Aleph — A Language for Typesetting

Luigi Semenzato* and Edward Wang

Computer Science Division, University of California,
Berkeley, California, 94720 USA
luigi@CS.Berkeley.EDU
edward@CS.Berkeley.EDU

ABSTRACT: *Aleph* is a programming language for batch typesetting systems. It has conventional programming-language features for implementing style packages, extensions, and even the typesetting engine itself. At the same time, an extensible syntax makes Aleph a convenient document input format. This paper introduces Aleph and demonstrates with examples the convenience of implementing typesetting algorithms in Aleph.

KEYWORDS: typesetting, text processing, Lisp, programming languages.

1 Introduction

Advances in workstation technology and user interface design in recent years have dramatically increased the quality of WYSIWYG typesetting systems. Yet, batch systems continue to be popular because they are naturally easier to program, and therefore more flexible and powerful. This is in spite of the fact that no existing system is based on what we would consider a *good* programming language. We believe that the underlying language strongly affects the overall quality of the system, for both the expert and the casual user.

Aleph is our attempt to design a good typesetting language, by turning the traditional language inside-out. Instead of accepting text with some embedded commands, Aleph is a programming language with *embedded text*. This shift in emphasis leads to two features that directly address the main weaknesses of conventional typesetting languages.

Traditionally, the input to a typesetting system is thought to be mainly text. In these systems, the commands are an afterthought. They tend to follow the syntactic and lexical conventions of the surrounding text, which are invariably awkward for a programming language. Whitespace and line breaks are usually significant, and control structures are often difficult to

*On leave from Dipartimento di Elettronica e Informatica, Università di Padova.

use and to read. In addition, changes to the syntax (sometimes a useful feature) can affect the commands making the change.

On the other hand, by thinking in terms of a programming language, it is natural to give the text (which is just embedded literal data) different syntactic and lexical rules. Thus each aspect of the language can have its own conventions, tailored to its specific purpose. In Aleph, the syntax extensions are supported by a flexible and powerful inheritance mechanism. Ye, they affect only the embedded text. The programming language has a fixed syntax.

In the traditional language, programming is done with macros, because they are the simplest to implement and the most natural way to extend a mostly textual language. Also, again because programming is an afterthought, these languages do little to support large or even medium-sized programs. Yet, they are used to write large format packages, which are in effect large programs.

With Aleph, we began the design with a programming language. Indeed, our current implementation is an extension of Common Lisp [Steele]. By adopting a proven programming language out right, we ensure the ability to write large programs, and avoid the difficulty of learning a new language. We have also taken care to follow accepted language design principles, using lexical scoping where possible and keeping side effects to a minimum. The result is, we believe, a consistent and easy-to-use language.

In this paper, we will often compare Aleph to Tex [Knuth]. We use Tex as the representative language because it is familiar, and because it produces high-quality output but is difficult to program. Large packages have been written for Tex, but such large programs are difficult to understand and to modify, and sometimes difficult to use. Latex, for example, is several thousand lines of code (many more if the various style files are counted) [Lamport]. It attempts to present a better interface than that of Tex, but the result is uneven. There are many places where small inconsistencies (such as commands that fail in certain contexts) make the whole system difficult to use.

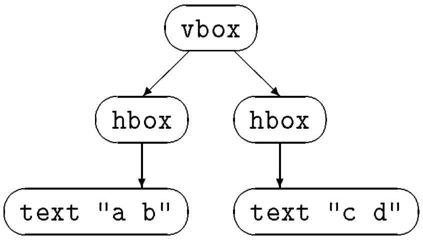
Section 2 is a quick overview of Aleph. Sections 3 and 4 show examples of Aleph programs to solve fairly difficult typesetting problems. This paper concentrates on the programming aspects of Aleph. A previous paper contains a more complete introduction to Aleph and a detailed description of the syntax extension mechanism [Semenzato]. We will assume some knowledge of Tex and Lisp.

2 Overview

In Aleph, text is represented by trees similar to nested Tex boxes and lists.[1] For example, a tree like the Tex box of boxes

```
\vbox{\hbox{a b}\hbox{c d}}
```

looks like this:



Trees are built by Lisp forms (called, surprisingly, *tree-building forms*).[2] The last example is constructed by

```
(vbox (hbox (text "a b"))
      (hbox (text "c d")))
```

Tree-building functions `vbox` and `hbox` are like their Tex counterparts. The function `text` makes a node with some text.

An input document, then, is a tree-building form or a sequence of such forms.

However, Lisp forms are ungainly things to enter and to read. Therefore, we have defined shorthands for most tree-building forms. For example, instead of

```
(group (text "A ")
       (italic (text "short"))
       (text " sentence."))
```

we simply say

```
[A @italic[short] sentence.]
```

We call the `[...]` construct an *Aleph string* (or just *string*). Using strings, the input to Aleph is reasonably readable:

[1] Aleph follows Tex's internal typesetting model partly because we use Tex as the typesetting engine (see Section 6), but mostly because we have no objection to that aspect of Tex.

[2] A *form* is what Lisp calls an expression.

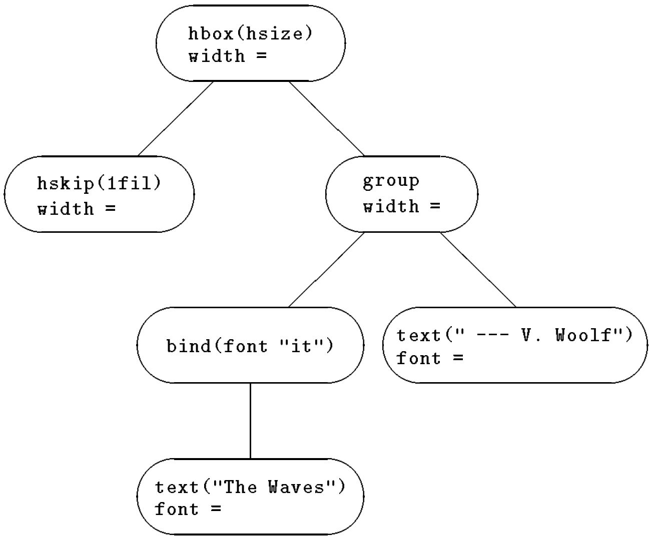
```
(par [``I see a ring,'' said Bernard,
``hanging above me.  It quivers and hangs
in a loop of light.''])

(par [``I see a slab of pale yellow,'' said Susan,
``spreading away until it meets a purple stripe.''])

(line (right [@italic[The Waves] --- V. Woolf]))
```

The characters in an Aleph string are interpreted according to user-definable rules called *modes*. Modes form a multiple-inheritance hierarchy. Adding new modes to the hierarchy is the basis for syntax extension in Aleph. The details of how to do this are described elsewhere [Semenzato].

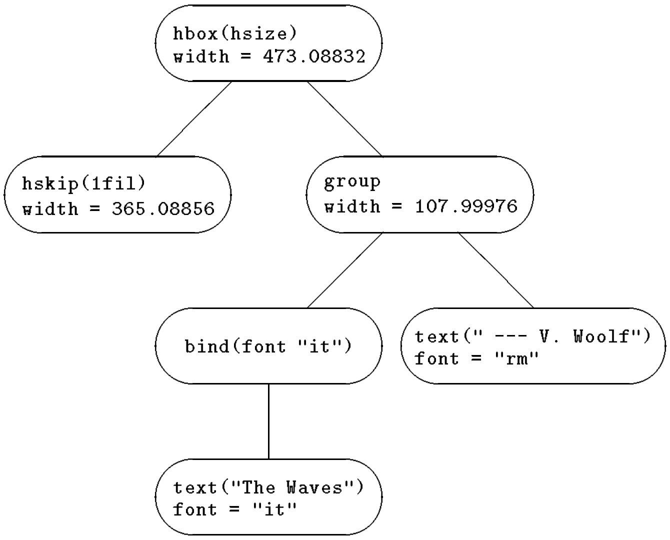
Because user control over syntax is limited to the text between `[` and `]`, the programming language, Lisp, remains constant.

Trees are turned into typeset output by a transformation pass we call *traversal*. Thus, instead of Lisp's read-eval-print loop, Aleph input is processed by a read-eval-traverse-print loop. Further processing, like resolving cross references, is done by additional tree-transformation passes before traversal.

The main job of the traversal pass is to finalize the size and shape of the objects in a tree. Mostly, this is a matter of filling in blanks. The last line of the previous example builds this tree:



The tree shows some blank fields where attribute values need to be filled in. In some cases, like the font attribute for text nodes, the value is found in some global parameter. These parameters can be locally rebound. For example, the bind node above changes the font parameter for the node below it. Other attributes are computed by the traversal (e.g., the various width fields). Thus, how a tree is typeset depends on the tree and the set of parameter values. This is the result (given some assumptions about the parameters):



In the general case, the action performed while traversing a node is determined by the traversal function for that node. Each kind of node has its specialized traversal function. In addition to computing attribute values, the traversal function can rewrite the node or the entire subtree below it. Very complex typesetting tasks can be done this way. The following sections give two examples of such fancy traversal functions.

Most simple tasks do not need traversal-time intervention. They can be done by building trees out of predefined nodes. For example, the function `italic` is a simple Lisp function that builds a bind node:

```
(defun italic (x)
  (bind :font "it" x))
```

Take then, your Magnificence, this little gift in the spirit in which I send it: wherein, if it be diligently read and considered by you, you will learn my extreme desire that you should attain that greatness which fortune and your other attributes promise. And if your Magnificence from the summit of your greatness will sometimes turn your eyes to these lower regions, you will see how unmeritedly I suffer a great and continued malignity of fortune.	Pigli adunque Vostra Magnificenzia questo piccolo dono con quello animo che io lo mando; el quale se da quella fia diligentemente considerato e letto, vi conoscerà dentro uno estremo mio desiderio che Lei pervenga a quella grandezza che la fortuna e le altre sue qualità gli promettano. E se Vostra Magnificenzia dallo apice della sua altezza qualche volta volgerà gli occhi in questi luoghi bassi, conoscerà quanto io indegnamente sopporti una grande e continua malignità di fortuna.

Figure 1: Sample output from `balance`

3 Example: Balancing Columns

The two examples we present in this section and the next are somewhat more elaborate than common typographic practice. They are, however, realistic, and they represent the Aleph equivalent of examples from *The TEXbook*.

The first example is the definition of a node with two children, to be typeset in two columns next to each other. The columns contain text in two languages, and we want to use column widths that give the columns the same height. This problem is proposed by Knuth at page 387 of *The TEXbook*. Figure 1 is the output from some sample text [Machiavelli].

For comparison, we show the Tex and Aleph code in Figures 2 and 3, respectively. The Tex version is from *The TEXbook*. The two versions use the same algorithm.

The Aleph code begins with the definition of a new node type, `:balance`, including a constructor function, `balance`, and a traversal function, `balance*`. To create a `:balance` node one would simply write `(balance` x y`)`, where x and y are the contents of the two columns:

```
(balance
  (par [Take then, your Magnificence, ...])
  (par [Pigli adunque Vostra Magnificenzia ...]))
```

The traversal function does all the work. The main loop (the `loop` form in `balance*`) tries different sizes for the left and right columns until it finds one that satisfies the constraints. The loop also exits if the iteration limit is reached. In this case, the solution is near optimal. The height of the

```
\newdimen\doublewidth \newdimen\delheight \newif\iffail \newcount\n
\newdimen\trialwidth \newdimen\lowwidth \newdimen\highwidth
\def\balancetwocols{\lowwidth=10em % lower bound on \trialwidth
  \highwidth=\doublewidth \advance\highwidth-10em % upper bound
  {\n=1 \hbadness=10000 \hfuzz=\maxdimen % disable warnings
    \loop \maketrial \testfailure \iffail \preparenewtrial \repeat}
  \maketrial} % now under/overfull boxes will be shown
\def\maketrial{\trialwidth=.5\lowwidth
  \advance\trialwidth by.5\highwidth
  \setbox0=\vbox{\hsize=\trialwidth \firstcol}
  \setbox2=\vbox{\hsize=\doublewidth\advance\hsize-\trialwidth
               \secondcol}}
\def\testfailure{\dimen0=\ht0 \advance\dimen0-\ht2
  \ifnum\dimen0<0 \dimen0=-\dimen0 \fi
  \ifdim\dimen0>\delheight \ifnum\n=10 \failfalse\else\failtrue\fi
  \else\failfalse\fi}
\def\preparenewtrial{\ifdim\ht0>\ht2 \global\lowwidth=\trialwidth
  \else\global\highwidth=\trialwidth\fi \advance\n by1 }
```

Figure 2: Tex's column-balancing code

```
(defnode :balance
    :constructor (balance (a b) (node :balance 0 a 1 b))
    :traverse-function 'balance*)

(defun balance* (n)
  (let* ((left (attr 0 n))    ; first subtree
         (right (attr 1 n))   ; second subtree
         (width (- (param hsize) !20pt))
         (left-min 0)
         (left-max width)
         (trials 0))
    (loop
     (let* ((left-width (/ (+ left-max left-min) 2))
            (l (bind :hsize left-width left))
            (r (bind :hsize (- width left-width) right))
            (delta (- (formatted-height l) (formatted-height r))))
       (if (or (= delta 0) (> trials 10))
         (return
           (traverse (line (group (vbox l) (hfill) (vbox r))))))
       (if (> delta 0)
         (setq left-min left-width)
         (setq left-max left-width))
       (incf trials)))))
```

Figure 3: Aleph's column-balancing code

columns is computed by `formatted-height`, a low-level function that could be written like this:

```
(defun formatted-height (x)
  (attr :height (traverse (vbox (copy-tree x)))))
```

Note that the Tex loop expands the macros representing the columns (`\firstcol` and `\secondcol`) many times.[3] The Aleph loop makes copies of trees instead, and traverses them. A rule of Aleph is that traversal functions must not have side effects outside the tree being traversed. So it is always safe to make copies of trees and do anything with them. In Tex it is often necessary to take precautions against the side effects of macro expansion.

4 Example: Alignment

This example is a `:table` node with functionality similar to Tex's `\halign`. Unlike the last example, we cannot compare the Aleph code to the Tex code, because `\halign` is written in WEB and is about 900 lines long. Figure 4 shows the bulk of the traversal function for `:table`, about 60 lines. We are not showing the constructor function and some auxiliary functions, about 40 lines. Figure 5 is the input for a table equivalent to the one on page 236 of *The TEXbook*.

The constructor function produces a `:table` node with children that can be either `:row` nodes or `:special-row` nodes. Entries in the `:row` nodes determine the column widths. The other nodes are used to span several columns or across the whole table.

One of the arguments to the table constructor is a template (fourth line in Figure 5). For each column, it specifies a list of functions, called *wrappers*. These are constructor functions that take one tree as argument and return another. During tree construction, table entries are passed to their wrappers and the results are stored in `:row` nodes. Typically, a wrapper will surround the entry with glue, or a node that modifies some typesetting parameter, such as the font. As wrapping is part of tree construction, the traversal function never sees the template.

The traversal function `table**` reorganizes the rows into columns to compute the column widths, then inserts padding between rows and columns as requested. Most of the complexity is due to the various options given to the user for specifying padding. Operations like transposing the rows into columns (`rows-to-columns`) and finding the width of a column (`column-width`) can be expressed very concisely in Lisp.

To be fair, `\halign` is somewhat more powerful than the `:table` node. Since `\halign` is based on textual substitution, the template can contain

[3] By the way, `\balancetwocols` as shown here expands into some extra whitespace, because of two unabsorbed end-of-line characters in its definition. We leave it to the reader to find them.

```
;;; The traversal function for :table
(defun table** (node)
  (let* ((rows (attr :children node))
         (columns (rows-to-columns rows))
         (widths (mapcar #'column-width columns))
         (pad (or (attr :pad node)
                  (/ (slack (attr :spread node) (attr :width node) widths)
                     (length columns))))
         (vpad (attr :vpad node))
         (rows-and-spaces (expand-rows rows widths pad vpad)))
    ;; Make a new node that's a vbox of all the rows, and traverse it.
    (traverse (vbox (group* rows-and-spaces)))))

;;; Convert the rows (excluding the special rows) to columns.
;;; The result is a list of list of nodes.
;;; The incoming rows are a list of nodes,
;;; each of which is a :row or a :special-row.
(defun rows-to-columns (rows)
  ;; remove special rows
  (setq rows (remove-if #'(lambda (x) (eq (operator x) :special-row)) rows))
  ;; (apply #'mapcar #'list ...) is an idiom for matrix transpose.
  (apply #'mapcar #'list (mapcar (attr-f :children) rows)))

;;; Given a column, return the maximum width of the elements.
(defun column-width (column)
  (apply #'max (mapcar #'formatted-width column)))

;;; Compute the difference between the specified table width (if any)
;;; and the natural table width.
(defun slack (spread table-width column-widths)
  (or spread
      (if table-width
        (- table-width (reduce #'+ column-widths))
        0)))

;;; Insert the correct vertical and horizontal padding into the table.
(defun expand-rows (rows widths pad vpad)
  ;; Do the horizontal padding within each row first.
  (let ((new-rows
         (mapcar #'(lambda (row) (expand-row row widths pad)) rows)))
    ;; Now do the vertical padding between the rows.
    (if vpad
      (cons (first new-rows)
            (mapcan #'(lambda (row) (list (vskip vpad) row))
                    (rest new-rows)))
      new-rows)))

;;; Insert horizontal padding into a row.
;;; The result is an hbox.
(defun expand-row (row widths pad)
  ;; The special rows are handled elsewhere.
  (if (eq :special-row (operator row))
    (special-row row widths pad)
    (hbox (group* (mapcar #'(lambda (row width) (expand-pad row width pad))
                          (attr :children row)
                          widths)))))
```

Figure 4: The `:table` traversal function

```
(table
 :pad !0.5cm
 :vpad !2pt
 :template ((right bf) (center it) center center left)
 :rows
 (((sl [American]) (sl [French]) (sl [Age]) (sl [Weight]) (sl [Cooking]))
  ((sl [Chicken]) (sl [Connection]) (sl [(months)]) (sl [lbs.]) (sl [Methods]))

  (:span-all (left (vbox [] !0.1in)))

  ([Egg] [Oeuf] [$-{2\over3}$] [$1\over6$] [Boil, Fry, Poach, Raw])
  ([Squab] [Poussin] [2] [$3\over4$ to 1] [Broil, Grill, Roast])
  ([Broiler] [Poulet Nouveau] [2 to 3] [1$1\over2$ to 2$1\over2$]
   [Broil, Grill, Roast])
  ([Fryer] [Poulet Reine] [3 to 5] [2 to 3] [Fry, Saut@'e, Roast])
  ([Roaster] [Poularde] [5$1\over2$ to 9] [Over 3] [Roast, Poach, Fricassee])
  ([Fowl] [Poule de l'Ann@'ee] [10 to 12] [Over 3] [Stew, Fricassee])
  ([Rooster] [Coq] [Over 12] [Over 3] [Soup stock, Forcemeat])))
```

Figure 5: Table example

macros or active characters that produce mode changes. The templates in Aleph only have an effect at tree-construction time. They cannot affect the way in which characters in the entries are interpreted.

We believe our code is easier to understand and to modify, and more robust and predictable than `\halign`. The very first comment to the alignment section in the Tex program says:

> It's sort of a miracle whenever `\halign` and `\valign` work, because they cut across so many of the control structures of TEX.

This sums up nicely the problems we are attempting to solve with Aleph.

5 Related Work

In his work on Lilac, Brooks has independently raised some of the same issues that spurred our work [Brooks]. In particular, he has chosen a program-on-the-outside input language, and functions over macros for programming. However, Lilac is a two-view editor rather than a batch typesetter. To facilitate the inverse mapping from output back to input, his input is a more limited, fully functional language. In contrast, Aleph must, and does, have better programming facilities and a more convenient source syntax for entering text. In addition, Lilac lacks the equivalent of the Aleph tree as a first-class, intermediate representation for text.

6 Final Comments

The examples we have shown all execute, but Aleph is not yet a complete system. It lacks, among other things, line-breaking and page-layout algorithms. To function, it generates output in Tex, and communicates with a Tex process running in parallel to perform tasks it cannot do itself.

We have shown Aleph used as a Tex-like low-level typesetter. It is also possible to build a more declarative interface on top of the core system, much like the way Latex is build on top of Tex [Lamport]. We believe such a system can be more consistent and easier to use than Latex, which does not completely hide some of the quirks of Tex.

Beyond that, since Aleph trees are flexible and user definable, it should be possible to define node types that correspond to an SGML-like descriptive markup. Such a document description can then be transformed (either at traversal time or by additional passes) into ready-to-typeset, low-level trees.

With its extensibility, Aleph is also the perfect framework for research into typesetting algorithms. As implemented in Lisp, it imposes no barrier between the core and user extensions. (Tex, for example, has primitives written in Pascal, which is inaccessible to users writing Tex macros.) As the system becomes more complete, we will be looking into these possibilities.

References

[Brooks] Kenneth P. Brooks. *A Two-view Document Editor with User-definable Document Structure.* Technical Report number 33. Systems Research Center, Digital Equipment Corporation. November, 1988.

[Kernighan] Brian W. Kernighan. *Issues and Tradeoffs in Document Preparation Systems.* in Proceedings of the International Conference on Electronic Publishing, Document Manipulation & Typography, R. Furuta, ed., pages 1–16. Cambridge University Press, Cambridge, 1990.

[Knuth] Donald E. Knuth. *The TEXbook.* Addison-Wesley, Reading, Mass., 1984.

[Lamport] Leslie Lamport. *Latex: A Document Preparation System.* Addison-Wesley, Reading, Mass., 1986.

[Machiavelli] Niccolò Machiavelli. *Il Principe.* 1513.

[Semenzato] Luigi Semenzato and Edward Wang. *A Typesetting Language Should be First a Programming Language.* Tex Users Group Annual Meeting, 1991.

[Steele] Guy L. Steele. *Common Lisp: the Language.* Digital Press, Burlington, Mass., 1984.