SKEME, (Symbolic Keyboard and Mouse Editor)
A Demonstration Program for
Multimodal Input of Mathematical Equations

Kevin Lin
Richard Fateman
University of Calif. Berkeley

1. Background

The potential for a wide variety of symbols, characters and font-sizes, as well as the need to support non-linear arrangements of objects make entering mathematical formulas into a computer a challenging task [2]. In a project at UC Berkeley, we are exploring how mathematical input can be improved by providing more than one mode of input. We propose that traditional mouse and keyboard input can be synchronized with voice recognition and handwriting recognition to improve accuracy and efficiency.

To begin with, we need a simple open-source base program that can graphically display mathematical symbols and is easily adaptable to incorporate voice recognition and handwriting recognition. SKEME (Symbolic Keyboard and Mouse Editor) is such a sample program, initially set up to allow you to enter mathematical equations using a keyboard and mouse input. It also allows you to edit the equation using a graphical user interface. SKEME can be expanded to incorporate voice and handwriting recognition. Because it is written in Common Lisp, it can be debugged interactively, and (along with other applications in Common Lisp) has access to Unicode, arbitrary-precision arithmetic, memory management, and an interactive graphics environment.

2. How to use SKEME

2.1 General Concepts

Starting from a blank screen, a right-click of the mouse reveals a menu which contains eight initial choices of mathematical/geometric positioning variations, such as fraction, super/subscript, etc. If you hover the mouse "on top" of a certain object X, then X will be highlighted red, and the right-click will reveal a different menu in which the selection incorporates the object X in various templates. You must make a choice of one of these to continue processing.

If your mouse is not "on top" of any object, the new object will be added to the root of the window in the place indicated by the location of the mouse, allowing you to set out any number of formulas on the same page.

The display program follows the TeX model, where three different "heights" of text script are offered, depending upon the depth of nesting. Although it is possible to use more than three different heights of fonts, a wider variation in sizes, except for large delimiters like parentheses, inhibits readability [3].

Once one or more formulas are available on the screen, they can each be converted to ordinary TeX (whose results are printed out in the interaction window, and can be pasted into text files, etc.) We do not require that the formulas be semantically well-formed

because it seems premature to make such determinations in a screen editor. If the statements are to be fed into a computer algebra system or transmitted using OpenMath, it may be advisable to perform further checks to assure their conversion into legal forms.

2.2 Navigation and Arrow Keys

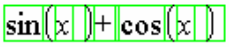
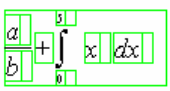
Given a selected sub-expression on the screen, you can move from one object to an adjacent one, by using the up, down, left, and right key arrows. To delete an object, press "backspace". If the mouse is hovering over a text box, deletion will be performed a single at a time. If the mouse is hovering over an empty object or the end of an object (the space after the last textbox), then the entire object will be deleted.

2.3 Bounding Boxes

You can enable or disable green display borders around the boxes from the right-click menu, selecting "Hide Bounding Boxes". Here is a comparison of mathematical input with box bounding boxes on and off.

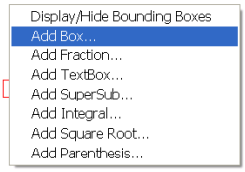
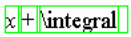
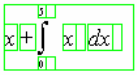
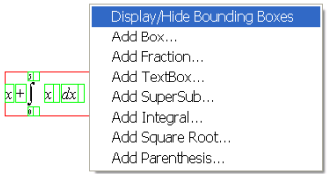
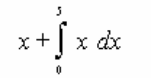
2.4 TeX Output

The output can be converted into TeX format by typing "(convert)" in the debug window. This command returns all the mathematical expressions on the screen as a series of TeX expressions. For several separate formulas, relative spatial location on the screen is not preserved in the TeX. The objects are output chronologically, according to the time when the formula was first begun [3].

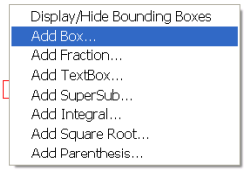
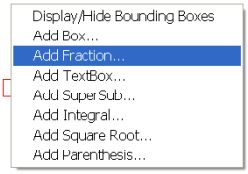
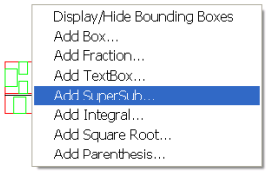
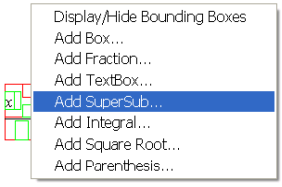
TeX Code	SKEME output	TeX output
`\+&& \cr $$\sin\left(x\right)+\cos\left(x\right)$$`	$\sin(x)+\cos(x)$ or 	$\sin(x)+\cos(x)$
`\+&& \cr $${{a}\over{b}}+\int^{5}_{0}{x}\,{dx}$$`	$\frac{a}{b}+\int_0^5 x\,dx$ or 	$\frac{a}{b}+\int_0^5 x\,dx$

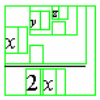
2.5 Examples

2.5.1 Basic Input Example #1

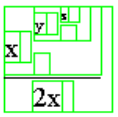
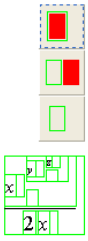
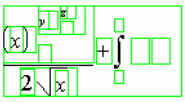
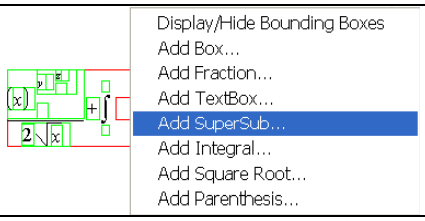
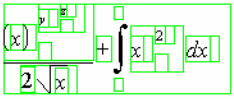
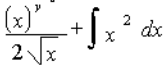
1. Right click: “Add box…”	
2. Hover the mouse over the box. Type: “x” + <space> + “+” + <space> + “\integral” + <enter> “\integral” will be converted into an integral object.	
3. Enter, “x” in the main field, “dx” in the variable field, “0” in the lower limit, and “5” in the upper limit	
4. Right Click, and select “Display/Hide bounding Boxes”	
5. Final Result	

2.5.2 Basic Input Example #2

1. Right click: “Add box…”	
2. Hover the mouse of the box. Right click: “Add Fraction…”	
3. Hover the mouse over the numerator of the box. Right Click: “Add Supersub…”	
4. Hover the mouse over the body of the Supersub. Type “x”.	

5. Right Click on the superscript box of the Supersub object. Select “Add Supersub…” Add “y” and “z” by hovering the mouse over the selected locations. Input “2x” in the denominator”	
6. Right Click, and select “Display/Hide bounding Boxes”. Final Result.	

2.5.3 Replacement Example

1. Start out with example 2.4.3	
2. Hover the mouse over the “x” in the denominator. Select “Add Square Root…” A second popup menu comes up. Select the top option.	
3. Hover the mouse over “x”. Select “Add Parenthesis…” Select the top option	
4. Hover the mouse over the end of largest box. Enter: “+” + <space> + “\integral” + “<enter>	
5. Select the mouse over the “main” box of the integral. Right click and select “Add SuperSub…”	
6. Add “x”, “2”, and “dx”, to the respective locations.	
7. The final result.	

2.6 Getting Started

A copy of the source code can be obtained from:
www.cs.berkeley.edu/~fateman/skeme.zip
The best way to observe the details is to try the program, so if you have a version of the program available, we suggest you open Allegro CL IDE, and use the debug window:

1. type `:cd <directory of SKEME>`
2. `:ld SKEME`
3. type: `(init)`. This will initialize the main window.

NOTE: The files *fpopup.lisp, ipopup.lisp, gpopup.lisp,* and *sspopup.lisp* support the graphical right-click interface in SKEME. The file *mathinput.lisp* contains the GUI and much of the implementation of SKEME. The file *io.lisp* contains the TeX converter. These files, or their compiled versions, are all loaded when *SKEME.lisp* is loaded.

3.0 A Preview of the Internal Organization

A more detailed exposition is in section 4 but we present this preview to give you an idea of what you are constructing.

3.1 Objects

There are seven types of objects in SKEME. They are as follows: box, fraction, textbox, superscript-subscript, integral, square root. Each is an instance of a lisp `defstruct bt` (short for box template).

1. **Box:** The box is a `bt` structure with type field which enables us to distinguish between specific boxes types, and is the basic form used in fraction, superscript-subscript, integral, and square root. If you select and then type into an empty box it becomes a TextBox (see below).
2. **Fraction:** Fraction… a `bt` with argument field of a list of length two that contains a numerator and a denominator. The numerator and denominator are essentially box objects, so fractions can be created recursively.
3. **TextBox:**. If you select and type into an empty box object, a textbox will be formed inside that box object. If you type [space], a new text object will be created. Thus, you can create a collection of 5 separate boxes from " a + b = c" objects for easy editing later on. The text box can be used to create other objects: If you type certain key words. the textbox can produce other kinds of objects: this is like a keyboard shortcut to mouse/menu activity:
   `\frac` = creates a fraction object
   `\supersub` = creates into a superscript/subscript object
   `\box` = creates a default box
   `\integral` = creates an integral object.
   `\sqrt` = creates a square root object
   `\paren` = creates a parenthesis object.
   `"("` = creates a parenthesis object.
   `"()"` = creates a parenthesis object.
   Also, recognized functions will be italicized and text fields containing parenthesis will be parsed (See Section 2.3).

4. **Superscript-subscript:** This is a box object that has typefield … and displays as a regular "main" box with a "super-scripted" box and a "sub-scripted" box. The "super-scripted" box and "sub-scripted" box are sized 1 point smaller than the main box.
5. **Integral:** This displays the common mathematical integral object. This object contains 4 editable boxes. The "main" box, the "variable" box, the "upper limit" box, and the "lower limit" box. The "upper limit" and "lower limit" boxes are sized 2 smaller than the main box.
6. **Square Root:** This creates a square root object. The argument field of the object contains one item, a box (which appears under the square root sign).
7. **Parenthesis:** This creates a parenthesis object. The argument field of the object contains one item, a box (which appears in between the open and clothes parenthesis).

3.2 Parsing of Parenthesis in Text and Recognition of Functions and Variables

SKEME follows the TeX format of recognizing mathematical functions. That is, mathematical functions such as sin, cos, and tans, as well as numbers, are set in Roman (regular print). Names of variables, such as *x*, and *y*, are italicized. Here is a list of common mathematical symbols adopted from TeX:

arccos	arcsin	arctan	arg	cos	cosh
cot	coth	csc	deg	det	dim
exp	gcd	hom	inf	ker	lg
lim	liminf	limsup	ln	log	max
min	Pr	sec	sin	sinh	sup
tan	tanh				

A list of these strings is given in the global variable `mathsymbollist`. A rule that multiple-letter names should be in Roman as in temp:=GCD(*y*,*z*) is not always correct, since we prefer *dx* to dx. Therefore a list of all the words (strings) that should be italicized is contained in the global variable `italicsymbols`. Both of these variables are initialized in the file *mathinput.lisp*, but any program can append items to these lists, or a programmer can edit the source data.

SKEME does make some judgment about well-formedness of expressions with regard to parentheses: It assumes that parentheses, in TeX `\left(` and `\right)` are balanced, and will insert these when it notices notices text fields such as "(x)."

4. Technical Details

This section is intended to give a high-level overview of the implementation, which can be viewed as a guide to further comments in the code. We also mention approaches to solving the harder problems, and bugs and improvements that can be made to the program.

4.1 Initialization

SKEME uses a conventional common-lisp graphics library in Allegro Common Lisp on Microsoft Windows to implement graphics.

First, the window is initialized as a class frame-window by the following:

```
(defclass my-window (frame-window)())
```
Whenever the window needs to be redisplayed,
`(invalidate defaultwindow)` is called from elsewhere in the code. This calls the function redisplay:
```
(defmethod redisplay-window ((window my-window) &optional
box)
      (call-next-method)
      (draweverything window liststufftodraw)
      (drawstatus window
    (setf defaultwindow window))
```
which redraws the objects onto the string. (draweverything is responsible for refreshing the display of data on the screen).

Function init initializes all the variables, and displays the main window using:
`(make-window :Math\ Input\ Demonstration :class 'my-window)`
All of the objects are stored in the variable `liststufftodraw`. Recursive objects are stored in the corresponding parent, as explained below.

4.2 Structures

As mentioned above, there are 7 types of objects. The objects are stored in a struct arrangement, based on the following specifications:
`(defstruct boxtemp type x y width height color size arguments counter)` describes the struct. The type determines which type of object it is, and each type of object has unique arguments. The counter can be used to determine if two structures are equal, under the logic that if two objects have the same counter identification, then they are equal.

1. **Fraction:** type: ‘fraction, arguments: `(numerator denominator),` where numerator and denominator are box structs.
2. **Supersub:** type: ‘supersub, arguments: `(main, superscript, subscript),` where main, superscript, and subscript are box structs.
3. **Textbox:** type: ‘text, arguments: `(string font),` where string is a lisp string object, and font is a lisp font object.
4. **Integral:** type: ‘integral, arguments: `(main, upperlimit, lowerlimit, variable),` where main, upperlimit, lowerlimit, and variable are box structs.
5. **Square Root:** type: ‘sqrt, arguments: `(contents),` where contents are box structs.
6. **Parenthesis:** type: ‘paren, arguments`(contents)`, where contents are box structs.

4.3 Extensions to SKEME

This section explains how objects with particular formats are defined in SKEME by illustrating how you can add yet another form. As a prototype**,** SKEME includes only a limited number of mathematical objects, each modeled by a collection of forms, and affecting various display and menu items. Here we describe how to add one more object to the program, and in the process briefly describe the purpose of each of the modified functions in SKEME.

Say you want to add a vertical form specific to an indexed intersection, denoted by $\bigcap_{y}^{x} z$,

Here are the steps necessary.

Change **show-my-menu** to add this indexed intersection object to the right click menu. add something to the effect of:

```
(make-instance 'menu-item :title "<what the user
sees>" :value '<type name>))
```

- Change **parse**: Parse reads characters, and if the character is a particular string, for example \BigCap, it will change the textbox into a corresponding object.

```
((equal mykeyboardstring "\\BigCap")
   (addboxinbox currentboxparent
   (bigcapfunction (boxtemp-x    currentboxparent) (boxtemp-y
   currentboxparent)
   (boxtemp-color currentboxparent) (boxtemp-size  currentboxparent)
   nil)))
```

- Change **draw**: In the draw function, add a case statement to handle the type of struct. For example

```
   ((eq (boxtemp-type item) 'BigCap)
(displayBigCap window (boxtemp-x item) (boxtemp-y item)
(boxtemp-width item) (boxtemp-height item)
(boxtemp-color item) (boxtemp-size item)
(boxtemp-arguments item)))
```

- Add the display function displayBigCap. This is responsible for drawing the object on the screen. The model of the existing display functions should provide a guide for most variations.

- Change **add**: Add will call subsequent functions and add the object to `liststufftodraw`, and then call `invalidate` to redraw the window. Add a condition in the cond statement to handle the new type. Ex:

```
(eq arg 'BigCap)
setf child (addBigCapfunction x1 y1 green 1 nil)))
```

- Add two functions to implement adding an object of the new type. Names could be typically: `addBigCap`, which calls `addBigCapfunc` which will initialize the struct.

- Implement graphical right-click menu. If the object you expect to add is something similar to square root, or parenthesis where there are defined options for areas to "add" a new object, then it may be unnecessary to change `addsub`. However, if the object you expect to add is geometrically more complicated, such as integral or supersub, then you may need to add a custom right-click menu. A right-click menu can be constructed using Allegro CL, creating a new popup-form, and inserting picture buttons as the buttons the user sees (Custom Icons may also need to be created). We set title-bar to nil, so we can use more screen real-estate for menu information, and remove this distraction. Also, the bolded sections of the following

code indicate the changes from the Allegro CL template for displaying buttons (dialog form).

```
(defun gform-picture-button-1-on-click (dialog widget)
(declare (ignore-if-unused dialog widget))
(setf returnval 'g_over)
(addsub 'g_over)
(close mywindow)
t)

(defun gform-picture-button-2-on-click (dialog widget)
(declare (ignore-if-unused dialog widget))
(setf returnval 'g_new)
(addsub 'g_new)
(close mywindow)
t)

(defun gform-picture-button-3-on-click (dialog widget)
(declare (ignore-if-unused dialog widget))
(setf returnval 'g_replace)
(addsub 'replace)
(close mywindow)
t)

(in-package :common-graphics-user)
(defvar x1 0)
(defvar y1 0)
(defvar x2 40)
(defvar y2 120)
(defvar mywindow nil)

;; The finder-function, which returns the window if it already
;; exists, and otherwise creates and returns it.
;; Call this function if you need only one copy of this window,
;; and that window is a non-owned top-level window.
(defun gpopup (x y)
(setf x1 x)
(setf y1 y)
(setf x2 (+ x 40))
(setf y2 (+ y 120))
(setf mywindow
(find-or-make-application-window :gform 'make-gform)))
```

Notice that `(addsub 'BigCap)` is a function call to SKEME. Thus, the function `addsub` may also need to be changed appropriately. Lastly, the x, y, coordinates are created based on user input. (`gpopup` is called by the function add).

- Change **`changeoverlapelement`**: `CheckOverlapElement` evaluates the x, y arguments and returns the smallest object that the mouse is over. Add a condition to deal with the type. Typically, you will need to call `checkoverlap` (which handles a list), or `checkoverlapelement` recursively to evaluate the children of the object.

- Change **`resizeh`**. `Resizeh` deals with horizontal resizing of the object.
  For Example:
```
 ((eq type 'BigCap)
(setf totalwidth (resize-BigCaph thing x y)))
```

- Add a function to horizontally resize the type BigCap, named `resize-BigCaph.` The function should determine the width of the object, and change the x values of all the children (and the object) to fit the argument x passed in (y can be ignored). Typically, this will be done recursively, as the children need to be resized before the parent.

- Change **`resizev`**. **`Resizev`** deals with vertical resizing of the object. Ex:
```
((eq type 'BigCap)
(setf totalwidth (resize-BigCapv thing x y))))
```

- Add a function to vertically resize the type BigCap, named `resize-BigCapv`. The function should determine the height of the object, and change the y values of all the children and the object to fit the argument y passed in (x can be ignored). Typically this is done recursively, as the children need to be resized before the parent.

4.4 Finding the width of a textbox.

While this task might seem intimidating given the variety of fonts and spacing, it is actually very simple, because finding box width is built-in:

```
 (setf afont (make-font :roman :times 20))
(font-string-width afont "hello")
```

This finds the string width "hello", using a Times new Roman font of size 20. However, it is conventional to increase the width by 1 or 2 pixels, if you want to draw a border around the text box without overlapping the text box. Also, a font size of 20 generally means that the height of the font is 20, but letters such as "p" and "q" will have tails that extend below the box. The maximum height of letters in a font or spacing between successive lines of text in a font can also be calculated by built-in functions font-size and line-height.

4.5 Adding an object within another object

This is probably the most complicated process in the program. First, we must check which object the mouse is over. Because the objects are arranged in a tree-like fashion (the children of objects are also objects), we must find the "deepest" child that the mouse is over. `Checkoverlap` also changes the overlapped items to red, as a side effect. This alerts you to which object the cursor is over. `Checkoverlap` will return the smallest object that the cursor is over. If this object is a textbox, `currentboxparent` will be set to the parent of the text box (this is useful changing text boxes into other objects).

Afterwards, the object you specify will be inserted into the child of the element returned by `currentboxparent`. The whole list must be resized horizontally and vertically. Resize, like `checkoverlap`, needs to be done recursively, and left to right. Each different type of object needs to be resized differently. For example, the subscript and superscript boxes of type supersub need to be handled in a way that preserves the visual

appearance of the subscript and superscript. Vertical resizing happens in a similar fashion.

Fractions require that numerator and denominator be centered horizontally with an appropriate width divide bar. Usually horizontally adjacent boxes need to be vertically centered. For example, text boxes, which are composed of one line, will appear centered in relation to fractions, which are composed of "two lines."

4.6 Bugs and Possible Areas for Improvement

4.6.1 Known Bugs and Issues

A list of bugs and dates of discovery and solution are maintained in the code comments..

4.6.2 Possible Areas for Improvement

1. Appearance and placement of objects. Currently, the integral sign is a text object in font symbol. The width and height of the integral is a bit rough, and needs improvement. The Square root symbol is drawn by using (draw-line…) commands. There is a "√" object in Symbol font, and this can be used to make better looking square roots. Furthermore, the placement of objects as well as the font type can be "tweaked" to better resemble TeX output.

2. Vertical Centering. Currently, the objects are centered vertically; however, this is visually incorrect, as the equations require that the equal sign be centered with the horizontal line of the fraction. This problem could be solved by adding more "information" into the fraction. More specifically, the arguments of fraction could have three arguments, the third, being the location of the horizontal line relative to the size of the fraction.

3. User interface: Adding menus, buttons, and a status bar would benefit the user of the program. There are several excellent models for improved visual appearance in JMathNotes and JOME.

4. Adding symbols such as alpha, beta, etc., as well as the recognition of arithmetic operators and common variables such as "x", "y", and "z"

5. Additional forms such as matrices.

6. More sophisticated display including multiple-lines, better TeX emulation.

5. Ideas of Implementing Voice Recognition

There are many examples from Microsoft Speech SDK that should facilitate development of a voice recognition module; in particular it is not necessary to build one from scratch. We have been using as models two examples: the Visual Basic Listbox application, and the Visual C++ RECO. This latter application allows the programmer to input a grammar. These programs also refer to voice-training capabilities to improve accuracy. In our case, we expect to pre-compile a grammar allowing input of numbers, letters, symbols, and components words that are likely to occur in defining equations, while leaving out other words of the English language. The adapted speech modules would be built into a COM object that could interface with lisp. The details of such need to be investigated, but the following will describe several general models to pursue such interaction between SKEME and the speech module [8].

- The simplest method of interaction between the two modules is by use of the windows OS clipboard. This would mean starting two independent processes. Once a speech text is recognized, this word or phrase is copied to the clipboard. The lisp module would have to poll the clipboard, and at that time, it would recognize that a new phrase has been acquired. The lisp program would then process the phrase, and set the clipboard to empty, telling the speech module that the word has been recognized. Of course this is inefficient and impractical.

- The simplest practical example would have the voice be an input device. Similar to mouse input, there is no interaction until the main program activates the microphone (because it is an appropriate time to listen for voice input… the mouse is above an existing text box, for example). At this point the voice recognition module receives a phrase. Then it forwards the phrase to the lisp program, which treats the "voice interrupt" like a mouse event. This is similar to the clipboard listed above; however, the lisp program has the ability to open and close the voice recognition program, assuming that it was implemented as an OLE object to the lisp program.

- In a more complicated approach, the lisp program and speech program would interact more closely. After the speech program passed in a phrase to the lisp program, the lisp program could then process the information, and then pass the results back to the speech program, indicating whether the phrase was "decoded", or is "unidentified". If the phrase is "unidentified", the speech program can analyze the data again and pass in an alternative. Similarly, the lisp program would know if the speech program is in the middle of processing a speech command, and could add a user interface toolbar or object to inform you that speech is being processed. Furthermore, if the speech module returns "too quiet", or "too much interference", you could be informed of this problem and speak again.

The speech module could pass to the lisp program: "a plus b quantity over c", with various markers pertaining to certainty, alternatives, etc. [11]Other than the grammar, and perhaps the spoken version of unusual symbols (integral), we do not expect any mathematics-specific requirements on the speech. For example, the fragment "a plus plus b" might pass the grammar check (a +(positive) b), and be passed to SKEME as "a++b". Whether this is ultimately meaningful or syntactically correct, the speech module, and in

fact, SKEME need not need check; it might be a problem further down the chain of processing. If the speech is used only for linear text inserted in a text box as in "Gamma of script capital A" then there is minimal correction opportunity. Should the speech aspect become an alternative for large pieces of formula, an alternative which we may pursue is whether some imposition of well-formedness beyond the speech grammar would end up being advantageous: we could, in principle decide that a++b was implausible and find another explanation (say a missing phoneme or misrecognized component [13].

The addition of speech as input is more complex than viewing it as another keyboard: there are other interaction between the lisp program and the speech module, such as passing error messages "too quiet", or "too much interference", which could be handled by further graphical interfaces in the lisp program.

6. Ideas of Implementing Handwriting Recognition

Recognition of Mathematical Expressions has proved to be a difficult computing task. However, the redundancy of multi-modal input can eliminate many potential errors in handwriting recognition [9]. Microsoft Tablet SDK contains examples of handwriting recognition; however, users without Microsoft Windows XP Tablet Edition or Office XP may not have handwriting recognition module installed. To add implementation of handwriting recognition to SKEME, we propose modeling the interaction by existing demonstration programs [7].

In the simplest approach, a "tap" with the stylus in a box would work as a mouse-click, but would provide a sketchpad for some math component. The detailed engineering of this, and its interaction with voice, is an opportunity for experimentation.

7. Selected Other Programs

We examined several other editing programs for mathematics input. Of the group, JMathNotes is perhaps the most successful program. It is remarkably strong, but suffers from difficulties in our testing of it with an untrained user. For example, we found inconsistencies in recognizing character such as "y", and differentiating between "s" and "5". Nevertheless, the system design and the range of options it provides shows prospects for as a good platform for handwritten mathematical input as we have seen [14].

TeXmacs is a fairly successful keyboard and mouse editor, and in particular has a growing following and a variety of useful plug-ins. The problems with TeXmacs include its complexity, depending on many other packages to be properly loaded, and the currently tentative port to non-Linux systems (requiring Cygwin). Since we are trying to use the Microsoft speech development package, a working version on Windows was important [15].

JOME was also considered as a base, which would have provided some of the needed facilities in Java, as well as provided support for OpenMath, in some sense an alternative with mathematical semantics to TeX. It has documentation in French and a substantial complexity. The requirement that expressions conform to its grammar makes it difficult to accept [1].

Current software such as *Mathematica* [4], Microsoft Equation Editor 3.0 [6], and Mathview and Expressionist [5]allow for mathematical input of symbols but they are limited to mouse and keyboard input and do not offer convenient ways to add speech and handwriting recognition modules. Numerous other systems such as EZMath [10], and FFES [12] were examined but generally failed on one facet or other of design, re-use, or availability.

8. Conclusion

SKEME offers a basic template for implementing multi-modal input of mathematical equations. The intent is to provide a prototype user interface with a framework in which more elaborate user interfaces can be built upon using Lisp common-graphics methods. This program addresses several problems with adding, editing, and displaying mathematical input, and provides a foundation to incorporate voice recognition and handwriting recognition.

9. Acknowledgments

This research was supported in part by NSF grant CCR-9901933 administered through the Electronics Research Laboratory, University of California, Berkeley and by the UC Berkeley College of Engineering UROP program.

9. References

[1] *JOME*. M@INLINE Group, 2000. http://mainline.essi.fr/jome/jome-en.html

[2] Kajler, Nobert and N. Soiffer. *A Survey of User Interfaces for Computer Algebra Systems*. Journal of Symbolic Computation, 25(2):127--159, February 1998.

[3] Knuth, Donald E. *The TeXBook*. Reading, Massachusetts: Addison Wesley Publishing Company, 1993.

[4] *Mathematica*. http://www.wolfram.com/products/mathematica/index.html

[5] *Mathview, LiveMath, Expressionist*, http://www.calculus.net/mathview/, http://www.livemath.com/matheq/

[6] *Microsoft Equation Editor 3.0* , http://www.microsoft.com/education/?ID=InsertEquation

[7] *Microsoft Speech SDK 5.1* http://www.microsoft.com/speech/download/sdk51/

[8] *Microsoft Tablet PC Platform SDK* http://members.microsoft.com/partner/products/windows/windowsxptabletpc/tabletplatform.aspx

[9] Nicholas Matsakis. *Recognition of Handwritten Mathematical Expressions*. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, May, 1999. http://www.ai.mit.edu/projects/natural-log/

[10] Raggett, Dave. *EzMath 1.1*. 1998. http://www.w3.org/People/Raggett/EzMath/

[11] Raman, T. V. *An Audio View of TeX Documents*. Proceedings of the TeX Users Group, 13:372-379, July 1992. http://www.tug.org/TUGboat/Articles/tb13-3/raman.pdf

[12] Smithies, Steve. *FFES*. Boston: Free Software Foundation, Inc., 1991. http://www.cs.queensu.ca/drl/ffes/

[13] Stevens, Robert and Alistair Edwards. *A Sound Interface to Algebra*. 1993. http://citeseer.nj.nec.com/stevens93sound.html

[14] Tapia, Ernesto. *JMathNotes* 0.3 Boston: Free Software Foundation, Inc., 2003. http://page.mi.fu-berlin.de/~tapia/JMathNotes/

[15] Van der Hoeven, Joris. *TeXMacs*. Boston: Free Software Foundation, Inc., 2003. http://www.texmacs.org/index.php3