

The Design of a User Interface  
for  
Computer Algebra Systems

Copyright ©1991

by

Neil Morrell Soiffer

# The Design of a User Interface for Computer Algebra Systems

Neil Morrell Soiffer

Computer Science Division  
University of California  
Berkeley, CA. 94720

## Abstract

This thesis discusses the design and implementation of natural user interfaces for Computer Algebra Systems. Such an interface must not only display expressions generated by the Computer Algebra System in standard mathematical notation, but must also allow easy manipulation and entry of expressions in that notation. The user interface should also assist in the understanding of large expressions that are generated by Computer Algebra Systems and should be able to accommodate new notational forms.

Building a good user interface requires many tradeoffs. As such, there may be no "best" solution for any individual subtask. This thesis discusses many of the tradeoffs involved and develops algorithms for displaying, entering, and selecting from expressions. Two new algorithms for incremental parsing of input are presented. Either algorithm can be coupled with the incremental reformatting and screen update algorithm developed in this thesis to produce an interface capable of correctly updating and displaying changes to relatively large expressions in real-time. Moreover, algorithms that exploit locality for the selection of subexpressions are also given. These selection algorithms are also capable of providing real-time feedback as to what is being selected.

Because Computer Algebra Systems generate large expressions, efficiency in *both* time and space are important. Data is presented that shows that using directed acyclic graphs as the fundamental underlying data structure instead of trees *significantly* decreases

time and space usage. The ability to display large expressions efficiently must be coupled with the ability to manipulate them into understandable forms. This thesis discusses three such techniques: elision, renaming of subexpressions, and line breaking. Two new algorithms are given for incrementally reformatting expressions in the presence of line breaks.

This thesis introduces the idea of *translations*. Translations convert between the form used by a Computer Algebra System and the mathematical notation displayed on the screen. Translations increase the portability of the interface and allow it to communicate with different Computer Algebra Systems during a single session. Translations can be grouped together to form *notation libraries* which can be used to eliminate some of the ambiguity inherent in mathematical notation.

# Contents

|   |            |
|---|------------|
| <b>Abstract</b>                                       | <b>i</b>   |
| <b>Table of Contents</b>                              | <b>iii</b> |
| <b>List of Figures</b>                                | <b>vi</b>  |
| <b>List of Tables</b>                                 | <b>ix</b>  |
| <b>Acknowledgements</b>                               | <b>x</b>   |
| <b>1 Introduction</b>                                 | <b>1</b>   |
| 1.1 Computer Algebra Systems . . . . .                | 2          |
| 1.2 Related Interface Work . . . . .                  | 7          |
| 1.2.1 CAS Interfaces . . . . .                        | 8          |
| 1.2.2 Numerical Interfaces . . . . .                  | 14         |
| 1.2.3 Document Processing Systems . . . . .           | 16         |
| 1.3 The Problem . . . . .                             | 17         |
| 1.3.1 Entering Expressions . . . . .                  | 18         |
| 1.3.2 Selecting and Editing Expressions . . . . .     | 19         |
| 1.3.3 Formatting and Displaying Expressions . . . . . | 20         |
| 1.3.4 Ambiguous Notation . . . . .                    | 22         |
| 1.3.5 Session Layout . . . . .                        | 22         |
| 1.3.6 Portability to Different CASs . . . . .         | 23         |
| 1.4 Roadmap to this Thesis . . . . .                  | 23         |
| <b>2 MathScribe</b>                                   | <b>27</b>  |
| 2.1 Description of MathScribe . . . . .               | 27         |
| 2.2 Overall Architecture . . . . .                    | 37         |
| 2.3 Implementation problems . . . . .                 | 39         |
| 2.4 History . . . . .                                 | 40         |
| <b>3 Translations</b>                                 | <b>42</b>  |
| 3.1 The Problem . . . . .                             | 43         |
| 3.1.1 Communication . . . . .                         | 44         |

|          |  |            |
|----------|--|------------|
| 3.1.2    | Translation Order . . . . .                                | 47         |
| 3.1.3    | Ambiguity and Notation Libraries . . . . .                 | 47         |
| 3.1.4    | Pattern Classification . . . . .                           | 49         |
| 3.2      | MathScribe and Other Systems . . . . .                     | 54         |
| 3.2.1    | CaminoReal . . . . .                                       | 55         |
| 3.2.2    | MathStation . . . . .                                      | 56         |
| 3.2.3    | MathScribe . . . . .                                       | 58         |
| 3.3      | Parser-based Translations . . . . .                        | 61         |
| 3.4      | Procedure-based Translations . . . . .                     | 65         |
| 3.5      | Rule-based Translations . . . . .                          | 66         |
| 3.6      | Summary . . . . .  | 72         |
| <b>4</b> | <b>Formatting and Drawing</b> . . . . .                    | <b>74</b>  |
| 4.1      | Boxes and Drawing . . . . .                                | 75         |
| 4.1.1    | Bounding Boxes . . . . .                                   | 75         |
| 4.1.2    | Drawing . . . . .  | 76         |
| 4.1.3    | Incremental Update . . . . .                               | 78         |
| 4.2      | Formatting . . . . .                                       | 81         |
| 4.2.1    | Primitive-based Formatting . . . . .                       | 82         |
| 4.2.2    | Procedure-based Formatting . . . . .                       | 88         |
| 4.2.3    | Macro-based Formatting . . . . .                           | 89         |
| 4.2.4    | Constraint-based Formatting . . . . .                      | 90         |
| 4.3      | Large Expressions . . . . .                                | 92         |
| 4.3.1    | Line Breaking . . . . .                                    | 92         |
| 4.3.2    | Elision . . . . .  | 99         |
| 4.3.3    | Renaming . . . . .   | 101        |
| 4.3.4    | Horizontal Scrolling . . . . .                             | 102        |
| 4.3.5    | Line Cutting . . . . .                                     | 102        |
| 4.3.6    | Structural Line Breaking . . . . .                         | 105        |
| <b>5</b> | <b>Entering Expressions</b> . . . . .                      | <b>107</b> |
| 5.1      | Templates . . . . .  | 110        |
| 5.2      | Overlays . . . . .   | 112        |
| 5.3      | Overlays with Precedence . . . . .                         | 115        |
| 5.4      | Parsing . . . . .  | 128        |
| 5.4.1    | Some Parsing-Related User Interface Issues . . . . .       | 128        |
| 5.4.2    | Error Recovery and Correction in a CAS Interface . . . . . | 130        |
| 5.4.3    | Incremental LR Parsing . . . . .                           | 133        |
| 5.4.4    | Attribute Grammars . . . . .                               | 136        |
| 5.5      | The MathScribe Parser . . . . .                            | 137        |
| 5.5.1    | An Extended Operator Precedence Grammar . . . . .          | 138        |
| 5.5.2    | Incremental Extended Operator Precedence Parsing . . . . . | 139        |
| 5.5.3    | Overloading Delimiters . . . . .                           | 145        |
| 5.5.4    | Error Recovery, Correction, and Tolerance . . . . .        | 147        |
| 5.6      | Other Forms of Input . . . . .                             | 149        |

|          |  |            |
|----------|--|------------|
| <b>6</b> | <b>Selection</b>   | <b>151</b> |
| 6.1      | Three Methods for Structural Selection . . . . .           | 152        |
| 6.2      | Non-Structural Selection and Multiple Selections . . . . . | 158        |
| 6.3      | Selection and Keyboard Input . . . . .                     | 161        |
| <b>7</b> | <b>Sharing and other Optimizations</b>                     | <b>164</b> |
| 7.1      | Sharing . . . . .  | 164        |
| 7.2      | Drawing Speedups . . . . .                                 | 174        |
| 7.2.1    | Avoiding Character Drawing . . . . .                       | 174        |
| 7.2.2    | Caching the Draws . . . . .                                | 175        |
| 7.2.3    | Partial Formatting . . . . .                               | 176        |
| 7.3      | Storage Management . . . . .                               | 178        |
| <b>8</b> | <b>Conclusions</b>   | <b>180</b> |
| 8.1      | Summary . . . . .  | 180        |
| 8.2      | Future Work . . . . .                                      | 182        |
| 8.2.1    | Direct Manipulation . . . . .                              | 182        |
| 8.2.2    | Large Expressions . . . . .                                | 182        |
| 8.2.3    | Session Layout . . . . .                                   | 183        |
| 8.2.4    | Help Systems . . . . .                                     | 184        |
| 8.2.5    | Graphics, Numerics, and Statistics . . . . .               | 184        |
| 8.2.6    | Avoiding Recomputation . . . . .                           | 187        |
| 8.3      | Conclusions . . . . .                                      | 187        |
|          | <b>Bibliography</b>  | <b>189</b> |
| <b>A</b> | <b>Translation Example</b>                                 | <b>205</b> |
| A.1      | Forward Translations . . . . .                             | 206        |
| A.2      | Backward Translations . . . . .                            | 211        |
| <b>B</b> | <b>DAG Algorithms</b>                                      | <b>214</b> |
| B.1      | Data Structures . . . . .                                  | 214        |
| B.2      | Algorithms . . . . .                                       | 217        |
| <b>C</b> | <b>Sharing Data</b>  | <b>220</b> |
| C.1      | Sharing Data . . . . .                                     | 220        |
| C.2      | Memory Usage Under Differing Sharing Scenarios . . . . .   | 231        |

# List of Figures

|      |   |    |
|------|---|----|
| 1.1  | Sample CAS Calculations . . . . .                           | 3  |
| 1.2  | Sample Output . . . . .                                     | 9  |
| 1.3  | Overall Architecture of CAS/Interface . . . . .             | 24 |
| 2.1  | Sample MathScribe Session . . . . .                         | 28 |
| 2.2  | MathScribe Control Panel . . . . .                          | 29 |
| 2.3  | Entering $\frac{a}{b+c}$ Using a Keyboard . . . . .         | 31 |
| 2.4  | Entering $\frac{a+b}{c}$ Using Overlays . . . . .           | 32 |
| 2.5  | Two MathScribe Form Menus . . . . .                         | 32 |
| 2.6  | An Example of Collapsing and Elision . . . . .              | 33 |
| 2.7  | An Example of Renaming . . . . .                            | 34 |
| 2.8  | Abbreviation Window . . . . .                               | 35 |
| 2.9  | Plotting Examples . . . . .                                 | 36 |
| 2.10 | Spreadsheet Example . . . . .                               | 37 |
| 2.11 | MathScribe Architecture . . . . .                           | 38 |
| 3.1  | Forward Translation Examples . . . . .                      | 44 |
| 3.2  | Patterns in Mathematica . . . . .                           | 44 |
| 3.3  | Centralized Architecture of Translations . . . . .          | 45 |
| 3.4  | Notation Libraries for Logical Operations . . . . .         | 48 |
| 3.5  | Idempotent Forward Translation Pattern Examples . . . . .   | 50 |
| 3.6  | Rewrite-once Forward Translation Pattern Examples . . . . . | 52 |
| 3.7  | Architecture of CaminoReal . . . . .                        | 56 |
| 3.8  | Forward Translation Algorithm . . . . .                     | 59 |
| 3.9  | Backward Translation Algorithm . . . . .                    | 60 |
| 3.10 | Example BNF Grammar . . . . .                               | 62 |
| 4.1  | Attributes of a Box . . . . .                               | 75 |
| 4.2  | Drawing Algorithms . . . . .                                | 77 |
| 4.3  | Incremental Formatting Update Algorithm . . . . .           | 80 |
| 4.4  | Formatting Functions in Mathematica . . . . .               | 82 |
| 4.5  | WYSIWYG Formatting Example . . . . .                        | 88 |
| 4.6  | A MathScribe Formatting Procedure . . . . .                 | 89 |

|      |  |     |
|------|--|-----|
| 4.7  | T <sub>E</sub> X Atoms . . . . .                                       | 91  |
| 4.8  | Line Breaking in Mathematica . . . . .                                 | 95  |
| 4.9  | Line Breaking in MACSYMA . . . . .                                     | 95  |
| 4.10 | Line Breaking in Reduce . . . . .                                      | 96  |
| 4.11 | Line Breaking in Maple . . . . .                                       | 96  |
| 4.12 | Indentation . . . . .  | 98  |
| 4.13 | Modified Drawing Algorithm for Line Breaking . . . . .                 | 100 |
| 4.14 | Modified Drawing Algorithm for Line Cutting . . . . .                  | 104 |
| 4.15 | Structural Line Breaking . . . . .                                     | 105 |
|      |  |     |
| 5.1  | Entering an expression using templates . . . . .                       | 111 |
| 5.2  | Example: applying the plus overlay . . . . .                           | 112 |
| 5.3  | Entering an expression using overlays . . . . .                        | 112 |
| 5.4  | Entering an expression using a parser . . . . .                        | 116 |
| 5.5  | Example: floating the plus overlay . . . . .                           | 117 |
| 5.6  | Left and right precedence examples . . . . .                           | 118 |
| 5.7  | Infix rotate left and rotate right overlays . . . . .                  | 119 |
| 5.8  | Chained infix rotate left and right . . . . .                          | 120 |
| 5.9  | Prefix Rotate . . . . .  | 121 |
| 5.10 | Postfix Rotate . . . . .   | 122 |
| 5.11 | Matching Parentheses . . . . .   | 125 |
| 5.12 | Example: Keyboard Input of if-then Construct . . . . .                 | 126 |
| 5.13 | Conversion of infix to prefix for delete . . . . .                     | 127 |
| 5.14 | Incremental Parser for LR(1) Grammar . . . . .                         | 135 |
| 5.15 | Example of <code>Split(H, left, right)</code> . . . . .                | 136 |
| 5.16 | A Non-incremental parsing algorithm for MathScribe's Grammar . . . . . | 140 |
| 5.17 | Parser actions ( <code>chooseAction</code> ) . . . . .                 | 142 |
| 5.18 | Delimiter Chain Example for Some Overloaded Delimiters . . . . .       | 143 |
| 5.19 | The Reduce algorithm . . . . .   | 144 |
| 5.20 | Examples of overloaded delimiters . . . . .                            | 146 |
|      |  |     |
| 6.1  | Three Alternative Methods of Selection . . . . .                       | 153 |
| 6.2  | Incremental Selection Algorithm for Point method . . . . .             | 156 |
| 6.3  | Selection Algorithm for Internal Rectangle method . . . . .            | 157 |
| 6.4  | Selection Algorithm for External Rectangle method . . . . .            | 159 |
| 6.5  | Selection by Circling . . . . .  | 159 |
| 6.6  | An Expression drawn at two Different Resolutions . . . . .             | 163 |
|      |  |     |
| 7.1  | Shared Boxes for $\cos(x) \cdot e^x + \sin(x) \cdot e^x$ . . . . .     | 167 |
| 7.2  | Memory Used: Comparison of Internal Box Sharing . . . . .              | 173 |
| 7.3  | Box Counts as Trees . . . . .  | 174 |
| 7.4  | Average Number of KBytes/Strip vs. Width of Strip . . . . .            | 176 |
|      |  |     |
| A.1  | Mathematica Formats . . . . .  | 206 |
| A.2  | Formatting Primitives . . . . .  | 207 |



|     |  |     |
|-----|--|-----|
| A.3 | Forward Translation: General and Power Rules . . . . .             | 208 |
| A.4 | Forward Translation Summation Rules . . . . .                      | 208 |
| A.5 | Forward Translation Product Rules . . . . .                        | 209 |
| A.6 | Forward Translation Product Rules for Leading Minus Sign . . . . . | 210 |
| A.7 | Backward Translation: General and Power Rules . . . . .            | 211 |
| A.8 | Backward Translation Summation Rules . . . . .                     | 212 |
| A.9 | Backward Translation Product Rules . . . . .                       | 213 |
|     |  |     |
| B.1 | PBox Data Structure . . . . .                                      | 215 |
| B.2 | Box Data Structure . . . . .                                       | 216 |
| B.3 | Path Data Structure . . . . .                                      | 217 |
| B.4 | A Formatting Procedure for DAGs . . . . .                          | 219 |

# List of Tables

|      |  |     |
|------|--|-----|
| C.1  | Sharing Data—X-relative Batch Example ( <i>with</i> invisible parentheses) . . .   | 221 |
| C.2  | Sharing Data—X-relative Batch Example ( <i>without</i> invisible parentheses) . .  | 222 |
| C.3  | Sharing Data—Width-relative Batch Example ( <i>with</i> invisible parentheses) .   | 223 |
| C.4  | Sharing Data—Width-relative Batch Example ( <i>without</i> invisible parentheses)  | 224 |
| C.5  | Sharing Data—Batch Example ( <i>with</i> invisible parentheses) . . . . .          | 225 |
| C.6  | Sharing Data—Batch Example ( <i>without</i> invisible parentheses) . . . . .       | 226 |
| C.7  | Sharing Data—Interactive Example ( <i>with</i> invisible parentheses) . . . . .    | 227 |
| C.8  | Sharing Data—Interactive Example ( <i>without</i> invisible parentheses) . . . . . | 228 |
| C.9  | Sharing Data—Matrix Inverse Example ( <i>with</i> invisible parentheses) . . . .   | 229 |
| C.10 | Sharing Data—Matrix Inverse Example ( <i>without</i> invisible parentheses) . . .  | 230 |
| C.11 | Sharing Data—X-relative Batch Memory Usage . . . . .                               | 232 |
| C.12 | Sharing Data—Width-relative Batch Memory Usage . . . . .                           | 233 |
| C.13 | Sharing Data—Batch Memory Usage . . . . .  | 234 |
| C.14 | Sharing Data—Interactive Memory Usage . . . . .                                    | 235 |
| C.15 | Sharing Data—Matrix Inverse Memory Usage . . . . .                                 | 236 |

## Acknowledgements

This thesis has been a long time in the making, moving me from the ranks of an  $n$ -year graduate student to at least an  $n^2$ -year student. A thesis that takes this long acquires quite a long list of people who have helped in numerous ways. Perhaps the greatest credit should go to my advisor, Micheal Harrison, who grabbed a floundering Ph.D. student and provided help, encouragement, patience, and a polynomial-algorithm for finishing. Credit should also be given to the other members of my committee, Richard Fateman and Philip Collela who insisted on quality work.

I received a great deal of technical help in addition to moral support. In particular, the following people are more than deserving of my thanks: Bruce Char for providing a number of ideas to explore and references to help me explore them; Kamal Abdali for not only his work on MathScribe, but also his undying enthusiasm and encouragement that helped MathScribe and this research get off the ground; Guy Cherry for his work on MathScribe and willingness to listen to crazy ideas; and the gang at the Textronix Computer Research Lab who put up with being dragged over to a screen to be guinea pigs or to evaluate ideas. Special mention should be given to Balaji Krishnamurthy, who as the Lab manager, fought for MathScribe and allowed me the time off to work on this thesis.

The mathematical editing community has been very helpful. They have managed to transcend the boundaries that companies typically impose on their proprietary work and provide answers as to how their systems work. Accordingly, I extend my thanks to: Dennis Arnon (Camino Real), Ron Avitzur (Milo), Alan Bonadio (Theorist), Benton Leong (Maple), Fred Mueller and Richard Smabey (MathSoft), and Albert Rich (Derive). These people have not only helped make this thesis more accurate, but have also helped by listening to ideas and providing feedback.

Of course I would be remiss in not thanking my many friends who provided encouragement and support through my numerous illnesses and setbacks. There is nothing more encouraging than their many visits and phone calls while lying around in a hospital

and feeling cruddy. Just the fact that I can not name them all inside of a page makes me feel good—I hope their omission does not make them feel slighted. At the risk of diminishing the importance of any of those friends, four people in particular stand out in my memory: John and Cathy Foderaro, Earl Cohen, and Dan Halbert. John and Cathy gave me not only the basics of chocolate and shelter on my trips to Berkeley, but also friendship. Dan consistently helped me along when I was feeling down by suggesting things to do and work on to get me started again. Earl helped at a point when I felt like quitting by insisting that an algorithm that is faster and uses less space than other algorithms is an algorithm worthy of publication.

Lastly I come to my wife, parents, and relatives. Perhaps the only thing more important than your friends is your family. I have been blessed in both categories. I would like to thank all of my relatives for their support over the years. In particular, I would like to thank my cousin, Bill Soiffer, whose continuing fifteen year bout with cancer has provided a shining example of courage and perseverance, and whose recent publication of a book shows that it can be done despite the odds.

Although my wife, Carolyn Smith, may have doubted at times that I would ever finish, she kept her pledge of “for better or worse”, despite the imbalance of the latter. When you have backing like her’s, how can you lose?

I would never have made it to college, let alone graduate school, if it had not been for the values and discipline instilled in me by my mother and my father, and for their continued well-wishes and encouragement. Sadly, my father died before it became clear that I would finish. He was the type of parent that wanted nothing more than for his children than to do better than him. To him and the rest of my family, I dedicate this thesis.

# Chapter 1

## Introduction

There are several problems with current computer algebra systems (CASs) that are interface related. These problems include: the use of an unnatural linear notation to enter and edit expressions, a complicated and error prone command-based method to select subexpressions, and the display of large expressions that run off the screen. We believe these problems, which are elaborated on in Section 1.3, have intimidated novice users and frustrated experienced users. A basic assumption of this thesis is that improving the interface to CASs will improve their usability and increase the number of users. The more natural and intuitive the interface (the closer it corresponds to pencil and paper manipulations), the more likely it is that people will want to take advantage of the CAS for its ability to do tedious computations and to verify derivations. Unlike pencil and paper, CAS interfaces can be interactive so that many new and interesting ways of solving problems are possible.

A number of systems have been built that solve one or more of these problems. However, none are designed to deal with the size of expressions easily generated by a CAS. In order to handle large expressions efficiently, new algorithms and data structures must be developed. In particular, new incremental algorithms for the input and redisplay of expressions are presented in this thesis. Other mathematical interfaces use a tree as the fundamental data structure to display an expression. In this thesis, the use of directed acyclic graphs instead of trees as an underlying data structure is explored. Sample tests and analysis show that DAGs significantly decrease *both* the *space* used by the interface and the *time* required to compute the display of large expressions. Other new work in this thesis includes the architecture for the interface and the concept of notation libraries.

The remainder of this chapter presents work that others have done related to the interactive display of mathematical expressions. Problems with CAS interfaces are discussed next. Finally, this chapter gives a road map to the rest of the thesis. We begin with an overview of CASs.

## 1.1 Computer Algebra Systems

A CAS is a collection of algorithms that can solve mathematical problems symbolically. The earliest CASs were often symbolic libraries similar to the numeric libraries that are frequently used today[143]. More modern CASs are interactive and often provide some numeric and graphical capabilities in addition to their symbolic capabilities. Figure 1.1 shows samples<sup>1</sup> from MACSYMA[109], one of the more powerful CASs.

An important distinction between a CAS and a numerical system is that the answer that a CAS returns is exact and does not involve floating point numbers (unless specifically requested). CASs typically use “infinite” precision integers, rational numbers, and arbitrary precision floating point numbers.

The capabilities of different CASs vary, but usually include the ability to perform standard arithmetic and to simplify polynomials. More sophisticated CASs include several of the following capabilities: matrices (arithmetic, inverses, determinants, eigenvalues), tensors, simplification functions (polynomial, trigonometric, factoring), series approximations, solving systems of linear and nonlinear equations, differentiation, definite and indefinite integration, definite and indefinite summation, limits, and solution of differential equations. They also contain a programming language so that users can extend the CAS to solve their own problems. More sophisticated CASs also include numeric and graphics capabilities, the ability to generate FORTRAN and C code, and the ability to output  $\text{\TeX}$  and troff formats for inclusion of answers in documents.

CASs have been used to solve problems in such diverse fields as high energy physics, celestial mechanics, and finite algebra. They have been used to generate sophisticated numerical methods and automatically write the code for those methods. Calmet and van Hulzen[167], Fitch[55], and Ng[127] survey a number of applications.

The first “prototype” CASs date back to the fifties: in 1953 Nolan[128] and

---

<sup>1</sup>The output has been augmented with  $\text{\TeX}$  commands for better display. Figure 1.2 on page 9 shows some unedited output.

*/\* Show that  $\sin(x)^2 + \cos(x)^2 = 1$  by differentiating \*/*

(c1) `diff(sin(x)^2+cos(x)^2,x);`

(d1)  $0$

*/\* The expression is a constant—determine the constant by substitution \*/*

(c2) `ev(sin(x)^2+cos(x)^2, x=%pi);`

(d2)  $1$

*/\* Intersection of two ellipses with swapped major and minor axes \*/*

(c3) `solve([x^2/a^2 + y^2/b^2=1, x^2/b^2 + y^2/a^2=1],[x,y]);`

(d3)  $\left[ \left[ x = -\frac{ab}{\sqrt{b^2 + a^2}}, y = -\frac{ab}{\sqrt{b^2 + a^2}} \right], \left[ x = -\frac{ab}{\sqrt{b^2 + a^2}}, y = \frac{ab}{\sqrt{b^2 + a^2}} \right], \right.$   
 $\left. \left[ x = \frac{ab}{\sqrt{b^2 + a^2}}, y = -\frac{ab}{\sqrt{b^2 + a^2}} \right], \left[ x = \frac{ab}{\sqrt{b^2 + a^2}}, y = \frac{ab}{\sqrt{b^2 + a^2}} \right] \right]$

*/\* Solution of an ODE \*/*

(c4) `ode2(x^2 * 'diff(y,x) + 3*x*y = sin(x)/x, y, x);`

(d4)  $y = \frac{c - \cos(x)}{x^3}$

*/\* Solution of an limit \*/*

(c5) `limit((1+x)^(1/x), x, 0);`

(d5)  $e$

Figure 1.1: Sample CAS Calculations

Kahrimanian[94] independently developed differentiation programs for the Whirlwind and UNIVAC I computers respectively. In 1962, Slagle finished SAINT[150], an integration program that worked via pattern matching and rivaled a college Freshman's ability to integrate textbook problems.

A number of CASs were developed in the sixties. Several of them evolved as new algorithms and hardware were developed and continue to be used today. These include:

**ALPAK**[23, 24, 87] A collection of assembly language routines for polynomial, rational function, matrix and truncated power series manipulation. ALPAK evolved into ALTRAN[79]. ALTRAN added a high-level user language to ALPAK along with

improved algorithms.

**FORMAC[174]** A collection of FORTRAN subroutines together with a preprocessor that converts the user language (a superset of FORTRAN) into a sequence of FORTRAN subroutine calls. FORMAC was converted in the late sixties to use PL/I.

**SAC-I[38]** A collection of state of the art (at the time) FORTRAN subroutines for polynomial and rational function manipulations. SAC-I used “infinite” precision integers. SAC-2[37] is a complete rewrite of SAC-I in ALDES[107], a high-level language designed for algorithmic descriptions. ALDES code is translated into FORTRAN.

**Mathlab-68[46]** The first interactive system designed for mathematicians and engineers to “experiment” with solutions to their problems. Mathlab-68 used Charybdis[119] to print expressions in two-dimensional form (within the limits of a character terminal).

**Reduce[81]** Originally designed to solve problems in Quantum Electro Dynamics, Reduce was generalized to handle more types of calculations by changing its internal representation. It has been expanded and rewritten several times since. Reduce has moved from being a batch system to a widely used interactive system. One design goal that led to the widespread use of Reduce was its portability, and therefore, its availability on many systems.

**Sammet[143]** surveys the capabilities of these and other early systems.

In the late sixties and early seventies, time-sharing systems became standard and most CASs became interactive. CASs developed in this period were much more powerful than those developed in the sixties. Although input remained FORTRAN-like, most systems printed expressions in two-dimensional form. In addition to the reimplementations mentioned above, systems of this period include:

**MACSYMA[109]** An outgrowth of both Mathlab-68 and the Symbolic Manipulation Laboratory (page 8), MACSYMA is intended to be useful to a large variety of users. MACSYMA uses several different internal forms for flexibility and efficiency. MACSYMA, written in MACLISP, initially ran only a few machines at MIT. Vaxima[58] was an implementation of MACSYMA for the DEC/VAX computer. Vaxima was later generalized to run under most UNIX implementations.



**Scratchpad/I**[77] Incorporated many of the same goals as MACSYMA. A significant difference is that Scratchpad/I's interface and internal evaluator use replacement rules as opposed to procedure invocation.

Some more recent systems are listed below.

**Maple**[32] Consists of a small kernel that uses hashing to uniquely store all expressions. Most of Maple's functions are written in the user language and put in libraries. Limiting the number of libraries used allows Maple to be run on smaller machines. Recently, Maple has been split into two parts: a user interface and an algebra engine[105].

**SMP**[36] Pattern matching is a fundamental paradigm of SMP; like Scratchpad/I, SMP's interface is non-procedural. SMP comes with a large library of rules; users can easily add more rules.

**Mathematica**[172] An outgrowth of SMP, Mathematica contains improved algorithms and extended capabilities. Mathematica separates the user interface from the algebra engine. A Macintosh front end for Mathematica is described in Section 1.2.1 (page 13).

With the possible exception of Maple, these systems are restricted to mainframes and workstations with larger memories because of their size. An exception to this is MuMath[142] which has recently been replaced by DERIVE[44]. Both of these systems were designed to run on personal computers with 512k bytes of memory. Their capabilities are more modest than the larger CASs.

The systems mentioned above are examples of general purpose CASs—they can solve problems in a variety of fields of mathematics. A number of special purpose CASs have also been implemented. These CASs tailor their internal data structures to a limited problem domain and hence, are far more efficient at solving problems in that domain. They also tend to have many more functions geared to solving special cases and hard problems in that domain. Some special purpose systems and their fields of specialization are:

**SCHOONSCHIP**[158] —Quantum Electro Dynamics

**CAMAL**[21] —Celestial Mechanics, General Relativity

**SHEEP**[62] —General Relativity

**CAYLEY**[29] —Group Theory

Macaulay[16] —Algebraic Geometry, Commutative Algebra

PARI[14] —Number Theory

Recent research in CASs has revolved around *domain-oriented* computation. A domain-oriented CAS is an object-oriented CAS in which the objects are algebraic domains. For example, polynomials can be defined with the requirement that their coefficients be members of a Ring. Similarly, the Euclidian algorithm can be defined to work on any two objects that belong to a Euclidean Domain. Newspeak[57], Scratchpad II[91], and Views[1] are examples of domain-oriented CASs.

Early work in computer algebra was often performed in the context of artificial intelligence (AI). However, the trend over the years has been away from heuristics and pattern matching and towards deterministic and nondeterministic (probabilistic) algorithms. There are exceptions to this trend however. Scratchpad/I, SMP, and Mathematica make heavy use of pattern matching and rules at the user level. PRESS[156], written in PROLOG at the University of Edinburgh, uses rules and meta rules as part of its basic problem solving mechanism. Fateman[49] discusses some of the drawbacks to this technique. In [26], Bundy presents extensions to the ideas embodied in PRESS. His book[27], covers the broader topic of mathematical reasoning. Calmet and Lugiez[31] propose a CAS design that contains AI techniques and domain-oriented algorithm design.

Other AI work has been directed towards helping users solve problems in CASs. Genesereth[68] discusses many of difficulties that CAS users have when trying to solve their problems. The MACSYMA Advisor[67] is an attempt to solve some of these problems by providing an interactive "consultant." Gardin and Campbell[65, 66] present a system for Reduce that attempts to help inexperienced users correct common "mistakes" when dealing with CASs (e.g., removal of unnecessary evaluations for procedure arguments and use of global flags).

Another related direction has been the use of CASs in education. To date, CASs use in classes has been limited but is growing rapidly; [45] and [155] discuss how CAS are used and might be used in education at both the high school and college levels from the computer algebra community's and educator's point of view respectively. [15] and [149] are among a number of texts that teach various fields of mathematics with the assumption that a CAS is being used. Some CASs specifically designed for teaching are MATHPERT[17], EQD[3, 159], and Bunny Numerics[74].

## 1.2 Related Interface Work

The first CASs were developed on batch processing systems before the advent of time-sharing systems. Input was supplied on punched cards, and the output was printed (usually some time later) on a line printer. The output could contain no special characters such as Greek letters or other mathematical symbols. Today, CASs are typically used on a time-shared computer system via an interactive terminal or on a workstation, but the form of the input and output has hardly changed during this time. Input is still a linear string of symbols, except it is typed on a keyboard instead of a keypunch. Output is still formed using the same limited character set, but it appears on a terminal screen instead of a line printer.

Meanwhile, very sophisticated display of mathematical expressions has become possible using typesetting systems. These systems were developed mainly to allow the inclusion of mathematical expressions in papers and books, and little of this technology has been applied to interactive user interfaces. Until recently, there has been surprisingly little work done on graphical CAS interfaces.

This thesis focuses on keyboard and mouse input; work on other forms of input is discussed in Section 5.6. To date, systems that allow two-dimensional input of expressions use one or more of the following mechanisms: *templates*, *overlays*, and *parsing*. These mechanisms are discussed fully in Chapter 5. Briefly, entering an expression via templates involves choosing a template representing some mathematical notation and then filling in the subparts of that template. For example, a user might choose a fraction template and then fill in the numerator and denominator of that template. Templates lead to a prefix style of entering expressions. Overlays are a variation on templates whereby a selected subexpression is substituted for one of the subparts of the template and then the entire template is substituted back in place of the original subexpression. Overlays allow a more natural infix style of input for many single-operator notations. Parsing involves using precedence relations to bind operands to operators and requires that every mathematical notation have a linearized form. In addition to allowing a natural infix style of input for single-operator notations, parsing allows a natural left-to-right ordering of input for multi-operator notations such as parentheses, integrals, and programming language constructs.

The rest of this section reviews previous and current work. It is divided into work directed towards CASs, work directed towards numerical computation, and work di-

rected towards typesetting. There has also been a great deal of work done in the area of programming environments that is relevant. That work is reviewed in Chapter 5.

### 1.2.1 CAS Interfaces

The earliest system to provide two-dimensional notation for display was Clapp and Kain's Magic Paper I[34]. Magic Paper I was developed in 1963 at Bolt Beranek and Newman and ran on a PDP-1. Minsky's Mathscape proposal[122] is also dated 1963. Martin refined Minsky's ideas and they were eventually incorporated into Martin's Symbolic Mathematical Laboratory[111] at MIT. Later systems regressed from Martin's ground breaking system.

Martin's Symbolic Mathematics Laboratory was a precursor to MACSYMA[109]. His system displayed mathematical expressions on the screen of a vector-based display, and was capable of displaying different fonts and special characters such as an integral sign. It also allowed the use of a light pen to select output from the screen. The light pen was used to select variables and operators. Selecting an operator selected the smallest subexpression containing that operator.

Martin's work never made it out of the laboratory. This is probably due to the fact that at that time graphic display hardware was very expensive and not widely available. There was not much standardization of the available display devices, which meant that porting a graphics system to different hardware was very difficult.

By 1978, many computer terminals were capable of random cursor positioning. Despite this fact, input for all CASs was restricted to be a linear string of symbols. Most CAS improve upon one-dimensional output by displaying subscripts and superscripts (exponents) on different lines, and by displaying quotients vertically. An example of Maple[32] output is shown in Figure 1.2a. Still another improvement is to use "line printer graphics" to crudely represent some special symbols, such as integration signs. Figure 1.2b also shows the same expression output by MACSYMA[109]. MACSYMA embodies more or less the state of the art in expression display for CAS on character output devices. Even the latest CAS, Mathematica[172], which has a separate front-end that runs on an Apple Macintosh, still uses line printer graphics such as dashes ("-") to draw division lines.

Two attempts at editors that understood MACSYMA's expression structure were implemented in 1979 and presented at the MACSYMA Users' Conference in Washington,

(a) Maple

$$\frac{2 \pi \int \cos\left(\frac{\sqrt{3} t}{2}\right), t}{3 e^{t/2}} + \frac{t}{f(t, 2/3)}$$

(b) MACSYMA

$$\frac{2 \%pi \int \cos\left(\frac{\sqrt{3} t}{2}\right) dt}{3 e^{t/2}} + \frac{t}{f(t, -)}_3$$

(c) T<sub>E</sub>X

$$\frac{2\pi \int \cos\left(\frac{\sqrt{3}t}{2}\right) dt}{3e^{t/2}} + \frac{e^t}{f(t, 2/3)}$$

Figure 1.2: Sample Output

D.C. Neither editor received much use and were abandoned until recently. They are described below.

MAC-ED[51], written by Fateman at Berkeley, was the less ambitious of the two implementations and was designed to run on character terminals which were prevalent among the MACSYMA user community at the time. MAC-ED's interface was basically that of a line-oriented editor: selections, changes, etc., were displayed on a new line. Users typed commands to move around the expression and change it by giving part numbers. For example, the command 2 would select the second subexpression of the previous expression and the command 2:x+y would replace the second subexpression of the previous expression by  $x + y$ . Commands were also provided that allowed a user to select a subexpression by searching for it. For example, the command `find(x+y, t)` selects the first occurrence of  $x + y$  in the expression  $t$ . MAC-ED provided two-dimensional display of expressions with elision of detail.

The other editor, written by Hoffman and Zippel at MIT, was an Emacs[153]-like editor for MACSYMA that was designed to run on cursor-addressable character terminals[83, 84]. The editor split the window into two parts: a window where expressions were displayed in their two-dimensional form (it used MACSYMA's output routines) and a buffer window

where users entered commands and expressions. The editor included commands for defining, copying, deleting, and replacing a selection. It also had commands for walking the expression tree and for applying a MACSYMA command to the selection. Selections were indicated by drawing a box with asterisks around the selection (the expression was redrawn in-place—there was no intelligent update of the screen). As with emacs, users could bind whatever function they wished to any key. This allowed frequently used and user-defined functions to be easily applied to expressions. In 1988, Symbolics reimplemented Hoffman and Zippel's editor to include mouse support and integrated it with the window system for the Symbolics 3600 series machines[100].

In 1978, Foderaro[56] added to MACSYMA the capability of generating eqn[97] commands and storing them in a file. The contents of this file can be incorporated into a paper and then run through eqn and troff. Fateman[48] wrote a version for T<sub>E</sub>X without line breaking in 1987. Antweiler, Strotmann, and Winkelmann added a similar feature to Reduce in 1989[8] that addresses line breaking.

Figure 1.2c also shows the previous example as displayed by T<sub>E</sub>X and demonstrates how much more comprehensible typeset output is. To take advantage of this, Foster[59] adapted Foderaro's program in 1983 so that the output was sent to eqn/troff immediately and then displayed on a bitmapped screen. The result was disappointingly slow—one to five minutes per page. Unaware of Foster's work, Leler[104] also wrote a program to display eqn/troff output on a workstation screen which had much better performance. Work on integrating typesetting systems and CASs was abandoned in favor of developing display programs from scratch for numerous technical reasons, some of which are mentioned in Section 1.3.3. Briefly, these include eqn's lack of a line breaking algorithm and the lack of access to troff's internal data structures which prevents the interface from knowing the position expressions on the screen.

DREAMS[59], written by Foster in 1984 as part of his master's project at Berkeley, could display a (single) MACSYMA output expression in a special window on a workstation screen using troff fonts. Anderson borrowed from Foster's work and implemented a system named EXED[6] similar to DREAMS that also allowed selection of a subexpression with the mouse but was not connected to a CAS.

In 1985, Soiffer[104] produced the first interface after Martin that could handle both input and output. This system (simply named the "Reduce pretty printer") was connected to Reduce[81] and split the (non-windowed) Tektronix workstation screen into

two regions, a display region and a dialogue region. The relative sizes of these regions could be changed by users. Each expression was displayed in its own *strip*. The size of the strip was based upon the size of the expression. As each new expression was displayed, the old expressions were automatically scrolled up in the display region. The contents of any strip could be scrolled horizontally independently of the other strips using a joy-disk. The entire display region could also be scrolled up or down using a joy-disk.

Subexpressions of an expression could be selected by users and entered back into Reduce. A selection was made by pushing a mouse button and moving the cursor over a subexpression with the mouse. Similar to EXED, the smallest subexpression whose bounding box contains the cursor was highlighted. When the mouse button was released the highlighted subexpression was entered back into the input stream.

The Reduce pretty printer did not break large expressions into several lines. Instead, it used the interactivity provided by the workstation to allow users to scroll the large expression horizontally. Additionally, large expressions and subexpressions could be collapsed manually (selected via the mouse) or automatically by setting width and depth limits. The subexpressions could be expanded back to their full form at any time.

The limitations of a single display area and the unnaturalness of having a linear input form and a two-dimensional output form were the motivations for implementing MathScribe, the prototype system discussed in Chapter 2. An early version of MathScribe[151] was demonstrated at SYMSAC '86 in Waterloo, Canada in the summer of 1986. The current version of MathScribe is described in Chapter 2. Briefly, MathScribe provides both two-dimensional input and output of expressions in a multi-window environment. Users can freely edit any displayed expression in-place and the expression is reformatted and displayed correctly after each keystroke. The primary mode of input is parser-based, but both templates and overlays are also supported. MathScribe also contains several features to aid in understanding large expressions.

Another system presented at SYMSAC '86 was PowerMath[43] by Davenport and Roth. PowerMath ran on a Apple Macintosh and was somewhat limited in what it could do because of the necessity of running in less than 512K bytes of memory. Both input and output are one-dimensional. PowerMath's novel features include using specialized windows for input, output, function definitions, and for value definitions. The function and value windows define the environment in which the computation is performed. By opening and closing windows, different results could be obtained.

Within a year after the SYMSAC '86 conference, several other systems were announced. The exact chronology is unclear. These systems included GI/S by Young[177], Milo by Avitzur[121], CaminoReal by Arnon[9, 10].

Young, as part of his Master's project at Kent State, produced an interface that was similar to the Reduce pretty printer in concept, but was more sophisticated in many ways. GI/S was connected to MACSYMA and ran on a Tektronix workstation. GI/S had a line-oriented editor and history mechanism in the dialogue area and allowed multiple windows instead of a single window in the display area. Young's interface also included the ability to draw graphics in a window. A significant difference between GI/S and the earlier systems was that it used a display structure that was not necessarily related to the underlying algebra system's representation. This allowed a user to select  $b + c*$  from the expression  $a - b + c * d$ . Another feature of GI/S is that alternate streams of computation could be handled in "scratch windows". All variables in a scratch window were local to that window. This was done by prepending the window name to all variables before sending the variables to MACSYMA, and stripping the prefix before printing the results.

Milo was developed as an aid to solving undergraduate physics homework at Stanford where Avitzur was a student. It was developed for the Apple Macintosh and allows text, expressions, and simple plots to be included in a document. The text, expressions, and plots are on separate "lines" (i.e., vertically separated regions of the window). Milo is a rudimentary CAS and includes some basic simplification commands and a simple pattern matcher. Users can enter rules and have them selectively applied to an expression (or subexpression) in order to simplify or solve a problem. Milo allows two-dimensional input using the overlay model of input.

Milo introduced a slightly different method of selection. Instead of using the smallest box surrounding the mouse, the mouse is used to define a rectangle and the selection is the smallest box enclosing the rectangle. The button down position defines the upper-left corner of the rectangle and the position of the cursor with the button held down defines the lower right corner. This approach allows easier selection of contiguous subexpressions. However, because of its input model, Milo does not allow selection of operators (for replacement/deletion). Milo maintains correct syntax at all times. Milo defines a large number of operators but is not extensible. It does not do line breaking or provide tools for dealing with large expressions (which are hard to generate without a CAS anyway).

CaminoReal was developed at Xerox PARC to explore the idea of "active doc-



uments". CaminoReal can be viewed as an extension to the Cedar environment's multimedia document editor Tioga[161]. CaminoReal adds not only the capability to enter and edit mathematical expressions to Tioga, but also the ability to perform computations on those expressions. Because of this, a user can mail a Tioga document to another user who can verify, evaluate, or change any expression or derivation in the document by invoking CaminoReal.

Entering and editing expressions is performed in a window separate from the main document using templates and/or overlays. Expressions can be freely copied to and from Tioga and are treated as atomic "glyphs" by Tioga and can appear anywhere any other character can appear. A flag is used to determine whether an expression should be displayed by Tioga "as is" or should be evaluated by CaminoReal before displaying.

Besides its integration with a document editor, CaminoReal is unique in that it allows access to several "algebra servers". Users can request that a computation be performed by a CAS that resides on a remote machine. The CAS is started up anew for each computation requested. The CASs were not modified in any way: CaminoReal knows the names of the functions corresponding to each operator for each system. The connection is too primitive for serious use though. Its problems include: lack of saved state, unhandled warning and error messages, the inability to interrupt the computation, and the lack of debugging support for the remote system. In addition to the remote server capability, CaminoReal contains a small, domain-oriented CAS (page 6).

Mathematica[172] is a relatively new CAS that is designed to be used with different front ends. All of the front ends that we have seen are conventional fixed width character interfaces. The Macintosh front end does not support structured equations, but does have the concept of a *notebook*. Notebooks allow text, equations, and graphics to be interspersed in a window (on separate "lines"), similar to Milo. To this it adds outlining: consecutive text, equations, etc., can be linked together and collapsed. These in turn can be linked together and collapsed again. This allows users to easily skip irrelevant or uninteresting parts of the notebook. Notebooks can be used as active documentation for new functions (the expressions can be evaluated) or as active tutorials on various subject matter.

The newest and most novel interface is Theorist[19]. Theorist, which runs on a Macintosh, also contains a small CAS. Like Milo, Theorist supports text, expressions, and graphics on separate "lines". Theorist's graphics package has a very sophisticated user interface that allows direct manipulation of the plots.

In Theorist, expressions can be entered from the keyboard using either the line model (Section 5.4.1) or as in a programming language. Theorist also supports palette entry. The method used to select expressions is similar to that used by Milo. Theorist also allows multiple selections. In general, operations work on each selection independently.

Each window starts with a separate list of predeclared variables and functions (e.g.,  $i$  and  $\sin$ ) which can be changed to support differing notations. Every variable and function used must be given a “type”. Like PowerMath and GI/S, calculations in different windows can produce different results because of differing declarations.

Theorist allows users to commute subexpressions by direct manipulation: a subexpression is selected and can be “dropped” back into the expression at legal places. Similarly, subexpressions (including variables) can be selected and moved to one side of an equation with appropriate results. For example, if  $x$  is selected in the equation  $ax + b = 0$  and moved to the right hand side, the equation becomes  $\frac{-b}{a} = x$ . Theorist attempts to perform only legal manipulations<sup>2</sup> based on the types of the variables being manipulated (e.g, two variables in a product that are declared to be matrices do not commute). Substitution is another direct manipulation that is supported. Substitutions are performed by selecting a defining equation and “dropping” it onto an occurrence of the defined variable.

Theorist recognizes that large expressions can occur and allows users to collapse large expressions into ellipsis (“...”). Sums and products are treated specially and collapse into a single term/factor with ellipses on either side; selection of the ellipsis moves to the next term/factor. Auto elision is also supported.

### 1.2.2 Numerical Interfaces

Two systems for numerical calculations that have two-dimensional editing capabilities were developed by MathSoft Inc. The first of these, MathCAD[113], was written by Razdow in 1985 and is designed for personal computers and limited to using their extended character set and low resolution. MathCAD allows users to mix text, equations, and plots together. They are not restricted to being on a separate line and can even overlap. MathCAD is similar to a spreadsheet in that a change to a value affects calculations and plots to the right and below the change.

---

<sup>2</sup>Determining the legality of a manipulation is quite difficult in general and is not always achieved in the current version of Theorist.

MathCAD uses its own internal numerical routines and provides a limited number of operators. As a user enters an expression, it is reparsed and displayed in its two-dimensional form. Unlike other systems, expressions in numerators, denominators, exponents, etc., must be surrounded by parentheses (i.e., users should consider the underlying linear syntax). The expression is redrawn without the extra parentheses when the focus is moved outside of the expression; when the focus moves back to the expression, the expression is redrawn with the parentheses. Users can move around the expression by the use of cursor keys or a mouse and can freely edit the expression.

MathSoft's second system is MathStation[115] and was developed by Razdow, Mueller, and Smaby in 1988. MathStation is similar to MathCAD in that it retains MathCAD's free-form mixing of text, equations, and plots and its spreadsheet metaphor. However, MathStation is designed to run on workstations and is much more sophisticated. It uses multiple fonts for text and is highly configurable by users. For example, users can define menus, key bindings, operators used, what operators look like (by writing PostScript), and how operators should be translated into FORTRAN. Users can also configure the system to use any set of graphical or numerical libraries desired, including ones the user wrote. MathStation's input model differs from MathCAD; MathStation uses the overlay model of input.

In 1990, Maple was connected to MathStation in order to add symbolic capabilities to MathStation. MathStation and Maple communicate using Maple's standard syntax. Much of Maple's syntax is the same as MathStation's syntax, but it is occasionally necessary to revert to (linear) textual mode in order to get at all of Maple's syntax and commands. Unlike CaminoReal's connections to remote algebra engines, MathStation maintains the connection throughout a MathStation session, allowing the solution of multistep problems. However, the connection does not allow a Maple command to be interrupted, nor does it support Maple debugging. MathStation is not designed to handle the intermediate and large-sized expressions that Maple can easily generate and slows down considerably when it must deal with them. MathStation does not perform automatic line-breaking, but does allow users to manually break expressions at certain points (e.g., in plus, minus, or times operators if they occur at the top level of the expression tree).

### 1.2.3 Document Processing Systems

Related to CAS interfaces are document processing systems that can layout mathematics. Document processors can be divided into two groups: batch-oriented processors (e.g., eqn/troff[131, 97] and T<sub>E</sub>X[98]) and WYSIWYG. Batch-oriented processors suffer from two major disadvantages: their linear syntax and their lack of interactivity. To enter an expression, a user must learn the document processor's linear language. Modification must also be done in this language; the output data structures are not designed for modification.

Most WYSIWYG editors do not support equations. Some exceptions to this are Star[154], Edimath[138, 139], Publisher[117], and FrameMaker 2.0[60]. FrameMaker 2.0 has Milo[121] embedded in it and is described above. Edimath, Star, and Interleaf are similar in the way they handle expressions. They allow limited structured input and editing of expressions. An expression consists of strings (for linear subexpressions) and structures (for expressions that have vertical motion). Strings are treated like any other text. Structures are entered in a prefix manner. For example, to enter  $\frac{a}{b}$ , the "fraction" form is selected, the numerator is filled in with  $a$ , we move to the denominator and then fill it in with  $b$ . This unnatural method of entering infix forms in a prefix manner is mitigated by the fact that linear forms are entered as a string with no structure. Hence, expressions such as  $ab + c$  can be entered in a natural fashion.

VORTEX[33] is a T<sub>E</sub>X-based attempt at merging together the features of a batch typesetting language and a WYSIWYG editor. VORTEX does not currently support two-dimensional mathematical input, but contains hooks to support the multiple representation paradigm when such a front end is developed.

INF<sub>⊕</sub>R[146] is an T<sub>E</sub>X-based WYSIWYG editor with an Emacs front-end. Because INF<sub>⊕</sub>R is written in Lisp, users can extend it by writing their own display forms in Lisp. Also, users can bind keys and use Emacs's command completion to conveniently enter mathematical notation with only a few keystrokes. In principle, INF<sub>⊕</sub>R can connect with MACSYMA or Reduce.

Several mathematics-only editors have been developed for the Apple Macintosh. Typical of these are Expressionist[18] and MathWriter[39]. MathWriter uses palettes and menus and presents the prefix-like, template approach mentioned above. Expressionist is more like Milo in that it uses overlays. Neither system can be extended to handle new display forms and are not designed to handle the large expressions produced by CAS.

INFORM[166] is a math editor that is designed as a subsystem of a WYSIWYG document editor. Unlike the other editors mentioned, INFORM uses a parser[82] based on the ideas of Kaiser and Kant[95]. The parser, display, etc., are based on a grammar that is preprocessed in a manner similar to YACC[93] to produce a running system. Because of the grammar, INFORM can be extended by an expert user, but not while the system is running. INFORM's parsing algorithm is discussed in more detail on pages 126–127.

The Mathematical Formula Editor (MFE) is a recently developed program by Nakayama[126]. MFE is designed as a set of procedures to be incorporated into other programs (in particular, those aimed at Computer Aided Instruction), and called by them for two-dimensional input and output. MFE is similar to WYSIWYG mathematical editors in that most operators are treated as text; only two-dimensional notations have structure. One novel feature of MFE is that quotients are entered by first typing the denominator and then the numerator—the normal order of entry for Japanese mathematics, the author's home country. An experiment by Nakayama with 17 and 18 year-old high school girls showed that even complex formulas could be entered with only 30 minutes of training[126].

### 1.3 The Problem

This section discusses problems with CAS interfaces. A number of these problems have been solved to one degree or another by other mathematical interfaces, most of which were developed after the bulk of MathScribe was finished in 1986. The main problems with CAS interfaces can be roughly divided into:

- entering expressions
- selection and editing of expressions and subexpressions
- formatting and displaying expressions (particularly large expressions)
- ambiguous notation
- session layout
- help systems
- graphics

- numerics

The later three items (help systems, graphics, and numerics) are not addressed by this thesis but are elaborated on in Section 8.2 (Future Work).

It is important that any solution to the above problems be space and time efficient because of the large size of expressions that CASs can generate. To date, all interfaces for two-dimensional editing with the exception of MathScribe, use trees and not DAGs as their fundamental data structure and are less efficient in their use of storage for large expressions. Trees require an amount of space that is linearly proportional to the number of nodes displayed. By contrast, DAGs only require an amount of space proportional to the number of unique nodes displayed. Section 7.1 presents a detailed comparison. Because very few of these interfaces have been connected to a CAS, the expressions that they must handle tend to be small and the decreased efficiency is not important for their intended purpose; their solutions may not be applicable to a CAS interface though.

It is also important that solutions to the above problems be extensible by users to handle new notations. Notation is used to convey information succinctly to the problem solver and mathematicians develop new notation when existing notation is clumsy or non-existent for the problem being solved. If an interface cannot handle a new notation, then it hinders problem-solving.

The remainder of this section presents the above problems within the framework of both *traditional* character-oriented CAS interfaces and more *modern* bitmapped/mouse-oriented interfaces.

### 1.3.1 Entering Expressions

In traditional CASs, expressions are entered in a conventional linear syntax. The lack of two-dimensional input leads to both syntactic errors (missing commas, parentheses, etc.) and structural errors (wrong expression, missing subexpression, etc.). It forces users to learn and use an unfamiliar and possibly clumsy notation. Traditional CASs parse input only after a complete command has been typed; most errors are not discovered until after the expression/command has been completely typed. No CAS attempts automatic error correction and only a few systems allow users to edit commands (usually with a line-oriented editor which has its own syntax to learn).

Some of the modern interfaces use two-dimensional input. Most of these inter-

faces use either templates or overlays that are chosen from menus or palettes (menus that are permanently on the screen) or from their keyboard equivalents. Entering expressions using templates and overlays is data-driven and is inherently extensible (i.e., the algorithm is independent of the template/overlay that is used). Overlays allow infix expressions to be entered in a natural left-to-right order, an advantage over templates. However, templates and overlays do not allow natural left-to-right input of a number of mathematical notations (e.g., integration) and programming language constructs<sup>3</sup>. Parsing allows natural left-to-right input for all of these cases. However, using a parser may require users to type parenthesis that are not normally used in the display of the expression. For example, the expression  $\frac{x-1}{x+1}$  might be entered as  $(x-1)/(x+1)$  using a parser; the parentheses surrounding the numerator and denominator are not used in the display of the expression. Also, parsers must tolerate incomplete syntactic forms that occur before an expression is fully entered.

Some traditional CASs, such as MACSYMA[109] and SMP[36], have extensible parsers that allow users to introduce new syntax (operators). However, the syntax extensions are restricted to be either prefix, postfix, infix, or matchfix (e.g., brackets) operators; other forms of syntactic extension such as programming constructs are not allowed. Most parsing algorithms are not extensible.

All of the modern interfaces mentioned in Section 1.2 enforce correct syntax at all times. The mathematical editors used in document processing systems treat most input as linear strings—"syntax" is limited to two-dimensional forms such as quotients, subscripts, and superscripts. Syntactic correctness can be a nuisance, particularly during editing.

Chapter 5 presents several approaches to entering expressions using a keyboard. Section 5.6 discusses alternatives to keyboard input such as handwriting tablets and voice.

### 1.3.2 Selecting and Editing Expressions

Commands to CASs often use parts of previous commands or previous output. *Selection* is the process of "grabbing" some subexpression of an expression. Traditional CASs perform selection through a few specialized commands such as *numerator* and *rhs* and through a general command (usually called *part*) which requires users to mentally walk through the underlying parse tree which can differ from the displayed expression. Command-

---

<sup>3</sup>Most CASs have a programming language embedded in them.

based selection is both error-prone and unnatural.

Modern interfaces use a mouse to select subexpressions, although selection techniques vary. Using a mouse for selection is essentially error-free because immediate feedback can be provided. Those interfaces that require syntactic correctness allow only syntactically correct subexpressions to be selected; most of the document processing interfaces allow more general selection because most of the expression is treated as a string. For structured notations, selection of multiple subexpressions that do not correspond to the internal representation remains problematical (e.g., selecting all diagonal entries in a matrix). Expression selection is the topic of Chapter 6.

CASs often rearrange expressions in order to more efficiently manipulate them. The resulting expression is often not in the form that users desire or expect[123]. Some traditional CASs allow in-place editing of expressions in order to manipulate subexpressions into the desired form (e.g., a factored denominator). However, just as with subexpression selection, users must mentally walk through the underlying parse tree.

### 1.3.3 Formatting and Displaying Expressions

Most traditional CASs display expressions in their natural two-dimensional form within the limits of a character terminal (Figure 1.2). Modern interfaces use bitmapped displays. Some early attempts used eqn[97]/troff[131] to improve the quality of the display of expressions. There are several problems with using batch-oriented word processors such as eqn/troff and T<sub>E</sub>X[98] to display expressions from a CAS. One problem is that large expressions run off the edge of the screen and there is no way to recover the lost information. Another problem is that troff and T<sub>E</sub>X simply “spray” the bits of an expression onto the screen without retaining any of its internal structure, so there is no way to interact with the output. Lastly, the fonts that are used are a problem. Traditional mathematical typesetting uses many different sizes of normal, italic, and special fonts and were designed with to be used with output devices capable of printing 300 or more dots per inch. The resolution of the display on a workstation is typically between 65 and 100 dots per inch<sup>4</sup>. At 65 to 100 dots per inch, we believe Eqn and T<sub>E</sub>X’s output looks poor—italic fonts are especially hard to read. The only way to compensate for this is to use larger-sized fonts. In order to make the smallest font used for superscripts readable, the default size font must be very large,

---

<sup>4</sup>Higher resolution screens are available, but they remain very expensive. The focus of this thesis is on software ideas that can be used on commonly available hardware today and in the near future.



which means that only small expressions will fit on the workstation screen.

CASs frequently generate large expressions. In fact, their ability to manipulate them is one of the main reasons for using a CAS. It is therefore surprising that the interfaces for CASs handle large expressions poorly. In general, large expressions are treated no differently than small expressions except that they are broken up and displayed over several lines<sup>5</sup>. Each system expends a different amount of effort in determining where a line break should occur. Because of line breaks, operators that have "vertical motion" (such as quotient) that span more than one line must be reformatted into a linear format. This makes it harder to comprehend an expression whose large size already makes comprehension difficult. The introduction of line breaks cause very large expressions to scroll off the top of the screen.

Another problem with large expressions is that they take a long time to display. There are two reasons for this.

1. Most systems are tree-based and format each subexpression in isolation from other subexpressions in the large expression. However, large expressions often have subexpressions that occur repeatedly within them, so the display of a common subexpression is (re)computed for each occurrence of the subexpression. If line breaking is used, the recomputation may be necessary if the same subexpression must be broken across lines and hence, displayed differently.
2. The introduction of a line break can cause the display algorithm to backtrack (because of vertical motion operators) and recompute the position of the subexpression.

For these reasons, large expressions can take longer to display than to compute. Chapter 4 discusses formatting expressions. Section 4.3 discusses large expressions in particular.

Most interfaces do not allow users to specify new display forms for new operators. Of the traditional CAS interfaces, only Mathematica allows users to specify a nonlinear display form for new operators, but the specification is limited to ascii strings. MathStation allows users to specify some kinds of new display forms, but the specification is in PostScript and is meant for expert users. Translations (Chapter 3) and extensible formatting algorithms (Section 4.2) allow the user to specify new display forms.

---

<sup>5</sup>A number of systems have some special printing functions. For example, Mathematica[172] has a function `Short` that prints an expression using a specified number of lines by only showing the first and last parts of the expression and eliding internal parts. MACSYMA[109] has a function `printpois` that can be used to print (large) Poisson series expressions in a more readable format.

### 1.3.4 Ambiguous Notation

There are two ways in which notation can be ambiguous. Generic notation uses the same notation to represent similar but different functions. For example, “+” is used to mean polynomial addition, matrix addition, etc. This is usually not a problem for CASs except when there are major semantic differences in meanings such as is the case for multiplication which is commutative for polynomials but noncommutative for matrices. Most CASs “solve” this problem by introducing a different operator for noncommutative multiplication. For domain-oriented CASs (page 6), this is not a problem.

The second and more fundamental problem is that different fields of mathematics and engineering use the same notation in different ways. Understanding a notation requires knowledge of the problem domain. For example,  $f'$  means “first derivative” in calculus and analysis and means ‘a variable different from  $f$ ’ in other domains. Other examples include  $\bar{x}$  (conjugation, mean, negation) and  $i$  (integer,  $\sqrt{-1}$ ). Conversely, different notations are used to mean the same thing. For example,  $\sqrt{-1}$  is usually represented by  $i$  in mathematics but by  $j$  in electrical engineering. Translations and notation libraries can be used to reduce the ambiguity problem and are discussed in Chapter 3.

### 1.3.5 Session Layout

In traditional CASs, input and output are mingled in a single window and scroll off the top of the window never to be seen again unless some provisions are made for preserving a record (e.g., by storing an expression in a variable or by support for scrolling in a terminal emulator). Many modern interfaces allow multiple windows and expressions to be added or deleted at any place in a window. Adding, deleting, or editing an expression that is not at the bottom of the window can confuse what appears to be a linear flow from the top of the window to the bottom of the window. On the other hand, the ability to delete unimportant results (diagnostic output, part selection, etc.) can clarify derivations and free valuable screen space.

PowerMath[43] avoids the linear flow problem by putting each expression into its own window. The result of a computation depends upon which windows are opened at the time of the computation. This can be confusing in the same way that adding or deleting an expression in the middle of a window can be confusing—there is not necessarily a connection between the assumptions and the results. Another problem with putting each expression in

its own window is that very few expressions can be “remembered” because window clutter soon takes over the screen.

In many interfaces, the scope of a variable is not limited to the window in which it is used. This can cause confusion because dependency information, such as assignment ordering, is not obvious across windows. Both GI/S[177] and Theorist[19] limit the scope of a variable to the window in which it is used. Theorist also maintains dependency information. This thesis does not present any solutions to the session layout problem although Section 8.2.3 briefly discusses some possible approaches.

### 1.3.6 Portability to Different CASs

Every CAS has its own (limited) user interface. Portability of the interface to other CASs is not really an problem with today’s CAS interfaces because they are tied to a particular CAS. However, a good interface requires a substantial amount of work and this work should not have to be duplicated for each CAS. Therefore, it is desirable to produce a portable interface that handles lexical, syntactical, and functional differences between different CASs.

There are two extremes to a portable CAS interface design: fully expose the underlying algebra system (both its strengths and its weaknesses) or try to hide the computer algebra system by defining a new syntax and set of functions. The latter approach requires writing numerous procedures for each algebra system to present a similar interface. For example, sending the simple expression  $a + b/c$  to Reduce returns the answer  $(ac + b)/c$  whereas Maple returns the original expression. The differences in canonical forms means that even the “plus” operation must be dealt with for each CAS. Completely hiding the particular CAS from users is beyond the scope of this research. A middle ground is to hide lexical and syntactic differences; function names, argument order, and function results would be different for each CAS except for those functions which have a common mathematical notation (e.g.,  $+$  and  $f$ ). This problem is discussed further in Section 3.1. Translations (Chapter 3) can be used to help solve the portability problem.

## 1.4 Roadmap to this Thesis

The focus of this thesis is on algorithms and data structures for *efficient* input, selection, and display of (large) expressions. Problems such as session layout and help sys-

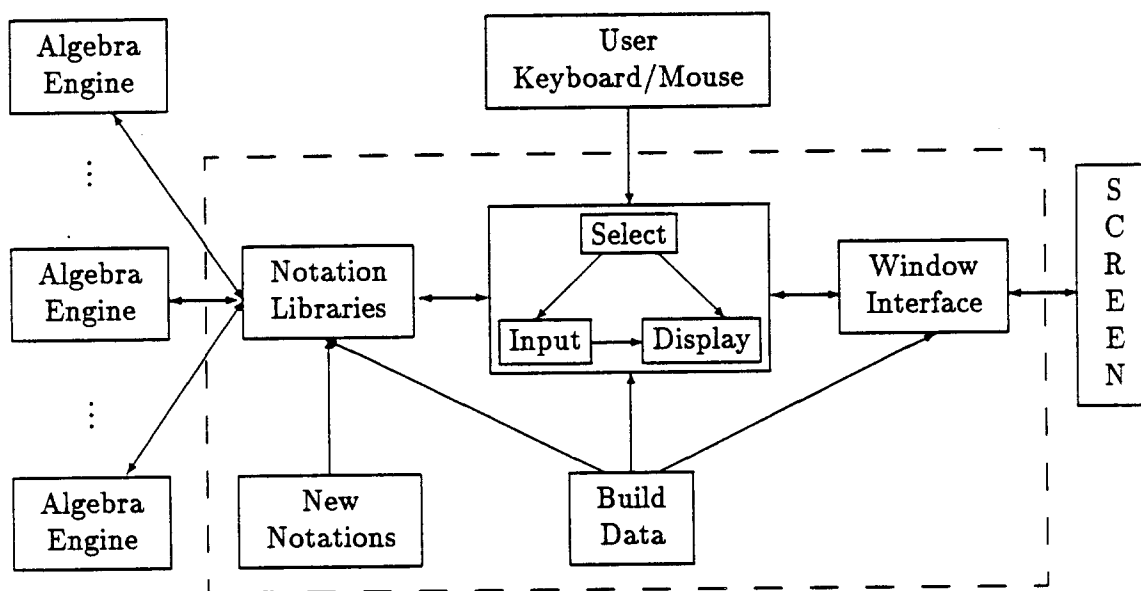


Figure 1.3: Overall Architecture of CAS/Interface

tems are not covered. However, it is impossible to divorce interface issues from algorithms, and this thesis covers several different paradigms where relevant.

A prototype of a CAS interface, MathScribe, was built by the author in order to develop and test many ideas and algorithms related to CAS interfaces. Based on the knowledge gained from the prototype, many new ideas were developed. Chapter 2 discusses what was implemented. The rest of the thesis covers ideas and algorithms that were either implemented in MathScribe or are a direct result of the experience gained from the implementation.

The overall architecture of the CAS and the interface is pictured in Figure 1.3. The arrows indicate the direction of data flow. The architecture is probably similar to other interfaces' architecture, although nothing has been published that describes overall architecture; MathScribe and the concepts discussed in this thesis predate most other interface work. As the figure shows, more than one CAS<sup>6</sup> can be connected to the interface. It is also conceivable that more than one screen might be used. The interface consists of all of the modules inside the dashed box in Figure 1.3.

The transformation subsystem ("notation libraries") serves as the interface between the algebra engine and the select/input/display routines. The transforms are two-

<sup>6</sup>A CAS without a user interface is referred to as a algebra engine.

way transformations, describing how to change the algebra engine's one-dimensional syntax/notation into more standard two-dimensional mathematical notation and vice versa. In MathScribe, these transformations are procedurally encoded. Parser-based and rule-based approaches, along with procedure-based approaches are presented in Chapter 3. Appendix A presents a large example of rule-based translations. Transformations help solve two problems: they isolate the interface from the CAS and increase the interface's portability to other systems and secondly, they help resolve the ambiguity of mathematical notation by allowing the same notation to translate into different CAS commands/objects. Notation libraries appear to be a new idea.

The transformation system has three inputs: the algebra engine, the CAS interface (MathScribe), and user defined transformations ("new notations"). The interface to MathScribe is well-defined and fixed, but the input form from the algebra engine varies from CAS to CAS. This form is given by user defined transformations. In MathScribe, the transformations are fixed at system-build time, but there are no algorithmic restrictions that prevent runtime changes to the transformations used. Switching the transformations used allows the interface to interact with several different algebra engines in addition to allowing application specific notations in a single session.

Once the expression is in "normal" mathematical notation, the formatter, the parser, and the selection mechanism interact using this notation to provide the look and feel of the interface. The notations that the interface supports are user defined. In MathScribe, these can only be defined at system build time, but as with the transformation subsystem, there are no algorithmic restrictions that prevent runtime changes to the syntax and formatting allowed. The selection mechanism is changeable only to a small degree on a per notation basis (e.g., users can define whether a subexpression is selectable and where the cursor should be placed in subexpression).

The display subsystem is responsible for the display of expressions. For each operator (notation), there is a set of rules/procedures that describe how to format that expression. This is discussed fully in Chapter 4. Expression display is not "typeset quality" because of the low resolution of bitmapped displays (Section 1.3.3). An informal experiment was conducted in which users were given their choice of (readable) italic fonts and smaller regular fonts—the smaller fonts were preferred because more expressions could be seen at one time.

The input subsystem is responsible for handling input that effects the structure

of expressions. This input can be either normal keyboard input or menu/palette selections corresponding to input forms. The input subsystem restructures the internal syntax tree appropriately after every character that is typed or after every menu/palette selection so that the modified expression can be formatted and displayed correctly. Reformatting the entire expression can be time consuming. Because real-time update of the display is necessary, an incremental reformatting and display algorithm was developed and is presented in Section 4.1.3. If a parser is used to handle input (other alternatives are presented in Chapter 5), the real-time constraints of updating the display require that the parser be incremental. One of the requirements of the interface is that it be extensible. This requirement rules out a large number of parsing techniques and lead to the development of a new incremental parsing algorithm. Entering expressions is the topic of Chapter 5.

The selection subsystem handles selection, cursor placement, and cursor movement. Chapter 6 discusses several different methods for selecting subexpressions and presents a number of fast algorithms for those methods. The algorithms presented in Chapter 6 have not been previously published.

There are a number of optimizations that can be made to improve the performance of the interface. The optimizations that are discussed in Chapter 7 appear to be new. Probably the most important optimization is to use a directed acyclic graph (DAG) instead of a tree to efficiently represent expressions (the parse/display tree). Section 7.1 discusses this optimization and the data collected to verify that this is a good idea is presented in Appendix C. MathScribe remains the only system to use DAGs, probably because using DAGs complicates the algorithms for parsing, formatting, and selection. To avoid complicating the presentation, the algorithms presented in the body of this thesis assume a tree data structure with parent pointers. Appendix B presents the data structures necessary if DAGs are used along with a sample algorithm using those data structures. All of the algorithms in this thesis are presented in pseudo-C++ code.

## Chapter 2

# MathScribe

This chapter describes MathScribe, a prototype CAS interface designed and implemented primarily by the author. MathScribe was built to test and develop many of the ideas and algorithms described in this thesis. It was also built to test and develop interface ideas for CASs, such as natural forms for entering and selecting expressions. The bulk of the development was performed in 1985 and 1986 and predates most of the interfaces described in Section 1.2 that allow two-dimensional input and output. A brief history of MathScribe, including a list of the CASs to which it has been ported, is given in Section 2.4. Most of the work with MathScribe has used Reduce as the CAS. Reduce was chosen because of its low cost and the easy accessibility of its source code.

In addition to presenting the features of MathScribe, this chapter also discusses the architecture of MathScribe and some problems with the software base upon which MathScribe is built. This chapter serves as a practical framework for the ideas and algorithms discussed in the remaining chapters of this thesis.

### 2.1 Description of MathScribe

This section presents a summary of MathScribe's features. For a full and complete description of MathScribe, see the MathScribe User's manual[114]. In developing MathScribe, a number of alternative ideas were explored. Some of the alternatives and their pros and cons are also discussed.

MathScribe is an X10[145] application, and a picture of a workstation screen running MathScribe is shown in Figure 2.1. A MathScribe session typically consists of a *control*

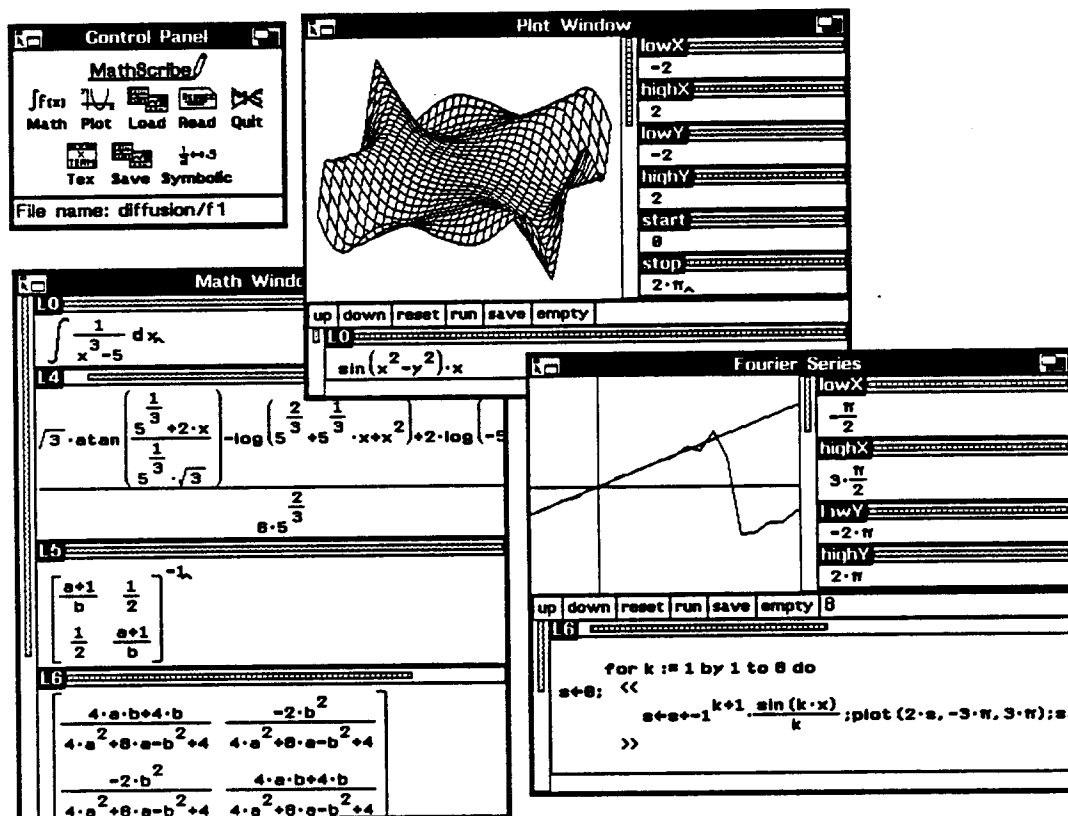


Figure 2.1: Sample MathScribe Session

panel along with a number of *math windows* and *plot windows*. These are described in detail below. Briefly, the control panel allows the user to open new windows. A math window is (conceptually) an infinite sheet of paper on which the user can do calculations, one calculation per *strip*. Strips can be added or deleted at any time. If an expression is too large to be visible in its entirety in a strip, it can be scrolled, collapsed, parts renamed, or parts elided in order to view the expression. A scroll bar allows strips that are off the top or bottom of the window to be made visible. A plot window is similar to a math window with the added capability to plot expressions, run animations, etc., in a special area at the top of the window.

Expressions can be selected using the mouse or the keyboard and can be cut and pasted within and between MathScribe windows. They can also be pasted into other X applications in a variety of forms (see description of "Xterm" control panel icon below). MathScribe uses the standard user-interface paradigm that commands operate on selections



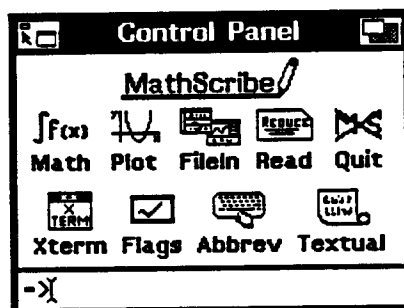


Figure 2.2: MathScribe Control Panel

and that text replaces selections. A single-level “undo” command is provided to recover from inadvertent changes.

Only three commands directly interact with the CAS: “do it,” “replace it,” and “display it”. These commands send the selected expression to the CAS. “do it” ignores the result of the computation (useful for side effects such as plots and assignments). “Replace it” replaces the selection with the result of the computation (useful for in-place manipulations). “display it” creates a new strip below the current one and displays the result in that strip. A fourth command, “plot it,” is available only in plot windows. It calls the CAS to evaluate the selected expression at given points and plots the results.

When MathScribe starts up, the user is presented with a transcript window and a control panel<sup>1</sup>. The transcript window is an standard terminal emulator window (“xterm”) and is used primarily for warnings and textual input. The control panel, as shown in Figure 2.2, contains icons that allow the user to:

**Math** Open a math window—the standard place for performing calculations.

**Plot** Open a plot window—the standard place for drawing plots. Plot windows are extensions of math windows, so calculations can be performed here also.

**FileIn** Read in one or more saved MathScribe windows. The user can save a MathScribe window individually or can save all the windows together. The state of the algebra system is *not* saved (i.e., the values of variables and function definitions), although settings of global flags listed in the *flags window* (see below) are saved and restored

<sup>1</sup>MathScribe is configurable in many ways, including the specification of which windows are brought up when MathScribe starts. For this presentation, the default configuration is assumed.

if the flags window is saved and restored. Plots and animations can be saved and restored depending on the value of a global flag.

**Read** Read in a Reduce file. Typically, the file contains a number of function definitions or batch calculations written in Reduce. A global flag controls whether or not the file's contents are displayed when read.

**Quit** Quit MathScribe. The user is asked for confirmation.

**Xterm** Specify the format for communicating expressions to other X applications via the cut buffer. If an expression is copied, MathScribe puts a copy of the expression in X's cut buffer so that other programs (such as "xterm") can retrieve it. The formats that MathScribe generates are currently limited to eqn and TeX, but it is easy to extend MathScribe so that it also generates Reduce, C, and FORTRAN formats.

**Flags** Open/close a window for setting Reduce flags. This provides a convenient way to find and set some commonly used global flags.

**Abbrev** Open/close the abbreviation window. Abbreviations provide a macro-like facility in MathScribe. These are discussed later in this section.

**Textual** Enter textual mode. The user communicates with Reduce in the transcript window using Reduce's standard character terminal-oriented input and output functions. To use MathScribe's windows, the user must return to window mode (by typing the command "visual").

In addition to these icons, the control panel also contains a text entry area that is used to specify file names for reading and writing.

Math windows consist of a sequence of strips, each of which can be individually scrolled horizontally in order to view parts of a large expression that might be clipped by the window. Strips can be added or deleted at any time. Expressions can be entered and edited only inside a strip. The current selection or text cursor position, along with its last change (for undoing), is saved for each strip.

To enter an expression, the user must first "activate" the strip with either the mouse or with keyboard window traversal commands. This "click to type" focus policy was decided upon after experimenting with a "focus follows pointer" policy (i.e., the active strip

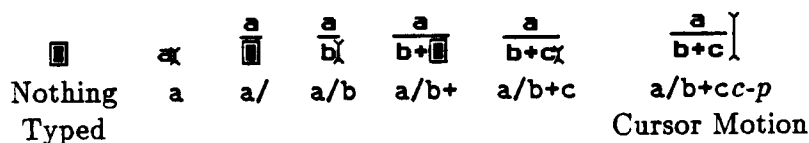


Figure 2.3: Entering  $\frac{a}{b+c}$  Using a Keyboard

is the strip with the cursor in it). The latter policy was rejected because the size of a strip grows and shrinks vertically as the expression changes size. This can result in the cursor accidentally moving to the strip below the active strip if the expression shrinks in size<sup>2</sup>. A simpler model is to have a single selection, etc., per window instead of per strip. The simpler model eliminates the need to choose a policy.

As the user enters an expression, MathScribe maintains correct syntax, typically by inserting dummy operands (“□”) or dummy operators (“□□”). The one exception to this rule is that unbalanced matchfix operators such as parentheses and brackets are allowed. Figure 2.3 shows the display after each character is typed in the expression  $\frac{a}{b+c}$ . Although MathScribe uses standard programming language syntax for entering expressions, there is an important difference: whenever an operator causes the text cursor to move to a different “line” (e.g., division, exponentiation, and diacritical marks), the cursor remains on that line until the user explicitly moves it to another line. Figure 2.3 demonstrates this feature: after typing  $a/b$ , the text cursor remains in the denominator. To return to the “main” line, the user must explicitly move the cursor “up” a level in the expression tree (“control-p” means “move to parent”). Remaining on the same line avoids extraneous (display) parentheses in many cases and more accurately models how mathematics is written by hand. Section 5.4.1 discusses this issue in more detail. Figure 2.3 also shows that the size of the text cursor changes to reflect the subexpression to which text is appended.

The expression  $\frac{a+b}{c}$  is a case where the parsing mode of input requires extraneous parentheses in order to delimit the numerator of the fraction. To overcome this problem, MathScribe also uses the *overlay* model of input. This model of input is similar to templates and is discussed in detail in Section 5.2. Overlays work by substituting the selected expression into the overlay template and then substituting the resulting expression in place

<sup>2</sup>An alternative approach that was not tried is to allow the user to manually size the strip and always ensure that the text cursor is visible by moving the expression up or down within the strip, similar to the way a multi-line text editor works. This is essentially the model that MathScribe uses for expressions that are wider than the strip’s width.

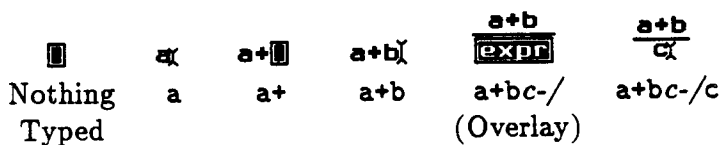


Figure 2.4: Entering  $\frac{a+b}{c}$  Using Overlays

of the original selection. Figure 2.4 shows how the above expression can be entered using overlays. “Control- /” is bound to the division overlay, which is one of many entries in the “common forms” menu (Figure 2.5a). Overlays allow entry in an infix-like manner for infix operators. They can also be used like templates by overlaying a dummy placeholder. MathScribe provides overlays for programming forms so that the user does not have to remember details of the CAS’s programming language. Figure 2.5b shows the menu for Reduce’s programming language.

MathScribe provides interactive horizontal scrolling, collapsing/elision, and re-naming (abbreviations) to help the user comprehend the large expressions that CASs easily and frequently generate; line breaking is not provided. This is partially for the reasons discussed in Section 4.3.1, and partially because of data structure/algorithmic problems

|   |    |
|---|----|
| ( <i>expr</i> )                         | ↑( |
| <i>expr</i> + <i>expr</i>               |    |
| <i>expr</i> - <i>expr</i>               |    |
| <i>expr</i> * <i>expr</i>               |    |
| <i>expr</i> / <i>expr</i>               | ↑/ |
| <i>expr</i> ^ <i>expr</i>               | ↑↑ |
| √ <i>expr</i>                           | ↑\ |
| ∫ <i>expr</i> d <i>var</i>              |    |
| d <i>f</i> ( <i>expr</i> , <i>var</i> ) |    |
| factor( <i>expr</i> )                   |    |
| solve( <i>eqns</i> , <i>vars</i> )      |    |
| Matrix( <i>rows</i> , <i>cols</i> )     |    |
| function( <i>args</i> )                 |    |
| <i>var</i> = <i>expr</i>                | ↑+ |
| <i>macro</i> <i>var</i>                 | ↑@ |

(a) Common Forms

|  |
|--|
| proc <i>name</i> ( <i>vars</i> ):          |
| <i>statement</i>                           |
| begin                                      |
| declare <i>var-type</i>                    |
| <i>statements</i>                          |
| end  |
| <i>expr1</i> ;                             |
| <i>expr2</i>                               |
| <<   |
| <i>statements</i>                          |
| >>   |
| if <i>cond</i>                             |
| then <i>expr</i>                           |
| if <i>cond</i>                             |
| then <i>expr</i>                           |
| else <i>expr</i>                           |
| for <i>var</i> := 1 by 1 to <i>stop</i> do |
| <i>statement</i>                           |
| foreach <i>var</i> in <i>list</i> do       |
| <i>statement</i>                           |
| while <i>cond</i> do                       |
| <i>statement</i>                           |
| repeat                                     |
| <i>statement</i>                           |
| until <i>cond</i>                          |
| go to <i>label</i>                         |
| <i>label</i> :                             |
| symbolic <i>proc/expr</i>                  |

(b) Programming Forms

Figure 2.5: Two MathScribe Form Menus

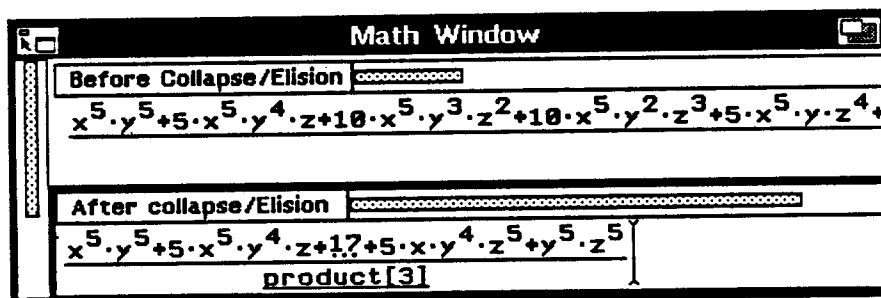


Figure 2.6: An Example of Collapsing and Elision

which are solved with the line breaking algorithm presented in that section.

**horizontal scrolling** If an expression is too wide to fit in the strip, the expression can be scrolled horizontally to view other parts of the expression. Interactivity is an advantage that MathScribe has over pencil and paper.

**collapsing/elision** The user can elide parts of an expression that are not interesting so that disparate parts of an expression, such as the first and last terms in a summation, can be viewed in a strip at the same time. The elided parts can be expanded at any time. If an entire subexpression is selected, it is “collapsed” to its operator’s name and number of terms. If only part of a subexpression is selected, then an ellipsis (“...”) is used, along with the number of terms that have been elided. Both forms of elision are shown in Figure 2.6.

**renaming** Large expressions often contain subexpressions that occur several times in the expression. If one of these subexpressions is selected, then it can be given a (short) name and MathScribe will search for all occurrences of that subexpression in the large expression and replace them with the new name as shown in Figure 2.7. The name and the subexpression are placed in the abbreviation window (Figure 2.8). The renamed subexpression can be expanded to its old form, either individually or globally (i.e., all occurrences) at any time.

If an expression containing renamed subexpressions is sent to the CAS for computation, the subexpressions are *not* expanded to their full form. This allows computations to be carried out more quickly because the large expression is reduced in size. Another advantage is that the results are computed in terms of meaningful quantities (the renamed subexpressions). A significant disadvantage is that the user must ensure that

The screenshot shows a window titled "Math Window" with two sections: "Before renaming" and "After renaming".

**Before renaming:**

|   |   |
|---|---|
| $\frac{x^3 \cdot y - x^2 - x \cdot y^3 + y^2}{x^3 - x^2 \cdot y^2 - x^2 \cdot y - x^2 + x \cdot y^2 + y^3 + y^2}$ | $\frac{x^2 - x \cdot y^2 - x \cdot y}{x^3 - x^2 \cdot y^2 - x^2 \cdot y - x^2 + x \cdot y^2 + y^3 + y^2}$ |
| $\frac{-x^2 \cdot y + x^2 - x \cdot y + y^3}{x^3 - x^2 \cdot y^2 - x^2 \cdot y - x^2 + x \cdot y^2 + y^3 + y^2}$  | $\frac{-1}{x^2 - x - y}$  |
| $\frac{-x^2 \cdot y + x \cdot y + y^2}{x^3 - x^2 \cdot y^2 - x^2 \cdot y - x^2 + x \cdot y^2 + y^3 + y^2}$        | $0$   |

**After renaming:**

|  |  |   |
|--|--|---|
| $\frac{x^3 \cdot y - x^2 - x \cdot y^3 + y^2}{d1}$ | $\frac{x^2 - x \cdot y^2 - x \cdot y}{d1}$ | $\frac{-x^3 + x \cdot y^2 + x \cdot y + y^2}{d1}$ |
| $\frac{-x^2 \cdot y + x^2 - x \cdot y + y^3}{d1}$  | $\frac{-1}{x^2 - x - y}$                   | $\frac{x^2 - x \cdot y - y^2}{d1}$                |
| $\frac{-x^2 \cdot y + x \cdot y + y^2}{d1}$        | $0$  | $\frac{1}{x - y^2 - y}$                           |

Figure 2.7: An Example of Renaming

the values of the renamed variables are not important to the calculation (e.g., that a variable of differentiation or integration is not in the renamed subexpression or that there is no algebraic dependence between renamed subexpressions that might lead to a hidden division by zero). The decision to not expand the renamed subexpression was somewhat arbitrary, and the inclusion of a user-settable switch to control whether expansion takes place is probably appropriate.

Abbreviations are the inverse operation of renaming and provide a simple macro-like facility in MathScribe. Abbreviations can be added, deleted, or changed at any time by editing a special math window known as the *Abbreviation Window*. The labels in the abbreviation window are the abbreviations and the expressions in the corresponding strips are their expansions. Figure 2.8 shows an abbreviation window. The window defines an abbreviation for “avg”. If the user types `avgESC`, then  $\frac{x+y+z}{3}$  replaces avg (the vertical bar represents the text cursor).

Abbreviations are implemented by scanning the abbreviation window’s labels, and if a match is found, the expression in the strip replaces the abbreviation. Renaming is implemented in a similar manner, except that the abbreviation window’s expressions are

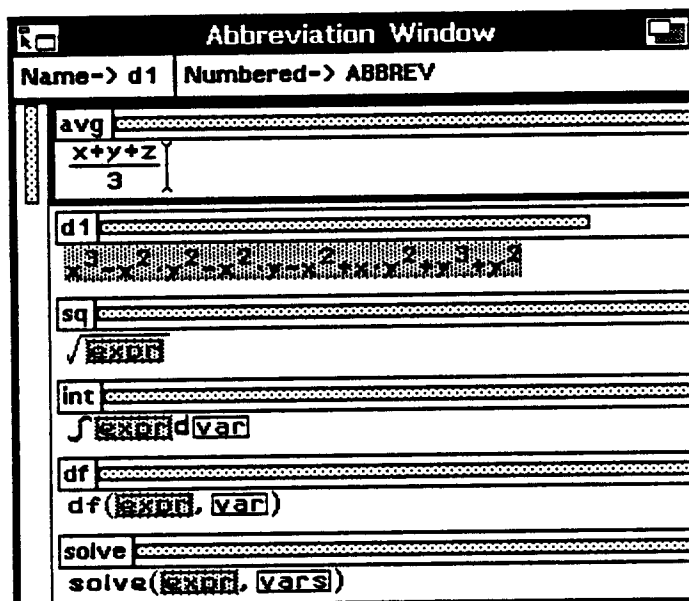


Figure 2.8: Abbreviation Window

scanned and if a match is found, the label replaces the subexpression. If no match is found, the user is prompted for an abbreviation. Because MathScribe stores all expressions uniquely, the expression search only involves pointer comparisons and is very fast.

Plot windows support standard two-dimensional plots, parameterized plots, and surface plots. Multiple plots can be drawn in a single plot window. The plots can be labeled with text or two-dimensional expressions, given axes, etc. Figure 2.9 shows several plots. Plotting is initiated with the “plot it” command and plotting options such as the domain and range of the plot are obtained from the options pane on the upper right side of the plot window. The options can be any expression and are evaluated by the CAS before each plot. A procedural interface to the plotting functions, callable from Reduce, is also supported.

Plotting was not the focus of the MathScribe project but was added because of user demand. The original idea was to use an existing plotting package as a “plotting engine” in much the same way as Reduce is used as an “algebra engine”. However, a number of (non-technical) factors prevented this and MathScribe contains its own plotting code. This allowed experimentation with plotting interfaces. In particular, MathScribe plots are not static pictures. Users can use the mouse to determine what a particular point is and frame regions in order to zoom in on interesting features. Because multiple plots can be drawn on the same plot frame, MathScribe supports an undo command that erases the last change

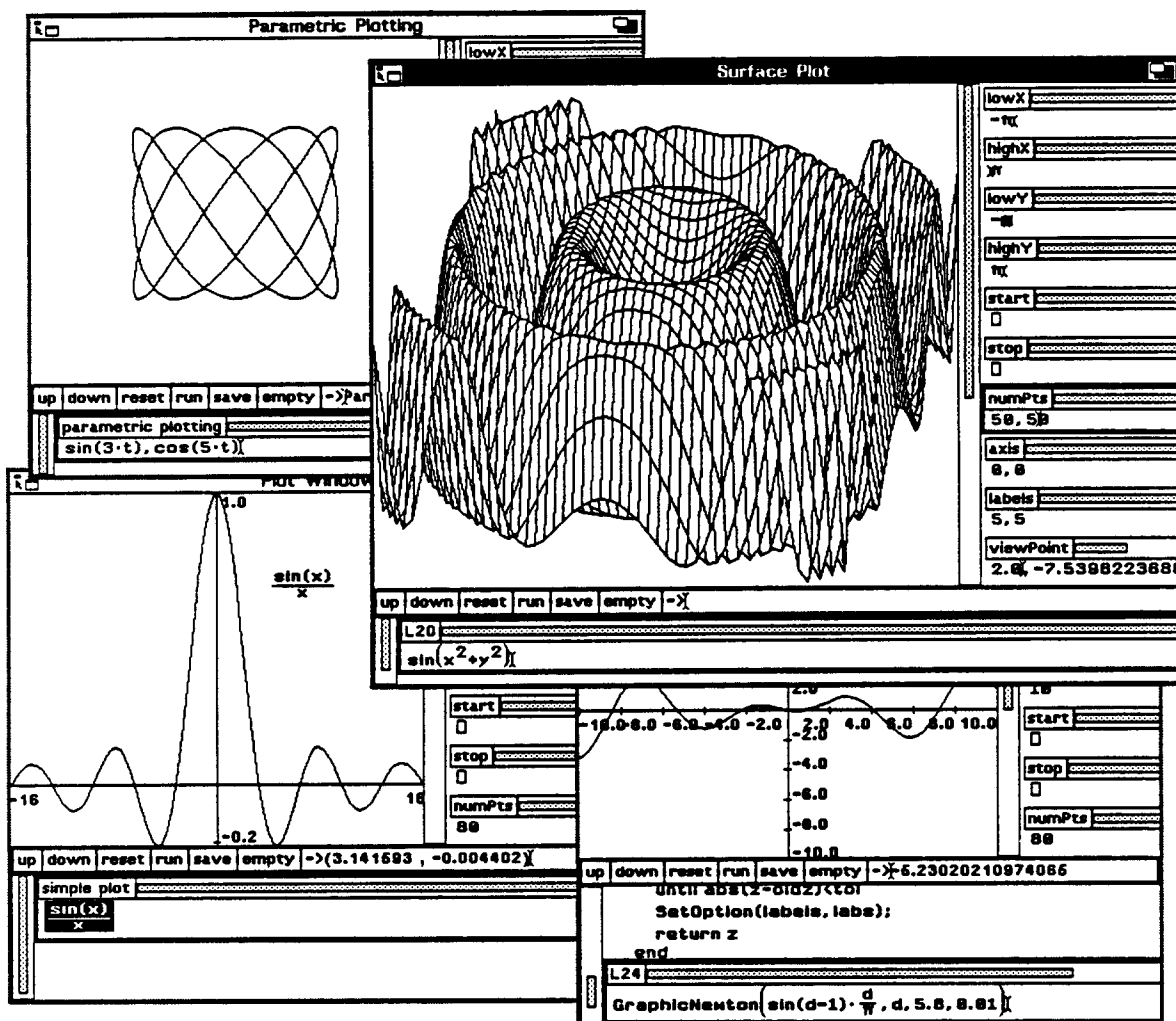


Figure 2.9: Plotting Examples

to the frame. Lastly, MathScribe supports animation: a series of plots can be drawn and then replayed. This feature is particularly useful to view time varying functions and surface plots rotated about some axis.

MathScribe adopted the idea of “spreadsheet” computations from MathCAD[113] by adding two new operators: “bind” and “calc”. Bind temporarily binds the result of evaluating the expression on the right hand side of the bind operator. Calc computes the result of evaluating the expression on the left hand side of the calc operator given the previous bindings and displays the result of the evaluation on its right hand side. The semantics of substitution (binding) are much more complicated for symbolic calculations than for nu-



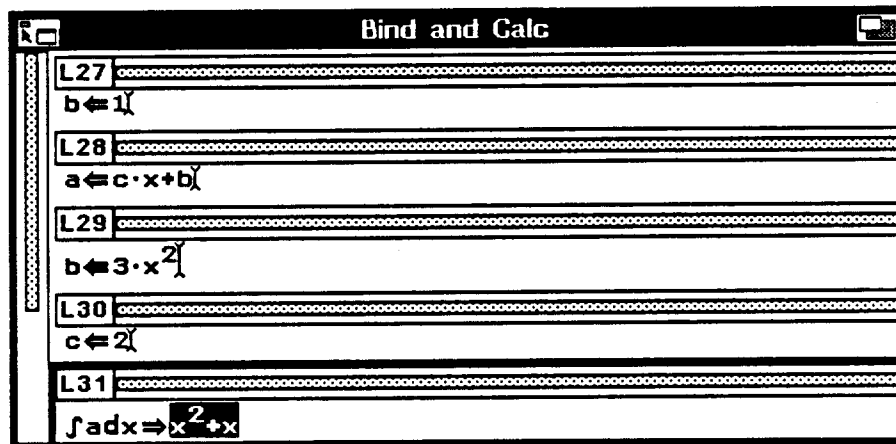


Figure 2.10: Spreadsheet Example

meric computations because an expression on the right hand side of a bind operator can involve variables that occur on the left hand side of other bind operators. This allows different results to be calculated depending upon whether bind evaluates its right hand, whether bindings are cumulative, and whether evaluation is done sequentially or in parallel. Abdali, Cherry, and Soiffer[2] discuss many different possible semantics. MathScribe uses semantics similar to those used for assignment in Maple and Reduce to produce the answer shown in Figure 2.10. Other answers are possible though. For example,  $\frac{cx^2}{2} + x$  corresponds to Macsyma's assignment semantics,  $bx + cx^2$  corresponds to the semantics of Reduce's substitution command (`sub`), and  $x^3 + x^2$  corresponds to the semantics of Reduce's `let` rules for the integral in Figure 2.10.

## 2.2 Overall Architecture

The overall architecture of MathScribe is shown in Figure 2.11. The arrows indicate the direction of data flow. The window interface handles screen layout, window events (expose, resize, add window/strip, and delete window/strip). It also handles key and mouse events, usually by passing them along to the select module. The algebra system interface translates algebra system data structures into MathScribe data structures and vice versa. This is handled by data-driven procedures set up at build time. The algebra system interface also redefines some internal algebra system procedures. These modifications typically deal with capturing output so that output does not go to "stdout," but instead is directed

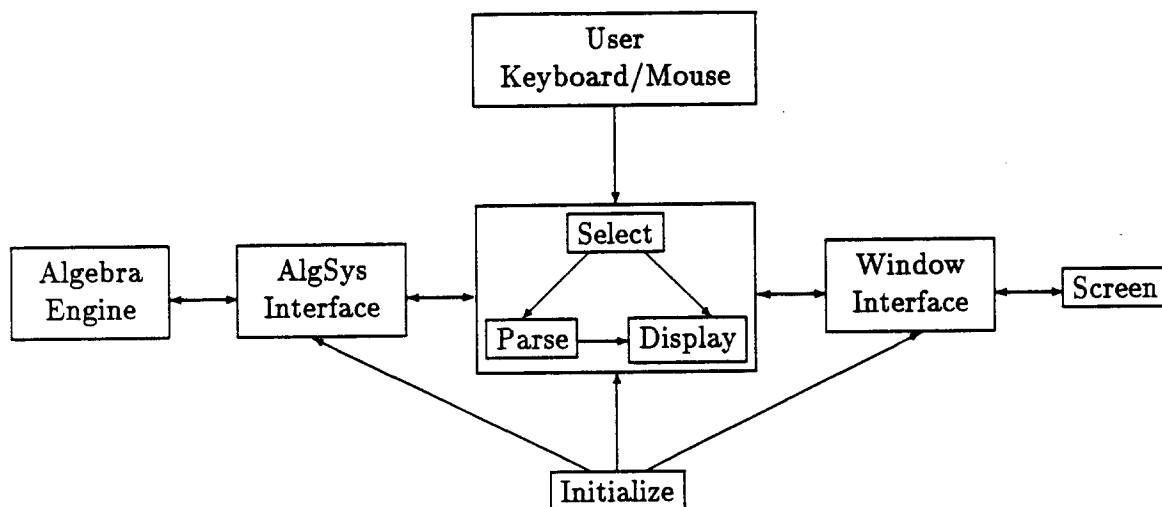


Figure 2.11: MathScribe Architecture

to an appropriate window (transcript window, error window, or question window).

The heart of MathScribe consists of the parse, select, and display modules. The parse module implements an incremental parser and a non-incremental scanner. It is called when an expression is modified, either by adding, deleting, or replacing characters or subexpressions. As the parser builds new subexpressions, it calls on the display module to format those expressions. The select module maintains the position of the text cursor or selection. It is typically invoked by mouse or keyboard events, and calls the parse and display modules as needed. The display module formats and draws expressions to the screen. Formatting is optimized by first checking to see if the expression has already been formatted; if so, no computation needs to be done. Drawing is optimized by not drawing the parts of an expression that are not visible; this is done by passing in a clipping rectangle to the drawing procedure.

Most of the knowledge about the CAS's operators and how they are entered, selected, and displayed is contained in data files that are read during startup. The data files also describe the contents and key bindings of most of the menus. Because the initialization process takes a significant amount of time (about 40 seconds on a Sun 3), startup is divided into two phases: a build phase and a run phase. The build phase initializes those parts of MathScribe that are strongly tied to the CAS and are not likely to vary from invocation to invocation such as how a CAS operator is displayed, etc. The run phase initializes those parts of the interface that are more likely to change such as the fonts used, how the menus behave, initial window placement, etc.

The parsing, selection, and display algorithms in MathScribe are designed to be extensible so that new operators can be added and deleted while MathScribe is running. However, a user interface to take advantage of the extensibility was not implemented<sup>3</sup>.

## 2.3 Implementation problems

MathScribe is built on top of X10.4 and Franz Lisp. While both of these systems were very useful to build upon, they are not perfect. This section discusses some important features that they lack.

X10.4 does not have off-screen “drawables” which makes server-side double buffering impossible. Double buffering is important to prevent “flashing” of the display (clear display, draw) for every character typed. The lack of double buffering requires duplicating some of the server code in MathScribe; expressions are drawn on a bitmap on the MathScribe side of the connection and then the bitmap is sent to the server (instead of sending `DrawLine` and `DrawCharacter` commands). Even though this is slower than clearing the changed area on the screen and then drawing the modified expression directly to the screen, the perceived effect is that it is faster.

A related problem is that whenever a window is moved or resized, the X server clears the window. This prevents smooth vertical scrolling without flashing.

Franz Lisp lacks multiprocessing. This, coupled with the decision to make MathScribe and Reduce run as a single process resulted in a number of problems related to Reduce computations taking minutes or even hours to complete. While Reduce is running, the interface cannot run. This has an unfortunate interaction with the X server: if the mouse enters a MathScribe window, if a MathScribe window has some part of it exposed, or any other X event occurs to a MathScribe window while Reduce is running, events are queued up in the MathScribe process' buffer. After some time, this buffer fills up and the X server suspends on its next attempt to write to the buffer, freezing the screen. A watchdog timer eventually awakens the server and the server closes the connection to MathScribe on the assumption that MathScribe has died because it is not responding to events. A complicated C-level “hack” involving I/O signal handling was necessary to circumvent the lack of multiprocessing in Franz Lisp. This hack allows the user to interact with MathScribe but in only the most trivial manner while Reduce is running (e.g., the user can move a window

---

<sup>3</sup>The changeability of the system was used and tested extensively during debugging.

or interrupt the computation).

As an experiment, Maple was connected to MathScribe running as a separate process (potentially on another machine) via Unix sockets. This version of MathScribe allows the user to continue to use MathScribe freely while Maple is computing. For larger computations, this is a significant advantage. The disadvantage of using two processes is the increased amount of memory used (expressions reside in both processes instead of being shared) and the increased time to communicate between the interface and the CAS.

Another problem with Franz Lisp concerns storage management. In order to build DAGs instead of trees, MathScribe stores the display structure corresponding to a subexpression in a hash table. The hash table is checked to see if a subexpression has already been formatted. When all of the trees that contain the subexpression are deleted, the subexpression is still contained in the hash table and the storage is not freed by the garbage collector. This is a problem with most garbage collectors and is discussed in detail in Section 7.3.

## 2.4 History

Work on MathScribe began in the fall of 1985. MathScribe was first demonstrated at the SYMSAC '86 conference in late July, 1986 on a Tektronix workstation using Reduce as the CAS. Since that time, in addition to data structure and algorithmic enhancements, the following features have been added: generation of  $\text{\TeX}$  and eqn output; error handling; file I/O; and plotting of 2D, parameterized, and surface plots. Most of the subsequent plotting work was implemented by Guy Cherry at Tektronix.

Although MathScribe uses Reduce as its CAS, it has also been ported to MACSYMA. Both of these implementations combined the interface and CAS in the same address space. A port of MathScribe (sans graphics) to Maple was performed in the fall of 1987 with the help of Benton Leong of the Maple group. In this port, MathScribe and Maple ran as separate processes that did not need to be on the same machine.

Originally, the Tektronix workstation on which MathScribe was developed did not have a window system and one was written to support MathScribe. When the X window system began to be adopted as a standard and was ported to the Tektronix workstation, MathScribe was ported to X10.4. MathScribe currently runs on Tektronix 4300 workstations and on Sun 3s.

MathScribe is written in a combination of Franz Lisp and C. MathScribe consists

of about 12,000 lines of Lisp code and 8,000 lines of C code (plus another 4,000 lines of X Window Toolkit code to extend the toolkit that MathScribe uses). MathScribe has also been ported from from Franz Lisp to Common Lisp. In recent years, a large number of improvements have been made to X (now X11) and related toolkits. These improvements would probably dramatically decrease the amount C code required to support the window interface. CLX[144], a Common Lisp library for X windows, would obviate the need to write C code altogether (i.e., window interface part of MathScribe could be written in Lisp). This would also simplify the code and increase its portability (the Lisp-to-C interface is not part of the Common Lisp standard).

MathScribe was intended to be a general interface to a CAS. However, it provides a large number of hooks so that applications can take advantage of its features. Cherry and Todd implemented Design by Dimension[164] on top of MathScribe. This application takes advantage of MathScribe's interactivity, graphics and two-dimensional equations to provide a nice interface to Reduce's equation solving capabilities.

## Chapter 3

# Translations

A CAS interface consists of two different interfaces, the interface that it presents to users and the interface to the underlying CAS. The relationship among these interfaces and the user is shown below symbolically.

$$\text{USER} \longrightarrow \text{CAS-INTERFACE} \iff \text{CAS}$$

This thesis assumes that the interface presented to users resembles standard mathematical notation (the  $\longrightarrow$  above). Converting that notation to the notation of the underlying CAS and vice versa is the topic of this chapter (the  $\iff$  above). We informally refer to such a conversion as a *translation*.

When all of the translations are collected together, we can view the notation conversion problem as one of writing a two-way translator implementing the double arrow above. If the CAS sends and receives strings that are flattened versions of trees, then the trees can trivially be reconstructed and the problem can be thought of as a tree-to-tree transformation problem. Tree transformations have been studied in the context of code generation for compilers. Another view of tree transformation is from the perspective of rule rewriting systems—work on pattern matching and unification is relevant here.

The approaches mentioned above, along with ad-hoc procedure-based techniques, are discussed in the context of translations later in this chapter. This chapter begins by elaborating on the problem to be solved. The ability of the user to add translations to accommodate new or different notations is important. This leads to the new idea of *notation libraries*. Translations, in the context of computer algebra, appear to be new also. The only two interfaces that have dealt with multiple systems are CaminoReal[9, 10]

(page 13) and MathStation[115] (page 15). Both are procedure-based; their approaches along with MathScribe's approach are discussed in Section 3.2. A large example of rule-based translations is given in Appendix A.

This chapter does not present new algorithms nor does it present a specific solution to the problem. Rather, it defines the problem and presents an architecture that has many desirable properties and outlines alternative methods and tradeoffs involved in implementing that architecture. The tradeoffs that a system designer makes decides what translations are possible; the translations that are actually used should be left up to individual users.

### 3.1 The Problem

Formatting an expression from a computer algebra system can be divided into two parts. The first involves deciding upon the overall format of the expression and is the topic of this chapter. The second part involves the exact placement of each subexpression (box) given a format and is discussed in Chapter 4.

The purpose of translations is to arrange the output of the CAS into a form comprehensible by humans, and in the reverse direction to translate mathematical notation into a form that the CAS understands (i.e., a CAS function). For example, if the CAS produces the (prefix) expression

```
Times[x, Power[y, -1]]
```

we need to decide whether we should display this as is, or as  $xy^{-1}$ ,  $\frac{x}{y}$ , or  $x/y$ . We refer to the CAS-to-mathematical notation translation as a *forward translation* and the mathematical notation-to-CAS translation as a *backward translation*. Figure 3.1 shows some examples of possible forward translations. Note that the first and the last translations are valid only if Plus is known to be commutative.

An implicit assumption about translations discussed in this chapter is that they are local in nature and do not take context into consideration. For example, a typical backward translation for  $i$  is `sqrt[-1]`. However, if  $i$  is used as a local variable in some programming construct, this translation translates  $i$  inappropriately. Writing an efficient context sensitive translator is a difficult problem.

The remainder of this chapter uses Mathematica's notation for patterns and rules. This notation is briefly summarized in Figure 3.2. See Mathematica[172, Section 2.3], for

| CAS Form                                  | Literal Display            | Translation?       |
|---|----------------------------|--------------------|
| Plus[-2, x]                               | $-2 + x$                   | $x - 2$            |
| Power[x, -1]                              | $x^{-1}$                   | $\frac{1}{x}$      |
| Power[x, Quotient[1, 3]]                  | $x^{1/3}$                  | $\sqrt[3]{x}$      |
| Power[Sin[x], 2]                          | $\sin(x)^2$                | $\sin^2 x$         |
| Quotient[Sin[x], Cos[x]]                  | $\frac{\sin x}{\cos x}$    | $\tan x$           |
| Plus[alpha, %pi, Sqrt[-1]]                | $alpha + \%pi + \sqrt{-1}$ | $\alpha + \pi + i$ |
| Plus[x, Times[-2, Plus[y, Times[-1, z]]]] | $x + -2(y - z)$            | $x + 2(z - y)$     |

Figure 3.1: Forward Translation Examples

a complete description and many examples of Mathematica's patterns. Unlike the one-way rewrite rules used in Mathematica, translations are normally bi-directional, although there are cases where only a one-way translation is appropriate.

### 3.1.1 Communication

There are two logical places to put the translation mechanism: as part of the front end to the CAS engine, or as part of the communication module of the interface. Placing the translation mechanism in the CAS front end simplifies the interface but places a burden on the CAS front end writer. Alternatively, centralizing the translation mechanism simplifies the CAS front end but complicates the interface. Centralization has the advantage that the

| Pattern           | Meaning   |
|-------------------|---|
| -                 | matches any expression  |
| --                | matches any sequence of one or more expressions   |
| ---               | matches any sequence of zero or more expressions  |
| x_, x_, x_        | as above, but match is named <i>x</i>   |
| x_:default        | matches any expression; if not present, value is <i>default</i>   |
| x_.               | matches any expression; default defined by operator<br>some defaults: <i>Plus</i> =0, <i>Times</i> =1, <i>Power</i> =1, ... |
| pattern ? test    | pattern is matched only if <i>test</i> is true  |
| expr /; condition | matches <i>expr</i> (containing a pattern) if <i>condition</i><br><i>condition</i> may involve several pattern variables    |

Figure 3.2: Patterns in Mathematica



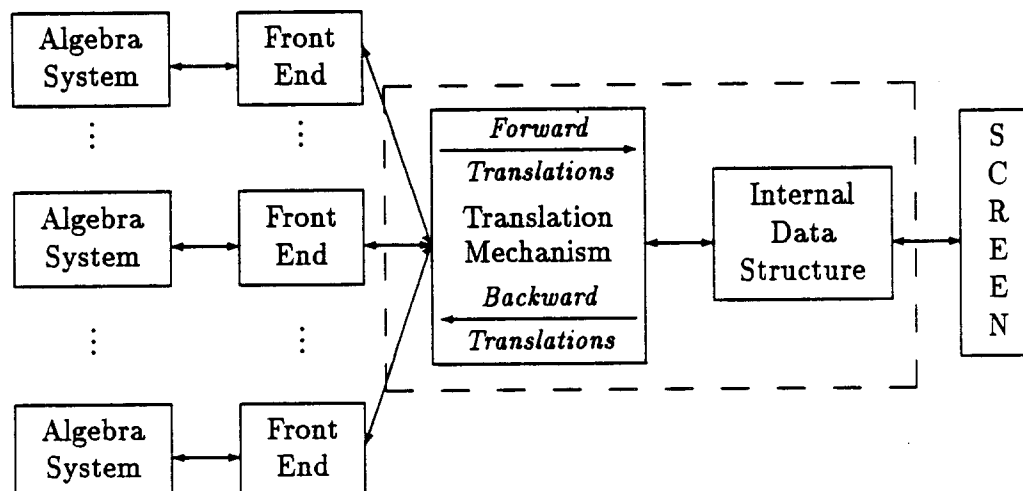


Figure 3.3: Centralized Architecture of Translations

translation mechanism is uniform across all systems and probably results in less complexity overall. Furthermore, if (as is desirable) the interface can communicate with more than one CAS engine, numerical engine, etc., during a session, then centralization results in a smaller overall system because the translation mechanism is not replicated in each engine. This is pictured in Figure 3.3

Decentralizing the translation mechanism results in communication in the interface language<sup>1</sup>. Centralizing the translation mechanism leaves open the question of what the front end sends to the interface and vice versa. CASs differ in many ways:

**Syntax:** The basic syntax of every CAS differs in both the language that they accept and the output that they generate. For example, most CASs use parentheses for function calls; Mathematica uses brackets. Some CASs allow syntactic extensions at run-time.

**Functions:** The functionality of every CAS is different. Even when two CASs have similar functionality, the function names along with their arguments might differ. For example, the definite integration function in MACSYMA has the form:

```
integrate(expr, var, low, high)
```

In Maple, the definite integration function has the form:

<sup>1</sup>A decentralized approach is used with MathLink[173], the communication standard of Mathematica. MathLink is briefly discussed in Section 3.2 (page 55). Because Mathematica is a CAS, this decentralized approach results in communication via MathLink that is in the internal format of Mathematica.

```
int(expr, var=low..high)
```

**Semantics:** Even when names and arguments are the same, the semantics of the function may differ. For example, the “plus” operation for rational functions in Maple does not combine its operands over a single denominator, whereas the “plus” operation for rational functions in Reduce does.

**Representations:** The internal representation used by CASs differ. For example,  $a - b$  is represented by Mathematica as `Plus[a, Times[-1, b]]` and by Reduce (using Mathematica’s notation) as `Plus[a, Minus[b]]`.

These differences, along with the extensibility of CASs (both syntactically and functionally), make a complete specification of the communication protocol impossible at the functional/semantic level. However, it is possible to specify a simple communication protocol that makes tree creation trivial and leaves function name, argument order mapping, and semantics to translations. This approach is compatible with Purtilo’s “software toolbus[136]”. The remainder of this chapter assumes that the CAS and the interface communicate via a linearized Abstract Syntax Tree (AST)<sup>2</sup>.

The interface’s internal data structure is a Concrete Syntax Tree (CST). A CST is used because the data structure is used by the parsing, selection, and display modules. If an AST were used, these modules must either temporarily expand the AST to a CST or derive the missing delimiters in another manner, greatly complicating their algorithms. Using the grammar of Section 5.5.1, the CST is frequently similar to the AST with the addition of delimiters as children of the nodes and does not use much extra space because delimiters are shared. Most backward translations can ignore the delimiters in the CST, but as shown in Section 3.1.4, looking at the delimiters is necessary in order to correctly translate a Plus node.

---

<sup>2</sup>An AST is a tree in which each internal node represents an operator and each child of an operator represents an operand. A Concrete Syntax Tree (CST) is a parse tree that may contain nodes corresponding to internal productions and children that represent delimiters. ASTs are a compact representation of a parse tree.

This chapter uses DAGs and trees interchangeably except where noted. The CAS/interface communication may be via an Abstract Syntax DAG in order to reduce communication costs.

### 3.1.2 Translation Order

Translations can be applied in parallel or in serial. In parallel application, several translations are applied at once and the results are combined to produce a new tree. Serial application is more common and intuitive: the result of one translation is used as input to another translation. Translations can be applied once or iterated until some condition is met. If the condition is that no further translations can be applied, then this is referred to as *idempotent* rewriting.

CASs allow users to define new functions, and a good interface allows users to define new notations. The translation mechanism should be extensible so that new functions can be mapped into mathematical notation and so that new notation can be mapped into CAS functions. The extension mechanism should be flexible enough so that adding a new translation does not require modifying large numbers of previously defined translations. Put another way, adding translations should only require local changes and should not require users to consider all translations that currently exist or might exist in the future.

Most of the translation mechanisms discussed later in the chapter can traverse the tree in either a top-down or a bottom-up manner. An important property to note is that in an AST, some information is implicit. A top-down algorithm allows each form to control how their subexpressions are matched. In a bottom-up algorithm, this information is not available. For example, the node for a Reduce procedure definition is:

```
(procedure name nil expr (param1 ...) body)
```

If “param1” were actually named Plus or some other name with a special notation, then a bottom-up algorithm might incorrectly format the parameters. This situation can be avoided if the CAS front end is required to “tag” every subexpression. If this is required, the node for a Reduce procedure definition becomes:

```
(procedure name nil expr (params param1 ...) body)
```

### 3.1.3 Ambiguity and Notation Libraries

Mathematical notation is ambiguous: there is not always a single translation from a mathematical notation into a semantic meaning (and hence a CAS function). For example, a calculus text might use the prime notation  $f'$  to mean derivative, whereas an algebra text might use the prime notation simply to distinguish  $f'$  from  $f$ . A similar situation exists

| Notations for Symbolic Logic       |   |
|------------------------------------|---|
| Not[p_]                            | $\longleftrightarrow \sim p$              |
| And[p_, q_]                        | $\longleftrightarrow p \wedge q$          |
| Or[p_, q_]                         | $\longleftrightarrow p \vee q$            |
| Or[Not[p_], q_]                    | $\longleftrightarrow p \rightarrow q$     |
| Or[And[p, q], And[Not[p], Not[q]]] | $\longleftrightarrow p \leftrightarrow q$ |
| Or[And[p, Not[q]], And[Not[p], q]] | $\longleftrightarrow p \oplus q$          |

| Notations for Circuit Logic        |                                       |
|------------------------------------|---------------------------------------|
| Not[p_]                            | $\longleftrightarrow \bar{p}$         |
| And[p_, q_]                        | $\longleftrightarrow p \cdot q$       |
| Or[p_, q_]                         | $\longleftrightarrow p + q$           |
| Or[Not[p_], q_]                    | $\longleftrightarrow p \rightarrow q$ |
| Or[And[p, q], And[Not[p], Not[q]]] | $\longleftrightarrow p \equiv q$      |
| Or[And[p, Not[q]], And[Not[p], q]] | $\longleftrightarrow p \oplus q$      |

Figure 3.4: Notation Libraries for Logical Operations

in the other direction—an expression might be displayed differently in different situations. For example,  $\sqrt{-1}$  is typically displayed as  $i$  in mathematics texts, but engineers typically write  $\sqrt{-1}$  as  $j$ . These differences can be accommodated if translations can be grouped together. We refer to the groupings as *notation libraries*.

A notation library typically defines translations for a particular problem domain and a particular CAS. Two examples of this are shown in Figure 3.4. The first example deals with logical operations in the context of symbolic logic. The second example deals with logical operations in the context of circuits where a different notation is used for the logical operations<sup>3</sup>. For simplicity, the right hand side of the translations is given in mathematical notation. In a real notation library, the right hand side would involve procedures, primitives, constraints or whatever mechanism is provided for producing mathematical notation (see Section 4.2). In the example, we assume that the CAS has functions `Not`, `And`, and `Or`, and that the CAS makes only trivial simplifications such as `Not[Not[x]]`  $\rightarrow$  `x` unless specifically asked to simplify or expand the expression. The first three translations in the example are two-way translations and handle cases when there is a one-to-one mapping of a CAS function to a mathematical notation. The latter notations map into more primitive CAS functions. These translations could have been defined as only being backward translations,

<sup>3</sup>The second example overrides the standard translations for `+` and `.` which is probably not desirable. If the underlying CAS is domain-oriented (page 6), then this would not be a problem and the translations would probably be of the form `Plus[x_, y_]`  $\longleftrightarrow$   $x + y$ .

but by defining them to be forward notations also, simple cases of the notation might be recognized and translated (in particular, if the CAS does not simplify them). However, expressions such as  $\sim p \rightarrow q$  would be simplified to  $p \vee q$  by the CAS and displayed as such by the interface. Similarly, expressions such as  $\sim p \vee q$  would be displayed as  $p \rightarrow q$  even if it was entered using the  $\vee$  notation. For this reason, it may be preferable to make the latter translations be backward translations only.

Notation libraries might implicitly rely upon other notation libraries being loaded. For example, a library that handles notations for “special” matrices (e.g., upper triangular) might assume rules have already been specified for normal matrix display, along with rules for the display of polynomials. Notation libraries should have a mechanism to specify upon which libraries they depend.

The combinatorial explosion of having many translations, one for each combination of mathematical subdiscipline, is eliminated by grouping translations together into notation libraries. This grouping allows the desired libraries to be chosen at run-time and the notations incrementally added to the current translations<sup>4</sup>.

### 3.1.4 Pattern Classification

We now turn our attention to some specific examples of translations in order to evaluate possible translation mechanisms discussed in the latter part of this chapter. The patterns are divided into those that directly produce mathematical notation and those that require further processing. This split may be reflected in an implementation by first handling those patterns that may require further processing and using an idempotent matching algorithm; patterns that produce mathematical notation can then be handled with a match-once algorithm.

Figures 3.5 and 3.6 show some common patterns used for forward translations. The result of a match is shown in the third column. The result is either another CAS form (Figure 3.5) or a formatting form (shown in mathematical notation for clarity in Figure 3.6). The patterns are grouped into different categories in order to highlight features that a translation mechanism might need to be able to handle. The example patterns are illustrative of the category; the desirability of each example is discussed in the corresponding subsections. Some translations are non-controversial and would probably be made most of

---

<sup>4</sup>Incremental addition/deletion from a table (or whatever mechanism is used for the translations) is not really necessary. A completely new table can be built from scratch, but this is likely to be inefficient.

| Classification | CAS Form                          | Result Form                  |
|----------------|-----------------------------------|------------------------------|
| Rearrangement  | Plus[x_, n_, y_] /; Number[n]     | Plus[x, y, n]                |
|                | Times[x_, Sum[z_], y_]            | Times[x, y, Sum[z]]          |
| N-ary          | Plus[x_, y_]                      | Plus1[x, "+", y]             |
|                | Plus[x_, y_, rest_]               | Plus1[x, "+", Plus[y, rest]] |
|                | Plus1[list_]                      | <i>list</i>                  |
|                | Plus[x_, Times[n_, y_]] /;        |                              |
|                | Number[n] && n < 0                | Plus1[x, "-", y]             |
|                | Plus[x_, Times[n_, y_], rest_] /; |                              |
|                | Number[n] && n < 0                | Plus1[x, "-", Plus[y, rest]] |

Figure 3.5: Idempotent Forward Translation Pattern Examples

the time while other translations are of more dubious value and would probably be avoided most of the time. As always, there is a broad middle ground where many people would favor a translation and many other people would reject it.

It is useful to tag (internally) the resulting mathematical notation with some indication of its semantic meaning to aid backward translation if identical notation is allowed for expressions with differing semantics. In the interests of brevity and clarity, the tag is omitted from the examples and the discussion. The translations in Figures 3.5 and 3.6 do not match a single CAS form, but are intended for illustrative purposes only. The examples use regular pattern variables (e.g,  $x_*$ ), sequence pattern variables (e.g,  $rest_*$  and  $x_{**}$ ) and conditional restrictions on the patterns (e.g,  $Number[n]$ ).

Figure 3.5 shows two types of patterns: rearrangement patterns and n-ary patterns. Rearrangement patterns reorder terms for more pleasing display. N-ary patterns deal with functions that do not have a fixed number of arguments; repeated matching is used to “process” the arguments one at a time. Parser-based and procedure-based translations discussed later in this chapter can produce mathematical notation for these patterns directly.

**Rearrangement** Rearrangement patterns reorder terms for more pleasing display; the reordering implicitly assumes that the expressions being reordered are commutative. This tends to make their use speculative at best. The first rearrangement pattern shown causes numbers to be printed at the end of a sum. For example, it translates  $2 + x$  into  $x + 2$ . The second pattern causes summations to be printed at the end of a product. For example, it translates  $3 \sum_{i=1}^n x^i (y + 1)$  into  $3(y + 1) \sum_{i=1}^n x^i$ . Because

they only rearrange terms, rearrangement patterns must typically invoke the matcher again on their output.

Many rearrangement rules used in standard mathematical notation are complicated and are not easily expressed as patterns. For example, one rule is that “variables” should be displayed in lexicographic order (i.e.,  $x, y, \dots$ ) followed by “constant variables” (i.e.,  $a, b, \dots$ ) in lexicographic order, followed by constants (e.g., 2) if used in a sum, but reversed if used in a product (e.g.,  $3ax + 2acy - ac + 1$ ). Moses’ paper on simplification[123] discusses ordering of expressions in standard mathematical notation more thoroughly.

N-ary N-ary patterns are used for CAS forms such as `Plus`, `Times`, and `List` that do not have a fixed number of arguments. N-ary patterns are a necessity imposed by most CASs and by the n-ary display nature of these constructs. There are many ways of handling n-ary patterns. The example in Figure 3.5 illustrates a simple method that makes use of an auxiliary function `Plus1` which is assumed to be associative (i.e., `Plus1[x, Plus1[y, z]]` is automatically rewritten into `Plus1[x, y, z]`<sup>5</sup>). The second pattern reinvoke the matcher on its third argument. The third pattern horizontally concatenates its arguments to produce mathematical notation. The last two patterns are a special case of `Plus` for dealing with negative terms. Most algebra systems represent expressions such as  $a + b - c + d$  as something like

```
Plus[a, b, Minus[c], d]
```

Without a special case pattern, this displays as  $a + b + -c + d$ . A translation that transforms *minus* into the equivalent difference

```
Plus[a, Subtract[b, c], d]
```

results in an expression with a reasonable format, but poor behavior when selecting subexpressions (Chapter 6 discusses selection). For example, in the above expression, it is not possible to select  $a + b$  because they are not children of the same parent. This translation also results in a much larger (deeper) tree for a very frequently used operator.

---

<sup>5</sup>Similar functionality can be achieved if the matcher supports explicit functions “Flatten” or “Append.”

| Classification | CAS Form                                       | Result Form                |
|----------------|--|----------------------------|
| Literals       | Pi   | $\pi$                      |
|                | Sqrt[-1]                                       | $i$                        |
| General and    | Power[x_, y_]                                  | $x^y$                      |
| Specialized    | Power[x_, Power[n_, -1]]                       | $\sqrt[n]{x}$              |
|                | Power[x_, Rational[1, n]]                      | $\sqrt[n]{x}$              |
|                | Power[x_, Rational[1, 2]]                      | $\sqrt{x}$                 |
|                | Power[x_, -1]]                                 | $\frac{1}{x}$              |
|                | Times[x_, Power[y_, -1]]                       | $\frac{x}{y}$              |
|                | Times[x_, Power[y_, -1]] /; Atom[x] && Atom[y] | $x/y$                      |
| Recurrent      | Times[[Sin[x_], Power[Cos[x_], -1]]            | $\tan x$                   |
|                | Or[And[p_, Not[q_]], Or[Not[p_], q_]]          | $p \oplus q$               |
|                | HGeo[Times[-1, n_], n_], x_] /; Integer[n]     | $\text{ChebyU}(n, 1 - 2x)$ |

Figure 3.6: Rewrite-once Forward Translation Pattern Examples

The large translation example in Appendix A shows another method for dealing with n-ary patterns (see Figures A.4 and A.5).

The summation example illustrates two points about backward translations. The first is that backward translations must sometimes use delimiters to determine the correct CAS form, and hence, backward translations cannot work from an AST. The second is that backward translations are not always defined by forward translations. For example, there is no pattern to match `Plus1[a, "+", b, "+", c]`.

Most translations produce mathematical notation directly; some examples are shown in Figure 3.6. The translations have been grouped together into the following categories:

**Literals:** These translations involve no pattern variables and are likely to be used.

**Special Cases:** This is a very common form of translations. Typically, there is a general translation and one or more special cases. The special cases involve either specifying more structure or filling in the pattern variables with constants. The most specific rule is the one that should be used. All forward translations fit into this framework by assuming that there are two general translations: one for constants that leaves the constant unchanged, and one for functions that displays the function in a standard



(linear) functional notation. Every other translation is a special case of one of these two translations.

The first four examples involving exponentiation and radicals are probably desirable in most instances. The later examples involving division are more questionable. Not only is  $x^{-1}$  sometimes more appealing, division may not be the appropriate notation for displaying some inverses.

**Recurrent Patterns:** A recurrent translation is one in which a pattern variable occurs more than once on the matching side of the translation<sup>6</sup>. Recognizing a recurrent pattern is beyond the ability of many algorithms. Recurrent translations often introduce short-hand notations for functions and are particularly useful for recognizing symmetry. This is demonstrated by the third recurrent pattern for transforming a special case of the hypergeometric function into a Chebyshev polynomial. The third pattern also demonstrates the usefulness of being able to solve equations in the pattern matcher, or at the very least, substituting values for the bound variable while performing the match. Without this ability, the pattern will not match `HGeo[-3, 3, 1/2, x]` because this is not structurally the same as the pattern. The pattern can also be written as

$$\text{HGeo}[m_, n_, \text{Rational}[1, 2], x_] /; m - n == 0$$

which causes evaluation of  $m-n$  and detects the match independent of the structure of the expression. Side conditions are discussed below.

It is arguable whether the Chebyshev and tangent examples are a proper use of patterns, particularly if the short-hand notation has a meaning to the CAS as might be the case for tangent.

The ability to test conditions while performing a match increases the functionality of a matcher significantly. Side conditions allow matchers to handle recurrent patterns, patterns that involve complex relationships between their pattern variables, and also "special case" patterns. For example, the recurrent pattern

$$\text{Or}[\text{And}[p_, \text{Not}[q_]], \text{Or}[\text{Not}[p_], q_]]$$


---

<sup>6</sup>Recurrent patterns are called nonlinear patterns in the pattern matching community[47]. However, because of possible confusion with the concept of mathematical linearity, nonlinear patterns are referred to as recurrent patterns in this thesis.

can be rewritten using side conditions as

```
Or[And[p_, Not[q_]], Or[Not[r_], s_]] /; p == r && q == s
```

where the condition has been written in standard Mathematica syntax for clarity.

A pattern to detect a special case of the hypergeometric function for Chebyshev polynomials of the second kind is

```
Times[Plus[n_, 1], HGeo[Times[-1, n_], Plus[2, n_], Rational[3, 2], x_]
```

which can be rewritten using a complicated side condition as

```
Times[a_, HGeo[b_, c_, Rational[3, 2], x_] /;
  Solve[{a == n+1, b == -n, c == n+2}, {n}] != {}
```

where {} indicates that there is no solution (match). This example illustrates the power and danger of side conditions in that they can call upon the CAS to impose constraints on the match, and these functions can be very costly and slow matching considerably.

Finally, the specialized pattern for  $x/y$  in Figure 3.6 can be rewritten using side conditions as

```
Times[x_, y_] /;
  Atom[x] && Head[y]==Power && Atom[Part[y, 1]] && Part[y, 2]==-1
```

However, as the above example demonstrates, moving structure into the condition makes the pattern less comprehensible and makes it impossible to order two identical patterns with different side conditions because it cannot normally be determined which side condition is more restrictive. Also, moving structure into conditions can slow down matching by preventing “compilation” of the pattern structure.

## 3.2 MathScribe and Other Systems

Most interfaces have been designed to work only with their internal data structures and can generate one or more fixed external formats (such as PostScript or TeX). Because their internal data structures are similar to ASTs, all that is required to generate the external form is a simple depth-first walk of the data structure. The reverse translation is very hard in general and is typically not performed. Only two interfaces have dealt with more than one compute engine: CaminoReal[9, 10] (page 13) and MathStation[115] (page 15). Although MathScribe only deals with one CAS at a time, its design goal for portability allows it to be easily extended to handle more than one engine at a time. All three systems are somewhat

ad-hoc in their translation mechanisms and are procedure-based (i.e., the translations are encoded procedurally).

Although this chapter discusses communication with CAS engines, it discusses only the “normal” case of handling mathematical expressions. It does not deal with lower-level details such as handling warning messages, error messages, requests for additional input, and intermediate results. Of the interfaces discussed below, only MathScribe handles all of these possibilities.

From a different perspective, Mathematica[172] has designed a communication protocol, MathLink[173], to enable Mathematica to be connected to different front ends. MathLink does not deal with translations; it merely provides a low-level set of functions so that expressions etc., can be sent to and received from Mathematica. The basic form of an expression sent through MathLink is Mathematica’s “FullForm”—the ASCII equivalent of Mathematica’s internal prefix functional form of an expression. Strings can be sent to and parsed by Mathematica by using the Mathematica function `ToExpression`, thereby avoiding internal form. Similarly, strings can be received from Mathematica (instead of internal form) by use of `ToString`.

Purtilo’s Polyolith[136] is similar to MathLink in that it is a communications protocol. It is far more general than MathLink though in that it allows an arbitrary number of modules to communicate with each other; Polyolith provides a module interconnection language to specify how these different modules communicate. This language is at the type level and does not deal with structural transformations as do translations. Minion[137], a CAS “management assistant” built on top of Polyolith, does not attack this problem either.

### 3.2.1 CaminoReal

CaminoReal uses three different CAS engines for its computations. The main compute engine for CaminoReal is its domain-oriented engine and this object-oriented approach strongly influenced the architecture of the translations: each domain is a separate class and has methods that can convert an expression in that domain into an AST (which is also a class). Other CASs are treated as separate classes, and routines are written for those classes that convert the AST into an appropriate string for the CAS engine and vice versa. This architecture is pictured in Figure 3.7. For both SMP[36] and Reduce[81], the AST-to-string conversion is a simple depth-first tree traversal; the string-to-AST conver-

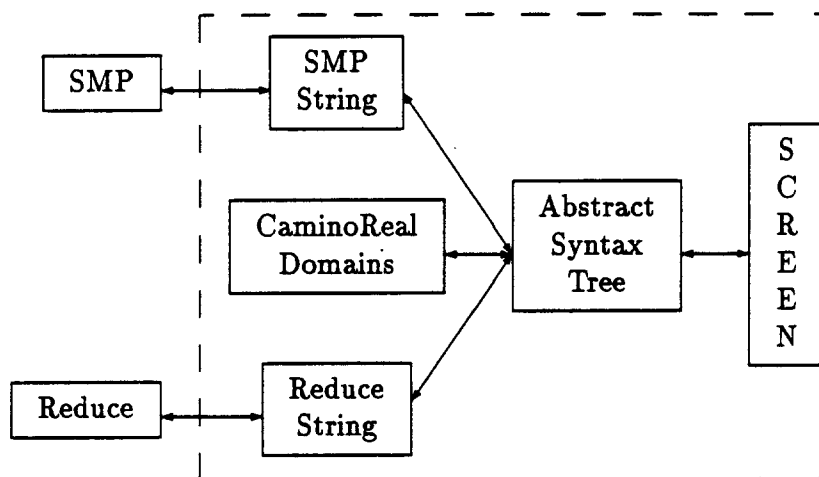


Figure 3.7: Architecture of CaminoReal

sion is basically a simple recursive descent parser. Because of the object-oriented nature of CaminoReal, other conversions can use different algorithms. Currently, CaminoReal only handles “simple” output from Reduce and SMP; it does not handle language constructs, etc. CaminoReal is written in Cedar/Mesa which allows dynamic loading/linking of code, so new methods (translations) can potentially be added during runtime, although CaminoReal was not designed with this in mind.

### 3.2.2 MathStation

MathStation allows users to define simple translations for Fortran and C-based libraries, but they are fixed after the system begins to run<sup>7</sup>. A separate mechanism is used to connect to a CAS engine—it is explained later. MathStation selects the translation to be used based on the signature of the tree node (the result type is not used). For example, two different translations are used for the expressions below even though the function they call has the same name.

```
Solve(f, 0.0, 3.14)
Solve(f, 0.0, 3.14, "Real")
```

MathStation’s translation mechanism only needs to be able to transform a tree into a Fortran/C procedure/function and determine the appropriate function calls to make. Because

<sup>7</sup>MathStation uses a number of configuration files. After these are read, it might be necessary to link in several libraries. Following this link step, the translations are fixed until the next invocation of MathStation.

the results are simple types (such as numbers and strings), there is no need for a reverse translation mechanism. Adding a new translation potentially involves defining the notation (how it is drawn, how to select from it, etc.), its interaction with the parser (its precedence, associativity, etc.), and its translation to Fortran. Below is an example of a rule in the MathStation grammar that defines a Fortran function INTGRT:

```
INTGRT(f, lower, upper, tolerance)
```

The function has arguments of type "function," "real," "real," and "real" and returns a result of type "real." The details of defining the integrate notation are omitted, but define a standard integrate notation  $\int_a^b f$ , where  $a$  is the first child,  $b$  is the second child, and  $f$  is the third child of the MathStation\_Integrate integrate tree node:

```
op_meaning MathStation_Integrate
  real:out real:in real:in function:in
  "INTGRT(#3, #1, #2, 0.001)"
```

The pattern macros are restricted to be children positions of the "MathStation\_Integrate" node.

If an expression can be evaluated by two different functions (e.g., two different integration functions), then it is necessary to have an invisible "parameter" on the screen to distinguish the trees. MathStation allows users to edit the invisible parameters in order to change the function that gets called. Alternatively, users can bind different keys/menus to the "same" notation but with different invisible parameters. Because the libraries are all linked together, the functions in each library must have different names.

The communication mechanism with CAS engines is different from that for numerical engines. MathStation uses an external string representation for an expression. MathStation was designed to be used with Maple, and the external representation used is the representation that Maple uses<sup>8</sup>. The string sent to the CAS is Maple input, and the string returned is one-dimensional Maple output (e.g.,  $a+b/c$ ). This string is sent to the front end for the CAS engine which translates MathStation's external representation into the appropriate input for that CAS. The front end also performs the reverse operation. The external representation is meant only for expressions; language constructs are not handled (they are ignored) and pose problems if there is no equivalent construct in Maple. This architecture allows per-system configurability of translations independent of MathStation,

<sup>8</sup>In fact, Maple is linked into MathStation and does not run as a separate process. Conceptually however, the link to Maple is not different than links to other CASs.

but requires a complete parser/rule system to be built for every CAS engine. It is not configurable by users at run-time.

### 3.2.3 MathScribe

MathScribe allows users to define arbitrary forward and backward procedural translations (in Lisp) at system build-time<sup>9</sup>. No run-time interface for defining additional translations was developed, but there is no algorithmic restriction to prevent run-time additions or deletions. The translations are associated with each operator (along with the procedures for selection, formatting, drawing, etc.). MathScribe can be extended to handle multiple CAS engines at once by extending the forward and backward translations fields to be an association list consisting of pairs “{AlgSystem, translation-procedure}”. The remainder of this section discusses MathScribe as connected to Reduce.

The translations MathScribe makes are from Reduce’s internal form (an AST) to MathScribe’s internal form (a CST) and vice versa. Forward translations use a top-down algorithm because they parse an AST; backward translations work off a CST and can use a bottom-up algorithm. The algorithms are given in Figure 3.8 and Figure 3.9. They are basically tree walks with checks to make sure that no node is traversed more than once (i.e., they work on DAGs). The algorithms contain no built-in knowledge of the translations, allowing easy addition and deletion of translations at run-time. All that is required is a way to map the head of an expression to a forward translation, if one exists.

The forward translation algorithm stores the original expression (before any translations) in the formatted boxes that are built in order to preserve any special information the CAS has associated with the expressions, such as flags used to avoid recomputation (e.g., factored, simplified) and type information. The cached expression is invalidated if any subexpression is changed by an editing command. The backward translation algorithm uses the cached expression if it is valid, otherwise it computes a translation and caches it in order to avoid recomputation.

MathScribe assumes that each child in the AST is really a subexpression. This is not always the case in Reduce, and forward translations are used to canonicalize the AST. For example, Reduce’s internal format of the for loop is:

---

<sup>9</sup>MathScribe is divided into three phases: building, starting, and running. See Section 2.2 for a description of these phases.

```

Box* ExprToBox(AST* expr)
// Returns the unique Box associated with 'expr'
{
    AST* original = expr;          // store original expr to save flags, etc

    // Check to see if a transformation is defined for 'expr'
    if (atom(expr)) {
        expr = ForwardTransformCheck(expr, expr);
    } else if (atom(Head(expr))) {
        expr = ForwardTransformCheck(Head(expr), expr);
    }

    // If 'expr' has already been formatted, there is nothing to do
    if (Formatted(expr)) return FormattedBox(expr);

    // Format the children (including the head), then format the current node.
    // This works when there are no children and when 'head' is non-atomic.
    Box** childrenBoxes = new Box[NumberChildren(expr)];
    for (int i=0; i < NumberChildren(expr); i++) {
        childrenBoxes[i] = ExprToBox(NthChild(expr, i));
    }
    // If 'Head(expr)' is not atomic or there is no operator defined for it,
    // then we format it like a linear function call.
    Operator *op = GetOperator(Head(expr)) | FunctionOperator;
    return op->format(original, childrenBoxes);
}

```

Figure 3.8: Forward Translation Algorithm

(for var (init step final) do/sum/product body)

The child (*init step final*) is not a subexpression. The translation for the Reduce for loop results in:

(for var init step final do/sum/product body)

This makes each child into a “real” subexpression. As initially configured, MathScribe is very conservative about defining translations, and most of the translations for Reduce are defined for language constructs to canonicalize them. An example of a translation for expressions is:

(difference  $x_-$   $y_-$ )  $\longrightarrow$  (plus  $x$  (minus  $y$ ))

This translation canonicalizes  $x - y$ . Another canonicalization is the translation:

(list  $x_-$   $y_-$  ...)  $\longrightarrow$  (list (comma  $x$   $y$  ...))

```

AST* BoxToExpr(Box* box)
// Returns an AST form of 'box'.
{
    // If the backward translation has already been computed, return it.
    if (box->ast) return box->ast;

    // Compute the AST for each child.
    // Many children ASTs will be NULL (the default for delimiters).
    AST** childrenAST = new AST[NumberChildren(box)];
    for (int i=0; i < NumberChildren(box); i++) {
        childrenAST[i] = BoxToExpr(NthChild(box, i));
    }

    // Compute the AST for 'box'.
    box->ast = box->operator->backwardTransform(box, childrenAST);
    delete childrenAST;
    return box->ast;
}

```

Figure 3.9: Backward Translation Algorithm

This allows lists to be treated as any other grouping operator (i.e, lists can use the formatting and drawing functions that are used by parentheses and brackets).

Translations are performed once only; they are *not* performed until there is no change. This prevents infinite recursion and allows translations such as the the `for` translation to be easily written (notice that the `for` translation returns a `for` expression).

The default backward translation for delimiters returns “nil,” for other atoms returns the atom (unless it was a special character such as a Greek letter), and for interior nodes is the name of the operator followed by the non-nil translations of its children. For example, `geq(≥)` uses the default

$$(\text{geq } x \ y) \longleftarrow x \geq y$$

Some other examples of backward translations used for Reduce are:

$$\begin{aligned}
 x &\longleftarrow (x) \\
 (\text{plus } x \ (\text{minus } y) \ z) &\longleftarrow x - y + z \\
 (\text{f } x \ y) &\longleftarrow f(x, y) \\
 (\text{for var } (\text{init step final}) \text{ do body}) &\longleftarrow \text{for } \textit{var} := \textit{init} \text{ by } \textit{step} \text{ to } \textit{final} \text{ do} \\
 &\qquad \qquad \qquad \textit{body}
 \end{aligned}$$

The definitions of all operators known to MathScribe are stored in a data file that is read during MathScribe build-time. The data file consists of an S-expression for



each operator (nonterminals and their associated delimiters are two separate operators). Each S-expression lists the functions to call for formatting, translating (forward and backward), drawing, placement of the text cursor, etc. The S-expression also contains parsing information such as the left and right precedence of the operator, its associated delimiter(s)/nonterminal(s), etc. Numerous default values make many of these entries small. As an example that uses translations, below is the entry for the Reduce operator for:

```
("for" 'NonTerminal :fixType 'nofix
  :transform
    '(lambda (expr)
      '(,(first expr) ,(second expr) ,@(third expr) ,@(cddddr expr)))
  :format '("for " ?1 " := " ?2 " by " ?3 " to " ?4 " " ?5 )
          (" " ?6))
  :algSysForm '(for ,(? 1 2) ,(? 1 4) ,(? 1 6) ,(? 1 8))
              ,(? 1 10) ,(? 2 2))
)
```

The *transform* clause gives the forward translation and the *algSysForm* clause gives the backward translation. The *format* clause describes how to format the for statement. A for statement is displayed on two lines, with the body of the for statement indented four spaces. The ?*n* variables represent the *n*th subpart of the transformed expression. As mentioned above, the forward translation is merely used to “flatten” the arguments to the for statement. The backward translation pulls out subnodes from MathScribe’s CST to convert back to Reduce’s form. The variable (? *m n*) represents the *n*th child of the *m*th subexpression.

### 3.3 Parser-based Translations

A parser-based approach to translation is appealing because there are numerous parser generators available and because parsing is relatively fast (linear in the size of the subject). Another appealing aspect of a parser-based approach is that the same mechanism can be used for both translations and input (see Section 5.4). The translations can be described using productions in a BNF grammar and can be pre-processed to speed execution. An example of a BNF grammar for some of the patterns in Figures 3.5 and 3.6 is given in Figure 3.10. The capitalized words are tokens returned by the scanner and the code enclosed in braces ({}) are the actions taken upon reducing the rule (only a few actions are given to keep the example small). The actions build appropriate nodes for the interface’s internal tree/DAG using the primitives list in Section 4.2.1.

```

expr:      var '[' exprList0 ']'          /* general rule for functions */
          { $$ = Horizontal({$1, "(", $3, ")"}, 0, 0); }
          | varNum                       /* general rule for vars, numbers*/

/* some general and special case translations for POWER */
| POWER '[' expr expr ']'
  { $$ = Script($3, { {$4, "Super"} }); }
| POWER '[' expr POWER '[' expr NEGONE ']' ']'
  { $$ = Radical($3, $6); }
| POWER '[' expr RATIONAL '[' ONE expr ']' ']'
  { $$ = Radical($3, $7); }

/* some translations for PLUS and special case for "a + -b" */
| PLUS '[' expr expr ']'
| PLUS '[' expr expr plusExprList ']'
| PLUS '[' expr TIMES '[' negNum expr ']' ']'
| PLUS '[' TIMES '[' negNum expr ']' expr plusExprList ']'
;

exprList0:                               /* a list with potential no elements */
  | exprList1 ;
exprList1: expr                           /* a list with at least one element */
  | exprList1 expr ;
plusExprList: expr
  | TIMES '[' NEGNUM ',' expr ']'
  | plusExprList ',' expr ;

/* Some book keeping rules for handling results from scanner */
varNum:  var | num ;
var:     PI                                     /* literal pattern */
        | ID ;
num:     negNum | ZERO | posNum ;
negNum:  NEGONE | OTHERNEGNUM ;
posNum:  ONE | OTHERPOSNUM ;

```

Figure 3.10: Example BNF Grammar

There are a number of problems and restrictions associated with a parser-based approach:

- Most parsers/languages are not extensible (see Section 5.4). As mentioned there, it is possible to recompute the tables and reload them, although this may be time consuming. Unlike the situation for input, incrementality of the parser is not required for translations because translations always build new parse trees from scratch. Hence, it is not necessary to compute new state information for “saved” parse trees as we

must do for input because there are no saved parse trees. Therefore, recomputation of the tables is not as costly for translations as it is for input.

- Adding (and deleting) productions from a grammar can “break” the grammar. For example, the grammar shown in Figure 3.10 is not an LR(1) grammar[5] because of the second production involving POWER: either NEGONE can be reduced (eventually) to an `expr` and the first POWER production applies, or we can shift on a `]` as intended. Adding a production for `Power[x_, Rational[1, 2]]` has a similar conflict with the third POWER production. In both of these cases, resolving the conflict by shifting on the token is the appropriate action to take because we want the most specific (longest) translation to apply. Similarly, if two reductions are possible, the longest reduction should be used.

The token-based approach to parsing advocated by Pratt[134] is easily extended without causing conflicts. A token-based approach requires using an extended BNF notation in which tokens have both left and right precedences.

- With a traditional parser, the translations must produce mathematical notation—multiple rewriting is not possible without costly repeated full parses. However, an incremental parser can be used to for idempotent matching by deleting the matched node and inserting the replacement, thereby restarting the parse with a minimal amount of overhead.
- Using a BNF grammar, it is not possible to specify recurrent patterns because there is no way to say that two non-terminals must be identical.
- It is not possible to specify side conditions using a normal parser. This is because a side condition cannot be tested until the parser is about to reduce the production. At this point, several other reductions may have already been made and these cannot be undone if the condition is not met (i.e., another potential rule cannot be tried if the current rule’s condition is not met). However, a special purpose incremental parser might be able to be used to overcome this problem by restarting the parser at the point just before the first token of the current production and somehow marking this state as invalid. This approach was not investigated.

Even if a method is developed that allows specification of side conditions, there are still the questions of what conditions can be specified and in what language are they

specified. The answers to these two questions are intertwined. One approach is to allow specification of side conditions in the language of the CAS and to evaluate the side conditions by the CAS. The drawback to this approach is that it is slow. On the other hand, this allows access to the full power of the CAS, including the ability to solve equations if necessary. Another approach is specify the side conditions in the underlying language of the parser (the language in which the parser actions are specified). This may mean that users are forced to learn a new language. In any case, users are forced to learn a new set of functions for manipulating the results of the parser in order to gain access to previous parts of the expression for testing.

- A minor problem is that N-ary patterns are not expressed naturally, but are expressed as

```

exprList1: expr          /* a list with at least one element */
          | exprList1 expr ;

```

This is a (simple) standard idiom that is easily learned. An alternative is to use a preprocessor to generate real BNF from a simplified form.

- A production uses not only non-terminals, but terminals as well. These terminals are typically generated by a separate scanner. Hence, not only must the user write a BNF grammar, the user must also provide some specification to the scanner as to what terminals the parser expects. This might be something as simple as a list of pairs of exceptions “(terminal, name-of-terminal)” that overrides some default rules or a more complicated specification such as that used for a scanner generator such as “lex”[106]. A combined lexical/syntactic specification such as GRAMOL[70] might simplify definition of translations. The latter part of Figure 3.10 illustrates some complications due to parser interactions with a scanner.
- From a practical viewpoint, many parser generators are not completely table-driven: actions are embedded as a case statement in the parser template (see, for example YACC[93]). This means that the underlying language must support dynamic linking/loading in order to support run-time additions/deletions of translations. Additionally, many parser generators are not designed to allow multiple instances to be included in the same program making connections to multiple CASs in the same session impossible.

Attribute grammars[5] are an alternative to LL and LR-based grammars. Attribute grammars associate with each node in the parse tree a set of values (attributes) and a set of equations of the form  $a = f(B)$ , where  $a$  is an attribute and  $B$  is a set of attributes of the node, its parent, or its children. Since  $f$  is arbitrary, an attribute grammar solution is similar to procedure-based solutions that provide frameworks for evaluation such as MathScribe (the root node should have an attribute that is the result of applying the translations). By examining attribute dependences, an optimal order of evaluation can be chosen statically. On the other hand, attributes must be propagated from node to node, which can be cumbersome and slow. Yellin and Mueckstein[176] give an algorithm for automatically inverting a restricted attribute grammar. This allows automatic generation of forward and backward translations from a single description. A number of variations on attribute grammars have been proposed in order to simplify description of the grammar transformations and to formally describe the resulting rewrite system. One example is Attribute Coupled Grammars[64] in which attributes are separated into syntactically-valued attributes and semantically-valued attributes. The latter are restricted to tree manipulation whereas the former are regular attributes. Another example is Tree Transformation Grammars[96]. They are specified by giving a collection of productions, each relating an input subgrammar to an output subgrammar. Most of the drawbacks of applying LL and LR grammars to translations also hold for applying attribute grammars and their variants.

### 3.4 Procedure-based Translations

Section 3.2 gave three examples of procedure-based translations. Procedures provide a general and powerful technique for dealing with translations. Using procedures, general and special cases can be handled, along with recurrent patterns (particularly easy if DAGs are used) and rearrangement patterns. Conditional expressions, if they do not require evaluation by the CAS, are trivially handled. Semantic pattern matching is more problematical because translation procedures are most naturally attached to the head of the related pattern and because semantic matching may require evaluation by the CAS (not the interface). A mechanism that allows easy evaluation of expressions by the CAS eliminates most problems<sup>10</sup>.

---

<sup>10</sup>Providing evaluation of expressions by the CAS is probably easy given that some mechanism must be in place to allow evaluation of displayed expressions.

Although conceptually any algorithm can be implemented, it is probably best to provide a framework for the translations as MathScribe does. This simplifies the addition and deletion of translations and lessens the chance of users making mistakes. The framework can provide for idempotent evaluation or translate-once evaluation (as MathScribe does). The latter is simpler and quicker, and still allows multiple rewrites on a translation-by-translation basis (i.e., each translation is free to invoke the general algorithm on the result of its translation).

Procedure-based translations have a number of practical and theoretical drawbacks. From a theoretical viewpoint, it is not possible to describe what the translations can do. From a practical viewpoint, adding translations at run-time requires an interpreted language or run-time linking/loading, which is not supported in most compiled languages today. Procedure-based translations require users to learn another language or use the CAS's native language (if the connection is to a CAS). The latter alternative can be slow if many expressions are sent to and received from the CAS.

There are three other serious disadvantages to procedure-based translations: users must write two separate procedures (one for each direction of the translation); users can make programming errors that corrupt memory, cause infinite loops, etc.; and writing the translations quite often involves pattern matching/syntax recognition, which can be somewhat awkward in a programming language. Rule-based and parser-based translations provide simpler and more natural notations.

### 3.5 Rule-based Translations

Most translations are naturally expressed as rules and so a rule-based tree pattern matching approach to translations is appealing. A large amount of work has been done on tree pattern matching in a variety of contexts: automatic theorem proving, logic programming, automatic implementation of interpreters and abstract data types, code optimization in compilers, and symbolic computation. Much of this work is concerned with determining if the rule set produces a unique answer. Typically, this is based on showing that the rules are *Noetherian* (i.e., every sequence of simplifications terminates) and that they are *confluent* (i.e., if two rules apply, then the results of the application will eventually be rewritten to a common form). While these are desirable properties, to date, algorithms based on them are very restrictive in the patterns that they allow. Many practical systems,

such as the ones used in CASs, make no guarantees about uniqueness, termination, etc.

If a set of rules is not confluent, then we must decide which match to choose if there are multiple matches. Multiple matches can occur either because a single pattern matches the subject tree in more than one position or because several different rules match the subject tree. For translations, it makes sense to order the rules so that the most specific rule that matches is the rule chosen; this concept is formally defined in [85] and is called *subsumption*. Subsumption defines only a partial order and we must still decide what to do if two unordered rules match at the same node. Some possibilities include: order of definition (as in Mathematica), size of match (as a linear string), or largest reduction (as a linear string). If one or more rules match at different nodes, one of the above possibilities could be used to choose the rule to apply or the first rule encountered during traversal could be used (instead of finding all possible rules). Several commonly used traversal strategies are: leftmost-outermost, leftmost-innermost, rightmost-outermost, and rightmost-innermost. Most of the algorithms presented choose the leftmost-innermost match which corresponds to a left-to-right, bottom-up traversal strategy.

An important consideration when choosing an algorithm is whether rewrite-once or idempotent matching is desired. Some algorithms can be adapted to avoid attempting a complete match on the modified subject tree. This section briefly summarizes some of the previous work done on pattern matching in the context of translations.

A naive algorithm for matching a set of  $n$  tree patterns (translations)  $p_i$  to a subject tree  $s$  performs a depth-first search of  $s$ . At each node of  $s$ , a recursive test is made to see if any of the  $p_i$  match at that node. In the worst case, this naive algorithm can take  $O(|s||p|)$  time, where  $|s|$  is the number of nodes in the tree  $s$  and  $|p|$  is  $\sum |p_i|$ . The expected time however, is linear in the size of the trees [157]. Besides being potentially slow, this algorithm must be modified to handle commutativity, associativity, recurrent patterns, etc. The work described in this section improves upon the naive algorithm either in time complexity or in increased functionality.

First-order unification [41] differs from tree pattern matching in two important aspects. First, pattern matching is asymmetric: there is a pattern and a subject and only the pattern may contain pattern variables. This allows a unification-based pattern matcher to avoid the time consuming "occurrence check." Second, unification only matches a tree against a tree, not against all proper subtrees of the subject as tree pattern matching does. Related to this second difference is that pattern matching is typically concerned with

matching a subject to a set of patterns, not just unifying a single subject and a single pattern. Nonetheless, unification can and has been adapted to tree pattern matching. For example, Milo[121] and Theorist[19] both use a simple version of unification for their rule rewrite systems. The running time of unification is linear in the size of the two patterns[110]; the simple algorithm presented given by Corbin and Bidoit[41] is quadratic, but is competitive with the complicated linear algorithm in practice.

An important property of translations, which is true of many other problems in tree pattern matching, is that the same set of translations are used many times (i.e., adding/deleting translations is an uncommon occurrence). This means that the translations can be preprocessed to speed up matching. Hoffmann and O'Donnell[85] exploit this property and present several algorithms for linear pattern matching. Their algorithms divide into top-down algorithms and bottom-up algorithms.

Hoffmann and O'Donnell's top-down algorithms reduce tree matching to string matching by regarding each path from the root of a pattern  $p_i$  to the leaf as a string in which path numbers are interleaved with the nodes. They then use the Aho-Corasick fast string matching algorithm[4] to match the strings. In order to correlate the string matches to a tree match, each node of the subject tree is assigned a counter that is incremented when a string match that begins at that node is detected. When the count reaches the number of path strings in the pattern, a tree match is found. This results in an algorithm with preprocessing time  $O(|p|)$  and matching time  $O(|s| \sum(\text{suff}(p_i)))$  where  $\text{suff}(p_i)$  is the maximum of the number of suffixes of the root-to-leaf strings of  $p_i$ . This number ranges from 1 in a balanced tree (all paths are of equal length and hence, none can be a suffix of another) to  $|p|/2$ . In the case that all of the patterns are balanced, the running time is  $O(|s|)$ . Hoffmann and O'Donnell also describe an algorithm by Lang et. al.[102] that is based on both Aho-Corasick string matching and Boyer-Moore[22] string matching and appears to achieve sub-linear speeds. Ramesh and Ramakrishnan[140] improve upon Hoffmann and O'Donnell's algorithms both by extending the algorithm to handle recurrent patterns and by reducing the time complexity to  $O(|s| \sum \text{suff}(p_i^*))$ , where  $p_i^*$  is the set of all root-to-leaf strings of  $p_i$  whose leaves are pattern variables. Note that  $\text{suff}(p_i^*) \leq \text{suff}(p_i)$ . Hoffmann and O'Donnell also present a variation on their algorithm that uses bit strings. This algorithm is useful when the maximum height of the patterns is less than the machine word length, which is a reasonable restriction for translations. This algorithm runs in time  $O(|s|)$ . Like all of the top-down algorithms mentioned, the preprocessing time of this algorithm is  $O(|p|)$ .



Unlike the other top-down algorithms that require  $O(|p|)$  space for their tables to drive the matcher, the bit string algorithm requires  $O(|p|^2)$  space. All of these algorithms can be adapted to efficiently handle idempotent rewriting.

Hoffmann and O'Donnell present a number of bottom-up matchings algorithms. An algorithm which runs in time  $O(|s|)$  is given, but this algorithm might require prohibitive amounts of space (and time) to store the preprocessing information. The basic idea behind the algorithm is to find, at each node of the subject tree, the set of all patterns and subpatterns that can match at that node. These sets are encoded by some enumeration. Given the match sets of all of the children, the match set for a parent node can be (pre)computed by considering all the possible combinations of the children match sets. If the match set for a node contains a pattern (as opposed to just subpatterns), then a match occurs at that node. Matching is accomplished by a simple post-order traversal of the subject tree: each node is assigned an enumeration based on the node's value and the enumerations of its children. Hoffmann and O'Donnell determine the enumeration by a simple table lookup. Although fast, this method uses a table per function whose size is  $q^{f_n}$ , where  $q$  is the number of matching sets ( $q \leq 2^{|p|}$ ) and  $f_n$  is the arity of the function  $f$  (constants have a single table entry). Thus this algorithm uses  $O(q^{\max(f_n)}|f|)$ , where  $|f|$  is the number of symbols (both functions and constants). Alternative encodings such as using a trie instead of a table can substantially reduce the amount of space used. However, because the number of match sets  $q$  can potentially be exponential in the size of the patterns, no encoding will have a good worst case performance. By placing restrictions on the patterns, the space requirements can be improved. Hoffmann and O'Donnell develop an algorithm for restricted patterns that uses  $O(|p|^{\max(f_n)}|f|)$  space for its tables. Pelegri[132] gives several algorithms with fewer restrictions and also shows how these algorithms can be adapted to handle recurrent patterns. Hoffmann and O'Donnell claim that in practice, their restrictions have not caused problems. They and Pelegri claim that, in practice, table size is not a problem for the unrestricted algorithms (i.e., worst case behavior is rare). However, for translations, the number of symbols and patterns is large and so these algorithms may not be well suited for this application. Hoffmann and O'Donnell also give an algorithm that uses bit strings to represent match sets. Variations on this algorithm are suggested that trade off speed for space; see [85] for details. These algorithms are easily adapted to efficient handling of idempotent rewriting.

All of the authors assume that functions have fixed arity which is not the case

for most CASs (e.g., `Plus` and `Times`). One possible way of handling  $n$ -ary functions is to convert them to binary functions before handing them to the algorithms. This is inefficient and requires reconversion to their “flat” form for use in selection, etc. A better method would be to adapt the algorithms to handle  $n$ -ary functions; because many of the algorithms are based on finite state automata, they can probably be modified to handle  $n$ -ary functions and patterns specified by regular expressions.

Hashing can be used to speed up matching of strings[80] and trees[42]. The hash function encodes the structures of the patterns and subjects in order to decrease the number of matches to be attempted. Hashing also helps in recurrent matching because identical structures are stored uniquely. Many conditions which restrict the domain of the pattern variable (e.g., `Integer[n]`) can be made part of the hash signature. Cowan and Griss[42] report that an implementation of a hashing pattern matcher for `Reduce` (without many optimizations) was a factor of four faster than `Reduce`'s matcher. Purdom and Brown[135] used a `contains_variable` field (a trivial structure encoding) to speed matching, in addition to using a DAG structure and an `already_simplified` field. They achieved speed up factors from 2.5 to 13; most of the speed up was attributed to the latter two features. Hash functions can be designed which are commutative and associative[73]. Hashing is mainly useful for structural matching.

Pattern matching was used by many of the early integration programs including the second stage of the `SIN`[124], an integration program developed by Moses. To support this stage, Moses implemented a pattern matcher called `SCHATCHEN`[50]. Patterns in `SCHATCHEN` are written in a lisp-like notation and require knowledge of both the representation of expressions and the operation of `SCHATCHEN`. Such knowledge includes the types of semantic matches that `SCHATCHEN` is capable of making. `SCHATCHEN` uses backtracking to find a match if a subpattern does not match.

Press[156, 20] uses pattern matching as its primary means of solving problems and includes some ideas related to hashing. The Press pattern matcher is intertwined with an equation solver and “knows” about commutativity and associativity of addition and multiplication. The pattern matcher works by recursively matching each subterm and solving equations if necessary to find the value of the pattern variable. Because of commutativity, the matcher has a great deal of freedom in choosing which subterm to match first and makes this choice based on a “fuzzy match”—a match based on the structure of the subterms and the subject to be matched, similar to a hash function.

Both SMP[36] and Mathematica[172] rely heavily upon pattern matching. The speed and power of their pattern matchers is important to the success of their systems. SMP's pattern matcher was reimplemented in 1984 by Greif[76] based on algorithms by McIsaac[118] for unification of ordinary, commutative, and associative functions, along with functions that have an identity. The two major ideas developed are early detection of failure and an ellipsis notation to handle n-ary patterns. McIsaac implemented his algorithms in Reduce. Even though they were much more general, McIsaac's matcher ran from approximately the same speed to forty times faster than Reduce's matcher. The Mathematica matcher is similar to the SMP matcher. It contains many optimizations for common, simple cases such as hashing the rule list when possible. Both of these matchers are mainly structural, although some semantic matching occurs through the evaluation of the bindings and through side conditions. Cooperman[40] implemented a matcher that can be used with MACSYMA[109] that appears to have capabilities similar to Greif's matcher.

Both MACSYMA[109] and Scratchpad[77] provide semantic pattern matching facilities that are based on "compilation" of the patterns. Both matchers can recognize  $(x - 2)(3x + 1)$  and  $x^2 + 2$  as quadratic functions of  $x$ . There are fundamental differences in their approach though. The MACSYMA matcher relies on the simplifier to determine semantic equivalence while the Scratchpad matcher preprocesses the patterns into their individual special cases and at match time, performs structural matching. In addition, the Scratchpad matcher compiles several patterns (a rule set) at once, whereas the MACSYMA matcher compiles a single pattern at a time. This is elaborated on below.

The basic idea behind the MACSYMA matcher[50] is to remove the fixed part of a pattern from both the pattern and the subject expression by subtraction, division, etc. The leading operator in the pattern is examined and rules specific for choosing a child pattern variable are invoked causing that variable to be matched against some part of the expression thereby "fixing" that variable. Transformation of (sub)expressions to canonical forms may also be used for picking out coefficients. The fixed part of the pattern is removed and the process is repeated; there is no backtracking. The semantic capabilities of the matcher derive from both its built-in rules for sums, products, and exponentiation and from invoking the simplifier when some part of the pattern is removed. All pattern variables have predicates which must evaluate to true in order for a match of that variable to succeed. The MACSYMA matcher performs an analysis of the pattern and writes a Lisp program to perform the matching; error checking is done at the time of compilation and

the resulting program can be compiled for further speed improvements.

The Scratchpad approach[92] first generalizes the patterns so that their leaves are expressions, and then replaces the leaves with predicates testing for the original condition (e.g.,  $n = 2$ ). The patterns are examined for special cases such as  $a = 1$  and  $a = 0$  in  $a \sin(x)$ ; these become separate patterns. Additionally, knowledge of the internal representation of expressions is used to reorder terms so that structural matching can occur. From this “recognition tree,” a decision tree consisting of **ands**, **ors**, and “pseudo-codes” is generated, optimized, and finally compiled. The Scratchpad matcher was compared against two modified versions of the Reduce matcher. The first version collected all matches and then applied side conditions, while the second version applied any side condition involving just the single pattern variable at the time the match test was made. Jenks reports that the Scratchpad matcher was 9 to 2000 times faster than the first version, and 6 to 200 times faster than the second version on a set of integration patterns. Some of the improvement can be attributed to compilation; compiled code is typically an order of magnitude faster than interpreted code.

Semantic matching, while useful in some cases, is slow and somewhat unpredictable. Compilation of the patterns speeds up matching and is not limited to semantic matching. However, compilation is mainly useful in languages/operating systems that provide an environment for dynamic linking of code (such as Lisp).

### 3.6 Summary

This chapter introduced translations and notation libraries to handle conversions between the mathematical notation displayed on the screen and the linear syntax used by the underlying CAS. Translations and notation libraries help to solve problems with portability of the interface to different CASs and ambiguous mathematical notation. In the context of CASs, these are new ideas.

No new algorithms were presented in this chapter. Instead, the problem was defined and a number of techniques were presented that can be used to solve the problem. The techniques presented were divided into procedure-based techniques, rule-based techniques, and parser-based techniques. They are summarized below.

**Procedure-based** Procedure-based translations are ad-hoc in nature, require the user to learn a new language, and require an interpreted language or run-time linking/loading

in order to allow run-time addition and deletion of translations. Moreover, the translations often require structure pattern matching which can be awkward in many programming languages. On the other hand, procedure-based translations offer the full power of a programming language to perform the translation and require minimal implementation effort.

**Rule-based** Rule-based translations use rules and a pattern matcher to perform translations. The power of the pattern matcher determines the types of rules that can be used—rules that involve recurrent patterns or that involve semantic checking significantly increase the complexity of the pattern matcher and its running time. Rule sets are easily modified at run-time, although interactions among rules can make their application unpredictable.

**Parser-based** Parser-based translations represent a compromise between the two techniques mentioned above. On the one hand, they are relatively fast, are simple to implement using a parser generator, and provide some of the notational convenience of rules if a BNF grammar is used. On the other hand, extensibility, limited pattern matching capability, and limitations to a single rewrite strategy are problems with this technique.

Which technique is used depends upon speed, space, complexity, and power tradeoffs. As with many systems related issues, there is no best answer for all systems.

## Chapter 4

# Formatting and Drawing

Formatting an expression from a CAS can be divided into two parts: deciding what the overall format of the expression should be and placement of the individual characters within that format. Chapter 3 discusses the overall format; this chapter discusses the details of positioning the characters and rendering them on the workstation screen.

Formatting is accomplished by associating with each subexpression a *bounding box* and positioning the bounding boxes appropriately. Drawing is typically accomplished by traversing the tree and rendering the characters at the leaves of the tree at the positions indicated by their boxes. The concept of a bounding box is old. It was used by Martin for his pioneering Symbolic Mathematics Laboratory[112] and has been used in most formatting systems including  $\text{\TeX}$ [98]. However, no algorithms for incremental updating of the bounding boxes have been published. Some attributes of bounding boxes, a drawing algorithm, and an optimal incremental update algorithm are presented in the next section.

The algorithm used for formatting should be extensible so that new mathematical notations can be added at run-time. Two different techniques that satisfy this requirement have been tried in CASs: procedure-based approaches associate a built-in or user-written procedure with every notation; primitive-based approaches use a set of primitives out of which a new notation is constructed. These and other techniques are discussed in Section 4.2.

This chapter concludes with several ways to format large expressions in addition to the standard idea of breaking the expression into several lines. Most of these ideas were used in MathScribe and have not been used in other systems. A new efficient algorithm for handling expressions that are split across several lines is presented.

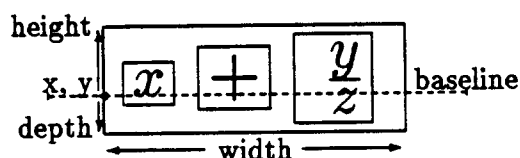


Figure 4.1: Attributes of a Box

## 4.1 Boxes and Drawing

Every expression is decomposed into rectangular *bounding boxes* (hereafter referred to as just boxes); one box per subexpression. Boxes abstract the information relevant to formatting a subexpression and are composed bottom-up. Every character is placed inside a box that describes its width, height and baseline (i.e., the point at which to horizontally align the box). Boxes are recursively constructed by positioning the children boxes and then determining the width, height, and baseline of the new box. The attributes of a box and other specifics of boxes are given in Section 4.1.1. The formatting algorithm does a post-order traversal of the tree and calls operator-specific procedures to format each box. Figure 3.8 shows the algorithm that MathScribe uses for formatting. Section 4.1.2 describes algorithms for drawing and Section 4.1.3 presents an algorithm for incremental reformatting.


### 4.1.1 Bounding Boxes

The important attributes of a box are its baseline, its  $x$  and  $y$  positions, and its size (width and height). Figure 4.1 shows these attributes. In general, the  $x$  and  $y$  position are relative to the parent<sup>1</sup>. Using relative positions allows boxes to be built from the bottom up in a single pass. It also allows the boxes to be shared which can significantly reduce memory usage (see Section 7.1). If the  $x$  and  $y$  positions refer to the upper or lower left corner of the box, then the *width*, *height*, and *baseline* are also attributes. To position boxes horizontally, this representation requires two passes over the children boxes: the first pass finds the maximum height of the children boxes and the second pass places the children boxes so that their baselines are all aligned appropriately. In a text formatting system, the second pass is necessary for left and right margin justification anyway. A slightly more efficient representation is for  $x$  and  $y$  to refer to left edge of the baseline of the box and have

<sup>1</sup>Another alternative is to make the first child box relative to the parent and to make succeeding child boxes relative to the previous child box. This representation offers opportunities for optimization of incremental update of linear formats—variably sized delimiters complicate this optimization.

*width*, *height* (amount above baseline), and *depth* (amount below baseline) as attributes. This allows boxes to be positioned horizontally in a single pass. Both T<sub>E</sub>X and MathScribe use the second representation.

MathScribe uses only one font size (the smallest one readable) in order to maximize the information on the screen. If more than one font size is desired, it is necessary to calculate the correct font size on the downward traversal of the tree. The algorithm in Figure 3.8 is easily modified to handle multiple font sizes by adding a `fontSizeDelta` list attribute (one entry per child) to the `Operator` data structure and incrementing an additional parameter of `ExprToBox` (`fontSize`) by this attribute.

In general, boxes do not overlap with other boxes. There are a few exceptions however. One exception is the “root of” box: the radical box  overlaps with the radicand box. Figure 4.6 shows some examples of formatting procedures.

It is both convenient and efficient to merge the box data structure with the data structure used for parsing, selection, translation into the algebra system form, etc. The added fields include the children of the box, the font to use to draw the box, and the character to draw. A complete list of the fields of the merged box data structure is given in Section 7.1.

### 4.1.2 Drawing

Given that the boxes are formatted as described above, rendering the expression consists of a simple depth-first traversal of the tree, calculating the absolute  $x$  and  $y$  positions of the boxes from the relative  $x$  and  $y$  positions by adding the children's positions to that of the parent's. There are a few simple optimizations that can be made. One is to never draw a box that is not visible. Another is to order the boxes from left-to-right (top-to-bottom) when possible. This allows for simpler testing of visibility and early termination of the loop. The procedures shown in Figure 4.2 assume that a drawing procedure is associated with each box. Associated with most leaves is a `CharacterDraw` procedure, although variably sized characters such as large parentheses use a different drawing procedure. Internal nodes in the tree generally use either the `DefaultDraw`, `LeftToRightDraw`, or the `TopToBottomDraw` (not shown).

The `Draw` procedure demonstrates another important optimization: double buffering. An early prototype of MathScribe drew directly to the screen by first clearing the



```

void Draw(Window *window, Box* box)
// Draws 'box' into 'window' by creating an invisible bitmap the size
// of the window, drawing into the bitmap, and finally copying the
// bitmap to the window (ie, double buffering is used).
{
    Bitmap bitmap(window->width, window->height); // creates blank bitmap
    Rect clipRect(0, 0, window->width, window->height);
    box->operator->draw(bitmap, 0, 0, box, clipRect);
    window->CopyFrom(bitmap);
}

void DefaultDraw (Bitmap& bitmap, int x, int y, Box* box, Rect& clipRect)
// Draws 'box' at (x, y) onto 'bitmap' using the clipping rectangle 'clipRect'
{
    for (Box* child = box->children; child; child = child->next) {
        int xChild = x + child->x;
        int yChild = y + child->y;

        // If the box is at least partially visible, draw it
        if (!(clipRect.offLeft(xChild, yChild) ||
            (clipRect.offRight(xChild, yChild) ||
            (clipRect.offTop(xChild, yChild) ||
            (clipRect.offBottom(xChild, yChild))))))
            child->operator->draw(bitmap, xChild, yChild, child, clipRect);
    }
}

void LeftToRightDraw (Bitmap& bitmap, int x, int y, Box* box, Rect& clipRect)
// This proc is optimized for when the boxes are ordered from left-to-right
{
    Box* child = box->children;

    // Scan until a portion of the box is visible
    while (child && clipRect.offLeft(x + child->x, y + child->y))
        child = child->next;

    // Draw the boxes that are at least partially visible
    while (child) {
        int xChild = x + child->x;
        int yChild = y + child->y;

        if (clipRect.offRight(xChild, yChild)) return;
        child->operator->draw(bitmap, xChild, yChild, child, clipRect);
        child = child->next;
    }
}

```

Figure 4.2: Drawing Algorithms

window and then drawing into it. This was done after every character that was typed and caused a very noticeable flash. The flash is avoided by drawing to an off-screen image and copying that image to the window: only those bits that are different change. Not only does this avoid the annoying flash, but it *appears* faster because the screen update operation (copy) is quicker than the drawing. Most interfaces now use double buffering.

### 4.1.3 Incremental Update

It is important to distinguish between syntax and semantics in the context of expression editing. The syntactic portion of input involves building the abstract syntax tree that underlies the display. The semantic portion consists of formatting the display. This requires computing the attributes shown in Figure 4.1. These computations can be done after the entire abstract syntax tree is built. However, it is far more efficient to avoid building the abstract syntax tree and instead, incrementally update the box attributes of the expression being modified.

A general incremental update algorithm is simple provided the following constraints are satisfied.

- The font size to use for the current box is known.
- The width and height of all of the children are either known or are a function of the width and height of known children. For example, the size of a left parenthesis is determined by the size of the box it surrounds (one of its siblings).
- The width and height of the current box is a function only of the children boxes.
- The position of the children relative to the current box is a function only of the children box's sizes and the "type" of the current box (e.g., quotient).

If a subtree is changed and the constraints above hold, then the only attributes that can change are those given below.

- If the *font size* of the current box changes, then *all* of the attributes of the current box and all of its children (recursively) must be reformatted. The reformatting can be optimized if the number of font sizes are limited<sup>2</sup>; reformatting stops when the

---

<sup>2</sup>Typically, only two or three font sizes are used with smaller fonts used in exponents, subscripts, limits, etc.

new font size equals the old font size. This can occur when the original size either increases or decreases.

- The *width*, *height*, and *depth* attributes of the current box along with the  $x, y$  attributes of siblings.
- All of the attributes of the ancestors of current box along with the  $x, y$  attributes of their siblings.

Thus the only boxes that need to be reformatted are (potentially) all of the children of the current box and all of the boxes that lie along a path from the current box to the root of the tree. The algorithm is given in Figure 4.3. An optimization incorporated in Figure 4.3 involves recognizing when the size of a new box (i.e., the *width*, *depth*, and *height* attributes) does not change. If this occurs, then the ancestors will not change, so we can terminate the traversal early. Also, only this subtree needs to be redrawn.

Although these constraints are normally met, some layout forms can cause problems. For example, two possible ways to lay out a definite integral box are:



In the first case, the limits of integration are considered part of the integral sign instead of part of the integral operator. If the integral sign varies in size with the size of the integrand, then we cannot determine the  $x$  and  $y$  positions of the limits until we know the size of the integrand.

Matrices can also be a problem if they are structured by rows or columns (i.e., as an array of arrays instead of a single  $m \times n$  array). If matrices are built from rows, then the conditions above are violated because the  $x$  and  $y$  position of elements in a row cannot be determined until we look at the width and height of all of the elements in the matrix, not just the elements in that row. However, a row/column representation can be more efficient during update because a change in one element of a matrix often has only a minor effect on the size of the matrix, perhaps requiring only a single row to change. The row representation also allows more sharing of the boxes (rows can be shared—see Section 7.1). The alternative representation as a list of  $m * n$  elements does obey the above properties at the expense of having to calculate the positions of all of its  $m * n$  elements if any one of its elements changes size.

```

Box* UpdateFormat(Box* box, int oldFontSize)
// 'box' has changed. It, its children and all of its ancestors are
// reformatted as necessary.
// Returns the root of the subtree that must be redrawn.
{
    // Change the size of 'box' and its children if 'oldFontSize' differs from
    // the current font size.
    Resize(box, oldFontSize);

    // Add parens to 'box' if its precedence is lower than its parent's
    // precedence. 'box' becomes a child of the paren box.
    box = AddParens(box);

    return UpdateAncestors(box);
}

Box* UpdateAncestors(Box* box)
// Reformat the box and all boxes up to the root of the tree (as necessary).
// Returns the root of the subtree that must be redrawn.
{
    // Save old dimensions -- if dimensions don't change, we can quit
    int oldWidth = box->width;
    int oldHeight = box->height;
    int oldDepth = box->depth;

    box = box->operator->format(operator, box->children);

    // If dimensions haven't changed, then we can stop
    if (oldWidth == box->width &&
        oldHeight == box->height &&
        oldDepth == box->depth) return box;

    // Check for root of tree -- if at root, we can stop
    if (box->parent == nil) return box;

    return UpdateAncestors(box->parent);
}

```

Figure 4.3: Incremental Formatting Update Algorithm

In both of these cases, the algorithm in Figure 4.3 continues to work as long as the integral and matrix formatting routines take responsibility for setting the  $x, y$  position of their grandchildren. However, the early termination optimization cannot be used unless the children are marked “special” and the stopping test is skipped for special boxes.

Finally, context-dependent layout forms can be a problem. For example, suppose that quotients whose numerators and denominators are small should be formatted as  $\frac{x}{y}$  normally, and as  $x/y$  if they appear in a script position. Then any change in a quotient’s position may require the quotient to be reformatted. For example, if the  $+$  in the expression  $x + 2 \cdot (t - \frac{x}{y})$  is changed to exponentiation, then the quotient should be reformatted into the linear format which the above algorithm will not do. It should be noted that context-dependent information requires propagation of contextual information which typically would be in the form of additional attributes. This obviously might require changes to the above algorithm in order to calculate and propagate the new attributes.

## 4.2 Formatting

Typesetting systems such as T<sub>E</sub>X use an elaborate set of rules and fonts to format expressions. However, because of the low resolution of displays (Section 1.3.3), it is our opinion that typesetting quality is neither necessary nor desirable. An informal experiment was conducted in which users were shown several expressions in both (readable) italic fonts and smaller regular fonts—the smaller fonts were preferred because more expressions could be seen at one time.

This section discusses four different techniques for extensible formatting of expressions: primitive-based techniques, procedure-based techniques, macro-based techniques, and constraint-based techniques. Each technique associates primitives, macros, etc., with a mathematical notation. Relevant work is presented within each classification.

Primitive-based formatting provides a fixed set of low-level functions for formatting and allows simple functional composition of the primitives. Access to the underlying Box data structure and to the window system is not provided. This insulates users from the implementation, but it also restricts their ability to define notations. Macro-based and procedure-based formatting extends primitive-based formatting by providing language-like features such as conditionals, variable assignment, and macro/function definition together with access to the Box data structure and window system. They are discussed separately

| Function   | Explanation   |
|--|---|
| SequenceForm[ $f_1, f_2, \dots$ ]                    | horizontally stack $f_1, f_2, \dots$  |
| ColumnForm[list, h, v]                               | vertically stack the elements in <i>list</i><br><i>h</i> : left justify, center, or right justify<br><i>v</i> : base line is centered, at top, or at bottom |
| MatrixForm[x]  | display $x$ as a matrix   |
| Subscripted[f[args], { $d_0, d_1$ }, { $u_0, u_1$ }] | arguments $d_0 \dots d_1$ are made subscripts<br>arguments $u_0 \dots u_1$ are made superscripts  |
| Prefix[f[x], h]                                      | print as $h x$  |
| Postfix[f[x], h]                                     | print as $x h$  |
| Infix[f[x, y, ...], h]                               | print as $x h y h \dots$  |
| PrecedenceForm[expr, n]                              | parenthesize <i>expr</i> with precedence $n$  |

Figure 4.4: Formatting Functions in Mathematica

because the exemplary systems are significantly different. Constraint-based formatting allows an equational specification of formatting relations.

#### 4.2.1 Primitive-based Formatting

Mathematica allows users to define their own output form for an operator. Mathematica calls the function *Format* to format an expression. Hence users can define output formats for their own functions or special cases. To assist users, Mathematica provides a number of built-in formatting functions; the relevant ones are shown in Figure 4.4. An example of their use involves printing integrals. In Mathematica, integrals that cannot be evaluated return the original answer and print as `Integral[expr, var]`. Integrals can be made to print more nicely (on an ascii terminal) by defining a format:

```
integralSign = ColumnForm["'", "[", "|", "'", "]", "'"],
                Center, Center]
Format[Integral[expr_, var_]] :=
    SequenceForm[integralSign, " ", expr, " d", var]
```

This results in the following output:

```

/
[      g[x]
Integrate[f[x]+g[x]/2,x]= | f[x] + ---- dx
                          ]      2
/
```

The same idea can be applied to handle definite integrals.

Mathematica's formatting functions can be used to construct a large number of mathematical notations. They are designed for character-oriented display devices and cannot handle delimiters, lines, or symbols whose correct size is a function the operands. For

example, the integral format above cannot be modified so that the size of the integral sign is a function of the integrand.

SMP is similar to Mathematica in its ability to allow users to define output formats by assigning to a special function associated with desired function. However, SMP has only a single output formatting function `Fmt` whose general form is:

```
Fmt[{{x1, y1}, {x2, y2}, ...}, expr1, expr2, ...]
```

The  $\{x_i, y_i\}$  pairs are the horizontal and vertical offsets for  $expr_i$  in units of the largest box. The pairs can also be given by a form that resembles Mathematica's patterns. Below are two examples of SMP's formatting (using Mathematica's notation for the sake of clarity):

```
integralSign = Fmt[{{0,2}, {0,1}, {0,0}, {0,-1}, {0,-2}},
                  "/", "[", "|", "]", "/"]
Fmt[{{x_?OddQ} : {x,0}, {x_?EvenQ} : {x,1}}, a, 2, b/c, 3, d, 4, f]
```

The second example prints as:

```
  2  3  4
   b
a - d f
  c
```

SMP passes the index of the argument to the pattern. If an index is not specified for an argument, the argument is printed immediately to the right of the last printed expression. Notice that the exponents of `a` and `d` are not correctly positioned; this is because `Fmt` aligns boxes with equal offsets.

In addition to `Fmt`, SMP also has a function `Prsize[expr]` that returns the size (width, height) of the box for an expression as a list. By combining `Fmt` and `Prsize`, it is possible to build output forms that Mathematica is not capable of building. For example, the following draws a line over `expr`:

```
line[expr_] := Fmt[{}, Repl["-", Prsize[expr][[1]]]]
overbar[expr_] := Fmt[{{0,0}, {0,1}}, expr, line[expr]]
```

`Repl` is a general SMP iterative function that repeats the first argument the number of times specified by the second argument. Because `Prsize[expr]` does not return information about the baseline's position, it is not possible to correctly draw large delimiters such as “}” because they would be centered incorrectly. Modification of `Prsize[expr]` to return a triple (width, depth, height) would correct this problem.

The following primitives are similar to Mathematica's primitives, but provide the functionality needed for a bitmap display (e.g., font size changes) and backward translations.

Unlike Mathematica's primitives, these primitives assume that translations have already inserted delimiters. These primitives are also similar to TeX's atoms (Figure 4.7).

Many of these primitives have a first argument `Op` that indicates the operator that generated this form. `InternalOp` is used if the primitive is used as a meaningless internal node in the display (see Figure 4.5 for an example). Adding `Op` simplifies backward translations and is required if different operators use identical notation. It also simplifies insertions of space and parenthesis if they are not performed by the translations.

**Horizontal(`Op`,  $\{b_1, \dots, b_n\}$ , `deltaX`, `deltaY`)**

The boxes  $b_1, \dots, b_n$  are formatted from left-to-right so that their baselines are aligned. The initial box is placed at (`deltaX`, `deltaY`).

The white space between the boxes can be determined either by table lookup as in `TeX`, passed in as a parameter, or based on `Op`. If table lookup is used, then internal nodes must be assigned character codes because it is very time consuming to scan down the edges of internal nodes to find the character codes of edge leaves. The internal node's character information, like `deltaX` and `deltaY`, can be stored as part of the class information and not part of the box itself. If parameter passing is used, then two parameters should be used to accommodate asymmetric spacing around delimiters such as comma.

In order to correctly format expressions that contain variably sized delimiters, this procedure makes three passes over the children. The first pass computes the maximum height and depth of the fixed size boxes. The second pass builds the variably sized boxes (delimiters) using the height and depth computed in the first pass. The last pass computes the positions of the children and width, depth, and height of this box.

**Vertical(`Op`,  $\{b_1, \dots, b_n\}$ , `justification`, `base`, `baseIndex`, `deltaX`, `deltaY`)**

This is similar to `Horizontal` except that boxes are formatted from top-to-bottom. The additional parameters determine the type of justification (left, center, right) and how the baseline is determined (top, baseline, bottom of *i*th box). For example, quotients are formatted with

```
justification=center, base=bottom, baseIndex=2, deltaX=0,deltaY=-3
```

These settings shift the quotient line up a little from the baseline of the parent expression. Given the frequency of occurrence of quotients, it may be useful to provide

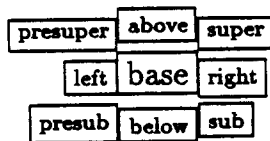


an optimized version for formatting only them. `Vertical` is useful for over and under bars, limits, etc.

Like the `Horizontal` primitive, three passes are used to handle variably sized boxes. The first pass computes the maximum width (instead of height and depth); the second and third passes are similar to the `Horizontal` primitive's second and third passes.

**Script**(Op, base,  $\{(b_1, p_1), \dots, (b_n, p_n)\}$ )/**Super**(Op, base, super)

`Script` is used for subscripts, superscripts, diacritical marks, limits, accents, etc. The base is formatted in the current font and the boxes in the script positions (given by the  $p_i$ s) are formatted in a smaller font if there is one. The general format is shown below.



There may be from one to eight children boxes given, and the  $p_i$ s should be distinct. Given the frequency of occurrence of exponents (superscripts), it may be useful to provide `Script` as an optimized version for formatting only superscripts.

**Tensor**(Op, base,  $\{(b_1, p_1), \dots, (b_n, p_n)\}$ )

`Tensor` is similar to `Script` except that the  $p_i$  are limited to being `sub` and `sup`; there can be multiple `subs` and `sups` and their order is important. The format for `Tensor` differs from `Script` in that the subscripts and superscripts do not overlap horizontally (e.g.,  $R_i^{jk}$ ). `TeX` formats tensors by allowing an empty base and using multiple base and subscript/superscript pairs. For example, the previous tensor expression is represented in `TeX` as `R_i {}^{\{jk\}} {}_1`.

**Matrix**(Op, m, n, align, open, close,  $\{c_{11}, \dots, c_{mn}\}$ )

This command is used for matrices and determinants. The  $m \cdot n$  elements are formatted into  $m$  rows and  $n$  columns, with each element centered, left, or right justified in the column as specified by `align`<sup>3</sup>. The two other boxes, `open` and `close`, are assumed to be variably sized delimiters or lines and grow to enclose the elements. The baseline is made to be half of the total height of the matrix.

<sup>3</sup>Mathematics texts usually right justify columns containing only numbers.

The  $x$  position of the elements cannot be determined until the maximum width of the elements in a column has been determined. The  $y$  position of the elements of a row are always identical and depend upon the  $y$  position of the previous row, the maximum depth of the elements of the previous row, and the maximum height of the elements of the current row.

#### Radical(Op, radicand, exponent)/Sqrt(Op, radicand)

Radicals are variably sized and must grow to contain the radicand. A few fonts, such as those used by  $\text{\TeX}$ , contain several differently size radicals and one extendible radical. However, most fonts do not contain more than one radical character; hence, this delimiter must be drawn for most font sets. The `exponent` is drawn in a smaller font size.

Given the relative frequency of occurrence of square roots, it may be useful to provide `Sqrt` as an optimized version for formatting only square roots.

#### HLine(width, extra)/VLine(height, depth, extra)

Two kinds of lines are needed: horizontal lines and vertical lines. Both kinds of lines are variably sized. They are used for quotients, determinants, and overbars among other notations. In many cases, such as quotient lines, the lines should extend slightly beyond their operands (parameters); this extension is specified by the `extra` parameter. In order to format and draw lines of the correct width, it is necessary to know the average thickness of "lines" in characters in a font.

#### Symbol(font, character)

A box for the given character in the given font is constructed. The width, depth, and height are all taken from font information. It may be useful to allow color specification also.

#### VariableSymbol(font, character, height, depth)

A character whose size depends upon `height` and `depth` is constructed. Examples of such characters include parentheses, braces, and the floor ( $\lfloor \rfloor$ ) operator. Many times (e.g., "{"), the `height` and `depth` of the character is made equal to the maximum of `height` and `depth` so that the character remains symmetrical.

variably sized symbols are constructed out of several font symbols. For example, a large "{ " is composed of the three fixed pieces:  $\{$ ,  $\}$ , and repeated pieces of  $\cdot$ .

Of the primitives listed above, only the leaf formatting primitives (`Line`, `Symbol`, and `VariableSymbol`) and `Radical/Sqrt` need special drawing functions. Only `Script`, `Tensor`, and `Radical` modify the font size.

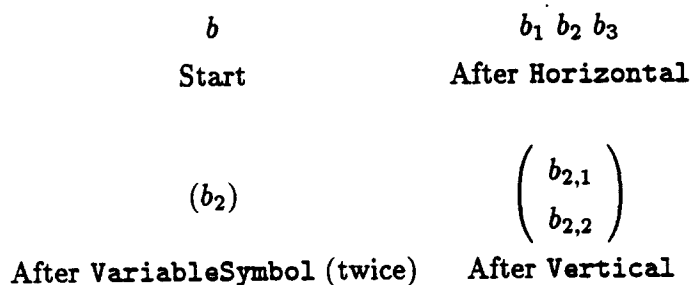
The leaf drawing primitives differ from the other primitives in that they are typically called from the scanner to create a token. The correct size for variably sized symbols may not be known by the scanner. A default size can be given; the size will likely be changed later when the parent box is constructed.

These primitives cannot reproduce every mathematical notation. However, with a rich font set (special math symbols, Greek letters, etc.), they are complete enough to reproduce all mathematical expressions in Spivak's book[152] on the  $\mathcal{A}\mathcal{M}\mathcal{S}$ - $\mathcal{T}\mathcal{E}\mathcal{X}$  macro package with the following exceptions:

- alignment of multiline expressions
- custom font size and style changes
- custom spacing/layout changes
- boxed formulas
- commutative diagrams
- arrows, braces, etc., above or below an expression that span the length of the expression (e.g.,  $\overbrace{x + \cdots + x}$ )
- dots across entire row or column of a matrix
- matrices whose columns are not all left justified, right justified, or centered
- dotless  $i$ 's and  $j$ 's when they are accented

As mentioned earlier, because of screen resolution limitations, exceptions such as small fonts and custom spacing may not be important. Other limitations, such as special characters, overbraces, etc., may not even be possible because of restricted display fonts.

There are other possible ways of specifying primitives. For example, "relative" positioning primitives such as `below` and `right` could replace the `vertical` and `horizontal` primitives.



```

CAS Form: Binomial[b2,1, b2,2]
Left Precedence: 0   Input Key: \b
Right Precedence: 0  Selection: Normal

```

Figure 4.5: WYSIWYG Formatting Example

It is not hard to imagine constructing an interactive tool to allow WYSIWYG construction of formats. In such a tool, the user constructs a format in a top-down manner: initially one box is displayed and the user replaces that box by a series of boxes listed in a palette/menu that corresponds to one of the possible formats. One of these boxes is selected and the process repeats. Additional information for each format such as the forward and backward translation, the keystrokes used for input, and precedence information must also be given. Figure 4.5 shows a sequence of steps used to derive a binomial coefficient notation for Mathematica. The translation for Mathematica is

```

Binomial[n,i]  $\longleftrightarrow$  Horizontal[{Binomial,
  VariableSymbol[SymbolFont, ‘{’, 0, 0]
  Vertical[InternalOp, {n, i}, center, bottom, 0, -1]
  VariableSymbol[SymbolFont, ‘}’, 0, 0]
}, 0, 0]

```

#### 4.2.2 Procedure-based Formatting

Both MathScribe and MathStation[115] have a procedure-based approach to formatting. Both rely on languages that allow dynamic loading of procedures: MathStation uses PostScript and MathScribe uses Lisp. In MathScribe, new formatting procedures are written with full access to the underlying Box data structure. An example of a MathScribe formatting procedure (rewritten into C++) is shown in Figure 4.6. Every mathematical notation has a formatting procedure associated with it. Precedence information must also be associated with the notation in order to determine if parentheses are needed.

Most notations can be formatted using procedures that are similar to the primitives

```

Box* FormatSuper(Box* box, Operator* op, CASForm* form, Box** children)
// Formats children as a superscript/exponent; 'box' is modified.
// The children are assumed to be the base, exponent operator, and exponent
// (in that order) with the exponent operator having 0 width and height
{
    // Add parens to 'base' if necessary (checks precedence info)
    Box* base = addParens(op, children[0]);
    Box* exp = children[2];
    int yExp = min(base->height, exp->height+exp->depth)/3
                - exp->depth - base->height;           // negative y-values are "up"

    /* set children's x, y, and parent values */
    base->x = base->y = 0;
    children[1]->x = base->width; children[1]->y = yExp; // values are irrelevant
    exp->x = base->width; exp->y = yExp;
    base->parent = exp->parent = children[1]->parent = box;

    /* set box's width, depth, height, etc */
    box->width = base->width+exp->width;
    box->depth = base->depth;
    box->height = exp->height - yExp;
    box->operator = op;
    box->CASform = form;
    box->children = children;
    return box;
}

```

Figure 4.6: A MathScribe Formatting Procedure

presented in Section 4.2.1. However, new procedures can be defined to handle custom spacing or context-sensitive formats (e.g., displaying quotients as  $x/y$  if  $x$  and  $y$  are tokens).

The drawbacks to procedure-based approaches to formatting are similar to the ones mentioned for procedure-based approaches to translation: an interpreted language or run-time linking/loading is required, which is not supported in most compiled languages today; the user must learn another language; and the user can make programming errors that corrupt memory, cause infinite loops, etc.

### 4.2.3 Macro-based Formatting

Macro-based formatting is similar to procedure-based formatting except that true procedures cannot be defined. Instead macros are defined and are either expanded when a statement that uses them is “read” or they are expanded when they are used. In the first

case, this makes macro definition ordering important. In the second case, running time and memory can be significantly increased. Macro-based systems typically do not allow access to the internal data structures and can also be thought of as primitive-based systems with an extension mechanism.

$\text{\TeX}$ [98] is an example of a very large and sophisticated macro-based formatting system.  $\text{\TeX}$  contains over 300 primitives and a macro language that allows the combination and definition of new control sequences. Because of this,  $\text{\TeX}$  is very extensible. In fact, plain  $\text{\TeX}$  consists of the primitives and about 600 new control sequences;  $\text{\LaTeX}$ [101] defines many more control sequences.

$\text{\TeX}$  is probably the most sophisticated formatter for mathematical expressions.  $\text{\TeX}$  is based on a model of boxes and *glue*. Glue is inserted as white space and has the ability to stretch and shrink. Glue is used in conjunction with *penalties* in order to break paragraphs into lines.  $\text{\TeX}$  does not break expressions across lines; this is left up to the user.  $\text{\TeX}$  generalizes the notion of a box to an *atom* when dealing with expressions. An atom consists of three fields: a nucleus, a superscript, and a subscript. Each field is either empty or consists of a character or (recursively) a math list. The thirteen different kinds of atoms  $\text{\TeX}$  uses are listed in Figure 4.7. The first seven of these correspond to character classes assigned to every character in a font. The character classes together with the font size determine the amount of glue used when constructing a *math list*—a list of atoms, glue and other  $\text{\TeX}$  items which are packed together horizontally. The complete rules  $\text{\TeX}$  uses for creating a math list are given in [98, Chapter 26 and Appendix G].

$\text{\TeX}$  uses three fonts sizes for formatting mathematics: text, sub/super-scripts, and “scriptscript” (sub/super-scripts within sub/super-scripts). Normal fonts used for  $\text{\TeX}$  must contain at least eight parameters that specify such values as the slant of the font and the interword stretchability/shrinkability. Fonts used for mathematics contain an additional 14 parameters that specify positioning of sub/super-scripts, numerator/denominators, etc. Fonts used by most windowing systems do not contain this information.

#### 4.2.4 Constraint-based Formatting

Formatting consists of positioning boxes which in turn consists of setting values for various attributes of the boxes. Setting these attributes can take the form of an equation, typically relating a box’s attributes to its children’s attributes (e.g.,  $b_w := c[2]_x + c[2]_w$ ).

| Name  | Explanation                   | Example  |
|-------|-------------------------------|--|
| Ord   | Ordinary character            | $x$  |
| Op    | Large operator                | $\Sigma$                                       |
| Bin   | Binary operator               | $+$  |
| Rel   | Relational operator           | $=$  |
| Open  | Opening delimiter             | $($  |
| Close | Closing delimiter             | $)$  |
| Punct | Punctuation delimiter         | $,$  |
| Inner | Delimited subformula          | $\frac{x}{y}$                                  |
| Over  | Overline                      | $\overline{x+y}$                               |
| Under | Underline                     | $\underline{x+y}$                              |
| Acc   | Accents                       | $\hat{x}$                                      |
| Rad   | Radicals                      | $\sqrt{x}$                                     |
| Vcent | box to be vertically centered | $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ |

Figure 4.7: T<sub>E</sub>X Atoms

This approach is used by Franchi-Zannettacci[61] for his high-level attribute grammar specification and by van Egmond, Heeman, and van Vliet in INFORM[166]. To date, no one has developed an extensible attribute grammar, so Franchi-Zannettacci's work is not applicable in this context. INFORM's parsing algorithm[82] is loosely based on Kaiser and Kant's algorithm[95] and is extensible. However, their implementation is similar to parser generators like YACC[93] in that INFORM pre-processes a grammar to generate code; hence, INFORM is not extensible either. Also, INFORM does not define a set of primitives that are callable at run-time. Instead, INFORM provides some useful functions such as `max_all` and `sum`, and requires users to write additional functionality in its base language (C++) when needed.

Constraint-based formatting requires solving equations to position the boxes and is likely to be slow unless the equations are compiled. This is because each box contains at least six attributes that must be computed, and *interpreting* the large number of equations that must be changed when the user makes a change to an expression is likely to be time consuming.

### 4.3 Large Expressions

CASs frequently generate large expressions and their ability to manipulate them is one of the main reasons for using a CAS. Mathematics texts use three techniques for displaying a large expression: breaking the large expression into several “lines,” elision of detail (using “...”), and renaming of subexpressions. With the exception of MathScribe, CASs use only line breaking when dealing with expressions that do not fit on a single line.

There are several problems with breaking expressions across lines, both from a technical point of view and from a user’s point of view. From the user’s point of view, the main drawback is the loss of two-dimensional notation: quotients, exponentials, etc., must be converted into a linear format if they do not fit on a single line. Another drawback is that it is hard to automatically break the expression in a meaningful way[98, pages 195-197][116]. Finally, very large expressions scroll off the top of the screen, necessitating another solution such as vertical scrolling. On the other hand, some expressions can be split so as to preserve their two-dimensional notation, and if they are small enough to fit in a window, line breaking allows them to be viewed in their entirety. From technical point of view, problems with line breaking include increased complexity, time, and memory usage. These problems are elaborated on in Section 4.3.1.

The remaining sections discuss other methods for displaying large expressions. The techniques discussed are designed for the display of expressions returned from the CAS and are not meant to be used during entry or modification of an expression (where the display is recomputed on a character-by-character basis) because a small change in the input could result in a disconcertingly large change in the display.

#### 4.3.1 Line Breaking

Breaking an expression across several lines is useful if the expression is not too large. A very large expression that is broken across lines will not fit vertically in a window. Unfortunately, without computing line breaks, it is not possible to precisely tell how large (vertically) the expression is. This section discusses several options related to breaking an expression across lines: surface-level versus deep-level line breaking, automatic versus manual line breaking, and first-fit versus optimal-fit automatic line breaking. Different techniques to determine the amount of indenting are also discussed.

Approaches that break an expression into several lines can be divided into surface-



level and deep-level approaches. Surface-level approaches allow line breaks only at places that do not destroy the two-dimensional structure of the expression. For example, if the root of the expression is a Plus operator, then a line break is allowed on either side of a “+” character. On the other hand, if the Plus is nested inside a Quotient, then the line break is disallowed because it requires breaking the quotient across several lines. A deep-level approach allows line breaks at any point. This requires every operator to have a linear display form. The obvious choice for the linear display form is to use the same output notation as used for entering the operator.

Line breaks can be determined automatically or manually by the user. Automatic line breaks can be determined on a line-by-line basis or by a global analysis of the expression. A simple left-to-right scan of the expression tries to put as much of the expression on the current line as will fit and is used by most CASs’ built-in pretty-printers. In order to determine if an expression fits on a line, the subexpressions must be formatted. If the expression does not fit on the current line, it might be necessary to reformat some or all of the subexpressions because the current expression’s display format must change to a linear notation. This is a recursive process that potentially can take an exponential amount of time (in the number of subexpressions). In practice, it is a very rare occurrence for a subexpression to be formatted more than twice because optimized formatting routines for linearly displayed expressions check to see if a line break is needed after formatting each subexpression instead of waiting until all of them have been formatted.

A “good” line break early in the expression might result in poor line breaks towards the end of the expression. To avoid this problem, a line breaking algorithm based upon  $\text{\TeX}$ ’s line breaking algorithm[98, Chapter 14][99] is used by Mathematica and a  $\text{\TeX}$ -Reduce Interface implemented by Antweiler, Strotmann, and Winkelmann[8]. The  $\text{\TeX}$  approach tries to find better break points by examining all possible break points in the entire expression instead of just possible break points on the current line. The modified  $\text{\TeX}$  algorithm works by adding glue and penalties between every subexpression. The glue and penalty values are based on the type of the subexpressions (see Section 4.2.3, character classes). Penalty values are also based upon other factors such as the parenthesis level; large penalty values indicate bad break points. The algorithm can be divided into three passes. The first pass calculates and inserts glue and penalty boxes between the subexpressions. The second pass calculates widths of the subexpressions and converts two-dimensional forms that do not fit on a line by themselves into linear notation. The third pass performs the actual

line breaking by considering all potential break points (penalty nodes). The amount the glue must shrink/stretch and the penalty of breaking at that point determine the *demerits* associated with a line break at that point. The sequence of line breaks that result in the smallest number of demerits is the one chosen. This sequence is efficiently computed by storing with each potential line break, a pointer to the previous line break that has the smallest number of demerits and the total number of demerits up to this point. At the conclusion of the algorithm, the optimal break sequence is given by the path indicated by the back pointers. Notice that subexpressions at the end of the expression can influence the way an expression is broken on the first few lines by contributing to the total demerits associated with the sequence. In order to reduce the number of possible break points, a user-settable tolerance value is usually provided. A line break is allowed at a certain point only if breaking the line at that point does not result in demerits exceeding the tolerance value. Larger tolerance values allow for more break points and hence, increase the running time of the algorithm. Smaller tolerance values do not allow many break points and can result in poor choices for line breaks.

Automatic line breaking is costly, requires more memory, and may not produce good break points. Timing tests reported in [8] indicate that the  $\text{\TeX}$  line breaking algorithm is five times slower than the same algorithm without line breaking. Formatting with line breaking requires more memory because of data structures used by the line breaking algorithm and because of a decreased amount of sharing—a subexpression that otherwise shares a `Box` with identical subexpressions cannot share that box if the expression must be split across a line. Finally, choosing good break points is hard. Figures 4.8–4.11 show the results of solving a general quadratic equation in several CASs with the line length reduced to force the expression to be printed on several lines. The results are not directly comparable because every CAS has a different syntax and internal representation which is reflected in the output. Nonetheless, the examples illustrate various breakpoints and the resulting displays. For example, Mathematica wastes vertical space in order to keep logical quantities such as quotients and square roots on a single line. Reduce on the other hand, puts as much as possible on the current line. The Maple example shows that reducing line length can (paradoxically) result in using fewer lines. This happens because the forced linear representation is occasionally more compact than the equivalent two-dimensional representation.

Automatic line breaking is not possible in the class of grammars accepted by

|  |  |
|--|--|
| <pre> {{x -&gt;       b      2      4 c       -(-) + Sqrt[--- - ---]       a      2      a       -----},       2 &gt; {x -&gt;       b      2      4 c       -(-) - Sqrt[--- - ---]       a      2      a       -----}} </pre> <p style="text-align: center;">Line Length = 30</p> | <pre> {{x -&gt;       b       -(-) +       a       Sqrt[--- - ---])       2      4 c       2      a       a       / 2}, &gt; {x -&gt;       b       -(-) -       a       Sqrt[--- - ---])       2      4 c       2      a       a       / 2}} </pre> <p style="text-align: center;">Line Length = 24</p> |
|--|--|

Figure 4.8: Line Breaking in Mathematica

|  |   |
|--|---|
| <pre> (d3) [x =       2       sqrt(b - 4 a c) + b       -----,       2 a       2       sqrt(b - 4 a c) - b       x = -----]       2 a </pre> <p style="text-align: center;">Line Length = 30</p> | <pre> (d8) [x =       2       - (sqrt(b - 4 a c)       + b)/(2 a),       2       x = (sqrt(b       - 4 a c) - b)       /(2 a)] </pre> <p style="text-align: center;">Line Length = 20</p> |
|--|---|

Figure 4.9: Line Breaking in MACSYMA

|   |   |
|---|---|
| $\{x = (-\sqrt{-4ac + b^2} - b) / (2a),$ $x = (\sqrt{-4ac + b^2} - b) / (2a)\}$ <p style="text-align: center;">Line Length = 39</p> | $\{x = -(\sqrt{-4ac + b^2} + b) / (2a),$ $x = (\sqrt{-4ac + b^2} - b) / (2a)\}$ <p style="text-align: center;">Line Length = 30</p> |
|---|---|

Figure 4.10: Line Breaking in Reduce

|   |   |   |
|---|---|---|
| $\frac{-b + (b^2 - 4ac)^{1/2}}{a},$ <p style="text-align: center;">Line Length = 30</p> | $\frac{-b + (b^2 - 4ac)^{1/2}}{a},$ <p style="text-align: center;">Line Length = 24</p> | $\frac{(-b + (b^2 - 4ac)^{1/2})}{a},$ <p style="text-align: center;">Line Length = 20</p> |
|---|---|---|

Figure 4.11: Line Breaking in Maple

many parsers including the MathScribe parser (Sections 4.1.3 and 5.4) because changing an expression can result in a change of display of an expression to the “left” (internally) of the current subexpression. For example, suppose we have  $\frac{\text{some-large-expression}}{\text{another-large-expression}}$  and typing one more character in the denominator causes the denominator to be split across lines. This in turn causes the quotient to lose its two-dimensional form, which in turn causes the numerator to have parentheses wrapped around it. The increase in size of the numerator may now force the line breaking algorithm to split the numerator apart. Hence a change in an expression to the right of a legally parsed expression may change its format (semantics of the parse) and thus is not admitted by most parsers. One way to circumvent this restriction is to consider automatic line breaking as a command, one that is invoked either manually

by the user or automatically at the end of parsing and formatting. This nonincremental approach can severely degrade responsiveness because it is invoked after every character that is typed.

If line breaks are allowed, there may be several boxes associated with each subexpression (one per line). This change affects formatting, selection, and display. Line breaks do not affect parsing if they are considered white space and are appended to tokens. Two different approaches to incremental handling of line breaks are presented. Neither approach has been implemented, so their relative merits are not known. The first approach uses several boxes per subexpression. The second approach formats the expression as if it were on one line and then splits that line during display according to the break points. The second approach is explained in Section 4.3.5 on Line Cutting. Allowing line breaks and multiple boxes per subexpression affects the formatting procedures as follows.

1. An inherited attribute `linearNotation` should be computed and passed into the parsing routines. If only surface-level formatting is permitted, then line breaks occurring in tokens in which `linearNotation` is false should be rejected.
2. If the procedure is used to format tokens or characters, then multiple boxes are used if the token or character has a line break appended onto it. The box for the line break has zero width, depth, and height.
3. If the procedure is used to format an internal node, then new boxes are constructed whenever a child with  $n > 1$  boxes is encountered. The first box is considered part of the current line and affects its width, depth, and height. Boxes  $2 \dots n - 1$  are on lines by themselves and remain unchanged. The last ( $n$ th) box begins the new "current" box and provides initial values for its width, depth, and height. The  $2 \dots n$  boxes of the child are positioned at  $(0, 0)$ ; a top-level routine positions the resulting root boxes at the appropriate indentation level and  $y$ -offset.
4. If the procedure is used to format an expression into a nonlinear notation and deep-level formatting is permitted, then if a child contains more than one box, the expression must be converted into linear notation and reformatted.

Delimiters whose size is a function of the size of the operands (such as parentheses) should probably continue to be a function of the operands even though this may result in "extra" white space. The waste occurs if there is a tall expression on one line and not on the

$$\begin{array}{cc}
 a \cdot b \cdot c & a \cdot b \cdot c \\
 \cdot(d + e) & \cdot(d + e) \\
 & +f + g
 \end{array}$$

Figure 4.12: Indentation

next line—tall delimiters result in both lines being tall. There does not appear to be any consistent treatment of parentheses in mathematical texts.

The above changes do not provide for automatic indentation. As with automatic line breaking, automatic indentation is not possible in the class of grammars accepted by many parsers because expressions to the right of a parsed expression (line) may change the indentation level. An example is shown in Figure 4.12. The second line is indented one level in the example on the left but must be indented two levels in the example on the right because of the line that follows it. Unlike line breaking, indentation is relatively cheap and can be performed non-incrementally after formatting is complete by setting the  $x$  and  $y$  coordinates of the root's line boxes. If automatic line breaking is used, indentation must be made a part of the line breaking procedure's decision process and cannot be done after formatting.

Mathematical texts do not appear to use any consistent rules for indentation if the expression is more than two lines long; two line expressions often (but not always) have their second line (almost) right justified. Different CASs indent differently as illustrated in Figures 4.8–4.11. Some rules that appear to be generally useful are given below.

- If an expression spans more than one line, the second through last lines are indented a level.
- If a line begins with an expression that is a subexpression of the expression on the line above or below the current line, it should be indented one level more than either of those two lines. This rule requires two passes: the first pass builds a tree of relative indentation levels and the second pass determines the absolute indentation. A simple variant is to indent one level for each level of nesting in the parse tree. This simple variant can easily lead to lines that are needlessly indented too far to the right.
- If an expression is inside some bracketing operator such as parentheses, then subsequent lines should begin slightly indented from the opening delimiter *if* the opening delimiter is close to the left edge of the expression.

- If an expression contains several relational operators such as =, the operators are aligned.

In order to accommodate multiple boxes per subexpression, the incremental update algorithm in Figure 4.3 must be changed to stop when *all* of the boxes remain unchanged. Formatting can be optimized by specifying a range of children boxes to reformat since changes are typically limited to the line on which they occur (variably sized delimiters are an exception).

The display procedures can be modified in either of two ways. One way is to change the  $(x, y)$  position passed as a parameter to be a list of pairs, one per line. A pointer to the current line's  $(x, y)$  position is incremented when a new line is reached (when `child->x >= child->next->x`). The drawback to this approach is that either a new list must be created for each call to a child or the current list must be explicitly incremented and decremented before and after each call; with a single box per subexpression, the new (single) values were created on the stack via parameter passing. The modified version of `LeftToRightDraw` of Figure 4.2 is given in Figure 4.13. The other way to modify the display procedures is to write an extra loop at the top level and display a single line at a time. This method requires several traversals of the tree (one per line) or the addition of pointers from the parent to the child indicating, for each line, the first child of the line and the subbox corresponding to that line<sup>4</sup>.

The structured selection paradigms of Section 6.1 must be modified to handle multiple boxes. Probably the most intuitive approach is to think of the subexpression as being on one line that is divided into several boxes. If more than one box is selected, then all boxes should be highlighted. The highlighting algorithm needs to deal with at most three rectangles: one for the first line, one for the second thru next to last line, and one for the last line.

#### 4.3.2 Elision

Elision is frequently used to suppress irrelevant parts of an expression when a pattern in the expression is apparent. Elision can be performed automatically or under user control. User controlled elision requires the ability to select contiguous subexpressions.

---

<sup>4</sup>Alternative representations exist including a more space efficient representation that utilizes a back pointer from a node to its containing box. However, back pointers prevent sharing which can dramatically reduce space usage (see Section 7.1).

```

Rect* MultiLineDraw(Bitmap& bitmap, Rect* boxes, Node* node, Rect& clipRect)
// This proc assumes that the children boxes are ordered from left-to-right.
// A pointer to the new "current" line box is returned.
{
    Node* child = node->children;

    while (child) {
        if (child->next) {          // multiple lines
            // for each line that the child is on, compute its x,y position
            for (lines = child->boxes, p= boxes;
                lines;
                lines=lines->next, p=p->next)) {
                p->x += lines->x; p->y += lines->y;
            }
            Rect* line = child->operator->draw(bitmap, boxes, child, clipRect);
            // reset x,y position
            for (lines = child->boxes, p= boxes;
                lines;
                lines=lines->next, p=p->next)) {
                p->x -= lines->x; p->y -= lines->y;
            }
            boxes = line;
        } else {                    // single line (optimization)
            child->operator->draw(bitmap, boxes->x+lines->x, boxes->y+lines->y,
                child, clipRect);
        }
        child = child->next;
    }
    return boxes;
}

```

Figure 4.13: Modified Drawing Algorithm for Line Breaking

Early versions of MathScribe used automatic elision. The user could set two variables, one controlling the depth at which subexpressions are elided and another controlling the width (number of subexpressions) at which subexpressions are elided. Other possibilities include pixel-based width and height settings or percentage-of-window-based settings. Automatic elision was dropped from MathScribe because it was found to be not very useful. In particular, large expressions tended to be wide, not deep, and there was not a good way to choose which subexpressions should be elided. MathScribe does allow user controlled elision (and expansion) via the mouse. Figure 2.6 on page 33 shows an example of this.

In order to implement elision, some extra storage must be used so that the elided parts of an expression can be expanded. In an object-oriented implementation based upon



display formats, a new class for elided formats can be derived from a standard base class. The new class stores the extra information in a new instance variable. An easy and time efficient value to store is a list of the boxes that were elided. This allows easy expansion and calculation of the CAS form of the expression.

An optimization discussed in Section 7.2.3 is partial formatting—formatting only the part of the expression that is visible. This optimization is particularly useful when used with automatic elision. However, as mentioned in that section, partial formatting does not work well if horizontal scrolling (with scrollbars) of expressions is also provided.

### 4.3.3 Renaming

Renaming is useful when a subexpression appears repeatedly in an expression. If a repeated subexpression is given a (short) name, then replacing the subexpression by that variable will decrease the size of the expression. An example of renaming is shown in Figure 2.7 on page 34.

Renaming is implemented by a simple depth-first search of the expression's tree and checking at each node to see if that node's value is the same as the node that is being renamed. If a DAG is used, this test is a trivial pointer comparison. The tree can be updated after each change by the simple algorithm of Figure 4.3 or it can be updated by intertwining the update algorithm with the tree walk; a node is updated if any of its descendants have changed.

Some extra storage must be associated with each renamed variable because renamed variables must be expandable to their original value. As with elision, a new class for renamed variables can be derived from a standard base class in a DAG-based object-oriented implementation. The new field contains a pointer to the renamed subexpression. The restriction to a DAG structure is necessary because we want new occurrences of the variable (e.g., user entry of the variable name) to refer to a renamed variable, not a normal variable. An alternative approach is to use a separate table that maps the variable name to the renamed subexpression. Both approaches assume a unique name for each renamed subexpression.

If an expression containing a renamed variable is sent to a CAS for computation, it is unclear whether the expression containing the renamed variable should be sent or whether the entire (expanded) expression should be sent. Sending the expression with the

renamed variable produces results in terms of the variable, which may be more meaningful. Additionally, the computation probably requires less time because the expression being manipulated is smaller. However, not expanding the expression can produce incorrect results. This can occur if there is a functional dependency on any of the variables that are hidden by the renaming. For example, this happens in integration if the renamed subexpression contains the variable of integration. Incorrect results can also occur if there is an algebraic dependence between renamed expressions; this might lead to a hidden division by zero. The decision as to whether to expand the expression or not is probably best left to the user.

#### 4.3.4 Horizontal Scrolling

A simple way to view a large expression is to display the expression on a single line that is clipped to the window's boundaries. To view parts of the expression that are not visible, the expression is panned or scrolled as was done in the Reduce pretty printer[104] and MathScribe. Horizontal scrolling takes advantage of the interactivity of the workstation screen.

Horizontal scrolling is very simple to implement: when the user indicates that the expression should be scrolled, the  $x$  position of the root box is changed appropriately and the expression is redrawn. Horizontal scrolling does not effect the sharability of boxes, nor does it require any extra time to format the expression.

#### 4.3.5 Line Cutting

An alternative to line breaking is *line cutting*. Line cutting formats an expression as if were on a single line and then "cuts" that line into multiple parts, displaying each cut part on a separate line. For example, the large expression  $(x + y)/(x + y + z)^5$  might be displayed as:

$$\frac{x + y}{x^5 + 5x^4y + 5x^4z + 10x^3y^2 + 20x^3yz + 10x^3z^2 + 10x^2y^3 + 30x^2y^2z + 30x^2yz^2 + 10x^2z^3}$$

$$+ 5xy^4 + 20xy^3z + 30xy^2z^2 + 20xyz^3 + 5xz^4 + y^5 + 5y^4z + 10y^3z^2 + 10y^2z^3 + 5yz^4 + z^5$$

Line cutting is used by DERIVE[44] for hardcopy output (the screen display uses horizontal scrolling). Line cutting preserves most of the two-dimensional notation to which we are

accustomed.

One problem with line cutting is determining where a cut should be made so that characters are not split across lines, or more generally, so that meaningful units such as tokens, base/exponents etc., are not split apart. A notion similar to  $\text{\TeX}$ 's glue is probably useful, where operators with higher precedence have stronger glue and thus, are less likely to be split apart. DERIVE does not tackle this problem.

The following algorithm, which can also be used for line breaking, assumes that cut points (line breaks) have already been chosen. It works by formatting the expression as if it were one long *virtual line* and then mapping the virtual line into *segments* corresponding to the cut points (line breaks) during display. An inverse mapping is used to map the segments on the screen back into the virtual line during selection. Hence, the formatting procedures remain largely unchanged while the display and selection procedures must be modified. This algorithm avoids splitting a box into several boxes corresponding to the cut points (line breaks).

The algorithm associates with each segment, a pair of  $(x, y)$  increments that are added to the computed  $(x, y)$  coordinates during display. During selection, an inverse set of increments are added to the window coordinates to obtain the coordinates in virtual line. The display algorithms are modified as follows.

1. At the time a character is drawn, its virtual absolute coordinates are known. The correct segment is determined and the associated  $(x, y)$  increment is added. This can be optimized to avoid the extra addition and to make a single comparison by using the fact that every child's  $x$  position is positive. Hence, if a parent is beyond the current segment, then all of its children are beyond the current segment. The segments and the increments are made relative to one another. For example, if the expression has  $depth + height = 60$ ,  $width = 360$  and segments beginning at 0, 100, 190, and 290, then we would have

| $x$ segment | $(x, y)$ increments |        |
|-------------|---------------------|--------|
| 0           | 0                   | 0      |
| 100         | -100                | 60     |
| 90          | -90                 | 60     |
| 100         | -100                | 60     |
| 70          | unused              | unused |

When an  $x$  value is greater than the  $x$  value of the next segment, the  $(x, y)$  increments are added to the current  $(x, y)$  values and the segment index is incremented.

```

void LeftToRightDraw(Bitmap& bitmap, int x, int y, Box* box, Segment* segment)
// This proc assumes that the children boxes are ordered from left-to-right.
// Clipping has been omitted for brevity's sake.
{
    Box* child = box->children;

    while (child) {
        int xChild = x + child->x;
        int yChild = y + child->y;

        if (xChild > segment->x) {
            x += segment->xIncrement; y += segment->yIncrement;
            xChild += segment->xIncrement; y += segment->yIncrement;
            segment++;
        }
        child->operator->draw(bitmap, xChild, yChild, child, segment);
        child = child->next;
    }
}

```

Figure 4.14: Modified Drawing Algorithm for Line Cutting

2. If the “character” to be drawn is a line, then the line is divided into segments if it extends beyond the current segment. This does not occur if line breaks are used.
3. If a box is to be highlighted, then it is split into several boxes if it extends beyond the current segment (similar to lines).

The modified version of `LeftToRightDraw` of Figure 4.2 is given in Figure 4.14. The selection algorithms are similarly modified (the paradigms discussed in Section 6.1 still apply). Because every line has identical height and depth, it is particularly easy to map window coordinates into virtual line coordinates; the segment is given by  $p = \lfloor y_{screen} / (h + d) \rfloor$  and the virtual line coordinates are given by

$$(x_{screen} + segment[p].x, y_{screen} + segment[p].y)$$

In the example above, the values in the segment tables would be: (0,0), (100,-60), (190,-120), and (290,-180).

If this algorithm is used for line breaking, then a list of the line breaks must be maintained. The  $x$  position of any line break can be quickly computed to determine the segment boundary. The values in the segment tables depend upon the indentation algorithm used. For line cutting, the indentation is likely to be zero.

$$\begin{array}{r}
 2 \cdot \sqrt{3} \cdot \tan^{-1} \left( \frac{2x-1}{\sqrt{3}} \right) \\
 - \\
 \log(x^2 - x + 1) \\
 + \\
 2 \cdot \log(x + 1) \\
 / \\
 6
 \end{array}
 \qquad
 \begin{array}{r}
 2 \cdot \sqrt{3} \cdot \tan^{-1} \left( \frac{2x-1}{\sqrt{3}} \right) \\
 - \log(x^2 - x + 1) \\
 + 2 \cdot \log(x + 1) \\
 / 6
 \end{array}$$

(a) Delimiters on Separate Lines                      (b) Delimiters on Same Lines

Figure 4.15: Structural Line Breaking

One drawback to this algorithm is that every line has the depth and height of the entire expression, potentially wasting screen space. Unfortunately, in order to incrementally calculate each line's true depth and height, we must associate with each subexpression a list of boxes similar to the algorithm given in Section 4.3.1 on line breaking and modify the formatting procedures to propagate this information.

#### 4.3.6 Structural Line Breaking

Large expressions can be broken up and displayed on several lines according to their structure. If an expression is too long to fit on a line, it is split across several lines (one line per subexpression) with its connective delimiter displayed between subexpressions. The subexpressions are indented and if they are too large to fit on the remainder of the line, the algorithm is applied recursively. Displaying large expressions in this way results in a tree-like display. Charybdis[119], an output program developed in 1967 by Millen, displayed large expressions in this manner. Figure 4.15a shows an example.

Displaying large expressions in a tree-like manner highlights the structure of the expression and can sometimes make a large expression easier to comprehend. It also has a technical advantage over regular line-breaking by maintaining the one box per expression relationship. Tree-like display suffers from a number of drawbacks though.

1. It is unusual and takes some time to get used to.
2. It can waste a significant amount of screen space. For example, a sum of many short terms that does not fit on a single line uses many lines, each line of which contains only a single (short) term. This can be somewhat mitigated by displaying the connective delimiter on same line as the following term as in Figure 4.15b. If this is done, then

either delimiters should not be selectable or selection must handle multiple rectangles for contiguous subexpressions.

3. Deeply-nested subexpressions can be indented past the end of a line. Millen[119] does not describe how Charybdis handles this problem. This problem is similar to that faced by many computer program pretty-printers. There are several techniques that can be used when displaying deeply-nested subexpressions.
  - Do not indent subexpressions any further. This makes it very hard to see structure.
  - Shift the subexpressions back to the left margin and continue displaying them as before. In this case, selections can span several rectangles.
  - Use either elision or renaming to handle the deeply-nested subexpressions.
4. Tree-like display suffers the technical disadvantage of requiring backtracking if an expression must be converted to linear form because it does not fit on a line.

## Chapter 5

# Entering Expressions

This chapter focuses on mouse and keyboard input for mathematical expressions. The last section discusses some alternative techniques for input such as handwriting recognition systems.

Mathematics is a special type of structured text. Although there has been some work in dealing with mathematical notation, there has been a far greater amount of work dealing with the structured notation of programming languages. One important difference between the two domains is that for mathematical formatting, the display must be updated and formatted properly after every character that is typed in order to correctly display the expression—this requires parsing the expression after every character. For programming languages, updating the display is relatively trivial (similar to text editing); parsing is not usually performed until the user signals to the system to “accept” the latest changes. This difference has a strong impact on both the input model and the error correction strategy.

In programming environments, two types of input models are commonly used: structured input and textual input. A third input model, the token model[25], is related to the textual model and is rarely used. Some systems allow both textual and structural input.

Structure editors enforce correct syntax at all times by allowing the user to enter only valid constructs. Legal constructs are typically selected from a menu of *templates* or from keyboard equivalents. Early systems, such as the InterLisp editor[163], the Cornell Program Synthesizer[162] and ALOE/GANDALF[129] worked this way. Structural input at the lowest levels of the tree (such as expressions and variable lists) can be tedious, and these systems allow users to enter these constructs textually. The input is not interpreted

until either the user moves the cursor outside of the construct or the user signals the system to parse the input. Because structure editors maintain correct syntax, syntactic error correction is not an issue in these systems. This input model is easier to implement because it does not require a parser.

In structure editors, users must enter programs in a top-down manner, not in the left-to-right manner in which they are displayed. Also, it is sometimes difficult to make simple changes to a program because the intermediate stages of the change do not naturally conform to a syntactically correct program. Text-based models do not have these problems. They must however cope with syntactically incorrect programs. Text-based editors deal with incorrect programs either by tolerating the errors or by correcting the errors, automatically or with user-feedback.

The earliest text-based incremental editor was Magpie[147]. Magpie divides the program up into fragments and the incremental LL(1) parser operates only until it reaches a syntax error inside a fragment; the rest of the fragment remains uninterpreted (the error is tolerated).

The Poe Language-Based Editor[54] is a cross between the two approaches. In Poe, users type part of a construct (typically the first delimiter) and the editor automatically “corrects” the input so that it is a valid construct; the correction is undoable. For example, if the user types `if`, then Poe expands this to a `if-then` construct. Thus, the user enters the program in a left-to-right manner. Subsequent editing is done structurally, so that syntactic correctness is maintained. Poe uses an LL(1) parser and the error correction method of Fischer, Milton, and Quiring[53] discussed in Section 5.4.2 to expand/correct user input. Poe does not use templates; however, the dummy place holders are left-hand sides of productions so that users are very aware of the underlying grammar.

One approach to allowing both a structural view and a textual view of a program is illustrated by Syned[86]. Syned allows users to enter and manipulate programs structurally. Program fragments can also be entered textually by typing in a special text editor window. Modifications can be made textually by (structurally) selecting some part of the program and opening a text editor window on that part of the program—the abstract syntax tree is unparsed into text. When the user is finished, the parser converts the text back into an abstract syntax tree. Syned uses an LR parser modified so that there is a start point corresponding to each non-terminal in the grammar. This allows the parser to be started on program fragments. It appears that Syned does not correct or tolerate errors, although



it does allow a program to contain unexpanded non-terminals.

PSG[12] smoothly integrates a text-based model with a structure-based model. The program can be entered using templates or users can type in part of the program and signal PSG when to parse the newly entered text. Modifications (deletions and replacements) work in a similar manner. The modifications in textual mode do not need to be contiguous. PSG uses an incremental LL(1) parser that can both tolerate errors and provide possible corrections of the errors to users.

Experience with programming environments has shown that the structural model is useful, but not for infix expressions where it is very unnatural. Nonetheless, this is the model adopted by many editors that handle mathematics such as Star[154] and Edimath[138]. These editors mitigate some of the drawbacks of top-down development for expressions by considering all linear constructs such as addition and multiplication to be unstructured strings, thus allowing a more natural input. Notation that involves vertical motion such as division and exponentiation must still be entered in an unnatural manner. A drawback to entering linear expressions as a string is that syntactically meaningless expressions are accepted.

A variation on templates that preserves the structural model of input but allows infix expressions to be typed in a natural manner we have called *overlays*<sup>1</sup>. This method is used in MathStation[115], Milo[121], Theorist[19] in parts of MathScribe. Expressionist[18] also uses overlays for nonlinear constructs, but does not implement automatic selection making their use similar to templates. Overlays are discussed in Section 5.2.

The text model is used by MathCAD[113], MathScribe, and in a revised version of GANDALF[95]. Text-based input uses "standard" linear notation to enter expressions. For example, / is used for division and ^ is used for exponentiation. Because the expression must be formatted after every character, the expression must be parsed after every character. This rules out the approach taken by PSG and the Cornell Synthesizer of delaying the parse until users have finished entering an expression and presents some unique problems. The text model of input requires the system to deal with syntactic errors. MathCAD contains only single character prefix, postfix, and infix operators and this makes error correction trivial in MathCAD. Error correction/tolerance is more of an issue for MathScribe because CASs contain language constructs in addition to simple mathematical operators. Error

---

<sup>1</sup>This technique has not been described in the literature.

correction strategies are discussed in Sections 5.4.2 and 5.5.4.

An important requirement for a CAS interface is that users must be able to define new notations. Defining the notation and how it is displayed is the topic of Chapter 4. This chapter covers incorporating the new notation into an expression (i.e., allowing users to enter the new expression). For tree-based models, extending the input is merely a matter of substituting in the new tree and propagating the resultant changes to the display (Section 4.1.3). For text-based systems, new notation requires extending the language accepted. Most work on extensible languages was done in the late 60's and early 70's. Algol68[168], IMP[88] and ECL[169] are three examples of extensible languages. Extensible languages have faded in popularity, possibly for the following reasons:

- adding new syntax creates a new language whose meaning might not be clear to readers of the code
- syntax extension does not generally provide much additional power
- most grammars are not extensible

In mathematics however, notation is very important, and hence, extensibility is an important requirement of an underlying grammar.

The remainder of this chapter presents several alternative algorithms for handling input. The algorithms in Sections 5.3 and 5.5.2 are new. Although the algorithms can be categorized as being either tree-based or text-based, the model of manipulation presented to users does not necessarily follow from the underlying algorithm: a tree-based algorithm can have the properties of a text-based algorithm and vice-versa. The implication of each algorithm on the model presented to users are also discussed. The first algorithms presented are tree-based, the latter algorithms presented are text-based.

## 5.1 Templates

The template approach to input is the simplest approach to implement. Each template is self-contained and knows how to format its subexpressions (subtrees). Templates enforce correct syntax by allowing only syntactically correct templates to be placed at subtrees in the abstract syntax; delimiters such as `if` and `+` are not selectable. Templates force a top-down (prefix-like) development of the input that is unnatural for expressions.

| Action                    | Display             |
|---------------------------|---------------------|
| Initial expression        | $\square$           |
| Use plus template         | $\square + ?$       |
| Use times template        | $\square ? + ?$     |
| Enter $a$                 | $a \square + ?$     |
| Use exponentiate template | $a \square^? + ?$   |
| Enter $x$                 | $a x^{\square} + ?$ |
| Enter 2                   | $a x^2 + \square$   |
| Enter $b$                 | $a x^2 + b$         |

Figure 5.1: Entering an expression using templates

Figure 5.1 shows the sequence of steps necessary to enter the expression  $ax^2 + b$ . Templates are used by many WYSIWYG expression editors (e.g., Star[154], Edimath[138]) and by many structure editors (e.g., Cornell Program Synthesizer[162], ALOE/GANDALF[129]). MathScribe abbreviations work this way. Because templates are so unnatural for entering an expression, some structure editors handle expressions as special cases. For example, the Cornell Program Synthesizer allows users to type text at the expression level. The text is parsed when the user indicates that he is through typing and the text is either accepted or an error is raised. Delayed parsing is acceptable for a structure editor because no special formatting is associated with mathematical expressions; it is not natural in this context though.

Templates are typically selected from either a palette or from a menu using the mouse. Keyboard equivalents are usually provided for quicker access to the templates.

N-ary templates such as statement lists are a special cases and are handled in a variety of ways in various systems. For example, ALOE/GANDALF has an "extend list" command that adds another child to the n-ary node.

Extensibility is simple with templates if the system provides some way of allowing the user to add and choose new templates. The algorithm for updating the display does not change if the constraints listed in Section 4.1.3 are valid. Note that in addition to providing the formatting rules for the template, the user must also supply precedence and associativity rules so that the system can determine if parentheses are needed around the newly inserted template.

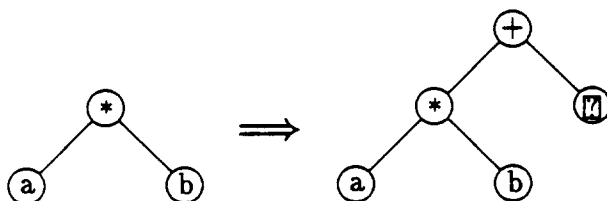


Figure 5.2: Example: applying the plus overlay

## 5.2 Overlays

Overlays are similar to templates. The difference is that instead of replacing the current node with the template, the current node becomes one of the children of the template. For example, if the current node is the expression  $ab$  and we apply the plus template, then  $ab$  becomes the first child of the plus template and we get  $ab+?$  as shown in Figure 5.2. This small change means that it is no longer necessary to enter an expression in a prefix-like manner. Instead, it can be entered in an infix-like manner if we interpret keys such as  $+$  and  $/$  as commands that correspond to the obvious templates. Input is further simplified if we adopt the convention that if nothing is explicitly selected, the text cursor implicitly defines a selection (e.g., the tree immediately to the left of the cursor). Figure 5.3 shows the sequence of steps necessary to enter the expression  $ax^2 + b$ . If the selected node is a dummy node, then overlays behave the same as templates.

In MathScribe, overlays are used when forms are selected from a menu. Milo[121] and MathStation[115] use overlays as their sole input model.

Extensibility for the overlays paradigm is very similar to extending the template paradigm. In addition to defining the template, the user must also designate one of the

| Action                           | Display          |
|----------------------------------|------------------|
| Initial expression               | $\square$        |
| Type $a$                         | $a$              |
| Type $*$ (use times overlay)     | $a\square$       |
| Type $x$                         | $ax$             |
| Type $\wedge$ (use expr overlay) | $ax^\square$     |
| Type 2                           | $ax^2$           |
| Select expression tree           | $\boxed{ax^2}$   |
| Type $+$ (use plus overlay)      | $ax^2 + \square$ |
| Type $b$                         | $ax^2 + b$       |

Figure 5.3: Entering an expression using overlays

dummy nodes in the new template as the substitution point. For an infix form, this would typically be the first operand of the infix operator. For an integral, it would be the integrand.

As with templates, *n*-ary operators require special treatment. For example, if the current node is  $a + b$ , and we apply the plus template, we must “merge” the  $a + b$  into the new plus node, not make it a child of the new node. Special treatment is also required for overloaded operators such as unary/*n*-ary – if we wish to use the same key command for both templates.

Overlays have a natural feel for prefix, infix, and postfix templates. However, it is less natural for other templates such as parentheses, integration, and programming language forms because overlays do not support the normal left-to-right entering of delimiters. Also, as with templates, changes to these forms can be awkward.

A slightly more sophisticated version of overlays does not require a single (static) substitution point in the template. Extra commands and/or context are used to determine which (dummy) node to use. Suppose that we have the expression  $a$  and we want to change the expression to  $\frac{?}{a}$ . With a standard division template, the first operand would be the substitution point and applying  $/$  to  $a$  results in  $\frac{a}{?}$ . If this were the only division template, we would have to go through some convoluted copying steps to get an  $a$  in the denominator. However, if there is a “insert last” command, obtaining  $\frac{?}{a}$  is simple. Alternately, contextual information can sometimes be used in choosing the substitution point. For example, the system can easily distinguish between  $|a$  and  $a|$ , where  $|$  indicates the text cursor. In the first case, the operand  $a$  is to the right of cursor and so it is natural that the overlay command places the operand to the right of the template’s delimiter. In the second case, the operand  $a$  is to the left of cursor and should be placed to the left of the overlay’s delimiter. Thus, applying  $/$  to the two cases above results in  $\frac{?}{a}$  for the first case and  $\frac{a}{?}$  for the second case. Ambiguous cases, such as  $a|b$  should probably be treated as applying the overlay to the expression to the left; an “insert last” command would handle the other case.

In this model, an extra parameter is passed into the overlay function. The parameter specifies the substitution point. In general, this might be an integer indicating the  $n$ th dummy node in the template as the substitution point, but in practice, it is more likely to be a boolean value indicating the substitution point should be the first or last dummy node in the template.

For delimiters that are used in more than one template, it is desirable to bind the templates to a single key if context can be used to choose one of the templates. A common

example of such a case is a delimiter that is used for both a prefix operator and an infix operator such as  $+$  and  $-$ . The infix template is used if there is a delimiter to the right of the selection (and we are inserting on the right).

As mentioned earlier, overlays allow users to enter infix, prefix, and postfix expressions in a left-to-right manner. It is desirable to make deletions act symmetrically to insertions (i.e., they should have a right-to-left action). This is easily done using the following rules that depend upon the type of the node to be deleted:

**operands** A dummy node replaces the operand node. An exception to this rule is when there is a template with the same delimiter that has fewer operands. For example, deleting the first operand of a *difference* template should result in a *minus* template. This exception is not related to the desire that deletion act in a “right-to-left” manner.

**delimiters** The child node corresponding to the substitution point of the template replaces the current node. This results in a right-to-left feel for postfix nodes, but for other nodes (e.g., parentheses), does not. However, these other types of nodes are either entered in a top-down manner or are prefix or infix nodes, and deleting their delimiter does not correspond to left-to-right input (i.e., the right dummy operand is what would normally be deleted).

There are several alternatives to the “unwrapping” approach that preserve the insert/delete symmetry. One alternative is to pick a non-dummy operand if the substitution point is a dummy operand. Another alternative is to only delete the node if all of the operands are dummy operands. Yet another idea is to replace an infix node with a dummy infix node; deleting the dummy infix node would use the unwrapping approach mentioned above. There are many more valid alternatives.

**dummy operands** As with deleting delimiters, the child node corresponding to the substitution point of the template replaces the current node. Because the deleted operand is a dummy operand, it is probably useful to choose some other (non-dummy) operand to replace the current node.

Like the template model, overlays enforces correct syntax at all times. However, if we allow the user to select delimiters, then it is possible to introduce incorrect syntax. For example, replacing an infix  $<$  with a postfix  $!$  results in incorrect syntax. Some possible approaches to handling this are listed below.

- Do not allow replacement that causes syntax errors.
- Introduce a “dummy” infix operator or operand, dependent on whether the arity of the operator decreases or increases. In the example above, we need to add a dummy infix operator whose left operand is a factorial expression. If we replace a postfix ! with an infix <, then we need to add a dummy operand.
- Selecting a delimiter can be viewed as selecting all of the corresponding operator’s operands. Applying a template is interpreted as substituting the operands into the template’s dummy nodes in some order (say, top-to-bottom, left-to-right). If there are more operands than dummy nodes, then the excess operands are not used in the template. If there are fewer operands than dummy nodes, then the resulting expression will contain some dummy nodes. As an example,  $a \boxtimes b$  becomes  $a \boxtimes b$  if we use the > template.

A “smarter” approach is to attach type-like information to each node so that substitutions can only occur on nodes with identical type fields. For example, `while` and `repeat` loops might have types (cond, expr) and (expr, cond) respectively. Substitution using type information allows a user to select the `while` and replace it with a `repeat` template and have the operands switch order.

- Allow the incorrect syntax by wrapping an error node around the syntactically incorrect subtree.

### 5.3 Overlays with Precedence

As a further step towards making expression input behave more like traditional programming language input, we can use precedence information to “float” the inserted templates to the appropriate level. The result is the same as parsing the equivalent string. This approach is described in detail by Kaiser and Kant[95] and is used in part of the structure-oriented editors in the GANDALF System[78] and the ISDS project[52]. An early version of MathScribe used this algorithm. Later versions of MathScribe use the incremental parsing algorithm described in Section 5.5.1. The latter algorithm proved to be a cleaner and simpler solution than this approach because of the large number of different “classes” of operators allowed in MathScribe. In all of these systems, templates or overlays are used

| Action             | Display          |
|--------------------|------------------|
| Initial expression | $\square$        |
| Type $a$           | $a$              |
| Type $*$           | $a\square$       |
| Type $x$           | $ax$             |
| Type $\wedge$      | $ax^{\square}$   |
| Type 2             | $ax^2$           |
| Type $+$           | $ax^2 + \square$ |
| Type $b$           | $ax^2 + b$       |

Figure 5.4: Entering an expression using a parser

for language-like templates; this technique is reserved for “expressions.” This is because of algorithmic limitations with this technique and the desire to maintain syntactic correctness.

Kaiser and Kant only discuss how to handle infix operators and parentheses. They suggest how the algorithm might be extended to other operators. The method presented in this section generalizes their technique so that parentheses and (many) language constructs are handled as part of a general technique, not a special case. As part of INFORM, Heeman[82] independently extended Kaiser and Kant’s algorithm. His approach is discussed at the end of this section.

Language-like templates, parentheses, etc., can be classified as being infix, prefix, postfix, or nofix depending upon their leftmost and rightmost delimiters: if a template requires operands on both its left and right sides, we classify it as an infix template; if it requires an operand only on its right side, it is a prefix template; if it requires an operand only on its left side, it is a postfix template; and if it does not require any operands on its left or right side (e.g., parentheses), we call it a nofix template<sup>2</sup>. The standard definitions of infix, prefix, and postfix operators are special cases of this definition. Infix, prefix, and postfix templates “float” to their proper position in the tree as described below. Nofix templates can never float and behave identically to regular overlays.

Figure 5.4 shows the sequence of steps necessary to enter the expression  $ax^2 + b$ . This differs from overlays in that it is no longer necessary to specifically select the expression tree  $ax^2$  as in overlays (Figure 5.2); the precedence relations define the appropriate selection in this case. Another example in which overlays and overlays with precedence behave differently is when we replace  $+$  with “ $\wedge$ ” in the expression  $a + bc$ . Using overlays results in  $a^{bc}$  and using precedence information results in  $a^bc$ .

<sup>2</sup>This terminology is adopted from MACSYMA[109].



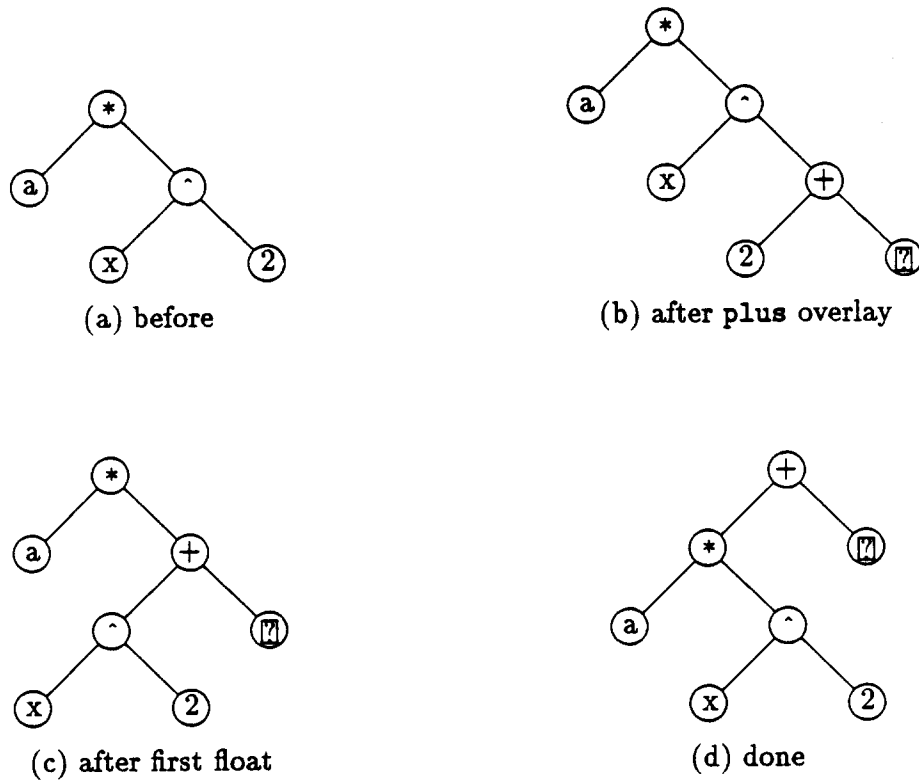


Figure 5.5: Example: floating the plus overlay

Overlays with precedence works by making the current node one of the children of the template (as with standard overlays). After inserting the child, the precedence relations between the child and parent are checked and the tree is “rotated” if necessary to preserve the *precedence constraints* (defined below).

As an example, consider what happens when we type the  $+$  in Figure 5.4. We begin with the tree in Figure 5.5a. Inserting the plus template results in Figure 5.5b. We now float the plus template to its correct level based on its precedence: children should always have a higher precedence than their parents. In this case, the plus template has a lower precedence than the power template, so we must float the plus template up a level in the tree. This involves “rotating” the child/parent pair as shown in Figure 5.5c. We must float the plus template up another level because it has a lower precedence than the times template. This rotation produces the tree in Figure 5.5d. The plus template is now the root of the tree and no further work needs to be performed.

This technique requires that every template have right and left precedences associated with all of its delimiters, with higher precedence implying a tighter binding. For

| Delimiter Type          | Left Precedence | Right Precedence |
|-------------------------|-----------------|------------------|
| prefix                  | maximum         | defined by user  |
| infix                   | defined by user | defined by user  |
| postfix                 | defined by user | maximum          |
| open delim [e.g., "("]  | maximum         | minimum          |
| close delim [e.g., ")"] | minimum         | maximum          |
| internal delims         | minimum         | minimum          |
| operands                | maximum         | maximum          |

Figure 5.6: Left and right precedence examples

example,  $*$  has higher left and right precedences than does  $+$ . Precedence information is also used to determine associativity (right associative delimiters have a higher left precedence than right precedence). The left and right precedence of a template is defined to be the left and right precedences of its leftmost and rightmost delimiters respectively. For prefix, infix, and postfix operators, the left and right precedence of the template is the same as the left and right precedence of their delimiters. However, for more complicated templates such as parentheses, integrals, and language constructs, there is more than one set of delimiter precedences. Left and right precedences for various kinds of delimiters are given in Figure 5.6. Notice that all delimiters that surround an operand in a template have minimum precedence when compared to their children. As explained below, this insures that expressions always remain children of the containing delimiters.

In this method, a correct tree always satisfies the following *precedence constraints* on the relation between a child and its parent.

1. The right precedence of a child template is greater than or equal to the left precedence of the delimiter to its right in the parent if one exists.
2. The left precedence of a child template is greater than or equal to the right precedence of the delimiter to its left in the parent if one exists.

When the tree is modified because of an insertion, deletion, or replacement, a *float* procedure is used to bring the tree back to satisfying these constraints. The float procedure consists of recursively checking the precedence constraints and performing either a rotate right or rotate left tree transformation depending on whether the first or second constraint is violated.

The rotate transformations for an infix template are shown in Figure 5.7. These transformations move the child template "C" above the parent template "P" and preserve

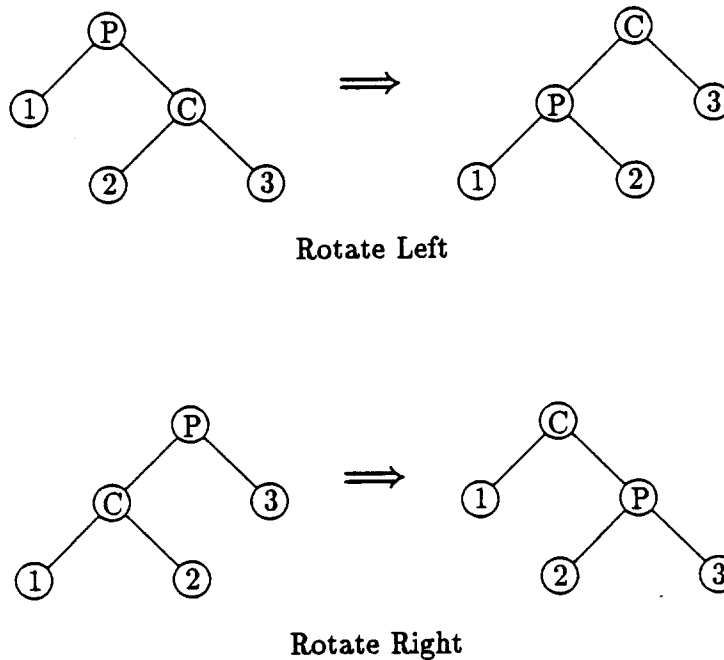


Figure 5.7: Infix rotate left and rotate right overlays

the order of the children of “P” and “C”. The rotations for prefix and postfix nodes are similar and are shown in Figures 5.9 and 5.10. They differ from the infix rotations because we must first walk up the tree to a point where the precedence constraints can be checked and the rotations must account for this extra “walk”.

The float procedure can be optimized by lumping together a chain of rotate left (right) transformations into a single transformation: the only changes to the tree are to the original parent’s *P* left (right) child, the child’s *C* right (left) child, and the stopping point’s *P* parent child (now equal to *C*). The intermediate nodes are not affected because the rotations preserve the order of the children and because after *C* has floated past the intermediate node, the old parent is restored. The optimized “chained rotate” for infix nodes are shown in Figure 5.8; the optimized rotates for prefix and postfix nodes are shown in Figures 5.9 and 5.10.

Insertions always occur at a leaf and float up to their appropriate level. The first step is identical to overlays: the template is wrapped around the (implicitly selected) leaf. The float operation is divided into three cases depending on the kind of template that is

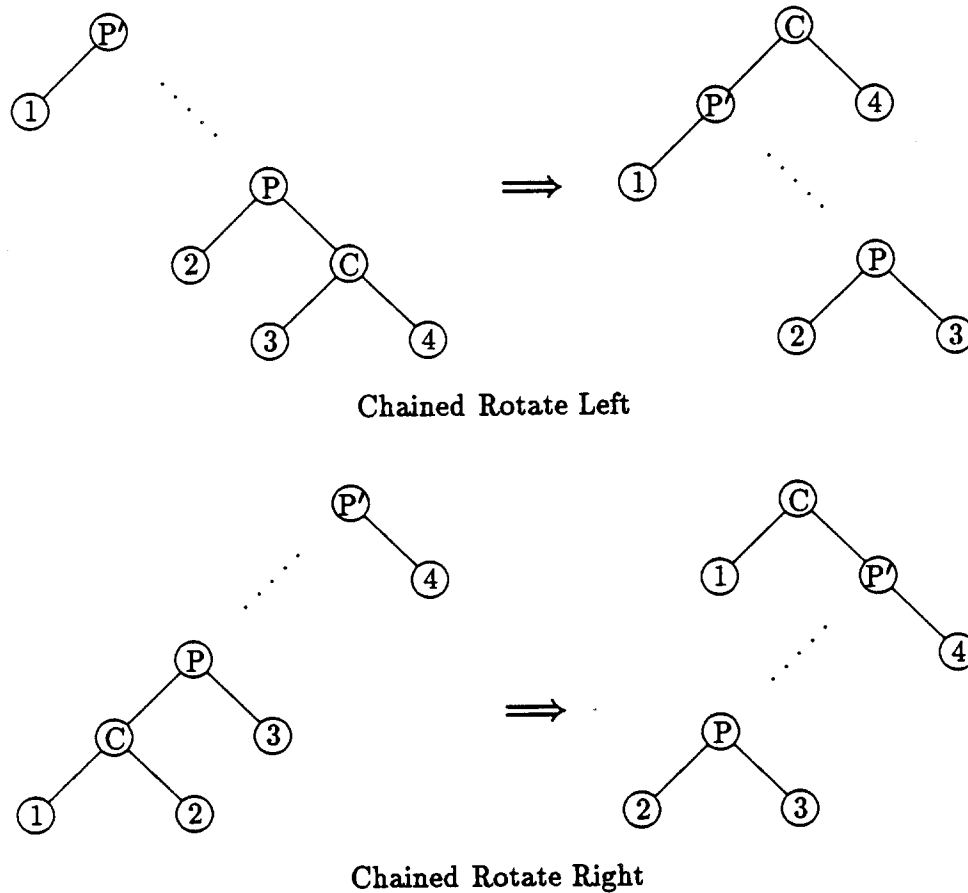


Figure 5.8: Chained infix rotate left and right

inserted:

**infix** The precedence constraints are checked and a rotate right or left is performed if either constraint is violated. The checking continues (recursively) on the parent. When the constraints are satisfied (or we are at the root), the template is at its correct precedence level.

This can be improved upon by using the chained-rotate left/right mentioned above and shown in Figure 5.8: we walk up the tree checking the constraint conditions and apply the appropriate chained-rotate transformation whenever the currently violated constraint is satisfied. As before, the checking continues (recursively) on the parent and we stop when both constraints are satisfied.

**prefix** This is similar to the infix case except that we first walk up the tree until the

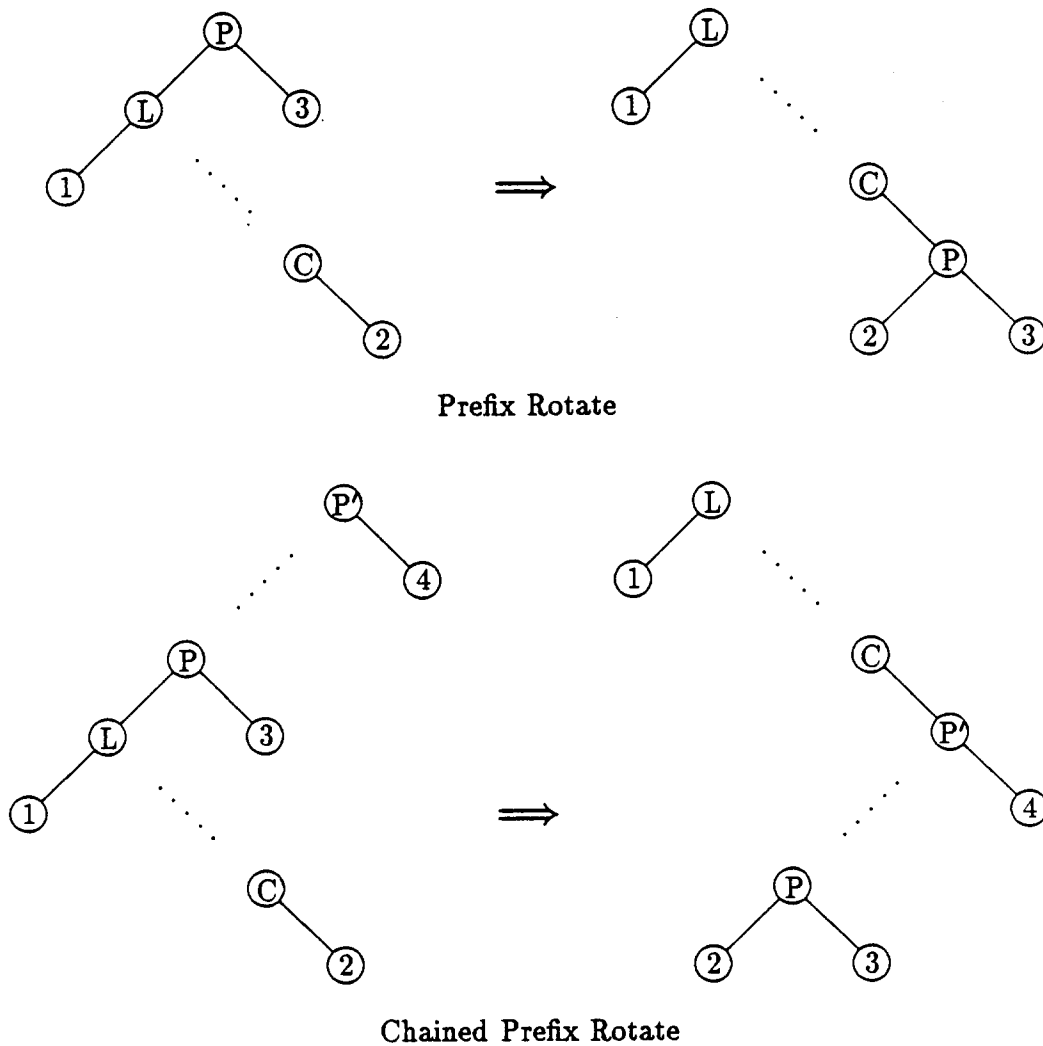


Figure 5.9: Prefix Rotate

template is no longer the rightmost child of its parent or we reach the root. In Figure 5.9, this node is labeled "L" (for leftmost node). Walking up the tree allows the right precedence of the (new) prefix template to be checked and floated appropriately. The first precedence constraint is checked and we perform the prefix rotate (right) as shown in Figure 5.9. The second precedence constraint does not need to be checked because prefix nodes should always have a maximum left precedence. As with infix nodes, the checking continues recursively on the new parent.

Similar to infix nodes, a chained-rotate optimization exists for prefix nodes (Figure 5.9). We first walk up the tree to find node "L". Then we walk up the tree

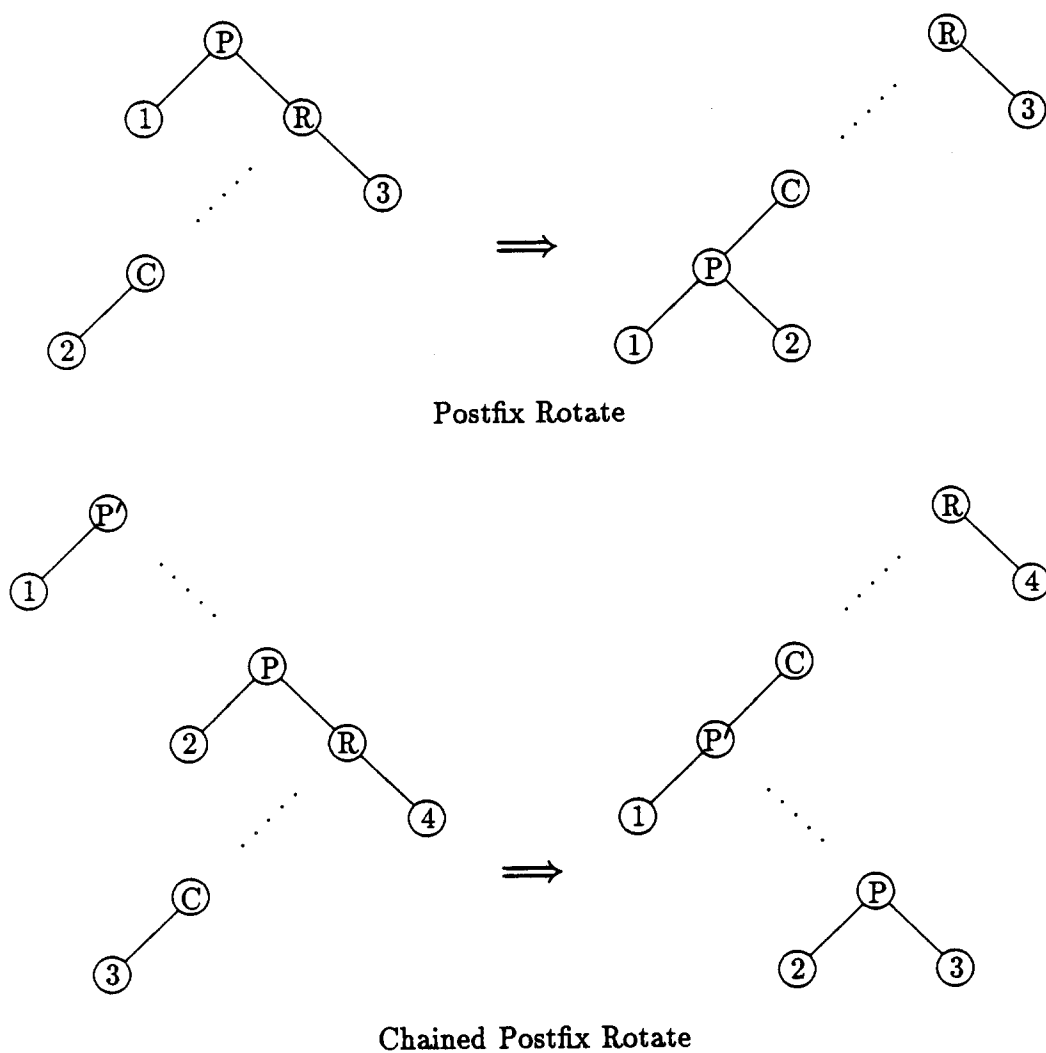


Figure 5.10: Postfix Rotate

checking the first constraint condition until it is satisfied (node  $P$  in Figure 5.9) and apply the chained-prefix-rotate (right) transformation. As before, after the rotation, the checking continues (recursively) on the new parent and we stop when both constraints are satisfied.

**postfix** This is symmetric to the prefix case: we walk up the tree until the template is no longer the leftmost child of its parent or we reach the root. Only the second precedence constraint needs to be checked. Figure 5.10 shows the appropriate transformations.

These transformations apply not only to standard operators, but also to language-like con-

structs such as *if-then* when it is interpreted as a postfix template.

The delete algorithm can be the same as for simple overlays. However, because overlays with precedence presents a more text-like model to the user, the delete operation should also present a text-like model. For example, if we start with the expression  $a \cdot b$  and type  $+$  after the  $a$ , we get  $a+? \cdot b$  in this model. With simple overlays, we get  $(a+?) \cdot b$ . Accordingly, deletion of the  $+$  should result in the original expression (under simple overlays, we do not get the original expression because the parentheses would remain). The delete algorithm of the previous section would result in either  $a$  or  $? \cdot b$ . In order to get the desired behavior, we need to look at the leftmost (rightmost) leaf node of the right (left) child and see if it is a dummy operand. This complication only affects infix nodes and the algorithm is:

1. If the leftmost leaf node of the right child is a dummy node, then the operator node is replaced with the right child and the left child replaces the dummy node. Note that for the trivial case of the right child being a dummy node, this simply results in replacing the operator node with the left child.
2. If the rightmost leaf node of the left child is a dummy node, then the operator node is replaced with the left child and the right child replaces the dummy node.
3. All other cases result in a syntax error and can be handled by not allowing the deletion, replacing the operator with a dummy infix operator (or some other default infix operator such as the Times operator), or by one of the other solutions discussed in the previous section.

As with simple overlays, incorrect syntax can occur if users are allowed to select delimiters. The same options for overlays also occur when precedence information is added. Assuming one of the options has been chosen (or a syntax error was not introduced), changing a delimiter requires checking the precedence constraints. Unlike insertion (which occurs at a leaf in the tree), replacement can occur anywhere in the tree. This means that in addition to checking the precedence constraints of the changed node and its parent, we must also check the constraints of the changed node and all of its children. Because we are simulating a left-to-right scan of the input, the children's precedences are checked in a left-to-right manner. If a precedence violation is detected, then the appropriate rotate transformation is applied. As with walking up the tree, we can optimize the downward walk

by iteratively walking down the tree until the precedence constraint holds and then using the appropriate chained rotate transformation.

Adding precedence information to the overlays paradigm does not change the extensibility of that algorithm. Assuming that the template has been defined, this algorithm only requires that left and right precedences be given.

This section presented overlays with precedence as an extension to overlays. Kaiser and Kant present a similar algorithm in another light: parsing without a parser. Parsing is discussed in the next section. One of the problems that incremental parsers must be able to handle is incorrect syntax caused by incomplete language constructs. Kaiser and Kant only discuss infix operators and parentheses and the only incorrect syntax that they allow are mismatched parentheses. We now show how to extend the algorithm to handle incorrect syntax.

Mismatched parentheses are easily handled by considering an inserted open (close) parenthesis as a prefix (postfix) template. After floating the parenthesis to its correct level in the tree, we attempt to match the unpaired parenthesis. This is done by walking up the tree until we encounter a minimum precedence node or we reach the root. If the stopping point is a pair of (matched) parentheses or an unmatched close (open) parenthesis, then we can match the newly inserted parenthesis as follows:

**Unmatched** The two parentheses form a matched pair. The close (open) parenthesis node is eliminated and the newly created open parenthesis node is converted into a matched parentheses node. This transformation is shown in Figure 5.11. The elimination of the close (open) node means the precedence constraints might be violated between "P" and "C", so it is necessary to try to float "C".

**Matched** The close (open) parenthesis is taken from the matched parentheses node and a matched parentheses node replaces the newly created open parenthesis node. This transformation is shown in Figure 5.11. The newly created open (close) node must now be floated because the precedence constraints might be violated. Finally, "C" must be floated also.

Other incomplete language-like constructs can be handled in similar way if each delimiter/keyword has a list of left and right legal matches. For example, suppose that we have the legal constructs "if-then" and "if-then-else". Then the properties of the keywords **if**, **then**, and **else** are:



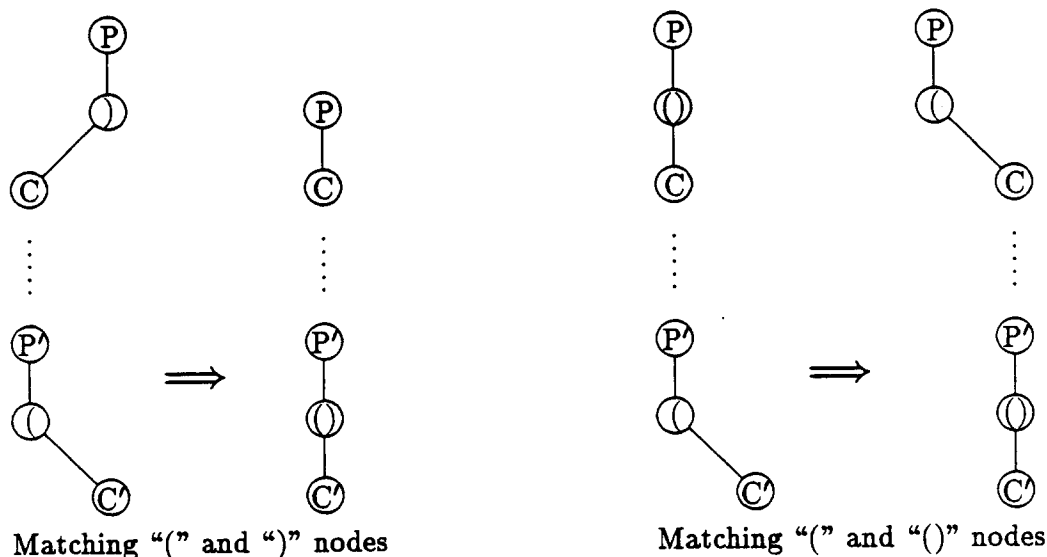


Figure 5.11: Matching Parentheses

| Keyword   | Type   | Left Prec | Right Prec | Left matches  | Right matches   |
|-----------|--------|-----------|------------|---------------|-----------------|
| if        | prefix | max       | min        | none          | then, then-else |
| then      | infix  | min       | low        | if            | none, else      |
| else      | infix  | min       | low        | if-then, then | none            |
| if-then   | prefix | max       | low        | none          | else            |
| then-else | infix  | min       | low        | if            | none            |

The "low" precedence is chosen so that it is greater than the left precedence of the statement separator but less than the left precedence of most other operators.

Figure 5.12 shows the sequence of parse trees corresponding to the input `if a<b then c`. After typing `if a<b`, we have the tree shown in Figure 5.12a. Figures 5.12b and 5.12c show the tree before and after floating the `then` keyword. A match is performed and the resulting tree is shown in Figure 5.12d. The final float of `<` does not change the tree.

The keyword/delimiter `then` is overloaded in the above example (it is used by both the `if-then` and `if-then-else` nodes). This does not cause a problem because both nodes use `then` as an infix keyword/delimiter and hence, the same template is used. However, other delimiters such as `-`, are used as both a prefix and infix delimiter, and as mentioned in the previous section, we would like to bind the different templates to the same key. The restrictions on overloading keywords/delimiters are the same as those given in Section 5.5.3.

The rule governing insertion is unchanged for overlays with precedence, however deletion is more complicated because the affected templates may have floated away from the operand that is being deleted. If we allow prefix/infix overloading, then the delete

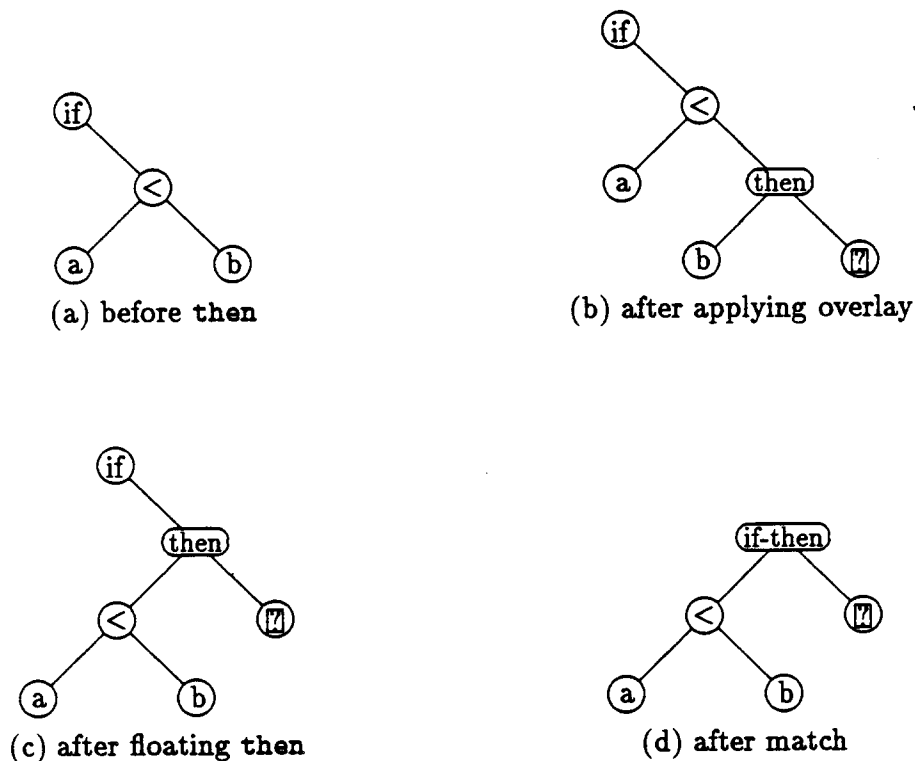


Figure 5.12: Example: Keyboard Input of if-then Construct

command must be modified to check for two cases. The first case occurs when we delete the left operand of an infix node that can be converted to a prefix node. In this case, we simply replace the infix node with the prefix node. The other case occurs when the rightmost left child of an infix node is deleted. If the infix node can be converted to a prefix node, then the tree transformation shown in Figure 5.13 is used. More precisely, if we delete node "2" in Figure 5.13, then:

1. Walk up the tree to the left to find node "L".
2. If the parent "P" is an infix node and has a prefix overloading "P'", perform the tree transformation shown in Figure 5.13.
3. Float "P'".

Heeman[82] extended Kaiser and Kant's parser in a more radical way as part of INFORM[166]. Heeman uses just three main operations to manipulate the parse tree. The first two operations are used to merge two trees together (one merge, match-merge, is used for parentheses and calls the other merge); the third operation is used to divide the tree

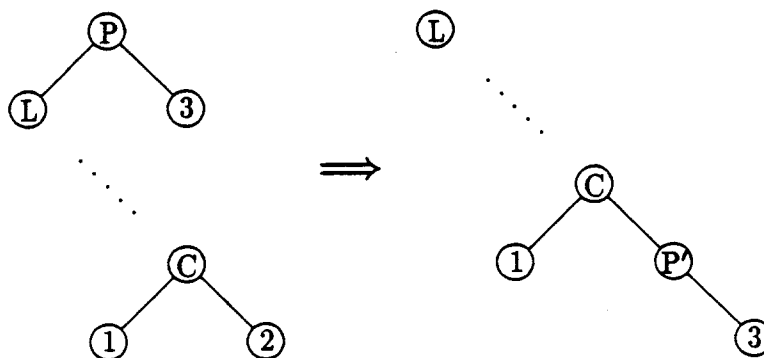


Figure 5.13: Conversion of infix to prefix for delete

into two trees. By combining these operations, insertions, deletions, and replacements can be performed. For example, insertions are performed by splitting the tree into two trees: a tree consisting of the tokens to the left of the insertion point and a tree consisting of the tokens to the right of the insertion point. The left tree is merged with tree representing the new string and the resulting tree is merged with the right tree.

Heeman's approach is simpler than the proposed extension given above in that there are only three main algorithms. However, the algorithms are slower than those above. The splitting and normal merging algorithm take time proportional to the depths of the trees involved with the merge operation requiring two passes in the case where characters are concatenated; the match-merge algorithm requires searching the trees to be merged for unmatched parentheses. Heeman's algorithms do not handle n-ary operations and overloaded delimiters, although it appears that they can be modified to accommodate them.

A significant speed up in his algorithms is achieved by the use of a mechanism similar to "stop points" discussed in Section 5.5.2. INFORM's grammar uses three nonterminals: *EXPR*, *START*, and *LREF*. A *LREF* denotes a delimiter. An *EXPR* denotes an operand that is constrained by precedence and associativity information. A *START* denotes a subformula and is used as a stopping point by the algorithms—they act as terminal nodes for the algorithms that descend the tree and as root nodes for algorithms that ascend the tree. Starts are used for the operands of fractions, subscripts, square roots, etc.

## 5.4 Parsing

Although incremental parsers are more complicated to implement than overlays, they allow a more natural left-to-right entry of programming language constructs and non-infix operations such as integration. An incremental parser is necessary because the time required to completely reparse even a moderately-sized expression after every character that is typed is too large for real-time redisplay of the screen. Reparsing after every character that is typed is necessary in order to correctly update the display—the display is the semantics of the parse. Some problems introduced by character-level incrementality are discussed in Section 5.4.2.

A problem with parsing that is not true of overlays or templates is that parsers (actually the grammars they accept) are not inherently extensible—none of the incremental parsing algorithms presented in the literature are for grammars that are extensible. In Section 5.5.1, an extensible incremental operator precedence[5] parser is presented. This algorithm is similar to Jalili and Gallier's algorithm[90] described in Section 5.4.3, except that it is optimized for an operator precedence grammar.

As the speed of machines increases, the need for an extensible parser diminishes. Instead of extending a running parser, it is possible to change (add to) a grammar, generate new tables for the grammar, throw out all of the state and/or structure information associated with existing expressions currently in the system, and finally, “reparse” all of the expressions using the new grammar. An order of magnitude increase in the speed of existing systems will probably make this approach feasible. Even if speed is no longer a factor, changing an  $LL(k)$  or  $LR(k)$  grammar and insuring that it remains  $LL(k)$  or  $LR(k)$  is not trivial. This problem was noted by Pratt[134] among others. An example of this problem is demonstrated in Figure 3.10 of Section 3.3.

### 5.4.1 Some Parsing-Related User Interface Issues

There are three user-interface issues that impact parsing. The first issue deals with single character tokens versus multi-character tokens. The second issue deals with parentheses and whether vertical motion operators such as division and exponentiation implicitly create parentheses or not. The last issue deals with strings and linear input.

Mathematical expressions typically contain many more single character tokens (usually variables such as  $x$  and  $y$ ) than multiple character tokens (usually functions such

as log and sin). For example,  $x^2 + 2xy + y^2$  uses only single character tokens. The interface can be optimized so that the input sequence 2xy is interpreted as implicit multiplication of the number 2 and the variables x and y if the scanner is modified so that single characters are considered tokens. The exceptions to this modification are numbers (tokens beginning with a digit or decimal point) and “escaped” variables (tokens beginning with an escape character). The latter exception allows for functions such as log and sin to be entered. CASs have many functions such as “simplify”, “substitute”, and “solve”. Because of this, it is not clear that such an optimization is desirable. Milo[121] is the only math interface that uses this optimization.

It is frequently the case that exponents, subscripts, numerators and denominators consist of a single token. For more complicated exponents that are not automatically handled by precedence considerations, parentheses must be used. The standard parsing model handles this well. For an input model that provides feedback at the character level, it is not clear that this is the desired model. In particular, the parentheses in the output are not necessary.

Vertical motion operators such as exponentiation move the text cursor to a new “line”. The model that most math interfaces implement is one in which the text cursor remains on the current line until the user takes some action to move off the current line (e.g., by moving the text cursor). MathCAD[113] is an exception and requires explicitly typing parentheses to remain in the exponent. In MathCAD, the redundant parentheses are displayed until the user moves the mouse outside of the expression’s box at which time the expression is redrawn without the parentheses. Moving the mouse back inside of the box causes the expression to be drawn with the parentheses again. Theorist[19] allows users to choose between the two models via the “Fortranize” switch.

For systems that allow unbalanced parentheses, a decision must be made as to whether the “scope” of parentheses extends beyond the current line. For example, suppose that we insert a “(” before the c in the expression  $a(\frac{b}{c} + d)$ . If we parse the expression in the standard manner, then the expression becomes  $a(\frac{b}{c+d})$ . This is the result that MathCAD and MathStation produce. If we limit the scope of parentheses to the current line, then the expression becomes  $a(\frac{b}{c} + d)$ . MathScribe limits the scope to the current line to avoid large changes in the display and to emphasize the “line” model of input.

MathScribe uses *invisible parentheses* to implement the “line” model of input.

Invisible parentheses are unique parentheses with zero width that do not affect the display<sup>3</sup>. They are always inserted in pairs whenever a vertical motion operator is entered. The incremental parsing algorithm (Section 5.5.2) never needs to parse more than what is inside paired parentheses, so we never parse more than a “line” at a time.

MathScribe puts expressions into two-dimensional form except when they are inside a string. Inside a string, the scanner flattens any trees to their textual form. Because a string is a token in the language, the contents are not formatted further. Hence, one way a user can enter an unformatted, linear expression is to enclose it in a string. Because the scanner is parameterizable, more than one type of string can be defined. In particular, a string that prints in a different font and has invisible delimiters can be defined. This idea is similar to ideas embodied in Syned[86] and PSG[12] in that textual editing can be performed in a structural editor.

String delimiter marks should be doubled when typed by the user in order to guarantee balanced marks and avoid interpreting all of the expressions to the right of the first delimiter as a string.

#### 5.4.2 Error Recovery and Correction in a CAS Interface

Most work on error correction and recovery has been targeted at batch-oriented compilers. Interactive environments have different needs. Many interactive environments such as the Cornell Program Synthesizer[162] prevent syntax errors from happening so that error correction and recovery is not an issue for the parser. CAS interfaces differ from previous interactive environments that must deal with syntax errors because they reparse the input after every character that is entered.

The character-level incrementality necessary for equation display impacts the error recovery and correction techniques used by the parser in the following ways.

- The parser must tolerate intermediate error states so that multi-character delimiter/keywords can be entered.
- The parser should never delete tokens. Imagine the frustration of someone trying to enter some subexpression only to have it deleted at a certain point because this was the “best” way to obtain a correct program.

---

<sup>3</sup>Invisible parentheses are simply another matchfix operator (Section 5.5.1) in MathScribe for which there is no corresponding key or input template. They have zero width; they are not displayed nor can they be selected. Thus, there is no way that the user can enter them. This guarantees their uniqueness.

- The parser should distinguish between tokens that were inserted to correct errors and those the user types. Tokens that were inserted to correct errors should be thrown out during re-parsing. This is important so that a user who is not watching the display does not have to be aware of corrections that the parser made to “intermediate” states of the program as it was being entered.

The latter two requirements were noted by Wegman and Alberga[170]. They also point out that in an incremental setting, the results of the previous parse are available and may serve as a good clue to an error correction routine.

Because of the need to format expressions correctly, previous mathematical editors and interfaces have enforced syntactic correctness or performed trivial error correction such as inserting dummy operands as needed. MathCAD[113], MathStation[115], and MathScribe tolerate mismatched parentheses. MathScribe is the only system that allows multi-character delimiter input and tolerates illegal delimiters because they may be prefixes of legal delimiters.

Experience with MathScribe has shown that the template/overlay approach is somewhat unnatural for dealing with programming language constructs. Tolerating or correcting errors is probably a better approach (at least making tolerance and/or correction an option). Section 5.5.4 presents a simple error tolerance algorithm for an operator precedence parser. The algorithm has not been implemented and its suitability is unknown.

The remainder of this section presents an overview of error correction techniques that other people have used. A general goal of most error correction techniques is that the correction algorithm should impose little or no overhead for parsing correct programs or correct portions of incorrect programs. Another important property of most error correction algorithms is that they should not delete input that has been reduced because in general, it is hard to undo the semantic actions that took place in conjunction with the reduction. Some techniques that have been used are:

**Panic Mode** A simple and fast technique that discards input until some special symbol such as ; or end is found. The parsing stack is popped until the special symbol is a legal next symbol. This not only invalidates the semantic actions that accompanied the reductions that were popped, but errors in the discarded input are not discovered. For an interactive interface, none of the parse stack or input would actually be discarded; they would be collected together in the reduction involving the special symbol and be

incorporated into an error format that linearly formats all of its children.

**Error States/Productions** Error productions can be added to the grammar that anticipate mistakes – unanticipated errors can cause problems. A similar technique is to add error actions to the parsing tables. Both techniques are tuned to the particular grammar and will not perform well for extensible grammars.

**Forward Context** Graham and Rhodes[75] describe an algorithm for a simple precedence parser that uses forward context (i.e., input past the point of the error) to help determine the best correction. Their algorithm works by forcing reductions before the error, restarting the parser after the error and stopping when another error is detected. Insertions, deletions, and replacements are then tried on potential right hand sides of productions around the point of the error.

**Multiple trials** After an error is encountered, the (LR) parser is restarted to accumulate forward context. The parse stack is initialized to the set of all states, and new legal sets of states are pushed onto the stack. The forward context accumulation stops when only an error state can be pushed or when the amount of forward context exceeds some predefined value. Additions and deletions are then considered and matched with the possible forward parses. Jalili and Gallier[90] use this technique for their incremental LR parser discussed in Section 5.4.3. Preference is given to those corrections that do not cause input that has already been parsed to be reparsed. A variation on this is to make the additions and deletions prior to the forward parse. Sets of states do not have to be used. This technique is used by the simple recovery phase of Burke and Fisher[28] for their error correction algorithm.

Correction typically consists of adding a token before the error token, deleting the error token, or replacing the error token with a new token. Fancier correction algorithms may also employ spelling correction and token merging. Cost vectors are often used to choose the best correction and can be used to tune the algorithm to the particular grammar. As mentioned earlier, deletion and replacement are probably poor ideas in an interactive environment. Fischer, Milton, and Quiring[53] describe a simple and fast algorithm for most LL(1) grammars that works by insertion only (with a user-supplied cost vector to determine the best insertion). Their algorithm precomputes tables in order to speed up error correction; precomputation is of dubious value for an extensible grammar.



Determining the quality of the correction is by nature subjective. A common measure adopted by many people is that a correction is *excellent* if it is the one a human reader would make, *good* if it results in a reasonable program with no spurious errors, and *poor* if it results in one or more spurious errors. Using these ratings, Burke and Fisher[28] claim a (78%, 20%, 2%) rating on a Pascal common error data base. Fischer, Milton, and Quiring's insert-only algorithm[53] rates (45%, 26%, 29%) versus a rating of (40%, 42%, 18%) for the Graham-Rhodes method on a 63 statement ALGOL program.

Many of the algorithms presented in the literature require that the parsing method have the *viable prefix property*<sup>4</sup>. This means that when the parser detects an error, all of the input to the left of the error is a prefix of some string generated by the grammar. LL and LR grammars have this property; operator precedence parsers do not. However, it appears that a number of these techniques (in particular, Fischer, Milton, and Quiring's algorithm[53]) can be adapted to the modified operator precedence parser presented in Section 5.5.2.

The error recovery algorithm presented Section 5.5.4 does not attempt to correct errors, but instead tolerates them by building successively larger error nodes around the point of error until the parser can be restarted. This algorithm is similar to a tolerance algorithm proposed by Leinius[103] and discussed in Graham and Rhodes[75]. Leinius's basic idea is to find successively larger *phrases* (roughly, possible right hand sides) containing the error and replacing them with any "locally correct" left hand side. The algorithm stops when a unique left hand side is found. Error tolerance, as opposed to error correction, is advantageous in an interactive environment because it minimizes the amount of change on the screen while an expression is being entered. On the other hand, error correction can be used to present the novice user with possible corrections.

### 5.4.3 Incremental LR Parsing

Ghezzi and Mandrioli[71, 72] first attacked incremental LR(0) parsing in 1977. Their algorithms were space intensive. Jalili and Gallier[90] designed an incremental LR(1) parsing algorithm that is both time and space efficient; it is described below. Ballance, Butcher, and Graham[13] adapted this algorithm to work off an abstract syntax tree instead of a concrete syntax tree, substantially reducing the space usage for most grammars. Yeh[175] has also designed an incremental LR(1) algorithm that appears to be competitive

---

<sup>4</sup>This is also referred to as the *correct prefix property* and the *valid prefix property* in the literature.

with Jalili and Gallier's algorithm.

Jalili and Gallier's algorithm requires storing only a minimal amount of extra information (the state of the *preceding* node) in the parse tree in order to incrementally reparse a change. A node  $n$  has a *preceding* node  $m$  if  $m$  is the left sibling of  $n$  or  $m$  is the preceding node of the parent of  $n$ . All nodes having  $m$  as a preceding node are said to be *adjacent* to  $m$ .

Jalili and Gallier's algorithm works by noting that a change to a node  $m$  affects all nodes adjacent to  $m$ . This means that a subtree can be reused if its preceding node did not change; that is, if the state on the top of the parse stack matches the *preceding state* of the next "token" (subtree) to be parsed<sup>5</sup>. If the state does not match, the subtree is split into smaller subtrees and the test is reapplied. Because all subtrees along the left edge of a tree share the same preceding state, a mismatch of states means that the splitting process is recursively applied until a terminal symbol is reached.

A simplified version of Jalili and Gallier's algorithm is given in Figure 5.14. Their full algorithm handles multiple insertions and deletions and transforms parse trees into their equivalent strings in order to detect the insertions and deletions<sup>6</sup>. Because a CAS interface's display must be updated after every change, the ability to handle multiple changes is not needed and the expense of string conversions and comparisons can be avoided. This simplification also eliminates a number of cases that must be checked in their algorithm. A particularly nice feature of Jalili and Gallier's algorithm is that it can use the tables generated by any LR(1) parser-generator unchanged.

The incremental parsing algorithm in Figure 5.14 makes use of the tree function `ParseTree::Split(node, left, right)`. `Split` divides `tree` (the parse tree) into a sequence of subtrees that are to the left and right of a path from `node` to the root of `tree`. All the subtrees to the left of `node` are pushed onto the stack `left` such that the top of `left` is the subtree to left of `node`; all subtrees to the right of `node` are pushed onto the stack `right` such that the top of `right` is the subtree to the right of `node` in `tree`. This is illustrated in Figure 5.15. In general, `left` corresponds to the parse stack and `right` corresponds to the "unparsed" tokens and trees. By allowing the stacks to be NULL and not pushing subtrees

<sup>5</sup>The preceding state is the state used by an LR(1) parser's "goto" function to compute the new state after a reduction. The new state is a function only of the preceding state and the next token in the input stream.

<sup>6</sup>For their application, parsing a programming language, the linear string exists for display purposes and no conversion needs to occur. Nodes in their parse tree contain pointers to the lines corresponding to leftmost and rightmost subnodes of the nodes, and their text editor marks whether line has changed.

```

Tree* Tree::IncrementalParse(Tree* left, Tree* right, char* newString)
// The characters from left..right are deleted (right=NULL implies no delete).
// 'left', 'right' are leaves of 'this'.
// The characters in 'newString' are inserted in their place.
// This procedure can be used for insertion, deletion, and replacement
// of a single string.
{
    Stack p;                // Empty parse stack
    Stack u;                // Empty 'unparsed' stack
    Tree *t = NULL;

    Split(left, &p, &u);    // Initialize 'p' and 'u'.

    if (right) {
        // Delete subtrees between 'left' and 'right'.
        do {
            t = u.Pop();
        } while (!t->Contains(right));

        // Throw away subtrees up to and including 'right'.
        t->Split(right, NULL, &u);
    } else {
        // 'right' is NULL -- don't delete 'left'
        p.push(left);
    }

    // Turn 'newString' into a sequence of "tree tokens" and push them onto 'n'
    // such that the leftmost token of 'newString' becomes the top of 'n'.
    // The preceding state of each token is made an illegal state (eg, -1).
    Tokenize(newString, &u);

    // Do incremental parse until there are no more tokens/subtrees on 'u'.
    while (!u.Empty()) {
        t = u.Pop();
        if (t->precedingState == p.Top()->state) {
            p.Push(t);
        } else {
            // State mismatch -- split left edge of 't' apart.
            t->Split(NULL, NULL, &u);

            // Top of 'u' is a token--do normal parse until it is shifted on 'p'.
            NormalParse(p, u.Pop());
        }
    }

    // 'p' should have a single element in it, the new parse tree.
    return p.Pop();
}

```

Figure 5.14: Incremental Parser for LR(1) Grammar

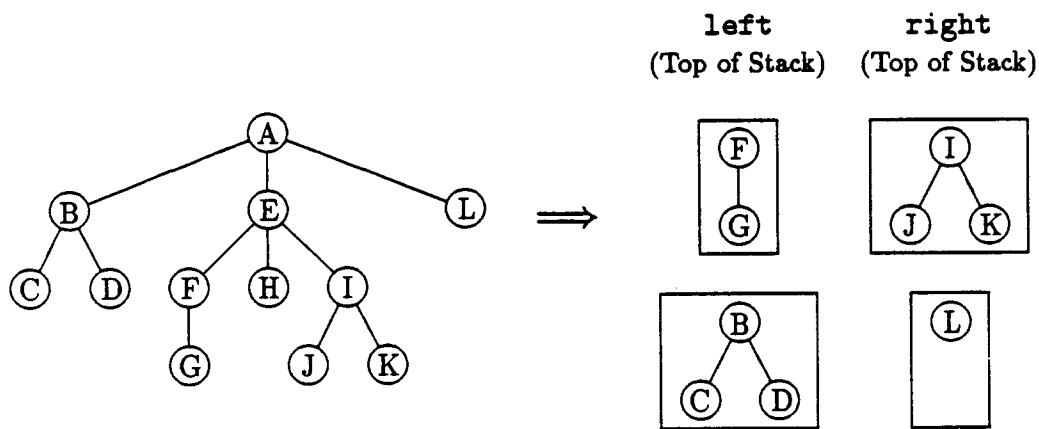


Figure 5.15: Example of Split(H, left, right)

onto NULL stacks, and by allowing a NULL value for node to mean the split path is to the left of the leftmost node, the single function `Split` replaces all of the special case functions used by Jalili and Gallier.

#### 5.4.4 Attribute Grammars

Attribute grammars[5] are an alternative to LL and LR-based grammars. Attribute grammars are not extensible, and because of this do not meet one of the basic requirements of a CAS interface. Nonetheless, a great deal of work has been done on incremental attribute evaluation, and many of the examples and systems built use attributes for formatting the display. This section briefly summarizes attribute grammars, incremental evaluation of attributes, and lists some relevant systems.

Attribute grammars associate with each node in the parse tree a set of values (attributes) and a set of equations of the form  $a = f(B)$ , where  $a$  is an attribute and  $B$  is a set of attributes of a node, its parent, its siblings, or its children. If  $a$  is a function only of the attributes of its children, then  $a$  is called a *synthesized* attribute. If  $a$  is a function only of the attributes of its siblings and its parent, then  $a$  is called an *inherited* attribute. Attributes form the semantics of the parse; in this case, the attributes define how an expression is formatted. The attributes of a bounding box are discussed in Section 4.1.1. They can be grouped as:

**Synthesized:** *width*, *height*, and *depth*

**Inherited:** *x*, *y*, and *fontSize*

Incremental update of attributes grammars is discussed by Reps, Teitelbaum, and Demers[141]. They present several algorithms for incremental update, including one which is asymptotically optimal in time. They and others have used attributes to control pretty-printing of code in language-based editors. Franchi-Zannettacci[61] takes a slightly different approach by using a higher-level language for easy specification of graphical languages. This higher level language is processed into an attribute grammar. The main example discussed in his paper is an expression editor. Attributes grammars can be slow due to the large number of attributes that must be updated.

## 5.5 The MathScribe Parser

The parser used for MathScribe was originally designed with the goal of handling the MACSYMA syntax. Because of this, it can accept roughly the same grammar that MACSYMA accepts. The MACSYMA parser is based on an extensible parsing technique developed by Pratt[134]. The terminology employed here is similar to that used by Pratt and the MACSYMA manual[109, Appendix 2]. This class of grammars was chosen because it is fast to parse, it is easily extensible by the user for many useful syntaxes, and because it handles the syntax found in most computer algebra systems. Pratt[134] discusses some of the problems with BNF-based grammars and user-extensibility.

The MathScribe parser differs from the MACSYMA parser in the following major ways.

- The MathScribe parser is an incremental bottom-up parser that is incremental at the character level. The MACSYMA parser is a top-down parser and is not incremental.
- MathScribe allows the user to define new “special” operators (see below).
- The MathScribe parser does not use the concept of “part of speech” that MACSYMA uses to distinguish between expressions that return boolean values and other special cases. “Parts of speech” are simply constraints on the types of operands that an operator can have. The MACSYMA parser uses these additional constraints to detect syntax errors in much the same way that type errors can uncover simple mistakes—it does not affect the abstract syntax tree. This notion is not used in MathScribe because an error of this kind does not affect the display of the expression (the semantics of the parse) and because a consistent implementation requires knowledge of the return

“type” of a user-defined function, something that most computer algebra systems (MACSYMA included) do not provide.

It is fairly simple to add the constraint checks to the MathScribe parser: during a reduction, the operator checks all of its arguments to make sure that they satisfy the constraint. If some operand does not satisfy the constraint, then an error is indicated in some way.

The remainder of this section describes the class of grammars accepted by the MathScribe parser, how the incremental parser works, and finally, how error recovery, correction, and tolerance are handled.

### 5.5.1 An Extended Operator Precedence Grammar

The MACSYMA parser groups all operators into the *syntactic categories*: *prefix*, *infix*, *n-ary*, *postfix*, *matchfix* (e.g., parentheses), *nofix*, and *special*. Nofix operators are operators with no arguments and are rarely used. Special operators include language constructs such as

```
if operand then operand
while operand do operand od
```

Special operators consist of a keyword followed by an alternating set of operands and delimiters, some of which may be optional (e.g., the increment in a for loop). MACSYMA allows users to define new operators in any syntactic category with the exception of special operators.

In MACSYMA and MathScribe, every delimiter has a left and a right binding power. Binding powers determine the relative precedence of operators and also their associativity. Binding powers serve the same purpose as precedence functions[5] in a standard operator precedence grammar. However, because the syntactic category of the delimiter is also known (and used), many of the error states that a precedence matrix contains (i.e., blank entries in the matrix) are detectable. Knowing the syntactic category also allows for overloading delimiters (see Section 5.5.3) such as unary/infix “-”.

MathScribe generalizes the MACSYMA parser by considering all operators to be special cases of the *special* operator. MathScribe classifies every delimiter as either prefix, infix, n-ary, postfix or postfix-0 (described below). *Nofix* delimiters can also be included, but this is not a very useful addition. Delimiters can be used by more than one operator and can belong to more than one syntactic category—see Section 5.5.3 for details.

A postfix-0 delimiter is similar to a postfix delimiter except that a left operand is optional. They are used for constructs such as function call where the number of parameters can be zero. The remainder of this section only distinguishes between postfix-0 and postfix delimiters and between n-ary and infix delimiters when necessary.

In MathScribe, special operators consist of one or more delimiters separated by operands: an optional prefix delimiter followed by zero or more infix delimiters, and optionally terminated by a postfix delimiter. Postfix-0 delimiters are a minor exception to this rule. Prefix, infix, and postfix operators are trivial cases of special operators. A matchfix operator is a special operator with two delimiters: a prefix delimiter and a postfix/postfix-0 delimiter. In MACSYMA, function call and array subscripting are built-in and similar notation (e.g., string indexing) cannot be defined by the user. In MathScribe, these notations are special operators with two delimiters: an infix delimiter followed by a postfix-0 delimiter. For example, the function call operator consists of two delimiters: an infix ( delimiter and a postfix-0 ) delimiter.

Delimiters are tagged as being either optional or required. Optional delimiters are useful in language constructs such as `for` loops. They can also be used to indicate an optional `else` clause for `if` statements.

### 5.5.2 Incremental Extended Operator Precedence Parsing

Parsing this class of grammars is straightforward and is similar to simple operator precedence parsing in that a decision as to whether to shift or reduce is based on the precedence of the delimiters. An incremental version of the parser requires initializing the parse stack and the unparsed token stream appropriately together with some minor modifications to the non-incremental version. For simplicity, we first present the standard parser (Figure 5.16) and then show how to modify it to be incremental.

The parser uses three stacks: a standard *parse stack* to hold the results of the parse so far; an *operator stack* containing operators corresponding to delimiters on the parse stack<sup>7</sup>; and an *unparsed stack* containing input we have yet to see. Because the parser is used in an incremental setting, the contents of the parse and unparsed stack are trees; tokens are trees with a single node (i.e., they are leaves).

Rather than having special symbols to handle "top of stack" and "end of input", a

---

<sup>7</sup>Because of delimiter overloading, there can be several operators corresponding to a single delimiter. The operator stack contains the operator corresponding to the delimiter, resolved up to the syntactic category.

```

Tree* Parse(char* string)
{
    // Initialize stacks with a unique matchfix "sentinel" operator.
    TreeStack parseStack(LeftMatchfixDelim);
    OperatorStack opStack(MatchfixOperator);
    TreeStack unparsedStack(RightMatchfixDelim);

    // Put tokens onto unparsed stack, with leftmost token being top of stack.
    unparsedStack.Push(Tokenize(string));

    while (!unparsedStack.Empty()) {
        Operator *newOp = chooseOp(opStack.Top(), unparsedStack.Top());
        switch (chooseAction(parseStack.Top()->syntaxCategory,
                             newOp->syntaxCategory)) {
            Shift:
                parseStack.Push(unparsedStack.Pop());
                if (newOp) then opStack.Push(newOp);
            ReduceShift:
                parseStack.Reduce(opStack, newOp);

                // Shift on delimiter if legal.
                if (legalSuffix(opStack.Top(), newOp)) {
                    parseStack.Push(unparsedStack.Pop());
                    opStack.Push(newOp);
                } else {
                    Error(parseStack, opStack, unparsedStack);
                }
            Reduce:
                parseStack.Reduce(opStack, newOp);
            ErrorOperand:
                parseStack.Push(DummyOperand);
            ErrorOperator:
                parseStack.Push(DummyOperatorDelimiter);
                opStack.Push(DummyOperator);
        }
    }

    // There should only be one item on parseStack -- the matching parens.
    // The parse tree is child of matchfix op.
    return parseStack.Top()->child[1];
}

```

Figure 5.16: A Non-incremental parsing algorithm for MathScribe's Grammar



unique matchfix operator (i.e., an operator that the user cannot define) is used. The parse stack is initialized with the opening delimiter, the operator stack with the matchfix operator, and the unparsed stack with the input to be parsed along with the closing delimiter (at the bottom of the stack).

Parsing begins by choosing an operator (or the syntactic category of a delimiter) based on what is on the top of the parse stack and what is on the top of the unparsed stack (`chooseOp`).

1. If the new token (i.e., the top of the unparsed stack) is not a delimiter, then nil is returned.
2. If there is only one operator corresponding to the delimiter, then it is returned.
3. Otherwise, if there is a delimiter on the top of the stack, then we choose the prefix operator associated with the delimiter—doing otherwise results in a “missing operand” error.
4. If there is an operand on the top of the stack, then we choose an infix or postfix operator associated with the delimiter—doing otherwise results in an “operand–operand” conflict.

These choices are discussed more fully in Section 5.5.3.

Once we have resolved the operator down to its syntactic category, `chooseAction` decides whether to shift, reduce, or error correct based on the syntactic categories of the top of the stack and the new (pending) operator. The actions the parser takes are summarized in Figure 5.17. The decision as to whether to reduce in the case that the top of the stack is an operand and the new token is an infix or postfix operator is based on the precedence of the operator on the top of the operator stack and the precedence of the new operator. Specifically, reductions are performed as long as the right precedence of `parseStack.Top()` is greater than the left precedence of `newOp`.

In the error cases, the parser shifts either a dummy operand or dummy operator onto the parse stack in order to preserve syntactic correctness. This simplifies formatting expressions because it guarantees that all operators have the correct number of operands<sup>8</sup>. Standard precedence parsers do not detect illegal right sides of productions until reduction

---

<sup>8</sup>The dummy operand/operator can be made to have zero width and height and thus be invisible if desired.

| Top Token | New Token |        |       |         |           |
|-----------|-----------|--------|-------|---------|-----------|
|           | operand   | prefix | infix | postfix | postfix-0 |
| operand   | E o-o     | E op   | R/S   | R/S     | R/S       |
| prefix    | S         | S      | E o   | E o     | R/S       |
| infix     | S         | S      | E o   | E o     | R/S       |
| postfix   | R         | R      | R     | R       | R         |
| postfix-0 | R         | R      | R     | R       | R         |

where

- R == Reduce the stack
- S == Shift new onto stack
- R/S == Possibly reduce the stack, followed by shift new onto stack
- E o-o == Operand-operand conflict (shift 'times', other?)
- E op == Error: shift on dummy (infix) operator
- E o == Error: shift on dummy operand

Figure 5.17: Parser actions (`chooseAction`)

time. However, following an idea first described by Graham and Rhodes[75], legal right sides are checked when a delimiter is first shifted onto the parse stack (`legalSuffix` in Figure 5.16). This provides better error detection/correction because errors are detected earlier. Error correction and the `Error` procedure of Figure 5.16 are discussed more fully in Section 5.5.4. The ability to check for legal right sides of productions (i.e., legal sequences of delimiters) is simplified by linking legal delimiters sequences together in a tree. An example is shown in Figure 5.18.

The reduction procedure is fairly straightforward because all errors have already been handled. It is shown in Figure 5.19. The only complication involves handling multi-delimiter special operators with optional delimiters such as a `for` statement. The algorithm given in Figure 5.19 avoids this complication by assuming the formatting procedures handle missing optional delimiters as they choose best—most likely by not displaying them. The reduce procedure must specifically check for n-ary and postfix-0 delimiters because their number of arguments is not known a priori. The reduce procedure continues as long as right binding power of the top of stack is not greater than the left binding power of the new operator.

The non-incremental parsing algorithm given in Figure 5.16 initializes some stacks and loops until all input is consumed. As described above, the loop consists of choosing an operator based on what is on top of the parse stack and what is on top of the unparsed

### Statements

**for** var **in** list [**while** condition] **do** expression  
**for** var **from** start **to** stop [**by** inc] **do** expression  
**while** condition **do** expression

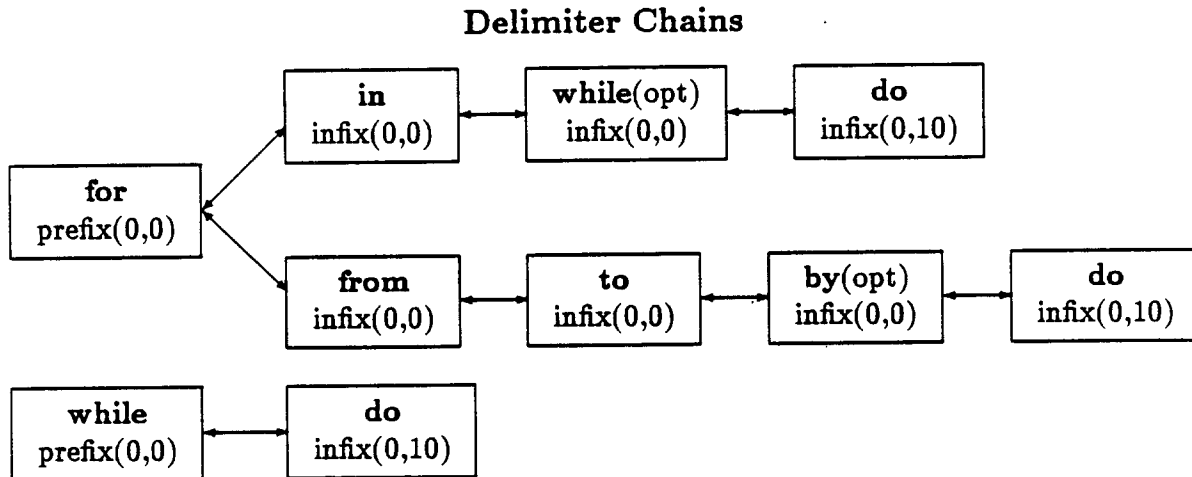


Figure 5.18: Delimiter Chain Example for Some Overloaded Delimiters

stack. The algorithm either shifts on the new token, reduces the stack, or shifts on an error token of some kind. In order to modify the parsing algorithm of Figure 5.16 so that it is incremental, we must initialize the stacks appropriately. The goal of the initialization is to set the parse stack and the operator stack to the states they would be in if this were not an incremental parse. Unlike the incremental parsing algorithm used by Jalili and Gallier for an LR(1) grammar (Section 5.14), it is not necessary to store parser states or perform a state check inside the shift/reduce loop for an operator precedence parser. This is because the decision as to whether to shift or reduce in an operator precedence parser is determined by simply comparing the left and right precedences of two adjacent operators; this relationship is not changed by adding or deleting delimiters that are not immediately to their left or right. Optimizations in the form of *stop points* limit the algorithm to dealing with only fragments of the program, further reducing the amount of parsing that needs to be performed.

The initialization splits the tree into two parts: the part of the tree to the left of the active node is pushed on the parse (and operator) stack, while the part to the right of

```

void TreeStack::Reduce(OperatorStack& opStack, const Operator* newOp)
// Reduce stack until rbp(opStack.top()) <= lbp(newOp).
// Parser is left associative for equal precedences.
{
    for ( Operator* topOp = opStack->Pop();
          topOp->rightPrecedence > newOp->leftPrecedence;
          topOp = opStack->Top() ) {

        int popAmount=0;
        // handle special case of missing (optional) operand.
        if (topOp->fixType == PostFix0 && Top(1) != Operand) {
            topOp = opStack->Pop();
        }

        // Work back through suffix chain of delimiters.
        // n-ary ops are implicitly chained together.
        while ((topOp->prefix != NULL) ||
              (topOp->type == NARY && topOp == opStack->Top())) {
            popAmount += topOp->operands;
            topOp = opStack->Pop();
        }
        popAmount += topOp->operands + 1;

        // 'popAmount' of trees are popped off the stack and turned into a list.
        // Format the list according to the operator.
        // Push the result (an operand) back onto the stack.
        Push(topOp->format(topOp, TreeList(this, popAmount)));
    }
}

```

Figure 5.19: The Reduce algorithm

the active node is pushed on the unparsed stack. The top tree in each stack must be further split apart: the right edge of the tree on the top of the parse stack is split apart while the left edge of the tree on the top of the unparsed stack is split apart. This further splitting is necessary for two reasons. First, it allows `Tokenize` to merge the rightmost and leftmost characters of the trees to the left and right of the change point if necessary. Secondly, it allows precedence relationships “near” the change to be recomputed and removes the need for a “state check” inside the shift/reduce loop of parsing algorithm.

Unlike the `Split` algorithm used by Jalili and Gallier, the splitting only needs to occur up to a *stop point*. Stop points are nodes in the parse tree that correspond to uniquely bracketed expressions such as paired parentheses and radicals. Systems that

implement a line model of input (page 129) also consider vertically displaced expressions such as numerators, denominators, exponents, and limits to be stop points. MathScribe puts invisible parentheses (page 129) around lines to create stop points. INFORM[82] uses the nonterminal *START* to indicate a stop point (page 127). Changes below a stop point cannot effect the parse tree above the stop point. Similarly, changes above a stop point (actually to the left or right) cannot affect the parse tree below (inside) the stop point. Thus, the edge splitting does not need to go all the way down to the character level but can stop at a stop point. In effect, stop points act as both root nodes and leaf nodes. Systems that allow unmatched parentheses (or more generally, illegal delimiter chains) cannot use these nodes as stop points because they may be broken apart by a newly entered parenthesis as shown below:

$$a \cdot (b + c) \rightarrow a \cdot (b) + c$$

In the example, a closing parenthesis is added after the *b*.

### 5.5.3 Overloading Delimiters

As noted earlier, delimiters can be overloaded. By looking at the parser action table (Figure 5.17), we can determine when delimiters can have more than one use. When a delimiter is first encountered (i.e., it is the top of the unparsed stack), we must choose whether the delimiter is either a prefix delimiter or an infix/postfix delimiter. In order to avoid an error condition, a prefix interpretation is chosen if the top of the parse stack is a prefix or infix operator. Similarly, an infix/postfix interpretation is chosen if the top of the parse stack is an operand or postfix operator (which eventually will be reduced to an operand). If the delimiter has an infix/postfix operator, we must know its left binding power in order to determine whether we must reduce the stack. Thus, several infix/postfix operators can use the same delimiter, but the delimiter's left binding power must be the same for all those operators.

After all reductions not involving the delimiter have been made, the delimiter's corresponding operator is shifted onto the stack. Its "right-side" properties are now required. At this point, we must resolve any infix/postfix delimiter into either an infix or a postfix operator. Its right binding power must also be determined. Figure 5.17 shows that in order to avoid an error condition, we should choose an infix interpretation if the next token is an operand or a prefix delimiter; we should choose a postfix interpretation

| Operator Classes  | Overloaded Delimiter | Example                       |
|-------------------|----------------------|-------------------------------|
| prefix/infix      | +                    | unary and n-ary plus          |
|                   | -                    | unary and infix minus         |
|                   | (                    | parenthesis and function call |
|                   | [                    | lists and array subscripting  |
| prefix/postfix    | "                    | string delimiters             |
|                   |                      | magnitude                     |
| infix/infix       | do                   | for and while loops           |
| postfix/postfix   | ]                    | lists and array subscripting  |
| postfix/postfix-0 | )                    | parenthesis and function call |

Figure 5.20: Examples of overloaded delimiters

(which eventually is reduced to an operand) if the next token is an infix/postfix delimiter. Unfortunately, the next token can be overloaded as either prefix or infix/postfix delimiter, and the decision as to which syntactic category to choose is based on whether we choose an infix or postfix interpretation for the current delimiter. Thus, a delimiter can *not* be used as both an infix and postfix delimiter in a single lookahead precedence parser unless the parser is modified to allow unresolved infix/postfix delimiters to be pushed on the stack and conflicts are always resolved in some fixed arbitrary manner (e.g., by choosing the longest (most specific) interpretation which is the infix interpretation in this case). Extending the algorithms to handle this case is not difficult.

As stated above about left binding powers, overloaded delimiters must have the same right binding powers if they are in the same syntactic category; prefix and infix/postfix overloadings can have different binding powers. Hence, binding powers are associated with the delimiter and its syntactic category and are inherited by the associated operator.

Overloaded delimiters occur quite frequently in standard notation. Some examples of allowed, useful overloadings are shown in Figure 5.20. The precise rules for overloading delimiters are listed below.

- A delimiter can be both a prefix delimiter and either an infix or postfix delimiter.
- Delimiters with operators in the same syntactic category must have the same binding powers.
- If a delimiter has several operators associated with it in the same syntactic category, then it must have a unique list of preceding delimiters. In particular, if a delimiter is used for a prefix, infix, or postfix operator, it cannot be used as part of another

operator in the same syntactic category.

These constraints can be checked quickly and easily. Notice that in Figure 5.18, `do` has a unique set of preceding delimiters in first two chains, but not in the third chain. This is not a violation of the overloading rules because the `while` in the third chain is a prefix operator and the `while` in first chain is an infix operator.

As an example of parsing with overloaded delimiters, suppose we wish to parse the expression `a-|b|`, where `-` is a prefix and infix delimiter and `|` is a prefix and postfix delimiter. When `-` is encountered, we choose the infix interpretation because an operand is on top of the stack. When the first `|` is encountered, we choose the prefix interpretation because an infix operator is on top of the stack. The second `|` becomes a postfix operator because an operand is on top of the stack; during reduction, it is matched with the prefix `|`. Notice that if we also allowed `|` to have an infix interpretation (as might be the case for the “value-at” notation  $|f(x)|_a$ ), then we cannot tell whether to choose the infix or postfix interpretation for the closing `|` for the input `a-|f(x)|-2` because the interpretation of `-` depends on the interpretation of `|`<sup>9</sup>.

#### 5.5.4 Error Recovery, Correction, and Tolerance

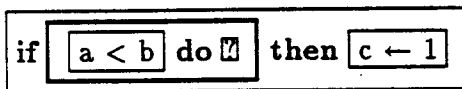
As noted above, simple error correction is performed by inserting a dummy operand if an operand is missing. This insures that all operators have the proper number of operands and can be formatted correctly. Error correction is also performed when two adjacent operands are detected; a dummy infix operator is inserted<sup>10</sup>. Error correction is more complicated for multi-delimiter operators (e.g., language-like operators) because delimiters must be paired correctly. Illegal suffix errors are detected during shifting. Correcting the error can be disconcerting because it might entail large changes in the display; instead we can instead employ error tolerance. This involves using a *n*-ary error operator whose children are the offending delimiter and its operands.

The simplest approach to error tolerance is to build an error operator around the offending token and restart the parser, incorporating more and more of the parse stack

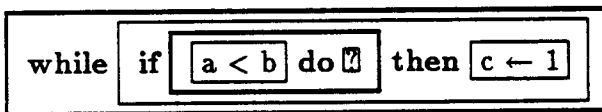
<sup>9</sup>The standard value-at notation  $|f(x)|_{x=a}$  avoids this problem because the closing `|` must be followed by an operand.

<sup>10</sup>In some computer algebra languages, this is not considered an error. MathScribe mimics mathematical notation in this case by interpreting `<name> <name>` as function application (e.g., `sin x`) and `<number> <operand>` as multiplication (e.g., `2x`); the function application and multiplication operators are automatically inserted. Other cases are considered errors and a dummy infix operator is inserted.

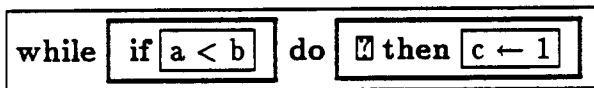
under the error operator until a valid prefix is found or the bottom of the stack is reached. A more sophisticated approach is to scan down the parse stack looking for valid prefixes (i.e., delimiters that precede the current delimiter for some operator) and skipping over invalid operators. If a match is finally found (i.e., all required delimiters are found), invalid interior non-matches are reduced to the error operator so that the matched operator can be legally reduced. If no match is found, we reduce the current delimiter to an error operator and restart the parser (possibly generating more errors). Using this scheme, the string `if a<b do then c<-1` is parsed as:



where the boxed text indicates the parse tree and the thicker box is an error operator. The dummy operand  $\square$  was automatically inserted. Notice that the subexpression  $a < b$  is parsed correctly even though it is a child of the error operator. A potential drawback to this error recovery scheme is that it favors the rightmost valid interpretation of the string. This is not always the most natural interpretation because mathematical notation is typically read from left-to-right. An example, the string `while if a<b do then c<-1` is parsed as



and not as



The rightmost interpretation is a result of using an operator precedence grammar; operator precedence grammars, unlike LL and LR grammars, do not have the “viable prefix property” [5].

Scanning down the stack for a match when an error occurs means that this algorithm is potentially an  $n^2$  algorithm, where  $n$  is the number of symbols parsed. In practice however, this is not a problem because: many reductions occur that decrease the size of the parse stack; the character level of incrementality severely limits the number of errors that occur; scanning only occurs for multi-delimiter operators; and the incremental nature of the parser means that  $n$  is a relatively small number.



## 5.6 Other Forms of Input

This chapter has dealt with algorithms for interpreting and manipulating what Wells[171] calls *keyboard languages*. These languages have the property that characters are placed on the screen by special keys and that the placement of the characters is controlled by these keys. In contrast to this approach, *pen languages* allow the user to draw characters on the screen in arbitrary positions and, through the use of handwriting recognition systems, interpret what characters have been drawn and their meaning (e.g., exponent, etc.).

In his thesis, Anderson[7] presents a top-down algorithm to recognize hand written mathematics. The algorithm is fairly general, but unfortunately involves backtracking and can be time consuming. The basic idea of the algorithm is to start at the "start" symbol for the language and try all possible derivations given the first character that is recognized. A clever partitioning of the rules based on terminals on the right hand side of the rule can drastically cut down on the number of rules checked, but does not eliminate the need to backtrack.

Anderson also presents an algorithm that converts the input into a linear string that can be parsed by a precedence parser. A similar algorithm is described by Martin[112]. The main restriction imposed by linearization is that any symbol used to indicate vertical displacement must be to the left of its operands. This means, for example, that the line used for division must extend out to the left of the numerator and denominator and that the limits of integrals, summations, etc., cannot always be centered over/under the sign. Note that many diacritical marks along with pre-subscripts and pre-superscripts are not allowed under this restriction. The algorithm works by scanning the input in a left-to-right direction and using a predefined linearization order for vertical displacement operators (which, because of the restriction above, it always encounters first). In a keyboard language, this restriction is not necessary because keystrokes/commands define the operator to use, and an arbitrary linear ordering can always be chosen.

Recent work at IBM by Orth[130] overcomes this restriction. Orth's recognition algorithm is divided into three phases: a geometric relation recognition phase, a geometric reconciliation phase, and a linearization phase. The first phase builds a set of geometric relations between all pairs of symbols on the tablet. Two examples of the relations used are: '*a* is on the same horizontal axis and the right of *b*' and '*a* is "near to" *b* relative to their *x*-coordinates and to the right of *b*'. The second phase prunes the set to eliminate redundant

relations. It also augments the set by constructing relations that may have been missed but must logically be present due to the presence of other relations in the set. The final phase linearizes the expression by sweeping across the relations from left-to-right forming several equations (because of vertical displacement) and finally merging these equations into a single equation.

Conversion to a linear string is an approach taken by most pen languages. None of the pen languages are incremental; they do not attempt to interpret the input until after the user signals that they are done drawing. We are not aware of any pen languages that allow the user to edit previous expressions. Large expressions that do not fit on a tablet are problematic for pen languages.

Another input form is voice. We are not aware of any work that has been done with voice and expression input. Voice input must eventually be interpreted as commands. As such, voice input is really just another form of "keyboard" input and the algorithms discussed in this chapter are appropriate for voice input.

## Chapter 6

# Selection

Many WYSIWYG drawing editors such as MacDraw[108] support selection of objects. They also support the notion of grouping objects together into a new (single) object. Once objects are grouped together, they cannot be individually selected unless they are ungrouped. Selection of expressions differs from drawing editors in that subexpressions of an expression are implicitly (recursively) grouped together and that selection of *either* a subexpression or the entire expression is allowed at any time. This chapter explores several ideas related to the selection of subexpressions from a displayed expression. The selection mechanism can limit selections to be syntactically meaningful quantities such as a single subexpression, or it can allow more general selections. Limiting selections makes it easier to select subexpressions, which is the most common thing to select. This is the approach taken by all interfaces except GI/S[177].

The first CAS interface to allow selection was Clapp and Kain's Magic Paper[34]. They used a light pen to underline the desired subexpression. Their paper does not give details of the mechanism, so it is not clear how they distinguish between selecting  $z$  and  $\frac{y}{z}$  in the expression  $x + \frac{y}{z}$ . Martin's Symbolic Mathematical Laboratory(SML)[111] also used a light pen for selection. In SML, a selection was determined by finding the smallest "box" containing the light pen. This is the method used by many systems today (substituting a mouse for the light pen).

Modern mathematical interfaces use mice, tablets or track balls instead of light pens. These input devices are often augmented by menu/keyboard commands to change the selection. An advantage of mice, etc., over menu/keyboard commands is that they provide "random access" to the subexpressions; menu/keyboard commands are limited to

“walking” through the display tree and are useful for “local” changes to the selection such as moving to a parent or sibling. For simplicity of discussion in this chapter, we assume that a mouse is used for selection, and that selection is performed while the user holds a mouse button down.

The next section analyses three methods and their algorithms for structural selection. While some of these methods have been used by other interfaces, the algorithms (and some optimizations to them) have not been described in the literature. Section 6.2 presents some ideas for non-structural selection. The chapter concludes by discussing some miscellaneous topics concerning keyboard input and selections.

## 6.1 Three Methods for Structural Selection

This section presents three different methods for selection: the **Point** method, the **Internal Rectangle** method, and the **External Rectangle** method. All three methods are similar in that only structurally meaningful subexpressions (boxes) can be selected. A very desirable feature of selection methods is that they provide constant feedback to the user of what is about to be selected while a mouse button is depressed. The feedback is given by highlighting the potential selection in some way. The **Point** and **Internal Rectangle** methods are fast enough to provide constant feedback even on “slow” machines.

Not every box that is displayed on the screen is selectable. Many interfaces do not allow the user to select delimiters. In addition, operators such as  $\sin^2$  and  $\int_b^a$  may not be selectable. Modifying the following algorithms to handle non-selectable boxes is simple.

We begin by describing the three methods. Their implementation, and an analysis of the algorithms (Figures 6.2, 6.3, and 6.4) are given later in this section.

**Point** While a mouse button is depressed, the cursor’s current position is used and the smallest (selectable) box containing the cursor is highlighted. This is illustrated in Figure 6.1a. This is the method used by many interfaces. If delimiters are selectable, it is sometimes hard to select subexpressions because the subexpression’s children might use up most of the area of the delimiter’s box. For example, it is hard to select  $b + d$  in  $\boxed{b + d} < \boxed{x}$  because the  $b$ ,  $+$ , and  $d$  boxes use up most of the area of the  $b + d$  box. This drawback can be mitigated by augmenting selection with keyboard traversal or some equivalent technique. In the example, selecting  $b$ ,  $+$ , or  $d$

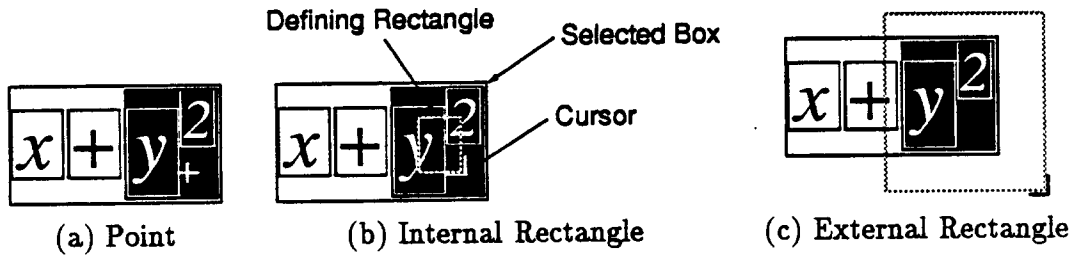


Figure 6.1: Three Alternative Methods of Selection

and then walking up one level using the keyboard selects  $b + d$ . Another drawback to this method is that it is not possible to select contiguous subexpressions.

**Internal Rectangle** This method is similar to the **Point** method except that this method uses a rectangle instead of a point to determine the selected box. One corner of the defining rectangle (usually the upper left) is determined by the cursor position when the button is first pressed. The opposite corner of the rectangle (usually the lower right corner) is determined by the current cursor position. The smallest box containing the rectangle is highlighted as shown in Figure 6.1b. In addition to solving the selection problem mentioned above with the **Point** method, this method trivially generalizes to allow selection of contiguous subexpressions.

**External Rectangle** A rectangle is defined as in the **Internal Rectangle** method described above. However, the *largest* box contained by the rectangle is selected instead of selecting the *smallest* box containing the rectangle. This is shown in Figure 6.1c.

This method is used by many drawing programs to select objects (which also have rectangular bounding boxes, although the boxes can overlap). One potential drawback to this method is that it is possible to select more than one box at a time as illustrated below:

$$\frac{a + b}{c + d}$$

The  $b$  and  $d$  boxes are both fully contained within the defining rectangle. The ability to select more than one box at a time might be considered useful if multiple selections are allowed. Alternatively, rules can be formulated to unify multiple selections into a single selection. For example, if more than one box is fully contained by the defining rectangle, then the selected box is the smallest box that contains all of the potential

selections. Many of the ideas presented in Section 6.2 are applicable to this problem. A drawback to this method is that it is possible to select “empty” subexpressions. Also, it is hard to select the entire expression if there is not a large border around the expression; in particular, this is true for large expressions that are clipped by the window. Selecting the entire expression is easy with the **Internal Rectangle** method because it is easy to find a point outside of the expression (e.g., any point outside of the window). Adopting the smallest box rule when multiple boxes would otherwise be selected eliminates this problem.

Although the **Internal Rectangle** method is superior to the **Point** method, most mathematical editor interfaces use the **Point** method; Milo[121], MathScribe[114], and Expressionist[18]/Theorist[19] use the **Internal Rectangle** method; no mathematical editor uses the **External Rectangle** method and it has not been mentioned in the literature. None of the algorithms have been described in the literature although their methods of operation (i.e., the user interface) are described in the appropriate user manuals.

Algorithms for all three methods are relatively straightforward. With the locality and incrementality optimizations described below, the **Point** and **Internal Rectangle** algorithms are very fast for the average case. The **External Rectangle** algorithm is slower and requires searching most of the expression tree. We begin by describing the **Point** algorithm.

The simplest algorithm for the **Point** method is, given the cursor coordinates relative to the root of the expression (the root box), recursively traverse each subexpression (child) of the tree if the cursor is inside of the child. If the cursor is inside of a box, but not inside any of its children, then we have found the smallest box containing the cursor and it can be highlighted. This algorithm is called every time the cursor moves while the mouse button is depressed. Although this simple algorithm is relatively fast, it can be improved upon. In particular, it should be noted that expression trees can be very broad (particularly for sums and products), so that this is not a  $\log(n)$  algorithm (where  $n$  is the number of boxes) because children of a box are not necessarily ordered. In the worst case (e.g., the expression is the sum of  $n - 1$  terms), this is a linear algorithm unless an external data structure is used to quickly determine within which child box, if any, the point lies<sup>1</sup>. Speed

---

<sup>1</sup>For MathScribe and probably other systems, the order of the children is constrained by the formatting and parsing modules so that left-to-right or top-to-bottom ordering is not guaranteed. However, it is possible to make one of the fields in the structure describing an operator specify whether the children are

is an important consideration because we are trying to provide "instant" feedback as to what the user is about to select.

The obvious algorithm can be improved upon by noting that between calls to the algorithm, the cursor position is not likely to change much. This means that the new box is likely to be the same, a child, a sibling, or the parent of the old box. Hence, if the search starts at the previous smallest box, it is very likely that the algorithm terminates rapidly. This does not improve the worst case running time, but does significantly improve its average running time under the locality assumption above. At worst, the running time is only increased by at most an additional  $\log(n)$  box visits (climbing the tree to the root). The **Point** algorithm is given in Figure 6.2.

The **Internal Rectangle** algorithm is given in Figure 6.3. It is an extension of the **Point** algorithm and uses the **FindSmallestBox** procedure from it. When the mouse button is first pressed, the smallest box containing the cursor is found and stored (**startBox**). When the cursor moves, we can quickly find the selected rectangle by observing that one of the boxes (**box1**) of the selected rectangle is constrained to lie on a path from **startBox** to the root, and the other box (**box2**) is constrained to be one of **box1**'s siblings. Hence we can find **box1** and **box2** by walking up the tree from **startBox** until we find a box that contains the cursor or we reach the root. If the cursor is contained in one of the stopping point's children, then we set **box2** to that child and **box1** to the box traversed before we stopped. If the cursor is not contained in one of the stopping point's children, then we set **box1** and **box2** to the stopping point. Because walking up the tree is faster than walking down the tree, this method is generally faster than calling **FindSmallestBox** with the previous value of **box2** and then finding a common ancestor of **startBox** and **box2**. The **SetSelection** method checks to see if **box1** and **box2** are the first and last children of their common parent, and if so, sets them to that parent.

It is often useful to be able to extend a selection, particularly when the subexpression is large and clipped by the window. The algorithm for extending a selection is basically the same as the algorithm for the **Internal Rectangle** method, except that starting point is the unextended selection instead of the cursor position at the time the mouse button is depressed.

The **External Rectangle** algorithm does not have as good an average case run-  

---

unordered, ordered by their  $x$ -coordinate, or ordered by their  $y$ -coordinate. If the children are ordered, some optimizations such as binary search are possible.

```

SelectionEvent(Event *event, SelectionData *data)
// Called by the window system when a mouse event occurs in this window.
// (x,y) is the cursor position relative to the window.
// 'data' is structure containing information about the box in this window.
{
    if (event->type == ButtonDown) {
        data->box = data->rootBox;
        data->x = data->box->x; data->y = data->box->y;
    }

    // Convert data x,y to box relative coords, find the smallest box,
    // and then convert data x,y back to box absolute coords.
    data->x = event->x - data->x; data->y = event->y - data->y;
    FindSmallestBox(data->box, data->x, data->y);
    data->x = event->x - data->x; data->y = event->y - data->y;
    if (data->box == NULL) data->box = data->rootBox;

    Highlight(data->box); // unhighlights old selection, if any.
    if (event->type == ButtonUp) SetSelection(data->rootBox, data->box);
}

FindSmallestBox(Box& box, int& x, int& y)
// Finds the smallest box containing (x,y).
// 'box' is set to the smallest box.
// In/Out invariant: (x,y) is the cursor position relative to 'box'.
{
    if (box == NULL) return;
    if (InsideBox(box, x, y)) FindSmallestChildBox(box, x, y) // Inside child?
    else {
        x += box->x; y += box->y; box = box->parent;
        FindSmallestBox(box, x, y); // Inside parent?
    }
}

FindSmallestChildBox(Box& box, int& x, int& y)
// (x,y) is inside of 'box' -- recursively check its children
{
    int xBox = x; int yBox = y;
    for (Box *child=box->children; child != NULL; child=child->sibling) {
        int xChild = xBox - child->x; int yChild = yBox - child->y;
        if (InsideBox(child, xChild, yChild)) { // Inside child
            FindSmallestChildBox(child, xChild, yChild);
            box = child; x = xChild; y = yChild;
            return;
        }
    }
    // (x,y) is *not* inside of any of box's children
}

```

Figure 6.2: Incremental Selection Algorithm for Point method



```

SelectionEvent(Event *event, SelectionData *data)
// Called by the window system when a mouse event occurs in this window.
// (x,y) is the cursor position relative to the window.
// 'data' is structure containing information about the box in this window.
{
    if (event->type == ButtonDown) {
        data->startBox = data->rootBox;

        // Convert data x,y to box relative coords, find the smallest box,
        // and then convert data x,y back to box absolute coords.
        data->x = event->x - data->startBox->x;
        data->y = event->y - data->startBox->y;
        FindSmallestBox(data->startBox, data->x, data->y);
        data->x = event->x - data->x; data->y = event->y - data->y;
        if (data->box == NULL) data->box = data->rootBox;

        Highlight(data->startBox, data->startBox);
        return;
    }

    // Convert x,y to 'startBox' relative coords, and then
    // walk up the tree until we find a box that contains (x, y)
    int x = event->x - data->x; int y = event->y - data->y;
    Box *box = data->startBox;
    Box *previousBox = box;
    do {
        x += box->x; y += box->y;
        previousBox = box;
        box = box->parent
    } while (box != NULL && !InsideBox(box, x, y));

    Box *box1 = box; Box *box2 = box;
    if (box == NULL) {
        box1 = box2 = data->rootBox;
    } else {
        // look for a child that contains (x,y)
        for (Box *child=box->children; child != NULL; child=child->sibling) {
            if (InsideBox(child, x - child->x, y - child->y)) {
                box1 = previousBox;
                box2 = child;
                break;
            }
        }
    }

    Highlight(box1, box2); // unhighlights old selection, if any.
    if (event->type == ButtonUp) SetSelection(data->rootBox, box1, box2);
}

```

Figure 6.3: Selection Algorithm for Internal Rectangle method

ning time as do the first two algorithms, although its worst case behavior is the same (i.e., linear in the number of boxes). The basic problem is that the corners of the defining rectangle are not related to boxes in the tree that constrain the selected box(es). In addition, the locality assumption used in the **Point** algorithm is not helpful here because new (unrelated) boxes might be included in the new defining rectangle—the entire tree must always be searched. With some appropriate restrictions on how this method works (e.g., at most one selection is allowed), it is possible to use locality to improve the algorithm's performance.

The simplest algorithm uses a depth first search of the entire tree, checking each box to see if it is inside of the the defining rectangle. If a box is inside of the rectangle, then it is highlighted; its children are not searched. This algorithm can be improved upon by recognizing that if the intersection of a box and the defining rectangle is empty (i.e., they do not overlap), then none of the box's children need to be checked for inclusion. If only a single selection is allowed and the “unifying” rule mentioned earlier (the single selection is the smallest box containing all of the potential selections) is used, then searching the children of a box can be terminated early when a second child is found. This is the version of the **External Rectangle** algorithm given in Figure 6.4. If multiple selections are allowed, the algorithm shown is easily modified by storing all the selections found inside the loop into a set and returning that set instead of returning **box** when a second selection is found.

## 6.2 Non-Structural Selection and Multiple Selections

Structural selection can be limiting. For example, there is no simple way to grab  $\frac{3x}{2}$  from the expression  $\frac{3x+y}{2}$ . Genesereth[69] developed “circling” as an alternative approach to structural selection to solve this problem. When a user circles two boxes that are not structurally related, heuristics choose a connecting operator. For example, a simple heuristic is to find the smallest box containing all chosen operands and use its operator as the connecting operator if there are enough operands selected. In Figure 6.5a, the selection would be  $\frac{b}{d}$ . In Figures 6.5b and 6.5c, it is less clear what a user intends or what the system chooses ( $\frac{b}{d}$ ?  $b + d$ ?  $\frac{b}{?+d}$ ?). Genesereth proposes a number of hueristics and some meta-rules to choose between several valid possibilities. Ambiguity and complexity are two problems with circling. Another problem with circling is corner cutting: if the circle passes through a character, is the character considered to be inside or outside the circled region? Genesereth

```

SelectionEvent(Event *event, SelectionData *data)
// Called by the window system when a mouse event occurs in this window.
// 'data' is structure containing information about the box in this window.
// data(x,y) is the position of the cursor when the button was first pressed.
{
    if (event->type == ButtonDown) {
        data->x = event->x; data->y = event->y;
        return; // zero-width rectangle can't contain any boxes
    }
    Rect rect = (data->x, data->y, event->x - data->x, event->y - data->y);
    Box* box = FindContainedBox(data->rootBox, 0, 0, &rect);
    Highlight(box); // unhighlights old selection, if any.
    if (event->type == ButtonUp) SetSelection(data->rootBox, box);
}

Box* FindContainedBox(Box* box, int x, int y, Rect* rect)
// Finds the largest box that is completely contained within 'rect'
// (x,y) are the offsets of 'box' relative to 'rect'.
{
    if InsideRect(rect, box, x, y) return box;

    Box *selection = NULL;
    // Look for a child that is contained by 'rect'.
    // If more than one child is found, then "unify" by returning 'box'.
    for (Box *child=box->children; child != NULL; child=child->sibling) {
        int xChild = x - child->x; int yChild = y - child->y;
        if OutsideRect(rect, child, xChild, yChild) continue;

        Box* found = FindContainedBox(child, xChild, yChild, rect);
        if (!found) continue;
        if (selection != NULL) return box; // more than one selection
        selection = found; // first valid selection found
    }
    return selection;
}

```

Figure 6.4: Selection Algorithm for External Rectangle method

$$\begin{array}{ccc}
 \frac{a+b}{c+d} & \frac{a+b}{c+d} & \frac{a+b}{c+d} \\
 \text{(a)} & \text{(b)} & \text{(c)}
 \end{array}$$

Figure 6.5: Selection by Circling

proposes including the character(s) if inclusion helps makes sense of the circled expressions. Another idea is a user settable parameter that specifies what percentage of a character must be inside a circled region in order to consider it as being inside the region<sup>2</sup>.

A much simpler solution to this problem is to allow more than one selection. For example, in the problem presented above, the user first selects  $3x$ , then the division line, and finally 2. If multiple selections are allowed and the order of the selections is important, then the order of the selections should be made apparent to the user. Selecting  $3x$ , the division line, 2 and pasting the selections should result in  $\frac{3x}{2}$  whereas selecting 2, the division line,  $3x$  and pasting the selections should result in  $\frac{2}{3x}$ . If only a small number of selections are allowed, the ordering can be visually indicated by making the secondary and tertiary selections lighter shades of gray (or some color sequence). This approach is not satisfactory for a larger number of selections. Another approach is to explicitly number the selections, but this is not an aesthetically pleasing approach. A "show ordering" mode might be a good compromise.

Theorist[19] is the only system that allows multiple selections. However, multiple selections in Theorist do not function as described above. Instead, Theorist (usually) applies a command to each selection individually, making order irrelevant. This interpretation, coupled with the inability to select delimiters and make multiple selections within an expression limits Theorist's selection capability to structural selection.

Textual selection permits some types of selection that are not possible with structural selection. Textual selection works by operating on the leaves of the tree. If a parser is used, the ordering of the leaves should be same as what the parser expects. With textual selection, it is possible to select  $b + c$  from the expression  $ab + cd$ . The resulting selection should be the same as if multiple structural selections were made (in the example,  $b$ ,  $+$ , and  $c$ ) in order to mesh with the cut and paste mechanism.

---

<sup>2</sup>Recent work by Jackson and Roske-Hofstrand[89] indicates that circling is competitive with picking and may be superior when selecting more than two objects. However, their notion of circling does not require a closed circle and is more closely related to gestures[30] than to Genesereth's idea of circling. Furthermore, their experiment used large objects (1 by 2 inches) that did not intersect. Algorithmic limitations prevent small circles from being detected and also require that the "circle" be concave. These are both serious caveats that prevent the application of their ideas in this context.

## 6.3 Selection and Keyboard Input

A *standard user interface paradigm* is that if something is selected, then commands operate on the selection and text replaces the selection. If nothing is selected, a text cursor is displayed and text is inserted at the text cursor; commands either signal an error or are applied to a global scope (e.g., the entire expression).

MathScribe uses a parser for normal input and uses the above rule. MathScribe also uses overlays (Figure 2.4 and Section 5.2) in its forms menus, and typically binds keyboard equivalents to control keys to distinguish between keyboard text and keyboard commands. Overlays provide a convenient way to avoid typing parentheses that are normally required by a parser. For example, in order to obtain the expression  $\frac{x+y}{2}$  using a parser, it is necessary to type  $(x+y)/2$  and then remove the extra parentheses by some means<sup>3</sup>. Using overlays, a user can type  $x+y/2$ , where  $/$  invokes the division overlay<sup>4</sup>.

Most systems that use overlays either do not use the standard paradigm or seem to be confused about its application, perhaps because they mix text and commands into text only. However, if keystrokes such as  $+$  and  $/$  are treated as keyboard equivalents of menu commands, then there is no problem using the standard paradigm<sup>5</sup>. For example, if  $2x$  is selected in the expression  $x + 2x$ , then typing  $/$  results in  $x + \frac{2x}{1}$  and typing  $y$  replaces the selection resulting in  $x + y$ .

The standard paradigm can be improved upon for overlays for the case when there is no selection. Instead of automatically selecting the entire expression, the subexpression to the left of the text cursor can be selected and the overlay applied to that selection. This typically allows more rapid keyboard entry because a user does not need to select a subexpression before applying an overlay. For example, to obtain the expression  $x + \frac{2x}{3}$ , the user types  $x+2*xp/3$ , where the italicized characters indicate control characters ( $p$  means move to parent). Without this improvement, the user must type  $x+2*xjp/3$ , where  $j$  selects the subexpression to the left of the text cursor.

For the overlay model of input with automatic selection of the box to the left of the

<sup>3</sup>In MathScribe, removing the parentheses can be done via “unwrapping”, “reformatting”, or explicitly deleting the parentheses. MathCAD[113] is the only other system that uses a parser. MathCAD redisplay the expression without parentheses when the focus moves outside of the expression. Parentheses are always required in MathCAD because it does not use the line model of input (Section 5.4.1).

<sup>4</sup>Because nothing was selected, the entire expression is automatically selected.

<sup>5</sup>This is an over simplification. For example,  $+$  inside of a string should just be the character “+”, not the command to use the  $+$  overlay. The keyboard bindings must be made to be context sensitive.

text cursor, it is important for the text cursor to indicate the box to its left. For example, in the integral  $\int t dt|$ , the cursor ( $|$ ) gives no indication whether overlays are applied to the variable of integration or to the integral. Many times an indication can be given by making the size of the text cursor vary with the size of the box to its left. In the integral example above, that does not help. It is up to the designer of the notation to resolve possible ambiguities by ensuring that either a larger (taller) box is built for the integral (e.g.,  $\int t dt|$  versus  $\int t dt|$ ) or a box with extra space on left and/or right is built (e.g.,  $\int t dt|$  versus  $\int t dt |$ ) or both (e.g.,  $\int t dt|$  versus  $\int t dt |$ ). This problem can be avoided entirely by using an unconventional cursor that encloses the box to the left (similar to the way a selection is indicated). MathStation[115] uses a box cursor.

A mouse provides “random access” to the expression tree. Keyboard commands allow local motion within the tree, but are not confined to walking the tree one node at a time. Three useful ways to move around an expression using keyboard commands are:

**Structural** Structural commands allow the user to move from a node in the tree to one of its children, its siblings, or to its parent. In addition to local motion commands, some macro commands such as “move to the root” and “move to the left-most leaf” are also useful. Most interfaces provide these commands. Structural commands can be used with elision to focus in on the local structure of a large expression.

**Textual** Textual commands allow the user to move around in terms of characters and ignore the structure of the expression. They operate on the leaves of the tree. If a parser is used, the ordering of the leaves should be the same as what the parser expects. In particular, after entering a string of characters (which may include operators and local motion commands), erasing those characters (backspacing or deleting) should be the inverse operation. Many common textual commands such as “move to the first character” and “delete a word” (name or operator) are easily implemented. Some commands such as “delete a line” are more problematical but can be handled if the interface uses the line model of input (Section 5.4.1).

**Geometrical** Geometrical commands allow the user to move up/down or left/right independent of the structure of the expression. Geometrical commands work best for low resolution grids such as character display terminals. Figure 6.6a shows an expression printed to character resolution. If  $x$  is selected, then moving to the right selects  $n$ ,

$$\begin{array}{c}
 1 \qquad 2 \\
 \text{---} + \text{-----} \\
 m \qquad 3 \\
 x \qquad n - 4 \\
 \text{-----} \\
 5 \\
 y
 \end{array}
 \qquad
 \frac{1}{x^m} + \frac{2}{y^{\frac{n^3-4}{5}}}$$

(a) Character Resolution

(b) High Resolution

Figure 6.6: An Expression drawn at two Different Resolutions

-, and 4 in that order. For higher resolution devices such as bitmapped terminals, this technique does not work well because of the large number of lines; typically there is no character exactly to the right or above another character. Figure 6.6b shows the same expression printed to a much higher resolution. It is not clear what “move right” should do if  $x$  is selected—move to  $m$ ,  $+$ ,  $y$ , or the division line in the exponent of  $y$  (its center line is closest to the center line of  $x$ )? MathCAD, which uses fixed sized characters, supports these commands.

Both structural and textual motion are very useful. However, providing two sets of key bindings overflows the available keys. The following two observations can be used to merge both types of motion onto one set of key bindings: if a subexpression is selected (i.e., something structural is selected), then motion tends to be structural; if nothing is selected (i.e., a text cursor is displayed), then motion tends to be textual. MathScribe uses these two observations: keyboard traversal is structural if a subexpression is selected and textual otherwise. This seems to work well in practice.

## Chapter 7

# Sharing and other Optimizations

This chapter presents optimizations that can decrease the amount of storage used and also can increase the performance of the interface. The most important optimization for a CAS interface is to use DAGs instead of trees in order to avoid recomputing formats of common subexpressions. Other optimizations include: storing and drawing strings instead of characters when possible; caching the strings or their bitmaps in order to avoid redrawing; and lazy formatting. Optimizations used by other interfaces have not been described in the literature. However, conversations with authors of several other interfaces indicate that none of these optimizations have been tried. In particular, no interface other than MathScribe has used DAGs. These optimizations, along with some storage management issues, are discussed in the remainder of this chapter.

### 7.1 Sharing

Symbolic manipulation of expressions often results in shared subexpressions. For example, consider the simplification:

$$\frac{d}{dx} \sin(x) \cdot e^x \rightarrow \cos(x) \cdot e^x + \sin(x) \cdot e^x \quad (7.1)$$

Among the subexpressions present in the simplification:  $e^x$  occurs three times,  $\sin(x) \cdot e^x$  occurs twice, and the token  $x$  occurs seven times. Taking advantage of the sharing can result in space and time savings. For some algorithms, such as inverting a matrix, the savings can be quite dramatic because each element of the matrix is divided by the (often



large) determinant of the matrix<sup>1</sup>. In a test described below involving matrix inversion, a DAG representation decreased memory consumption by more than 95% as compared to a tree.

Because of shared subexpressions, algebra systems manipulate DAGs, not trees<sup>2</sup>. However, the formatting/display subsystems of current CASs and mathematical interfaces treat expressions as trees. As described in Chapter 4, displaying an expression consists of two passes over the expression. The first pass determines the size and position of the subexpressions; the second pass prints them out. Displaying an expression on the screen obviously requires traversing the expression as a tree (shared subexpressions must be displayed more than once), but the formatting phase can share subexpressions except in the presence of line breaks. Because the formatting phase in current algebra systems does not use DAGs, formatting an expression can take substantially longer than the actual computation. In the worst case, it can take an exponential amount of time in the number of nodes in the DAG (see page 93).

Although DAGs are *theoretically* superior to trees for expression formatting, it is not immediately obvious that *in practice* DAGs are superior; much of the complexity in MathScribe is a direct result of using DAGs and not trees. In order to determine exactly how much sharing occurs in displaying expressions, MathScribe was instrumented to collect data about a session. Data was collected from three sample runs. Two of the samples were batch files and one was an interactive session.

The first sample run was a standard Reduce test file that is used to demonstrate some of the capabilities of Reduce. This file contains some examples of expansion, substitution, differentiation, integration, factorization, and matrix manipulation. Because these are short demonstrations, this likely represents a conservative estimate of the amount of inter-expression sharing normally present.

The second sample was an interactive session that attempted to find a filter response as a function of frequency and to plot the result. The session was about an hour in length and at the end, twenty strips were still active and included in the count. Some strips contained moderately large expressions. Results gathered from other interactive sessions

---

<sup>1</sup>In MACSYMA, it is possible to set a flag so that the determinant is factored out of the inverse, resulting in a much smaller inverse.

<sup>2</sup>Most CASs' algorithms treat the DAGs as trees and the sharing that occurs is accidental (but frequent) as in the example above. However, some algorithms in some systems do use the DAG structure. For example, the MACSYMA simplifier tags expressions as "simplified" in order to avoid resimplification. Maple always uniquely stores subexpressions.

indicate that the amount of sharing found in this sample is typical of an interactive session.

The third and last sample involved finding the inverse of a 5 by 5 Vandermonde matrix. The inverse has a large amount of sharing in it because many of the denominators are the same. The size of the inverse dwarfs any contributions by the few other expressions in this sample. This sample represents an extreme for the amount of sharing possible.

In MathScribe, each strip is undoable and an undo box is stored for each strip. For the two batch samples, the undo boxes were not included in the counts; the undo boxes were included in the count for the interactive sample. Inclusion of undo boxes does not significantly affect the counts for sharing, but roughly doubles the counts when there is no sharing. As mentioned in Section 5.4.1, invisible parentheses surround many boxes in MathScribe. Because some implementations may not use invisible parentheses, two sets of counts are shown: one including the invisible parentheses and one excluding the invisible parentheses. The counts are given in Tables C.1–C.10.

In order to share boxes, the boxes must be split into two separate parts: a position independent part and a position dependent part. These parts are referred to as Boxes and PBoxes respectively and their contents are described in more detail below and in Appendix B. The position dependent part may be computed in a number of different ways that affects shareability. This is discussed below.

Figure 7.1 shows the Boxes and PBoxes used to represent the right hand side of Equation 7.1 when Boxes are shared<sup>3</sup>. If PBoxes are also shared, and if *cos* and *sin* are the same width, then the PBoxes for  $(x)$ ,  $\cdot$ , and  $e^x$  can also be shared. However, these boxes probably cannot be shared because it is unlikely that *cos* and *sin* have the same width when variable-width fonts are used.

For both Boxes and PBoxes, two types of sharing are possible: sharing within expressions (*local sharing*) and sharing across all expressions (*global sharing*). These types of sharing are compared against no sharing of subexpressions (i.e., a tree) in the tables. Data was collected for all the possible combinations of sharing of both Boxes and PBoxes. For each type of sharing, four numbers can be computed: the number of Boxes used, the number of PBoxes used, the number of references to Boxes used, and the number of references to PBoxes used. These numbers are sometimes related to each other and these relations are noted below.

---

<sup>3</sup>Because of space limitations, Figure 7.1 only shows the right hand side of Equation 7.1 down to the token level. A complete figure would also include the character Boxes and PBoxes that make up each token.

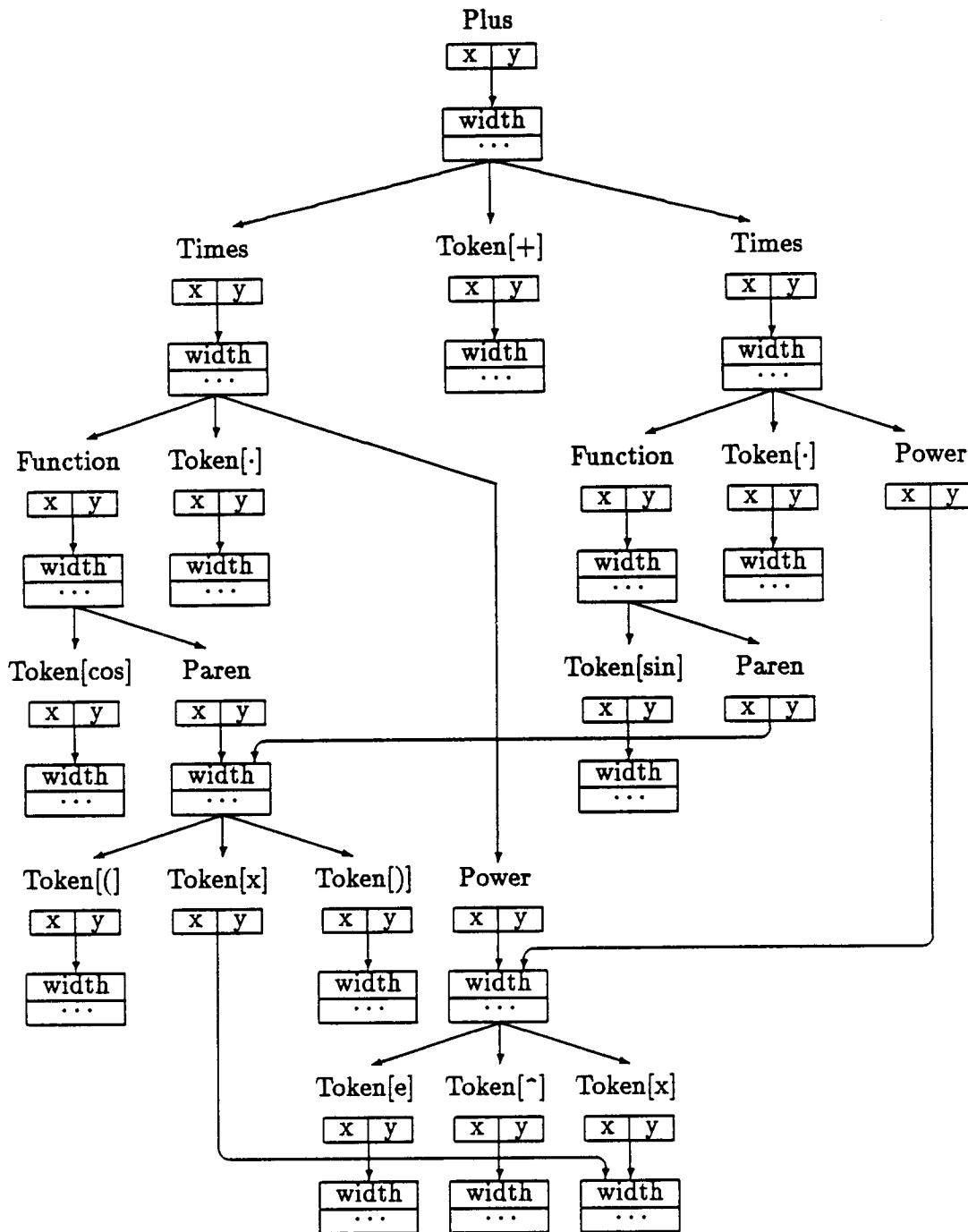


Figure 7.1: Shared Boxes for  $\cos(x) \cdot e^x + \sin(x) \cdot e^x$

- Boxes are pointed to by PBoxes. This means that the number of references to Boxes is equal to the number of PBoxes. Therefore, Box references are not explicitly shown in the table.
- If PBoxes are not shared, then the number of PBoxes is equal to the number of references to the PBoxes, and so the latter line is not shown in the tables in these cases.
- If Boxes are not shared, then the number of Boxes is equal to the number of PBoxes and so is not shown in the tables for these cases.

It should be noted that if Boxes are shared, then some implicit sharing of PBoxes occurs even if PBoxes are not explicitly shared. This is because the children of a shared Box are PBoxes and thus are shared too. The same is true of shared PBoxes and unshared Boxes. The only case in which there is no sharing is that of unshared Boxes and unshared PBoxes—this corresponds to a tree.

For each type of sharing, the counts are broken down into three categories of boxes: *internal*, *token*, and *character* boxes. These are described below. This breakdown allows for comparison and analysis of alternative implementations; some implementations may use different amounts of memory for these categories or may avoid character boxes completely.

**internal** Nodes of the DAG that correspond to non-trivial subexpressions. For example, in Expression 7.1,  $\cos(x)$  and  $\sin(x) \cdot e^x$  are two internal nodes.

**token** Nodes representing variables, numbers, delimiters, and other tokens. Tokens are usually composed of individual characters. For uniformity's sake in MathScribe, even single character tokens such as "x" have children (in this case, the character "x"). The right hand side of Expression 7.1 contains the tokens: *sin*, (, *x*, ), ·, *e*, +, and *cos*. Lines and special delimiters that can never be decomposed further (such as integral signs) are tokens.

**character** The characters that compose the tokens. These are what are actually drawn. The characters in the right hand side of Expression 7.1 are: *s*, *i*, *n*, (, *x*, ), ·, *e*, +, *c*, and *o*. Character boxes are useful for uniformity in the algorithms and because tokens may be composed of characters from different fonts (e.g., Greek letters and special symbols).

To summarize: Tables C.5–C.10 show the number of Boxes and parent-relative PBoxes used by all possible combinations of sharing (and not sharing) Boxes and PBoxes for three sample MathScribe sessions. For each session, two tables are shown: the first table includes invisible parentheses in the counts and the second table excludes them. Each count is broken down into internal nodes, token nodes, and character nodes to allow a more complete analysis of the counts. Data was also collected for two different sibling-relative representations of PBoxes for the Reduce (batch) sample session. This data is presented in Tables C.1–C.4.

Given the numbers in Tables C.1–C.10, it is possible to compute the amount of memory used by differing sharing scenarios. These computations, along with the percentage savings achieved because of sharing, are shown in Tables C.11–C.15. These computations assume that structures can be stored directly in other structures such as arrays. This is true for languages such as C and Pascal, but is not (normally) true for languages such as Lisp and Smalltalk which use pointer-based allocation for most data (i.e., an array of PBoxes in Lisp and Smalltalk consists of pointers to PBoxes, and thus uses more storage than the calculations show). In order to perform the calculations, the sizes of PBoxes and Boxes must be known. The (possible) contents of the PBox and Box data structures are given below. The actual contents vary depending upon the system and design. For example, if an object-oriented approach to design is taken, there is likely to be a base Box object and subclasses for InternalBox, TokenBox, and CharacterBox. CharacterBoxes do not need fields to store children (there are none), but may use fields to store font information.

PBoxes have the following fields:

**x, y** The  $x$  and  $y$  positions of the Box. Tables C.5–C.10 are computed with  $x$  and  $y$  relative to the parent as is typically done. A representation in which the  $(x, y)$  position is relative to the previous sibling's  $(x, y)$  position results in greater sharing of PBoxes. This is because of the likelihood that subexpressions following a delimiter will have identical positions relative to the delimiter. Data for this representation was computed for the Reduce session only and is shown in Tables C.1 and C.2. Even greater sharing can be achieved using a  $(x, y)$  position relative to the previous child's  $(x + width, y)$  position. Data for this representation was also computed only for the Reduce session and is shown in Tables C.3 and C.4. Sibling-relative representations can slow the drawing of large expressions because the  $x$  position of a box can only be computed by

scanning all of the elements before it.

**box** The unpositioned box. If boxes are shared, this is a pointer. If boxes are unshared, this structure is immediately present in the PBox structure.

Section B.1 discusses size considerations for these fields.

Unlike PBoxes, there are many different possible fields for a Box structure. Below is a list of likely fields and what they are used for:

**width** The width of the box. This number must be at least 32 bits because of the prevalence of long expressions.

**height, depth** The height of the box above and below the baseline. Without line breaking, boxes do not tend to be tall and these two fields could possibly be two shorts (16 bits).

**operator** A pointer to the "class" or "operator type" of the box. The class contains data common to all instances of this class such as precedence information, how the box is drawn, etc.

**children** The PBoxes contained within this box. If sharing is used, then the number of children never changes size and an array is a compact representation. If neither Boxes nor PBoxes are shared, then typing or editing can change the number of children and so a list may be a more preferable representation in this case. Appendix B discusses this more fully. In an object-oriented design, this field may not be present for characters and simple (single font) tokens. The memory usage calculations assume an array is used.

**refCount** The number of references to this Box. This field is not needed if boxes are unshared or if garbage collection is used instead of reference counting.

**algebra system form** A cache of the CAS's form. This is useful if the CAS uses simplification flags or uses static typing information. This field is not needed for characters.

**key** Used for hashing if the Boxes are shared.

**fontSize** The `fontSize` (actually nesting level) at this node. This information is stored for incremental update by the formatting routines.

**font** A pointer to the font information for this character or token.

**display string** This is used in tokens (possibly in combination with a font field) to speed up drawing. This field makes it unnecessary to traverse the children character boxes to draw the token. It is only useful for tokens that do not have characters from multiple fonts.

**parent** A pointer to the parent box. This can only be used if neither Boxes nor PBoxes are shared. A parent pointer greatly simplifies many algorithms.

In order to determine the amount of memory used by each scheme, we must know the amount of memory consumed by the Box and PBox structures. The calculations in Tables C.11–C.15 are based on the following assumptions.

- The basic word size is 32 bits. Pointers and integers are 32 bits.
- The PBox structure is three words long if Boxes are shared and is two words long if boxes are unshared. In the unshared case, Box structures are included directly in the PBox structure (not possible in Lisp as mentioned above). In this case, the size of a PBox is really two words plus the size of a Box structure, but the Box structure is counted separately.
- The Box structure can be as long as eleven words and as short as four words if only the fields width, height, depth, operator, and children are used and height and depth are only 16 bit fields. More realistic sizes range from six words to nine words. The tables assume seven words per Box. An object-oriented design might use different amounts of memory for internal Boxes, token Boxes, and character Boxes. For example, character Boxes have no need of a children field.

The amount of memory used per box substantially changes the amount of memory used. However, the percentage differences between the type of sharing shown in the tables does not change much more than 4% between Boxes whose sizes are six words and Boxes whose sizes are nine words.

Tables C.1–C.10 provide the raw data necessary to compute memory usage for alternative scenarios. Tables C.11–C.15 are the memory usage calculations. The remainder of this section analyzes the numbers in those tables. Unless otherwise stated, the numbers used are from the tables which do not include invisible parentheses in the counts.

MathScribe and most other systems use a parent-relative representation for the  $x$  and  $y$  fields of a box. If sharing is used, the tables show that a *width-relative* representation is 20% more space efficient than a parent-relative scheme. As noted earlier, this representation can slow down the drawing of large expressions. Also, although 20% represents a moderate space savings, compared to a tree representation, the width-relative representation saves only 3% more space than the parent-relative representation.

Sharing both Boxes and PBoxes typically results in the greatest space savings: 87–90% for the reduce session, 92% for the interactive session, and 96% for the matrix example. These numbers are determined by comparing the “Global Box, Global PBox” line to the “Unshared Box, Unshared PBox” line. This is a very significant savings.

Sharing an expression can add time and complication to an algorithm. Therefore, it makes sense to avoid sharing when its impact is not large. The next several paragraphs explore ways to share less.

If only Boxes are shared (as in MathScribe), the space savings over a tree representation are 86%, 91%, and 95% for the Reduce, Interactive, and Matrix examples respectively; the PBox representation is irrelevant here. These numbers are obtained by comparing the “Global Box, Unshared PBox” line to the “Unshared Box, Unshared PBox” line. Compared to total sharing, 7%–37% more space is used with this representation.

A similar pattern is true if only PBoxes are shared (“Unshared Box, Global PBox”). In this case however, the PBox representation is important. With a parent-relative representation, the savings are 82%, 88%, and 94%—somewhat less than if only Boxes are shared. However, if a width-relative representation is used, the savings are 90% for the Reduce session, which is greater than the Box-only case and are actually slightly greater than total sharing. This seeming contradiction is accounted for by realizing that if Boxes are not shared, then the Box can be part of the PBox structure instead of requiring an (extra) pointer to the Box structure.

Sharing requires that it must be possible to quickly determine whether two Boxes or PBoxes are equivalent. This is much easier to do with character and token boxes than with internal boxes. To compute the savings, we add together the amount of memory used for shared character boxes, for shared token boxes, and for unshared internal boxes and compare this to the memory required for a tree. Depending upon what type of sharing is used and which session is used, the memory savings are 75%–85% compared to a tree. Although this represents a very significant savings, total sharing saves an additional 30%–



|             | Memory Used (kbytes) |            |      | Percentages          |             |                     |
|-------------|----------------------|------------|------|----------------------|-------------|---------------------|
|             | Boxes Shared         |            |      | Token + Char<br>None | All<br>None | All<br>Token + Char |
|             | All                  | Token+Char | None |                      |             |                     |
| Reduce      | 88                   | 133        | 629  | 21%                  | 14%         | 66%                 |
| Interactive | 32                   | 55         | 372  | 15%                  | 9%          | 58%                 |
| Matrix      | 103                  | 321        | 2020 | 16%                  | 5%          | 32%                 |

Figure 7.2: Memory Used: Comparison of Internal Box Sharing

70% compared to sharing only tokens and characters. These numbers are summarized in Figure 7.2.

Eliminating character Boxes does not save much memory because the small number of different characters used are shared between all tokens, particularly in a width-relative PBox representation. For the Reduce demo file, eliminating character Boxes only saves 5% of the space required otherwise. The small savings in memory probably does not justify the complication to the selection algorithm. For tree-based implementations (i.e., when there is no sharing), the amount of space saved by eliminating character Boxes is large ( $\approx 45\%$  for the Reduce demo file).

Global sharing can cause some significant memory-management problems (see Section 7.3). Local sharing (sharing within a single expression) can solve some of those problems: when the expression is deleted, its hash table can be deleted. Local sharing results in a savings of 55%–65% as compared to a tree (the matrix session is dominated by a single expression and is not applicable here). Global sharing results in an additional 70%–75% savings over local sharing. These numbers were determined by using the “Global Box, Global PBox,” “Local Box, Local PBox,” and “Unshared Box, Unshared PBox” lines.

These results superficially contradict the results of Ponder[133]. Ponder studied the tradeoffs involved in hashing expressions (i.e., storing expressions uniquely) in MACSYMA. He concluded that hashing is not a good idea for MACSYMA because large common subexpressions are infrequent and because the underlying Lisp system already hashes atoms. There are several important differences between his work and this work. This data measures the amount of sharing in the final result, not in intermediate computation. More importantly for display systems, not only is sharing of subexpressions important, but so is sharing of tokens and characters.

| No Sharing  |          |        |       |         |             |
|-------------|----------|--------|-------|---------|-------------|
| Session     | internal | tokens | chars | int+tok | int+tok+chr |
| Reduce      | 2576     | 7672   | 7655  | 10248   | 17903       |
| Interactive | 955      | 2729   | 6910  | 3684    | 10594       |
| Matrix      | 7778     | 27607  | 22062 | 35385   | 57447       |

Figure 7.3: Box Counts as Trees

## 7.2 Drawing Speedups

This section presents several techniques to reduce the amount of time required to render an expression on the screen. Two important techniques were already covered in Section 4.1.2: never draw a box that is not visible (i.e., off the edge of the window) and order boxes from left-to-right (top-to-bottom) when possible so that invisible boxes can be rapidly discarded without needing to check their visibility. The following speedups can be used in combination with each other or with the above techniques. Only partial formatting was tried in MathScribe.

### 7.2.1 Avoiding Character Drawing

A simple way to speed up drawing is to draw tokens instead of characters whenever a token is composed of a single font, which is the typical case. Drawing tokens eliminates an extra level of recursion, and for most window systems, reduces the number of calls for multi-character tokens<sup>4</sup>. This idea was not implemented in MathScribe, but the data collected indicates the amount of speedup that can be expected. Because invisible parentheses are not drawn, only the counts without invisible parentheses are relevant. The “unshared Box, unshared PBox” line of Tables C.6, C.8, and C.10 are relevant (drawing involves a tree traversal) and these counts are summarized in Figure 7.3. If all tokens are composed of a single font, then drawing tokens eliminates 43%, 65%, and 38% of the recursive calls for the three sample sessions respectively. The only sample session that involves a significant number of multi-character tokens is the interactive session, which contains a number of calls to Reduce functions as well as *sin* and *cos*. For this session, the number of Draw commands can be reduced by as much as 60%.

<sup>4</sup>Most windowing systems support a string drawing command that draws several characters at once.

### 7.2.2 Caching the Draws

Drawing an expression involves sending Draw commands to the window system to render the characters, lines, etc., onto an invisible bitmap, and then copying that bitmap onto the screen (see Section 4.1.2). This presents two opportunities for speeding up subsequent draws: storing either the drawing commands or the bitmap. These two ideas are discussed below. The commands or bitmap can be stored either locally (with the strip in MathScribe) or in a global cache. In either case, if the expression changes or if its position relative to the window changes and the entire expression was not drawn the first time, the cache must be invalidated.

Storing the drawing commands or bitmap presents a standard time versus space tradeoff. To mitigate the memory expense, the Box data structure for the expression can be thrown out if a bitmap exists; only the CAS form needs to be kept around. For the active strip, it is a good idea to keep the Box data structure around so that the structure does not need to be regenerated during selection or reparsing on each keystroke.

Storing and redrawing the bitmaps is obviously faster than storing the draw commands and reexecuting them. However, bitmaps consume more space. For example, the bitmap for Equation 7.1 in MathScribe requires 972 bytes, whereas the simple encoding of string and line drawing commands requires only 368 bytes (288 bytes if the Font field is factored out). Drawing commands typically require the position (which can be 16 bits), the string to draw, and the font in which to draw the string. Drawing commands for lines and variable height symbols are more complicated, but occur less frequently. Factoring out the font information (i.e., having separate lists for each font) saves space. Sharing is also possible, but since pointers (or even short indices) consume the same amount of space as four (or two) characters, it is unlikely that enough (if any) space can be saved to justify the extra complication.

Some measurements of how much memory bitmaps would use and how much memory can be saved by throwing out the Box data structure were collected in MathScribe and are shown in Figure 7.4. A substantial amount of memory can be saved if we limit the bitmap's width to the width of the window in which it is contained. In use, MathScribe windows tend to be less than 600 pixels wide. Limiting the bitmap's width in this case uses less 1/3 of the space of unlimited bitmaps. As Figure 7.4 shows, even large limits such as 1200 pixels (typical width of a large screen) save substantial amounts of memory.

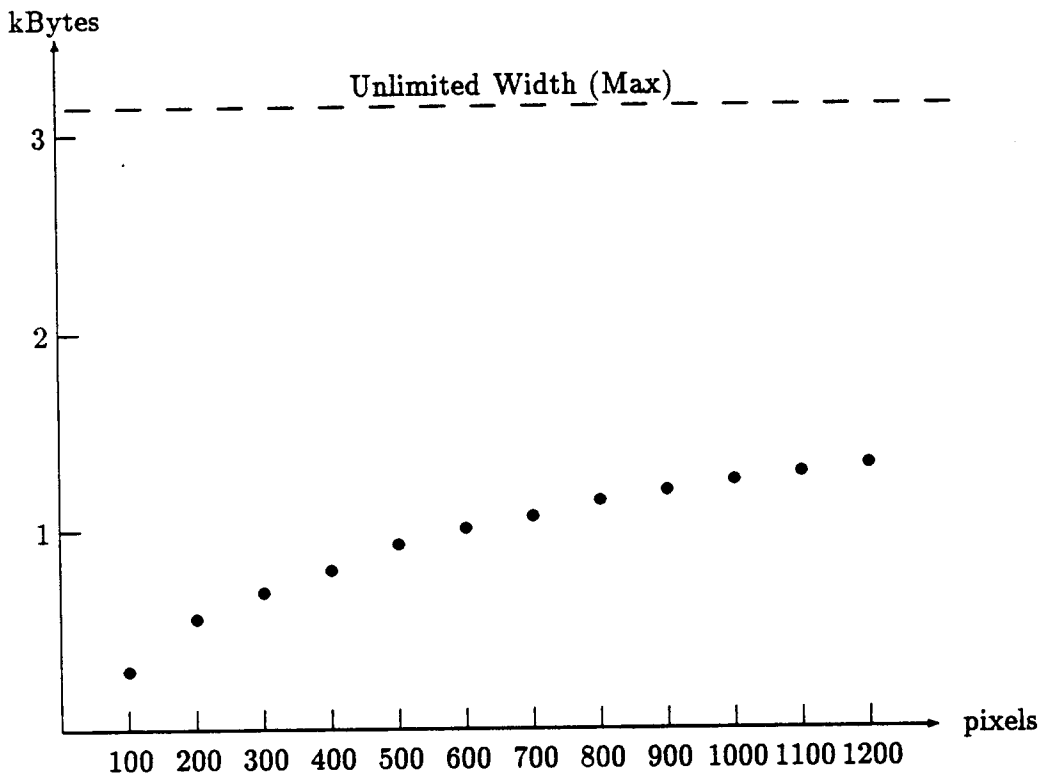


Figure 7.4: Average Number of KBytes/Strip vs. Width of Strip

The drawback to limiting the width is that horizontal scrolling requires regeneration of the bitmap.

### 7.2.3 Partial Formatting

*Partial formatting* is an idea that can be used to speed the formatting and initial display of a large expression by formatting only those parts of the expression that are visible in the window. Partial formatting can also be used to conserve space.

Formatting an expression (as in Figure 3.8) consists of recursively formatting the children to determine their width and height and, using that information, positioning the children relative to the current expression's position. In order to use partial formatting, the algorithm in Figure 3.8 must be modified as follows.

- The  $(x, y)$  position at which the expression is to be drawn, along with a clipping rectangle the size of the containing window, must be passed to `ExprToBox`.
- A new set of formatting procedures must be added that take expressions (not boxes) as arguments. In those new procedures that order the boxes from left-to-right or top-

to-bottom, the current  $(x, y)$  position is kept up-to-date and a check is made to see if the next subexpression to be formatted will be completely outside of the clipping rectangle. If so, the subexpression is not formatted and a special `UnformattedBox` is built that has zero width, depth, and height. These formatting procedures differ from the ones described in Chapter 4 in that they intermingle forward-translation with formatting.

`UnformattedBoxes` have a special drawing function associated with them. A box's drawing function is invoked whenever the box must be drawn (i.e., when it is visible). The drawing function for an `UnformattedBox` first calls the normal formatting routines to expand the unformatted expression as much as necessary. It then calls the incremental update algorithm (Figure 4.3) to update the format of its ancestors, etc. Finally, the drawing function "returns" to the top-level so that the new (expanded) root box is drawn correctly.

Partial formatting was implemented in `MathScribe` but was abandoned because of its complexity and because of its anomalous behavior with respect to horizontal scrolling. The complexity stemmed from the formatting, display, and parsing subsystems having to handle an additional data type (expressions instead of boxes). The selection mechanism was not complicated by `UnformattedBoxes` because they were not visible and hence, not selectable. The anomalous behavior with horizontal scrolling occurs for two reasons. The first reason is that the width of an expression that contains `UnformattedBoxes` is not accurate and will change as `UnformattedBoxes` are expanded. The second reason is that scrolling was *sometimes* subject to long delays due to the need to expand `UnformattedBoxes` when they become visible. Large differences in drawing time can be particularly frustrating in interactive applications[148].

An idea that is similar to partial formatting is automatic elision. Elision is discussed in Section 4.3.2. If an expression exceeds a width or depth bound, then the parts of the expression past that bound are elided (using "..."). Because these parts are not displayed, they do not have to be formatted and so the ideas mentioned above about partial formatting are applicable. Whereas the complexity issue is still valid (although drawing is not affected), anomalous scrolling behavior is not a problem because the elided boxes are only expanded under user control.

### 7.3 Storage Management

Even though MathScribe is written in a Lisp that provides garbage collection, storage management is still a problem. This is because every expression is stored in a hash table so that the uniqueness of a subexpression can be determined. However, this causes a problem for garbage collection because even if the expression is no longer referenced as a subtree, it is still contained in the hash table and thus, is not collected. *Weak pointers*[120] solve this problem. A weak pointer is a pointer to an object that the garbage collector does not follow when determining what data is reachable; the pointer is invalidated if the object is collected. Unfortunately, very few Lisp systems implement weak pointers in their garbage collectors so manual storage management of the hash table and its contents is usually required.

One method to do this is with reference counting. Reference counting has the advantages of no pause time and minimal storage waste (i.e., when a box is no longer needed, it is freed immediately). However, reference counting wastes an extra slot in the Box structure and requires careful analysis of the algorithms to determine when a Box's count must be incremented or decremented.

Another method is to basically duplicate the garbage collector but ignore the hash table reference. The major drawback to these schemes is that they can cause a significant pause. However, this time can be "hidden" by tacking the garbage collection onto the end of a computation by the algebra engine.

Collecting the boxes in the hash table requires finding all top-level references to a box and tracing all of their references. The two basic garbage collection algorithms are "mark and sweep" and "stop and copy" although there are many variants on these algorithms[35]. Both algorithms require that a box that is visited be marked. The stop and copy collector has a particularly nice implementation for MathScribe.

1. A new hash table is created (not necessarily the same size as the previous hash table). The new size can be a function of either the number of "live" boxes from the previous pass or a function of the number of boxes in the current hash table or some combination of the two.
2. To determine whether a box has been marked, we merely need to see if the box is in the new hash table using a pointer comparison for the equality test.

3. In a language that is garbage collected, we need only move the global pointer to the hash table from the old hash table to the new hash table. In a non-garbage collected language, we must first free all the non-referenced boxes. We could do this in one of two ways:

- For all boxes in the old hash table: if the box is not in the new hash table, free the box.
- In step 2 above, when we mark a box, we remove it from the old hash table and insert it in the new table. This means that every box remaining in the old hash table can be freed.

The pause time issue can be lessened by using a generation-based technique[165]. This is particularly effective for boxes created when an expression is entered from the keyboard because their life times are short. Another technique for reducing the amount of garbage to be collected is to determine if a box is shared. If a box is not shared, then it can be changed (in particular, its list of children can be changed). To determine if a box is shared, we must not only know that only one box points to the box in question, but also that only one box points to each of its ancestors. Hence, this idea is really only applicable to a reference counted scheme.

# Chapter 8

## Conclusions

### 8.1 Summary

Central to this thesis is the assumption that notation is important to mathematicians. It is not only a shorthand, but it also helps in understanding, structuring, and solving problems. This thesis presents various problems and solutions associated with the input and display of mathematical notation. Some of these problems, such as large expressions, are mainly relevant in the context of an interface to a CAS. Not surprisingly, some of the results, such as sharing common subexpressions, may only be worthwhile in this context. Other problems, such as ambiguous notation, are common to all mathematical interfaces, and the solution provided by notation libraries is applicable in many domains.

Most of the results in this thesis are solutions to problems caused by the sheer size of expressions that CASs can generate. The new results in this thesis include incremental algorithms for input, selection, and display of expressions. These algorithms, together with the use of DAGs instead of trees to represent expressions, result in dramatic decreases in *both* time and space usage and is what sets MathScribe apart from other mathematical interfaces that have been developed. Below is a summary of the highlights of this thesis in the order that they were presented.

- Rule-based pattern matching can be used to transform the CAS's expression form into more standard mathematical notation and vice versa. If the rules are grouped into "notation libraries" that can be added and deleted from a rule table, then some of the ambiguity of mathematical notation can be overcome. From an implementation



standpoint, notation libraries help to separate the interface from the algebra engine and increase the interface's portability. By changing the rule table, more than one CAS can be used during a session. Notation libraries can be used to connect the interface to numeric and graphic "engines" also (Section 8.2.5).

- Large expressions are easily generated by CASs. A number of optimizations are given for drawing large expressions. An incremental formatting and screen update algorithm is also presented that is used whenever an expression is changed. In particular, it is called after every character is typed during expression entry.

Two incremental algorithms for line breaking are presented. However, these algorithms cannot be directly used by most parsers because the possible changes to the parse/display tree are outside the class of grammars accepted by most parsers. Instead, they must be used after the parse/display tree has been updated.

A small set of primitives are presented that can be used to display a wide variety of mathematical notations.

- Entering an expression can be roughly divided into two approaches: tree-based and text-based. The tree-based approach uses templates or variations on templates called overlays to construct the expression. Tree-based approaches tend to require the expression to be syntactically correct at all times. The text-based approach uses a parser to structure the expression. Because the expression must be displayed correctly after every character that is entered, the parser must be invoked on a per-character basis. Due to the potentially large size of expressions, incremental parsing is essential and an incremental algorithm for an extensible grammar is presented. Parsing does not require syntactic correctness, and some use of error tolerance is desirable, particularly for programming forms. The incremental algorithm developed is capable of error tolerance. The algorithm only deals with syntax; the semantics of the parse is how the display data structures are affected. An incremental expression/screen update algorithm was presented in Chapter 4 to handle the semantic changes.
- Several issues involving selection of expressions are explored and some fast algorithms to select subexpressions are presented.
- Sharing common subexpressions can save large amounts of space and time. Using DAGs typically results in a 90% reduction in the amount of space used compared

to a tree representation. For some special cases, even more dramatic space savings can be achieved. The space savings are often accompanied by similar savings in the time used to format the expressions. Unfortunately, sharing common subexpressions complicates the algorithms for input and selection of expressions. The algorithms for DAGs, along with measurements and analysis comparing the space usage of tree and DAG representations are presented.

## 8.2 Future Work

### 8.2.1 Direct Manipulation

A windowed environment should not only allow selection of subexpressions, it should also allow them to be moved and manipulated. For example, if a term in a sum is selected, it should be possible to move the selection to any place in the sum provided that addition is commutative. Similarly, it should be possible to move terms from one side of an equation to the other side of the equation.

MathScribe's direct manipulation capabilities are limited to selection, deletion, and copying. Two more recent interfaces, Milo[121] and Theorist[19], provide more extensive direct manipulation capabilities. They both allow the user to commute terms and apply the distributive and anti-distributive laws via direct manipulation. Theorist also provides direct manipulation equivalents to the `solve` and `substitute` commands of CASs. Neither system has a "cancel common terms in a quotient" command.

Correct implementation of these manipulations requires knowledge of the underlying types of the variables or expressions being manipulated. Because both Milo and Theorist have their own built-in algebra engine, this knowledge is available. Most CASs either do not keep this information or do not make it available.

### 8.2.2 Large Expressions

MathScribe has three features to help users comprehend large expressions: interactive horizontal scrolling, elision of unwanted detail, and renaming of (common) subexpressions. Automatic elision of detail via user settable variables (for both width and depth) was contained in early versions of MathScribe, but was dropped because it was found to be useless. Other techniques to help users comprehend large expressions are possible. An

interesting idea developed by Fumas related to elision is a fisheye view[63]. A fisheye view presents details close to the area of focus and elides details that are further away. For trees, Fumas suggests using the degree of interest (DOI) function:

$$DOI_{\text{fisheye}(tree)}(x, f) = -(d_{tree}(x, f) + d_{tree}(x, root))$$

where  $x$  is a node in the tree,  $f$  is the current focus, and  $d(x, y)$  is the path length distance from  $x$  to  $y$ . The first term in the above function makes nodes close to the focus more “interesting” and the second term makes nodes close to the root more interesting. This function works well for programs, but is not that useful for large expressions because large expressions tend to be wide and not deep (e.g., sums, products), and this function ranks all siblings not on the path from the focus to the root as equal. Hence, all of the terms in a sum are displayed or none of them are. To correct this problem,  $d_{tree}(x, f)$  could be made a function of the physical distance from  $x$  to  $f$  instead of (just) its distance in the tree. Also, the apriori importance component ( $d_{tree}(x, root)$ ) should be adjusted to recognize that the first and last terms of wide expressions are more important than the middle terms and that some delimiters are more important than others (e.g, “ $f$ ” and “ $lim$ ” are more important than “ $+$ ”).

### 8.2.3 Session Layout

A windowed environment for computation allows for a different and more sophisticated model of user interaction over the older “terminal-style” of interaction which required the user to always perform computations at the “bottom” of the current session. A window environment allows editing and computation at any point in a window and allows several windows (sessions) to be active.

The “terminal-style” enforces a linear flow of information from the beginning of the session (the logical top of the screen) to the end of the session (the logical bottom of the screen). The linear temporal flow of the session is particularly important when using assignments to variables and/or function definitions in a symbolic setting. Moses[125] discusses problems with assignment semantics and evaluation in a CAS. As Moses mentions in that paper, substitution semantics have been proposed as a replacement for assignment semantics by several people. Abdali, Cherry, and Soiffer[2] discuss several different substitution semantics models for a CAS. An interface that allows the user to change expressions

in-place and allows the user to move expressions around makes it difficult to tell what variable has what value. The elimination of assignment to variables would probably result in less confusion.

Theorist[19] solves the linear flow problem by displaying input lines as *assumptions* and output lines as *conclusions*. Theorist internally maintains the assumptions on which a conclusion is based. If an assumption is changed, then all conclusions dependent on the assumption are turned into assumptions. Assumptions and conclusions apply only to window in which they are contained; each window represents a different derivation.

#### 8.2.4 Help Systems

Documentation for CASs, particularly on-line documentation is limited at best. On-line documentation with cross-links (i.e., hyper-text) is very useful for CASs because they contain a very large number of commands and global flags. In addition to documentation, context-sensitive menus and command completion are useful.

Most CASs contain a programming language. Novice and infrequent users of CASs are often unfamiliar with the language syntax which leads to errors. Templates containing programming language constructs and error correcting parsers can be considered part of a help system.

CASs are very complex programs and are often used to try and solve large and complicated problems. Most CASs have several generic commands such as "solve" and "integrate". For complicated problems, these generic commands do not always find an answer. However, answers can sometimes be obtained by manipulating the problem before invoking the generic command (e.g., by a change of variable) or by calling a specialized function that makes extra assumptions about the problem. While there is no substitute for mathematical experience, a number of "tricks" can often be suggested by an automated consultant. The work listed at the end of Section 1.1 [65, 66, 67, 68] is directed at this problem.

#### 8.2.5 Graphics, Numerics, and Statistics

A very severe limitation of most present day CASs is that they represent isolated components in the set of computational tools that a scientist or engineer employs in his routine. Thus, it may often be necessary for a scientist to do symbolic computation on

one system, save the resulting expressions manually and incorporate them in a program for numerical computation on a different system, and then plot numerical results using yet another system. Worse, he will often need to cycle through these operations several times before obtaining a satisfactory solution. A much more desirable scientific computing environment is one in which the symbolic, numeric, and graphic facilities are integrated and presented via a uniform interface. Since mathematical expressions are the main objects of manipulation in each of the three components above, most design considerations for a CAS user interface apply also to the interface for such an integrated environment.

A computing environment of the kind envisaged above can be thought of as consisting of pluggable modules providing symbolic, numeric, graphic, and statistical analysis capabilities, all connected to an interface. The modules can reside on different processors remotely attached to the processor hosting the interface. The environment need not be limited to one instance of a particular module, but could include several instances and implementations. For example, one may prefer to use different CASs for different problem domains due to their individual strengths and weaknesses. One might even have several copies of the same CAS connected for parallel computations. The translation mechanism discussed in Chapter 3 allows for these possibilities.

Many of the structuring ideas in this thesis apply to these areas also. Just as we can divide a CAS into its "algebra engine" and its interface, we can divide out the graphics, numeric, and statistical engines. This division is sketched below.

**User Interface** For a graphics system, the user interface is responsible for presenting the plots, etc, on the screen. It is also responsible for providing easy control of the parameters of the plots. By having control of the screen, the user interface can easily provide for overlays of graphs, labeling, scaling, animation, and other effects. A good graphical interface allows for interaction with the graphs (e.g., pointing to the graph and finding out what point in space it represents), so the graphics engine should send graphical objects to the interface that are in "real world" coordinates.

Numerical and statistical libraries often have very long parameter lists. A good user interface would either provide templates (and defaults) or a help system. Additionally, it would help the user sort through and choose the appropriate numerical routine.

**Algorithms** These are the heart of the graphics and numeric engines and do not belong in the interface. However, there are gray areas. For example, hidden surface elimination

is often performed by painting a surface from back to front, allowing the nearer surfaces to obscure the more distant surfaces. The way an object is rendered is properly part of the user interface though. Hence, the user interface may need to know some simplistic graphical algorithms.

Another gray area is the resolution of the device. This is important for some plotting routines that use adaptive techniques. The user interface must pass resolution information to the graphics engine.

**Function Evaluation** The algebra engine should be responsible for function evaluation because the function to be evaluated may be defined by the user in the CAS's programming language or may contain built in functions of the CAS. This can be very inefficient relative to the numeric (etc.) engine performing the function evaluation because of the high cost of interprocess communication and interlanguage communication. Additionally, many CASs do not use hardware floating point operations because they must deal with (potentially) infinite precision. In many, if not most cases, it is possible to translate the function to be evaluated into the native language of the engine (e.g., Fortran, C, or a vendor-defined application language). Depending upon the engine, this translation can be either interpreted or compiled to obtain better performance.

This division, along with the adoption of standards for communication among the parts, would allow the user to to compose an "ideal" system consisting of the user's preferred algebra, graphics, and numerics packages. A step in this direction has been taken by MathStation[115] with its user-configurable numeric engines.

Standard protocols for communicating expressions are useful not only for interprocess communication, but also for external communication. Some work on the representation and transmission of mathematical expressions is reported on by Arnon[9, 11], but progress has been slow. If such a standard is adopted by the document processing community, the communication of mathematical results to other colleagues would become a possibility. An important fallout of the exchange is *active documents*: equations embedded in a document that can be extracted and manipulated. This was one of the primary motivating factors for implementing CaminoReal (discussed on page 13). Mathematica's notebooks are another example of an active document. The current lack of a standard prevents the exchange of equations between different systems.

### 8.2.6 Avoiding Recomputation

Some systems mark computations (subexpressions) with additional information to avoid costly recomputations. For example, MACSYMA[109] uses flags to indicate that an expression has been simplified (so the simplifier can skip this subexpression) or that an expression has already been factored (so the factorizer can skip this subexpression).

Another example is Scratchpad II[91]. Scratchpad II is a strongly typed system in which every expression belongs to some algebraic domain. The algebraic domain is inferred from what the user typed and the types of previous expressions that are contained in the newly entered expression[160]. Inferring the domain of an expression can be time consuming.

Because the results of one computation are frequently used as part of another computation, it is desirable to keep any special information about the expression around so that it can be sent back to the algebra engine when a computation is requested. This is true not only if a result of a previous computation is cut and pasted into a new computation, but also if the result of an old computation is edited and resent to the algebra engine. This requires the interface to know when the special information is valid, something it cannot do without knowing details about the information being stored. However, a large class of important information is computed bottom-up, and the invalidation procedure for this class is simple: all special information on the path from the changed node to the root is invalidated.

## 8.3 Conclusions

The focus of most implementation efforts in Computer Algebra during the last decade has been on algorithms, language-related issues, efficiency, and portability. With a few exceptions, interface design and implementation has been relegated to a secondary role. This has been partly due a shortage of resources and limitations of the existing hardware, and partly due to the perception that a good interface was just a matter of "bells and whistles". As others who have implemented interfaces have noted [9, 105, 177], implementing an interface is a considerable task. This thesis has shown that developing a good CAS interface capable of handling large expressions not only requires "software engineering", but also requires the development of new and efficient algorithms.

Excellent user interfaces are available for many engineering and business computing environments. They have contributed immensely to the widespread use of those systems as well as to increased user productivity. Document processors are a case in point. They have transformed the process of writing. The development of good user interfaces for CASs holds out the same promise for those who use mathematics.



# Bibliography

- [1] S. K. Abdali, G. W. Cherry, and Neil Soiffer. An Object Oriented Approach to Algebra System Design. In *Proc. 1986 Symposium on Symbolic and Algebraic Computation*, pages 24–30. ACM, July 1986.
- [2] S. K. Abdali, G. W. Cherry, and Neil Soiffer. Spreadsheet Computations in Computer Algebra. In *Proc. EuroCal '87*, 1987.
- [3] Tryg A. Ager, R. A. Ravaglia, and Sam Dooley. Representation of Inference in Computer Algebra Systems with Applications to Intelligent Tutoring. In *Proc. Computers and Mathematics*, pages 215–227. Springer-Verlag, 1989.
- [4] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, 18(6):333–340, June 1975.
- [5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [6] Richard J. Anderson. EXED: A Prototype Environment for Algebraic Computation. Unpublished Manuscript, Computer Science Dept, UC Berkeley, 1983.
- [7] Robert H. Anderson. *Syntax-Directed Recognition of Hand-Printed Two-dimensional Mathematics*. PhD thesis, Harvard University, 1968.
- [8] Werner Antweiler, Andreas Strotmann, and Volker Winkelmann. A T<sub>E</sub>X-REDUCE-Interface. *ACM SIGSAM Bulletin*, 23(2):26–33, April 1989.
- [9] Dennis Arnon, Richard Beach, Kevin McIsaac, and Carl Waldspurger. CaminoReal: An Interactive Mathematical Notebook. In *Proc. EP'88 International Conference on*

- Electronic Publishing, Document Manipulation, and Typography*, 1988. Also available as Technical Report EDL-89-1, Xerox PARC, 1989.
- [10] Dennis Arnon, Carl Waldspurger, and Kevin McIsaac. CaminoReal User Manual Version 1.0. Technical Report CSL-87-5, Xerox PARC, 1987.
- [11] Dennis S. Arnon. Report of the Workshop on Environments for Computational Mathematics. *ACM SIGSAM Bulletin*, 21(4):42-48, November 1987.
- [12] Rolf Bahlke and Gregor Snelting. Context-sensitive editing with PSG environments. In *Proc. International Workshop on Programming Environments*, Lecture Notes in Computer Science. Springer-Verlag, 1986.
- [13] Robert A. Ballance, Jacob Butcher, and Susan L. Graham. Grammatical Abstraction and Incremental Syntax Analysis. In *Proc. SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 185-198. ACM, 1988.
- [14] C. Batut, D. Bernardi, H. Cohen, and M. Olivier. *User's Guide to PARI-GP, Version 1.34*, October 1990.
- [15] Bauldry and Fiedler. *Calculus Laboratories With Maple*. Brooks/Cole, 1990.
- [16] David Bayer and Michael Stillman. The Design of Macaulay: A System for Computing in Algebraic Geometry and Commutative Algebra. In *Proc. 1986 Symposium on Symbolic and Algebraic Computation*, pages 157-162. ACM, July 1986.
- [17] Michael J. Beeson. Logic and Computation in MATHPERT: An Expert System for Learning Mathematics. In *Proc. Computers and Mathematics*, pages 202-214. Springer-Verlag, 1989.
- [18] Allan Bonadio. *Expressionist Reference Manual*. Allan Bonadio Associates, 1579 Doleres St., San Francisco, CA, 94110, 1987.
- [19] Allan Bonadio and Erik Warren. *Theorist Reference Manual*. Prescience Corporation, 814 Castro St, San Francisco, CA, 94114, 1987.
- [20] Alan Borning and Alan Bundy. Using Matching in Algebraic Equation Solving. In *Proc. 7th International Joint Conference on Artificial Intelligence*, pages 446-471, 1981.

- [21] S. R. Bourne and J. R. Horton. The Design of the Cambridge Algebra System. In *Proc. of the Second Symposium on Symbolic and Algebraic Manipulation*, pages 134–143. ACM, March 1971.
- [22] R. S. Boyer and J. S. Moore. A Fast String Searching Algorithm. *Communications of the ACM*, 20(10):762–772, October 1977.
- [23] W. S. Brown. The ALPAK system for Non-numerical Algebra on a Digital Computer—I. Polynomials in Several Variables and Truncated Power Series with Polynomial Coefficients. *Bell Systems Technical Journal*, 42(5):2081–2119, September 1963.
- [24] W. S. Brown, J. P. Hyde, and B. A. Tague. The ALPAK system for Non-numerical Algebra on a Digital Computer—II. Rational Functions of Several Variables and Truncated Power Series with Rational Function Coefficients. *Bell Systems Technical Journal*, 43(2):785–804, March 1964.
- [25] G. Brun, A. Businger, and R. Schoenberger. The Token-oriented Approach to Program Editing. *ACM SIGPLAN Notices*, 20(5):17–20, February 1985.
- [26] A. Bundy. Discovery and Reasoning in Mathematics. In *Proc. International Joint Conference on Artificial Intelligence*, pages 1221–1230, 1985.
- [27] Alan Bundy. *The Computer Modelling of Mathematical Reasoning*. Academic Press, Inc., 1983.
- [28] Michael G. Burke and Gerald A. Fisher. A Practical Method for LR and LL Syntactic Error Diagnosis and Recovery. *ACM Trans. On Programming Languages and Systems*, 9(2):164–197, April 1987.
- [29] G. Butler and J. Cannon. CAYLEY Version 4, The User Language. In *Symbolic and Algebraic Computation: International Symposium ISSAC '88 Proceedings*, Lecture Notes in Computer Science 358, pages 456–466. Springer-Verlag, 1988.
- [30] W. Buxton, E. Fiume, R. Hill, A. Lee, and C. Woo. Continuous Hand-gesture Driven Input. In *Proc of Graphics Interface*, pages 191–195, 1983.
- [31] Jacques Calmet and Denis Lugiez. A Knowledge-based System for Computer Algebra. *ACM SIGSAM Bulletin*, 21(1):7–13, February 1987.

- [32] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, and Stephen M. Watt. *Maple User's Guide*. WATCOM Publications Ltd., 1985.
- [33] Pehong Chen. *A Multiple Representation Paradigm for Document Development*. PhD thesis, EECS Dept, University of California, Berkeley, California, 1988. Report No. UCB/CSD 88/436.
- [34] Lewis C. Clapp and Richard Y. Kain. A Computer Aid for Symbolic Mathematics. In *Proc. AFIPS Fall Joint Computer Conference*, volume 24, pages 509–517. AFIPS Press, 1963.
- [35] Jacques Cohen. Garbage Collection of Linked Data Structures. *ACM Computing Surveys*, 13(3):341–367, September 1981.
- [36] Chris Cole and Stephen Wolfram. *SMP Handbook*. Caltech, 1981.
- [37] G. E. Collins. The SAC-2 Computer Algebra System. In *Proc. EuroCal '85, vol. 2*, Lecture Notes in Computer Science 204, pages 34–35. Springer-Verlag, 1985.
- [38] George E. Collins. The SAC-I System: An Introduction and Survey. In *Proc. of the Second Symposium on Symbolic and Algebraic Manipulation*, pages 144–152. ACM, March 1971.
- [39] J. Robert Cooke and E. Ted Sobel. *MathWriter*. Cooke Publications, Ithaca, New York, 1986.
- [40] Gene Cooperman. A Semantic Matcher for Computer Algebra. In *Proc. 1986 Symposium on Symbolic and Algebraic Computation*, pages 132–134. ACM, July 1986.
- [41] Jacques Corbin and Michel Bidoit. A Rehabilitation of Robinson's Unification Algorithm. In *Proc. Information Processing 83*, pages 909–914, 1983.
- [42] Richard M. Cowan and Martin L. Griss. Hashing—The Key to Rapid Pattern Matching. In *Symbolic and Algebraic Computation, Proc. EUROSAM '79*, Lecture Notes in Computer Science 72, pages 266–278. Springer-Verlag, 1979.
- [43] James H. Davenport and C. E. Roth. PowerMath—A system for the MacIntosh. In *Proc. 1986 Symposium on Symbolic and Algebraic Computation*, pages 13–15. ACM, July 1986.

- [44] *The DERIVE User Manual: Mathematical Assistant for Personal Computers, Version 2.* Soft Warehouse, Inc., Honolulu, Hawaii, 1990.
- [45] *Special Session on Symbolic Mathematical Systems and Their Effects on the Curriculum*, 1984. Available as ACM SIGSAM Bulletin, 18/19(4/1):3-62, November 1984/February 1985.
- [46] Carl Engelman. The Legacy of Mathlab 68. In *Proc. of the Second Symposium on Symbolic and Algebraic Manipulation*, pages 29-41. ACM, March 1971.
- [47] Joost Englefriet. Bottom-up and Top-down Tree Transformations—A Comparison. *Mathematicas Systems Theory*, 19(3).
- [48] Richard Fateman. T<sub>E</sub>X Output from MACSYMA-like Systems. *ACM SIGSAM Bulletin*, 21(4):1-5, November 1987.
- [49] Richard Fateman, Alan Bundy, Richard O'Keefe, and Leon Sterling. Commentary on: Solving Symbolic Equations with PRESS. *ACM SIGSAM Bulletin*, 22(2):27-40, April 1988.
- [50] Richard J. Fateman. The User-Level Semantic Matching Capability in MACSYMA. In *Proc. of the Second Symposium on Symbolic and Algebraic Manipulation*, pages 311-323. ACM, March 1971.
- [51] Richard J. Fateman. MAC-ED, an Interactive Expression Editor for MACSYMA. In *Proc. 1979 MACSYMA Users' Conference*, pages 336-343, 1979.
- [52] Peter H. Feiler and Gail E. Kaiser. Display-Oriented Structure Manipulation in a Multi-Purpose System. In *Proc. IEEE Computer Society's Seventh International Computer Software and Applications Conference (COMPSAC'83)*. IEEE, November 1983.
- [53] C. N. Fischer, D. R. Milton, and S. B. Quiring. Efficient LL(1) Error Correction and Recovery Using Only Insertions. *Acta Informatica*, 13(2):141-154, 1980.
- [54] C. N. Fisher, Gregory F. Johnson, Jon Mauney, Anil Pal, and Daniel L. Stock. The Poe Language-Based Editor Project. In *Proc. ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 21-29. ACM, April 1984.

- [55] J. Fitch. A Survey of Symbolic Computation in Physics. In *Symbolic and Algebraic Computation, Proc. EUROSAM '79*, Lecture Notes in Computer Science 72, pages 30–41. Springer-Verlag, 1979.
- [56] John K. Foderaro. Typesetting Macsyma Equations. Master's thesis, University of California, Berkeley, December 1978.
- [57] John K. Foderaro. *The Design of a Language for Algebraic Computation Systems*. PhD thesis, EECS Dept, University of California, Berkeley, 1983.
- [58] John K. Foderaro and Richard J. Fateman. Characterization of VAX MACSYMA. In *Proc. 1981 Symposium on Symbolic and Algebraic Manipulation*, pages 14–19. ACM, August 1981.
- [59] Gregg Foster. DREAMS: Display Representation for Algebraic Manipulation Systems. Technical Report UCB/CSD 84/193, UC Berkeley, April 1984.
- [60] *Frame Maker Reference Manual, Version 2.0*. Frame Technology Corporation, San Jose, California, 1989.
- [61] Paul Franchi-Zannettacci. Attribute Specifications for Graphical Interface Generation. Technical Report 937, INRIA, December 1988.
- [62] Inge Frick. SHEEP and Classification in General Relativity. In *Proc. EuroCal '85, vol. 2*, Lecture Notes in Computer Science 204, pages 161–162. Springer-Verlag, 1985.
- [63] George W. Fumas. Generalized Fisheye Views. In *Proc. CHI '86*, pages 16–23. ACM, April 1986.
- [64] Harald Ganzinger and Robert Giegerich. Attribute Coupled Grammars. In *Proc. ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 157–170. ACM, June 1984.
- [65] F. Gardin and J. A. Campbell. Tracing Occurrences of Patterns in Symbolic Computation. In *Proc. 1981 Symposium on Symbolic and Algebraic Manipulation*, pages 233–238. ACM, August 1981.

- [66] F. Gardin and J. A. Campbell. A Knowledge-based Approach to User-friendliness in Symbolic Computing. In *Computer Algebra—Proc. EuroCal '83*, Lecture Notes in Computer Science 162, pages 267–274. Springer-Verlag, 1983.
- [67] Michael R. Genesereth. An Automated Consultant for MACSYMA. In *Proc. 1977 MACSYMA Users' Conference*, pages 309–314, 1977. NASA CP-2012.
- [68] Michael R. Genesereth. The Difficulties of Using MACSYMA and the function of User Aids. In *Proc. 1977 MACSYMA Users' Conference*, pages 291–307, 1977. NASA CP-2012.
- [69] Michael R. Genesereth. The Use of Semantics in a Tablet-Based Program for Selecting Parts of Mathematical Expressions. In *Proc. 1979 MACSYMA Users' Conference*, pages 328–335, 1979.
- [70] C. Genillard and A. Strohmeier. GRAMOL: A Grammar Description Language for Lexical and Syntactic Parsers. *ACM SIGPLAN Notices*, 23(10):103–122, October 1988.
- [71] Carlo Ghezzi and Dino Mandrioli. Incremental Parsing. *ACM Trans. On Programming Languages and Systems*, 1(1):58–70, July 1979.
- [72] Carlo Ghezzi and Dino Mandrioli. Augmenting Parsers to Support Incrementality. *J. ACM*, 27(3):564–579, July 1980.
- [73] Gaston H. Gonnet. New Results for Random Determination of Equivalence of Expressions. In *Proc. 1986 Symposium on Symbolic and Algebraic Computation*, pages 127–131. ACM, July 1986.
- [74] Craig Graci, Jack Narayan, and Randy Odendahl. Bunny Numerics: A Number Theory Microworld. In *Proc. Computers and Mathematics*, pages 228–239. Springer-Verlag, 1989.
- [75] Susan L. Graham and Steven P. Rhodes. Practical Syntactic Error Recovery. *Communications of the ACM*, 18(11):639–650, November 1975.
- [76] J. M. Greif. The SMP Pattern Matcher. In *Proc. EuroCal '85, vol. 2*, Lecture Notes in Computer Science 204, pages 303–314. Springer-Verlag, 1985.

- [77] J. H. Griesmer and R. D. Jenks. SCRATCHPAD/1—An Interactive Facility for Symbolic Mathematics. In *Proc. of the Second Symposium on Symbolic and Algebraic Manipulation*, pages 42–58. ACM, March 1971.
- [78] A. Nico Habermann and David S. Notkin. *The Second Compendium of GANDALF Documentation*, chapter The GANDALF Software Development Environment. CMU Department of Computer Science, 1982.
- [79] Andrew. D. Hall. The ALTRAN System for Rational Function Manipulation—A Survey. In *Proc. of the Second Symposium on Symbolic and Algebraic Manipulation*, pages 153–157. ACM, March 1971. Also available in *Communications of the ACM*, 14(8), 517–521, August 1971.
- [80] Malcolm C. Harrison. Implementation of the Substring Test by Hashing. *Communications of the ACM*, 14(12):777–779, December 1971.
- [81] Anthony C. Hearn. *Reduce User's Manual, Version 3.0*. The Rand Corporation, April 1983.
- [82] Franz C. Heeman. Incremental Parsing in INFORM. Technical Report IR-137, Free University, Amsterdam, The Netherlands, November 1987.
- [83] Carl W. Hoffman and Richard E. Zippel. An Interactive Display Editor for MACSYMA. In *Proc. 1979 MACSYMA Users' Conference*, page 344, 1979.
- [84] Carl W. Hoffman and Richard E. Zippel. MACSYMA Display-Oriented Expression Editor Preliminary Description and Command Summary. Handed out at 1979 MACSYMA Users' Conference, 1979.
- [85] Christoph M. Hoffmann and Michael J. O'Donnell. Pattern Matching in Trees. *J. ACM*, 29(1):68–95, January 1982.
- [86] J. R. Horgan and D. J. Moore. Techniques for Improving Language-Based Editors. In *Proc. ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 7–14. ACM, April 1984.
- [87] J. P. Hyde. The ALPAK system for Non-numerical Algebra on a Digital Computer—III. Systems of Linear Equations and a Class of Side Relations. *Bell Systems Technical Journal*, 43(4):1547–1562, July 1964.



- [88] Edgar Irons. Experience with an Extensible Language. *Communications of the ACM*, 13(1):31–40, January 1970.
- [89] Heffrey C. Jackson and Renate J. Roske-Hofstrand. Circling: A Method of Mouse-based Selection Without Button Presses. In *Proc. CHI '89*, pages 161–166. ACM, May 1989.
- [90] Fahimeh Jaliili and Jean H. Gallier. Building Friendly Parsers. In *Proc. 9th ACM Symposium on Principles of Programming Languages*, pages 196–206. ACM, 1982.
- [91] R. D. Jenks and B. M. Trager. A Language for Computational Algebra. In *Proc. 1981 Symposium on Symbolic and Algebraic Manipulation*. ACM, August 1981. Also available in SIGPLAN NOTICES, 16(11), November 1981.
- [92] Richard D. Jenks. A Pattern Compiler. In *Proc. SYMSAC '76*, pages 60–65. ACM, 1976.
- [93] S. C. Johnson. *UNIX Programmer's Manual*, volume 2b, chapter Yacc: Yet Another Compiler-Compiler. Bell Laboratories, second edition, 1978.
- [94] H. G. Kahrimanian. Analytical Differentiation by a Digital Computer. Master's thesis, Temple U., May 1953.
- [95] Gail E. Kaiser and Elaine Kant. Incremental Parsing without a Parser. *Journal of Systems and Software*, 5:121–144, 1985.
- [96] S. E. Keller, J. A. Perkins, T. F. Payton, and S. P. Mardinly. Tree Transformation Techniques and Experiences. In *Proc. ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 190–201. ACM, June 1984.
- [97] Brian Kernighan and Linda Cherry. *UNIX Programmer's Manual*, volume 2b, chapter Typesetting Mathematics—User's Guide. Bell Laboratories, second edition, 1978.
- [98] Donald E. Knuth. *The T<sub>E</sub>Xbook*. Addison-Wesley, 1984.
- [99] Donald E. Knuth and Michael F. Plass. Breaking Paragraphs into Lines. *Software—Practice and Experience*, 11(11):1119–1184, November 1981.
- [100] Frank Krausz. A Better User Interface for Symbolics Lisp Machine MACSYMA. *MACSYMA Newsletter*, 5(3):3–5, July 1988.

- [101] Leslie Lamport. *L<sup>A</sup>T<sub>E</sub>X: A Document Preparation System*. Addison-Wesley, 1986.
- [102] H. W. Lang, M. Schimmler, and H. Schmeck. Matching Tree Patterns Sublinear on the Average. Technical report, Dept. of Informatik, Univ Kiel, Kiel, W. Germany, 1980.
- [103] R. P. Leinius. *Error Detection and Recovery for Syntax Directed Compiler Systems*. PhD thesis, University of Wisconsin, Madison, 1970.
- [104] Wm Leler and Neil Soiffer. An Interactive Graphical Interface for Reduce. *ACM SIGSAM Bulletin*, 19(3):17-23, August 1985.
- [105] Benton Leong. Iris: Design of an User Interface Program for Symbolic Algebra. In *Proc. 1986 Symposium on Symbolic and Algebraic Computation*, pages 1-6. ACM, July 1986.
- [106] M. E. Lesk. *UNIX Programmer's Manual*, volume 2b, chapter Lex—A Lexical Analyzer Generator. Bell Laboratories, second edition, 1978.
- [107] R. Loos. The Algorithmic Description Language ALDES (Report). *ACM SIGSAM Bulletin*, 10(1):15-39, 1976.
- [108] *MacDraw Manual*. Apple Computers, Inc., Cupertino, California, 1984.
- [109] *MACSYMA Reference Manual*. Laboratory for Computer Science, MIT, tenth edition, January 1983.
- [110] A. Martelli and U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258-282, April 1982.
- [111] William Martin. *Symbolic Mathematical Laboratory*. PhD thesis, M.I.T., 1967.
- [112] William A. Martin. Computer Input/Output of Mathematical Expressions. In *Proc. of the Second Symposium on Symbolic and Algebraic Manipulation*, pages 78-89. ACM, March 1971.
- [113] *MathCAD 2.0*. MathSoft, Inc., One Kendall Square, Cambridge, MA, 02139, October 1987.
- [114] *MathScribe User's Manual, Version 1.0*. Tektronix Laboratories, 1988. 164 pages.

- [115] *MathStation, Version 1.0*. MathSoft, Inc., One Kendall Square, Cambridge, MA, 02139, April 1989.
- [116] Séamus McCague. Phototypesetting—An Art. In *Proc. First International Conference on Text Processing Systems*, pages 35–54. Boole Press Limited, October 1984.
- [117] Ellen McCarthy, Carolyn Holland, and Judith Lehman. *The Publisher User Manual*. ArborText Inc., Ann Arbor, MI, mic 3.2.7 edition, August 1987.
- [118] Kevin McIsaac. Pattern Matching Algebraic Identities. *ACM SIGSAM Bulletin*, 19(2):4–13, May 1985.
- [119] J. K. Millen. Charybdis: A Lisp Program to Display Mathematical Expressions on Typewriter-like Devices. Technical Report MTP-63, The MITRE Corporation, August 1967.
- [120] James S. Miller. *MultiScheme: A Parallel Processing System Based on MIT Scheme*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, August 1987.
- [121] *Milo User's Guide*. Paracomp, 123 Townsend St., Suite 310, San Francisco, CA, 94107, 1988.
- [122] M. L. Minsky. MATHSCOPE: Part I—A Proposal for a Mathematical Manipulation-Display System. Technical Report MAC-M-118, Artificial Intelligence Project, Project MAC, MIT, November 1963.
- [123] Joel Moses. Algebraic Simplification: A Guide for the Perplexed. In *Proc. of the Second Symposium on Symbolic and Algebraic Manipulation*, pages 282–304. ACM, March 1971. Also available in *Communications of the ACM*, 14(8), 527–537, August 1971 (Section on Lexicographic Ordering not in revised version).
- [124] Joel Moses. Symbolic Integration: The Stormy Decade. In *Proc. of the Second Symposium on Symbolic and Algebraic Manipulation*, pages 427–440. ACM, March 1971. Also available in *Communications of the ACM*, 14(8), 548–560, August 1971.
- [125] Joel Moses. The Variety of Variables in Mathematical Expressions. In *Proc. 1977 MACSYMA Users' Conference*, 1977. NASA CP-2012.

- [126] Yasutomo Nakayama. Mathematical Formula Editor for CAI. In *Proc. CHI '89*, pages 387–392. ACM, May 1989.
- [127] E. Ng. Symbolic-Numeric Interface: A Review. In *Symbolic and Algebraic Computation, Proc. EUROSAM '79*, Lecture Notes in Computer Science 72, pages 330–345. Springer-Verlag, 1979.
- [128] J. Nolan. Analytical Differentiation on a Digital Computer. Master's thesis, MIT, May 1953.
- [129] David Notkin. The GANDALF Project. *Journal of Systems and Software*, 5:91–105, 1985.
- [130] D. L. Orth. Linearizing Handwritten Mathematical Notation. In *Submitted to SIGPLAN '90 Conference on Programming Language Design and Implementation*. ACM, 1990.
- [131] Joseph F. Ossanna. *UNIX Programmer's Manual*, volume 2b, chapter Nroff/troff User's Manual. Bell Laboratories, second edition, 1978.
- [132] Eduardo Pelegri-Llopart. *Rewrite Systems, Pattern matching, and Code Generation*. PhD thesis, EECS Dept, University of California, Berkeley, 1987.
- [133] Carl Glen Ponder. *Evaluation of "Performance Enhancements" in Algebraic Manipulation Systems*. PhD thesis, EECS Dept, University of California, Berkeley, 1988. Report No. UCB/CSD 88/438.
- [134] Vaughan R. Pratt. Top Down Operator Precedence. In *Proc. 1st ACM Symposium on Principles of Programming Languages*, pages 41–51. ACM, 1973.
- [135] Paul Walton Purdom Jr. and Cynthia A. Brown. Tree Matching and Simplification. *Software—Practice and Experience*, 17(2):105–115, February 1987.
- [136] James Purtilo. Polyolith: An Environment to Support Management of Tool Interfaces. In *Proc. ACM SIGPLAN Symposium on Language Issues in Programming Environments*, pages 12–18. ACM, June 1985.

- [137] James Purtilo. Minion: An Environment to Organize Mathematical Problem Solving. In *Proc. 1989 Symposium on Symbolic and Algebraic Computation*, pages 147–154. ACM, July 1989.
- [138] Vincent Quint. An Interactive System for Mathematical Text Processing. *Technology and Science of Informatics*, 2(3):169–179, 1983.
- [139] Vincent Quint. Interactive Editing of Mathematics. In *Proc. First International Conference on Text Processing Systems*, pages 55–68. Boole Press Limited, October 1984.
- [140] R. Ramesh and I. V. Ramakrishnan. Nonlinear Pattern Matching in Trees. In *Proc. 15th INTL Colloquium on Automata, Languages and Programming (ICALP 88)*, Lecture Notes in Computer Science 317, pages 473–488. Springer-Verlag, July 1988.
- [141] Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental Context-Dependent Analysis for Language-Based Editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, July 1983.
- [142] A. Rich and D. Stoutmeyer. Capabilities of the MUMATH-78 Computer Algebra System for the INTEL-8080 Microprocessor. In *Symbolic and Algebraic Computation, Proc. EUROSAM '79*, Lecture Notes in Computer Science 72, pages 241–248. Springer-Verlag, 1979.
- [143] Jean E. Sammet. Survey of Formula Manipulation. *Communications of the ACM*, 9(8):555–569, August 1966.
- [144] Robert Scheifler et al. *CLX Interface Specification, Version 4*. MIT, 1987.
- [145] Robert Scheifler and Jim Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.
- [146] William F. Schelter. Sample  $\mathbb{N}F_{\oplus}\mathbb{R}$  Display. Unpublished Manuscript, Dept. of Mathematics, Univ. of Texas, Austin, 1987.
- [147] M. D. Schwartz, N. M. Delisle, and V. S. Begwani. Incremental Compilation in Magpie. In *Proc. ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 122–131. ACM, June 1984.

- [148] Ben Shneiderman. Response Time and Display Rate in Human Performance with Computers. *ACM Computing Surveys*, 16(3):265–285, September 1984.
- [149] Steven Skiena. *Implementing Discrete Mathematics*. Addison-Wesley, 1990.
- [150] J. R. Slagle. A Heuristic Program that Solves Symbolic Integration Problems in Freshman Calculus. *J. ACM*, 10(4):507–520, October 1963.
- [151] Carolyn Smith and Neil Soiffer. MathScribe: A User Interface for Computer Algebra Systems. In *Proc. 1986 Symposium on Symbolic and Algebraic Computation*, pages 7–12. ACM, July 1986.
- [152] M. D. Spivak. *The Joy of T<sub>E</sub>X: A Gourmet Guide to Typesetting with the A<sub>M</sub>S-T<sub>E</sub>Xmacro package*. American Mathematical Society, 1986.
- [153] Richard Stallman. Emacs, the Extensible, Customizable Self-Documenting Display Editor. Technical Report 519, Artificial Intelligence Lab, MIT, 1979.
- [154] *8010 STAR Information System Reference Library*. Xerox Corporation, 1984.
- [155] Lynn Arthur Steen, editor. Number 8 in Mathematical Association of America Notes, October 1988.
- [156] Leon Sterling, Alan Bundy, Lawrence Byrd, Richard O’Keefe, and Bernard Silver. Solving Symbolic Equations with PRESS. In *Computer Algebra—Proc. EUROCAM ’82*, Lecture Notes in Computer Science 144, pages 109–116. Springer-Verlag, 1982.
- [157] J. Steyaert and P. Flajolet. Patterns and Pattern Matching in Trees. *Information and Control*, 58:19–58, 1983.
- [158] H. Strubbe. Presentation of the SCHOONSCHIP System. In *Proc. of EUROSAM ’74*, pages 55–60, 1984. Available as ACM SIGSAM Bulletin, 8(3), August 1974.
- [159] P. Suppes, T. A. Ager, P. Berg, R. Chuaqui, W. Graham, R. E. Maas, and S. Takahashi. Applications of Computer Technology to Pre-College Calculus: First Annual Report. Technical Report 310, Institute for Mathematical Studies in the Social Sciences, Stanford, April 1987.

- [160] Robert S. Sutor and R. D. Jenks. The Type Inference and Coercion Facilities in the Scratchpad II Interpreter. In *Proc. SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 56–63. ACM, June 1987. Also available as SIGPLAN NOTICES, 22(7), July 1987.
- [161] Daniel C. Swinehart, Polle T. Zellweger, Richard J. Beach, and Robert B. Hagmann. A Structural View of the Cedar Programming Environment. *ACM Trans. On Programming Languages and Systems*, 8(4):419–490, October 1986.
- [162] Tim Teitelbaum and Thomas Reps. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Communications of the ACM*, 24(9), September 1981.
- [163] Warren Teitelman et al. *InterLisp Reference Manual*. Xerox Palo Alto Research Center, third edition, 1978.
- [164] P. H. Todd and G. W. Cherry. Symbolic Analysis of Planar Drawings. In *Symbolic and Algebraic Computation: International Symposium ISSAC '88 Proceedings*, Lecture Notes in Computer Science 358, pages 344–355. Springer-Verlag, 1988.
- [165] David Ungar and Frank Jackson. Tenuring Policies for Generation-Based Storage Reclamation. In *Proc. OOPSLA '88*, pages 1–17. ACM, 1988. Also available as SIGPLAN NOTICES, 23(11), November, 1988.
- [166] S. van Egmond, F. C. Heeman, and J. C. van Vliet. INFORM: an Interactive Syntax-Directed Formulæ-Editor. *Journal of Systems and Software*, 9:169–182, 1989.
- [167] J. A. van Hulzen and J. Calmet. *Computer Algebra Symbolic and Algebraic Computation*, chapter Computer Algebra Applications. Springer-Verlag, second edition, 1983.
- [168] A. van Wijngaarden, B. J. Mailloux, L. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. Revised report on the Algorithmic Language Algol68. *Acta Informatica*, 5:1–236, 1975.
- [169] Ben Wegbreit. The ECL programming system. In *Proc. AFIPS*, pages 253–262. AFIPS Press, 1971.

- [170] M. Wegman and C. N. Alberga. Parsing for a Structural Editor (part II). Technical report, IBM Research, January 1982.
- [171] Mark B. Wells. A Review of Two-Dimensional Programming Languages. In *Proc. of a Symposium on Two-Dimensional Man-Machine Communication*, pages 1–10. ACM, 1972. Also available in SIGPLAN NOTICES, 7(10), October 1972.
- [172] Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley, 1988.
- [173] Stephen Wolfram. *MathLink, The Mathematica Communication Standard, C Language Version*. Wolfram Research, Inc., preliminary edition, 1990.
- [174] John Xenakis. The PL/I-FORMAC Interpreter. In *Proc. of the Second Symposium on Symbolic and Algebraic Manipulation*, pages 105–114. ACM, March 1971.
- [175] Dashing Yeh. Automatic Construction of Incremental LR(1) Parsers. *ACM SIGPLAN Notices*, 23(3):33–42, March 1988.
- [176] Daniel M. Yellin and Eva-Maria M. Mueckstein. The Automatic Inversion of Attribute Grammars. *IEEE Transactions on Software Engineering*, SE-12(5):590–599, May 1986.
- [177] Douglas Young and Paul S. Wong. GI/S: A Graphical User Interface for Symbolic Computation Systems. *Journal of Symbolic Computation*, 4:365–380, 1987.



## Appendix A

# Translation Example

This appendix presents a complete example of translation rules (Chapter 3). For concreteness, we assume that we are translating from Mathematica format into the formatting primitives presented in Section 4.2.1 and vice-versa. A summary of Mathematica's relevant format is given in Figure A.1. The primitives are summarized in Figure A.2. The example presented here was written and tested using Mathematica V1.2 and its rule system.

The translations presented in this appendix handle the field of rational functions over the integers. Rational functions have the form  $p(x_1, \dots, x_n)/q(x_1, \dots, x_n)$ , where  $p, q \in \mathbb{Z}[x_1, \dots, x_n]$  with  $q(x_1, \dots, x_n) \neq 0$ . The translations can handle both factored and expanded polynomial forms. An example of an expression that is handled by the translations is:

$$-x - y + 3(1 + x)(x - y)^2 - \frac{(x - y)^2(4 + x^2 + y^2)}{(-1 + x)(x^2 + y^2)^2} + \frac{1}{2(x + y + z)^3}$$

There are numerous ways to display rational functions. For example, the expression  $-y/2$  can also be displayed as:  $-\frac{y}{2}$ ,  $\frac{-y}{2}$ , and  $\frac{-1}{2}y$  along with several other possible notations. As a demonstration of the ease of changing styles, translations that perform the first two notations are shown. This is an example of notation libraries.

After the Mathematica form is translated into the proper notation, the formatting engine turns the notation into **Boxes** (see Sections 4.1 and B.1). The formatting engine adds spaces and parentheses as needed; parentheses are removed when the internal format is converted to the formatting primitives for backward translation. The formatting engine also adds line breaks. For simplicity, this example leaves formatting tokens up to the formatting engine. In a real system, these would be calls to `Symbol` and `VariableSymbol`.

| Expression      | Mathematica Form                   |
|-----------------|------------------------------------|
| $v$             | <code>v</code>                     |
| $f[x]$          | <code>f[x]</code>                  |
| $x + y + \dots$ | <code>Plus[x, y, ...]</code>       |
| $x - y$         | <code>Plus[x, Times[-1, y]]</code> |
| $xy \dots$      | <code>Times[x, y, ...]</code>      |
| $x^n$           | <code>Power[x, n]</code>           |
| $1/x^n$         | <code>Power[x, -n]</code>          |
| $-nx$           | <code>Times[-n, x]</code>          |
| $m/n$           | <code>Rational[m, n]</code>        |
| $-m/n$          | <code>Rational[-m, n]</code>       |

Figure A.1: Mathematica Formats

Forward and backward translations rules have the same structure. Both work like a top-down recursive descent parser[5]. The major difference is that instead of a single look-ahead token being used, an entire pattern is used. Once a pattern is recognized, the subparts of the pattern are translated.

Rational functions consist mostly of sums and products. Because these are n-ary operations in Mathematica, the translations for handling them are more complex than ordinary rules and their backward translations are correspondingly complex. The existence of mapping functions would decrease the translations' complexity; the examples in this appendix do not use mapping functions. The power rules in Figures A.3 and A.7 are more typical of the rules for other operators and notations such as relational operators, etc.

## A.1 Forward Translations

There are three important features of Mathematica's internal form that should be mentioned in order to understand the forward translation examples. The first is that Mathematica canonicalizes `Times` by making numbers be the first term in a product. The second feature is that rational numbers are treated differently than integers and are represented as `Rational[m, n]`, with `m` negative if the rational number is negative. The presence of the `Rational` form adds several patterns that would not be needed otherwise. The last feature is that terms such as  $-x$  are represented as `Times[-1, x]` by Mathematica.

```

Horizontal(Op, {b1, ..., bn}, deltaX, deltaY)
Vertical(Op, {b1, ..., bn}, justification, base, baseIndex, deltaX, deltaY)
Script(Op, base, {(b1, p1), ..., (bn, pn)})
Super(Op, base, super)
Tensor(Op, base, {(b1, p1), ..., (bn, pn)})
Matrix(Op, m, n, align, open, close, {c11, ..., cmn})
Radical(Op, radicand, exponent)
Sqrt(Op, radicand)
HLine(width, extra)
VLine(height, depth, extra)
Symbol(font, character)
VariableSymbol(font, character, height, depth)

```

Figure A.2: Formatting Primitives

The remainder of this section discusses the translations in Figures A.3–A.6. These translations deal with tokens, powers, sums, and products—Mathematica treats denominators in a quotient as terms raised to the  $-1$  power. Applying the `Format` function to an expression produces its forward translation.

Figure A.3 shows translations for tokens, rational numbers, and powers. Not all of the power translations are needed for rational functions, but are included to provide more examples of formatting translations with fixed arity. They translate the forms:  $x^y$ ,  $x^{1/n}$ ,  $x^{1/2}$ , and  $x^{-n}$ , where  $n$  is a positive integer. The last translation is an example of using side conditions to restrict the pattern. Notice that once a pattern is recognized, `Format` is applied to the non-token subparts, thus providing top-down recognition.

The translations in Figure A.4 deal with sums. The sum translations use an auxiliary function `FormatPlus` to iterate through the terms of the sum, inserting  $+$  or  $-$  between the terms and applying `Format` to each term. `FormatPlus` uses the impossible value  $0$  as a signpost to determine when it is finished. Initially, there is nothing to the left of  $0$ , but as the iteration proceeds, parsed terms and delimiters are placed there until nothing remains on the right of the  $0$ . When this finally happens, the formatting primitive `Horizontal` is produced. The patterns recognized are:  $-m/n x$ ,  $-n x$ , and  $x$ .

The translations for products are similar in nature to the translations for sums, but are more complicated because the factors in the product must be separated into those

```
(*
 * Format tokens and rational numbers in the obvious manner
 *)
Format[x_] := x
Format[Rational[m_, n_]] := Horizontal[Rational, {m, "/", n}, 0, 0] /; m > 0
Format[Rational[m_, n_]] := Horizontal[Rational, {"-", -m, "/", n}, 0, 0] /; m < 0

(*
 * Some rules for powers.
 *)
Format[Power[x_, y_]] := Super[Power, Format[x], Format[y]]
Format[Power[x_, Rational[1, n_]]] := Radical[Format[x], n]
Format[Power[x_, Rational[1, 2]]] := Sqrt[Format[x]]
Format[Power[x_, n_]] := Vertical[Times, 1, HLine[1, 4], Format[Power[x, -n]],
    center, bottom, 2, 0, -3] /; IntegerQ[n] && n < 0
```

Figure A.3: Forward Translation: General and Power Rules

```
(*
 * Rules for summation:
 *   print in horizontal forms interspersed with "+" unless a term of
 *   the form '-n * x' is found, in which case a "-" is used as the delimiter
 *)

(* start 'Plus' matching off, initializing 'sum'
 * do not print initial plus if pattern does not match a form that uses "-"
 *)
Format[Plus[x_, y_]] := FormatPlus[0, x, y]
FormatPlus[0, x_, y_] := FormatPlus[Format[x], 0, y]

(* recursive case *)
FormatPlus[sum_, 0, Plus[x_, y_]] := FormatPlus[sum, 0, x, y]

(* patterns recognized *)
FormatPlus[sum_, 0, Times[Rational[m_, n_], x_], y_] :=
    FormatPlus[sum, "-", Format[Times[Rational[-m, n], x]], 0, y] /; m < 0
FormatPlus[sum_, 0, Times[n_, x_], y_] :=
    FormatPlus[sum, "-", Format[Times[-n, x]], 0, y] /;
    NumberQ[n] && n < 0
FormatPlus[sum_, 0, x_, y_] := FormatPlus[sum, "+", Format[x], 0, y]

(* all done -- output Horizontal *)
FormatPlus[sum_, 0] := Horizontal[Plus, {sum}, 0, 0]
```

Figure A.4: Forward Translation Summation Rules

```

(*)
* Rules for products -- collect positive and negative powers
* Formatting proceeds top-down, calling 'Format' when we place a term
*   in the numerator ('num') or denominator('denom').
* '0' is used as a marker to separate 'num' and 'denom' for matching purposes
*)

(* start 'Times' matching off, initializing 'num' and 'denom' *)
Format[Times[x_, y_]] := FormatTimes[0, 0, x, y]
Format[Times[Rational[m_, n_], y_]] := FormatTimes[m, 0, n, 0, y]

(* recursive case *)
FormatTimes[num___, 0, denom___, 0, Times[x_, y_]] :=
  FormatTimes[num, 0, denom, 0, x, y]

(* handle powers, placing them in 'num' or 'denom' *)
FormatTimes[num___, 0, denom___, 0, Power[base_, exp_], y___] :=
  FormatTimes[num, 0, denom, Format[Power[base, -exp]], 0, y] /;
  NumberQ[exp] && exp < 0
FormatTimes[num___, 0, denom___, 0, Power[base_, exp_], y___] :=
  FormatTimes[num, Format[Power[base, exp]], 0, denom, 0, y]
FormatTimes[num___, 0, denom___, 0, x_, y___] :=
  FormatTimes[num, Format[x], 0, denom, 0, y]

(* all done -- output quotient *)
FormatTimes[num_, 0, 0] := FormatTimesLine[num]
FormatTimes[0, denom_, 0] :=
  Vertical[Times, 1, HLine[1, 4], FormatTimesLine[denom],
    center, bottom, 2, 0, -3]
FormatTimes[num_, 0, denom_, 0] :=
  Vertical[Times, FormatTimesLine[num], HLine[1, 4], FormatTimesLine[denom],
    center, bottom, 2, 0, -3]

FormatTimesLine[line_] := line
FormatTimesLine[x_, rest_] := Horizontal[Times, {x, rest}, 0, 0]

```

Figure A.5: Forward Translation Product Rules

that belong in the numerator and those that belong in the denominator. The latter are distinguished by being powers with a negative exponent or being the denominator of a Rational. For example, the expression  $(xy^2)/(4z^3)$  has the Mathematica format:

```
Times[Rational[1, 4], x, Power[y, 2], Power[z, -3]]
```

In order to distinguish between what belongs in the numerator, what belongs in the denominator, and what has yet to be parsed, two zeros are used in the auxiliary function

```

Format[Times[-1, y_]] := FormatTimes["-", 0, 0, y]
Format[Times[Rational[1, n_], y_]] := FormatTimes[0, n, 0, y]
Format[Times[Rational[-1, n_], y_]] := FormatTimes["-", 0, n, 0, y]
Format[Times[n_, y_]] := FormatTimes["-", 0, 0, -n, y] /;
    NumberQ[n] && n < 0

```

(a) Minus Sign in Numerator of Fraction

```

Format[Times[-1, y_]] :=
    Horizontal[Times, {"-", FormatTimes[0, 0, y]}, 0, 0]
Format[Times[Rational[1, n_], y_]] := FormatTimes[0, n, 0, y]
Format[Times[Rational[-1, n_], y_]] :=
    Horizontal[Times, {"-", FormatTimes[0, n, 0, y]}, 0, 0]
Format[Times[Rational[m_, n_], y_]] :=
    Horizontal[Times, {"-", FormatTimes[-m, 0, n, 0, y]}, 0, 0] /;
    m < 0
Format[Times[n_, y_]] :=
    Horizontal[Times, {"-", FormatTimes[-n, 0, 0, y]}, 0, 0] /;
    NumberQ[n] && n < 0

```

(b) Minus Sign in Front of Fraction

Figure A.6: Forward Translation Product Rules for Leading Minus Sign

`FormatTimes` in manner similar to the way in which they were used in `FormatPlus`: numerators are collected before the first zero and denominators after it and before the second zero.

The first three patterns in Figure A.5 initialize `FormatTimes` and handle the recursion. The next three translations are used to recognize the patterns:  $base^{-n}$ ,  $base^{exp}$ , and  $x$ . The last three patterns handle translation to the formatting primitives. They deal with the cases of nothing in the denominator, nothing in the numerator, and something in both the numerator and denominator. These three patterns make use of the second auxiliary function `FormatTimesLine` which produces a `Horizontal` primitive if there is more than one factor in the term.

The translations in Figure A.5 do not handle minus signs. These translations were separated out so that the minus sign could be treated in two different ways: either as part of the numerator or in front of the fraction as in  $\frac{-x}{2y}$  and  $-\frac{x}{2y}$  respectively. Separating the translations into two sets, either of which can be chosen by a user, is a simple example of forming a notation library. The translations for handling the minus sign are shown in Figures A.6a and A.6b. Because Mathematica arranges products such that the first factor is a number, the translations that determine the placement of the minus sign need only

```

BFormat[x_] := x
BFormat[Horizontal[Rational, {m_, "/", n_}, 0, 0]] := Times[m, 1/n]
BFormat[Horizontal[Rational, {"-", m_, "/", n_}, 0, 0]] := Times[-1, m, 1/n]

(*
 * Some rules for powers.
 *)
BFormat[Super[Power, x_, y_]] := Power[BFormat[x], BFormat[y]]
BFormat[Radical[x_, n_]] := Power[BFormat[x], Power[n, -1]]
BFormat[Radical[x_, n_]] := Power[BFormat[x], Rational[1, n]]
BFormat[Sqrt[x_]] := Power[BFormat[x], Rational[1, 2]]
BFormat[Vertical[Times, 1, HLine[1, 4], Super[Power, x_, n_],
  center, bottom, 2, 0, -3]] :=
  BFormat[Super[Power, x, -n]]

```

Figure A.7: Backward Translation: General and Power Rules

check for a negative number once, before calling `FormatTimes`.

## A.2 Backward Translations

Ideally, the backward translations should be created from the forward translations or vice-versa. Unfortunately, of the work surveyed in Chapter 3, only the work of Yellin and Mueckstein[176] mentioned in Section 3.3 deals with reverse translation. Automatic creation of backward translations from forward translations does not appear to be too difficult for simple single-pattern forward translations; multiple-pattern translations appear to be more difficult.

Fortunately, it is not necessary for a backward translation of a forward translation to be the original expression, only a computationally equivalent form. For example, the following translation sequence is acceptable:

```

Times[-1, x, Power[y, -1]]
   $\xrightarrow{\text{Forward}}$  Horizontal[Times, {-, Vertical[Times, x, HLine[1, 4], y, ...]}, 0, 0]
   $\xrightarrow{\text{Backward}}$  Times[-1, Times[x, Power[y, -1]]

```

Mathematica simplifies the result expression to the first expression. Because only computational equivalence is necessary and because we do not have to worry about a separate `Rational` form, the backward translations are somewhat simpler than their forward counterparts.

```

(*)
* Rules for summation:
*   forms for sum are '... + x ...' and '... - x ...'
*   the initial time, there may be no sign present
*)

(* start 'Plus' matching off, initializing 'sum' *)
BFormat[Horizontal[Plus, {sum__}, 0, 0]] := Plus[BFormatPlus[0, sum]]
BFormatPlus[0, x_, rest___] := BFormatPlus[BFormat[x], 0, rest]
BFormatPlus[0, "-", x_, rest___] := BFormatPlus[Times[-1, BFormat[x]], 0, rest]

BFormatPlus[done__, 0, "+", x_, rest___] := BFormatPlus[done, BFormat[x], 0, rest]
BFormatPlus[done__, 0, "-", x_, rest___] :=
    BFormatPlus[done, BFormat[Times[-1, BFormat[x]]], 0, rest]
BFormatPlus[done__, 0] := done

```

Figure A.8: Backward Translation Summation Rules

Backward translations are invoked by applying `BFormat` to an expression. Figure A.7 shows the backward translations for tokens, rational numbers, and powers. These are the counterparts of the forward translations shown in Figure A.3. These are simple single-pattern translations and are easily derived from the forward translations.

The backward translations for sums are shown in Figure A.8. Like their forward translation counterparts, they use zero as a signpost. The delimiter `-` signals the need to construct a term of the form `Times[-1, x]`.

Like sums, the backward translations for products are somewhat simplified versions of their forward translation counterparts. They are shown in Figure A.9. Three rules are necessary, one to recognize each final form generated by the forward translations for products. Because only computational equivalence is necessary, a single rule to handle minus signs works for either notation generated by the translations in Figure A.6.



```

(*)
* Products are actually quotients except when the numerator or denominator = 0
* Must handle these three patterns.
* A leading "-" must be turned into a '-1'
*)
BFormat[Horizontal[Times, {num__}, 0, 0]] := Times[BFormatTimes[0, num]]

BFormat[Vertical[Times, 1, HLine[1, 4], Horizontal[Times, {denom__}, 0, 0],
            center, bottom, 2, 0, -3]] :=
    Power[Times[BFormatTimes[0, denom]], -1]
BFormat[Vertical[Times, num_, HLine[1, 4], denom_,
            center, bottom, 2, 0, -3]] :=
    Times[BFormatTimes[0, num], Power[Times[BFormatTimes[0, denom]], -1]]

BFormatTimes[0, "-", rest__] := BFormatTimes[-1, 0, rest]
BFormatTimes[done___, 0, x_, rest___] := BFormatTimes[done, BFormat[x], 0, rest]
BFormatTimes[done___, 0] := done

```

Figure A.9: Backward Translation Product Rules

## Appendix B

# DAG Algorithms

Section 7.1 shows that the use of DAGs instead of trees can significantly reduce both time and space requirements of the interface. Unfortunately, using DAGs increases the complexity of the algorithms. Because of this, the algorithms presented in the main body of the thesis assume a tree representation. In this appendix, we present the data structures necessary in order to use DAGs. One of the formatting algorithms presented in the main body of the text is shown here rewritten to use DAGs. The data structures are presented first followed by the sample algorithm. The algorithm and data structures are presented in pseudo-C++ code.

### B.1 Data Structures

The primary data structure used by the formatting, drawing, parsing, and selection procedures is a tree-like data structure that contains both the box structure used for formatting, drawing, and selection, and the parse tree structure used by the parser (or overlay algorithms). As discussed in Section 7.1, this data structure must be split into a position dependent part and a position independent part in order to take advantage of sharing. The position dependent part is called a **PBox** and is shown in Figure B.1. The position independent part is called a **Box** and is shown in Figure B.2. In the remainder of this section, “box” is used to generically refer to the **Box** and **PBox** data structures.

A **PBox** structure typically consists of **x**, **y**, **box** fields. Using a 32 bit word for each field, the size of a **PBox** is 96 bytes. Trying to shrink this number presents some problems. Large expressions tend to be wide but not tall. For example, using a relatively small font

```

class PBox {
// The position dependent part of a box.

friend class Box;
public:
    // public methods: ....
protected:
    long x;           // x-pos of box relative to parent
    int y;           // y-pos of box relative to parent
    Box* box;        // position independent part of box
};

```

Figure B.1: PBox Data Structure

whose average character is 9 pixels wide ( $\text{height} = 10$ ,  $\text{depth} = 2$ ), the large expression  $(w + x + y + z)^{20}$  is 284,783 pixels wide in MathScribe—over 200 (large) screens wide. The expression has a total height of only 18 pixels. For this reason, it is probably best to use at least 32 bits for  $x$  and 16 bits for  $y$ . A width-relative representation reduces the number of bits needed for the  $x$  field in most cases, but for some matrix representations, large matrices are still a problem. Another consideration in choosing the size of the  $x$  and  $y$  fields is whether higher-precision devices such as printers are supported. In this case, floating point numbers might be appropriate for the  $x$  and  $y$  fields.

If a system allows the user to manually override the automatic placement algorithms (for fine tuning), then the PBox structure would also include `deltaX` and `deltaY` fields. Because these are used for fine tuning only, these could be represented by shorts or bytes. If a byte is used, then because `deltaX` and `deltaY` can be either positive or negative, the maximum number of bits the user can fine tune an expression is limited to 128. Even with today's more expensive displays, this translates into a maximum displacement of about an inch.

The Box structure shown is decomposed into several subclasses. The use of subclasses not only allows object-oriented design, but also allows each subclass to be smaller, thus reducing the overall memory needs of the interface. The subdivision breaks Boxes into internal boxes (`InternalBox`), token boxes (`TokenBox`), and character boxes (`CharBox`). The data structures are shown in Figure B.2. An important point to note about sharing boxes is that once a box is created, it cannot be changed unless all boxes that contain it are to be changed also (which was never the case in MathScribe). For example, if a user types

```

class Box {
// The position independent part of a box.
// This class is an abstract class that provides common functionality
//   for its subclasses -- it has no public constructors.

public:
    // public methods ....
protected:
    unsigned long width;           // width of box
    unsigned short height;        // amount above baseline
    unsigned short depth;         // amount below baseline
    short fontSize;               // size of font--cached for incremental algs
    LineBreakInfo lineBreak;      // info about line breaks in children
};

class InternalBox: Box {
// An internal node in the DAG

public:
    // public methods ....
protected:
    unsigned int numChildren;     // number of children
    PBox* children;               // children of Box (array of PBoxes)
};

class TokenBox: InternalBox {
// An internal node in the DAG corresponding to a token
// Tokens have several "special" properties that other internal nodes lack.

public:
    // public methods ....
protected:
    char* str;                    // string used to speed drawing if all chars
                                // are from the same font; null otherwise
    Font* font;                   // font used for drawing
};

class CharBox: Box {
// Characters -- what is actually drawn

public:
    // public methods ....
protected:
    unsigned char ch;             // the character to display
    Font* font;                  // font to use for this character
    FontClass fontClass;         // class of font (used by scanner)
};

```

Figure B.2: Box Data Structure

```

class Path {
public:
    // public methods ....
protected:
    PBoxStack path;
};

```

Figure B.3: Path Data Structure

a 0 after the 1 in  $x + 1$ , it is incorrect to simply append a 0 CharBox to the children of the TokenBox of 1, because this TokenBox might also be used in the expression  $y + 1$  elsewhere. Instead, a new TokenBox must be created to represent 10. Similarly, a new InternalBox must be created to represent  $x + 10$  (the  $x$  TokenBox remains unchanged). Because addition and deletion of children are not allowed, the children “list” can be efficiently represented by an array as is shown for the InternalBox structure<sup>1</sup>.

One of the major complicating factors of using DAGs instead of trees is that DAGs lack parent pointers. This means that a having a pointer to a box provides no information about its relationship to the expression containing the box. In particular, we cannot compute its position on the screen, find its context for parsing, etc. To alleviate this problem, an auxiliary data structure called a Path is used. A Path is essentially a stack of boxes whose top is the current box and whose bottom is the root of the expression. Boxes and paths are passed as pairs. Whenever an algorithm moves to a child of a box, that child is pushed on the path; if the algorithm needs to move to a parent, the parent is found by removing the current box from the path. The Path data structure is shown in Figure B.3. Paths are not only useful for DAG-based implementations, but can also be used for tree-based implementations to save space: paths represent active parent pointers; most parent pointers in a tree are unused at any one point in time and consume space.

## B.2 Algorithms

In this section, we present the algorithm in Figure 4.6 rewritten to use DAGs. This algorithm is used to format superscripts. In Figure B.4, we assume that Boxes are

---

<sup>1</sup>Optimizations for special common cases of one, two, or three children can be done such that only one “chunk” of memory is allocated from the free store in these cases as opposed to two chunks—one for the Box and one for the children field. This is not shown in order to keep the presentation as clear and simple as possible.

unique and cannot be modified. We further assume that storage management is taken care of elsewhere. Section 7.3 describes some ways of collecting unused Boxes. Figure B.4 illustrates these points.

Algorithms using DAGs differ from tree algorithms in three important ways: modifying the box is prohibited, a 'path' must be constructed from the modified box to the root, and unique boxes are constructed to minimize storage costs.

```

Box* FormatSuper(Operator* op, CASForm* form, Box** children, Path& path)
/* Formats children as a superscript/exponent.
 * A new box is returned and path is modified as necessary.
 * The children are assumed to be the base, exponent operator, and exponent
 * (in that order) with the exponent operator having 0 width and height
 */
{
    // Add parens to 'base' if necessary (checks precedence info)
    // If base is on 'path', then we must add parens to the path
    Box* base = children[0];
    if (path.last() == base) {
        // add parens and format parens accordingly if parens are required
        Box* temp = addParens(op, base)
        if (temp != base) {
            path.addLast(temp);
            base = temp;
        }
    } else {
        base = addParens(op, base);
    }

    /* calculate some size info */
    Box* exp = children[2];
    int yExp = min(base->height, exp->height+exp->depth)/3
               - exp->depth - base->height;    // negative y-values are "up"

    /* must build a list of new children because 'base' may have changed */
    Box* newChildren[3];
    newChildren[0] = base;
    newChildren[1] = children[1];
    newChildren[2] = children[2];
    InternalBox* box = UniqueInternalBox(op, children);
    if (!box) {
        /* constructor adds the new box to the hash table */
        box = new InternalBox(op, form,
                             base->width+exp->width, base->depth, exp->height - yExp,
                             children);
        /* set children's x, y values -- delim's values (...[1]) are irrelevant */
        box->children[0].x = box->children[0].y = 0;
        box->children[1].x = box->children[2].x = base->width;
        box->children[1].y = box->children[2].y = yExp;
    }

    if (path.last() == base || path.last() == children[1] || path.last() == exp) {
        path.addLast(box);
    }
    return box;
}

```

Figure B.4: A Formatting Procedure for DAGs

# Appendix C

## Sharing Data

### C.1 Sharing Data

This section contains tables showing the number of Boxes, PBoxes, and references to PBoxes (“PBox Refs”) used by three sample runs of MathScribe. These data structures are described in Section B.1 The contents of the tables and an analysis of what they show is given in Section 7.1. Briefly: there are nine different possible combinations of sharing Boxes and PBoxes: global sharing, local sharing within a single expression, and no sharing at all (i.e., a tree). There are also different possible representation for PBoxes. The tables present the number of Boxes, PBoxes, and references to PBoxes used in three different MathScribe sessions. Tables C.1–C.6 compare different PBox representations for the same session. Tables C.5–C.10 use a parent-relative representation for PBoxes and show results for three different sessions: a reduce demo script, an interactive session, and a large matrix inverse example.

For each session, two tables are shown: the first table includes invisible parentheses in the counts and the second table excludes them. Each count is broken down into internal nodes, token nodes, and character nodes to allow a more complete analysis of the counts.



| Box Counts      | internal | tokens | chars | int+tok | int+tok+chr |
|-----------------|----------|--------|-------|---------|-------------|
| Global Boxes    |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| Boxes           | 1251     | 197    | 52    | 1448    | 1500        |
| PBoxes          | 1405     | 696    | 100   | 2101    | 2201        |
| PBox Refs       | 2186     | 3177   | 273   | 5363    | 5636        |
| Local PBoxes    |          |        |       |         |             |
| Boxes           | 1251     | 197    | 52    | 1448    | 1500        |
| PBoxes          | 1587     | 1436   | 182   | 3023    | 3205        |
| PBox Refs       | 2186     | 3177   | 273   | 5363    | 5636        |
| Unshared PBoxes |          |        |       |         |             |
| Boxes           | 1251     | 197    | 52    | 1448    | 1500        |
| PBoxes          | 2186     | 3177   | 273   | 5363    | 5636        |
| Local Boxes     |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| Boxes           | 1340     | 383    | 75    | 1723    | 1798        |
| PBoxes          | 1405     | 696    | 100   | 2101    | 2201        |
| PBox Refs       | 2227     | 3418   | 431   | 5645    | 6076        |
| Local PBoxes    |          |        |       |         |             |
| Boxes           | 1670     | 1341   | 1129  | 3011    | 4140        |
| PBoxes          | 1765     | 1992   | 1201  | 3757    | 4958        |
| PBox Refs       | 2456     | 4272   | 1392  | 6728    | 8120        |
| Unshared PBoxes |          |        |       |         |             |
| Boxes           | 1670     | 1341   | 1129  | 3011    | 4140        |
| PBoxes          | 2456     | 4272   | 1392  | 6728    | 8120        |
| Unshared Boxes  |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| PBoxes          | 1405     | 696    | 100   | 2101    | 2201        |
| PBox Refs       | 2279     | 3569   | 592   | 5848    | 6440        |
| Local PBoxes    |          |        |       |         |             |
| PBoxes          | 1765     | 1992   | 1201  | 3757    | 4958        |
| PBox Refs       | 2522     | 4502   | 1770  | 7024    | 8794        |
| Unshared PBoxes |          |        |       |         |             |
| PBoxes          | 4153     | 10826  | 7655  | 14979   | 22634       |

Table C.1: Sharing Data—X-relative Batch Example (*with invisible parentheses*)

| Box Counts      | internal | tokens | chars | int+tok | int+tok+chr |
|-----------------|----------|--------|-------|---------|-------------|
| Global Boxes    |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| Boxes           | 945      | 195    | 52    | 1140    | 1192        |
| PBoxes          | 1033     | 538    | 100   | 1571    | 1671        |
| PBox Refs       | 1733     | 2712   | 273   | 4445    | 4718        |
| Local PBoxes    |          |        |       |         |             |
| Boxes           | 945      | 195    | 52    | 1140    | 1192        |
| PBoxes          | 1160     | 1181   | 182   | 2341    | 2523        |
| PBox Refs       | 1733     | 2712   | 273   | 4445    | 4718        |
| Unshared PBoxes |          |        |       |         |             |
| Boxes           | 945      | 195    | 52    | 1140    | 1192        |
| PBoxes          | 1733     | 2712   | 273   | 4445    | 4718        |
| Local Boxes     |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| Boxes           | 991      | 359    | 75    | 1350    | 1425        |
| PBoxes          | 1033     | 538    | 100   | 1571    | 1671        |
| PBox Refs       | 1754     | 2844   | 431   | 4598    | 5029        |
| Local PBoxes    |          |        |       |         |             |
| Boxes           | 1174     | 1149   | 1129  | 2323    | 3452        |
| PBoxes          | 1233     | 1551   | 1201  | 2784    | 3985        |
| PBox Refs       | 1870     | 3370   | 1392  | 5240    | 6632        |
| Unshared PBoxes |          |        |       |         |             |
| Boxes           | 1174     | 1149   | 1129  | 2323    | 3452        |
| PBoxes          | 1870     | 3370   | 1392  | 5240    | 6632        |
| Unshared Boxes  |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| PBoxes          | 1033     | 538    | 100   | 1571    | 1671        |
| PBox Refs       | 1777     | 2955   | 592   | 4732    | 5324        |
| Local PBoxes    |          |        |       |         |             |
| PBoxes          | 1233     | 1551   | 1201  | 2784    | 3985        |
| PBox Refs       | 1893     | 3535   | 1770  | 5428    | 7198        |
| Unshared PBoxes |          |        |       |         |             |
| PBoxes          | 2576     | 7672   | 7655  | 10248   | 17903       |

Table C.2: Sharing Data—X-relative Batch Example (*without* invisible parentheses)

| Box Counts      | internal | tokens | chars | int+tok | int+tok+chr |
|-----------------|----------|--------|-------|---------|-------------|
| Global Boxes    |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| Boxes           | 1251     | 197    | 52    | 1448    | 1500        |
| PBoxes          | 1347     | 236    | 52    | 1583    | 1635        |
| PBox Refs       | 2186     | 3177   | 273   | 5363    | 5636        |
| Local PBoxes    |          |        |       |         |             |
| Boxes           | 1251     | 197    | 52    | 1448    | 1500        |
| PBoxes          | 1563     | 991    | 151   | 2554    | 2705        |
| PBox Refs       | 2186     | 3177   | 273   | 5363    | 5636        |
| Unshared PBoxes |          |        |       |         |             |
| Boxes           | 1251     | 197    | 52    | 1448    | 1500        |
| PBoxes          | 2186     | 3177   | 273   | 5363    | 5636        |
| Local Boxes     |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| Boxes           | 1304     | 217    | 52    | 1521    | 1573        |
| PBoxes          | 1347     | 236    | 52    | 1583    | 1635        |
| PBox Refs       | 2210     | 3319   | 290   | 5529    | 5819        |
| Local PBoxes    |          |        |       |         |             |
| Boxes           | 1670     | 1341   | 1129  | 3011    | 4140        |
| PBoxes          | 1741     | 1468   | 1129  | 3209    | 4333        |
| PBox Refs       | 2456     | 4272   | 1392  | 6728    | 8120        |
| Unshared PBoxes |          |        |       |         |             |
| Boxes           | 1670     | 1341   | 1129  | 3011    | 4140        |
| PBoxes          | 2456     | 4272   | 1392  | 6728    | 8120        |
| Unshared Boxes  |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| PBoxes          | 1347     | 236    | 52    | 1583    | 1635        |
| PBox Refs       | 2237     | 3423   | 295   | 5660    | 5955        |
| Local PBoxes    |          |        |       |         |             |
| PBoxes          | 1741     | 1468   | 1124  | 3209    | 4333        |
| PBox Refs       | 2495     | 4451   | 1452  | 6946    | 8398        |
| Unshared PBoxes |          |        |       |         |             |
| PBoxes          | 4153     | 10826  | 7655  | 14979   | 22634       |

Table C.3: Sharing Data—Width-relative Batch Example (*with* invisible parentheses)

| Box Counts      | internal | tokens | chars | int+tok | int+tok+chr |
|-----------------|----------|--------|-------|---------|-------------|
| Global Boxes    |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| Boxes           | 945      | 195    | 52    | 1140    | 1192        |
| PBoxes          | 991      | 233    | 52    | 1224    | 1276        |
| PBox Refs       | 1733     | 2712   | 273   | 4445    | 4718        |
| Local PBoxes    |          |        |       |         |             |
| Boxes           | 945      | 195    | 52    | 1140    | 1192        |
| PBoxes          | 1136     | 896    | 151   | 2032    | 2183        |
| PBox Refs       | 1733     | 2712   | 273   | 4445    | 4718        |
| Unshared PBoxes |          |        |       |         |             |
| Boxes           | 945      | 195    | 52    | 1140    | 1192        |
| PBoxes          | 1733     | 2712   | 273   | 4445    | 4718        |
| Local Boxes     |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| Boxes           | 971      | 215    | 52    | 1186    | 1238        |
| PBoxes          | 991      | 233    | 52    | 1224    | 1276        |
| PBox Refs       | 1740     | 2790   | 290   | 4530    | 4820        |
| Local PBoxes    |          |        |       |         |             |
| Boxes           | 1174     | 1149   | 1129  | 2323    | 3452        |
| PBoxes          | 1209     | 1230   | 1129  | 2439    | 3568        |
| PBox Refs       | 1870     | 3370   | 1392  | 5240    | 6632        |
| Unshared PBoxes |          |        |       |         |             |
| Boxes           | 1174     | 1149   | 1129  | 2323    | 3452        |
| PBoxes          | 1870     | 3370   | 1392  | 5240    | 6632        |
| Unshared Boxes  |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| PBoxes          | 991      | 233    | 52    | 1224    | 1276        |
| PBox Refs       | 1749     | 2843   | 295   | 4592    | 4887        |
| Local PBoxes    |          |        |       |         |             |
| PBoxes          | 1209     | 1230   | 1129  | 2439    | 3568        |
| PBox Refs       | 1879     | 3471   | 1452  | 5350    | 6802        |
| Unshared PBoxes |          |        |       |         |             |
| PBoxes          | 2576     | 7672   | 7655  | 10248   | 17903       |

Table C.4: Sharing Data—Width-relative Batch Example (*without* invisible parentheses)

| Box Counts      | internal | tokens | chars | int+tok | int+tok+chr |
|-----------------|----------|--------|-------|---------|-------------|
| Global Boxes    |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| Boxes           | 1251     | 197    | 52    | 1448    | 1500        |
| PBoxes          | 1759     | 1166   | 155   | 2925    | 3080        |
| PBox Refs       | 2186     | 3177   | 273   | 5363    | 5636        |
| Local PBoxes    |          |        |       |         |             |
| Boxes           | 1251     | 197    | 52    | 1448    | 1500        |
| PBoxes          | 1837     | 1800   | 208   | 3637    | 3845        |
| PBox Refs       | 2186     | 3177   | 273   | 5363    | 5636        |
| Unshared PBoxes |          |        |       |         |             |
| Boxes           | 1251     | 197    | 52    | 1448    | 1500        |
| PBoxes          | 2186     | 3177   | 273   | 5363    | 5636        |
| Local Boxes     |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| Boxes           | 1443     | 444    | 103   | 1887    | 1990        |
| PBoxes          | 1759     | 1166   | 155   | 2925    | 3080        |
| PBox Refs       | 2318     | 3683   | 494   | 6001    | 6495        |
| Local PBoxes    |          |        |       |         |             |
| Boxes           | 1670     | 1341   | 1129  | 3011    | 4140        |
| PBoxes          | 2034     | 2381   | 1248  | 4415    | 5663        |
| PBox Refs       | 2456     | 4272   | 1392  | 6728    | 8120        |
| Unshared PBoxes |          |        |       |         |             |
| Boxes           | 1670     | 1341   | 1129  | 3011    | 4140        |
| PBoxes          | 2456     | 4272   | 1392  | 6728    | 8120        |
| Unshared Boxes  |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| PBoxes          | 1759     | 1166   | 155   | 2925    | 3080        |
| PBox Refs       | 2652     | 4387   | 1063  | 7039    | 8102        |
| Local PBoxes    |          |        |       |         |             |
| PBoxes          | 2034     | 2381   | 1248  | 4415    | 5663        |
| PBox Refs       | 2828     | 5085   | 2168  | 7913    | 10081       |
| Unshared PBoxes |          |        |       |         |             |
| PBoxes          | 4153     | 10826  | 7655  | 14979   | 22634       |

Table C.5: Sharing Data—Batch Example (*with invisible parentheses*)

| Box Counts      | internal | tokens | chars | int+tok | int+tok+chr |
|-----------------|----------|--------|-------|---------|-------------|
| Global Boxes    |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| Boxes           | 945      | 195    | 52    | 1140    | 1192        |
| PBoxes          | 1342     | 994    | 155   | 2336    | 2491        |
| PBox Refs       | 1733     | 2712   | 273   | 4445    | 4718        |
| Local PBoxes    |          |        |       |         |             |
| Boxes           | 945      | 195    | 52    | 1140    | 1192        |
| PBoxes          | 1402     | 1544   | 208   | 2946    | 3154        |
| PBox Refs       | 1733     | 2712   | 273   | 4445    | 4718        |
| Unshared PBoxes |          |        |       |         |             |
| Boxes           | 945      | 195    | 52    | 1140    | 1192        |
| PBoxes          | 1733     | 2712   | 273   | 4445    | 4718        |
| Local Boxes     |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| Boxes           | 1064     | 417    | 103   | 1481    | 1584        |
| PBoxes          | 1342     | 994    | 155   | 2336    | 2491        |
| PBox Refs       | 1815     | 3049   | 494   | 4864    | 5358        |
| Local PBoxes    |          |        |       |         |             |
| Boxes           | 1174     | 1149   | 1129  | 2323    | 3452        |
| PBoxes          | 1493     | 1939   | 1248  | 3432    | 4680        |
| PBox Refs       | 1870     | 3370   | 1392  | 5240    | 6632        |
| Unshared PBoxes |          |        |       |         |             |
| Boxes           | 1174     | 1149   | 1129  | 2323    | 3452        |
| PBoxes          | 1870     | 3370   | 1392  | 5240    | 6632        |
| Unshared Boxes  |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| PBoxes          | 1342     | 994    | 155   | 2336    | 2491        |
| PBox Refs       | 1955     | 3833   | 1063  | 5788    | 6851        |
| Local PBoxes    |          |        |       |         |             |
| PBoxes          | 1493     | 1939   | 1248  | 3432    | 4680        |
| PBox Refs       | 2012     | 4278   | 2168  | 6290    | 8458        |
| Unshared PBoxes |          |        |       |         |             |
| PBoxes          | 2576     | 7672   | 7655  | 10248   | 17903       |

Table C.6: Sharing Data—Batch Example (*without* invisible parentheses)

| Box Counts      | internal | tokens | chars | int+tok | int+tok+chr |
|-----------------|----------|--------|-------|---------|-------------|
| Global Boxes    |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| Boxes           | 297      | 129    | 41    | 426     | 467         |
| PBoxes          | 405      | 392    | 351   | 797     | 1148        |
| PBox Refs       | 503      | 840    | 683   | 1343    | 2026        |
| Local PBoxes    |          |        |       |         |             |
| Boxes           | 297      | 129    | 41    | 426     | 467         |
| PBoxes          | 446      | 534    | 518   | 980     | 1498        |
| PBox Refs       | 503      | 840    | 683   | 1343    | 2026        |
| Unshared PBoxes |          |        |       |         |             |
| Boxes           | 297      | 129    | 41    | 426     | 467         |
| PBoxes          | 503      | 840    | 683   | 1343    | 2026        |
| Local Boxes     |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| Boxes           | 329      | 171    | 90    | 500     | 590         |
| PBoxes          | 405      | 392    | 351   | 797     | 1148        |
| PBox Refs       | 533      | 923    | 723   | 1456    | 2179        |
| Local PBoxes    |          |        |       |         |             |
| Boxes           | 617      | 523    | 353   | 1140    | 1493        |
| PBoxes          | 786      | 1026   | 1348  | 1812    | 3160        |
| PBox Refs       | 900      | 1770   | 1919  | 2670    | 4589        |
| Unshared PBoxes |          |        |       |         |             |
| Boxes           | 617      | 523    | 353   | 1140    | 1493        |
| PBoxes          | 900      | 1770   | 1919  | 2670    | 4589        |
| Unshared Boxes  |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| PBoxes          | 405      | 392    | 351   | 797     | 1148        |
| PBox Refs       | 605      | 1079   | 923   | 1684    | 2607        |
| Local PBoxes    |          |        |       |         |             |
| PBoxes          | 786      | 1026   | 1348  | 1812    | 3160        |
| PBox Refs       | 1054     | 2123   | 2387  | 3177    | 5564        |
| Unshared PBoxes |          |        |       |         |             |
| PBoxes          | 1390     | 3599   | 6910  | 4989    | 11899       |

Table C.7: Sharing Data—Interactive Example (*with invisible parentheses*)

| Box Counts      | internal | tokens | chars | int+tok | int+tok+chr |
|-----------------|----------|--------|-------|---------|-------------|
| Global Boxes    |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| Boxes           | 232      | 127    | 41    | 359     | 400         |
| PBoxes          | 328      | 350    | 351   | 678     | 1029        |
| PBox Refs       | 407      | 741    | 683   | 1148    | 1831        |
| Local PBoxes    |          |        |       |         |             |
| Boxes           | 232      | 127    | 41    | 359     | 400         |
| PBoxes          | 350      | 481    | 518   | 831     | 1349        |
| PBox Refs       | 407      | 741    | 683   | 1148    | 1831        |
| Unshared PBoxes |          |        |       |         |             |
| Boxes           | 232      | 127    | 41    | 359     | 400         |
| PBoxes          | 407      | 741    | 683   | 1148    | 1831        |
| Local Boxes     |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| Boxes           | 256      | 160    | 90    | 416     | 506         |
| PBoxes          | 328      | 350    | 351   | 678     | 1029        |
| PBox Refs       | 425      | 812    | 723   | 1237    | 1960        |
| Local PBoxes    |          |        |       |         |             |
| Boxes           | 465      | 443    | 353   | 908     | 1261        |
| PBoxes          | 619      | 891    | 1348  | 1510    | 2858        |
| PBox Refs       | 733      | 1481   | 1919  | 2214    | 4133        |
| Unshared PBoxes |          |        |       |         |             |
| Boxes           | 465      | 443    | 353   | 908     | 1261        |
| PBoxes          | 733      | 1481   | 1919  | 2214    | 4133        |
| Unshared Boxes  |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| PBoxes          | 328      | 350    | 351   | 678     | 1029        |
| PBox Refs       | 450      | 1003   | 923   | 1453    | 2376        |
| Local PBoxes    |          |        |       |         |             |
| PBoxes          | 619      | 891    | 1348  | 1510    | 2858        |
| PBox Refs       | 783      | 1893   | 2387  | 2676    | 5063        |
| Unshared PBoxes |          |        |       |         |             |
| PBoxes          | 955      | 2729   | 6910  | 3684    | 10594       |

Table C.8: Sharing Data—Interactive Example (*without* invisible parentheses)



| Box Counts      | internal | tokens | chars | int+tok | int+tok+chr |
|-----------------|----------|--------|-------|---------|-------------|
| Global Boxes    |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| Boxes           | 852      | 27     | 23    | 879     | 902         |
| PBoxes          | 1385     | 913    | 24    | 2298    | 2322        |
| PBox Refs       | 3012     | 4098   | 25    | 7110    | 7135        |
| Local PBoxes    |          |        |       |         |             |
| Boxes           | 852      | 27     | 23    | 879     | 902         |
| PBoxes          | 1395     | 919    | 24    | 2314    | 2338        |
| PBox Refs       | 3012     | 4098   | 25    | 7110    | 7135        |
| Unshared PBoxes |          |        |       |         |             |
| Boxes           | 852      | 27     | 23    | 879     | 902         |
| PBoxes          | 3012     | 4098   | 25    | 7110    | 7135        |
| Local Boxes     |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| Boxes           | 865      | 35     | 24    | 900     | 924         |
| PBoxes          | 1385     | 913    | 24    | 2298    | 2322        |
| PBox Refs       | 3024     | 4124   | 30    | 7148    | 7178        |
| Local PBoxes    |          |        |       |         |             |
| Boxes           | 871      | 48     | 37    | 919     | 956         |
| PBoxes          | 1401     | 931    | 37    | 2332    | 2369        |
| PBox Refs       | 3027     | 4139   | 38    | 7166    | 7204        |
| Unshared PBoxes |          |        |       |         |             |
| Boxes           | 871      | 48     | 37    | 919     | 956         |
| PBoxes          | 3027     | 4139   | 38    | 7166    | 7204        |
| Unshared Boxes  |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| PBoxes          | 1385     | 913    | 24    | 2298    | 2322        |
| PBox Refs       | 4086     | 6012   | 875   | 10098   | 10973       |
| Local PBoxes    |          |        |       |         |             |
| PBoxes          | 1401     | 931    | 37    | 2332    | 2369        |
| PBox Refs       | 4096     | 6050   | 887   | 10146   | 11033       |
| Unshared PBoxes |          |        |       |         |             |
| PBoxes          | 13408    | 38867  | 22062 | 52275   | 74337       |

Table C.9: Sharing Data—Matrix Inverse Example (*with invisible parentheses*)

| Box Counts      | internal | tokens | chars | int+tok | int+tok+chr |
|-----------------|----------|--------|-------|---------|-------------|
| Global Boxes    |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| Boxes           | 763      | 25     | 23    | 788     | 811         |
| PBoxes          | 1292     | 877    | 24    | 2169    | 2193        |
| PBox Refs       | 2905     | 3938   | 25    | 6843    | 6868        |
| Local PBoxes    |          |        |       |         |             |
| Boxes           | 763      | 25     | 23    | 788     | 811         |
| PBoxes          | 1302     | 880    | 24    | 2182    | 2206        |
| PBox Refs       | 2905     | 3938   | 25    | 6843    | 6868        |
| Unshared PBoxes |          |        |       |         |             |
| Boxes           | 763      | 25     | 23    | 788     | 811         |
| PBoxes          | 2905     | 3938   | 25    | 6843    | 6868        |
| Local Boxes     |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| Boxes           | 776      | 31     | 24    | 807     | 831         |
| PBoxes          | 1292     | 877    | 24    | 2169    | 2193        |
| PBox Refs       | 2905     | 3976   | 30    | 6881    | 6911        |
| Local PBoxes    |          |        |       |         |             |
| Boxes           | 779      | 40     | 37    | 819     | 856         |
| PBoxes          | 1302     | 891    | 37    | 2193    | 2230        |
| PBox Refs       | 2905     | 3985   | 38    | 6890    | 6928        |
| Unshared PBoxes |          |        |       |         |             |
| Boxes           | 779      | 40     | 37    | 819     | 856         |
| PBoxes          | 2905     | 3985   | 38    | 6890    | 6928        |
| Unshared Boxes  |          |        |       |         |             |
| Global PBoxes   |          |        |       |         |             |
| PBoxes          | 1292     | 877    | 24    | 2169    | 2193        |
| PBox Refs       | 3865     | 5954   | 875   | 9819    | 10694       |
| Local PBoxes    |          |        |       |         |             |
| PBoxes          | 1302     | 891    | 37    | 2193    | 2230        |
| PBox Refs       | 3865     | 5984   | 887   | 9849    | 10736       |
| Unshared PBoxes |          |        |       |         |             |
| PBoxes          | 7778     | 27607  | 22062 | 35385   | 57447       |

Table C.10: Sharing Data—Matrix Inverse Example (*without* invisible parentheses)

## C.2 Memory Usage Under Differing Sharing Scenarios

The following tables display the amount of memory used by three sample runs of MathScribe. The sample runs, their contents, and an analysis of the results are presented in Section 7.1. The numbers in the following tables are computed from the previous tables (Tables C.1–C.10). The computations are briefly summarized below.

Each table shows the amount of memory used (in kilo-bytes) under differing sharing scenarios. The counts assume:

- The natural word size of the machine is 32 bits.
- PBoxes are three words long if Boxes are shared and are two words long if Boxes are unshared.
- Boxes use seven words and their size is uniform across internal, token, and characters boxes. Percentage differences from those shown in the table are less than 4% between Boxes whose sizes are six words and Boxes whose sizes are nine words.

| X-relative Reduce Test ( <i>with invisible parentheses</i> ) |          |        |       |         |             |
|--|----------|--------|-------|---------|-------------|
| Memory Usage (kBytes)  | internal | tokens | chars | int+tok | int+tok+chr |
| Global Boxes   |          |        |       |         |             |
| Global PBoxes  | 59       | 26     | 4     | 85      | 89          |
| Local PBoxes   | 61       | 35     | 5     | 96      | 101         |
| Unshared PBoxes  | 60       | 43     | 5     | 102     | 107         |
| Local Boxes  |          |        |       |         |             |
| Global PBoxes  | 62       | 32     | 5     | 94      | 99          |
| Local PBoxes   | 76       | 77     | 50    | 153     | 203         |
| Unshared PBoxes  | 74       | 87     | 47    | 161     | 208         |
| Unshared Boxes   |          |        |       |         |             |
| Global PBoxes  | 58       | 38     | 6     | 97      | 103         |
| Local PBoxes   | 72       | 88     | 49    | 160     | 209         |
| Unshared PBoxes  | 146      | 381    | 269   | 527     | 796         |

| X-relative Reduce Test ( <i>without invisible parentheses</i> ) |          |        |       |         |             |
|---|----------|--------|-------|---------|-------------|
| Memory Usage (kBytes)   | internal | tokens | chars | int+tok | int+tok+chr |
| Global Boxes  |          |        |       |         |             |
| Global PBoxes   | 45       | 22     | 4     | 67      | 71          |
| Local PBoxes  | 46       | 30     | 5     | 76      | 81          |
| Unshared PBoxes   | 46       | 37     | 5     | 83      | 88          |
| Local Boxes   |          |        |       |         |             |
| Global PBoxes   | 46       | 27     | 5     | 73      | 78          |
| Local PBoxes  | 54       | 63     | 50    | 117     | 167         |
| Unshared PBoxes   | 54       | 71     | 47    | 125     | 172         |
| Unshared Boxes  |          |        |       |         |             |
| Global PBoxes   | 43       | 30     | 6     | 74      | 80          |
| Local PBoxes  | 51       | 68     | 49    | 119     | 168         |
| Unshared PBoxes   | 91       | 270    | 269   | 360     | 629         |

Table C.11: Sharing Data—X-relative Batch Memory Usage

| Width-relative Reduce Test ( <i>with</i> invisible parentheses) |          |        |       |         |             |
|---|----------|--------|-------|---------|-------------|
| Memory Usage (kBytes)   | internal | tokens | chars | int+tok | int+tok+chr |
| Global Boxes  |          |        |       |         |             |
| Global PBoxes   | 59       | 21     | 3     | 79      | 82          |
| Local PBoxes  | 61       | 29     | 4     | 90      | 95          |
| Unshared PBoxes   | 60       | 43     | 5     | 102     | 107         |
| Local Boxes   |          |        |       |         |             |
| Global PBoxes   | 60       | 22     | 3     | 82      | 85          |
| Local PBoxes  | 76       | 71     | 49    | 146     | 196         |
| Unshared PBoxes   | 74       | 87     | 47    | 161     | 208         |
| Unshared Boxes  |          |        |       |         |             |
| Global PBoxes   | 56       | 22     | 3     | 78      | 81          |
| Local PBoxes  | 71       | 69     | 45    | 140     | 185         |
| Unshared PBoxes   | 146      | 381    | 269   | 527     | 796         |

| Width-relative Reduce Test ( <i>without</i> invisible parentheses) |          |        |       |         |             |
|--|----------|--------|-------|---------|-------------|
| Memory Usage (kBytes)  | internal | tokens | chars | int+tok | int+tok+chr |
| Global Boxes   |          |        |       |         |             |
| Global PBoxes  | 44       | 19     | 3     | 63      | 66          |
| Local PBoxes   | 46       | 26     | 4     | 72      | 77          |
| Unshared PBoxes  | 46       | 37     | 5     | 83      | 88          |
| Local Boxes  |          |        |       |         |             |
| Global PBoxes  | 45       | 20     | 3     | 64      | 68          |
| Local PBoxes   | 54       | 59     | 49    | 113     | 162         |
| Unshared PBoxes  | 54       | 71     | 47    | 125     | 172         |
| Unshared Boxes   |          |        |       |         |             |
| Global PBoxes  | 42       | 19     | 3     | 61      | 64          |
| Local PBoxes   | 50       | 57     | 45    | 107     | 152         |
| Unshared PBoxes  | 91       | 270    | 269   | 360     | 629         |

Table C.12: Sharing Data—Width-relative Batch Memory Usage

| Reduce Test ( <i>with invisible parentheses</i> ) |          |        |       |         |             |
|---|----------|--------|-------|---------|-------------|
| Memory Usage (kBytes)                             | internal | tokens | chars | int+tok | int+tok+chr |
| Global Boxes                                      |          |        |       |         |             |
| Global PBoxes                                     | 63       | 31     | 4     | 95      | 99          |
| Local PBoxes                                      | 64       | 39     | 5     | 103     | 108         |
| Unshared PBoxes                                   | 60       | 43     | 5     | 102     | 107         |
| Local Boxes                                       |          |        |       |         |             |
| Global PBoxes                                     | 69       | 40     | 7     | 109     | 116         |
| Local PBoxes                                      | 79       | 81     | 51    | 160     | 211         |
| Unshared PBoxes                                   | 74       | 87     | 47    | 161     | 208         |
| Unshared Boxes                                    |          |        |       |         |             |
| Global PBoxes                                     | 72       | 58     | 10    | 130     | 140         |
| Local PBoxes                                      | 83       | 104    | 52    | 186     | 238         |
| Unshared PBoxes                                   | 146      | 381    | 269   | 527     | 796         |

| Reduce Test ( <i>without invisible parentheses</i> ) |          |        |       |         |             |
|--|----------|--------|-------|---------|-------------|
| Memory Usage (kBytes)                                | internal | tokens | chars | int+tok | int+tok+chr |
| Global Boxes   |          |        |       |         |             |
| Global PBoxes  | 48       | 28     | 4     | 76      | 80          |
| Local PBoxes   | 49       | 34     | 5     | 83      | 88          |
| Unshared PBoxes                                      | 46       | 37     | 5     | 83      | 88          |
| Local Boxes  |          |        |       |         |             |
| Global PBoxes  | 52       | 35     | 7     | 87      | 93          |
| Local PBoxes   | 57       | 67     | 51    | 124     | 175         |
| Unshared PBoxes                                      | 54       | 71     | 47    | 125     | 172         |
| Unshared Boxes                                       |          |        |       |         |             |
| Global PBoxes  | 55       | 50     | 10    | 105     | 114         |
| Local PBoxes   | 60       | 85     | 52    | 145     | 198         |
| Unshared PBoxes                                      | 91       | 270    | 269   | 360     | 629         |

Table C.13: Sharing Data—Batch Memory Usage

| Interactive Test ( <i>with invisible parentheses</i> ) |          |        |       |         |             |
|--|----------|--------|-------|---------|-------------|
| Memory Usage (kBytes)                                  | internal | tokens | chars | int+tok | int+tok+chr |
| Global Boxes   |          |        |       |         |             |
| Global PBoxes  | 15       | 11     | 8     | 26      | 34          |
| Local PBoxes   | 15       | 13     | 10    | 28      | 38          |
| Unshared PBoxes  | 14       | 13     | 9     | 27      | 37          |
| Local Boxes  |          |        |       |         |             |
| Global PBoxes  | 16       | 13     | 9     | 29      | 38          |
| Local PBoxes   | 30       | 33     | 33    | 63      | 96          |
| Unshared PBoxes  | 27       | 35     | 32    | 62      | 95          |
| Unshared Boxes   |          |        |       |         |             |
| Global PBoxes  | 17       | 18     | 16    | 35      | 51          |
| Local PBoxes   | 32       | 44     | 57    | 76      | 133         |
| Unshared PBoxes  | 49       | 127    | 243   | 175     | 418         |

| Interactive Test ( <i>without invisible parentheses</i> ) |          |        |       |         |             |
|---|----------|--------|-------|---------|-------------|
| Memory Usage (kBytes)                                     | internal | tokens | chars | int+tok | int+tok+chr |
| Global Boxes  |          |        |       |         |             |
| Global PBoxes   | 12       | 10     | 8     | 22      | 30          |
| Local PBoxes  | 12       | 12     | 10    | 24      | 34          |
| Unshared PBoxes   | 11       | 12     | 9     | 23      | 32          |
| Local Boxes   |          |        |       |         |             |
| Global PBoxes   | 13       | 12     | 9     | 24      | 34          |
| Local PBoxes  | 23       | 28     | 33    | 51      | 84          |
| Unshared PBoxes   | 21       | 29     | 32    | 51      | 83          |
| Unshared Boxes  |          |        |       |         |             |
| Global PBoxes   | 13       | 16     | 16    | 30      | 45          |
| Local PBoxes  | 25       | 39     | 57    | 64      | 120         |
| Unshared PBoxes   | 34       | 96     | 243   | 130     | 372         |

Table C.14: Sharing Data—Interactive Memory Usage

| 5 by 5 Vandermonde Matrix Inverse ( <i>with invisible parentheses</i> ) |          |        |       |         |             |
|---|----------|--------|-------|---------|-------------|
| Memory Usage (kBytes)   | internal | tokens | chars | int+tok | int+tok+chr |
| Global Boxes  |          |        |       |         |             |
| Global PBoxes   | 51       | 27     | 1     | 79      | 80          |
| Local PBoxes  | 51       | 28     | 1     | 79      | 80          |
| Unshared PBoxes   | 59       | 49     | 1     | 107     | 108         |
| Local Boxes   |          |        |       |         |             |
| Global PBoxes   | 52       | 28     | 1     | 79      | 81          |
| Local PBoxes  | 52       | 28     | 2     | 80      | 82          |
| Unshared PBoxes   | 59       | 50     | 1     | 109     | 111         |
| Unshared Boxes  |          |        |       |         |             |
| Global PBoxes   | 65       | 56     | 4     | 120     | 124         |
| Local PBoxes  | 65       | 56     | 5     | 122     | 126         |
| Unshared PBoxes   | 471      | 1366   | 776   | 1838    | 2613        |

| 5 by 5 Vandermonde Matrix Inverse ( <i>without invisible parentheses</i> ) |          |        |       |         |             |
|--|----------|--------|-------|---------|-------------|
| Memory Usage (kBytes)  | internal | tokens | chars | int+tok | int+tok+chr |
| Global Boxes   |          |        |       |         |             |
| Global PBoxes  | 47       | 26     | 1     | 74      | 75          |
| Local PBoxes   | 47       | 26     | 1     | 74      | 75          |
| Unshared PBoxes  | 55       | 47     | 1     | 102     | 103         |
| Local Boxes  |          |        |       |         |             |
| Global PBoxes  | 48       | 27     | 1     | 74      | 75          |
| Local PBoxes   | 48       | 27     | 2     | 75      | 77          |
| Unshared PBoxes  | 55       | 48     | 1     | 103     | 105         |
| Unshared Boxes   |          |        |       |         |             |
| Global PBoxes  | 61       | 54     | 4     | 115     | 119         |
| Local PBoxes   | 61       | 55     | 5     | 116     | 120         |
| Unshared PBoxes  | 273      | 971    | 776   | 1244    | 2020        |

Table C.15: Sharing Data—Matrix Inverse Memory Usage



