A Text Processing Language Should be First a Programming Language

Luigi Semenzato, Edward Wang
Computer Science Division, University of California, Berkeley, California, 94720
Internet: `luigi@ginger.Berkeley.EDU`, `edward@ucbarpa.Berkeley.EDU`

Abstract

Historically, typesetting languages have been designed for the entry of text. An embedded command language has since become important, indeed essential, but has remained a second-class citizen, sometimes masquerading as text, invariably clumsy and inadequate. We have designed a language that is a full-function programming language *with embedded text.* This shift in emphasis results in a level of consistency, flexibility, and power not otherwise possible.

Introduction

A batch-style computer typesetting system accepts text files as input, to produce formatted documents as output. Most such systems are extensible. They allow definitions of new document styles and commands. Some, like TEX, also allow the input syntax to be changed. To do all this, the format of the input must be a complex language. The design of this language affects the robustness, ease of use, and overall quality of the whole system.

A document, therefore, is a mix of text and commands, some of which define new commands or make syntax changes. Existing systems have emphasized the text portion of the input. In these languages, the commands are an afterthought. They often follow the inconvenient lexical conventions of the surrounding text, and make awkward programming languages. This paper describes our attempt to reach a better design, by turning the traditional language inside out, giving priority to commands and programming. We call this system and its language *Aleph.*

An Aleph document is a sequence of commands, some with embedded text as arguments. The commands are in a programming language with a fixed syntax. Text, on the other hand, can have a user-specified syntax. Each command builds an internal representation of a portion of the document. This representation is then processed to produce the output.

Aleph is an evolving design. Its current realization (sometimes called Aleph_0) is written in Lisp. Our immediate goal is not to produce a complete typesetting system, but to design a language that is a tool for both writing the system and using it. One consequence is that the Aleph system does no actual typesetting, but generates TEX as output.

The sections of this paper describe selected aspects of Aleph, in this order: basic constructs, extensible syntax, internal representation, implementation. The rest of this introduction is a discussion of some of the issues in typesetting-language design.

Syntax separation. Commands should not obey the syntax of the text around it.[1] For example, it is often convenient to ignore whitespace and line boundaries in a program, but not always possible in the text of a document. In TEX, it is sometimes hard to predict whether spaces and newlines in and around commands will be part of the output. User-defined syntax is a useful feature, but exacerbates the problem — commands that change the syntax may affect themselves.

In Aleph, commands (both definitions and invocations) are in a language with a fixed syntax, while embedded text follows a different set of rules. Syntax changes for text are well supported.

Programming. Extensibility is a very desirable feature in a batch typesetting system. It should be supported with a full-function programming language.

Extensibility is essential if new document styles are to be written, and in practice, all but the most casual users define shorthands for frequently used text and command sequences. For the latter, a macro language is the natural choice — after all,

[1] We use the words *lexical* and *syntactic* interchangeably, partly because text-processing languages have little of what can be called syntax, but mostly because *lexis*, a candidate counterpart for *syntax*, is not a common computer-science term.

nothing is easier to understand than textual substitution. Indeed, existing systems have preferred macro languages over more procedural ones. On the other hand, a document style is a large program. LaTeX, for example, is 2000 lines of code. Real programs need real programming-language features. TeX, for one, has conditionals and loops, but no real data structures or indeed any support for writing large programs. In addition, macros themselves become unwieldy in large programs. That TeX allows fine control over macro expansion is an indication of its complexity.[2]

Intertwined with issues of linguistic power is the fact that typesetting systems are always *implemented* in one language (a general-purpose programming language) while they *implement* another. (Most complete systems, of course, are written in both.) This practice limits the power of user-written programs — when a primitive to do something does not exist, it cannot be done. The existence in TeX of complex functions as primitives (such as `\halign`) may be an instance of this.

Aleph is a full-function programming language, with data types to represent textual objects and functions to manipulate them. Users at all levels use the same language. There is no barrier between what the user can do and what the system can do.

Aleph and Lisp

Aleph is embedded in Common Lisp. In other words, Aleph is implemented in Lisp as a set of functions, data types, and syntax extensions. An Aleph programmer must use at least as much Lisp as Aleph.

Lisp is an expression language. Every program construct is a value-producing expression called a *form*. A function-call form is surrounded by parentheses: `(f 1 2 3)`. Here, `f` is the name of the called function. It is passed three arguments: 1, 2, and 3. Identifiers like `f` are called *symbols*. In this paper, a symbol can be any sequence of letters and -s. A symbol in the first position of a function-call form is a function name. A form that is a symbol alone is a variable. The form `(f x y z)` calls `f` with the values of variables `x`, `y`, and `z`. Forms can be nested: `(f 1 (g (h 2) 3) 4)`.

Common Lisp also has characters and strings. A string is enclosed in double-quotes: `"in double quotes"`. A character is written with the the prefix `#\`. For example, `#\a` is `a`, `#\%` is `%`, and `#\\` is `\`.

A symbol that begins with a colon, `:`, is a *keyword*. A keyword is an uninterpreted identifier that stands for itself. It is used like the identifiers defined by an enumerated type in C or Pascal.[3]

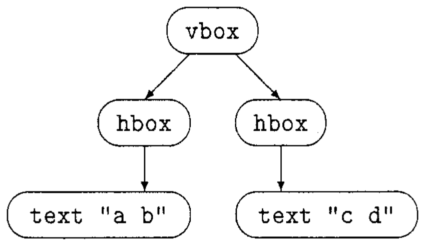
Not all forms in parentheses are function calls. There are built-in and user-defined forms that have special syntax (nevertheless made out of symbols and parentheses), and interpret arguments in special ways. The most visible ones in Aleph are those that begin with `def`.

We now know enough Lisp to understand the Aleph extensions.

A document (or a part of a document) in Aleph is represented by a tree, like nested TeX boxes and lists. For example, the TeX box of boxes made by

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

would have a fairly similar Aleph tree:



Trees are constructed using *tree-building functions* — Lisp functions that create tree nodes. The last example is constructed by the form

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

An Aleph document is just a sequence of such tree-building forms. However, entering a large document with nested forms is rather clumsy. For most forms, there is an equivalent *Aleph string* that is more concise.

An Aleph string (or just *string*, when confusion with Lisp string is unlikely) is enclosed in brackets: `[` and `]`. For example, `[some text]` is equivalent to `(text "some text")`. As in TeX, newlines and tabs in Aleph strings are treated like spaces, and consecutive spaces are treated like

[2] Macros are not inherently less powerful. After all, we know that lambda calculus is turing-complete. TeX's own linguistic problems are also quite complex. They are in part due to the need to delay execution in some situations. In any case, complexity is perhaps not a deadly sin, but the apparent unpredictability that comes with complexity is.

[3] If this is confusing, then just treat keywords as strings — think `"xyzzy"` when you see `:xyzzy`. Keywords have no meaning except in their name and in their use.

one. `[some␣␣text]` is not the same as `(text "some␣␣text")`.

The equivalence between a string and its corresponding form is strict. The string actually *becomes* the form as it is read by Lisp. The rest of the Lisp system never sees Aleph strings.

Since they are equivalent, strings and forms can be mixed freely. We now know enough to enter a simple document:

```
(par [A very short document of
      a single short paragraph
      of a middling sentence.])
```

Just as we can go from Lisp to string, we can go from string to Lisp. The string

```
[an @(it [italicized]) word]
```

has a string nested in a Lisp form that is in a string. It is equivalent to

```
(group (text "an ")
       (it [italicized])
       (text " word"))
```

which is in turn equivalent to

```
(group (text "an ")
       (it (text "italicized"))
       (text " word"))
```

Since this string-Lisp-string double take is so common, we have defined a shorthand for it:

```
[an @it[italicized] string].
```

The escape character, `@`, is very much like \ in TeX. A number of `@`-triggered featured are defined in Aleph, and the user can define more. This and other forms of user control over strings are the subject of the next section.

Mode and Syntax

A *mode* governs the way Aleph strings are turned into tree-building forms. In TeX, the equivalent concept is implicitly defined by the catcodes. Aleph, on the other hand, supports a data type, `mode`, that encapsulates all the information that defines a mode.

For example, to define a mode in which the character `%` expands to the italicized word "*Aleph*," we would write

```
(defsyntax aleph ()
  (#\% (it [Aleph])))
(defmode aleph aleph)
```

The first statement creates a new syntax table, `aleph`, with the character definition. The second statement creates the the new mode, also named `aleph`, that uses the new syntax (named by the second `aleph` on the line). (We often, but not always, use the same name for a mode and its syntax.) The new mode can now be invoked using an escape sequence:

```
[...@$aleph[% is embedded in Lisp]...].
```

We can also give `aleph` a pair of delimiters:

```
(defmode aleph aleph
  :open #\{
  :close #\}),
```

and use them to invoke the mode more concisely:

```
[...{% is embedded in Lisp}...].
```

This is one of the reasons for separating `defmode` and `defsyntax`. A syntax is the character definitions used by a mode. The mode itself uses a syntax, but may also have some supporting attributes.

A syntax can be built on top of an existing syntax (assuming we already have a `verbatim` syntax defined):

```
(defsyntax valeph (verbatim)
  (#\% (it [Aleph])))
(defmode valeph valeph)
```

Thus, `valeph` has the behavior of `verbatim` but also recognizes `%`.

A syntax can be a combination of others. We could have (and indeed should have) defined `valeph` like this:

```
(defsyntax valeph (aleph verbatim))
```

The syntaxes in Aleph form an inheritance hierarchy. Each syntax definition specifies a list of parent syntaxes (multiple inheritance) and some local additions. Looking up the definition of a character in a syntax is a matter of trying, in order and until a definition is found, the local definitions and then the parents (left to right). In each parent, the same process is repeated.

When modes nest (such as in `[...{...}...]`), the lookup is first done in the closest enclosing mode, then repeated in surrounding modes (inside out), until a definition is found. Inside `[...{% is embedded in Lisp}...]`, the definition for `%` is found in mode `aleph`, but the other characters behave as they would outside the braces. This nesting is lexical, even when a string goes in and out of Lisp: `[...{@(it [%...])}...]`.

The full form of `defsyntax` looks like this:

```
(defsyntax ⟨name⟩ (⟨parent⟩...)
  ⟨default-definition⟩
  (⟨chars⟩ ⟨definition⟩)
  ...)
```

⟨*Chars*⟩ is either a single character or a Lisp string representing a set of characters. ⟨*Definition*⟩ is the definition given to the character or characters. Any number of (⟨*chars*⟩ ⟨*definition*⟩) pairs can be specified. Characters not explicitly mentioned receive

⟨*default-definition*⟩, which can be left out, to leave them undefined. So far, we know a character definition can be a Lisp form. It can also be one of several keywords, some of which we will see later. In the most extreme case, a definition can be a Lisp function. We won't use any of these in this paper.

A syntax or mode can be changed: character definitions and parents can be added and deleted; modes can lose or gain delimiters.

The basic Aleph defines these modes:

```
(defsyntax delimiter ())
(defsyntax escape ()
  (#\@ :escape))
(defsyntax standard ()
  (#\Newline :space)
  (#\Space :space)
  (#\Tab :space)
  ... more definitions ...)
(defsyntax default
    (delimiter escape standard))
(defsyntax group ())
(defmode group group
  :open #\[
  :close #\])
```

`Default` is the outer-most syntax of all Aleph strings. `Escape` contains the single character `@`. `Delimiter` contains the delimiters defined with `defmode`. `Standard` is the rest of the definitions for the `default` mode. `Group` defines no characters. It is the syntax for the delimiters `[` and `]`. `Delimiter` is initially empty, but `(defmode group ...)` soon adds two definitions to it.

`Escape`, `delimiter`, and `standard` are separate syntaxes to allow modes to inherit them independently. For example, one may wish to define a mode that behaves like the LaTeX verbatim mode but also recognizes the escape character:

```
(defsyntax weak-verbatim
    (escape verbatim))
```

This approach allows a change to the escape character to be effective everywhere.

The escape character behaves like a mode, but without a fixed closing delimiter. The `dispatch` syntax controls escape-sequence processing. These escape sequences are supported:

`@(...)`
: This is the escape into Lisp we have seen. The Lisp form should be a tree-building form.

`@`⟨*symbol*⟩
: This is equivalent to `@(`⟨*symbol*⟩`)`. ⟨*Symbol*⟩ must be a reasonable-looking Lisp symbol (made out of letters and -s).

`@`⟨*symbol*⟩⟨*delimited-string*⟩...
: If the `@`⟨*symbol*⟩ sequence is followed immediately by an opening delimiter (defined in syntax `delimiter`), then the delimited string becomes the argument of ⟨*symbol*⟩:

  `@(`⟨*symbol*⟩ ⟨*delimited-string*⟩`)`

  ⟨*Delimited-string*⟩ can be repeated any number of times. For example, `@f[Aleph][Beth]` is the same as `@(f [Aleph] [Beth])`.

`@$`⟨*symbol*⟩⟨*open-delim*⟩⟨*text*⟩⟨*close-delim*⟩
: Enter mode ⟨*symbol*⟩ for the duration of ⟨*text*⟩. ⟨*Text*⟩ can contain any character other than ⟨*close-delim*⟩. ⟨*Open-delim*⟩ is any character. ⟨*Close-delim*⟩ is `)`, `]`, `}`, or `>`, if ⟨*open-delim*⟩ is `(`, `[`, `{`, or `<`, respectively. Otherwise, ⟨*close-delim*⟩ equals ⟨*open-delim*⟩. This is how modes without delimiters are invoked.

`@;`
: The rest of the line, including the end-of-line character, is ignored.

`@`⟨*accent*⟩
: A number of accents are defined in Aleph.

`@\`⟨*char*⟩
: The character ⟨*char*⟩.

`@`⟨*char*⟩
: This is equivalent to `@\`⟨*char*⟩, if ⟨*char*⟩ has no defined behavior (one of the above).

Flexibility: mechanism and policy. The user of a mode is not necessarily the writer of the mode. This is particularly true when canned Aleph code from a library is used. LaTeX, for example, has such a library. When a mode or a syntax is to be reused, the programmer must anticipate the possible uses and choose the implementation accordingly. To do this requires some skill, but also a flexible syntax mechanism.

For example, the mode `aleph`, though frivolous, belongs to a common class of user-defined modes. It defines only a few characters, so must be used in conjunction with another mode (if nothing else, with `default`). We expect such a mode to be used in several different ways, depending on the user's needs:

- Enter mode when necessary, using delimiters or `@$`.
- Use everywhere, by making it a parent of `default`. An Aleph function is provided to do this.
- Combine with other modes to make new ones (like `valeph`).

As we have seen, the definition of `aleph` does allow this freedom.

Consistency. A mode like LaTeX's `verb` is very easy to define in Aleph:

```
(defsyntax verb () :char
  ;; make newline act like a space too
  (#\Newline (text " ")))
(defmode verb verb)
```

All characters are given the definition `:char` (meaning just the character itself). Using `verb` looks like its LaTeX counterpart: `@$verb|...|`.

It is simple to define `verb` in Aleph—we do not have to write catcode-changing macros. Its other advantage is the consistency of behavior. Wherever `@` is recognized, `@$verb|...|` can be used. Unlike in LaTeX, there are no unpleasant surprises depending on context or content.

Trees

As mentioned, an Aleph program constructs a tree that represents the document internally. Nodes in the tree have a *type* that indicates what object each node stands for. The type is named by a Lisp keyword. For instance, a node of type `:box` represents a box, and its children the content of the box; a node of type `:par` represents a paragraph, and its children text or other material that needs to undergo line breaking. We have given examples of how to construct such trees in earlier sections.

A tree fully specifies a document fragment, but requires some processing before it can be used for output. Aleph performs such processing in a *traversal* pass.

Aleph provides a number of primitive node types. One can also define new types in the following way:

```
(defnode ⟨type⟩
  :constructor ⟨c-function⟩
  :traverse-function ⟨t-function⟩
  :output-function ⟨o-function⟩)
```

Here ⟨*type*⟩ is an arbitrary keyword denoting the node type. ⟨*C-function*⟩ is a function that constructs a node of that type. If one is not supplied, a standard constructor is provided, with a name equal to the node type (without the colon). ⟨*T-function*⟩ is the traversal function for nodes of this type. ⟨*O-function*⟩ is called during a similar traversal to output the document.

Each instance of a node has an associated set of named values called *attributes*. Attribute names are also Lisp keywords. For instance, a `:box` node has a `:direction` attribute indicating if its components should be stacked horizontally or vertically; a `:par` node has a `:width` attribute, whose numeric value selects the width to be used for line breaking.

A few attributes are assigned at node construction time. Other attributes represent printing information, such as the final position and size of the formatted object. These attributes are filled in by the tree traversal. This starts at the root of the tree and proceeds by calling the traversal function of each node it visits. Besides computing attributes, traversal functions are also allowed to modify the tree locally.

To clarify these concepts, we introduce a simple example. We add the node type `:f-box`. This node has a single child representing some printable object. If the width of the object is less than 1 inch, it is printed centered in a 1-inch horizontal space; otherwise three dollar signs are printed.[4]

```
(defnode :f-box
  :traverse-function #'trav-f-box)
```

The traversal function for `:f-box` is `trav-f-box`; its output function is the default output function, which just outputs the node's children.

```
(defun trav-f-box (n)
  ;; First visit the (only) child
  ;; of this node.
  (traverse (child n))
  ;; Then destructively modify this node.
  ;; Change its type:
  (setf (type n) :box)
  ;; Specify the width:
  (setf (attr :width n) !1inch)
  ;; Change its child:
  (setf
   (child n)
   ;; Use a centering construct
   (center
    (if (< (attr :width (child n))
           !1inch)
        ;; and inside it, put
        ;; either the old child
        (child n)
        ;; or three dollar signs.
        [$$$])))
  ;; Traverse the modified node
  ;; to set the glue.
  (traverse (child n)))
```

This example contains a few unfamiliar but quite simple Lisp and Aleph constructs:

- the `defun` form defines a Lisp function named `trav-f-box`, that takes the single argument `n` and operates on it;

[4] The letter *f* in `f-box` stands for FORTRAN.

- the Aleph form (`child` *x*) refers to the value of the single child of *x*, and the form (`attr` *name* *x*) refers to the attribute *name* of node *x*;
- `setf` is the Lisp assignment operator. (`setf` *place value*) replaces the old value of *place* with *value*. So (`setf (child n) ...`) replaces the child of `n`;
- !⟨*number*⟩⟨*unit*⟩ is Aleph's way of specifying a length;
- `center` is an Aleph function that returns a group with appropriate glue for centering.

This example reveals that our typesetting primitives are very similar to those of TEX. In fact, we think that most of TEX's primitives are well designed and we are not attempting to improve on them.

One should define new node types with their own traversal functions only when direct access to the typesetting engine is needed. We expect style writers to be able to do most of their programming at the level of mode definition and tree construction. The system programmer (us) should provide enough node types to satisfy the most common needs.

Current Status and Future Directions

As we are submitting this paper, the implementation of Aleph contains the described syntax mechanisms and intermediate representation. We have also defined a small number of node types, most notably paragraphs, boxes, and glue. The output routines produce plain TEX. TEX is also used in interactive mode to perform some computations currently not implemented in Aleph, such as finding the widths of objects in our table constructor. The Aleph process communicates with the TEX process through Lisp streams connected to a UNIX socket pair.

Aleph relies on TEX for ligatures, line breaking, math, and output. As a consequence, we expect the exact semantics of traversal and retraversal to evolve, as more is demanded of them. Also, it is at present difficult to estimate the system's efficiency, though we believe the tree-and-traversal model is not fundamentally inefficient.

Of the missing features, ligature and math are perhaps the hardest for our model. We plan to tackle them first. Unrelated to TEX, we are also considering ways to extend the syntax mechanism to recognize multicharacter sequences.

Aside from completing this implementation and refining it into a practical tool, our work suggests many other research directions. For instance, to what extent is Aleph's intermediate representation suitable for a WYSIWYG-style document editing, with incremental processing? And if it is, would it simplify the task of integrating programmatic and WYSIWYG interfaces? We have not tried to answer these questions, but we hope that our work, by allowing one to look at an old problem in a new way, will provide both a stimulus and a vehicle for further research.

Acknowledgments

We thank Ethan Munson for his useful suggestions. Luigi Semenzato did part of this work at the Dipartimento di Informatica, Università di Padova, Italy.

Bibliography

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

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

Appendix

Table example

This is an example of use of our table primitive, with the resulting output. The `table` constructor is a Lisp macro. Macros are a powerful feature of Lisp that we cannot attempt to explain here. In this context, just think of a macro as a function with a more flexible argument-passing mechanism.

```
;;; We thank Marcia Feitel for correcting an important omission.

(line
 (center (bf (bind :size 12 [From page 236 of the TeXbook, more or less]))))

(vskip !0.5in)

(line
(center
(table
  ;; Half of the padding goes before the column, half after the column.
  :pad !0.5cm
  ;; The vertical padding goes between rows.
  :vpad !2pt
  ;; The template is a list of column descriptors.
  ;; Each descriptor is a function, or a list of functions,
  ;; called in turn with each corresponding entry in a row
  ;; as argument.
  :template ((right bf) (center it) center center left)
  :rows
  ;; These are the rows.  Each row is a list of entries.
  (((sl [American]) (sl [French]) (sl [Age]) (sl [Weight]) (sl [Cooking]))
   ((sl [Chicken]) (sl [Connection]) (sl [(months)]) (sl [lbs.])
    (sl [Methods]))

   ;; A special row that spans all columns.
   (:span-all (left (vbox [] !0.1in)))

   ;; $ is the Aleph delimiter for the tex-math mode, an escape into TEX's math mode.
   ([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])

   ))))
```

From page 236 of the TeXbook, more or less

American Chicken	*French Connection*	*Age (months)*	*Weight lbs.*	*Cooking Methods*
Egg	*Oeuf*	$-\frac{2}{3}$	$\frac{1}{6}$	Boil, Fry, Poach, Raw
Squab	*Poussin*	2	$\frac{3}{4}$ to 1	Broil, Grill, Roast
Broiler	*Poulet Nouveau*	2 to 3	$1\frac{1}{2}$ to $2\frac{1}{2}$	Broil, Grill, Roast
Fryer	*Poulet Reine*	3 to 5	2 to 3	Fry, Sauté, Roast
Roaster	*Poularde*	$5\frac{1}{2}$ to 9	Over 3	Roast, Poach, Fricassee
Fowl	*Poule de l'Année*	10 to 12	Over 3	Stew, Fricassee
Rooster	*Coq*	Over 12	Over 3	Soup stock, Forcemeat