local, the gap only has to move over small distances. If the gap is completely filled, the buffer is expanded and a new gap is created. Since all characters are handled equally, the buffer gap method is rather easy to implement.

The list-of-lines method In the list of lines method, the text is divided into lines and each line is stored separately. Often, the length of a line is limited by some magic number, like 255, and each line will use the same buffer size. Operations on lines, like cursor movements, are simple to implement in this model, but other operations, like searching or word wrapping, are more complicated.

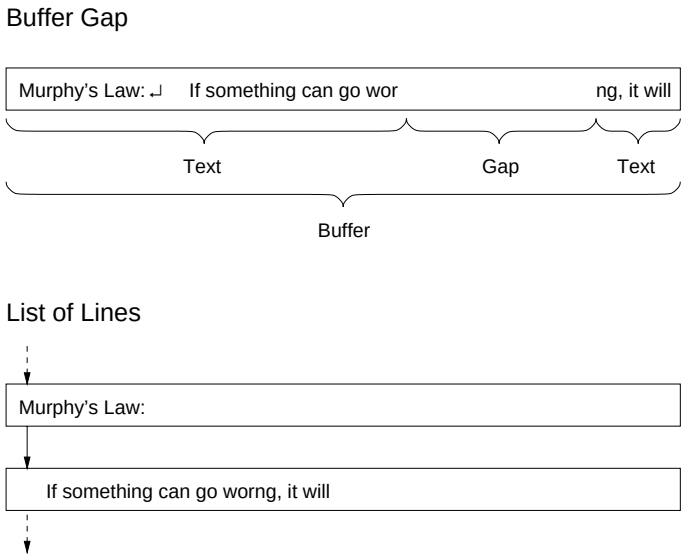


Figure 4.1: The buffer-gap and list-of-lines methods.

A combined method Both methods have their drawbacks, but it is possible to combine the two methods, for example by dividing the text into logical parts and using the buffer-gap method on those parts. A logical part might be a line, a paragraph, a chapter or a memory block. The main advantage is that the buffer does not need to be as large as in the full buffer-gap method and it might allow the text to be larger, although such memory management should be a task of the operating system. The main disadvantage is that two different memory models are used, which complicates algorithms and introduces all kinds of exceptions on buffer boundaries.

Decision 7
Since the buffer-gap method is the easiest method to implement and has no major drawbacks on the performance of the basic edit actions, this method is used to store the plain text.

The buffer-gap method is implemented with the following type:

```
BufferType = {
      buffer : String;    % contains the text
      size : integer;     % size of the buffer
      gap : integer;      % size of the gap
      gappos : integer;   % position of the gap
}
```

4.2 Character encoding

The character encoding is very important, as technical documents contain a lot of special symbols. To be able to handle those symbols correctly, the character encoding either has to contain them or has to support a means of adding them. The only official character encodings that contain sufficient special symbols or allow additional symbols, are the Unicode and ISO-10646 encodings[77, 69], which are similar and designed to (eventually) contain all the characters used around the World. By using the Unicode encoding internally, it is possible to access almost every symbol. However, there are a few disadvantages: Unicode is not widely supported yet and it requires double-byte strings to handle it efficiently (if UTF16 is used). On the other hand, since the number of required symbols easily exceeds the magic number 256, a double-byte encoding is needed anyhow. Furthermore, the use of a private encoding would also not be supported, with the main difference that support for it is unlikely to be added to operating systems or toolkits at all.

Decision 8
The Unicode encoding is used internally to be able to support all the mathematical symbols.

Although Unicode support is still missing on most UNIX systems, there is a movement towards Unicode support in several systems, toolkits and programs. Until sufficient support for Unicode is available, the system should include basic support on its own.

Unicode contains large collections of characters which are very difficult to support correctly, such as the Arabic, Hebrew and Japanese writing systems. Supporting these correctly is not the main target of this project, but it would be nice if support for these scripts could be added in the future.

An earlier version of the MathPad system[80] used a font-based encoding. Although it worked correctly, there were compatibility problems when fonts were added. For example, different fonts could contain the same symbols, which results in strange behaviour of search functions, as visually equal symbols are not equal within the system. In general, encodings which combine several encodings by using escape sequences, as in ISO-2022, Extended UNIX Code (EUC) and the wide characters of UNIX [70, 53, 32], are difficult to extend with new characters and difficult to manipulate properly due to state information.

4.3 Mathematical expressions

For mathematical expressions, the use of some kind of tree makes manipulating expressions easier, since the structure is always available. What kind of trees to use

depends on the operations that are needed and the content that should be allowed[1]. Since the information that will be stored is not known a priori, the tree structure should allow many common mathematical structures and easy selection and manipulation. Certain mathematical structures, such as arrays and graphs, might not be easy to select or manipulate when stored as a tree, but for such structures, there should at least be an improvement compared to the use of plain text.

In general, a tree structure consists of one root node which defines the complete expression. Each node in the tree contains a collection of subnodes and some information on how to combine those subnodes. In a tree, it is not possible to have cycles, since this would result in an infinite recursive structure, which is impossible to display.

Binary trees In a binary tree, each node has at most two subnodes and the information to combine the subnodes is usually fixed and known in advance. Since a tree can only contain two subnodes, the structure is very easy to implement and manipulate. Figure 4.2 gives a visual representation of the expression $A * (B + C)$. In a tree, the structure of the expression is clear and parentheses are not needed.

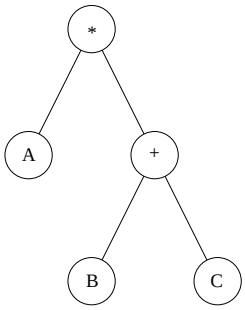


Figure 4.2: The binary tree representing $A * (B + C)$.

Certain mathematical structures are difficult to represent with a binary tree. For instance, the expression $a \leq b < c$ cannot be stored in a binary tree in a logical sense, since it is an abbreviation of the expression $a \leq b \wedge b < c$. A user might want to select either $a \leq b$ or $b < c$ in the abbreviated expression, which would not be possible with binary trees unless a sophisticated reshape mechanism is available. Other structures need more than two arguments to be valid, such as most of the large operators, like $\sum_{i=k+2}^{N} x^i$, where three or four arguments are required.

N-ary trees In an $n-$ary tree, each node has up to n subnodes, with information on how to combine those subnodes. Since the number of subnodes is still fixed, it is still rather easy to implement. Figure 4.3 shows an n-ary tree for the previous sum

[1]The design of trees was constructed in co-operation with Olaf Weber, as part of a combined masters thesis[80]

expression, where the summation node contains four subnodes and the equals sign is part of the summation node.

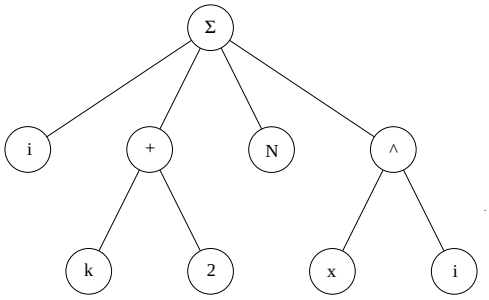


Figure 4.3: An n-ary tree.

A fixed number of subnodes might not be enough. Mathematical calculations often involve very large expressions, consisting of a list of similar expressions, separated by relators. Since the number of subexpressions is not known in advance, an n-ary tree will not be sufficient to store those expressions. Furthermore, different relators might be involved, which have to be part of the node.

Rose trees In a rose tree, each node contains a list of subnodes. Since the list of subnodes is not restricted by any size, each node in the tree can contain an unlimited number of subnodes. It seems that rose trees do not have limitations with respect to the common mathematical notations, since all subexpressions can be stored in the list of subnodes and the node itself can contain the information on how to combine these subnodes in the correct manner.

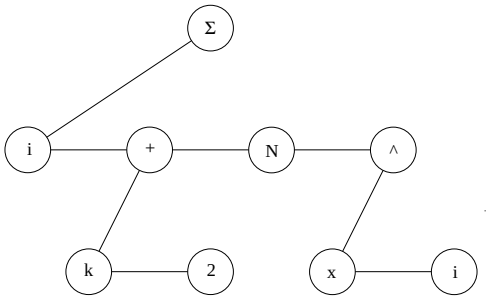


Figure 4.4: A rose tree.

Figure 4.4 shows the rose tree equivalent to the n-ary tree from figure 4.3. The horizontal links between the nodes construct the list of subnodes, while the vertical links create the tree structure. To allow expressions like $a + b + c$ and $a \leq b < c$,

the binary operators are moved one level down, to be part of the list of subnodes, as shown in figure 4.5. In this example, the superscript operator in x^i is not regarded

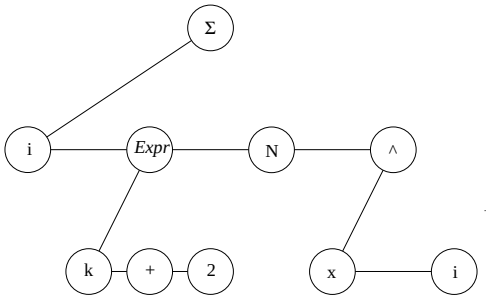


Figure 4.5: A rose tree, enabling larger expressions.

as an infix binary operator, as it behaves differently. Due to its two-dimensional layout, the left expression of the operator would require parentheses, while the right expression doesn't, as in $(x+1)^{i+1}$. Furthermore, as an infix operator, it would require that the right expression should be displayed in a small font, which requires that any node can change the appearance of non-subnodes. It would be possible to build some mathematical knowledge about such operators into the system, but that would result in hundreds of exceptions, as the user is allowed to extend the system with new notations and operators.

Decision 9
The mathematical expressions are stored in a rose tree, where the binary operators are part of the list of subnodes.

Each node has to contain some information on how the subnodes are combined, which is stored in a formatting string. The formatting string contains place holders indicating where the subnodes have to be inserted. For identifiers, the string consists of the name of the identifier. For operators, it consists of the definition of the operator and for expressions, it consists of a list of place holders. An example is shown in figure 4.6, where the small rectangles indicate the place holders.

4.4 Combining text and expressions

A document can contain both text and expressions and there needs to be some way to combine these two objects. Due to the flexibility requirement, the user should be able to combine text and expressions freely. That is, an expression is allowed to contain text and the text is allowed to contain expressions.

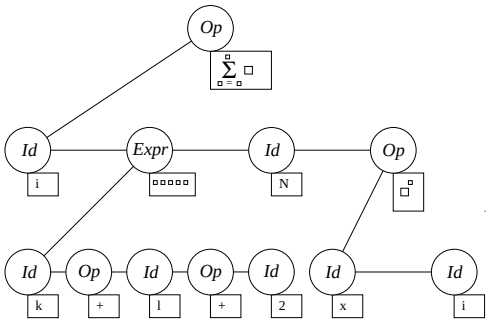


Figure 4.6: A rose tree with formatting labels at each node.

To allow text inside expressions, the text should be handled as a valid subnode of an expression. This can easily be achieved by defining a subnode for text, which contains the text itself as formatting information.

To insert expressions into the text, the text might be split up into two parts with the expression in between, or the text might contain place holders to indicate where the expression needs to be inserted. Since a mathematical text usually contains many expressions, the first method would divide the text into many small pieces, which removes the advantages of the buffer-gap method. Using meta-symbols does not have this disadvantage, but it introduces problems related to linking the expressions to the right meta-symbols. However, to enable code reuse and to keep the document model simple, all the nodes are handled in the same way by expanding the structure where needed. Since the formatting string contains place holders to indicate subnodes, the text will contain place holders to indicate positions where expressions have to be inserted. Since the buffer-gap method is used to store the text, this method will also be used to store the identifiers and expression sequences.

Decision 10
Text is stored as a special kind of node from a rose tree, where the expressions in the text are stored in the list of subnodes and the text itself is used as the formatting information.

The main advantage of this resulting document model is the fact that most text operations can be applied to expressions without modifications. For text, a selection consists of a sequence of characters. For expressions, a selection consists of a sequence of expressions and operators, which are represented by special characters, the place holders.

By treating text as a special node, one general node structure suffice to store the different types of nodes:

```
NodeType = {
      format : BufferType;  % contains the text
      kind : integer;       % kind of node (Expression,
                            %  Operator, Identifier, Text)
      first : NodeType;     % first sub-node
      right : NodeType;     % node to the right in the list
}
```

4.5 Displaying a document

To display a document, the information in each node should be sufficient to make a correct projection from the list of subnodes to some output medium, for example the screen or a file.

The standard method to generate formatted output from a tree is by doing a tree traversal with recursion, as the output of a tree consists of the output from the subtrees, separated by formatting information from the root node. This method works for both the screen output and file output. However, a document can get very large and regenerating the screen output after every edit action would take too much time. To optimise the regeneration of screen output, the recursive tree traversal is replaced with a non-recursive traversal, which can start and stop at any position in the tree.

Decision 11
The tree drawing algorithm uses a non-recursive algorithm to enable partial screen updates.

In general, there are two methods to store locations in trees, such that they can be used to walk through the tree and that they store the location uniquely. The first method stores the path from the root to the referenced node, usually by storing the sequential position of each subnode. These paths are rather difficult to store and manipulate. The second method only stores the referenced node, while the tree contains the information to extract the path from that node to the root node, which makes operations on paths from the root node to the referenced node more difficult. Furthermore, the tree has to contain extra information and store a reference to the parent node in each subnode. However, in the selected tree structure, either the subnode or the formatting string has to contain information about the position where the subnode has to be used. By storing the information in the subnode, that information can also be used to construct the path from a node to the root node.

Decision 12
A position in a tree is stored as a reference to the specific node. The tree itself contains the information to walk through the tree in the form of references to parent nodes.

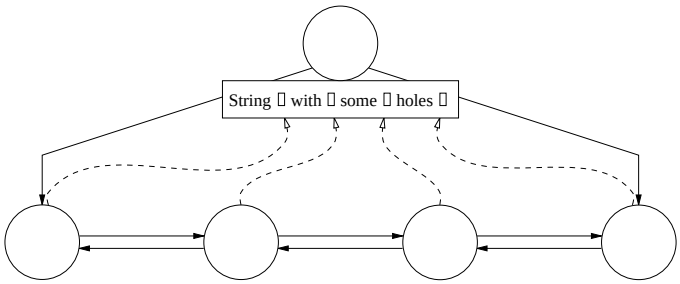


Figure 4.7: The final tree structure.

Figure 4.7 shows the general tree structure. A node contains a string with holes which determines how the tree is displayed and where the subnodes have to be added. In this example, the string contains four holes and four subnodes. Each node contains references to the first and last nodes of the list of subnodes. Since each node is also part of a list of subnodes, a node also contains references to the next and previous nodes in that list, thus forming a doubly linked list of subnodes. To be able to walk through the tree, a node contains a reference to the parent node, which has as attribute the position of the place holder within the string of the parent node. These parent references are drawn with dashed lines, as they might change due to modifications in the string of the parent. The type definition of such a reference, called a `Marker`, is:

```
MarkerType = {
      node : NodeType;   % the node it refers to
      pos : integer;     % position within the format string
      next: MarkerType;  % next Marker referring to the same node
}
```

The type definition of a node then becomes:

```
NodeType = {
      format : BufferType;     % contains the text
      kind : integer;          % kind of node
      father : MarkerType;     % parent reference
      first,last : NodeType;   % first and last sub-node
      left,right : NodeType;   % left and right neighbors
      list : MarkerType;       % list of markers referring
                               % to this node
}
```

The `list` attribute is used to update the markers that refer to this node, which is needed when the format string of this node changes. The list contains all the `father` markers from the subnodes and possibly some markers related to selections or caching.

The non-recursive traversal takes a marker and increases it according to the visual representation. The following pseudo-algorithm gives the general structure of such a traversal, which starts at marker A and ends in B, assuming that A and B are valid markers with the same tree:

```
|[  p := A
  ; DetermineState(p)
  ; do p≠B  →
          Process(p.node.format.buffer[p.pos])
         ;if p.pos = p.node.size  →  p := p.node.father
                                     ;p.pos := p.pos + 1
          □ IsPlaceHolder(p.node.format.buffer[p.pos])  →
                p := Subnode(p)
          □ else → p.pos := p.pos + 1
          fi
    od
]|
```

Here, $DetermineState(p)$ restores the state for a given marker p, $Process(c)$ changes the state according to the character c, $IsPlaceHolder(c)$ determines whether character c is a place holder, $Subnode(p)$ returns the marker referring to the first character in the node attached to the place holder at marker p. Figure 4.8 shows the traversal path though the tree representing "Einsteins formula $E = m \cdot c^2$ relates energy to mass.", where every node in the tree is represented by its formatting information. With the non-recursive algorithm, the traversal can start at any location within the tree and move forward. The algorithm for moving backwards is similar.

Since screen updates occur very often, it is the first location for optimisation, as a flickering screen due to updates is very annoying. The following methods have been used to improve the screen updates:

- When text is entered, only the line with the changes is updated. Usually the first part of the line will not change, so that part is only used to calculate the correct positions, but it is not actually redrawn.
- When the user selects a part of the document, only the part that visually changes has to be updated.
- It is possible that the height of a line changes due to an additional character, which requires that the complete line has to be updated, together with the text below that line. To handle this correctly, the height of each line is cached for reference.
- When the user clicks somewhere on the screen, the mouse position has to be translated into a position within the document, which is known as origin

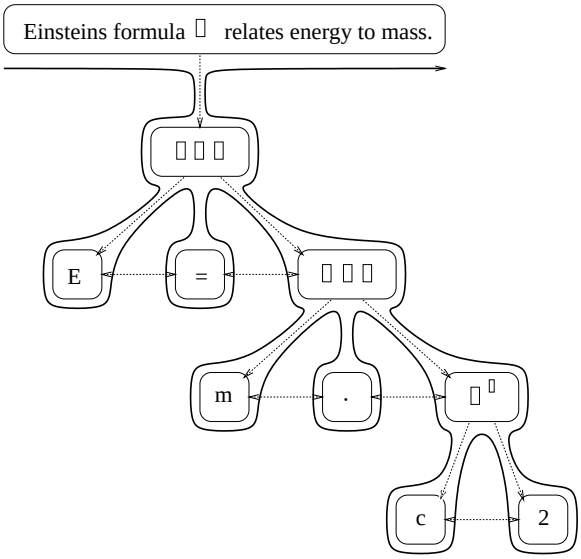


Figure 4.8: A traversal through a rose tree

tracking. Instead of caching all the information that is needed to update the screen, the screen update function is used to trace the given position. The cache with the line heights is used to improve its performance.

- Since tabbing positions and margins are important for many layout structures, but difficult to reconstruct dynamically, tabbing positions are also cached, just like the height of each line.
- To determine the correct state of the screen update routine, such as which font is used, only formatting information on the path to the root node is taken into account. This means that formatting information of a node only affects the subtrees of that node.

Other methods for improving the screen update time are not used, due to their memory requirements and complexity. These methods include:

- The results of the screen update function could be cached, such that the next time, the layout calculations are not needed. When such a cache is used, it will contain a copy of the document that is displayed and this copy has to be synchronised with the original, which becomes very complex. Furthermore, this copy would require a large amount of additional memory, especially for the mathematical content, which requires font switches and additional layout information.
- The window system, X11 in this case, can handle some of the screen updates by keeping a copy of the screen[51]. However, there is no guarantee that the

window system will actually support this, as it requires sufficient memory, which might not be available. Therefore, the application needs to have a fall-back algorithm, in case this support is missing.

- The screen update could use the double buffering method[21], where the screen update is first drawn in a separate image and then copied to the real location. Since the window system can be used over a network, the separate image has to be part of the window system, otherwise the image has to be transported over the network. Again, the window system might not have the required memory to support this, and a fall-back algorithm is needed.

Since a fall-back algorithm is needed, that algorithm has to be optimised in order to improve the screen update time. However, if this fall-back algorithm is fast enough, it can be used for all purposes and other methods are not needed.

The optimised algorithm is fast enough for normal purposes and works sufficiently in a variety of environments: low-end PCs with Linux or FreeBSD, X terminals with limited memory, remote X sessions (over a modem) and X emulators under MS Windows.

4.6 Templates

The tree is formatted according to the strings contained in each node, so the mathematical operators and notations have to consist of at least some formatting string. Entering these formatting strings by hand will lead to errors and inconsistencies. Instead, a method is needed to faithfully reproduce these formatting strings without adding complexity to the interface. In accordance to the templates used for repeatedly drawing the same shapes in civil engineering, templates are introduced to repeatedly apply the same formatting strings.

Decision 13
Mathematical operators and notations are represented by templates, which contain the information needed to format those entities correctly.

In order to format combinations of operators correctly, the *precedence* of the operators is important, as it determines where the parentheses have to be added. The system should add these parentheses automatically, but the user should be able to overrule the system, as the user is in control.

For better readability of large expressions, the *spacing* around operators is important, as it suggests the structure of the expression. Although the use of default spacing around certain operators improves the readability, such as the spacing rules in LaTeX, the spacing should depend on the size and structure of the expression, as incorrect spacing would suggest the wrong structure, thereby introducing confusion to the reader. Therefore, each operator can be surrounded by default spacing, the system

can add spacing according to the structure and the user can overrule the spacing that is calculated, as the user is in control.

Instead of copying the template specific information into the tree structure for each instance of the template, a reference to the template is used. This will reduce the memory consumption, since the formatting strings are shared, but has the drawback that there will be extra complexity due to the two different memory models, namely, the editable and uneditable formatting string. However, allowing the user to modify copied formatting strings would certainly introduce inconsistencies in the appearance of operators and notations and it should be prohibited anyhow.

To allow templates to appear within the tree structure, the following additional attributes are needed:

```
NodeType = {
      ...
      stencil : boolean;   % does the node use a template?
      template : pointer;  % reference to the template
      opspace : integer;   % user added operator spacing
      parens : boolean;    % add parentheses?
}
```

4.6.1 Versions

Certain operators can be presented in different ways and the user has to be able to select the appropriate presentation for a given context. For example, the division operator can be presented as /, ÷ or - and the integral notation depends on whether it is definite or indefinite and whether it is used inline or displayed on a separate line. Adding support for these different presentations is easily achieved by allowing templates to have multiple formatting strings and allowing the user to switch between those formats.

Decision 14
A template can contain several versions to allow multiple presentation of the same mathematical concept.

As an example, the integration template is used. The following table shows some of the versions that such a template could contain.

Description	Layout
Indefinite, displayed integral	$\int \square\partial\square$
Indefinite, inline integral	$\int \square\partial\square$
Definite, displayed integral	$\int_{\square}^{\square} \square\partial\square$
Definite, inline integral	$\int_{\square}^{\square} \square\partial\square$

It is possible to add more versions, such as contour or double integrals.

4.6.2 Stencils

Since the mathematical context is not known in advance, the user and maintainer have to be able to create or adjust templates. To manage all those templates, related templates can be collected in files, which are called stencils, analogous to the sheets with templates cut out of them, used in chemistry and civil engineering.

Decision 15
Related templates are stored in stencils.

How these templates, versions and stencils are used is the responsibility of the user and maintainer. The user can decide to create a template with equality and inequality as two versions of that template to be able to switch between them more easily. Mathematically, these two relations are not two representations of the same concept, but the user has the choice to create two different templates, one for each relation.

The templates and stencils are similar to macros and style files in LaTeX, where a macro is used to abbreviate a common construct and a style file combines related macros for easy management.

4.6.3 Manipulating templates

Since the user might be working in a field where notational conventions change, the system has to allow the user to modify the templates and versions of templates. This has implications on how templates are used internally and how documents are stored. The following problems might occur:

- A template can be modified while it is used in a document. After the modification, the document has to be updated to use the modified template.
- The number of place holders in a template might change. If a place holder is removed, the content of that place holder is lost. If a place holder is added, all occurrences of the template will contain empty place holders.

- A template might be modified while a document is not loaded. For consistency reasons, the occurrences of that template in such a document should be updated when the user visits that document. Since modified templates might corrupt a document, the user should also be able to load the document without updating the modified templates.
- Old documents might still use templates which are not accessible anymore in any of the available stencils.
- Several stencils might contain the same template. If a template is updated, it should be updated in all stencils.
- The number of versions can be changed. For example, a version can be removed, which could corrupt the document.

The same problems also occur in markup languages that allow the user to define macros, such as LaTeX, troff or XML. If a macro is modified or removed, the document that uses it can become incorrect. Similar problems occur with user-defined functions in programs or libraries.

When a template is modified, the document can be updated by correcting all references to that template. Since a user might be working on several documents at the same time, updating all those references might take some time. By using an intermediate table, all the references can be updated at once by updating that table. Moreover, this requires that no additional actions are needed when templates are changed.

If a place holder is added to a template, any occurrence of that template will require an additional argument to fill that new place holder. As it is impossible to fill in the place holder automatically, the user has to fill it in later. To use the table as suggested in the previous paragraph, the subnode that corresponds to the place holder has to be added automatically when needed.

If a place holder is removed from a template, the content of that place holder will be lost or at least become inaccessible. As it is possible that the place holder has been removed by accident, the user should be able to restore the original state by restoring the template. Therefore, the content of a disappearing place holder is not removed from the tree. Instead, it will remain inside the tree until it is actually removed. Therefore, the tree does not have to be updated when a place holder is removed.

If a version of a template is removed, any occurrence of that version in a document will become incorrect. Therefore, the user should remove such occurrences before the version is removed from the template. If the removed version is still used in a document, its occurrences will be replaced by a different version of the template, which might corrupt the document. Therefore, users should be aware of the problems that might arise when a version is removed from a template.

If a template appears in a document that is not loaded at the moment that the template is changed, it should be possible to update the document while it is loaded in order to have a consistent use of templates. To achieve this, each template will

receive a unique number, which is used during the lifetime of that template. A document on disk will contain that unique number to allow correct retrieval of the used template.

If a user changes the order of versions within a template, the system has to be able to retrieve the correct version when the template is used. Therefore, each version from a template will receive a unique number on construction, just like the template. With these unique numbers, the system is able to decide when two templates are equal and which versions are related.

If templates change over time, old documents get unmanageable due to missing or incorrect templates. To ensure that old documents remain usable, each document will include the templates that are used in that document. When the document is loaded, it is possible to replace the templates from the document with the ones from the stencil files. However, if the stencil files and their templates have changed, the document could be incorrect according to these modified templates. In that case, the document can be loaded without any stencil files, such that the templates from the document itself are used.

It is possible that a template appears in multiple stencil files. If a template is updated in one of the stencil files, then it should be updated in all of them in order to keep documents consistent. When all the stencils containing the modified template are loaded, this is not a problem, as all stencils can be adjusted at the same time. However, if there exists a stencil with the modified template which is not loaded, that stencil has to be synchronised once it is loaded. Since a stencil might also be used as a backup for a template when the user is experimenting, it should be possible to load a stencil without synchronising it.

To handle all of the above problems, the system contains a database of templates, where the unique number of the template is the key to the database. The database contains all the templates that are needed to handle the documents and stencils. Since documents on disk can contain templates, some of the templates in the database might not be part of any stencil. The following strategy is used when a template is loaded from disk, either from a document or from a stencil.

- If the template does not appear in the database, then it is added to the database.
- If the template does appear in the database, but it is not part of any of the loaded stencils and it originates from a stencil, then the template from the database is replaced by the new template.
- Otherwise, the template is ignored and the one from the database is used.

Thus, templates from stencils have precedence over templates from documents. As it is possible to load a document when no stencils are loaded, the templates from the document are always accessible.

The precedence of stencils over documents introduces a new feature, although it is quite difficult to use. Different stencils can contain templates with the same unique

number, but these templates might be different in each stencil. By switching between the different stencils, different templates can be used for the same document, thereby changing the appearance of the document. The stencils are in this case used as style sheets, defining how templates should be displayed.

4.7 Generating markup output

Mathematically oriented documents can be edited on screen in a readable fashion. However, a user might want to make a hard copy of the document, submit it to a journal or conference, make it available on Internet or send it to a colleague. This requires that the document is converted to a suitable format, for example, PostScript to make a hard copy or LaTeX to submit it.

For text, this conversion is straightforward. Since the user is allowed to add plain markup sequences to the text, the system assumes that the text either contains correct markup sequences or does not contain any markup sequences at all. Either way, plain text is not modified while the markup output is generated, unless some characters are not available in the character encoding of the markup language.

For the mathematical part of the document, which is created using the templates, it is impossible to generate correct markup output without at least some hints from the author of these templates, as the information that is used to display a template on screen is usually not sufficient. Therefore, the author of a template has to define the markup that has to be generated for that template, where the place holders are used to indicate the position of the arguments. This markup definition resembles the formatting information that is used for displaying a template on screen. Therefore, they are both called *formats*. Thus, the *screen format* defines how a template is displayed on screen, while the *output format* defines how a template is converted to the output medium or markup language. To remove the burden of defining the output format for each and every template, the system is able to generate the output format automatically if the screen format is not too complex. When this automatically generated output format is incorrect, a user who is fluent in the markup language can modify the output format where needed.

For the target users, the LaTeX markup language is most important, as it allows high quality output and it is used by technical journals and conferences. Therefore, the generation of LaTeX output has received extra attention, as that output should be error free and ready to submit to a journal. This last requirement indicates that the generated LaTeX markup should be similar to hand-written LaTeX markup and that the use of additional macros is restricted. This complicates the LaTeX generation algorithm, due to the following features in LaTeX:

- LaTeX uses a *math mode* to typeset mathematics correctly.
- There are several ways to switch between text mode and math mode. For example, the commands `\begin{math}`, `$`, `\(`, `\begin{displaymath}`, `$$`, `\[` and `\begin{equation}` can be used to enter math mode.

- The user can define new commands that also switch between the different modes.
- Certain symbols are only available in math mode.
- Other symbols are only available in text mode.
- Switching between the different modes might introduce additional spacing.
- It is possible to nest the different modes, for example to have text within mathematics within text.

Since the use of math mode is rather complex and error sensitive, the output generator tries to solve the problems that are introduced by these features. To release the user from the burden of knowing which symbol can be used in which mode, the output generator will switch between the two modes where needed. To achieve this, the generator has to keep track of the current mode, which is almost impossible due to the freedom of entering LaTeX markup in plain text. Therefore, the generator assumes that for each expression in the text, the generator has to switch to math mode and for all text within an expression, it has to switch to text mode. Since the output format of a template might contain LaTeX markup to switch to a different mode, the author of the template can notify the output generator of such an event, to ensure that the generator works correctly. When a symbol is used which is not available in the current mode, the generator will temporarily switch to the correct mode and add that symbol. Since each switch to a different mode might introduce additional spacing, the number of temporary switches is minimised by combining them where possible.

As the markup output is generated less often and performance optimisation is not an issue, the output generation algorithm is recursive, where the output format is used as a guideline for the recursive calls. In pseudo code, the generation algorithm $GenerateOutput(A)$ behaves as follows, assuming the output for node A is needed.

```
[[  s , i := FormatString(A) , 0
  ; if IsTextNode(A) → EnterTextMode()
    □ else → EnterMathMode()
    fi
  ; do s[i]≠0 →
        if IsPlaceHolder(s[i]) →
            GenerateOutput(Subnode(s[i]))
        □ else → Output(s[i])
        fi
       ;i := i+1
    od
  ; if IsTextNode(A) → LeaveTextMode()
```

```
        □ else → LeaveMathMode()
        fi
    ]‖
```

The function *FormatString*() returns the format string for a given node. The functions *EnterTextMode*() and *EnterMathMode*() are used to indicate the preferred output mode, as used by the *Output*() function. As it is possible to combine text and mathematics freely, these functions use a stack of preferred modes to switch correctly. The functions *LeaveTextMode*() and *LeaveMathMode*() are used to operate correctly with the stack of modes. The function *Output*(*c*) generates the markup output for a character *c* and looks as follows:

```
‖[  if Macro(c , PreferredMode)≠Empty  →
          newmode , macro := PreferredMode , Macro(c,PreferredMode)
    □ Macro(c , TextMode)≠Empty  →
          newmode , macro := TextMode , Macro(c,TextMode)
    □ Macro(c , MathMode)≠Empty  →
          newmode , macro := MathMode , Macro(c,MathMode)
    □ else → newmode , macro := PreferredMode , DefaultMacro
    fi
  ; if CurrentMode≠newmode → SwitchToMode(newmode)
    □ else → skip
    fi
  ; Print(macro)
]‖
```

The function *Macro*() returns the markup macro for a given character and mode. The function *SwitchToMode*() switches to a new mode by adding the correct markup to the output and sets *CurrentMode* to the new mode. The function *Print*() adds a string to the output and keeps track of the correct termination of markup macros.

4.8 The window environment

The standard window environment is used to display the documents and stencils. As the user might be working on several documents and stencils at the same time, it should be possible have multiple documents and stencils open. There are several models which achieve this, as shown in figure 4.9.

- A *single document interface* is used to display a document. When the user switches to a different document, the content of the window is replaced by the new document. At all times, only one document is visible, which requires some

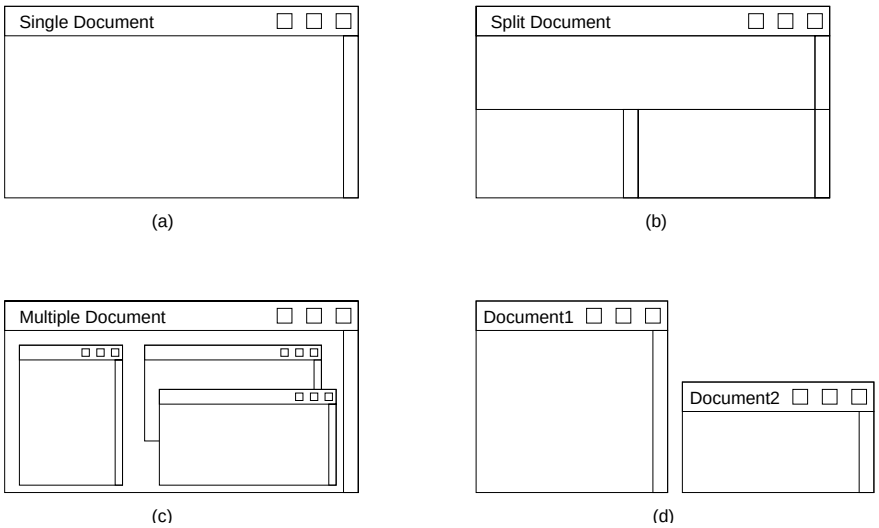


Figure 4.9: Different window interfaces:
(a) single document interface, (b) split document interface,
(c) multiple document interface, (d) multiple top-level interface

cognitive overhead when something has to be copied from one document to another. Since stencils are used as a source for copying templates, this model is not very useful.

- A *split document interface* is used to display several documents. The user can split the window horizontally or vertically to create several subwindows, one or each document or stencil. Since the user has to be able to manipulate these subwindows, some internal window management is needed to open, close, split or combine subwindows, which is likely to differ from the window management that is used for the normal windows. Furthermore, this model requires the user to split a rectangular window into a suitable configuration, which might be quite difficult.

- A *multiple document interface* (MDI) is used to display the several documents, where each document is displayed in its own window and these windows are kept together with one top-level[2] window. The user can manage the document windows similarly to normal windows and is able to position windows more flexibly. The top-level window provides the general functionality of the program, for example, to open new document windows or to perform some action. This model is regarded as better than the previous models, as it allows the user to

[2]A top-level window is a window which is manageable by the window manager. As a result, it might appear in task bars and icon managers.

handle multiple documents with the same principles as managing the window environment. The MDI is commonly used in the MS-Windows environment.

- A *multiple top-level interface* (MTI) is used to display the several documents, where each document is displayed in its own top-level window. The main difference with the MDI is that the top-level window to keep everything together is missing. Instead, the user is initially presented with one main window, which provides the functionality to open additional windows. Since all the windows are top-level windows, the user can use the general window management routines to manipulate windows. Although the model is similar to the MDI, it is more flexible with respect to window management, as there is no restriction to place everything within a single rectangular window.

We want to have a flexible system with multiple documents and stencils, so the choice would be either the MDI or the MTI. On the platform of choice, UNIX with the X Window system, the MDI is not common and developing one would be non-trivial, due to the variety of window managers with their specific behaviour. By using the MTI, the user is free to choose a familiar window manager, which might have special support for MTI applications, such as virtual desktops or icon grouping.

Decision 16
The multiple top-level interface is used to allow multiple documents and stencils to be open at the same time. The management of the top-level windows is left to the window manager, which can be selected by the user.

There are several ways to build a window application:

- With an integrated environment, the application can be constructed by copying window elements onto an empty window and the interface can be constructed very quickly in a WYSIWYG way. For the actual functionality of the application, the developer has to fill in the functions that are called when buttons or menu items are selected. Such integrated environments, like Devguide, are not very common for the X windows system and they are usually platform specific and restricted by licenses.

- With a toolkit[74, 15, 52, 26], the low-level details of the window elements are handled by the toolkit, but these elements have to be combined by using a programming language. A toolkit can shorten the development time when standard elements can be used. Furthermore, the application will have the same look-and-feel as other applications developed with the same toolkit.

- With a low-level library[50], everything has to be constructed from scratch, but the developer has full control over the final result. If non-standard window elements are needed, the low-level library is the only way to construct these. Usually, it is possible to access the low-level library when a toolkit is used, to enable the construction of additional window elements.

Since integrated environments were not available on the intended platform at the time the decision had to be made, the choice was to use either a toolkit or the low-level libraries. Although the use of a toolkit would have shortened the development time, the special features that are needed to allow mathematical formulae on window elements, were missing in every toolkit available at that time. Therefore, the toolkit that would have been used, would have to be extended with additional elements, which required the use of the low-level library.

Selecting a toolkit is a tricky problem. The selected toolkit will impose a particular look-and-feel upon the application and since different applications use different toolkits, it is unlikely that all applications will have the same look-and-feel. Furthermore, a toolkit is often related to a window manager to create a consistent interface, such as the Openlook and Motif toolkits and window managers[74, 37]. If such a toolkit is used in combination with a different window manager, the interface is not consistent anymore and strange behavior might be the result. For example, many Openlook applications don't provide an option to end it, as the window manager is supposed to provide that function. However, with some window managers, ending such an application becomes clumsy. Due to all the available toolkits and window managers, there is no consistent look-and-feel within the X window environment and a selection based on the look-and-feel is biased. Another selection criteria could be the license under which the toolkit can be used, as certain toolkits cannot be distributed freely. However, freely distributable toolkits are usually not supported by a company, which means that you might have to solve certain problems with the toolkit yourself.

Since the number of necessary window elements is rather limited and most of them have to be constructed from scratch, the decision was made not to use any toolkit at all. Some knowledge of the low-level library is already needed for these special window elements and constructing a concise personal toolkit would suffice for most purposes. The needed window elements include buttons, scrollbars, menus and canvasses, each of which is easy to construct with the low-level library. Furthermore, by constructing a personal toolkit, we have full control over the final layout and we can give it a certain look-and-feel. Since the functionality is more important than the appearance, this toolkit is kept rather simple. For portability reasons, it does not use any fancy 3D or colour effects, as it has to be usable on monochrome or grayscale monitors as well.

Decision 17
Instead of using an existing toolkit and extending it with new elements, a personal toolkit is constructed using the low-level libraries. Since the functionality is more important than the appearance of the interface, this toolkit will not contain features where they are not needed.

When there is some consensus on the use of toolkits and window managers, the knowledge of the low-level libraries can be used to extend some of these toolkits with new elements. Then, everything can be ported to the extended toolkit. An experimental port to Windows95[7] shows that the functionality is well separated from the interface and that such a port should not be too difficult.

4.8.1 The window elements

The private toolkit only supports a small collection of window elements, which are sufficient to construct an interface. It is always possible to construct additional elements or improve the existing ones, but as stated earlier, the functionality is more important than the appearance. Therefore, the toolkit is similar to the standard X Toolkit[52], which is used by several common or related X applications, such as xterm, xman, ghostview and xdvi.

The *button* element is used to either call a function or open a popup menu, depending on which mouse button is used. When a popup menu is available, the default function from that menu will be used if the menu is not to be displayed. The button contains a description of the functions that can be found on the menu, which might be of a mathematical nature. When the button is used, visual feedback is given to indicate whether the function will be called or not. If the popup menu has to appear, it will appear before the mouse button is released, which allows the user to select a function from the menu in a single action.

A *popup menu* consists of a title and a list of items, each describing either a function or a submenu. When the item is selected, the related function is called or the submenu is opened. By dragging the mouse pointer across a popup menu, submenus can be opened in a single action. As with buttons, the description of an item might consist of mathematical content. Since certain menus might be used very often, it is possible to pin-up a popup menu by selecting its title, which allows the user to select items from that menu more easily. This pin-up feature is also available in the Openlook toolkit.

The *scrollbar* is used to navigate through documents which are too large to fit on the screen at once. The scrollbar behaves similarly to the scrollbars from the X Toolkit. By clicking with the first mouse button, the line in the document at that position will move to the top of the window. By clicking with the second mouse button, the document will move to the relative position within the document, according to the position within the scrollbar. By clicking with the third mouse button, the line at the top of the window will move to the position of the mouse. By dragging with the second mouse button down, the document can be move around to visually search for something. Since LaTeX is used to divide a document into physical pages, it is not possible to scroll through a document page by page in the normal sense. However, it is possible to scroll screen by screen, by clicking with the first or third mouse button on the lowest part of the scrollbar.

For entering small parts of text, such as a file name or option, a *one-line text edit* element is constructed. For consistency, it is possible to enter mathematical symbols in those text elements as well.

To allow easy selection of documents, a *file selector* is created by combining the previous elements. The filename is either entered in the text edit element or can be selected from a list of files and directories. To be consistent with many command line shells, the text edit element supports filename completion, username expansion and environment variable substitution. To improve the browsing of files, the file selector

separates the directories and files into two lists and these lists are connected to the filename completion from the text edit element. Usually, the user is only interested in files with a certain extension, so it is possible to specify that extension. Once the correct filename is selected, the user can click on one of the buttons to perform some action on the file.

For displaying documents, an element is constructed which can be used as a *canvas*. The display algorithm for documents will draw the content on that canvas. Since a document can get very large, the canvas is used together with a horizontal and vertical scrollbar and with a number of buttons to perform document specific actions.

Mathematical notations often use symbols which do not appear on the average keyboard. To allow easy access to those symbols, a *symbol palette* is constructed which can display several pages of special symbols. By clicking on a symbol on the palette, it will be inserted at the position of the cursor. Since not everybody will need the same symbols, it is possible to adjust the symbol pages with a configuration file or interactively. For keyboard oriented users, it is possible to access the symbols by defining keyboard shortcuts for the often used symbols.

To access all the stencils and templates, a *template palette* is constructed, which will display all the templates from a stencil. By clicking on a template in the palette, that template will be inserted at the cursor position if that is possible. Since a template can contain several versions, it is possible to open a popup menu with the available versions by clicking with the third mouse button. To improve the accessibility of templates and their versions, these popups can be pinned up to allow direct access.

To edit the stencil or modify templates, the template palette also gives access to a *definition window* through a button. The definition window is used to display the internal details of the template and allows the user to modify it by means of buttons, menus, keyboard commands, symbol palettes and template palettes. Once the definition of the template is correct, it can be added to the stencil.

For *find-and-replace* actions, a special window is constructed, which consists of two canvasses, one for the find expression and one for the replace expression. On these canvasses, place holders are numbered to indicate how expressions should match and how such expressions should be replaced, where place holders with the same number match equal expressions. Since multiple find and replace actions might be needed, such actions can be stored in an internal list or in a file.

4.9 Box structures

Since mathematical notations are often two-dimensional, the display algorithm should be able to handle those. The display algorithm uses boxes to draw the mathematical text, in a manner similar to the use of boxes in the TEX system[36]. However, the display algorithm should be fast enough for interactive updates, even on less powerful machines, which restricts the complexity of box constructs. For example, TEX uses a very powerful and complex box language, but it requires so much time that users of TEX were accustomed to run it during coffee breaks or during the night. For an

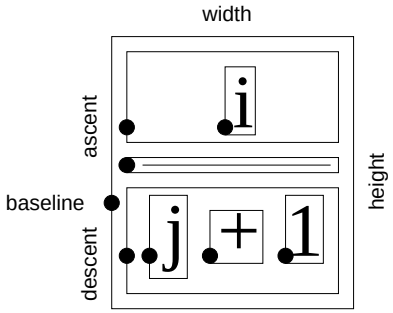


Figure 4.10: The box structure.

interactive editor, trained users can type plain text at rates of 300 keys per minute, which requires five screen updates every second. To ensure that the user doesn't get a headache after several minutes, these screen updates should be fast enough to remove any flickering. Therefore, the box language is kept simple, yet powerful enough to support most mathematical constructs and some textual constructs.

Decision 18
Boxes are used for improving the layout of mathematical content. Due to performance issues, the number of box constructs is kept small.

With the box language, the display algorithm can construct boxes by opening and closing the appropriate boxes. When a box is closed, the box layout algorithm will calculate the properties of the boxes, such as height, width, ascent and descent and place its internal boxes at the correct positions. Once the top-level box is closed, the complete collection of boxes is drawn on the canvas.

Figure 4.10 shows the box structure for the expression $\frac{i}{j+1}$, which is used to explain the properties of boxes. As usual, each box has a *height* and a *width* to indicate its size. However, for positioning text correctly, each box also has a *baseline*, which indicates how that box is aligned with other boxes horizontally. The position of the baseline is indicated by the dot on the left line of the box. The part above the baseline is called the *ascent* and the part below the baseline is called the *descent*. To be able to position boxes correctly, each box also contains the coordinates of that box, relative to the position of the enclosing box, where the left end of the base line is used as a reference point, indicated by a big dot. Since boxes can contain subboxes, the boxes are stored in a tree structure, where each box contains a list of subboxes.

To construct mathematical notations, different types of boxes are allowed, which is achieved by adding a *box-type* attribute. The following types are currently available:

- The *text* box is used for displaying text. Although it is possible to use one box for each character, the text box combines as many characters a possible to improve performance. To store the content of the box, a box is extended

with a *string*, its *length* and the *font* attributes. To handle *italic correction*[3], an attribute is added to store the correction needed for the last character.

- The *back* box is a box for inserting negative horizontal spacing, thereby moving leftwards. Normally, while boxes are constructed, a box is extended rightwards. However, in the tabbing environment, which behaves like the tabbing environment in LaTeX, it is possible to move leftwards, overwriting previous boxes. A back box is just an empty box of a certain width, where the right-side of the box is aligned with the right-side of the previous box. Due to the negative spacing, the width of a box might not be equal to the sum of the width of its sub-boxes. Therefore, an attribute is introduced to store the *maximal width* of a box.

- The *newline* box is used to signal the end of a line. When the end of a line is encountered, the rest of the line needs to be cleared. A newline box is regarded as an empty box of infinite length.

- The *stack base* box is used to place three subboxes, *top*, *gap* and *bottom*, above each other. If more boxes have to be stacked together, nested stack boxes can be used. To determine the baseline of the stack, the gap box is used, where the baseline of the stack is aligned with the baseline of the gap. In a variation of the stack base box, called the *stack center* box, the baseline of the gap box is positioned half the height of an 'x' above the baseline of the stack. Although the difference is small, it is sufficient to construct many mathematical notations, such a superscripts, subscripts, underlines, overlines and fractions. The example section contains some examples on how the stacks are used.

- The *bar* box is a box for adding a vertical bar, where the height of the bar depends on the boxes that appear on the same line. Usually, the bar is used in combination with the stack, to add a delimiter.

- The *stipple* box, *line* box and *space* box are horizontally stretching boxes for adding lines, stipple lines or spaces. The space box is used in combination with the stack box to align its subboxes. The line box is used for constructing fractions or underlines. The stipple box can be used for highlighting purposes.

With these boxes, many mathematical notations can be built, although some are still missing, such as arrays, roots and scalable delimiters. However, the missing notations are less common and can be simulated to some extent with the available box constructs. Scalable delimiters are particularly difficult to support directly, unless boxes for vector drawing are added.

For drawing the document, this collection of box constructs would be sufficient. However, for editing the document, a cursor is needed to indicate the insert position, origin tracking is needed to link mouse clicks to selected objects and inversion is needed to indicate selections. To support these additional features, two additional

[3]Italic correction is needed when italic text is ended, for example, "*some stuff*" versus "*some stuff*".

box types are added, the *cursor* box for adding cursors and the *node* box for keeping track of *objects*. To support inversion, each box is extended with an attribute that indicates which *colour combination* is used. These additional features do not influence the layout algorithm, otherwise the layout would depend on the cursor position or selected area. For origin tracking, the boxes are rebuilt, without actually drawing them. When the correct box is found, the object that is responsible for that box is extracted and returned to the origin tracking routine.

To reduce flickering and improve performance, the screen updates are based on a line-by-line algorithm. Therefore, all the boxes have to be closed at the end of a line, in order to calculate the final position of each box. After the line is drawn, the boxes have to be reopened again, in order to continue correctly with the next line. For normal text, this is not a problem, but when a linebreak appears within a stack box, the layout results are undefined. Luckily, the stack is mainly used for mathematical structures, where linebreaks are very rare. In the event that someone uses the stack for plain text, for example to simulate underlining, the text is still readable, but not complete, for example, part of the underlining is missing.

When the boxes are drawn on the canvas, the original content of the canvas has to be cleared. The easiest way to achieve this is by clearing everything at once and then drawing the boxes. However, that could introduce flickering, as the time between clearing and drawing is larger. Therefore, the canvas is cleared just before the boxes are drawn. To achieve this, each box is either totally covered by its subboxes or it contains no subboxes at all. Then, each box without subboxes has to clear the canvas before drawing, while each box with subboxes can rely on the subboxes to clear everything correctly. An additional advantage of this method is the ability to use "exclusive or" operations on boxes when selections are made, which requires that the operation is performed an odd number of times.

To further improve the drawing performance, each box is extend with a *style* attribute to indicate whether that box has to be drawn or not. This allows the screen update routine to indicate which boxes are used for calculations only, as in the case where only the right half of a line has to be drawn in order to update the screen. Such partial redraws require some additional calculations, but it reduces the number of drawing requests for the X server, which possibly require network communications and calls to complex graphics routines.

By combining all the attributes, the final box structure looks as follows:

```
BoxType = {
      special : integer;           % type of box
      width, ascent,
         descent : integer;        % dimension
      x, y : integer;              % position
      itemnr, itemtot : integer;   % string-pool item
      fattrib : integer;           % font attributes
      lright : integer;            % italic correction
      maxwidth : integer;          % negative spacing
      color : integer:             % selection scheme
```

```
        funcarg : pointer;          % origin tracking
        style : integer;            % drawing style
        fbox : BoxType;             % first sub-box
        lbox : BoxType;             % last sub-box
        nbox : BoxType;             % next box
}
```

4.10 Control characters

Since the layout of notations is defined in the screen format of versions from templates, these formats have to be used to construct the boxes correctly. It is possible to achieve this by defining a small markup language to describe the boxes. However, such a language would require that the maintainer of templates has to learn yet another markup language. Furthermore, processing this markup language would require additional parsing time while the document is displayed, which slows down the screen updates. To reduce the parsing time, each markup sequence can be replaced by a control character, which removes the lexical analyser state of the parsing routine. Since there is a private area within the Unicode encoding with sufficient positions, the use of control characters is not a problem.

To relieve the maintainer of templates of the burden to learn a new markup language, the control characters are made available through popup menus whereby the system will ensure that control characters are used correctly. For example, if two control characters are used as opening and closing brackets, then the system will assure that those characters are used as a pair and are correctly nested in combination with other bracket-like control characters.

> **Decision 19**
> Control characters are used in the screen format of a template to indicate how boxes have to be constructed. Since the place holders have to be inserted in the screen format as well, they are also treated as control characters.

The tree traversal algorithm will use the screen format to traverse the tree and process the characters one by one. The procedure that processes these characters interprets the control characters and builds the boxes where needed. Normal characters need no special treatment and are just added to a text box.

The following control character are currently used by the system. Although certain control characters are equal in appearance, such as the closing brackets, the system might use different characters for them. Since the user can only edit these characters in pairs, the internal difference is not important to the user.

E, O, I, V, T The place holders, which indicate where a subtree has to be inserted. Since templates can contain multiple place holders of the same type, each place holder can have an index number associated with it to determine the unique subtree.

[**Name:,**] To give a hint to the user of a template, a place holder can be given a name to indicate its purpose. These control characters are used to enclose the name of the place holder, just before the place holder itself is used.

[**text**], [**math**], [**disp**] The LaTeX mode indicators, which are used to steer the LaTeX output algorithm when the output format is not standard.

[**attribute=value:,**] The font attribute control characters, which change the attributes of the current font. A configuration file determines which attributes there are, which values these attributes can take and how attributes and their values are ordered. The ordering on the attribute values is used to determine which attribute combination should be used as an alternative when a combination is not supported by fonts. The default configuration file defines the attributes, values and ordering as supported by LaTeX. That is, the attributes are family (roman, sans serif and type writer), series (medium and bold), shape (upright, italic, slanted and small caps) and size (10 values). This corresponds to 240 attribute combinations.

[**SizeR***n*,] The control characters for relative size change, which are similar to the font attribute control character. The relative size changes are used when the size of text has to change relative to the actual size. The sizes are available as discrete values as defined by the configuration file for font attributes and are sorted. The value n determines how much the size increases or decreases.

[**Tabbing:,**], [**Display:,**], [**set**], [**tab**], [**back**], [**plus**], [**minus**], [**push**], [**pop**] The tabbing control characters, which are used for indentation and other specific layout features. The tabbing controls are based on the commands used by LaTeX, except that certain commands behave slightly different in order to work around the problems found in the LaTeX versions. The main differences are that the [**tab**] command does not fail due to undefined tab positions and that the movement of the left margin works correctly in combination with nested tab positions.

A short explanation of these commands might be needed for those who are not familiar with the tabbing environment of LaTeX. The [**Tabbing:** and [**Display:** commands open the tabbing environment, where the later one also moves the left margin forward by a standard number of tabs. The [**set**] command defines a tab position at the current location. The [**tab**] command jumps to the next tab position or moves forward over a specific distance. Similar to the behaviour in LaTeX, the tab position might be left of the current position, thereby moving backward. The [**back**] command moves to the previous tab position and can be used at the beginning of a line only. The [**plus**] command moves the left margin one tab position forward. The [**minus**] command moves the left margin one tab position backwards. The [**push**] and [**pop**] commands are used to enter a temporary tabbing environment, to change tab positions locally.

[**StackB:, :, :,**], [**StackC:, :, :,**], [**Line**], [**Fill**], [**Dots**], [**Bar**] The control characters for placing boxes above each other and to align them correctly. The [**StackB: : : **] commands are always used in combination and the three fields that they

construct are combined into one stack, where the middle field is used for vertical alignment. Horizontally, the fields are centred, unless the stretching commands ([**Line**], [**Fill**] and [**Dots**]) are used to fill the space that is left over. If multiple filling commands are used within the same field, the space is equally divided over them. The [**StackC: : :**] commands are similar to the [**StackB: : :**] commands, except that the middle filed is aligned above the baseline of the surrounding box. The [**Bar**] command will insert a vertical line, with a length equal to the height of the field in which it appears. It is also possible to use the [**Bar**] command outside a stack environment, in which case the height of the line is used.

[**Space**] The control character for adding micro spacing to operators. The system uses micro spacing around operators to indicate the structure. For non-operators, where the location for adding such micro spacing is not clear, this structure-dependent spacing is added by hand with this character.

There are still sufficient positions available to add more control characters. Possible extensions might be control characters for colour changes, language tagging, country tagging, table construction or conditional output.

4.10.1 Some examples

These commands might not make much sense at first, but they are easy to use once you get the hang of it. In general, if you need different margins, you use the tabbing environment with its commands, if you need things above each other, you use the stack environment with its commands, and if you need both, you combine the environments. If you need to highlight some text, you use the font attribute commands.

The commands are entered in the definition window, which is used to construct templates. By means of popup menus and keyboard short-cuts, the screen format is created and special features are added. Since the control commands affect the layout of the screen format, the user can switch between an interpreted mode and uninterpreted mode to improve editing capabilities. In the interpreted mode, the screen format is displayed as it will appear within a document and the commands themselves are not visible. In the uninterpreted mode, the screen format will display the commands in a verbatim manner to allow easier manipulation of the screen format.

In the following examples, the screen formats are displayed in the uninterpreted way, where the control commands are displayed in the small, bold style. For completeness, the LaTeX output for each example is given in the output format, where the small bold style is used to indicate automatically generated LaTeX output. This layout convention is also used by MathJpad itself.

Font change

Font changes are very common, especially in technical reading, where it is used to highlight keywords, examples and keyboard input. Creating a template which changes the appearance of text is straightforward. The "Font" menu from the define window contains the commands to change font attributes and the commands are inserted by selecting the appropriate menu item. The template in the following example is used to make the content **bold**. The screen version contains the command to change the font attributes to bold and a text place holder to contain the text.

Screen: [**Series=Bold:**T1]
Output: **{\bfseries** T1**}**
Name: Bold

Quote

When text is quoted from someone else, it it usually displayed in a separate paragraph with an indented left and right margin. It is possible to simulate this partially with the tabbing environment, where the left margin can be moved rightward without problem, but the right margin cannot be moved. Since the templates are not required to be perfect, such small differences are not a problem.

The quotation uses the tabbing environment to be able to move the left margin. Within this environment, a tab is used to move forward and then the margin is moved rightwards to the current position. After the quoted text, which is entered in a place holder, the margin is moved back and the tabbing environment is ended. Since the quoted text has to appear as a separate paragraph, line breaks are added at the beginning and end of the template.

Screen:
[**Tabbing:**[**tab**][**plus**]T1[**minus**]
]
Output:
\begin{quote}T1
\end{quote}

Name: Quote

In this template, the output format is modified to replace the automatically generated tabbing environment, which is used to simulate visual behaviour. Like the quote environment from LaTeX, the quote template can be used recursively.

Itemise

A common way to list a number of items is with a bullet list, where the left margin is moved rightwards and each item is preceded by a big dot. Again, simulation of

this environment is not to very difficult with the tabbing environment. However, two versions are needed to make it work: one for the environment which defines the margin and one for adding the items and moving the margin.

The first version just opens the tabbing environment, sets a tab at the appropriate position and contains a place holder for the items that will be inserted. The second version adds the big dot, moves to the next tab position, adjusts the left margin, enters a text place holder for the content and restores the left margin.

Screen:
[**Tabbing:** [**set**]
T1]
Output:
\begin{itemize}
T1
\end{itemize}

Name: Environment

Screen: •[**tab**][**plus**]**T**1[**minus**]

Output: \item **T**1

Name: Item

It is possible to simulate the itemise environment differently. The version for the environment could adjust the margin, while the version for the item would temporarily move it back to add the big dot. Another option is to remove the text place holder from the item version, as that text could also be inserted in the environment itself. One advantage of the given approach is that each item with its description is easily selected by selecting the big dot, which is part of the template.

Proof step

Since the Eindhoven style of proof presentation is often used by the initial users of Mathpad, support for that notation should be available. This proof style uses indentation to improve readability and uses hints to explanation each proof step. The following example shows a trivial proof of two steps:

$$
\begin{array}{ll}
 & A \wedge (A \vee B) \\
= & \quad \{ \quad \wedge \text{ distributes over } \vee \quad \} \\
 & (A \wedge A) \vee (A \wedge B) \\
= & \quad \{ \quad \wedge \text{ idempotent} \quad \} \\
 & A \vee (A \wedge B)
\end{array}
$$

Each step in the proof notation is actually a binary operator which relates two expressions and contains a hint with a short explanation. Therefore, the template that is used for a proof step should be a binary operator.

In the proof notation, there are four tab positions, three of which are also margin positions. The first tab position is used for aligning the binary relators of each step, the second tab position is used for aligning the expressions, the third tab position is used for aligning the opening brackets ({) of each hint and the fourth tab position is used for aligning the hints. Since a hint might consist of multiple lines, the left margin should be moved to the fourth tab position in order to align those lines correctly. Since the expressions require that the margin is located at the second tab position, the template for the proof step can assume that it is possible to move the margin one tab position leftwards. An additional assumption is that the environment where this template will be used has defined the tab positions that are needed.

Screen: **[minus]**
=[tab][tab]{[tab][plus][plus][plus]T1[minus][minus] }

Output: \-
=\>\>\{\>\+\+\+**T**1\-\-∼∼∼\}

Name: Proof step

Accent

Most accents are positioned above or below a character and many accented characters are available by default. However, if a certain combination is missing, a template can be used to place one character above another. For example, characters with a macron accent (¯) are often missing, although the macron accent itself is available. With the stacked boxes, it is possible to combine the macron accent with any character that is available, simply by placing the macron accent in the top box and the character or place holder in the gap box.

Screen: **[StackB:¯:T1:]**
Output: \={**T**1}
Name: Macron Accent

Super- and subscript

Superscripts and subscripts are often used in mathematics and with the stacked boxes, Mathʃpad is able to support them too. A superscript is an expression that is positioned above the baseline and in a smaller font. A subscript is similar, except that it is positioned below the baseline. To align the super- and subscripts correctly, the **[StackC:::]** version of the stacked boxes are used, which gives the best results. Since super- and subscripts are also used recursively, the size of the expression depends on the level of recursion and is therefore relative to the actual size.

Screen: [**StackC:**[**SizeR-1:**E1]**::**]
Output: ^{E1}
Name: Superscript

Screen: [**StackC:::**[**SizeR-1:**E1]]
Output: _{E1}
Name: Subscript

Summation

The summation-like notation is also commonly used in mathematics, where the scope of an operation is given by placing the start condition below and the end condition above an enlarged operator symbol. The expression on which the operator acts is given after the operator. For example, the expression

$$\sum_{i=1}^{n} i^2$$

symbolises the formula "the summation from i is 1 to n of square i". It is not very difficult to support such a notation with Mathƒpad. With stacked boxes, the expressions are placed above each other and with a relative size change, the sigma sign can be enlarged and the range expressions reduced.

Screen: [**StackB:**[**SizeR-1:**E3]**:**[**SizeR+1:**Σ]**:**[**SizeR-1:**E2]] E1
Output: \sum_{E2}^{E3} E1
Name: Summation

Although it is possible to use this summation template recursively, it is not very common in mathematics and proper support from LaTeX might be missing.

Fraction

Fractions are also commonly used and support within Mathƒpad is again achieved with the stacked boxes, where the middle box contains a line that extends according to the size of the nominator and denominator. The correct alignment of the fraction is achieved by using the [**StackC:::**] version, which aligns the line of the fraction roughly with the position of the minus sign. Again, relative size changes are used to reduce the size of sub-expressions and recursive usage will result in smaller and smaller expressions.

Screen: [**StackC:**[**SizeR-1:**E1]**:**[**Line**]**:**[**SizeR-1:**E2]]
Output: \frac{E1}{E2}
Name: Fraction

Framed boxes

Although framed boxes are not very common, they are useful for highlighting purposes. Since framed boxes are not directly supported, they can be simulated with a combination of stacked boxes and extendible lines.

Screen: [**StackB:**[**Line**]:[**Bar**]**T**1[**Bar**]:[**Line**]]
Output: \fbox{**T**1}
Name: Framed Box

Scalable delimiters

Scalable delimiters are often use in combination with two-dimensional arrays or to describe several options, as in the following expression.

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x \leq 0 \end{cases}$$

Although these scalable delimiters are difficult to support, it is possible to simulate them with stacked boxes, although they will not scale automatically. By placing the correct symbols above each other, a large delimiter is constructed. The special symbols to do this are available in the Adobe symbol font and have been specifically constructed for this purpose.

The large delimiter will not scale automatically, but it is possible to construct multiple versions, such that the user of the template can switch to the correct version when that is important.

Screen: [**StackB:**⎧:⎨:⎩]
Output: \left\{
Name: Large {

Tables and arrays

Tables and arrays are not easy to capture in a tree structure, which requires the table to be broken into either one row of columns or one column of rows. In each case, manipulations on either rows or columns are easy, while manipulations on the other orientation are difficult. To correctly support tables and arrays, the tree structure has to be abandoned and an n-dimensional array structure is required. Combining the tree structure with the array structure is possible, but it would make the calculations for screen layout and updates very complex due to performance requirements. Since tables are used less often by the initial target users, direct support is not needed, as long as it is possible to simulate them reasonably.

In markup languages such as LaTeX, Troff and HTML[39, 22, 58], a table is described in its reading order, that is, on row-by-row basis, where each row describes the cells for the different columns. Since the templates are used to generate such markup

output, the template for constructing tables will describe one row at a time. To align the cells in each row, the tabbing environment is used, where the tab positions are defined in a template that adds the table environment. The use of the tabbing environment only allows columns to be left-aligned on screen, although the markup output could contain instructions to align them otherwise.

By choosing the row-based approach, operations on columns become more difficult. However, if a column-based approach with stacked boxes were to be used, operations on rows would be difficult and, furthermore, the alignment of rows would be impossible and the generation of markup output much more complex.

Screen: **[Tabbing:T**2**[set]T**3**[set]T**4**[set]**
T1**]**
Output: \begin{tabular}{**T**2**T**3**T**4**}**
T1
\end{tabular}
Name: Table environment, 3 columns

Screen: **T**1**[tab]T**2**[tab]T**3

Output: **T**1 & **T**2 & **T**3 \\

Name: Table row, 3 columns

An explanation might be in order. In the first version, the tab positions are set by filling the place holders **T**2, **T**3 and **T**4 with the appropriate alignment commands, where spaces are added to widen the column. The rows are inserted in place holder **T**1. The second version is used for adding rows, where each place holder contains one element.

As these two versions show, different templates are needed for tables with a different number of columns, which makes adding or removing a column difficult, although the find-and-replace utility can be used. Furthermore, to support tables with frames, versions have to be defined to add horizontal and vertical lines. With these templates, simple tables can be simulated on screen adequately, but tables with special requirements such as cells that span across multiple rows or columns are not possible in general.

4.11 Keyboard handling

As the keyboard will be the most important form of interaction with the user, the keyboard handler should be powerful enough to support the common methods of keyboard interaction:

- key combinations, such as ‘ctrl-c’, ‘shift-F7’ or ‘ctrl-alt-delete’.
- prefix keys, such as ‘ctrl-x ctrl-c’, ‘alt f x’ or ‘compose e ^’
- keyboard modes, such as an insert mode and a command mode
- nested keyboard modes, such as a Pascal mode within an Emacs mode.
- temporary keyboard modes, such as the incremental search mode from Emacs, where the selected mode is used as long as the pressed keys are defined in that mode.

Furthermore, the keyboard interaction has to be configurable, as different users have different requirements.

In order to support these different methods, the keys and the functions that are connected to them are stored in a keymap table. A key combination is regarded as a single key, where only the modifier keys, such as control, shift and alt, can be used in combination with any other key. A key combination such as ‘stop-a’ or ‘esc-a’ is therefore not possible, but such key combinations are not very common anyway.

To support prefix keys, a key can be connected to a keymap table instead of a function. When the prefix key is pressed, the keymap table that is connected to the key will be used to handle the next key. If the next key is not defined in combination with the prefix key, then the key will be ignored and the original keymap table will be restored.

To support keyboard modes, each mode can use its own keymap table, where the keys are connected to functions. In order to switch between the keyboard modes, a special function is supplied to switch to a different keymap table, where the name of the keymap table is used as unique identifier.

To support nested keyboard modes, a keyboard handler uses a stack of keymap tables to handle the keys. When a key is pressed, the stack of keymap tables is used to find the function connected to that key in the top-most table. If the key appears to be a prefix key, all the tables on the stack will either enter the prefix state or be disabled temporarily.

To support temporary keyboard modes, a keymap table can be placed on the stack of the keyboard handler, such that it will be removed when a key is pressed which isn’t defined in that keymap table. As a temporary mode might be used to construct some argument to a function, as in the universal argument mode of Emacs, it is possible to specify two functions which will be executed when the temporary mode is removed: one function is called just before the mode is removed, which can be used to set default values; the other function is called after the undefined key is handled, which can be used to clean up state information used by the temporary mode.

Certain modifier keys, such as the shift modifier, are often used by the operating system or the window system to adjust the key accordingly, as in ‘shift-a’ versus ‘A’. However, when a keymap table is used to find functions connected to keys, it is useful to ignore such modifiers, as the difference between ‘ctrl-a’, ‘ctrl-A’ and ‘ctrl-shift-A’

is often not clear to the user. Therefore, when a keymap table is defined, it is possible to specify which modifiers can be ignored for certain key combinations.

Section 5.2.4 contains some additional information on how the keyboard interface is defined.

4.12 Unicode support

Unicode support is not yet very common on UNIX systems or it is unportable. The C language defines a wide character datatype together with a collection of functions to handle such characters or the strings constructed with them. However, these wide characters are intended for Asian languages[32], so the library functions might not work correctly when they are used to store Unicode characters. Furthermore, the wide characters are not very portable. So, in order to get basic Unicode support, a small library is constructed to handle Unicode characters, strings and fonts.

The Unicode characters are stored internally in the UTF16 encoding, which uses 16 bits for each characters and uses surrogates to access the 16 other planes reserved by the ISO-10646 encoding. Although the full 32 bit encoding could be used to work around the problems related to surrogates, such an encoding would double the memory requirements for strings, while the surrogates are not used yet. As for now, Unicode characters are stored in a 16-bit integer type, while the surrogates are unsupported. When surrogate support is required, it will not be very difficult to use a 32-bit integer type.

The character properties, as defined in the Unicode data table, are accessible through functions and tables, similar to the character properties defined in the standard C header file `ctype.h`. In order to handle the large collection of symbols and properties efficiently, two-layered tables are used to store the spare or monotonic tables.

For the conversion of data encoded using other standards, mapping tables are constructed, which are stored on disk, such that the library is kept small and easy extendible. As most existing encodings are either 8-bit or 16-bit, it is easy to store these mapping tables as two-layered tables. Certain encodings for the Far-East can be regarded as a 12 bit encoding, where some characters are encoded by 8 bits, while others are encoded by 16 bits. As these encodings do not use any state information, it is possible to store its mapping table in a two-layered table, where the table is accessed as follows: if the 8-bit character is defined in the table, then that value is used, otherwise the next 8 bits are used to construct a 16-bit character, which will be mapped according to the table.

The string manipulation functions as found in the C headers `string.h` and `widec.h` cannot be used on the Unicode strings due to the incompatible types. Therefore, the functions from `string.h` are re-implemented to work on Unicode strings, which isn't that difficult.

In order to draw a Unicode string on screen, the library has to provide some facilities to access all the Unicode symbols. Although the X window system can work directly

with fonts in the Unicode encoding, the main problem is that there are only a few of such fonts, which clearly isn't sufficient for a LaTeX user with 240 possible font attribute combinations. Therefore the library supports the definition of virtual Unicode fonts, where existing fonts with different encodings are combined to construct a partial Unicode font. These virtual fonts allow the use of all existing fonts on the system, as long as a mapping table is available for the encoding that they use. The virtual fonts are constructed with a configuration file, which contains the instructions on how to combine fonts and the attributes of the virtual fonts. If multiple virtual fonts are defined, the font attributes will be used to find character glyphs when they are missing in a certain virtual font. As the X window system gives an application access to thousands of fonts, the library will only load the existing fonts on demand, that is, when it is needed to actually draw something.

Although the library is kept as portable as possible, the part that constructs the virtual fonts is destined to be less portable, as it contains system specific calls to access the fonts. However, these calls are well isolated and easy to replace.

Chapter 5

Integrating tools

A technical document often contains information from different sources, such as plain text, mathematical computations, program listings and illustrations. Each of these sources are related to special tools to manipulate them, such as spelling checkers, algebra systems, compilers and image manipulation software. For the author of a document, switching between all these tools might be cumbersome and introduce errors due to version management. Support for accessing these additional tools from within the document editor is needed in order to improve the usability.

5.1 Communication between tools

There are several methods to communicate between different tools. Since the target operating system is UNIX, the described methods are most common for that environment, although methods used in other operating systems are also mentioned.

5.1.1 Pipes

The most common way to combine tools in UNIX is by using pipes, where several tools process the information independently in sequential order. It works on the principle that each tool processes its input and generates output using standard channels (stdin/stdout), where a special program called a shell is able to connect the output channel from one tool with the input channel from the next tool, as shown in figure 5.1. Under a multiple-processing environment like UNIX, pipes are very powerful, as the information is processed by different tools at the same time whenever that is possible, whereby results are available as soon as possible. Under a single-process environment like DOS, pipes are rather clumsy, as each tool in the pipe is executed in sequence and the information that travels through the pipe is buffered between processes.

The UNIX environment contains many small tools for processing textual informa-

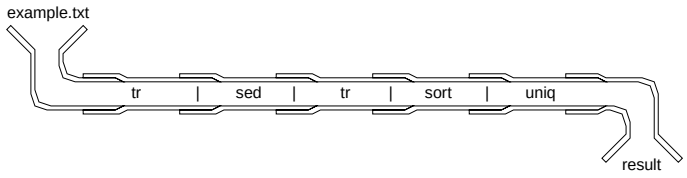


Figure 5.1: A pipe of five tools.

tion, such as `grep` (searching), `sed` (string manipulation), `awk` (table processing), `tr` (character transformation) and `sort` (ordering). By combining these small tools, a powerful and flexible solution is constructed for many text processing problems. For example, to determine how often each character is used as the first character of a word in the file `example.txt`, the following pipe can be used, also shown in figure 5.1:

```
tr -cs '[:alpha:]' '[\n*]' < example.txt     |
  sed 's#^\(.\).*#\1#g'                      |
  tr  '[:upper:]' '[:lower:]'                |
  sort                                       |
  uniq -c
```

The first `tr` command divides the words over lines, the `sed` command removes everything except the first character on the line, the second `tr` command converts everything to lowercase, the `sort` command orders everything and finally, the `uniq` command counts how often each line occurs in succession. The action of "piping" these together is denoted by the "`|`" symbol.

Although pipes are mainly used for processing text, it is also possible to process binary information, such as sound or image data. A common use of binary pipes is in combination with compression programs, as in

```
tar cf - /usr/demo | gzip > demo.tar.gz
```

where the intermediate archive might be too large to fit on the disk directly. The image manipulation package called `netpbm` allows images to be manipulated with a pipe, where each tool performs an operation on the image, such as conversion, rotation, scaling or colour reduction and where all non-conversion tools operate on a common image format. For example, to scale down a TIFF image to 25%, rotate it 90^0 clockwise, reduce the number of colours to 64 and save it as a GIF image, you can use the command

```
tifftopnm photo.tif | pnmscale 0.25 |
  pnmrotate -90 | ppmquant 64 | ppmtogif > photo.gif
```

Since the netpbm tools require no interaction with the user, they are especially useful for converting large amounts of image data automatically. Again, many small

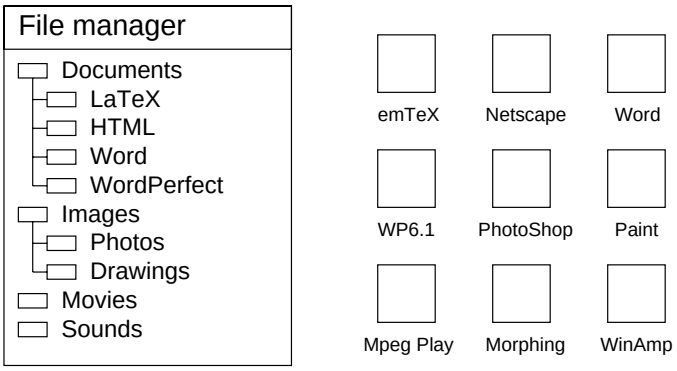


Figure 5.2: A file manager with helper applications

tools are combined into a powerful and flexible solution for many image processing operations.

The use of pipes requires some overhead from the operating system, as the multiple processes introduce context switches between those processes. Furthermore, different tools require different options and script languages, which requires cognitive overhead from the user. As a result, there are tools that combine the functionality of several tools into one. The tool `perl` is a combination of several, if not all, text processing tools, together with a programming language and a shell.

5.1.2 Helper applications

A file manager is an application which tries to give the impression that it supports any file format that is available on the system. However, it would be impossible to construct a tool which supports every file format natively, as the resulting application would work like a Swiss army knife: you can use it for everything, but if you want to do it right, you need a specialised tool. Therefore, a file manager uses a database with helper applications, which connects each file format to an application that can handle that format, as shown in figure 5.2. When the user wants to work on a file, the database is used to determine which application has to be started.

Due to the large variety of file formats and tools, the user might be presented with several different interfaces, depending on which tools are available and how the database is constructed. For example, the database might connect different image formats to different applications, while the user would expect that all these image formats are handled by the same application. Therefore, such a database should be easy to adjust by the user.

The database has to determine the correct file format for each file in order to start the right application. A common method uses the file extension to determine the file format, but that method might fail, as different file formats can use the same

extension. Another method uses the content of the file to determine the file format, as most file formats contain a magic sequence to detect incorrect files. In order to check the magic sequence of a file, part of the file has to be accessed, which influences the performance. However, the method might also fail, as a given file might match multiple magic sequences. It is possible to combine both methods, where the content is used to check for magic sequences and the extension is used as an additional check. Due to performance issues, the method based on extensions is most commonly used.

Independently of the chosen method, the user has to be able to specify how files with multiple formats should be handled. An example of such a file is a self-extracting archive, which is simultaneously an application and an archive. As an application, the file is platform specific and is useless on the wrong platform. As an archive, the file is platform independent and can be used on any platform. However, the file only has one extension, often the one of the application, which hides the fact that it is usable as an archive. With the magic sequence method, this hidden information can be revealed and used.

A WWW browser can be regarded as a file manager where the only operation on files is to view them. Therefore, the browser uses a similar database to start the correct application when a file has to be viewed, except that this database is based on the mime-types which are part of the transfer protocol. The http server determines the file format of the file that is transferred and sends its format as mime-type at the beginning of the transmission. The browser uses that mime-type to handle the transferred file correctly. For this to work correctly, the server has to determine the right mime-type and the browser has to be able to handle that mime-type. Since the mime-types are usually determined by checking the extension[1], there is room for errors on the server side, due to incorrect and unlisted extensions. And since the collection of mime-types is expanding due to new file formats, the browser might not be able to handle certain mime-types, although the browser might suggest installing additional applications in order to handle them correctly.

5.1.3 Client – server

In the client-server model, a client application connects to a server and sends a number of requests. The server handles the requests by performing an action on the data that it maintains and responds to the client with an answer. Examples of the client-server model are:

- The X window system, where the X server manages the display of a computer and an X application can connect to manipulate the display, for example by opening a window or writing some text. Figure 5.3 shows the architecture of the X window system.
- The FTP and WWW sites, where a server manages a collection of files and a browser retrieves those files, for example when the user clicks on a hypertext

[1]The magic sequence method could be used, although it could cause performance problems. As the file is already accessed, the performance issue should be minor compared to network issues.

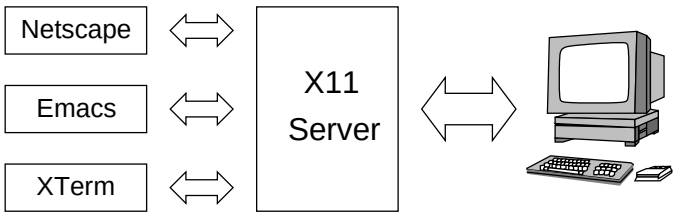


Figure 5.3: The client–server model of the X window system.

link.

- The NFS service, where an operating system manages a large, central file system and other operating systems can access that file system through its manager.
- A flight booking system, where a central server manages seats on airplanes and travel agents book a flight by reserving seats.

In a client-server model, the interaction between the client and the server is well defined by means of a protocol. If a client is compliant to the protocol, that client can connect to any server which is also compliant. As a result, compliant clients and servers are interchangeable.

5.1.4 Dedicated link

When two applications are highly related, a dedicated link between these applications can be constructed by adding knowledge to each application about the other application. Often, dedicated links are constructed between a general purpose application and a special purpose application, such as a link between a text editor like Emacs[67] and a proof engine like PVS[65], as shown in figure 5.4. For a normal user, the dedicated link between the two applications is so tight, that the distinction between the different tools is difficult to make.

In order to build a dedicated link, the source of the applications has to be available or the applications have to support a method for extending the system with new features. A successful method to achieve this is the use of an internal interpreter, such as the lisp interpreter in the Emacs editor. By constructing interpreted files, the application is extended without the need to rebuild it. When the interpreted language is powerful enough to start other applications and communicate with them, the creation of links to other applications is possible. For the Emacs editor, dozens of links are available, each consisting of some lisp files to handle the communication and an application to communicate with.

Another common method uses a library which handles the connection with the related application. By using that library, any application can create a connection and communicate. For example, a mathematical application can use the MathLink

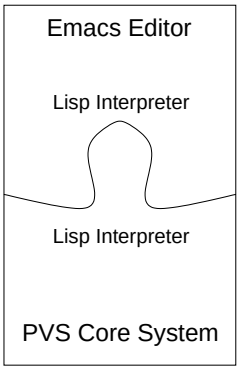


Figure 5.4: The dedicated link between Emacs and PVS

library[64] to handle connections with the Mathematica core system, thereby allowing complex computations to be performed by Mathematica[83]. Or, an application can use the WWW library to handle Internet related connections, thereby making the application Internet-aware and allowing access to files on remote systems.

5.1.5 Plug-In libraries

When the functional interface to the other application is not complex, plug-in libraries can be used to perform these tasks. In a plug-in library, a fixed set of commands is provided, which can be called by the host application. When the host application needs to perform a task which is supported by a plug-in, the plug-in is loaded and the related commands are called. A plug-in library works like an interpreted file, except that a plug-in consists of native machine code, which improves the performance, but makes it system dependent.

Netscape[43] is an example of an application that uses plug-ins, which is shown in figure 5.5. In order to handle objects which are not supported natively by Netscape, it allows the user to provide plug-ins for such objects, which will display the object as if it was supported natively. The plug-in is provided as a native dynamic library and contains functions to initialise and finalise the plug-in and to handle operations on the object. Since the plug-in uses native code, it is possible to transform an existing application into a plug-in and let it take advantage of WWW related technologies.

The Gnu Image Manipulation Program (GIMP)[24] and Adobe PhotoShop[1] use plug-ins to perform all kinds of operations on images. Many of the operations on images are similar to the operations available in the netpbm tool collection, but the performance is better due to shared memory and less conversion steps.

The Generalised Display Processor (GDP)[56] uses a script language called LOOKS, which allows run-time library linking, thereby combining flexibility, extendibility and high performance. The GDP is used as a basis for several research projects on 3D

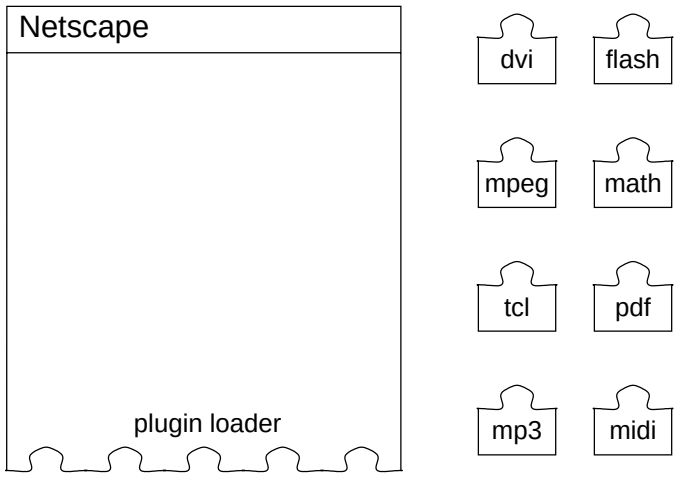


Figure 5.5: Netscape with plug-ins

modelling and animations, where extensions can be constructed independently.

Java applets[49] can also be seen as plug-ins, although they are not compiled to native machine code. Instead, an applet consists of bytecode, which is interpreted by a virtual machine running on a host operating system. Due to the virtual machine, an applet runs on any operating system for which a virtual machine is available, although the performance might be a problem due to the interpretation. For applications with high requirements on computational power, such as virtual reality, an applet can use a system specific library by using so called native methods. With these libraries, the system independent nature of Java is lost, but the performance is much better.

5.1.6 Software bus

A software bus is used when related applications need to work together without knowing in advance which other applications are available. The software bus forwards requests from one application to another, where the applications themselves know nothing about the other applications on the bus. The communication is based on a protocol, similar to the client-server model. However, there is no distinction between a client and a server, as an application might act as both at the same time.

Some existing software buses are:

- The ToolTalk service[75], which receives (unaddressed) messages from applications and delivers them to the appropriate applications. The ToolTalk service is designed for combining facilities from independently developed applications, which are otherwise difficult to achieve.
- CAS/PI[30], a distributed architecture for computer algebra systems, that en-

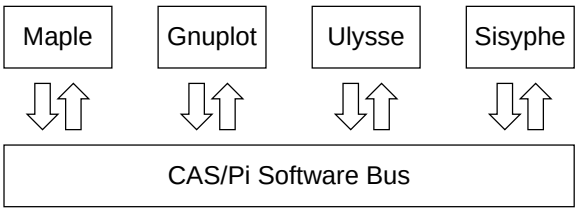


Figure 5.6: The software bus in CAS/PI.

ables communication between different algebra systems by use of a software bus and data manager. Figure 5.6 shows the architecture of CAS/PI.

- The ToolBus[4], a software architecture intended for building distributed applications. Components of the application communicate with each other over the bus and each component is specified in terms of process algebra. The ToolBus is the basis for the revised ASF+SDF Meta-environment.

5.1.7 Object sharing

While the previous methods are based on communication between applications, the object sharing method is based on communication between single objects within those applications. Within an object-oriented framework, a network-transparent architecture handles this communication between objects, such that the location of objects is irrelevant. As a result, an application can work with objects as if they are available locally, while they might in fact be stored on a remote system and managed by a different application.

The most important object sharing technologies are:

- CORBA[48], the Common Object Request Broker Architecture, allows applications to communicate with one another no matter where they are located or who has designed them. CORBA hides anything that is not related to the object itself, such as the programming language or the operating system, from the application which uses the object.

- OLE[46], Object Linking and Embedding, is a compound document architecture based on COM, the Common Object Model of Microsoft. OLE is used to combine objects such as text, spreadsheets and images into one document (see figure 5.7), where different applications are used for manipulating each type of object. Although OLE allows object sharing, it is not network transparent. Furthermore, OLE is proprietary to Microsoft.

- OpenDoc[3] is a compound document architecture based on SOM, the Systems Object Model of IBM. OpenDoc is similar to OLE, except that it allows network

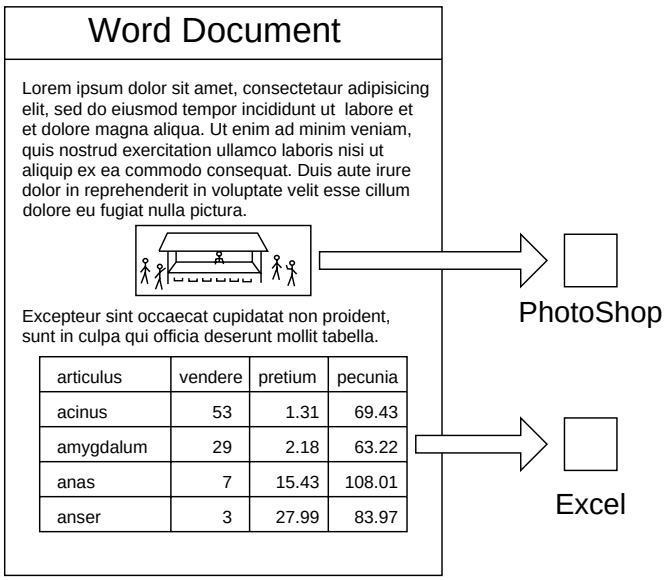


Figure 5.7: Object sharing in word.

transparency. Although OpenDoc is designed to be platform independent, it seems to be used mainly on Apple Macintosh.

The object sharing method is very flexible, as objects can be supported by multiple applications and combined where possible. Furthermore, high-level features such as distributed computing, accounting and mobile computing are supported by the object sharing architecture, thereby simplifying the applications that support the objects.

Object sharing is mostly used for documents with embedded objects, such as images, spreadsheets, animations or sound. However, within such documents, shared objects are often treated differently from native objects, while the combination of different objects is restricted or impossible.

5.2 Integrating existing tools

Since Mathƒpad documents are converted to LaTeX documents and the spell checking might be performed by an application like `ispell`, some level of integration with existing applications is necessary. Certain applications are available as source code, thereby allowing modifications which simplifies integration. However, modifying the source code of an existing application to add specific extensions is not advisable if that application is actively maintained, as the modifications have to be applied to every new release of the tool and users have to install the modified versions, thereby introducing version management problems with the modified application. Therefore,

an application should be integrated without any modifications, as that would ensure integration with as many applications as possible, whether the source is available or not.

Since an integrated application is not modified, all the modifications for the integration have to be applied to the MathʃʃpadMathʃʃpad system. However, if many applications were integrated, the system would become unmanageable due to all these extensions. Therefore, an architecture is needed to integrate the system with other applications in a useful manner. The applications that we have in mind are mathematical engines like Maple and Mathematica[60, 83], theorem provers like PVS[65], spell checkers like ispell and several compilers.

As the applications are not modified, the existing interface will be used. For applications with a graphical user interface, it is very difficult to integrate those with other applications, as the interaction with the user is almost impossible to simulate automatically. Therefore, integration with such applications will not be supported. For applications which provide interpreted configuration languages, it is possible to construct scripts to simplify the interaction. For applications, like ispell, that provide special features for integration with other systems, these features are used for the integration. For all other applications, the integration is based on a simulation of the user, that is, MathʃʃpadMathʃʃpad sends valid input and parses the output of the application.

As the architecture to integrate applications should allow easy addition of new applications, it should be easy to configure and extend. Since the object sharing architectures and software buses are not available or not used widely on UNIX systems, the use of dedicated links, plug-ins, helper applications and pipes seems to be the best choice. As the dedicated links are best achieved by using a script language, such a script language is added to the system in order to extend it with dedicated links. By making the language powerful enough, it can also support the extendibility of the system by means of plug-ins, helper applications and pipes. Although the use of software buses and object sharing architectures is not required yet, it would be nice if the language would allow support for those features in the future.

5.2.1 The interpreted language

To make MathʃʃpadMathʃʃpad easily extendible, an interpreted language has been constructed to combine the extensions. That language is based on the Guarded Command Language (GCL), which contains the basic building blocks as found in most imperative languages, such as constants, variables, assignments, statements, blocks and functions. Although there are some small differences, the language is straightforward.

As the language is used to extend the system by using plug-ins, everything that can be provided by means of plug-ins is not included in the language itself. This is achieved by using a general mechanism that allows plug-ins to define new types, functions and variables and to overload the meaning of available operators. Internally, the parser and interpreter use several databases for available constants, types, variables and functions.

Constants and variables

Constants and variables are treated by the interpreter in the same way, except that constants cannot be changed or passed as a call-by-reference parameter to a procedure. Therefore, every constant that appears inside a definition file results in an unnamed variable with a fixed value. The interpreter supports three kind of constants: integers, reals and strings. Integer constants are specified according to the usual convention, which includes the octal notation, decimal notation, hexadecimal notation and character notation. Real (floating point) constants use the standard floating point notation of programming languages. String constants are similar to the strings in C, where the usual escape sequences can be used to add special characters. As Mathʃpad uses Unicode strings internally, the string constants are first converted to the Unicode encoding, according to the character encoding of the script files, which is UTF-8 by default. After that conversion, the escape sequences are expanded. For easy use of Unicode characters, the `\uhhhh` sequence is available, similar to the one available in Java. Examples of constants are:

`5481`	integer, decimal notation
`012551`	integer, octal notation
`0x1569`	integer, hexadecimal notation
`'\u2709'`	integer, character notation
`54.81`	real
`5.481E+1`	real
`5481E-2`	real
`"Postcode"`	string
`"\u260E 112"`	string

Variables are either declared globally or locally, where block structures are used to create local environments. These blocks can be nested and the usual scoping rules apply to the variables defined inside a block. The name of a variable is case sensitive and consists of a combination of Latin characters, digits and underscores, where the first character cannot be a digit. As the language contains several keywords, these keywords are not allowed as variable names. Furthermore, the names of types are also not allowed as variable names.

To allow output arguments to procedures, the interpreter supports references to variables, where the conversion between referenced variables and non-referenced variables is performed automatically where needed.

Some examples of variable definitions are:

```
Var Int counter;
Var Real pi;
Var String pc_5481;
Var StringRef return_argument;
```

Expressions

The interpreter supports a standard set of operators to construct expressions. However, there is no default semantics for these operators. Instead, an extension can define the semantics of an operator by providing a function with the appropriate argument types. With these functions, the interpreter can determine the semantics of an expression by matching functions to the operators from the expression. The available operators are, in order of their precedence:

Operators	Intended meaning
`! ~`	Logical and bitwise not
`* / %`	Multiplication, division and remainder
`+ -`	Addition and subtraction
`<< >>`	Bit shifting
`= < > <= >= !=`	Comparison
`&`	Bitwise and
`^`	Bitwise exclusive or
`|`	Bitwise or
`&&`	Logical and
`^^`	Logical exclusive or
`||`	Logical or

These operators are based on the C programming language, although most of them are available in other programming languages as well. It is not very difficult to add new operators, for example to support operations on sets or lists, but it requires certain additions to the lexical analyser. Currently, these operators are constructed from ASCII characters, as that character set is the common denominator of all encodings. It would be nice if the set of mathematical operators from Unicode could be used instead, but that requires a precedence relation on these operators and an extension of the lexical analyser generator.

There are certain limitations to the operator overloading. Similar to C++, it is not possible to change the precedence, associativity or arity of an operator or to define new operators, as it introduces ambiguity and makes expressions very difficult to read by both humans and computers. For example, the operator `^` can be overloaded to mean "to the power" in floating point arithmetic. With that meaning, the precedence of `^` is incorrect. If the precedence could be changed, certain expressions would be very difficult to handle. The expression `2^3+5.1` is such an expression. As the `^` operates on two integers, its precedence is less than the precedence of the `+`, so it should be read as `2^(3+5.1)`. However, `3+5.1` is a floating point value, so the `^` operator means "to the power", in which case the precedence and the resulting value are incorrect. Therefore, adjusting properties of operators is not possible. The definition of new operators might introduce ambiguity as well, as the new operators can consist of combinations of existing operators. Furthermore, different modules could define the same operator with different precedences or arities.

In C++, operators can only be overloaded when at least one operand is an object of the class that defines the operator. This ensures that the built-in operator meaning cannot be re-defined. As the interpreted language does not have built-in operator meaning, the requirement is not needed. However, without that requirement, operators can be overloaded with a non-standard meaning, thereby making expressions unreadable, even for standard types such as integers or strings. Therefore, an extension should only overload operators with meaningful operations on types supported by the extension.

Statements

The constructs for statements are based on the Guarded Command Language and include the *multiple assignment*, *procedure calls*, the *sequential composition*, the *selection* and the *iteration*.

The *multiple assignment* is part of GCL for convenience and it is a generalised version of the normal assignment, as multiple variables are changed in a single assignment. The syntax of the multiple assignment is:

$$v_1 \, , \ldots , v_n \;\; := \;\; e_1 \, , \ldots , e_n$$

for variables v_1 to v_n and expressions e_1 to e_n. The semantics of this assignment is that the expressions e_1 to e_n are evaluated in some order and the resulting values are assigned to the variables v_1 to v_n in some order.

The C programming language also allows multiple assignments in one statement by using either $v_1{=}e_1$, $v_2{=}e_2$ or $v_1 = v_2 = e_2$. However, the semantics of both statements is very different. The first statement assigns different values to different variables, but assignments are made in a sequential order. It is mainly used at positions where only one statement is allowed, such as in for loops. The second statement uses a linked assignment, where assignments are used as expressions. It is mainly used in initialisations, where multiple variables are set to the same value, usually zero.

A *procedure call* allows structure and abstraction by means of hiding the internal computation. The syntax of a procedure call is

$$procedure\_name(arg_1 \, , \ldots , arg_n)$$

where the semantical meaning is that the arguments arg_1 to arg_n are evaluated and the procedure *procedure_name* is called with the resulting values.

The *sequential composition* combines two statements into a new statement. The syntax of the composition is:

$$S_1 \, ; S_2$$

where the semantical meaning is that statement S_1 is executed first and statement S_2 is executed next.

The *selection* allows the program to execute statements under certain conditions. The syntax of the selection is:

```
if      (B1)   S1
elseif  (B2)   S2
...
elseif  (Bn)   Sn
else           Sr
fi
```

The semantical meaning is that a statement S_i is executed for which guard B_i is true. If none of the guards is true, statement S_r will be executed. This semantical meaning differs from the meaning of the selection in GCL, where the unguarded S_r is not allowed and where the execution would abort if none of the guards is true. These adjustments are made to make the semantics more familiar to common practice in programming languages, where a selection doesn't abort execution and unguarded else clauses are allowed.

The *iteration* allows the program to execute statements repeatedly, according to certain conditions. The syntax of the iteration is similar to the syntax of the selection:

```
do      (B1)   S1
elsedo  (B2)   S2
...
elsedo  (Bn)   Sn
od
```

The semantical meaning is that a statement S_i is executed if guard B_i is true, until all guards are false. Notice that the else clause is not allowed in an iteration, as it would result in endless loops.

Most programming languages do not support a multi-guarded iteration directly, but it is not very difficult to construct it by means of existing iterations. For example, the above GCL iteration can be constructed in C with the following code:

```
while (1) {
  if          (B1)   { S1 }
  else if     (B2)   { S2 }
  ...
  else if     (Bn)   { Sn }
  else break;
}
```

Function definitions

For abstraction, reusability and readability, the language allows function definitions. As the interpreter performs type checking on the arguments of functions, each function definition provides a prototype for that function by listing the arguments and

their types. With the prototype, the arguments are also available as local variables. Within the function definition, statements, blocks and local variables can be used to complete the calculation.

It is currently not possible to define recursive functions as the interpreter does not handle local variables in a suitable way yet. However, many recursive functions can be written in a non-recursive fashion, especially if stack operations are available. Furthermore, recursive functions usually require a performance which is easier to achieve in a plug-in using a compiled language.

The arguments to functions are passed either by reference or by value, depending on the prototype of the function. As the arguments can be used as local variables within the function, the call by value conversion will copy values to temporary variables. To release the user of the burden to dereference the arguments, the interpreter performs this task automatically: a value is converted to a reference by taking the address of the value, a reference is converted to a value by dereferencing the reference and copying the value. Due to this automatic conversion, values and references to values are used without distinction.

It is possible to define local variables in functions by using blocks, which can be nested. The normal scoping rules apply to the nested blocks and it is possible to define variables that hide existing variables, although that is not advisable with respect to readability.

Internally, a user defined function is stored as list of variables, which includes the arguments and a graph of expressions. When the function is called, the variables are initialised and the starting point in the graph is selected. Then, the expression of the current point in the graph is evaluated and depending on the result of the expression, the next point in the graph is selected. This is repeated until no next point is available. When the function is finished, the local variables are cleaned up. Figure 5.8 shows the graph of expressions for the GCD function, which calculates the greatest common divisor of two integers.

5.2.2 Plug-Ins

The extensions might become too complex to construct them in the interpreted language and the efficiency of the interpreter might become a bottleneck. To allow complex extensions, the interpreter is extended with a run-time library loader which enables the dynamic loading of additional compiled code by the interpreter. An extension can use a compiled library or plug-in to provide the interpreter with new functions which require high performance or low-level access, while the interpreted language can be used to customise the extension.

The plug-in extends the interpreter with new items, such as new object types, operator definitions and functions. To ensure that the capabilities of the plug-in mechanism are sufficient, the interpreter does not provide any types, operators or functions by default, except for the ones which are vital, such as the functions to open files to be interpreted. As a result, the interpreter usually loads a collection of default plug-ins

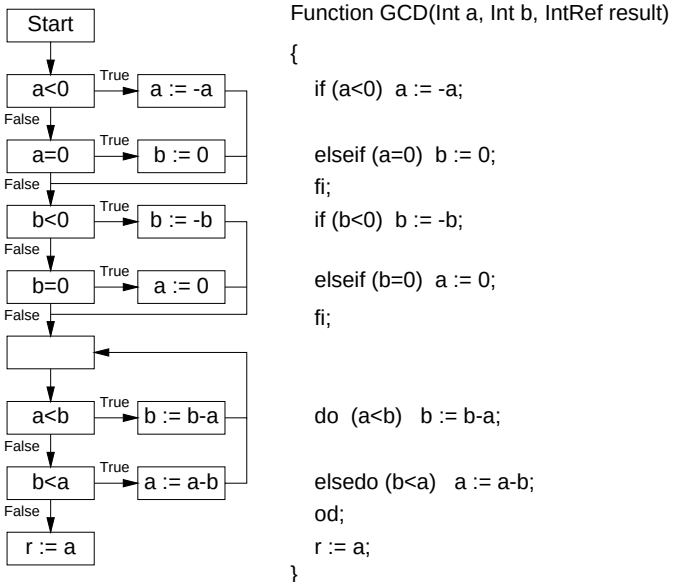


Figure 5.8: The graph of expressions for the GCD function.

to allow standard types such as integers, floating points and strings.

When a plug-in is loaded with the system function `dlopen`, the interpreter has to extract the new items from the library. This is done with the system function `dlsym`, but it requires the name of an item in the library. Although it would be possible to provide these names through the interpreted language, it is easier to add a function with a standard name to the library which will add all the items to the interpreter and initialise any data used within the library. By using such a standard function, all the information that is needed to load the library is contained within the library itself. Furthermore, the function `dlsym` has to be called only once with the name of the standard function and loading a plug-in behaves similarly to loading an interpreted file.

In the following discussion of the plug-in features, a plug-in to handle strings will be used as leading example. The strings as defined by this plug-in are similar to strings in C, that is, arrays of characters with a termination null character. The plug-in library is written in C, so be warned for upcoming C code, which is quite nasty due to all the type conversions.

Type definitions

A plug-in can extend the language by defining a new type. In order to handle such a type internally, the plug-in has to provide functions to construct, destroy and copy objects of the given type. The function to add a new type is:

```
Type define_type(String name, Integer size,
                 Function construct(Var ObjectPointer),
                 Function destruct(ObjectPointer),
                 Function copy(Var ObjectPointer, ObjectPointer))
```

The `name` argument is used by the parser of the interpreted files to identify variables of the given object. The other arguments are used to handle objects internally during the interpretation phase, where the `size` argument indicates the size of an object, such that the default function can be applied. The `construct` function is used to initialise objects, the `destruct` function is used to destroy objects and the `copy` function is used when objects are passed as a call-by-value argument to a user defined function. These three functions can use reference counting on objects if the object and its operations allow it.

The function returns a type identifier, which can be used for further reference by the plug-in. There is no guarantee that the type identifier will be the same in different instances of the plug-in, as it depends on the number of types that are already defined.

As a plug-in might want to refer to types supplied by different plug-ins, two lookup functions are available to get the name of the type when a type identifier is given and to get the type identifier when the name is given:

```
Type lookup_type(String)
String lookup_typename(Type)
```

For each type provided by a plug-in, the interpreter will define a reference type, which is used to pass output arguments to functions. For a given type `Name`, the interpreter will define `NameRef` and conversion between the two is done automatically.

Example The string plug-in defines the type `String`, which is just a pointer to an array of characters. As each string requires an unknown amount of memory, the plug-in has to handle the memory allocation to store the strings correctly. To make sure that no memory leaks occur, each operation on strings has to make sure that no memory is lost. This is achieved by using reference counting, where each operation updates a counter correctly in order to keep track of the number of references to a particular string, such that the string can be released when it is no longer needed.

```
typedef Uchar* String;

void construct_string(void **object)
{
   *object=NULL;
}
void destruct_string(void *object)
{
   decrease_refcount(object);
}
```

```
void copy_string(void **object, void *orig)
{
   increase_refcount(orig, free);
   *object=orig;
}

int init_library(void)
{  Type stringtype;
   ...
   stringtype = define_type("String", sizeof(String),
                            construct_string,
                            destruct_string,
                            copy_string);
   ...
}
```

The functions `construct_string`, `destruct_string` and `copy_string` are used by the interpreter. The plug-in handles NULL strings as empty strings, which results in a simple construct function. The reference counting mechanism will handle the cleanup operation when the string is not needed anymore, so the destruct function only calls `decrease_refcount`. As reference counting is used, a string value can be copied by increasing the reference counter, so the copy function consists of calling `increase_refcount` and making an assignment.

Prototypes

In order to verify that all functions and expressions are valid, function prototypes are used. A function prototype consists of a list of type identifiers and a function caller. The list specifies the type of each argument, which is used by the interpreter to perform certain conversions. The function caller receives as arguments a function internal to the plug-in and a list of pointers, each pointing to an object of the appropriate type. The function caller performs the required conversions on those pointers and calls the provided function.

A prototype is defined with the following function:

```
Prototype define_prototype(Type argumentlist[], Integer length,
                           Type result,
                           Function caller(Function,Pointer[]))
```

As the type identifiers are not fixed, the list of types has to be entered by using the return value of `define_type` or by calling `lookup_type` with the names of the correct types. The function `caller` is usually a very simple function, which converts the pointers and passes them to the given function.

The returned prototype is used to extend the interpreter with functions from the plug-in, which comply to that prototype.

Example Although the plug-in supplies more prototypes, the following example should be sufficient to show how the definition of prototypes works.

The definitions of the caller functions are a bit obscure due to the use of C and the way it passes arguments to functions. To ensure that the arguments are passed correctly without having to rewrite every function, the caller function receives a list of pointers to the appropriate objects. By casting those pointers to the correct type, the arguments are passed correctly to the functions that are called. This results in the ugly code for these caller functions.

In the `init_library` function, four prototypes are defined and assigned to variables. When a large number of prototypes has to be defined, the arguments to `define_prototype` can be stored in an array and an iteration over the array can be used instead.

```
int call_stringstringref(int (*function)(), void **arguments)
{
   return (*function)(*((String*)arguments[0]),
                      *((String**)arguments[1]));
}

int call_stringstring_resstring(String (*function)(),
                                void **arguments)
{
  *(String**)arguments[2] = (*function)(*((String*)arguments[0]),
                                        *((String*)arguments[1]));
  return (*((String**)arguments[2]))!=NULL;
}

int call_stringstring_resinteger(int (*function)(),
                                 void **arguments)
{
  *(int*)arguments[2] = (*function)(*((String*)arguments[0]),
                                    *((String*)arguments[1]));
  return 0;
}

int call_stringinteger_resinteger(int (*function)(),
                                  void **arguments)
{
  *(int*)arguments[2] = (*function)(*((String*)arguments[0]),
                                    *((int*)arguments[1]));
  return 0;
}

int init_library(void)
{  Type list[2];
```

```
   Type inttype;
   Prototype protoassign, protocombine,
             protocompare, protoelement;
   ...
   list[0]=stringtype;
   list[1]=ToRefType(stringtype);
   protoassign = define_prototype(list, 2, NoType,
                                  call_stringstringref);
   list[1]=stringtype;
   protocombine = define_prototype(list, 2, stringtype,
                                   call_stringstring_resstring);
   inttype=lookup_type("Int");
   protocompare = define_prototype(list, 2, inttype,
                                   call_stringstring_resinteger);
   list[1]=inttype;
   protoelement = define_prototype(list, 2, inttype,
                                   call_stringinteger_resinteger);
   ...
}
```

Function definitions

The functions internal to the plug-in are made available to the interpreter by calling the function

```
FuncRef define_function(String name, String description,
                        Prototype prototype, Function function())
```

`name` is used by the interpreter to identify the function. `description` gives an (informal) description of what the function does, which is used to provide some help and to summarise the content of the plug-in. `prototype` provides the prototype of the supplied function, which includes the number of arguments, the type of arguments and the result type. `function` is the compiled function that is added, which can be called with the function caller provided by `prototype`.

Example Again, only a small collection of the available functions is given to indicate how the definition of functions works. After the C functions are defined, they can be added to the interpreter.

```
void stringassign(String s, String *t)
{
   increase_refcount(s,free);
   decrease_refcount(t);
   *t=s;
}
```

```
int stringelement(String s, int pos)
{
   if (pos<0 || !s)
      return 0;
   else
      return s[pos];
}

String stringadd(String s1, String s2)
{
   int j = Ustrlen(s1);
   int i = j+Ustrlen(s2)+1;
   String t;
   t = malloc(i*sizeof(Uchar));
   Ustrcpy(t,s1);
   Ustrcpy(t+j,s2);
   increase_refcount(t,free);
   return t;
}

int stringequal(String s1, String s2)
{
   return !Ustrcmp(s1, s2);
}

int stringsmaller(String s1, String s2)
{
   return Ustrcmp(s1, s2)<0;
}

int init_library(void)
{
   FuncRef assign, element, add, equal, smaller;
   ...
   assign  = define_function("StringAssign",
                             "Assign one string to another",
                             protoassign, stringassign);
   element = define_function("StringElem",
                             "Get a certain character of a string",
                             protoelement, stringelement);
   add     = define_function("StringAdd",
                             "Concatenate two strings",
                             protocombine, stringadd);
   equal   = define_function("StringEqual",
                             "Are two strings equal?",
                             protocompare, stringequal);
```

```
   smaller = define_function("StringSmaller",
                             "Is the first string smaller?",
                             protocompare, stringsmaller);
   ...
}
```

As reference counting is used, the assignment of strings is equal to increasing and decreasing the reference counters of the appropriate strings and making an assignment. The function for accessing characters within a string, `stringelement`, checks the lower bound (0) of the string, but not the upper bound, as that requires the calculation of the length of the string, which influences the performance of the function too much. Therefore, when the characters in a string are used in the interpreted language, the user has to do the bound checking, which isn't very complicated. The `stringadd` function combines two strings by concatenating them, which requires new memory to be allocated and the two strings to be copied. To ensure that the allocated memory is deallocated in time, the reference counter is set. The comparison function `stringequal` and `stringsmaller` just call the standard function `Ustrcmp`, which compares Unicode strings similarly to the system function `strcmp`.

The name of the function is rather verbose and should be self-explanatory, although a description is given. It is possible to use these functions in the interpreted language, but it easier to use the operator overloading, which is discussed next.

Operator overloading

An operator is overloaded by first defining a function with `define_function` for handling the specific combination of arguments and then defining that the operator will be handled by that function, by calling:

```
define_operator(Operator operator, FuncRef function)
```

As `function` includes a prototype, the argument types and result type for `operator` are extracted from the information provided by `function`.

Example In the `init_library` function, everything is available to overload the operators.

```
int init_library(void)
{  ...
   define_operator(OPASSIGN, assign);
   define_operator(OPARRAY, element);
   define_operator(OPADD, add);
   define_operator(OPEQUAL, equal);
   define_operator(OPLESS, smaller);
}
```

With the overloading of these operators, the following expressions are valid in the interpreted language:

```
Var String s,t;
Var Int i;

s := "Hello";
t := s + " world!";
if (s<t)  i:=s[0]; fi;
```

Internal variables

A plug-in might also make its internal variables available to the interpreter, which enables easier configuration of the plug-in by means of assignments and inspections. The plug-in can use such variables internally like any normal variable. Internal variables are added to the interpreter with the function

```
define_program_variable(Type type, String name, Pointer address)
```

Example The string library does not require any internal variables to be made available to the interpreter, as the interpreter is able to handle string constants directly. As an example, the library adds the variable `easter_egg` to surprise users when they try to define a variable with that name.

```
String easter_egg=NULL;

int init_library(void)
{
   ...
   define_program_variable(stringtype, "easter_egg",
                           &easter_egg);
}
```

5.2.3 Menus

As the extensions provide the interpreter with new functions, which have to be accessible to the user, the language is extended with interface specific features, such as the ability to define input methods with keyboard mappings and to define popup menus.

Although it would be possible to define popup menus by calling the appropriate functions provided by the interpreter, the menus are such an important part that easy definition of such menus is required. Therefore, the interpreter allows the definition of popup menus with a special language construct, which has the following form:

```
Menu identifier {
        Options Pin;
        Title "title";
Default "Item 1"     :  function1(arg1, arg2);
        "Item 2"     :  function2(arg1);
        Separator;
        "Item 3"     :  submenu_identifier;
}
```

The `identifier` is used to refer to the menu when it is used as a submenu of other menus. The option `Pin` is used to specify that the menu can be pinned up on the screen, such that the menu is directly accessible, which is especially useful for nested or often used menus. The `Title` is used as the window title. After the global information, the menu items are listed, which might be either a description and a function call, or a description and an identifier of another menu, or a separator. When the attribute `Default` is used, then the given menu item will be selected by default, that is, when the item is selected where the given menu appears as submenu. For example, when `Item 3` is select while the submenu is not suppose to open, the default action for menu `submenu_identifier` will be executed.

As an interpreted file might want to add new items to an existing menu, it is possible to extend a menu by defining it a second time but only specifying the additional items. Items that already appear in the menu will be replaced by the newly specified items.

5.2.4 Keyboards

Many users prefer to use the keyboard instead of menus. However, different users are familiar with different types of keyboard definitions. Therefore, the interpreter allows the construction of keyboard definitions by a special construct, as keyboard definitions should be easy to construct or modify.

As keyboard definitions differ greatly from application to application, the interpreter supports several common key handlers, such as prefix keys, modes, temporary modes and stacks. Furthermore, as every window might have a different content, the keyboard handler might be specified on a window by window basis.

The special construct provided by the interpreter looks similar to the construct to define menus, except that keys are specified instead of strings. As the back quote (‘) is not commonly used in programming languages, it is used by the interpreter to denote keys.

```
Keyboard dummy {
        Clear;
        ‘Help‘       :  open_sensitive_help();
        ‘Left‘       :  move_backward(1);
        ‘Right‘      :  move_forward(1);
```

```
        ‘ ‘ ... ‘~‘  :  self_insert(pressed_key,1);
        ‘CM-Delete‘  :  reboot();
        ‘F35‘ ‘F13‘  :  easter_egg:=easter_egg+"Easter! ";
        ‘Again‘      :  use_map("dummy");
}
```

5.2.5 Translations

For customising the interface, the interpreter allows the user to specify translation strings for all the messages used internally by the application or plug-ins. These internal strings are all ASCII strings due to the limitations of programming languages. Therefore, a conversion is already required, as Mathʃpad uses Unicode strings for all messages.

As customisation of the interface is important, the language is extended with a new construct to specify translations:

```
Translation Dutch {
        "English"       :  "Engels";
        "File"          :  "Bestand";
        "Directory"     :  "Map";
        "Screen"        :  "Scherm";
        "Are you sure?" :  "Weet u het zeker?";
}
```

If no translation strings are specified, the internal strings are used after a default conversion to Unicode.

5.2.6 Scripting

The interpreted language is also used as a scripting language, where the commands are executed as they appear. This enables easy customisation and execution in batch mode. Furthermore, it allows other applications to send execution requests to the interpreter, similar to remote procedure calls.

5.2.7 Missing features

It is not yet possible to specify watch functions, that is, functions that are called when certain events happen, for example when a variable changes. Such watch functions might be needed when the interface depends on certain variables and has to be consistent all the time, as in popup menus with grey items. The variables internal to a plug-in are the main hurdle to add this feature, as the plug-in might have functions with side-effects on the registered variables, which makes it difficult to trigger the watch functions.

Another use of watch functions would be to extend existing functions by calling the watch function just before or after another function, as is possible in the configuration language of Emacs. Although such extensions are useful, they might lead to race conditions, where watch functions call each other recursively.

The language does not support type constructors, such as arrays, records, lists or sets. Nor is there a way for a plug-in to extend the language with type constructors. It would require functions with arguments of some generic type, which disables the type checking.

5.3 An example connection: PVS

Technical reports often contain parts written in multiple languages, such a mathematical language, a programming language, a script language and a natural language. The use of multiple languages can easily introduce errors due to manipulations of the document. Furthermore, the different parts can get out of sync, where the program in the report is not the program that is compiled. To reduce such errors, a tool is needed to verify that the report is correct up to a certain point.

The Ph.D. thesis by Matteo Vaccari[78] is an example of such a technical report with multiple languages. It contains a description of circuits in a mathematical language with properties and calculational rules on those circuits, the properties are proven with PVS and the circuits are tested with Tangram. The translations between these languages had to be made by hand, but it would be nice if some system would support these translations to some extend. As Vaccari wrote his thesis with Mathʃpad, a connection to PVS would be a good test case for the Mathʃpad architecture.

5.3.1 The PVS system

From a user interface point of view, PVS is an extension of Emacs, which connects the proof engine to the Emacs interface and the Tcl toolkit. The user can edit files containing theorems and use the proof engine to construct the proof interactively. Since the proof engine is basically a lisp interpreter with state information, the user interface of the engine is hidden from the user by a collection of pull-down menus in Emacs. With these menus, the user can perform all the actions that might be needed to manipulate files, theorems, lemmas and proofs. However, to construct a PVS proof, the user has to enter plain lisp commands to apply tactics to a goal. To allow some proof planning, an interface with a Tcl program is available to keep track of the subgoals.

Since PVS is a closed system and cannot be modified, the available PVS interface had to be used. The first problem was the existing Emacs interface, which complicates the communication with PVS. Luckily, the PVS system consists of a core system connected to Emacs with a collection of Emacs lisp files, as shown in figure 5.9. Emacs contains a lisp interpreter which is used to load the lisp files for the PVS communication. These configuration files extend Emacs with new functions and

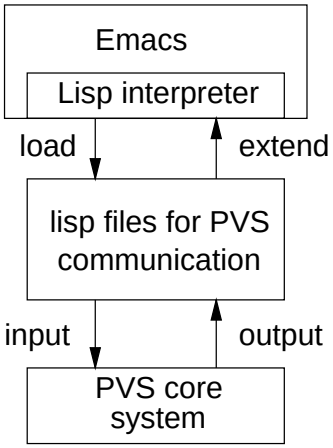


Figure 5.9: The PVS structure

menus to provide a PVS specific interface. With these additional functions, Emacs is able to communicate with the PVS core system, using its standard input and output.

By using the PVS core system directly, the communication is simplified and easier to maintain. However, the protocol used between Emacs and the core system is not documented, probably because the constructors of PVS didn't envisage a different interface to that core system. Therefore, the protocol had to be extracted from the lisp configuration files used by Emacs and the messages that are sent between Emacs and the core system. With some detective work in the form of a wrapper script that monitors these messages, we were able to reconstruct the most important parts of the protocol, which was sufficient to use the core system without the Emacs interface.

The PVS core system is a lisp interpreter and receives lisp commands as input, which can be used to update or inspect the state of the system or to prove a theorem. The output of the core system consists of a combination of commands to update the state of Emacs and the results of proving a theorem. The core system might also construct temporary files and instruct Emacs to open them, which is mainly used for help files and the Tcl interface. The Emacs interface cleverly hides the lisp input with a collection of pull-down menus, while the mixed output is parsed and separated into several buffers. For the average user, only the buffer with the results of a proof are of interest.

The PVS core system operates in three modes: a mode for managing the state, a mode for making the proofs and a debug mode. Since the active mode affects the commands that Emacs has to send, the system uses synchronisation points when it switches to a different mode and notifies Emacs. The debug mode is only used if the system receives incorrect input, and this mode is ended by resetting PVS.

To construct a different user interface for the PVS core system, the user interface had to simulate the actions performed by Emacs, such that the core system could

A law about *map* and *fold* is the following: given R and S such that

$$R \circ S\times S \;=\; S \circ R$$

then

$$fold_n.R \circ map_n.S \;=\; S \circ fold_n.R$$

The proof is by induction on n; for $n = 1$ it is trivially true. For $n + 1$ we have

$$\begin{array}{ll} & fold_{n+1}.R \circ map_{n+1}.S \\ = & \quad \{ \text{ definitions } \} \\ & R \circ \iota\times fold_n.R \circ S\times map_n.S \\ = & \quad \{ \text{ fusion } \} \\ & R \circ S\times(fold_n.R \circ map_n.S) \\ = & \quad \{ \text{ induction hypothesis } \} \\ & R \circ S\times(S \circ fold_n.R) \\ = & \quad \{ \text{ proviso: } R \circ S\times S \;=\; S \circ R\text{; fusion } \} \\ & S \circ R \circ \iota\times fold_n.R \\ = & \quad \{ \text{ definition } \} \\ & S \circ fold_{n+1}.R \end{array}$$

Figure 5.10: The presentation style for proofs

not notice the difference. As our plan was to hide PVS as much as possible from the user, only a subset of the actions available in Emacs were made available in the new interface.

5.3.2 The PVS interface plug-in

The PVS module consists of a plug-in for communicating with the PVS core system and an interpreted file to adjust the interface with MathſPad. The purpose of the plug-in is threefold. First, it interprets the MathſPad document to extract theorems and proofs. Second, it generates the input that is sent to the PVS core system. Third, it parses the output that is generated by the core system.

As a MathſPad document is structured, the generation of theorems from a proof given in the Eindhoven style is not so difficult. For the example in figure 5.10, each of the five steps has to be correct, so we can generate a theorem for each step. Since the syntactical differences between the PVS input and the MathſPad version are not very different, generating these theorems is straightforward, once the definitions of

the templates are correct. Extracting the proof for these theorems is also possible, as the hint contains keywords that indicate which strategies are applicable. In the example, the keyword "fusion" indicates that the fusion lemma is used as a rewrite rule. The keyword "definition" indicates that some definition has to be expanded and the keyword "induction" indicates that a premise is used as a rewrite rule, where the premise can be constructed from the proof itself by using the first and last expressions. However, the hints are not always precise enough, as PVS often requires more detailed information on how to apply a rule. The additional details are automatically applied by a human reader of the formatted proof, without complaining. The reader will apply the trivial laws, such as the "identity of composition" and "associativity of composition", when needed and the direction in which an equality law is used is determined by trial and error. A complex dialogue with the author could be used to get these additional details, but a better method would be to define additional PVS strategies to simulate the behaviour of a human reader:

- a strategy to apply a rewrite rule in both directions,
- a strategy to retry a given strategy after applying the trivial laws, if that strategy fails the first time,
- a strategy to apply a rewrite rule modulo composition.

These strategies have their limitations, as it is likely that rewrite rules are applied incorrectly. However, the theorems are usually small and their proofs are short, so it is less likely that something will go wrong. In the event that a theorem cannot be proven, an indication that the given hint is not sufficient to prove that step should be a reasonable reply from the system, as a reader might have the same problems with it as PVS.

For the other part of the example, the extraction of the theorems would require a combination of natural language processing and logical reasoning, for which a general solution is difficult. Therefore, this part is still missing from the constructed PVS interface.

Once the theorems and proofs are known, they have to be converted to the specification language used by PVS. Since the output generated by Math∫pad depends on the templates that are used, it is possible to generate valid PVS input from Math∫pad expressions without much additional programming. However, the expressions appear in a certain context and the identifiers should have a certain type, otherwise PVS will generate parse or type errors. Although the context and type information can be stored in the document as hidden information, a default context is used, where certain definitions and identifiers are predefined. This approach is quite common in documents with many identifiers, as it releases the author from the burden of mentioning the type of an identifier over and over again.

In order to give feedback to the user, the PVS output should be parsed and converted to familiar syntax. If a theorem is correct, all is well and a simple message should be sufficient. Otherwise, a warning or error message should be generated, indicating the problem and possibly a solution. If PVS does not need the generated proof

completely, it might be that the given hint is incorrect or over-complete and Mathʃpad will suggest to adjust the hint in order to avoid confusing the reader. If PVS is unable to prove the theorem, the hint might be incomplete or an error might have occurred. By inspecting the output and comparing the expressions Mathʃpad could suggest that an identifier is incorrect or that a particular law might be applied. Since the user is not familiar with the PVS language, the output of PVS is parsed and shown in the language as used in the document with a familiar syntax. However, templates are used to generate the PVS expressions, which allows ambiguity.

The PVS output also contains commands which are handled by Emacs. For each command, the PVS module will either ignore it or translate it to the new interface. For example, after the PVS core system has finished a proof, it will tell Emacs to open a buffer with the PVS file that contains the proven theorem. In Mathʃpad, that PVS file is generated by a step in a proof and of no interest to the user, so Mathʃpad will highlight the step that generated the PVS file.

The plug-in adds three functions: `pvs_check_hint()`, to check the selected hint, `pvs_start()`, to start PVS, and `pvs_add_keyword()`, to define a keyword like "induction" mentioned above and the related PVS strategies. These functions, together with the already available functions, are used in the pop-up menus to extend the interface of Mathʃpad, for example to start PVS and to check a selected hint. The plug-in also adds the variables `pvs_initialized` and `pvs_in_checker`, which can be used to inspect the status of PVS, and `pvs_context_dir`, `pvs_hint_file` and `pvs_lemma_name`, which are used to customise the generation of PVS files.

Since Mathʃpad uses Unicode internally, the strings that are part of the plug-in, such as error messages, have to be converted to Unicode before they are used. This conversion uses a translation table to check whether the string has been customised by the user. This leaves a plug-in with an additional method of customisation: by converting a string with the translation table, it can be adjusted by the user. In the PVS module, the string "PVS_HEADER" is used as the header of the PVS file, which defines the context of the generated theorem. By defining a translation for this string, the correct header is used.

In addition to the theory-specific keywords, there are four keywords with a special meaning. Each of these keywords is used in a special case:

- INITSTEP is used to initialise the PVS proof and to remove universal quantifiers.
- FINISHSTEP is used to finalise the PVS proof by applying all the trivial steps,
- EXPRESSIONSTEP is used when an expression occurs within a hint. Expressions in hints are regarded as assumptions and will result in a premise.
- STOPPVSPROOF is used when the proof fails and PVS has to leave the proof mode.

Without these four keywords, a correct proof script cannot be constructed. Therefore, the interface definition file has to define these keywords with the `pvs_add_keyword` function.

5.3.3 The definition file

The PVS plug-in handles the communication with the PVS core system and provides the interpreted language with a collection of high-level functions. With these functions, the popup menus of MathSpad have been extended with PVS specific commands or submenus. The interpreted language is also used to initialise and customise the PVS plug-in, for example by filling the keyword list and setting up the context. Some parts of the definition file for PVS are shown in figure 5.11 on page 114.

First, the plug-in is included, meaning that the functions and variables from that plug-in become available to the interpreter. It is also possible to include other definition files, which can be used to divide the different aspects of the interface over separate files.

After the plug-in has been included, the function `pvs_reset` is defined, which is used to reset PVS if something goes wrong. This function could also be part of the plug-in, but defining it in the interface definition file is more flexible, as it can be adjusted more easily.

Once all the functions are available, they can be linked to a pop-up menu and the keyboard. The interface definition language has special constructions to make this as easy as possible. A pop-up menu is defined by making a list of menu items, each containing a description and either the function to be called or the submenu to be opened. In the example, the menu called `PVSMathSpad` gives the user access to four PVS-specific functions. The menu itself is added as a submenu to the menu called `Misc`, which lists miscellaneous features.

Three functions are made available through keyboard shortcuts. After the `Meta-p` prefix, the key `s` will start PVS, the key `c` will check the selected hint and the key `r` will reset PVS.

To customise the PVS plug-in, two translation strings are defined. As explained earlier, a translation for the string "PVS_HEADER" is given to set the context for the generated theorems. In general, this translation mechanism is used to customise the messages from MathSpad, as these are all in English and perhaps not clear enough (as in 'folder' versus 'directory').

At the end, the variables are initialised and the database of keywords is filled. At this point, the definition file is used as a script file to execute the functions while the definition file is loaded, which is used to further customise the plug-in.

```
Include "libpvs.so"

Function pvs_reset()
{
    if (pvs_initialized) {
        send_signal(2, "PVS Session");
        send_string(":reset\n", "PVS Session");
        pvs_in_checker := 0;
    }
}
Menu PVSMathSpad {
    Options Pin;
    Title "PVS Link";
    "Start"      : pvs_start("PVS Session");
    "Check Hint" : pvs_check_hint(1);
    "Reset"      : pvs_reset();
    "Exit"       : send_string("(pvs::lisp (ILISP:ilisp-restore))
                       (pvs-errors (exit-pvs))\n", "PVS Session");
}
Menu  Misc {
    "PVS"   : PVSMathSpad;
}
Keyboard Global {
    `M-p` `s` : pvs_start("PVS Session");
    `M-p` `c` : pvs_check_hint(1);
    `M-p` `r` : pvs_reset();
}
Translation English {
  "PVS-shell"           : "PVS Session";
  "PVS_HEADER"          : " [t: TYPE+] : THEORY
 BEGIN
 IMPORTING tuples[t]
  n,m: VAR upfrom(1)
  R,S,T,U: VAR rel
";
}

pvs_context_dir := "/home/river/pvs-test";
pvs_hint_file := "hint";
pvs_lemma_name := "hintlemma";
pvs_add_keyword("STOPPVSPROOF", "(quit)\nY\n\"nil\"\nno\n",0);
...
pvs_add_keyword("induction",
                "(then* (inst?)(ground)
                  (try-triv-step (bidi-replace*)))\n", 1);
```

Figure 5.11: The interface definition file

Chapter 6

Discussion and conclusions

The development of MathSpad started in 1992 with much fewer requirements, as it was a final year project[80], which had to be finished in six months. The final year project was completed in co-operation with Olaf Weber. During the years that followed, new requirements were added and the existing system was extended in order to meet these requirements. Some of these additional requirements include the use of templates for entering text-related constructs like section titles and support for external systems. In December 1993, the system was first released as version 0.31 and a questionnaire was sent to get some feedback. In June 1995, version 0.50 was released, followed by version 0.60 in March 1996. The current version, which required a large rewrite of the internal system, was released in January 2000 as version 0.80.

This chapter consists of three sections. The first section discusses the current system and which parts of the system might require some improvements. The second section contains possible application areas of the current system and discusses which modifications might be needed to support these areas better. The third section contains a short description on how to build a similar system again, using the experiences with the current system and making use of current technology.

6.1 The current system

The current system works according to requirements of the target users. That is, it helps in preparing large and complex mathematical documents and generates error-free LaTeX code for the structures that are used. As it is possible to include raw LaTeX markup, it is still possible that there are markup errors in those parts. The flexibility of the templates is used for several purposes, which weren't anticipated when the system was first constructed, such as generating multiple views on the same document or entering letters using templates as fill-in forms. At the moment, the system is freely available on the Internet, including the online help and a tutorial. It works on several UNIX systems and it should not be difficult to install.

As with many document preparation systems, Mathʃpad is never finished and it is always possible to extend the system with new features. Although Mathʃpad works properly and it is always possible to enter plain markup to the document, there is room for improvements. Some of the problems with the current system are described in the following sections.

6.1.1 The learning curve

The Mathʃpad system is a structure editor and therefore quite different from existing text editors or document processing systems. Furthermore, Mathʃpad uses LaTeX for printing purposes and some basic knowledge of LaTeX is still needed. As a result, it might take quite some time to learn to use the system, even when the online tutorial is used to get started.

It is possible to reduce the learning curve by copying the behavior of another system, such as Word, WordPerfect, Emacs or VI, depending on the background of the user. For example, Microsoft Word is able to mimic the behavior of WordPerfect, in order to make the transition to Word easier and Emacs provides a VI mode for VI users. However, to mimic the behavior of such systems, you have to translate operations in those systems to operations on tree structures, such that the user doesn't notice the difference. This is rather difficult due to the operations that are allowed in those systems, such as free selection and deletion.

6.1.2 The interface

The interface that Mathʃpad presents to the user is simple yet functional. The interface resembles the style used by several tools available at the time the interface was constructed, but most modern applications use more appealing interfaces with lots of colourful buttons with icons. In order to make Mathʃpad more accessible, the interface of Mathʃpad should be updated to the current standards. In the updated interface, which could look similar to the interface of Word or WordPerfect, the templates for adding textual items could be made available through small buttons, making it easier to get started with the system.

To update the interface, the use of a toolkit is advisable. However, the situation has not changed much in six years. There are still about a dozen toolkits available with a different look-and-feel and putting mathematical symbols or notations on popups is still difficult. The experimental port to Windows95 indicated that the internal processing is portable, except for the drawing-related parts. Therefore, reconstructing a new interface should not be too difficult, once the drawing routines are sorted out. The main problem might be related to the use of dynamically generated popup menus for templates and the use of mathematical notation on those menus.

By choosing a graphical toolkit which is available on multiple platforms, such as Tcl/Tk or Java, only one interface has to be written to support these platforms and native methods could be used to access the internal workings. However, Mathʃpad has special requirements with respect to fonts and in the toolkits that are available

for multiple systems, only the common features are available, which means that vital information about character sizes is not directly accessible, which makes it quite difficult to use the fonts properly.

By using a platform-specific toolkit, the interface has to be ported to a different system in order to use the application there. However, the toolkit would more likely follow the look-and-feel of the platform it is written for and it would allow access to platform-specific data. For the Microsoft platform, one of the visual interface builder tools can be used to construct the skeleton of the interface and the internal functions can easily be connected to the buttons and menu items. For the UNIX platform, the situation is a bit different. Most UNIX variants, such as Solaris, Linux, Irix and HP, use their own toolkit with their own look-and-feel, which makes it difficult to write an application that works properly on all platforms, especially when interaction between graphical applications is required. On Linux, the KDE environment with its Qt toolkit is common and it also works on several other UNIX platforms, although the look-and-feel might differ slightly. So choosing the Qt toolkit covers most UNIX systems, although the Qt license agreement requires that the source code is made available. Furthermore, this toolkit has the same look-and-feel as the Windows toolkit, so it allows Window users to use the system more easily as well, for example through an X emulator. Another advantage is that it supports Unicode to some extent. Other operating systems, such as MacOS and BeOS, use their own toolkits, which makes it more difficult to port to those systems.

Although the use of a toolkit guarantees a common look-and-feel, certain parts of Mathʃpad are difficult to translate into that look-and-feel. For example, most toolkits require Cut, Copy and Paste operations to be located in an Edit menu and it would be possible to comply with that. However, the multiple selection mechanism is very useful when mathematical content is edited, as it contains more duplication of content than other data formats, such as text, spreadsheets or images. Supporting multiple selections is therefore useful, but it is not possible to make these selections in a similar way, as the look-and-feel guide lines determine how the mouse buttons should be used, either with or without modifier keys such as shift or control. As a result, a different and more complex method of making selections is required.

6.1.3 Converting legacy documents

One of the problems of starting with Mathʃpad is the fact that all the old documents are not converted to the WYSIWYG representation of Mathʃpad. It is possible to import plain text into a document, but that doesn't improve the editing facilities on that text. However, parsing documents in a markup language that allows macro definitions, as LaTeX does, is very difficult due to these user-defined macros. As the templates can be used to generate such markup documents, it might be possible to parse the documents using the information available in these templates. That way, a parser can be constructed by defining the correct templates and if strange markup sequences are used, the parser can easily be extended by defining additional templates. An experimental version of such a parser was tested and it was possible

to parse LaTeX and HTML documents partially.

The problem with such a parser is that the input documents are usually constructed by hand and not regular enough for the parser to work properly. Furthermore, if the parsed document is used to generate new output, that output should be equal to the original input, as any difference could influence the meaning of the markup sequences. Even for automatically generated markup, such as HTML documents produced by a document preparation system, there might be errors, such as tags that are nested incorrectly tags. As a result, the parser should be able to handle all kinds of errors gracefully.

A different approach to the parsing problem would be to build the document structure while the input is read, such that the position within the structure is used to guide the parser. That way, it is possible to parse input that is delayed, such as computation results or HTML documents received over slow network connections, while the document is regularly displayed. This assumes that the input is correct, but it is possible to handle errors like missing end tags gracefully. The position in the structure can be used to calculate the closure-set for that position and what should happen in case of errors. For example, the HTML markup sequence `<B><I>Example</B></I>` is incorrect. While parsing that sequence, the structure for `<B><I>Example</I></B>` is constructed and the incorrect `</B>` tag is found. The parser would expect a `</I>` tag first, but the position in the structure indicates that that tag is probably missing, as `</B>` is the closing tag for an element at a higher level. As the markup is incorrect, the parser might handle the situation in different ways depending on the markup language or context: regard it as a typing error , regard it as a missing tag or unparse the constructed structure.

6.1.4 Multiple output formats

At the moment it is possible to generate many different markup languages, but only one at a time. That makes it difficult to generate multiple output formats from the same source. However, in order to interact with multiple systems at the same time, multiple output formats should be supported simultaneously. That way, LaTeX output can be used for printing, HTML output for publishing on the Internet, Maple output to do calculations or generate images and C output to generate a program. At the moment, this is possible but very tedious and risky. To do it, you have to load a stencil file with the correct collection of templates, which define the required output format. After the stencil file is loaded, you can generate the required output. The main problem with this approach is that certain templates might not be defined in the loaded stencil file, so that the output can contain markup sequences from different languages. Another problem is that fact that the system only allows one definition of a template at any time and a template is only replaced if no loaded stencil file refers to it, which means that other stencil files have to be closed before the stencil with the required output definitions is loaded.

It would not be very difficult to allow multiple output definitions for each version of a template. However, the impact of the extension is large. The interface for

defining templates has to provide facilities for selecting the output formats and for defining them. The generation of output has to be modified such that the correct output format is selected, probably with some mechanism to select alternatives when definitions are missing. As the templates are stored in stencil and document files, storing the complete template with all its output formats would increase the size of document files dramatically, so it should be possible to specify which output formats have to be saved. Finally, if multiple output formats are possible, it is likely that multiple screen formats are requested as well, with all the related features such as switching between screen formats.

Once multiple output formats are available, it is possible to generate the output in several stages. For example, a LaTeX preamble output mode could be used to generate a list of all the packages and definitions that are required by the document. The generation of LaTeX output would then involve generating the LaTeX preamble and generating the LaTeX document content. For HTML, similar output stages could be defined, such as a table of contents, an index and a glossary.

If multiple screen formats are available, it would be possible to have style sheets and document related screen versions. The different formats could also be used for handling pretty printing of the document, although that would require a close interaction with the display engine.

6.1.5 Additional layout constructions

Within screen formats, several special characters are available to instruct the display engine to format the text differently. The special characters were initially only used for the tabbing environment and the stack construct. Later on, support for font changes, size changes and colour changes were added. It would not be too difficult to add support for other constructs, such as tables and images. However, the document display engine is not able to handle these extensions properly, as many of the optimisations are based on a line oriented document, which requires that the complete table has to be redrawn after every change to its content. Furthermore, the document is scrolled on a line basis instead of a pixel basis and as a table would be regarded as a single line, positioning the document properly becomes impossible for large tables.

In order to allow more layout constructions, the document display engine has to use the recursive method instead of the tree-traversal method that is currently used. As the display engine is used after every change to the document, it has to use clever caching of previous results and make use of clever screen updates, especially for plain text and tables.

The document display system was developed six years ago for computer systems at that time, which means that the memory footprint had to be small and complex layout algorithms had to be avoided. That way, it was possible to use the system on an i486 with 8MB, a 50Mhz processor and a 120MB disk. Currently, even a simple computer system is ten times more powerful in every aspect (speed, memory and disk storage). Therefore, updating to a memory or processing intensive display system should not cause performance problems.

6.1.6 LaTeX and Unicode

MathJpad allows the complete set of Unicode character to be entered, although it is not always possible to display them properly. With special input methods, it is possible to enter Chinese, Cyrillic, Greek or Korean, but it is not possible to convert all these characters to correct LaTeX markup. One of the problems is that LaTeX requires special environments and encodings for each language, which is difficult to add automatically when characters are entered freely.

The Ω system[25], a Unicode-capable extension of TeX, is able to handle Unicode encoded input, but it still requires the correct language environment in order to switch to the correct fonts and perform the correct manipulations on the document. To some extent, it is possible to add some of these tags automatically, which would allow free typing within MathJpad, but it will not always result in the correct output due to missing or incorrect fonts.

6.2 Application areas

Although the system is developed for document manipulation and preparing reports, papers or books with mathematical content, it is also possible to use the system for other purposes without too many problems.

6.2.1 Markup for dummies

The main problem with the LaTeX markup language is that you have to learn the markup sequences in order to use it properly, which takes quite some time. With MathJpad, it is possible to construct document templates, similar to the ones available in Word or WordPerfect, such that someone only has to fill in the place holders and click on the Print option to get a hard copy. All the LaTeX markup and commands to print the document are hidden from the user.

As the user does not know anything about LaTeX, all the characters that are entered into the document should appear as such in the output. That means that all special LaTeX characters generate markup sequences to produce these characters in the final output, instead of their ASCII representation, which would be interpreted by the LaTeX system.

6.2.2 Teaching and presentations

By using the scripting language, it is possible to set up a document for a slide presentation and with creative use of the template mechanism, it is even possible to create visual effects, such as text fading in or entering the page from the right or bottom. And as MathJpad handles formulæ as well as it handles text, it is possible to add lots of formulæ to the presentation. Of course, professional presentation software provides many more animations to lighten up the presentation, but such

features should be used with care, as they distract the audience from the subject. Furthermore, an application like PowerPoint is not well suited for presenting lots of formulæ.

The advantage of using a document preparation system for giving a presentation is that you can easily annotate your presentation. Comments from the audience can be added and mistakes can be corrected while the presentation is given. Furthermore, if the system provides an interface to another system, that system can be demonstrated without the need to switch to a different system (and showing all the mess on your screen). In a presentation at the Formal Methods conference in 1999[79], the connection with PVS was demonstrated in that way, which left quite an impression (if I may believe Tony Hoare).

Another application of MathʃPad might be for students to take notes during lectures. When templates and their keyboard shortcuts are installed properly, it is possible to enter formulæ very fast, which allows taking notes directly into a MathʃPad document. That way, it is easier to adjust the notes during the lecture, in case the teacher makes a mistake, to work out the notes later on or to print the notes at the end of the semester.

6.2.3 Interface for mathematical engines

The PVS interaction shows that it is possible to connect other systems to MathʃPad, although it requires some additional programming work. If the parsing routines for the output produced by the system could be generated from the template definitions, that part of the task could be reduced dramatically, making it easier to construct a connection.

The internal script language is easy extendable with new functionality through the use of dynamic libraries. By providing sufficient functionality to make connections with external systems more feasible, such as direct access to the internal document structure and several connection mechanisms, such as sockets, software busses or Corba, MathʃPad can be used as a general interface for several mathematical systems, by either using their command line interface or by using specific connection libraries such as MathLink.

If multiple output formats are available, it is even possible to have the same document processed by several mathematical systems at the same time, in order to verify results or get the fastest results. The MathʃPad document would serve as an intermediate translation format between the connected systems.

For theorem provers, the same argument holds, although it might be more difficult to use different systems on the same input. In symbolic algebra systems, the theory and automation is well established and most systems will come up with the same results and the systems use similar notations. For theorem provers, there is a wide variety of systems, formalisms and notations, from fully automated proving to step-by-step proving. Due to this possibly interactive nature of the theorem prover, it is more difficult to embed it in a different interface. Furthermore, theorem provers often do

not provide a command line interface or connection library, which makes it even more difficult.

6.2.4 Literate programming

By defining the correct template, it is possible to write programs with Mathʃpad in a publication style, such as the guarded command language. By using different output formats, it is possible to generate both the documentation and the program from the same source document. Although this will certainly work for small examples, it might not scale very well, as larger programs require more management tools, such as revision control, debugging facilities, project management and optimisation profiling. To support such tools, the Mathʃpad documents should be suitable for revision control systems, which extract changes to files to allow regeneration of all previous versions. At the moment, the document format is not well suited for that. To allow debugging, it is necessary that positions within the program text are mapped correctly to positions in the tree structure. Although possible, the current system does not support such tight integration.

To convince people that literate programming is possible with Mathʃpad, the system itself should be written using literate programming. To do that, Mathʃpad has to be bootstrapped, that is, written using itself and thus ensuring that the system is stable enough to handle large literate programs. However, the current source would have to be converted to a literate program or rewritten completely within Mathʃpad, which is quite a large effort.

6.3 Rebuilding the system

Mathʃpad was constructed with technologies and systems that were available in 1993. Since then, a lot has changed, for better or worse. Therefore, if Mathʃpad were to be reconstructed from scratch with current technology, other decisions would be made.

The current system was constructed for writing technical documents and being able to print them. Therefore, the LaTeX markup language was important as an output format as it generates high quality printed documents. For that purpose, the system works well. Due to the Internet, documents are more often distributed in an electronic form, usually in different document formats, such as the HTML and XML markup languages or PDF layout format. Therefore, the new system should include support for those document formats, as well as the technologies used on the Internet, such as hyperlinks, style sheets and document scripting.

When support for these Internet technologies is available, the question arises what the main purpose of the new system will be.

- A document preparation system
- A presentation tool for teaching technical material

- An interface for other systems
- A browser for technical contents

Each purpose requires different features. The document preparation system requires good editing facilities and dynamic screen updates. The presentation tool requires some scripting facilities and textual animations. The interface for other systems requires advanced scripting facilities and easy extendibility by providing communication protocols. Depending on its usage, the browser requires support for all XML related technology, for Java applets and for plug-ins.

As there are already several browsers available, building a new browser is not really necessary, as it might be possible to extend an existing browser like Netscape with the functionality to handle mathematical objects or non-standard notations. If the existing browser fully supports XML, it should be a matter of defining the correct XML schema and related style sheet.

The interface for other systems is useful, but as long as it is not clear which systems will be used as a background engine, the new system has to be modular and easily extendable. As the MathML and OpenMath markup languages are designed by several companies selling mathematical engines, general support for those languages should allow easy communication with several symbolic algebra systems. As MathML is an application of XML technology, general support for XML is a requirement.

The presentation tool is a logical extension of the document preparation system. Once a document is written, it can be used in a presentation immediately by using the same system to present the document. However, with some additional features, the presentation can be made more attractive to the audience. With limited scripting facilities, it is possible to perform some textual animations, such as text moving in from some direction or stepping through a list of bulleted items.

For educational purposes, a combination of a document preparation system, a presentation tool and an interface for other systems is useful. Although a browser is useful as well, it would probably take too much effort to build it from scratch. Furthermore, a document preparation system can already be used for that purpose to some extent and otherwise an existing browser could be extended with the required features for handling technical content properly.

By combining the three purposes, the resulting system is well suited for use in education:

- The document is used in the lecture to give a presentation, similar to using slides or other presentation software, such as PowerPoint.
- The students can use their own copy of the document to take notes, either during the lecture or afterwards, depending on whether they are allowed to use computers during lectures.
- The document can contain exercises for the students, which can be carried out using the system. Afterwards, the student can give a presentation using the same system.

- If a connection to an external system is available, some of the exercises can be carried out or marked using the external system.
- The document can contain additional information for students who are unable to gasp the standard information. By using hyperlinks, this addition information might be located anywhere on the Internet.

Several of the features mentioned are already used in the interactive algebra course[17] developed by Cohen in Eindhoven. This course uses HTML documents and Java applets to present the algebraic theory, examples, exercises and background information. With the use of special Java applets, called gapplets, a connection with the GAP system[41] can be used for doing calculations. One possible problem with the interactive course is that it is not yet possible to add notes to the individual pages, such that personal comments or corrections can be added. However, this problem is inherent in the usage of the CD-ROM storage medium and the limitations of Internet browsers. Nevertheless, the interactive algebra course is impressive and it takes a lot of time and effort to create such an interactive course. As there is no specialised software support for constructing such a course, a powerful mathematical document preparation system might be useful for preparing similar courses.

There are several advantages for using electronic documents during lectures, especially if they can be adjusted easily.

- It is easy to highlight parts of the document by using the selection or scripting facilities.
- With hyperlinks, it is easy to refer to other documents or parts of the same document, which allows the lecturer to easily skip parts of the lecture which are clear and give more information on the parts which are not clear.
- The lecturer can annotate the presentation with remarks from students.
- It is possible to have reliable dynamics, such as showing the steps in a proof.
- With scripting facilities, it is possible to replay the presentation, which allows students to repeat the presentation.
- If the document contains sufficient information, it is possible to follow lectures from a distance. Otherwise, technologies such as RealAudio or WebCam can be used to transmit the additional information provided by the lecturer.

For this to work, the system has to provide presentation specific features, such as animations, highlighting, hyperlinking, script recording and script playback.

During a presentation, the keyboard and mouse are often not an appropriate interface to the computer, as it requires the lecturer to be very close to the computer, often sitting down behind a desk and being obscured by equipment. Therefore, other input methods might be useful, such as sound recognition using a microphone, 3D gestures using a 3D mouse or data glove or written gestures using a PalmPilot. These alternative input methods are also useful for normal interaction with the system, as they might reduce the chance of RSI.

6.3.1 New technology

The new system can make use of new technology, which wasn't either available, widely used or mature enough when the current system was built. Some of the technologies which might be useful are:

XML The XML markup language and related technology[47] seems to be the new standard for storing structured documents and processing them. Although SGML, the basis for XML, has been around for a long time, it was considered to be too complex to be useful. The simplifications in XML and the success of HTML make the technology more accessible and more tools have become available for handling XML documents. As XML originated from the WWW consortium, it is probably a format that will be used for quite some time, although it is difficult to predict how it will develop in the future. Therefore, using XML as a basis for the revised system would be a good choice, as other XML tools can be used for processing documents. The related technology includes:

- **XSL** The extensible style language used for formatting XML documents
- **XSL transformations** The transformation system for XSL to generate other formats from XML documents.
- **MathML** An XML schema for adding mathematical content to XML documents.
- **DOM** The document object model that allows easy access to the content of an XML document through a standard interface.
- **SAX** The simple API for XML that allows easy interaction with existing XML parsers.

Although the XML markup language is quite different, it should still be possible to generate sensible LaTeX output for printing purposes.

Corba With Corba technology[48], it is possible to build distributed systems, such that documents can be manipulated by multiple authors, for example, several students working on a common project. However, to be able to use this technology, the internal structure has to support network transparent storage and editing operations, as part of the document may be stored and edited on another system.

Internet Browsers By using XML as the native storage format of the new system, normal Internet browsers like Netscape or Internet Explorer can be used to view the documents, assuming that the support for XML and XSL will be available in those systems. It might be necessary to provide additional functionality in the form of plug-ins or extensions in order to handle certain mathematical notations properly.

OpenMath The OpenMath society has defined a standard for representing mathematical formulæ and their semantics[9], such that they can easily be exchanged

between different systems. By supporting the OpenMath standard (and the related MathML recommendation), interaction with mathematical systems could be established. With the use of content dictionary, it is also possible to represent programs in the OpenMath standard, thereby allowing interaction with programming related systems like interpreters, compilers and debuggers.

Scripting There are several scripting languages which can be used within documents, such as Javascript, ECMAScript[20], VBscript and Tcl/TK[54]. With these scripting languages, it is possible to make the content of documents more interactive and responsive to actions of the user.

By using XML as a basis for the new system, the structure of the internal workings are determined to some extent. The Corba support will complicate the internal workings as network transparency has to be added in the form of conversion methods to access the ORB. The OpenMath support can be accomplished by providing conversion methods between the internal representation and the OpenMath representation, where the Corba support can be used for the communication with other systems.

The use of XML as basis for documents means that the structure of documents is more rigid than the structure that Mathʃpad currently uses, as XML documents have to comply to the structure described in their XML schema. Therefore, the new system has to provide sufficient features to ensure that this structural integrity doesn't become a hurdle in using the system. Similar requirements appear when support for OpenMath is added, as it adds semantical meaning to the formulæ, which has to be correct as well.

Although it is possible to build some the described features and technology into the current Mathʃpad system, the internal storage structure with its selection and display features make it very difficult to support these extensions properly.

Bibliography

[1] Adobe. *Adobe Photoshop version 4.0*. Adobe Press, 1997.

[2] Adobe Systems Incorporated. *PostScript Language Reference Manual*. Addison-Wesley Publishing Company, 1990.

[3] Inc. Apple Computer, editor. *OpenDoc Programmer's Guide for the MacOS*. Addison-Wesley, 1995.

[4] J.A. Bergstra and P. Klint. The discrete time toolbus. Technical Report P9502, Programming Research Group, University van Amsterdam, 1995.

[5] Don Bolinger and Tan Bronson. *Applying RCS and SCCS*. O'Reilly, September 1995.

[6] Bert Bos, Håkon Wium Lie, Chris Lilley, and Ian Jacobs. Cascading style sheets, level 2. Available online, at `http://www.w3.org/TR/REC-CSS2/`, May 1998.

[7] Arald den Braber and Frits Lucas. Mathpad for Windows95. Master's thesis, Eindhoven University of Technology, 1998.

[8] Tim Bray, Jean Paoli, and C.M. Sperberg-McQueen. Extensible markup language (XML) 1.0. Available online, at `http://www.w3.org/XML/`, February 1998.

[9] O. Caprotti, D.P. Carlisle, and A.M. Cohen. The OpenMath standard. Available online, at `http://www.openmath.org/`, August 1999.

[10] Per Cederqvist et al. *Version Management with CVS*, 1993. Available online, at `http://www.cyclic.com/`.

[11] Robert J. Chassell and Richard M. Stallman. *Texinfo, the GNU Documentation Format*, 3.12 edition, February 1998.

[12] Chicago Press, editor. *The Chicago Manual of Style*. The University of Chicago Press, 14th edition, 1993.

[13] P. Chisholm. Calculation by computer. In *Third International Workshop Software Engineering and its Applications*, pages 713–728, Toulouse, France, December 1990. EC2.

[14] Ian Clatworthy. *SDF User Guide*, May 1999. Available online, at `http://www.mincom.com/mtr/sdf/`.

[15] Englewood Cliffs. *OSF/Motif : programmer's reference. - Release 1.1.* Prentice Hall, 1992.

[16] Cobb Group. *Word 6 for Windows companion.* Microsoft Press, 3rd edition, 1994.

[17] Arjeh Cohen, Hans Cuypers, and Hans Sterk. *Algebra Interactive.* Springer, 1999.

[18] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, Berlin, 1990.

[19] ECMA. User interface taxonomy. Technical Report ECMA TR/61, European Computer Manufacturers Association (ECMA), 1992.

[20] ECMA. ECMAScript language specification - 3rd edition. Technical Report ECMA-262, European Computer Manufactures Association (ECMA), December 1999. Available online, at `http://www.ecma.ch/stand/ECMA-262.htm`.

[21] Ian Elliott and David P. Wiggins. Double buffer extension protocol. Technical report, X Consortium, Inc., 1996.

[22] Sandra L. Emerson and Karen Paulsell. *Troff typesetting for UNIX systems.* Prentice-Hall, 1987.

[23] Benno Fuchssteiner and Klaus Gottheil et al. *MuPAD : Multi Processing Algebra Data Tool : Tutorial, MuPad version 1.2.* John Wiley and sons, Chichester, New York, 1994. See also `http://www.mupad.de/`.

[24] *The Gimp User's Manual*, 1999. Available online at `http://manual.gimp.org`.

[25] Yannis Haralambous and John Plaice. Multilingual typesetting with Omega, a case study: Arabic. In *Proceedings of the International Symposium on Multilingual Information Processing*, pages 63–80, 1997. Available online at `http://www.gutenberg.eu.org/omega/`.

[26] Dan Heller. *XView Programming Manual.* O'Reilly & Associates, Inc., 1990.

[27] G. Huet, G. Kahn, and C. Paulin-Mohring. The Coq proof assistant – a tutorial. Technical Report 178, INRIA, July 1995.

[28] Patrick Ion and Robert Miner. Mathematical markup language (MathML) 1.0 specification. Available online, at `http://www.w3.org/Math/`, April 1998.

[29] Ian Jacobs and Laurence Rideau-Gallot. A Centaur tutorial. Technical Report 140, INRIA Sophia-Antipolis, July 1992.

[30] Norbert Kajler. CAS/PI: a portable and extensible interface for computer algebra systems. In *Proceedings of ISSAC'92*, pages 376–386. ACM Press, July 1992.

[31] Brian W. Kernighan and Lorinda L. Cherry. A system for typesetting mathematics. *Communications of the ACM*, 18:182–193, 1975.

[32] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, 1988.

[33] Jeffrey H. Kingston. *A User's Guide to the Lout Document Formatting System*. Basser Department of Computer Science, The University of Sydney 2006, Australia, August 196.

[34] P. Klint. *The ASF+SDF Meta-environment User's Guide*, June 1995.

[35] D.E. Knuth. Literate programming. *Computer Journal*, 27(2):97–111, 1984.

[36] Donald E. Knuth. *The TEXbook*. Addison-Wesley Publishing Company, 1986.

[37] Shiz Kobara. *Visual design with OSF/Motif*. Addison-Wesley, 1991.

[38] Doug Kramer. How to write doc comments for javadoc. Available online, at `http://java.sun.com/products/jdk/javadoc/`.

[39] Leslie Lamport. *LATEX: A Document Preparation System*. Addison-Wesley Publishing Company, 1994.

[40] Mark Leisher. Input method design. A tutorial at the Ninth International Unicode Conference, September 1996.

[41] Steve Linton. The GAP 4.2 reference manual. Available online, at `http://www-history.mcs.st-and.ac.uk/~gap/`, February 2000.

[42] Monique H. Logger. An integrated text and syntax-directed editor. Technical Report CS-R8820, Centrum voor Wiskunde en Informatica, 1988.

[43] Jason J. Manger. *Netscape Navigator*. McGraw-Hill, 1995. See also `http://www.netscape.com/`.

[44] MathType, the best thing for writing equations since chalk! Available online, at `http://www.mathtype.com/`.

[45] The MathWorks, Inc. *Using MATLAB*, December 1996. Available online, at `http://www.mathworks.com/`.

[46] Microsoft, editor. *OLE2 Programmer's Reference*, volume one. Microsoft Press, 1994.

[47] Michael Morrison et al. *XML Unleashed*. Sams Publishing, December 1999.

[48] Thomas J. Mowbray and William A. Ruh. *Inside CORBA : distributed object standards and applications*. Addison-Wesley, 1997.

[49] Patrick Naughton. *The Java Handbook*. Osborne McGraw-Hill, 1996.

[50] Adrian Nye. *Xlib reference manual*. O'Reilly, 3rd edition, 1992.

[51] Adrian Nye. *Xlib programming manual*. O'Reilly, 3rd edition, 1993.

[52] Adrian Nye and Tim O'Reilly. *X Toolkit intrinsics programming manual*. Addison-Wesley, 3rd edition, 1993.

[53] Sandra Martin O'Donnell. *Programming for the World: A Guide to Internationalization*. Prentice Hall, 1994.

[54] J.K. Ousterhout. An X11 toolkit based on the TCL language. In *Proceedings of the 1991 Winter USENIX Conference*, pages 105–115, 1991.

[55] Lawrence C. Paulson. *Isabelle: a Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer – Berlin, 1994.

[56] Eric Peeters. *Design of an Object-Oriented, Interactive Animation System*. PhD thesis, Eindhoven University of Technology, December 1995.

[57] V. Quint and I. Vatton. Grif: an interactive system for structured document manipulation. In J.C. van Vliet, editor, *Text Processing and Document Manipulation, Proceedings of the International Conference*, pages 200–213. Cambridge University Press, 1986.

[58] Dave Raggett and Ian Jacobs. Hypertext markup language home page. Available online at `http://www.w3.org/`, 1999.

[59] Dave Raggett, Jenny Lam, and Ian Alexander. *HTML 3, Electronic Publishing on the World Wide Web*. Addison-Wesley, 1996.

[60] Darren Redfern. *The Maple Handbook*. Springer, 1996.

[61] Brian K. Reid. A high-level approach to computer document production. In *Proceedings of the 7th Symposium on the Principles of Programming Languages (POPL)*, pages 24–31, 1980.

[62] Martin Reiser. *The Oberon system : user guide and programmer's manual*. Addison-Wesley, 1991.

[63] Thomas W. Reps and Tim Teitelbaum. *The synthesizer generator : a system for constructing language-based editors*. Springer, 1989.

[64] Wolfram Research. *MathLink reference guide : Mathematica version 2.2*. Wolfram Research, 1993. See also `http://www.wolfram.com/`.

[65] John Rushby. The pvs specification and verification system. Available online at `http://pvs.csl.sri.com/`, November 1998. Contains many references to information on PVS.

[66] Ben Shneiderman. *Designing the User Interface*. Addison Wesley Longman, Inc., 3rd edition, 1998.

[67] Richard Stallman. *GNU Emacs Manual*. Free Software Foundation, 9th edition, August 1993.

[68] JTC 1 subcommittee. Information processing – text and office systems – standard generalized markup language (SGML). Technical Report ISO 8879:1986, International Organisation of Standardisation (ISO), 1986.

[69] JTC 1 subcommittee. Information technology – universal multiple-octet coded character set (UCS) – part 1: Architecture and basic multilingual plane. Technical Report ISO/IEC 10646-1:1993, International Organisation of Standardisation (ISO), 1993.

[70] JTC 1 subcommittee. Information technology – character code structure and extension techniques. Technical Report ISO/IEC 2022:1994, International Organisation of Standardisation (ISO), 1994.

[71] JTC 1 subcommittee. Information technology – processing languages – document style semantics and specification language (DSSSL). Technical Report ISO/IEC 10179:1996, International Organisation of Standardisation (ISO), 1996.

[72] JTC 1 subcommittee. Information technology – input methods to enter characters from the repertoire of ISO/IEC 10646 with a keyboard or other input device. Technical Report ISO/IEC 14755:1997, International Organisation of Standardisation (ISO), 1997.

[73] Bernard Sufrin and Richard Bornat. User interfaces for generic proof assistants part II: Displaying proofs. In R.C. Backhouse, editor, *Workshop on User Interfaces for Theorem Provers*, Computing Science Reports, pages 147–156, July 1998. See also: `http://www.win.tue.nl/~martijno/uitp/papers/Sufrin.ps.gz`.

[74] Inc. Sun Microsystems. *OPEN LOOK Graphical User Interface Functional Specification*. Addison Wesley, 1990.

[75] Sun Microsystems Company. *ToolTalk 1.0 Programmer's Guide*, December 1991.

[76] Microsoft Technical Support. Rich text format (RTF) specification and sample RTF reader program. Technical report, Microsoft, May 1997. Available online, at `http://www.wotsit.org/`.

[77] Unicode Consortium. *The Unicode Standard : Version 2.0*. Addison-Wesley, 1996.

[78] Matteo Vaccari. *Calculational Derivation of Circuits*. PhD thesis, Dipartimento di Informatica, Università degli Studi di Milano, May 1998. See also: `http://gongolo.usr.dsi.unimi.it/~matteo/tesi.ps.gz`.

[79] Richard Verhoeven and Roland Backhouse. Interfacing program construction and verification. In J.M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 – Formal Methods*, volume 1709 (II) of *LNCS*, pages 1128–1146, Toulouse, September 1999. Springer.

[80] Richard Verhoeven and Olaf Weber. Mathpad : implementatie van een formule editor. Master's thesis, Eindhoven University of Technology, 1992.

[81] W3c – the world wide web consortium. Available online, at `http://www.w3.org/`, 1999.

[82] Larry Wall. *perl - Practical Extraction and Report Language*. Available online, at `http://www.cpan.org/`.

[83] Stephen Wolfram. *The Mathematica Book*. Cambridge University Press, 3rd edition, 1996.

[84] Allen L. Wyatt, Steve Dyson, and Daniel J. Fingerman. *WordPerfect 6 for Windows : the complete reference*. Osborne, 1994.

Index

Samenvatting

Iedereen heeft wel eens een briefje geschreven en weet uit ervaring hoe vervelend het is als je fouten maakt. Als je het met de hand schrijft, moet je opnieuw beginnen en als je het op een oude typemachine typt, moet je met typex gaan knoeien. Met een computer en tekstverwerker heb je het voordeel dat het resultaat nog niet op papier staat en dat je door de computer geholpen wordt om fouten te vinden en te verbeteren. Voor veel brieven en andere normale teksten werkt dit fantastisch en het is zelfs mogelijk om je tekst te versieren met allerlei plaatjes.

Voor technische teksten, zoals wiskunde boeken en wetenschappelijke rapporten, is het allemaal niet zo rooskleurig. De auteur moet de complexe materie goed begrijpen om de tekst uit te kunnen werken. Meestal moet hij dat zelf doen omdat het erg omslachtig is om iemand anders duidelijk te maken wat precies de bedoeling is. Bij het maken van fouten wordt de auteur vaak niet geholpen, omdat de tekstverwerker de technische beschrijvingen en formules niet kan controleren. Die controle wordt meestal uitgevoerd door verschillende deskundigen, die de tekst lezen.

Ook het maken van het tekst zelf is niet zo geavanceerd. In plaats van te werken met een "wat-ik-zie-is-wat-ik-krijg" (WIZIWIK) scherm, wordt er een cryptische taal (meestal LaTeX) gebruikt om de tekst te beschrijven. Deze cryptische taal heeft als voordeel dat de uiteindelijke kwaliteit uitstekend is, vooral voor technische teksten, maar er zijn een paar nadelen. Voor eenvoudige tekst is de cryptische taal nog goed te lezen, maar naarmate er meer formules verschijnen wordt alles minder leesbaar. Verder komen in technische teksten vaak formules voor die weinig van elkaar verschillen, zodat in de cryptische tekst alles op elkaar begint te lijken en fouten moeilijk zijn te vinden en te herstellen.

Het Mathſpad systeem is ontworpen om deze problemen gedeeltelijk op te lossen. Mathſpad wil:

- de leesbaarheid van de technische tekst verbeteren
- het werken met formules vereenvoudigen
- de teksten eenvoudiger controleren op correctheid.

De leesbaarheid van technische teksten verbeteren

De auteur werkt rechtstreeks met leesbare formules die bijna WIZIWIK op het scherm staan, terwijl de cryptische taal wordt gebruikt om de uitstekende kwaliteit te waarborgen. Aangezien verschillende auteurs met verschillende notaties werken, is het systeem zo ontworpen dat de auteur gemakkelijk nieuwe notaties kan definiëren. Het systeem is daardoor niet beperkt tot een kleine groep.

Het werken met formules vereenvoudigen

Om het werken met formules te vereenvoudigen, worden ze opgeslagen als bomen. Net als bij echt bomen is het eenvoudiger om takken (deelformules) te selecteren dan de bijbehorende bladeren (letters). De auteur hoeft een formule nu niet langer letter voor letter te schrijven, maar kan een aantal deelformules gebruiken. Voor formules is dit zeer belangrijk, want een ontbrekende letter betekent vaak een foute formule. Helaas zijn er nogal wat verschillen tussen het werken met deelformules en het werken met letters, zodat de auteur enige tijd nodig heeft om aan het idee te wennen en de mogelijkheden volledig te benutten.

Binnen de wiskunde en technische vakken is het gebruik van speciale symbolen zeer normaal. Zo heeft elk lettertype een bepaalde betekenis en bij uitputting van de gewone letters wordt er al snel gebruik gemaakt van de Griekse en Hebreeuwse letters. Hetzelfde geldt voor de operatoren. Bij standaard rekenen heb je voldoende aan $+$, $-$, $\times$, $/$ en $=$, maar een behendig wiskundige varieert en combineert al naar gelang de situatie er om vraagt. Zo betekenen de operatoren $<$, $\prec$, $\subset$, $\sqsubset$, en $\lhd$ vaak dat iets kleiner is ten opzichte van iets anders, maar het is niet toegestaan het verkeerde symbool te gebruiken. Om de auteur voldoende keus te geven, geeft het Mathʃpad systeem de gebruiker toegang tot een zeer grote collectie symbolen (Unicode) en een aantal methoden om de symbolen te variëren en combineren. En voor het geval de Griekse en Hebreeuwse letters op zijn, zijn onder andere ook de Russische letters en Chinese ideogrammen beschikbaar.

De teksten eenvoudiger controleren op correctheid

Met dit alles is het mogelijk om de technische teksten te maken, maar is het niet mogelijk om de teksten op correctheid te controleren. Daarom is het Mathʃpad systeem uitgebreid met een eigen taal om nieuwe onderdelen eenvoudig toe te kunnen voegen. Via deze taal kan het Mathʃpad systeem worden gecombineerd met bestaande systemen. Op die manier kan elk onderdeel van de tekst worden gecontroleerd door het bijbehorende systeem, aangenomen dat zo'n systeem bestaat en kan samenwerken met Mathʃpad.

Curriculum Vitae

Petrus Hendricus Franciscus Maria Verhoeven (Richard)

9 februari 1969	Geboren te Veghel
juli 1987	Diploma VWO Mgr. Zwijsen College, Veghel
december 1992	Doctoraal examen Technische Informatica Technische Universiteit Eindhoven
1993 tot 1998	Wetenschappelijk programmeur Faculteit der Wiskunde en Informatica Technische Universiteit Eindhoven
1998 tot 2000	Onderzoeksmedewerker Nederlandse Organisatie voor Wetenschappelijk Onderzoek verbonden aan de Technische Universiteit Eindhoven
januari tot juni 2000	Research Assistant School of Computer Science & Information Technology University of Nottingham

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1996-1

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms*. Faculty of Mathematics and Computer Science, KUN. 1996-2

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-3

M.G.A. Verhoeven. *Parallel Local Search*. Faculty of Mathematics and Computing Science, TUE. 1996-4

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory*. Faculty of Mathematics and Computer Science, KUN. 1996-5

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems*. Faculty of Mathematics and Computing Science, TUE. 1996-6

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance*. Faculty of Mathematics and Computer Science, UvA. 1996-7

H. Doornbos. *Reductivity Arguments and Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1996-8

D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-9

A.M.G. Peeters. *Single-Rail Handshake Circuits*. Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism*. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference*. Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics*. Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth*. Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context*. Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types*. Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics*. Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution*. Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing*. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing*. Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering*. Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation*. Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller*. Faculty of Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model*. Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing*. Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes*. Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search*. Faculty of Mathematics and Natural Sciences, Univ. Leiden. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection*. Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Schedulere Optimization in Real-Time Distributed Databases*. Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics*. Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems*. Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems*. Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules*. Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System*. Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars*. Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Progam Construction*. Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic*. Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms*. Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols*. Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor*. Faculty of Mathematics and Computing Science, TUE. 2000-05