# PANTECHNICON, OR 'LETS SUPPOSE THE INMOST SECRETS OF EMACS, TEX AND HYPERCARD LIE OPEN TO PROGRAMMERS'

Robin J. Popplestone

Computer and Information Science Department
University of Massachusetts

COINS Technical Report 90-13

# Pantechnicon, or 'Lets suppose the Inmost Secrets of Emacs, Tex and Hypercard Lie Open to Programmers'. [1]

Robin J. Popplestone

Department of Computer and Information Science
University of Massachusetts at Amherst, 01003 USA

*ABSTRACT*

This paper describes the development of an environment, *Pantechnicon*, which provides for the active display of information predominantly in the *typeset paradigm* familiar to users of TEX. Pantechnicon provides an editing environment which allows users to compose a document in a manner similar to EMACS, but extended by the ability to compose technical matter such as mathematics, tables or figures using a *technical keyboard* which appears in a window and is mouse-activated. The internal form of the document is manipulable by programmers in Common Lisp, ML, POP-11 and Prolog, running under the POPLOG system. Functions written in these languages can be associated with a displayed object as *scripts*, making the document *active* with Hypertext like qualities. The system is designed so that the incremental operations associated with editing have $O(\log n)$ time complexity, so that it scales appropriately to treat large documents.

Main topic: **User Interfaces**

Secondary topic: **Automated Reasoning, Constraint-based Systems**

[1]Preparation of this paper was supported in part by NSF grant number IRI-8709949, in part by ONR grant number N00014-84-K-0564. and University Reseach Initiative Grant N00014-86-K-0764.

# 1 Introduction

In the cuisine of many lands, visual presentation is regarded as being as important as any other part of the preparation of a meal. The users of computers, no less, should be able to feast their eyes upon the image of their information structures presented to them by their machines. Pantechnicon is a system that supports the generation and presentation of *documents, active with fine granularity,* in the *typesetting* paradigm immortalised for the computational world by Knuth [9], as opposed to the *typescript paradigm* of most current editors such as Emacs, the *extended typescript paradigm* of word-processors like MacWrite, the *desktop paradigm* of most window systems or the *technical drawing* paradigm of many CAD systems.

The computational correlates of these paradigms are the kind of relationships that can exist between displayed entities. The typescript paradigm embodies a left/right relationship only between individual characters of fixed dimensions, and an above-below relationship only between lines of characters; only the dimensional limitations are relaxed in the extended typescript paradigm. The desktop paradigm is based primarily on the over/under relationship between windows, which results in obscuration of one window by another. The technical drawing paradigm does not *per se* impose significant relationships between displayed entities although such relationships may be imposed by the associated application software.

The typeset paradigm imposes both left-right and above-below relationships between displayed entities. However, these relationships are not restricted to holding respectively between characters and between lines as in the typescript paradigm — they can hold between any displayed entities. Moreover there is a hierarchic *box/sub-box* structure present in the TEX embodiment of the typeset paradigm, to which Pantechnicon adheres in broad outline. While the primary paradigm supported by Pantechnicon is the typeset paradigm, it does support others, for example the technical drawing paradigm is (in a limited way) provided by the *picture* command (see section 4).

I shall refer to the representation of a document as a set of pixels in a window as the *iconic form* of the document, as opposed to its *internal form* as a tree data-structure from which the iconic form is derived.

The main design goals of Pantechnicon are as follows:

- To permit a user to *compose* a document freely. Pantechnicon allows one to compose plain text exactly as if one were using an ordinary editor, using a combination of key-strokes and mouse actions.
- Conversely, Pantechnicon is also intended to support the production of *constrained documents.* In actuality, all documents are subject to constraints. Both natural language and computer programs are subject to constraints on syntax, although the those on the latter are better formalised and more rigorous than those on the former. It is characteristic of documents produced for professional purposes that they are strongly constrained, whether they be engineering designs, mathematical formulae, computer programs or legal documents. A creative professional will act largely within these con-

straints, although one may need to reformulate them on occasions. Pantechnicon is designed to permit both syntactic and semantic constraints to be enforced. Necessary freedom of composition is combined with enforcement of constraints by the provision of a system of *watermarked pedigrees* for formalisms within a document. These capabilities form the basis of Pantechnicon's capability of supporting Text Based AI.

- To support the *typeset paradigm* as the default. Most professional activity produces documents which fit this paradigm, with included sub-documents, usually called *figures*, which themselves lie outwith the paradigm, but which are readily incorporated in the system design. Even in such an apparently visual area as engineering design[13, 14], it is essential to realise that only a small proportion of the design documentation has the form of large scale engineering drawings — the great bulk of components of most designs are bought-in, and much of the rest consists of standard form-features. Information derived from text-books, manuals, codes-of-practice, catalog entries, and justifications for particular choices form a the basis of large portion of the documentation of a design.

  For interactive use, a somewhat simpler implementation of the typeset paradigm than that embodied in TEX has been adopted, but the availability of this paradigm is seen as essential to the presentation of formal structures to the human user as a readily comprehensible image.

- To permit the creation of *scriptable active documents.* Any entity appearing in the iconic presentation of a Pantechnicon document can be made to respond in an arbitrary way to events (e.g. mouse hits) occurring within it. The behaviour so occasioned can be determined by a script written in any of the AI languages supported by the POPLOG system[8] in which Pantechnicon is implemented.

  Entities are scriptable at a fine grain, which extends down to the tips of the document-tree. The default script for any entity is "edit behaviour", which allows it to be extended and modified by mouse and keyboard events initiated by the user.

- To employ a technology that *scales acceptably.* Pantechnicon aims to support very large documents, so the complexity of the algorithms it employs must be appropriate. It is argued, in section 6 that the typeset paradigm supports good performance by virtue of a order isomorphism between the internal and iconic forms.

- To be portable across various machines, and to be able to produce output on printers at full printer resolution, and to be usable on the smaller workstations. One feature of this portablility is the constant relationship between the various AI languages themselves and to the windowing system enforced by the POPLOG system. Workstations, however, do not provide a uniform physical interface to people — the keyboard layouts differ, and mice are not all blessed with the same number of buttons. Pantechnicon provides a two-stage event-interpretation process. The first stage maps from events produced by the window system to 'Pantechnicon events'. E.g. pressing mouse button #1 may be mapped to the Pantechnicon event *begin_selection*. The second stage responds to the Pantechnicon event in a manner appropriate to the Pantechnicon entities in which it occurs.

- To be readily extensible, and accessible as an open architecture to AI programmers. The main implementation decision which makes it possible to meet this requirement is that a strategic ensemble of procedures in the system are *pure functions* without side-effects. This functional programming style makes it much easier to preserve consistency between the document present as a data-structure and its form as an image presented to the user, and hence renders the task of incorporating new entity types into the system much easier. The other essential attribute is that these functions be generic, in the sense that they are incrementally extendable for new forms of data.

An initial version of Pantechnicon exists currently — it is described in section 7. This paper describes the design of Pantechnicon, so that the present tense is used of all features of the system.

# 2 The User's Perception of Pantechnicon

Pantechnicon appears to the first-time user as an editor. The mouse can be used to *select* some portion of the document as the target for subsequent *insertions* or *deletions*, to *move* some portion of the document, or to *choose* an action from a menu. In so far as they are meaningful, key-bindings are compatible with EMACS. Cursor moving keys generate a *synthetic selection*, which is equivalent to an appropriate mouse-generated event.

However one will find that one has additional options to those normally provided. One menu-item serves to create a new window which is a *technical keyboard*. This provides a number of successively expanding options. For example the mathematics option allows one to compose mathematics, and is itself structured (e.g. there is a calculus sub-option, and a set-theory sub-option). The digital circuit option allows one to incorporate standard symbols for gates, multiplexers, counters, etc. in the user document (usually but not necessarily in a *picture*). The technical keyboard acts like a "soft" keyboard in that "pressing" any key, using the mouse, inserts the symbol or formula which is the key into the current edit buffer, at the currently selected place.

The user can create a customised 'soft' keyboard to his own specification. In effect, all one has to do is to create a document laid out as the desired keyboard, and then convert selected text items to being keys using a special menu-selection.

One of the keys of the technical keyboard inserts a picture into the current document. Once inserted, a picture can be made larger or smaller by mousing its frame. Initially pictures are empty. A mouse-selection on any unoccupied part of the picture inserts a dummy entity into the picture at that place. This entity can subsequently be edited in the standard way — e.g. plain text can be typed in, or the technical keyboard used to insert special symbols. Pantechnicon departs from TEX in its treatment of lines in a picture. To make it easier to adjust the appearance of a picture, the terminations of lines are attached to other objects, so that when they are moved the lines move also to preserve connectedness. This also makes it easier to attach semantics to lines, e.g. as connections in a logic diagram.

Mathematical formalisms may also be input using key-strokes, which can be faster for the practiced user than extensive use of the mouse. For this purpose, and others, a command line is provided at the base of each edit window which provides a (very limited) typescript 'second view'[3] of the document. If part of a mathematical expression is selected, its typescript form appears in this command line, which can then be edited in the manner of the command line of VED[8]. E.g. the command line $(x + y)/2$ will produce the iconic form $\frac{x+y}{2}$.

The result of any editing activity is a *document* whose form, modelled after that of TEX input, is an open standard within Pantechnicon. Thus, application programs can readily operate upon part or all of the document. The more advanced user is able, e.g., to write a program that will examine a picture that represents the design of an electronic circuit together with some test instruments, such as an oscilloscope, and to simulate that circuit, making its iconic form display the results of that simulation.

# 3 Comparison with previous work

The elements of which Pantechnicon is composed are familiar to workers in the field. Among editors, the various versions of Emacs are user-programmable. Most, however, are only programmable in the restricted 'Mock Lisp'[15] which is not a general purpose programming language. The Zemacs system, available on LISP machines, permits LISP users to have access to the edit buffer, and provides capabilities closer to those of Pantechnicon, albeit restricted to the typescript paradigm. Pantechnicon is distinguished from the tools commonly available in LISP environments by its adherence to the *typeset paradigm*, rather than the window paradigm.

My debt to TEX should be manifest: TEX is not, however, designed for interactive use, and would not be easy to adapt. TEX transforms a source document into a *dvi* file which embodies the decisions about where entities should be placed. Originally I tried to adapt this paradigm to interactive use by creating a procedure which would map an input data-structure in TEX-like form to an output data-structure with *dvi* like attributes extended by back-references to source text (to support the interpretation of mouse-hits). Maintaining consistency during edit operations using this architecture proved remarkably difficult, and I was driven to employ the present architecture of *decorating the source tree* with typesetting information, as described in section 6. TEX and LATEX are the major source of naming conventions in Pantechnicon, and of the details of arguments of commands. Logically, some departures from these conventions are necessary. For example the text of a section is a parameter of the *section* command in Pantechnicon.

While Pantechnicon's procedures do not produce output of the refinement of TEX output, the results are nevertheless pleasing to the eye It is intended to provide an option to output a document in TEX.

The Hypertext-like capabilities of Pantechnicon derive from the active nature of the iconic form of the text. The *ref* and *label* commands provide the ability to explore the text in a non-sequential manner. Among Hypertext systems, MUE[16] stands out as using a presentation

form of hierarchically organised rectangular boxes, which has a conceptual relationship to the typeset paradigm.

The overall organisation and functional programming architecture of Pantechnicon resemble that of the Two-view Document Editor[3], although this is conceived of as an editor rather than an environment.

Pantechnicon is distinguished from the mathematical tools (e.g. Mathematica) available for the Macintosh and other computers by its open architecture, and orientation towards supporting a wider set of formalisms than the pre-20th century mathematics which are the typical targets of such tools.

# 4 The internal representation of a document

The functions which operate on documents are abstracted away from commitment to any particular concrete representation, although by default a document is a Prolog term. In Poplog, this is not a data-type confined simply to the Prolog subsystem, but can be created and manipulated by suitable calls in POP-11 and LISP. It was chosen over the other alternative, the LISP S-expression as being more manifestly an applicative structure — lists tend to become overloaded as rag-bags for holding all kinds of data. The typescript form of Prolog terms, broadly familiar to most computer users, is used for keyboard input of mathematical expressions, although a the use of conversion function to the internal form preserves the abstraction.

Thus a document is a tree, of which the nodes are, by default, non-atomic Prolog terms. The functor of each of these terms is referred to as a *command*. E.g. $hbox(a, b)$ has the command *hbox*.

The tips of the tree are any other entity which POPLOG can represent. Detailed correspondences between the representation of the various datatypes of LISP, ML, POP-11 and Prolog are available within the POPLOG online manual. The typeset form of any one of these data-types is fully programmable, and may be active, allowing the construction of browsers.

The POP-11 *word* datatype, which is physically identical with the Prolog *atom*, has a special role in Pantechnicon, and imposes certain style constraints on the Pantechnicon presentation of formalisms. These constraints can best be characterised as being those of *engineering mathematics*. Words which happen to be the names of TEX mathematical symbols are presented as the typeset form of that symbol. E.g. "alpha" appears as $\alpha$. The underscore is taken to refer to a subscripting operation. E.g. *v_in* is presented as $v_{in}$.

The default presentation for LISP symbols, which are disjoint from the word/atom datatype discussed above, has not yet been determined. Since one of the more important formal objects to be displayed are S-expressions, the succinct mathematical style of expression rather militates against the common programming style which might be parodied as:

```
(THIS-IS-A-FUNCTION-FOR-FINDING-THE-DISTANCE-BETWEEN-TWO-POINTS
 THE-FIRST-POINT
 THE-SECOND-POINT)
```

Turning now to nodes, there are essentially two kinds, distinguished by principal functor: *typeset nodes* and *application domain* nodes.

Typeset nodes correspond to TEX/LATEX commands, and determine the spatial relationships between their sub-documents quite explicitly. E.g.

$$vbox(cat, dog) \quad \text{produces} \quad \begin{matrix} cat \\ dog \end{matrix}$$

Application domain nodes have a structure that is determined primarily by considerations other than typesetting. The only application domain currently implemented is the *math* domain. In this domain, the prolog term *diff(x/(x+1))* is presented as $\frac{d}{dx}\frac{x}{x+1}$. Pantechicon recognises a particular sub-document as being in an application domain by the presence of a application domain entry command as the principal functor of the sub-document — e.g. in $math(x + y)$, $x + y$ is recognised as being in the *math* application domain.

The rationale of providing application domain commands is to permit application-oriented data-structures to be present in a document. Thus the terms of the mathematical application domain have the 'natural' form which can be evaluated by the Prolog evaluation predicate *is*, rather than a form oriented to typesetting which they would have in TEX. It is intended to extend the definition of document-trees to permit LISP S-expressions to be used as a representation of mathematics, in a $math_{lisp}$ application domain.

Associated with an application domain is a function for transforming expressions of that domain into typeset form. This function must make use of the *lab* command (q.v.) to permit the iconic form of sub-documents to be related to the source form.

Below are summarised the main forms of document node available in Pantechnicon:

- $bf(Doc)$ Set the text specified by *Doc* in bold face.
- $rm(Doc)$ Set the text specified by the *Doc* in roman face.
- $expand(Symbol)$ Expand *Symbol* to a size to match other text which occurs with it in an *hbox* or *vbox*. Tex conventions are used for symbol naming.
- $frac(P, Q)$ Set the fraction $\frac{P}{Q}$. The base-line of the resulting box is chosen so that the bar of the fraction will align with a preceding minus sign.
- $hbox(D_1, D_2 \ldots D_n)$ Set the text specified by commands $D_1..D_n$ in a horizontal line, with the base-lines of each aligned horizontally, and aligned with the base-line of the resulting box.
- \(*hline*) produces an expandable horizontal line.

- *vbox*$(D_1, D_2 \ldots D_n)$ Set the text specified by commands $D_1 \ldots D_n$ in a vertical line, left justified.

- *subsuper*$(C, D, E)$. Set $C_D^E$ Either the sub-script or the super-script can be a null string.

- *framebox*$(m, dx, dy, j_x, j_y, T)$. $T$ is to be typeset in a box with dimensions $dx$,$dy$ and justified within that box as specified by $j_x$ $j_y$. $x$ justification is "l", "r", "c" and $y$ justification is "t","c","b". The box is put on the screen with a border.

- *picture*$(x, y, P_1, \ldots P_n)$ Creates a picture, with documents $D_1 \ldots D_n$ placed as specified by *put* commands

- *put*$(x, y, D)$ $D$ is to be placed at location $x, y$ in the current picture (which must be defined immediately above the *put* command in the tree).

- *lab*$(i_1, i_2, \ldots i_n, D)$ This form of *label* command is used to establish a correspondence between application documents and their typeset counterparts. This correspondence is required by the event handlers that deal with pointer events (e.g. mouse hits). Essentially $i_1 \ldots i_n$ provide a path in the original application-domain document to the sub-document whose iconic form is produced by the typeset document so labelled.

- *label*$(L)$. This generates a tree address in the document tree which is associated with the label name $L$, and which can be referred to in corresponding *ref* commands. It also generates a serial number, which may be present in the iconic form of the *ref* command.

- *ref*$(L)$. This allows a reference to be made to a label elsewhere in the document, both for the TEX-like purpose of providing a reference number, and for the Hypertext-like purpose of allowing the user to jump to it.

- *include*$(File)$. The saved form of a document contained in *File* is to be a sub-document. Provided the necessary auxiliary information is available, the physical inclusion of the file in the document tree in main memory is delayed until reference within it is made.

- *key*$(Doc)$ This command places the document in the window, with a frame round it. When a *select event* occurs within it, rather than passing on the event to sub-boxes, instead it inserts the document into the current edit window. Thus the *key* command is used to construct specialised "keyboards" containing e.g., mathematical symbols.

- *math*$(Doc)$. *Doc* is a mathematical application document, to be translated and typeset.

Additional commands are being formulated. In particular a set of document structuring commands, *paragraph*, *section*, *chapter* and *book* are necessary to provide a bounded branching factor on the document tree.

# 5 Pedigrees and Watermarks

In this section I describe the apparatus by which Pantechnicon essays to grant the user legitimate freedom of expression whilst restraining him from the licentious fabrication of falsehoods. Lest an accusation of the aforesaid moral turpitude be laid at my door, I hasten to add that the contents of this section represent a proposed extension to Pantechnicon.

Certain commands occurring in a document create *formal sub-documents*, which can have a *pedigree*. A formal sub-document — let us call it indiscriminately a *formula* — may take a variety of forms, e.g. an equation, a table, a function definition. A pedigree is related to the concept of a *justification* in a Truth Maintenance System[6], and to a citation or reference in TEX: it fully characterises the process by which a formala is derived, which includes the possibility that it is asserted without justification.

A legitimate derivation of a formula $E$ from a formula $E_1, E_2 \ldots E_n$ is achieved by applying a function $f_{derive}$ to the $E_i$, possibly with additional arguments. E.g. $E_1$ might be a rational function to be expressed as partial fractions, $f_{derive}$ might be a function for expressing rational functions as partial fractions, and $E$ might be the partial fraction form of $E_1$. Another form of derivation might be to select particular values from a tabulated relationship. The pedigree of $E$ specifies the $E_i$, $f_{derive}$ and any additional arguments. The $E_i$ are referred to 'by name', i.e. by using a *label* referring to a place in the current document, or a *citation* of any other document, so that their provenance is manifest.

Editing a formula is both possible and at times desirable, provided that the editing action explicitly nullifies the pedigree. Given the open architecture of Pantechnicon, it will be possible, accidentally or maliciously, to create bogus pedigrees. As a safeguard against an occasion so unhappy, pedigrees contain a *watermark* which is obtained by applying a hashing function during the derivation, access to the hashing function being limited.

# 6 The Implementation and Performance of Pantechnicon

The dominant paradigm of any system for presenting and modifying information structures strongly determines the design of the algorithms of acceptable complexity for performing such presentations and modifications. In Pantechnicon the internal form of a document is a tree, with a branching factor that users are encouraged to keep bounded by the availability of a hierarchy of document-structuring commands. An ordering relationship can be associated with this tree, as in [4]. This ordering is the inverse of what one might informally describe as the tree/sub-tree relationship. To be more precice, the ordering is defined on *paths* (or *occurrences*) which define how to reach a particular sub-tree.

The iconic form of a document can be thought of as consisting of functions (binary for black and white) which are defined on sets of pixels constituting a rectangular *box*. Set inclusion on these boxes defines an ordering relationship on the iconic form. *The ordering defined on*

*the iconic form is order-isomorphic to the inverse of the ordering defined on the tree form*[2].

The complexity of the algorithms employed in Pantechnicon depends crucially upon the document being a tree of bounded branching factor $k$. The complexity of mouse interactions depends in addition upon the order isomorphism between the iconic and internal forms.

The connection between the internal and iconic forms of a document is made by the *value* function. *value* is a pure function, memoised[11, 12] for good performance. In the most simple cases[3], $value(D, Font)$ is a *box record* which specifies the width, height and ascent[7] of the iconic form of $D$, together with a *map* which is a vector specifying where the iconic forms of the sub-documents of $D$ should go relative to the iconic form of $D$

Because of the memoisation, the complexity of *value* is history-dependent. Suppose $D$ is a document which, considered as a tree, has depth $d$. Suppose $D$ has $k_1 < k$ children, $D_1 \ldots D_{k_1}$. Then, if $D$ is a new document, not yet in the memory of *value*, to determine $value(D, Font)$, *value* must be called recursively $k_1$ times, yielding boxes $B_1 \ldots B_{k_1}$. In Pantechnicon, any processing to determine the position of the iconic forms of the $B_i$ in the iconic form of $D$ is of complexity $O(k)$, since a bounded number of passes is used to satisfy the spatial relationships, and only the fixed-size components of the $B_i$ are examined, and not the *map*. Note that determining *value* does NOT actually paint the iconic forms on the screen. We have two limiting cases

1. If no sub-tree whatever of $D$ has been incorporated in the memory of *value* then the complexity of *value* is $O(k^d)$.

2. If every proper sub-tree of $D$ has been incorporated in the memory of *value* then the complexity of *value* is $O(k)$, assuming that access to the address-hashed memory of *value* takes constant time.

Case (1) holds when a document is to be typeset for the first time — all of the functions which present the iconic form of a document to the user evaluate *value* for the whole document. However if a document is created incrementally by editing actions, with *value* fully memoised (i.e. no purging of the memory takes place), case (2) holds for all the calls of *value* occurring during the creation of a document. The most important edit action is the replacement of a sub-tree $D_1$ at a path $p$, written $D[p \leftarrow D_1]$ in [4]. This has complexity $O(kd)$, since every tree-node along the path $p$ of length $\leq d$ has to be re-built, each taking time $\leq k$. Each replacement leaves $\leq d$ nodes whose *value* is not memoised, requiring time $O(kd)$ to perform.

The time-complexity for interpreting mouse-hits can be analysed in a similar way, because of the order-isomorphism between the internal and iconic trees. A selection arising from a mouse hit generates a path of length $\leq d$. In determining each step of the path, $O(k)$ calls of *value* are required, each taking constant time if *value* is kept fully memoised. Thus the time complexity of interpreting a mouse-hit is $O(dk)$.

[2]Pictures are an exception to this, which means that they should be 'small'.

[3]Application domain documents have extra stages of evaluation involved before a box becomes available. The complexity arising from this additional work is domain-specific

The most meaningful measure of the size of a document from the user's point of view is the number of leaves, $n$, say. Provided the tree is balanced[4] and $k$ is bounded, $O(dk)$ complexity translates to $O(\log n)$ complexity, and $O(k^d)$ complexity translates to $O(n)$ complexity.

The complexity of the operations required to maintain the iconic form that the user sees consistent with the internal form is not discussed here. Incremental updating of the screen is discussed in [3]. Much of the work in obtaining adequate performance is more to do with careful use of block-moves of pixels to obtain small constant multipliers rather than being related to the size of the document itself.

The assumption that the *value* function is fully memoised is reasonable for a small document created in a single session. Clearly the issue arises as to whether the memory should be persistent between sessions. In the case of a large document created over many sessions, full memoisation is undesirable, since the whole document tree may not be present in main memory. The compromise to be adopted is to save the memory down to the *paragraph* level between sessions. The rationale for this is that paragraphs are commensurate in size to windows — typically only a few paragraphs will appear in a window — and that the time-complexity of presenting a document to a user at a location in the document specified by a path (e.g. if he requests to access a reference generated by $ref$) will typically be dominated by the cost of painting into the window the tips of the tree that actually appear, if the path is not 'near' any path which was previously displayed. Thus the cost of recreating the *value* memory for the part of the document that actually appears will be commensurate with that of actually painting it on the screen.

# 7 Discussion

An initial implementation of Pantechnicon exists. It is possible to create edit windows and enter a mixture of plain text and mathematics into them, to mark text with a mouse and replace it. The response to key-depressions is usably fast, though further work on incremental repainting is needed to reduce flicker. Work on writing a document to backing store, reading it back and writing in Tex format is in progress. The implementation currently runs under Suntools using the Poplog Window Manager, but will migrate to using the X-toolkit interface provided in POPLOG Version 14. The system does not yet provide mouse-mediated access to scripts in the way that Hypercard does, nor are data-structure browsers or debuggers based on Pantechnicon available. A rewrite-rule based algebraic simplification system will shortly be integrated with Pantechnicon.

The translation from the mathematical application domain to the standard typeset form is currently done in Prolog, which is a convenient formalism for expressing such a 'case based' transformation. The use of Prolog for this purpose does however create a problem in that it does not support the distinction made in LISP by $EQ$ versus $EQUAL$, or = versus == in POP-11, which makes control of the space complexity of the program difficult.

[4]This depends upon the compositional habits of the users, and should be capable of experimental verification once a corpus of documents is established.

# References


[1] Adobe Systems Inc [1985], PostScript Language Reference Manual, Addison Wesley, Reading MA.

[2] Barrett,R., Ramsay,A. and Sloman A.,[1985] POP-11 A Practical Language for Artificial Intelligence, Ellis Horwood, Chichester, England and John Wiley N.Y.,USA.

[3] Brooks,K.P [1988] A Two-view Document Editor with User-definable Document Structure, Ph.D. Dissertation, Dept. Computer Science, Stanford University. (Available from Digital Systems Research Center, Palo Alto CA USA).

[4] Buchberger B and Loos R., [1982] 'Algebraic Simplification' in eds. B.Buchberger G. E.Collins and R.Loos in cooperation with R.Albrecht, Computer Algebra, p30.

[5] Clocksin W. and Mellish C. 1981, Programming in Prolog, Springer Verlag N.Y..

[6] de Kleer,J. 1984 Choices without Backtracking, Proc Conf AAAI., pp 79-85.

[7] Gettys J., Newman R. and Scheifler,R.W.,[1988], X-Lib — C Language X-Interface — Protocol Version 11, MIT Project Athena Report.

[8] Hardy S., 1984, A New Software Environment for List-Processing and Logic Programming, in Artificial Intelligence, Tools, Techniques and Applications, eds. O'Shea,T. and Eisenstadt,M., Harper and Row, N.Y..

[9] Knuth, D.E.,[1984], The TEXbook, Addison Wesley, Reading MA.USA.

[10] Lamport, L., [1986], Latex, A Document Preparation System, Addison Wesley, Reading MA.USA.

[11] Michie,D. 1968 "Memo functions and Machine Learning, *Nature, 218* 19-22.

[12] Popplestone,R.J. 1967 Memo functions and the POP-2 language. *Research Memorandum MIP-R-30* Edinburgh: Department of Machine Intelligence and Perception.

[13] Popplestone R.J. 1985, An Integrated Design System for Engineering, Proceedings of the 3rd ISRR, Gouvieux, France.

[14] Popplestone, R.J. 1988 "The Edinburgh Designer System as a Framework for Robotics: The Design of Behavior", Artificial Intelligence for Engineering Design Analysis and Manufacture, Vol 1 No 1.

[15] Stallman R. (1986), GNU Emacs Manual, Fifth Version, Free Soft Foundation, Cambridge, MA USA.

[16] Travers,M. A Visual Representation for Knowledge Structures, Proc. Hypertext 89, ACM, NewYork, USA.