Towards an Interchange Standard for Editable Documents

by Jim Mitchell and Jim Horning

Version 1.3/August 31, 1982

The **Interscript** standard will define a digital representation of editable documents for exchange among different editing systems. A *script* is the representation of a document in the Interscript format; it can be transmitted from one editor to another over a network, or can be stored for later editing. A script is not limited to any particular editor: if a script contains editable information some of which is not understandable by a particular editor, it is still possible to edit the parts of the document understood by that editor without losing or invalidating the parts it does not understand.

This draft is a proposal for the technical content of the Interscript standard. It defines and explains the proposed standard, gives examples of its use, explains how to *externalize* documents from an editor's private format as scripts, and how to *internalize* scripts into an editor's private format. It also indicates a number of issues that must still be resolved to establish a practical standard.

The standard provides for documents with

> a dominant hierarchical structure (e.g., book/chapter/section/paragraph...) while also providing for documents needing more general structure than a single tree (e.g., for graphics, for certain kinds of document formatting, or for cross-references in a textual document),
>
> formatting information (e.g., margins, fonts, line widths, etc.),
>
> definitional structure (such as styles or property sheets), and
>
> intermixed kinds of editable information (e.g., text with imbedded graphics).

This draft deals primarily with the contents of Layers 0 and 1 (the base language) of the proposed standard.

Contents

1. Introduction

Interscript provides a means of representing editable documents. This representation is independent of any particular editor and can therefore be used to interchange documents among editors.

The basis of Interscript is a language for expressing editable documents as *scripts*. Scripts are created by computer programs (usually an editor or associated program); scripts are "compiled" by programs to produce whatever private format a particular editor uses to represent documents.

1.1. Rationale for an interchange standard

As office systems proliferate, being able to interchange documents among different editing systems is becoming more and more important. Customers need document compatibility to avoid being trapped in evolutionary cul-de-sacs and having to pay the awful price of converting documents from one product's format to another's (even within one company's product line sometimes).

Now, an editing program typically uses a private, highly-encoded representation for documents in order to meet goals of performance and functionality. Generally, this means that different editors use different, incompatible private formats, and the user can conveniently edit a document only with the editor used to create it. This problem can be solved by providing programs to convert between one editor's private (or file) format and another's. However, a set of different editors with N different document representations requires N(N-1) conversion routines to be able to convert directly from each format to every other.

This N(N-1) problem can be reduced to 2(N-1) by noticing that we could write N-1 conversion routines to go from F_1 (format for editor$_1$) to F_2,. . .,F_N, and another N-1 routines to convert from F_2,. . .,F_N to F_1. Except when converting from or to F_1, this scheme requires two conversions to go from F_i to F_j (j=i); this is a minor drawback. Choosing which editor should be editor$_1$ is a more critical issue, however, since the capabilities of that editor will determine how general a class of documents can be interchanged among the editors.

This presents a truly difficult problem in the case that there is no single functionally dominant editor. If the pivotal editor$_1$ doesn't incorporate all of the structures, formats, and content types used by all of the others, then it will not be possible to faithfully convert documents containing them. Even if we had a single editor that was functionally dominant, it would place an upper bound on the functionality of all future compatible editors. Since there are no actual candidates for a totally dominant editor, we have chosen instead to examine in general what information editors need and how that information can be organized to represent general documents.

Since we are not proposing an editor, we do not need to design a private format for its documents; we only need an external representation that is capable of conveying the content, form, and structure of editable documents. That external representation has only *one* purpose: to enable the interchange of documents among different editors. It must be easy to convert between real editors' formats and this *interchange encoding*.

Using a standard interchange encoding has the additional advantage that much of the input and output conversion algorithms will be common to all conforming editors. For example, when a new version of an existing editor is released, the only differences in the new version's conversion routines will be in the areas in which its internal document format has changed from its previous form; this represents a significant saving of programming.

1.2. Properties that any interchange standard must have

An interchange encoding for editable documents must satisfy a number of constraints. Among these are the following:

1.2.1. Universal character set

Scripts must be encoded using the graphic (printable) subset of the ISO 646 printing character set. As well as the obvious rationale that these characters are guaranteed not to have control significance to any devices meeting the ISO standard, it has the additional advantage that a script is humanly readable.

1.2.2. Encoding efficiency

Since editable documents may be stored as scripts, may be transmitted over a network, and must certainly be processed to convert them to various editors' private formats, it is important that the encoding be reasonably space-efficient.

Similarly, the time cost of converting between interchange encoding and private formats must be reasonably low, since it will have a significant effect on how useful the interchange standard is. (If the overheads were small enough, an editor might not even use a private file format for document storage.)

1.2.3. Open-ended representation

Scripts must be capable of describing virtually all editable documents, including those containing formatted text, synthetic graphics, scanned images, etc., and mixtures of these various modes. Nor may the standard foreclose future options for documents that exploit additional media (e.g., audio) or require rich structures (e.g., VLSI circuit diagrams, database views). For the same reasons, the standard must not be tied to particular hardware or to a file format: documents will be stored and transmitted using a variety of media; it would be folly to tie the representation to any particular medium.

1.2.4. Document content and form

The complete description of a document component usually requires more than an enumeration of its explicit contents; e.g., paragraphs have margins, leading between lines, default fonts, etc. Scripts must record the association between attributes (e.g., margins) and pieces of content.

Both the contents and attributes of typical documents require a rich value space containing scalar numbers, strings, vectors, and record-like constructs in order to describe items as varied as distances, text, coefficients of curves, graphical constraints, digital audio, scanned images, transistors, etc.

1.2.5. *Document structure*

Many documents have hierarchical structure; e.g., a book is made of chapters containing sections, each of which is a sequence of paragraphs; a figure is embedded in a frame on a page and in turn contains a textual caption and imbedded graphics; and the description of an integrated circuit has levels corresponding to modular or repeated subcircuits. The standard should exploit such structure, without imposing any particular hierarchy on all documents.

Hierarchy is not sufficient, however. Parts of documents must often be related in other ways; e.g., graphics components must often be related geometrically, which may defy hierarchical structuring, and it must be possible to indicate a reference from some part of a document to a figure, footnote, or section in way a that cuts across the dominant hierarchy of the document (section 1.6.4).

Documents often contain structure in the form of indirection. For instance, a set of paragraphs may all have a common "style," which must be referred to indirectly so that changing the style alone is sufficient to change the characteristics of all the paragraphs using it. Or a document may be incorporated "by reference" as a part of more than one document and may need to "inherit" many of its properties from the document into which it is being incorporated at a given time.

1.2.6. *Transcription fidelity*

It must be possible to convert any document from any editor’s private format to a script and reconvert it back to the same editor’s private format with no observable effect on the document’s content, form, or structure. This characteristic is called *transcription fidelity*, and is a *sine qua non* for an interchange encoding; if it is not possible to accomplish this, the interchange encoding or the conversion routines (or both) must be defective.

1.2.7. *Script comprehension*

Even complicated documents have simple pieces. A simple editor should be able to display parts of documents that it is capable of displaying, even in the presence of parts that it cannot. More precisely, an editor must, in the course of *internalizing* a script (converting it from a script to its private, editable format), be able to discover all the information necessary to recognize and to display the parts that it understands. This must work despite the fact that different editors may well use different data structures to represent the content, form, and structure of a document.

At a minimum, this requires that a script contain information by which an editor can easily determine whether or not it understands a component well enough to display or edit it, and that it be able to interpret the effect that components which it does not understand have on the ones it does. For example, if an editor does not understand figures, it should still be possible for it to display their embedded textual captions correctly, even though a figure might well dictate some of its caption's content or attributes such as margins, font, etc.

This constraint requires that an interchange encoding must have a simple syntax and semantics that can be interpreted readily, even by low-capability editors. Along with the desire for openendedness (section 1.2.3), this suggests a language with some form of "extension by definition" built around a small core.

1.2.8. *Regeneration*

Processing a script to internalize it correctly is only half the problem. It is equally important that an editor, in *externalizing* a script from its private document format be able to *regenerate* the content, form, and structure carried by the script from which the document originally came. In particular, when regenerating a script from an edited document, it should be possible to retain the structure in parts of the original script that were not affected by editing operations. For example, an editor that understands text but not figures should be able to edit the text in a document (although editing a caption may be unsafe without understanding figures) while faithfully retaining and then regenerating the figures when externalizing it.

This problem is much less severe when an editor is transcribing a document that it "understands" completely, e.g., because the entire document was generated using that editor.

1.3. **What the Interscript standard does *not* do**

There are a number of issues that the Interscript standard specifically does not discuss. Each of these issues is important in its own right, but is separable from the design of an interchange representation

1.3.1. *Interscript is not a file format*

The *interchange encoding* of a script is a sequence of ASCII/ISO 646 characters. The standard is not concerned with how that representation is held in files on various media (floppy disks, hard disks, tapes, etc.), or with how it is transmitted over communications media (Ethernet, telephone lines, etc.).

1.3.2. *Interscript is not a standard for editing*

A script is not intended as a directly editable representation. It is not part of its function to make editing of various constructs easier, more efficient, or more compact: those are the purview of editors and their associated private document formats. A script is intended to be internalized before being edited. This might be done by the editor, by a utility program on the editing workstation, or by a completely separate service.

1.3.3. Combining documents is not an interchange function

This exclusion is really a corollary of the statement, "A script is not intended as a directly editable representation." In general, it is no easier to "glue" two arbitrary documents together than it is to edit them.

1.3.4. Interscript does not overlap with other standards

There are a number of standards issues that are closely related to the representation of editable documents, but which are not part of the Interscript standard because they are also closely related to other standards. For example, the issues of specifying encodings for characters in documents, how fonts should be named or described, or how the printing of documents should be specified (i.e., Interpress) are not part of this work.

1.4. Concepts and Guiding Principles

1.4.1. Layers

The Interscript standard is presented in layers:

Layer 0 defines the syntax of scripts; parsing reveals the dominant structure of the documents they represent.

Layer 1 defines the semantics of the base language, particularly the treatment of bindings and environments.

Layer 2 defines the semantics of properties and attributes that are expected to have a uniform interpretation across all editors.

Various Layer 3 extensions will define the semantics of properties and attributes that are expected to be shared by particular groups of editors.

The present document focusses almost exclusively on Layers 0 and 1, although some of the examples illustrate properties and attributes likely to be defined in Layer 2.

1.4.2. Externalization and Internalization

Transcription fidelity requires that any document prepared by any editor can be externalized as a script that will then be internalized by the editor without loss of information. Ease of internalization requires that the Interscript base language contain only relatively few (and simple) constructs. We resolve this apparent paradox by including within the base language a simple, yet powerful, mechanism for abbreviation and extension.

A script may be considered to be a "program" that could be "compiled" to convert the document to the private representation of a particular editor, ready for further editing. The Interscript language has been designed so that internalizing scripts into typical editors' representations can be performed in a single pass over the script by maintaining a few simple data structures.

1.4.3. Content, Form, Value, and Structure

Most editors deal with both the *content* of a document (or piece of a document), and its *form*. The former is thought of as "what" is in the document, the latter as "how" it is to be viewed; e.g., "ABC" has a sequence of character codes as its contents; its format may include font and position information. Interscript maintains this distinction.

The distinction between the *value* and the *structure* of both content and form within a document is also important. When viewing a document, only the value is of concern, but the structure that leads to that value may be essential to convenient editing. An example of structure in content is the grouping of text into paragraphs; in form, associating a named "style" with a paragraph.

Content: Text and graphics are common special cases. Interscript's treatment of these has been largely modelled on that of Interpress. Other kinds of content may be represented by structures built from character strings, numbers, Booleans, and identifiers.

Form: Interscript provides for open-ended sets of *properties* and *attributes*. Properties are associated with content by means of *tags*. Attributes are *bindings* between names and values that apply over some *scope* (sections 1.4.4.2 3). The way the contents of a document are to be "understood" is determined by its properties; Interscript makes it straightforward to determine what these properties are without having to understand them.

Structure: Most editors structure the content of a document somehow into words, sentences, paragraphs, sections, chapters; or lines, pages, signatures, for example. This assists in obtaining private efficiency, but, more importantly, provides a conceptual structure for the user.

Full transcription fidelity requires that the Interscript language be adequate to record any structure that is maintained by any editor for either form or content. Of course, some editors provide a number of different structures. A general structure, of which all the editors we know use special cases, is the labelled directed graph. Interscript provides this structure, without restricting the purposes for which it may be used. There are also two specializations of general graphs that occur so frequently that Interscript treats them specially:

> Sequences: The most important, and most frequent, relationship between values is logical adjacency (sequentiality), which is represented by simply putting them one after another in the script.
>
> Ordered trees: Most editors that structure contents have a "dominant" hierarchy that maps well into trees whose arcs are implicitly labelled by order. (Different editors use these trees to represent different hierarchies). Interscript provides a simple linear notation for such trees, delimiting *node* values by braces ("{" and "}"). If an editor maintains multiple hierarchies, the *dominant* one is the one transcribed into the tree structure and used to control the inheritance of attributes.

Structure for content beyond that contained in the dominant hierarchy is represented by explicit *links* in the script; any node may be labelled as the *source* and/or the *target* of any number of links. A link whose target is a single node uniquely identifies that node; links with multiple targets may be used to represent sets of nodes.

Typical structures recorded for form are expressions (indicating intended relations among attribute values) and sharing (representable by indirection). Interscript allows expressions to be composed of literals, identifiers, operators, and function applications, and permits the use of identifiers to represent expressions.

1.4.4. Features of the Base Language

1.4.4.1 Values

Expressions in a script may denote

- Literal values of primitive types
    - Booleans: F, T
    - Integers: . . . 3, 2, 1, 0, 1, 2, 3, . . .
    - Reals: 1.2E5, . . .
    - Strings: <this is a string>
    - Universal names: TEXT, XEROX, PARAGRAPH
- Structured values
    - Nodes
    - Vectors of values
    - Environments
- Generic operations
    - Invocations
    - Applications
    - Selections
- Operations specific to particular types
    - Arithmetic
    - Comparison
    - Logical
    - Subscript
    - . . .
- Bindings
- Labels
    - Tags
    - Targets
    - Sources
    - Link introductions
- Expressions to be evaluated at the point of invocation

1.4.4.2 Environments and Attributes

Environments bind attribute identifiers to values (or expressions denoting values), in various modes:

"_" denotes a local binding, which may be freely superseded,

":=" denotes a global binding, which creates or modifies an attribute in the outermost environment.

NULL denotes the "empty" environment, containing bindings for no attributes. The (implicit) outermost environment binds each identifier id to the corresponding *universal* name ID (written with all capital letters).

Each piece of content in a document has its own environment. Editors will use relevant attributes from that environment to control its form.

Attributes may also be used in scripts for two structuring purposes:

abbreviation: an identifier may be bound to a *quoted expression*; within the scope of the binding, the use of the identifier is equivalent to the use of the full expression;

indirection: reference *through* an identifier permits information (such as styles) to be defined in one place and shared throughout its scope; this is an example of structure (which must be preserved) in the form of a document.

1.4.4.3 Inheritance

The dominant hierarchy of a document is represented by grouping its pieces within nodes, which are the most obvious form of content structuring. They also control the scope of bindings.

The environment of a node is initially inherited from its containing node (except for the outermost node, which inherits it from the editor), and may be modified by bindings. A binding takes effect at the point where it appears, and its scope extends to the end of the innermost node containing it, with two exceptions:

any binding except a definition may be superseded by a (textually) later binding (if the later binding is in a nested node, the outer binding’s scope will resume at the end of the inner node), and

a global binding extends over the all of the document lexically to the right of the binding.

Attributes are inherited only via environments following the dominant structure. Thus the choice of a dominant structure to represent scripts from a particular editor will be strongly influenced by expectations about inheritance.

Attributes are "relevant" to a node if they are assumed by any of its tags. In general, a node’s environment will also contain bindings for many "latent" attributes that are either relevant to its ancestors (and inherited by default) or are potentially relevant to its descendants.

The interior of each node is implicitly prefixed by *Sub*, which will generally be bound in the containing environment to a quoted expression performing some bindings, applying some labels, and/or supplying some initial content.

1.4.4.4 Expressions

Expressions involving the four infix operators (+, , *, /) are evaluated right-to-left (a la APL); since we expect expressions to be short, we have not imposed precedence rules.

Parentheses are used to delimit vector values. Square brackets are used to delimit the argument list of an operator application and to denote environment constructors, which behave much like records.

The notation for selections (conditionals) follows Algol 68:

(<test> **|** <true part> **|** <false part>)

This is consistent with our principles of using balanced brackets for compound constructions and avoiding syntactically reserved words; the true part and false part may each contain an arbitrary number of items (including none).

1.4.4.5 Tags and Links

A tag is written as a universal name followed by $''. A tag, `U,` labels a node that contains it with its associated properties and also invokes the component of the outermost environment X with the name `U`. Tags are either present in a node or absent, whereas attributes have values that apply throughout a scope.

Layer 2 of the standard will be primarily concerned with the definition of a (small) set of standard properties that are expected to be shared among all conforming editors. For each standard property, it will describe

the associated tag that denotes it,

the assumptions it implies about the contents (values that must/may be present and their intended intepretation, invariant relations that are to be maintained, etc.),

the assumptions it makes about the environment (attributes that must be present and their intended intepretation).

Links enable a script to model associations that cut across its dominant structure: a link set denotes a set of directed arcs from each of its *source* nodes to all its *target* nodes. There are several ways this facility can be used:

(ST) A link set with a single source node and a single target node models a simple reference from one node in a document to another.

(S*T) For a link set with a single target node and multiple source nodes, each source node can be viewed as "pointing to" that target node.

(ST*) The symmetrical extreme case of a single source node and multiple target nodes corresponds closely to an entry in an index, which refers to all the places where some term is used (section 1.6 contains an example).

(S*T*) Finally, multiple source and target nodes in a link set can be used for all the cross references within a document of the form "see sections 1.6, 1.7, 2.3".

To use links, a script must declare the "main" identifier of a *link set* ("LINKS" *id*) at the root of a subtree containing all its sources and targets, and textually preceding them. Once this main identifier has been introduced, nodes can be labelled as sources for subsets of this linkset. For example, the *label* "*id*.a.b:" would make a node a target for source nodes containing references of the sort "^*id*", "^*id*.a", or "^*id*.a.b".

1.4.5. Script comprehension

The Interscript standard applies to interchange among editors with widely varying capabilities. It will be important to define some structure to the space of possible scripts, just as Interpress has for printable documents. Dimensions in which we foresee reasonable variations in script comprehension are:

Abbreviations: only editor-supplied   defined in document.

Dominant structure: single-layer   arbitrary.

Other structure: no links or indirections   links and indirections preserved.

Bindings: Local only and global (:=).

Selection: No conditionals   conditionals.

Numbers: Integers only   floating point.

See section 2.4 for further details.

1.4.6. Internalizing a Script

The private representations of low-capability editors are not generally adequate to provide a full-fidelity internalization of every script produced by a high-capability editor. Thus, when internalizing a script, some information may not be viewable or editable. The Interscript language has been designed to simplify value-faithful internalization, even if structure is lost, and content-faithful internalization, even if form is lost or the conversion of form to additional content to allow it to be examined (and perhaps even edited) by a low capability-editor. The standard provides some simple conditions under which a low-capability editor can safely modify parts of a document that it understands fully, without thereby destroying the value or structure of parts that it is not prepared to deal with.

A script may be internalized into an editor’s (private or file) representation as follows:

Parse the entire script from left to right.

As each literal is encountered in the script, convert it to the editor’s representation.

As each abbreviation (free-standing invocation) is encountered in the script, replace it with the value to which it is bound in the environment.

As each structure is recognized in the script, represent the corresponding structure in the editor’s representation, if possible; if not, use the semantics of Interscript to compute the value to be internalized.

Update the environment whenever a binding is encountered or a scope is exited, according to the semantics of Interscript.

Transfer the values of all attributes relevant to each piece of content from the current environment to the editor’s representation, if possible; if not, apply an invertible function to convert the attribute-value binding into additional content.

Determine the properties of each node from its tags; this list will be complete at the end of the node. A node is *viewable* if any of its tags denotes a property in the set of those the editor is prepared to display; it is *understood* if they are all in the set of those the editor is prepared to edit.

Record the sources and targets of all links; for any link, these lists will be complete at the end of the node in which its main identifier was introduced. Translate each link to the corresponding editor structure, according to the properties of the node that introduces it.

Of course, any process yielding an equivalent result is equally acceptable.

1.5. Introduction to the Interscript Base Language

This section is intended to lead the reader through a set of examples, to show what the language looks like and how it is used to represent a number of commonly occurring features of editable documents. The examples purposely use rather long identifiers and lots of white space to make them more readable. In actual use, programs, not people, will generate and read scripts; names will tend to be short; and logically unneeded spaces and carriage returns will tend to be omitted.

1.5.1. Simple text as a document

The following script defines a document consisting of the string "The text of the main node of example 1.5.1"; no font, paragraph structure, or formatting information is supplied. This example will gradually be expanded to represent accurately figure 1.5.1, below. The numbers at the left margin do not form part of the script; they are used to refer to the various lines in the discussion below.

```
0   Interscript/Interchange/1.0
1   {<The text of the main node of example 1.5.1>}
2   EndScript
```

Line 0 is the header denoting version 1.0 of the interchange encoding. Line 1 is the entire body of this script: it contains a single *node* enclosed in {} which in turn contains a single string value enclosed in <>. Line 2, with the keyword "EndScript" marks the end of script.

The text of the main node of
example 1.5.1
          The text of the *first* subnode of example 1.5.1

Example 1.5.1: A simple document

The next version of the example adds the *tag*, TEXT$ to the node. The identifier TEXT is called a *universal name* (or atom), which is indicated by its being composed of all uppercase letters. Universal names have no definition within the base language (they are expected to be defined in Layers 2 and 3).

```
0   Interscript/Interchange/1.0
1   {TEXT$
2   <The text of the main node of example 1.5.1>
3   }
4   EndScript
```

A tag is denoted by placing "$" after a universal name. A node’s tags are strictly local (they are not inherited by other nodes in the script) and serve as "type information" about the node. The tag TEXT$ labels this node as one that can be viewed as textual data. Tags can also create implicit indirections; see section 1.6.5.

```
0   Interscript/Interchange/1.0
1   {PARAGRAPH$
2   leftMargin_3.25*inch rightMargin_5.0*inch
3   <The text of the main node of example 1.5.1>
4   }
5   EndScript
```

This example shows how auxiliary information, such as margins, may be associated with a node of a script. The *binding* leftMargin_3.25*inch adds the attribute leftMargin to the node's environment and binds the value of the expression 3.25*inch to it (inch is a value whose dimensions are inches/meters; meters are the standard Interscript units of distance). The bindings to leftMargin and rightMargin convey the fact that this node has margins for display. To denote the change in character of the node, we have tagged it as PARAGRAPH instead of TEXT. Figure 1.5.1 uses these margins for its first line of text.

```
0   Interscript/Interchange/1.0
1   {PARAGRAPH$
2   leftMargin_3.25*inch rightMargin_5.0*inch
3   <The text of the main node of example 1.5.1>
4       {PARAGRAPH$ leftMargin_+0.5*inch
5       <The text of the first subnode of example 1.5.1>
6       }
7   }
8   EndScript
```

We have further elaborated the example by nesting another text node in the primary one, with its text following the primary node's text and with an indented leftMargin. The binding leftMargin_+0.5*inch is a contraction of leftMargin_leftMargin+0.5*inch. The right side of the binding is evaluated, and since there is as yet no binding in the inner node's (lines 4 6) environment for leftMargin, it is looked up in the environment of the containing node (lines 1 3). The value of the right hand side expression is thus 3.75*inch. This value is then bound to the identifier leftMargin in the inner node's environment. Since no value is bound to rightMargin in the inner node's environment, it will have the same rightMargin as its parent node.

```
0   Interscript/Interchange/1.0
1   p _ 'PARAGRAPH$ leftMargin_3.25*inch rightMargin_6.0*inch'
2   {p rightMargin_5.0*inch
3   <The text of the main node of example 1.5.1>
4       {p leftMargin_+0.5*inch
5       <The text of the first subnode of example 1.5.1>
6       }
7   }
8   EndScript
```

One can also define an *abbreviation* by binding a sequence of unevaluated expressions to an identifier and subsequently using the identifier to cause those expressions to be evaluated at the point of invocation. This example binds the *quoted expression* 'PARAGRAPH$ leftMargin_3.25*inch rightMargin_6.0*inch' to the identifier p. When p is invoked in lines 2 and 4, the quoted expression replaces the invocation and is evaluated there.

Invoking p places the tag PARAGRAPH$ on the node, sets the leftMargin to 3.25*inch and the rightMargin to 6.0*inch. In line 2, the rightMargin is then rebound to 5.0*inch, overriding the default binding created by invoking p. Similarly, the binding for leftMargin in line 4 overrides the one resulting from invoking p, resulting in its leftMargin being 3.75*inch and its rightMargin being 6.0*inch.

An identifier can also be bound to an *environment value* as a convenient record-like manner of naming a set of related bindings. For example, a font might be defined as follows (a more complete definition is given later in section 1.6.3):

```
font _ [ | family_TIMES size_10*pt face_[ | weight_NORMAL style_ROMAN slant_NIL] ]
```

This defines font to be the environment formed by taking the empty or NULL environment and altering it according to the series of bindings following the initial "[|." In this case font is an environment having bindings for three attributes, family, size, and face. face is itself bound to an environment (with attributes weight, style, and slant). The set of default bindings in font specify a normal weight (non-bold), non-italic Times Roman 10-point font.

We can incorporate this font definition in the example and then use it to indicate that the word "first" in the subnode should be in italics:

```
0    Interscript/Interchange/1.0
1    p _ 'PARAGRAPH$ leftMargin_3.25*inch rightMargin_6.0*inch'
2    font _ [ | family_Times size_10*pt face_[ | weight_NORMAL style_ROMAN slant_NIL] ]
3    {p rightMargin_5.0*inch
4    <The text of the main node of example 1.5.1>
5         {p leftMargin_+.5*inch
6         <The text of the >
7         font.face.slant_ITALIC <first> font.face.slant_NIL
8         < subnode of example 1.5.1>
9         }
10   }
11   EndScript
```

Bindings affect node contents to their right: so, "first" will be italic, while " subnode of example 1.5.1" will be non-italic due to the binding immediately preceding it. If we expected to switch between italics and non-italics frequently, it might be profitable to introduce abbreviations to shorten what must appear. For example, in the scope of the definition

```
l _ [ | i _ 'font.face.slant_ITALIC'  nI _ 'font.face.slant_NIL']
```

line 7 could be abbreviated

```
l.i<first>l.nI
```

History Log

Edited by Mitchell, September 1, 1981 3:12 PM, added first version of glossary

Edited by Mitchell, September 7, 1981 2:11 PM, wrote parts of introduction

Edited by Mitchell, September 10, 1981 10:14 AM, added Tab def to Star property sheets

Edited by Mitchell, September 14, 1981 9:54 AM, renumbered chapters and did minor edits

Edited by Mitchell, September 16, 1981 8:42 AM, folding in comments from JJH's review and added sections on rendition and transcription fidelity

Edited by Mitchell, September 18, 1981 1:56 PM, folded in comments from JJH's review

Edited by Horning, May 3, 1982 6:02 PM, Folded in comments from Truth copy

Edited by Mitchell, May 10, 1982 3:28 PM, changed "Interdoc" to "Interscript", "rendering" to "internalizing", and "transcribing" to "externalizing" plus various edits necessitated by these substitutions.

Edited by Mitchell, August 23, 1982 2:45 PM, making final version of this report: eliminated const bindings, changed discussion of links; changed examples.