LIMITED DISTRIBUTION: FOR XEROX INTERNAL USE

# Towards an Interchange Standard for Editable Documents

**by Jim Mitchell (Mitchell.PA) and Jim Horning (Horning.PA)**

**Revision 1.1/May 10, 1982**

The **Interscript** standard will define a digital representation of editable documents for exchange among different editing systems. An Interscript *script* 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 document is a draft of 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 in an editor's private format into 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.

> **Note:** This draft is being circulated to interested parties within Xerox to report preliminary ideas. It should not be interpreted as a definitive proposal, and should not be distributed outside.

XEROX
**PALO ALTO RESEARCH CENTER**
**COMPUTER SCIENCE LABORATORY**
**3333 Coyote Hill Road / Palo Alto / California 94304**

# Towards an Interchange Standard for Editable Documents

**by Jim Mitchell and Jim Horning**

**Revision 1/May 4, 1982**

The **Interscript** standard will define a digital representation of editable documents for exchange among different editing systems. An Interscript *script* 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 document is a draft of 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 in an editor's private format into 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 a more general structure than a tree (e.g., for graphics or 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 that 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, or *scripts*. Scripts are created by computer programs (usually an editor or associated program); scripts are "compiled" by programs to produce whatever private or file format a particular editor uses to represent editable documents.

### 1.1. Rationale for an interchange standard

An editing program typically uses a private, highly-encoded representation for documents in order to meet its performance and functionality goals. 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 a document 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 N 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 some of the structures, formats, or content types used by 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 structure, form, and content of editable documents. That external representation has only one purpose: to enable the interchange of documents among a 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. In fact, when adding a new version of a previous editor, the only differences in the new version's conversion routines will be in the areas in which its internal format has changed from its previous form; this represents a significant saving of programming. Finally, no special routines or human procedures would be needed to upgrade documents to a new version of

an editor, since each conforming editor will be capable of understanding and producing the interchange representation anyway.

## **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 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.

*1.2.5. Document form and content*

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 and pieces of content.

The contents of a document must be represented by a rich space containing scalar numbers, strings, vectors, and record-like constructs in order to describe items as varied as distances, text, coefficients of curves, graphics constraints, digital audio, scanned images, transistors, etc.

Attribute values should also be described in this rich value space.

*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 form, structure, or content. 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. Comprehending scripts*

Even complicated documents have simple pieces. A simple editor should be able to display document components that it is capable of displaying, even in the presence of components that it cannot. More precisely, an editor must, in the course of *internalizing* a script , 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 structure, form, and content 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 that 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 structure, form, and content carried by the script from which the document originally came.

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. However, 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 transcribing from its private format.

## **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 rendition 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.

## 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, DDD, Explanation

## **1.4**. **Concepts and Guiding Principles**

### *1.4.1*. *Layers*

The Interscript standard is presented as a sequence of 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*. *Transcription and Rendition*

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 in 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.

Another useful distinction is between the *value* and the *structure* of either form or content within a document. 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 name-value pairs that apply throughout a *scope*, and are placed in the *environment* by means of *bindings*. Contents are not always present to be simply displayed as text. The way the contents of a document are to be "viewed" 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; or . . . . 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.

Content structure 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 constant binding (definition), which may not be superseded within the containing node or any of its subnodes,

> We expect definitions to be used by sophisticated editors for such things as styles. Some scripts will come with a prefix containing non-standard property and attribute definitions that are global to the document. There may be standard libraries containing definitions that allow complex documents to be edited in terms of properties and attributes understood by simpler editors.

":=" denotes a global binding, which prevents the variable name from being reused for any other purpose.

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 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 entire document.

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 repeated content.

*1.4.4.4 Expressions*

Expressions involving the four infix ops (+, , *, /) 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) is borrowed from 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 Labels*

A tag is written as a universal name followed by $''. A tag, `t,` labels a node that contains it with its associated properties and also indirectly refers to the component of the environment with the name "defaults.`t`". Properties 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 implies about the environment (attributes that must be present and their intended intepretation).

A label L! on a node makes that node a target of the link L (and its prefixes); a label L@ makes it a source. The "main" identifier of a link must be introduced (using id@!) at the root of a subtree containing all its sources and targets, and textually preceding them. Each link represents a set of directed arcs, one from each of its sources to each of its targets. Multiple target labels make a node the target of multiple links. Labels provide a very general mechanism for recording structure, such as cross-references, not captured by linear order or the dominant hierarchy.

*1.4.5. Comprehending scripts*

The Interscript standard applies to interchange among editors with widely varying capabilities. It will be important to define some structure to the space of possibilities, 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  const (=), 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 that results from externalizing a document prepared by a high-capability editor. Thus, when internalizing a script, some information may be lost. 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.

**HISTORY LOG**

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.

# Towards an Interchange Standard for Editable Documents

**by Jim Mitchell and Jim Horning**

**May 10, 1982  4:13 PM**

**File: Interdoc-1.5.bravo**

## 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>}
```

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 <>.

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   }
```

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 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   }
```

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 constant whose dimensions are inches/meter; 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   }
```

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   }
```

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. The binding operator is = instead of _ to denote the fact that this binding may not be superseded in this node or any of its subnodes; for this reason such a binding is called a *definition*. 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). Since font is bound using "=", it cannot directly be changed in its scope, although its components can be since they are bound using "_". 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  }
```

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
```

## 1.6. Further Examples

This section gives some more realistic examples of the use of the Interscript language and explores the issues of making sets of standard definitions for use in scripts.

### *1.6.1. A Laurel Message*

Here is a possible Interscript transcription of a Laurel message:

```
0   Interscript/Interchange/1.0                                   -- standard heading --
1   {LAURELMSG$                                                   -- tag for a Laurel document --
2       Sub='PARAGRAPH$ leftMargin_1.0*inch rightMargin_7.5*inch'
3       justified_F                                               -- "_" means overridable default --
4       font.family_TIMES font.size_10
5       leading.x_1
6       leading.y_1                                   -- overridable default leadings --
7       heading@!                                             -- declare a label --
8       laurelInfo =                                  -- Laurel information for easy access; none is changeable --
9           (Heading.time@ Heading.from@ Heading.subject@ Heading.to@ Heading.cc@)
```

```
10        {<Date: > {Heading.time! <18 June 1981 9:18 am PDT (Thursday)>}
11        <From: > {Heading.from! <Mitchell.PA> AUTHENTICATED$}
12        <Subject: > {Heading.subject! <A Sample Document Syntax>}
13        <To: > {Heading.to! <Horning.PA>}
14        <cc: > {Heading.cc! <Mitchell, Interscript.PA>}}
15        leading.y_6                                   -- override outer y leading --
16        {<text of paragraph1>}                        -- node which is a paragraph --
17        {<text of paragraph2>}
18        {<text of paragraph3>}
19  }
```

Line 1 tags this document (by tagging its root node) as a Laurel message, and line 2 tags its subnodes (starting on lines 10, 16, 17, and 18) as paragraphs with default margins. Lines 3 6 bind some other attributes, likely to be relevant to paragraphs. Line 7 declares the main link identifier heading, and lines 8 9 bind to laurelInfo a vector of source links whose targets are the parts of the document of interest for mail transport. Lines 10 14 have similar structures: each consists of a string followed by a node containing a target link for the label heading and text for that Laurel "field." Line 11 is additionally tagged as AUTHENTICATED. Lines 16 18 contain paragraphs constituting the body of the message.

Alternatively, the external environment might well contain a definition of laurel60 that establishes a suitable environment for a Laurel 6.0 document:

```
1   laurel60= ’
2         time@! from@! subject@! to@! bodyNodes@! cc@!
3         LAURELMSG$
4         cr = <#13#> tab = <#9#>
5         p=’PARAGRAPH$ leftMargin_1.0*inch rightMargin_7.5*inch’
6         justified_F
7         font.family=TIMES font.size=10
8         margins.left_2540 margins.right_19050
9         leading.x_1 leading.y_1                   -- overridable default leadings --
10        printForm=
11          ’{p <Date: > time@ tab
12             <From: > from@ cr
13             <Subject: > subject@ cr
14             <To: > to@
15             leading.y_6
16             bodyNodes@
17             <cc: > cc@
18        }’
19        heading = ’LAURELHEADING$ Sub_’TEXT$ LAURELFIELD$’ ’
20        body = ’Sub_’p bodyNodes!’ ’
21        ’
```

One advantage of using source labels for the "bodies" of the To:, From:, etc. fields (lines 11 14, 17) is that they can represent sets of nodes as well as single nodes.

Now the Laurel document would be described by the following script:

```
22  Interscript/Interchange/1.0                     -- standard heading --
23  {laurel60%                                  -- invoke Laurel 6.0 definitions
24    {heading%                                 -- invoke heading style --
25      {time! <18 June 1981 9:18 am PDT (Thursday)>}
26      {from! AUTHENTICATED$ <Mitchell.PA>}
```

```
27        {subject! <A Sample Document Syntax>}
28        {to! <Horning.PA>}
29        {cc! <Mitchell, Interscript.PA>}
30      }
31     {body%                                          -- Invoke body style --
32        {<text of paragraph1>}
33        {<text of paragraph2>}
34        {<text of paragraph3>}
35      }
36   }
```

Invoking laurel60 in line 23 introduces the quoted expressions heading and body into the root node's environment, tags it as LAURELMSG and declares the labels time, from, etc. It also acquires a definition for a print form, which could be used to format the message for sending to a printer. The "%" (indirection) operator indicates that this is intentional structure, to be preserved by each internalization, rather than merely an abbreviation. Thus the message heading and body should "see" the effects of any future changes made to laurel60, by editing its definition. By contrast, p is used as an abbreviation; when the script is rendered, its *value* may safely be copied at each use.

Look at the definition of heading (line 19): the right side is a quoted expression sequence. The first expression of the sequence produces the tag LAURELHEADING$ and the second binds the quoted expression 'TEXT$ LAURELFIELD$' to Sub. As a result, each subnode of the one beginning on line 24 will be initialized by invoking Sub from its containing node, which gives each the tags TEXT$ and LAURELFIELD$.

Similarly, the definition of body (line 20) defines Sub, and the nodes on lines 32 34 will be initialized by invoking p and having the target link bodyNodes placed on it. Labelling the set of body nodes this way means that the source link, bodyNodes@, in printForm (line 19) denotes the entire sequence of body nodes, in left-to-right depth-first tree order.

### *1.6.2. A page of a Star document*

This example is taken from page 71 of the Star Functional Specification and shows one page of a paginated document with a diagram and a footnote (we recommend that you have that page in front of you when analyzing this transcription):

```
-- pages 1 .. 6 supposedly precede this one --
{pg.a7!
    Sub_'PARAGRAPH$'
    {<Many of these conclusions are based on prior experience>
        {fn.n1!                                  -- just a unique label: fn! introduced somewhere earlier --
        FOOTNOTE$
        <See the 1970 report titled "Organizational Changes and Sales Margin" and other documents referenced in that
        document. Further reports are available if you need them.>
        }
    < which has shown our techniques to be valid. Other data can be collected by future changes to your accounting and
    billing packages, which will allow us to perform even better analyses and lead to better problem discovery and
    correction.>
    }
    {<The results of the sales analysis suggest that certain organizational changes can improve the overall efficiency of the
    operation. The March figures, in particular, bear this out. You will note below a suggested change that we feel will
    correct the problems noted in the analysis above.>
    }
```

```
Sub_'FRAME$'                    -- change to subnode tag FRAME --
{Alignment.horizonally_FlushLeft Alignment.vertically_Floating
    height_2.8*inch width_3.67*inch
    edges.expandingRightEdge_T
    border_dots1
    -- change to default subnode environment Rectangle with solid, double width outline --
    Sub_'RECTANGLE$ lineType.width_2 lineType.style_solid'
    rect@!                      -- declare label class to be used below --
    {rect.a1! UpperLeft_(.0254 .07)     shading_7  height_.01 width_.027    {Title <Headquarters>} }
    {rect.a2! UpperLeft_(.073 .015)     height_.01 width_.018    {Title <Staff Support>} }
    height_.013             -- attribute value shared by following subnodes
    {rect.a3! UpperLeft_(.02 .03)       width_.025    {Title <Development>} }
    {rect.a4! UpperLeft_(.02 .03)       width_.028    {Title <Manufacturing>}  }
    {rect.a5! UpperLeft_(.042 .055)     width_.016    {Title <West Coast>} }
    {rect.a6! UpperLeft_(.067 .055)     width_.016    {Title <East Coast>} }
    -- default subnode environment is LINE with solid, double width outline --
    Sub_'LINE lineType.width_2 lineType.style_solid'
    ln@!
    {ln.out1!   rect.a1@   ln.in34@}
    {ln.out2!   rect.a2@   ln.out1@}
    {ln.in3!    ln.in34@   rect.a3@}
    {ln.in4!    ln.in34@   rect.a4@}
    {ln.in34!   ln.in3@    ln.in4@}
    {ln.out4!   rect.a4@   ln.in56@}
    {ln.in56!   ln.in5@    ln.in6@}
    {ln.in5!    ln.in56@   rect.a5@}
    {ln.in6!    ln.in56@   rect.a6@}
}               -- end of Frame1 --
Sub_'PARAGRAPH$'                    -- restore default subnode initialization to PARAGRAPH --
{<The process of switching to this new organization will not be an easy one. However, the reports seem to suggest many
reasons why it should not be postponed. In particular, the separation of Manufacturing from Development should have
significant impact.>}
{<Also, we feel strongly that merging East and West Coast Development will help. As we have suggested in past reports,
there has always been considerable replication of effort due to this geographic separation. You will recall the events
leading up to the initial contract with our firm.>}
}           -- end of page --
```

### *1.6.3. Some Star property sheets*

Here a few of the definitions invoked in the above example (these were derived from page 148 of the Star Functional Specification). Some of them simply give default values for various attributes; some, like default.font, define a collection of related attributes as an environment; and most are quoted expression sequences for providing abbreviations or "decorating" nodes with tags and their environments with relevant attributes. These definitions would exist in the external environment for Star produced scripts. They would be made accessible to other editors as part of the definition of XEROX.Star.Version1.

#### *1.6.3.1. Font-related defaults and definitions*

```
baseline_0              -- the base line for characters --

underlined_F            -- whether or not text in node is to be underlined --

strikeOut_F             -- whether or not text in node is to have strike-out line through it --

-- there is no rhyme and little reason behind the names of type fonts. The following definition is intended to provide enough
choice, using standard "terms" to name any existing font in an arbitrary font catalog (of course, it doesn't, but perhaps it is
close enough) --
default.font = [ |              -- Definition --
    family_Times                -- a font family name --
```

```
face_[ |                          -- Definition --
    weight_NORMAL                 -- In (EXTRALIGHT, LIGHT, BOOK, NORMAL, MEDIUM,
                                      DEMIBOLD, SEMIBOLD, BOLD, EXTRABOLD, ULTRABOLD,
                                      HEAVY, EXTRAHEAVY, BLACK, GROTESQUE) --
    lineType_SOLID                -- In (SOLID, INLINE, OPEN, OUTLINE, DISPLAY, SHADED) --
    proportions_NORMAL            -- In (NORMAL, CONDENSED, EXPANDED, EXTENDED,
                                          WIDE, BROAD, ELONGATED) --
    style_ROMAN                   -- In (ROMAN, GOTHIC, EGYPTIAN, CURSIVE, SCRIPT) --
    slant_NIL                     -- In (NIL, ITALIC, OBLIQUE) --
    swash_F          -- T => use swash capitals --
    lowercase_T      -- T => use lowercase letters --
    uppercase_T      -- T => use uppercase letters --
    smallCaps_F      -- T => use small capitals --
    ]
size_10*pt                                        -- distance --
]

-- some useful font shorthands: --
Helvetica = 'font _ [default.font% | family_HELVETICA]'
Italic = 'font.face.slant_ITALIC'
Bold = 'font.face.weight_BOLD'
Helvetica10BI = 'Helvetica font.size_10*pt Bold Italic'
```

### *1.6.3.2. Footnote-related definitions*

```
fnCount:=0                        -- global variable for counting footnotes
FOOTNOTE = 'fnCount:=+1 font.size_8*pt FootnoteRef%'

FootnoteRef = '{FOOTREF$ baseline_+5*pt fnCount}'          -- raise 5 pts --
```

### *1.6.3.3. Paragraph-related definitions*

```
Tab = [ |
    position_0
    type_LEFT                     -- In (LEFT, CENTERED, RIGHT, DECIMAL) --
    ]

MakeTabs='n_0 tabs_(RecursiveMakeTab[Value])'
RecursiveMakeTab='(EQ[Value 0] | NIL | n_+.25*inch  [Tab | position_n ] RecursiveMakeTab[Value-1])'

Default.PARAGRAPH = 'Indent = [ | Left_0.0 Right_0.0]                    -- distance --
    Alignment_FLUSHLEFT                   -- In (FLUSHLEFT, FLUSHRIGHT, BOTH, CENTERED) --
    Justified_F
    leading_[leading | between_1*pt  above_12*pt  below_0]
    charStyle_[|
        Normal_'font_default.font'
        Emphasis1_'font_default.font Italic'
        Emphasis2_'font_default.font Bold'
        ]
    Hyphenation_F
    KeepOn_NIL                        -- In (NIL, SamePageAsNextParagraph) --
    MakeTabs[8]                       -- binds tabs to a sequence of 8 tabs (0, .25 inch, .50 inch, . . .) --
    charStyle.Normal                  -- initializes to normal style
```

### *1.6.3.4. frame, rectangle, and line definitions*

```
Def.UpperLeft = 'UpperLeft_(0.0 0.0)'      -- Def is just a convenient environment in which to put useful auxiliary
definitions --

Def.lineType = '
    lineType_[ |
        Visible_T
```

```
        Width_1
        Style_SOLID]                                -- IN (SOLID, DOT, DASH, DOTDASH, DOUBLE, . . .) --
    ’

Def.Shading = ’Shading_0’

Def.Box = ’Def.UpperLeft  Def.lineType  Def.Shading’

Frame = ’FRAME$  Def.Box’

Rectangle = ’RECTANGLE$  Def.Box
    Constraint_MagnifyOnly                  -- IN (NIL MagnifyOnly) --
    ’

Def.LineEnd = ’
    LineEnd_(LeftUpper_Flush  RightLower_Flush)          -- IN (Flush Round Square arrow1 arrow2 arrow3) --
    ’

Line = ’LINE$  constraint_FixedAngle  Def.lineType  Def.LineEnd’

Title = ’CAPTION$ Paragraph’
```

### *1.6.4. Using links*

Links are intended to provide the means for associating nodes in non-hierarchical ways. They can be used for referring to figures, examples, tables, etc., for describing tables of contents, for denoting index items, keeping lists, etc.

#### *1.6.4.1. References to figures*

The following outlines how the labelling facilities and global bindings can be used to generate references to (source links for) a figure whose number may not be known at the point of reference. The identifier n5 is assumed to have been generated by the program that produced the script and is assumed to be unique over the target labels with naming prefix "figures." in the script.

```
figures@!  figCount:= 0                                  -- should appear in a script’s root node --
makeFigureNum = ’HIDDEN$ figCount:=+1 figCount’

{. . . figures.n5@ . . .}                                -- ref to node with label figures.n5! --

{ . . . {figures.n5! makeFigureNum} . . .}               -- a hidden node holding the figure number --
```

The node in which the figure number for figure n5 is defined contains a tag, HIDDEN, which means that the node is not to be considered a part of the dominant structure for display purposes even though it is part of it. The node’s sole content is the value of figCount after it has been (persistently) incremented by 1. Because figCount is bound with ":=", the scope of the binding is global.

#### *1.6.4.2. Collections of index items*

Assume that the word "framble" is to be considered an index item in certain places where it occurs in a document. The link class Indexable@! should be introduced at the root of the document, and each to-be-indexed occurrence of "framble" in a string, e.g., <When a framble is found, it . . .>, should be replaced by the sequence <When a > framble% < is found, it . . .>. Somewhere in the script within the scope of the declaration of Indexable, at the root of a subtree containing all the uses of framble should be the following definition:

```
framble='{HIDDEN$ indexable.framble! pageNumber} <framble>'
```

Invoking framble results in the appearance of a hidden node containing the current page number (assumed to be held in the attribute pageNumber) and labelled as being in the set of target links indexable and indexable.framble. The index for the document might then contain the following entry for "framble":

```
{INDEXENTRY$ <framble> indexable.framble@}
```

This entry contains the minimal information needed to generate the sequence of page numbers corresponding to indexable occurrences of framble. If some occurrences are considered primary and some secondary, then these mechanisms can be generalized to have framble defined as

```
framble=[ | primary='{HIDDEN$ indexable.framble.primary! pageNum} <framble>'
          secondary='{HIDDEN$ indexable.framble.secondary! pageNum} <framble>']
```

Primary references are denoted in the script as framble.primary% and secondary ones as framble.secondary%. Similarly, the index entry takes the form:

```
{INDEXENTRY$ <framble> indexable.framble.primary@  indexable.framble.secondary@}
```

### *1.6.5. Using indirections*

Indirections provide a way to centralize (and delay) the binding of information within a document. They can be used to share information that is intended to be consistent.

#### *1.6.5.1 Styles and style sheets*

Documents generally follow stylistic conventions for presenting different kinds of content. E.g., major headings may be in bold face with twelve points of extra leading, minor headings in italic with six points of extra leading. If this information is explicitly bound for each piece of content, then a stylistic change may require locating and changing all the relevant bindings (note that italic is likely to be also used for other purposes, such as *emphasis*). If, however, the binding is done indirectly, through a *style*, a single change will be effective for all places where the style is referenced. Note that each occurrence of a tag implicitly establishes an indirection through the same identifier; this is convenient in associating styles with semantically meaningful tags. For example:

```
MajorHeading = 'PARAGRAPH$ Bold leading_+12'
MinorHeading = 'PARAGRAPH$ Italic leading_+6'
```

#### *1.6.5.2 Technical terms*

Terminology may be undergoing change while a document is in production. For example, the previous version of this document used "mark" for what is now called "tag." One way to defer decisions on terminology, while ensuring that each version of the document is self-consistent, is to use an indirect reference for each occurrence of a term that may have to be rebound later.

## 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 17, 1981 1:37 PM, folding in JJH's edits.

Edited by Mitchell, September 18, 1981 12:45 AM, added considerable annotation of examples.

Edited by Horning, May 4, 1982 12:30 PM, Fold in Truth Copy edits

Edited by Horning, May 10, 1982 4:12 PM, changed "Interdoc" to "Interscript", "rendering" to "internalizing", and "transcribing" to "externalizing" plus various edits necessitated by these substitutions.

## 2. The Language Basis: Syntax and Semantics

### 2.1. Grammar

Our notation is basically BNF with terminals quoted and augmented by the following conventions:

a sequence enclosed in [ ] brackets may occur zero or one times;

a construct followed by * may occur zero or more times;

parentheses ( ) are used purely for grouping.

```
script        ::=   versionId node
versionID     ::=   "Interscript/Interchange/1.0   "
item          ::=   content | binding | label
content       ::=   term | node
term          ::=   primary | primary op term
op            ::=   "+" | " " | "*" | "/"
primary       ::=   literal | invocation | indirection | application | selection | vector
literal       ::=   Boolean | integer | intSequence | real | string | universal
invocation    ::=   name
name          ::=   id ( "." id )*
indirection   ::=   name "%"
application   ::=   ( name | universal ) "[" item* "]"
universal     ::=   ucID ( "." ucID )*
selection     ::=   "(" term "|" scope* "|" scope* ")"
vector        ::=   "(" scope* ")"
node          ::=   "{" scope* "}"
scope         ::=   ( binding | label )* content content*
binding       ::=   name mode rhs
mode          ::=   "_" | "=" | ":" | ":="
rhs           ::=   content | op term | "'" scope* "'" | "[" [ item* ] "|" binding* "]"
label         ::=   tag | link
tag           ::=   universal "$"
link          ::=   id "@!" | name "@" | name "!"
```

### 2.2. Discussion of Features

[Note that we have a formal semantic definition for this language that is every bit as precise as the grammar above. However, we have not yet figured out how to present it in a form that humans find equally palatable, so we postpone it to an appendix.]

```
primary       ::=   literal
literal       ::=   Boolean | integer | intSequence | real | string
```

The primitive elements by which the value of a document is represented.

term ::= primary op term
op ::= "+" | " " | "*" | "/"

Both the primary and the term must reduce to numbers; the arithmetic operators are evaluated right-to-left (*a la* APL, without precedence) and bind less tightly than function application. The result is a real if either operand is.

invocation ::= id

Id is looked up in the current environment; depending on its current binding, this may produce contents, bindings, and/or labels; if the rhs bound to id was quoted, that expression is evaluated in the current environment. In the (implicit) outermost environment, every id is bound to the corresponding universal (ID).

invocation ::= name "." id

Qualified names represent lookup in "nested" environments; name must have been bound to an environment, in which id is looked up.

indirection ::= name "%"

This indicates an intentional indirection through name, which should be preserved as part of the structure; replacing the indirection by its value in the current environment is a value-preserving loss of structural fidelity. (An invocation that is simply a name is an abbreviation that need not be preserved.)

universal ::= ucID ( "." ucID )*

Universals are like names, but written entirely in upper case letters. They are presumed to be defined externally, so they are not looked up in the environment.

application ::= ( name | universal ) "[" item* "]"

If the application involves a universal (either explicitly, or because the name is bound to a universal), the corresponding function is applied to the argument list that results from evaluating item*. Part of the definition of Layer 2 will involve the specification of a small set of standard functions, which may be expanded in various Layer 3 extensions.

If name is not bound to a universal, the current environment is temporarily augmented with a binding of the value of item* to the identifier value, and the value of the application is the result of evaluating name in that environment; this allows function definition within the language.

Neither form of application changes the environment of succeeding expressions.

selection ::= "(" term "|" scope1* "|" scope2* ")"

This is a standard conditional item sequence, using syntax borrowed from Algol 68. The value and effect are those of item1* if the term evaluates to "T" in the current environment,

those of item2* if it evaluates to "F".

vector ::= "(" scope* ")"

Parentheses group a sequence of items as a single vector; bindings in scope* affect the environment of items to the right in the containing node, but labels have no meaning.

node ::= "{" scope* "}"

Nodes have nested environments, and affect the containing environment only through global (:=) bindings to ids. Scope* is implicitly prefixed by an invocation of Sub, which may be bound to any sequence of items intended to be common to all subnodes in a scope.

item* ::= ""

The empty sequence of items has no value and no effect; this is the basis for the following recursive definition.

item* ::= item1 item*

In general, the value of a sequence of items is just the sequence of item values; binding items change the environment of items to their right in the sequence.

binding ::= name mode rhs

This adds a single binding to the current scope (i.e., to its associated environment); bindings have no other "side effects" and no value (i.e., they do not change the length of a containing vector or node value).

binding ::= name mode op term

"name mode op term" is just a convenient piece of syntactic shorthand for "name mode name op term".

rhs ::= "’" scope* "’"

A quoted rhs is evaluated in the environment of invocation, rather than the environment current at the point of binding.

rhs ::= "[|" binding* "]"

This creates a new environment value that may be used much like a record.

rhs ::= "[" item* "|" binding* "]"

This creates a new environment value that is an extension of the environment that is the value of item*.

tag ::= universal "$"

This gives the containing node the property denoted by the universal.

link ::= id "@!"

This introduces the set of links whose main component is id, and defines their scope.

link ::= name "@"

This identifies the immediately containing node as a source of the link name.

link ::= name "!"

This identifies the immediately containing node as a target of each of the links that is a prefix of name.

### 2.3. Safety Rules for Low-capability Editors

Conservative rules for editor treatment of script nodes created by other editors:

It's OK to display a node if
- you understand at least one of its properties.

It's OK to edit (the items in) a node if
- you understand *all* of its (local) properties, and either
  - you don't remove any of them, or
  - you also understand *all* properties of its parent.

It's OK to copy a node if
- you understand *all* properties of its new parent,
- no labels are moved outside their scope, and
- the two environments have the same bindings for all attributes that you don't either
  - understand, or
  - know can't be relevant, and anywhere in the node or its subnodes.

It's OK to delete a node if
- you understand *all* properties of its parent.

[Less stringent rules will suffice if the document is merely to be viewed, rather than edited, using the original editor.]

### 2.4. Encodings

[Any resemblance between the following material and the corresponding section of the Interpress standard is purely an intentional consequence of plagiarism.]

The script for a document can be encoded in many different ways. This section gives the rules for designing encodings. The purpose of these rules is to ensure that information is not

lost or added by conversions from one encoding to another. There are two types of encodings: a single interchange encoding and many possible private encodings.

The interchange encoding is used to transmit a script from one site to another when the two sites must be assumed to be arbitrarily different. A private encoding is used to transmit scripts from one site to another when the two sites share the private encoding conventions. For example, a line of document-preparation products made by the same manufacturer might share a private encoding, which can be used to transmit documents from one editor in the product line to another; presumably this encoding is designed to make these transfers simpler or more efficient. However, when one of these editors transmits a document to an unknown editor, the interchange encoding must be used. The interchange encoding is designed to allow easy generation, transmission, and interpretation by many different editors, possibly at the expense of compactness and speed of encoding and decoding.

### *2.4.1. The interchange encoding*

The interchange encoding is designed to simplify creation, communication and interpretation of scripts for the widest possible range of editors and systems. For this reason, a script in the interchange encoding is represented as a sequence of graphic (printable) characters taken from the ASCII set; the subset of ASCII used is also a subset of ISO 646. Communication of a script in the interchange encoding requires only the ability to communicate a sequence of ASCII characters; Interscript does not specify how the characters are encoded. In effect, we define a text representation of the commands to be executed.

The choice of a text format for the interchange encoding leads to rather lengthy scripts in some cases. The bulk of an interchange script presents no great problem for document storage, since a document need not be stored in this form. Rather, as it is transmitted, the sending editor can translate its own private encoding into the interchange encoding. Similarly, the receiving editor can translate the interchange encoding into its own, usually different, private encoding for storage. However, a bulky interchange script may be more expensive to transmit. If a document consists mostly of text, the interchange encoding is quite efficient very few characters are required in addition to those appearing in the document itself.

*Character set*. The character set used in the interchange encoding is described by the ISO 646 7-bit Coded Character Set For Information Processing Interchange. The interchange encoding interprets the 94 characters of the G1 set defined in the International Reference Version (ISO 646, Table 2) and the space character (2/0). This set of 95 characters is called the interchange set. Note that except for the concise "string" encoding of vectors described below, the interchange encoding has nothing to do with the integers corresponding to the characters, but depends only on the character set itself.

**It is extremely important to understand that the choice of the ISO standard for the interchange format has nothing to do with character mappings in Interscript fonts**. Although these mappings must adhere to a character set standard that is shared by interchanging editors, that standard is not part of Interscript. It is expected that Xerox will develop a separate corporate standard in this area.

If the underlying encoding of the ISO character set can also encode other characters (e.g., the control characters (0/0 through 1/15) and del (7/15), or another group of 128 characters if eight bits are being used to encode each character), these are ignored in interpreting an interchange script. This does not mean that these characters are converted to spaces, but that they are treated as if they were not present.

There are several reasons for this choice:

Control characters may be inserted freely by software that generates the interchange encoding. For example, carriage returns (0/13), line feeds (0/10), and form feeds (0/12) may be inserted at will to conform to limitations that may be imposed by an operating system. Restrictions on line length or the use of fixed-length records thus become straightforward.

Control characters may be removed or inserted freely by software that receives the interchange encoding. In this way, the receiving software can adhere to any restrictions imposed by its operating system.

The absence of control characters allows certain kinds of "non-transparent" data communication methods (such as binary synchronous communication) to be used freely.

A minor disadvantage of these conventions is that if a script is typed in, care must be taken not to omit a significant space at the end of a line. Since scripts are normally generated by programs, this is not important. A system for manually generating (and perhaps interactively debugging) Interscript should provide for various convenience features on input, and for prettyprinting the script on output.

Any number of space characters may also be added after any token without changing the meaning. Throughout the following, a delimiter is a space or comma, which may be omitted if the next character is not an alphanumeric, " " or ".".

VersionId. The first characters of an interchange script conforming to this version of the Interscript standard must be "Interscript/Interchange/1.0 ". Note that the VersionId is of variable length, and ends with a space. These conventions simplify the design of systems that must deal with more than one kind of encoding.

If a privately encoded script can be interpreted as a sequence of characters, its first characters must be "Interscript/private/i.j", where private is replaced by an appropriately chosen hierarchical name that identifies the encoding, e.g., "Xerox/860", and i.j is replaced by an appropriate version identification, e.g., "2.4"; the resulting header would be "Interscript/Xerox/860/2.4".

A private encoding that cannot be interpreted as a sequence of characters (e.g., a binary, word-oriented encoding on a 36-bit machine which packs five 7-bit characters into a word) should use any available convention to make its scripts self-identifying.

Following the versionId is a node constituting the body of the script, with values encoded as follows.

Integer. An integer is represented in radix 10 notation using the characters "0" through "9" as digits, followed by a delimiter. A negative integer is preceded by a minus sign " ". Thus the decimal number 1234 is encoded as "1234 ", and 1234 is encoded as " 1234 ". The

delimiter may be empty if the following character is a letter.

A sequence of integer literals in the range 0..255 can be represented in radix 16 notation using the characters "A" through "P" as digits ("A" corresponds to 0, "P" to 15). The entire sequence is enclosed in "#" brackets. For example, the integer 93 is represented as "#FN#", and the sequence of integers 93, 94, 95, 96 as "#FNFOFPGA#". These sequences require only two characters for each integer (plus two characters of overhead). Note that there is no delimiter between the integers in this encoding. Ordinary integer literals, with their delimiters, may be included in the sequence; e.g., 7, 93, 400, 40 could be represented by "#7,FN400CI#".

Booleans are represented by the characters "F" and "T", followed by a delimiter.

Real. A real is represented using Fortran E or F notation, with a trailing delimiter. Thus "12.34 " is the same as "1.234E1 ". Minus signs may precede the mantissa or the exponent: " 12.34E 3 ".

Identifier. An identifier is encoded by its characters (which are limited to letters and digits), followed by a delimiter: "x ", "arg1 ". The first character of an identifier must be a letter, and must be written in lower case to distinguish identifiers from universals. Other letters may be written in either case for readability, since case is not significant in distinguishing identifiers.

Vector. A vector is encoded by surrounding a sequence of values with parentheses, "(" and ")".

String. A text vector usually contains integers that are interpreted as character codes. Often these codes lie in the range 32 to 126 inclusive, which are the numbers assigned to the characters of the interchange set by ISO 646. It is convenient to encode an element of such a vector by the character whose ISO code is the desired value. Such a string can be encoded by surrounding the characters with "<" and ">", thus "<Hello!>". If the string contains elements outside the allowed range (i.e., if the value is less than 32 or greater than 126) or the value 62 **or XX** (the ISO codes for the characters ">" and "#"), those elements must be represented as integers inside "#" brackets, as described above. The two-character encoding of small integers is designed to make escape sequences compact. Thus "<Hello!>", "<Hello#CB#>", "<Hel#GMGP#!>", and "<Hello#33#>" are all equivalent.

Universal names. A universal is encoded by giving its name in upper case letters, followed by a delimiter. E.g., "TEXT ".

Node. A node is encoded by a "{", followed by a sequence of items, followed by a "}".

Comment. The beginning and end of a comment are both marked by a double minus sign: the sequence " " <any characters other than " "> " " is a comment and may occur between any two tokens. Comments are ignored in rendering the script.

The tokens of the interchange encoding are defined by the following BNF grammar, together with rules about delimiters:

> The delimiter that terminates an identifier or universal may only be empty if the next character is not an alphanumeric, " ", or ".".
>
> The delimiter that terminates an integer may only be empty if the next character is not a digit, "E", "F", " ", or ".".
>
> extra delimiters may be inserted after any token.

```
token          ::=    literal | id | ucID | op | bracket | punctuation | comment
literal        ::=    Boolean | integer | intSequence | real | string
Boolean        ::=    ( "F" | "T" ) delimiter
delimiter      ::=    " " | "," | empty
empty          ::=    ""
integer        ::=    [ " " ] digit digit* delimiter
digit          ::=    "0" | "1" | "2" | "3" | "4"  | "5"  | "6" | "7" | "8" | "9"
intSequence    ::=    "#" intOrHex* "#"
intOrHex       ::=    integer | hexChar hexChar
hexChar        ::=    "A" | "B" | "C" | "D" | "E"  | "F"  | "G" | "H" | "I" | "J" | "K" | "L" | "M" |
                      "N" | "O" | "P"
real           ::=    [ " "  ] digit digit* "." digit* [ "E" integer ] delimiter
string         ::=    "<" stringElem*  ">"
stringElem     ::=    stringChar | intSequence
stringChar     ::=      any character but "#" or ">"
id             ::=    lowerCase idChar* delimiter
idChar         ::=    letter | digit
letter         ::=    lowerCase | upperCase
lowerCase      ::=    "a" | "b" | "c" | "d" | "e"  | "f"  | "g" | "h" | "i" | "j" | "k" | l" | "m" | "n" |
                      "o" | "p" | "q" | "r" | "s" | "t" | "u"  | "v"  | "w" | "x" | "y" | "z"
upperCase      ::=    hexChar | "Q" | "R" | "S" | "T" | "U"  | "V"  | "W" | "X" | "Y" | "Z"
ucID           ::=    upperCase* delimiter
op             ::= "+" | " " | "*" | "/"
bracket        ::=    "(" | ")" | "{ " | "}" | "<" | ">" | "[" | "]" | " "'
punctuation    ::=    "." | ";" | ":"  | "=" | "_" | "!" | "#" | "@" | "|"
comment        ::=    "  " commentString "  "
commentString  ::=          any sequence of characters not containing "  "
alphanumeric   ::=        letter | digit
```

A simple listing of an interchange script can just print the character sequence, with line breaks every n characters, or perhaps at the nearest convenient delimiter. Such a listing is reasonably easy to read, so that problems can be tracked down simply by studying it. Additional help in reading the file can be furnished by utility programs which format the file for more pleasant reading.

### *2.4.2. Normalization*

Every encoding must define a normalization function N, which maps a script in the encoding into another script in the encoding which generates the same output. N must be

idempotent (i.e., $N^2$=N); it may not change the fidelity level of the script (see 2.4.3). If a script violates the definition of Interscript, a normalization function may report this fact instead of producing a normalized result. In other words, normalization need not be defined on erroneous scripts.

The purpose of this function is to make possible a precise description of the rules for private encodings in section 2.4.4. The idea is that when an encoding provides several ways of saying the same thing (typically a basic way, and some more concise ways which work in common special cases), the normalized script will uniformly choose one way of saying it. Note that the normalized script is not intended for any purpose other than precisely defining a notion of equivalent script; it is neither especially compact nor especially readable.

The normalization function for the interchange encoding is defined as follows:

Comments are omitted.

Delimiters are replaced by empty if possible, otherwise with ",".

An integer encoded in hex is replaced by the same integer encoded in digits; except in strings, "#" brackets are replaced by parentheses.

Leading zeros are dropped from a digits encoding of an integer.

Reals are uniformly encoded in E format with a single non-zero digit to the left of the "." and no trailing zeros; 0 is encoded by "0.0".

An upper case letter in an identifier is replaced by the corresponding lower case letter.

Each direct invocation (abbreviation) is replaced by its binding.

### *2.4.3. Level restriction*

For each rendition fidelity level L of Interscript, there is an (idempotent) *level restriction function* RIL which converts an arbitrary interchange script into an interchange script of level L. An interchange script is of level L if RIL applied to it is the identity. A restriction function replaces an excluded structure with its value according to the semantics of Interscript, converts excluded form information into additional content with a special property, and removes excluded tags.

### *2.4.4. Private encodings*

A private encoding may use any scheme for expressing the content of a script. Certain requirements are imposed on private Interscript encodings to ensure that they can express the entire content of a script at a given level, and no more. Since no general statements can be made about the bits, characters or other low level constituents of a private encoding, these constraints are stated in terms of the existence of certain functions that convert private encodings to interchange encodings and vice versa. An encoding for which these functions do not exist is not an Interscript encoding. The recommended way of demonstrating that the functions exist is to exhibit them as executable programs. This makes it easy to run test cases.

A particular private encoding has a fixed fidelity level. Informally, this means that it can encode any script of that level.

For any private Interscript encoding P of fidelity level L, the following functions must exist:

NP, the normalization function for P; see 2.4.2.

CPI, a conversion function from a script in P to an interchange script of level L.

CIP, a conversion function from an interchange script of level L to a script in P.

If a script violates the definition of Interscript, a conversion function may report this fact instead of producing a converted result. In other words, conversion need not be defined on erroneous scripts.

Given these functions, we can define functions which convert normalized private scripts to normalized interchange scripts of level L and conversely:

NPI=NIoCPI

NIP=NPoCIP

In other words, first convert to the other encoding, and then normalize. These functions must be inverses of each other.

This means that after normalization (which does not change the output), a private script can be converted to an interchange script and then back to the same private script, and vice versa. Hence it seems reasonable to say that the private encoding can express exactly the same information.

**[We need to say similar things about editor representations, transcription fidelity, and rendition fidelity.]**

Many tricks are available for designing private encodings with desirable properties. With some knowledge of the statistics of actual scripts, encodings can minimize the number of bits required to represent the average script, by Huffman or conditional coding of the primitives. For example, if strings consist primarily of ordinary written English text, an encoding with five bits per character might be attractive: lower case letters except "q", "x", and "z" (23), space, comma space, semicolon space, colon space, dot space space one upper case character, escape to upper case, one upper case character, escape to digits, one digit character (32 total). The upper case and digits sets would be analogous. A more complex, but perhaps even more compact encoding would take account of the letter frequencies in English text. Similarly, the most common labels can be encoded compactly.

There are other useful ideas for private encodings. The bracketting constructs may be replaced by constructs with explicit length fields; these can be shorter, it is easy for the decoder to skip the bracketted constructs, and if the script is damaged it is easier to recover than from the loss of a closing bracket. Hints can be associated with nodes that will speed translation to a particular editor’s representation.

In designing a private encoding, it is advisable to handle all the constructs of Interscript reasonably compactly, rather than allowing some "unpopular" ones to be encoded very clumsily. Otherwise scripts originally generated in another encoding may cause terrible performance.

# Towards an Interchange Standard for Editable Documents

**by Jim Mitchell and Jim Horning**

**May 10, 1982  5:40 PM**

**File: Interdoc-3ff.bravo**

## 3. Higher-Level Issues

### 3.1. Standard and Editor-Specific Transcriptions:

We need a two-level structure for documents expressed in the base language to be both (a) interchangeable among different editors, and (b) retain information of special significance to a specific editor. We call (a) the interchange standard information, or standard information and (b) editor-specific information.

Basically, an editor X is free to couch properties in its own terms, which can make it easy for it to consume a script produced by itself, but it must provide a set of mappings which will transform properties into the interchange standard. The recommended method for doing this is to invoke its name as the very first item in the root node of any X-specific subtree. The rules for inheritance of properties mean that often only the root node of a document will need to have this property, but there is nothing wrong with nodes being in different editor-specific terms provided they invoke the appropriate editor properties.

Now, to be a valid standard script, the document must have the definition of the name X placed in the script itself (There is nothing wrong with having libraries of editor-specific _ standard mappings in a library of some sort to avoid having copies of them in each script).

When X parses an X-specific script, it will use its X-specific attributes and never invoke the mappings from X-specific information to standard terms; i.e., it can use a null definition for the name X. However, when such a document is interpreted by some other editor Y, any time it tries to access a standard name, the mapping from that name to the corresponding expression in terms of the X-specific values in the script will have been provided by the definition of X. What guarantee is there that this can always be done?

It is worth noting first that we are speaking here of a script being rendered for an editor, rather than produced. Consequently, it will never be necessary to access standard names in left-hand contexts; i.e., to do bindings that are not part of the script in order to interpret it. It may, however, need to access the components of environments in order to render the script into its private format. These are always values in right-hand side contexts, and must be computed in terms of the X-specific information that X put in the script. We can examine this issue on a case-by-case basis. Below is a list of examples of possible editor-specific uses of the base language and the mappings that would allow another editor to treat the document in standard terms:

Symbolic values used instead of numbers: supply standard values for the symbolic values:

Standard:

```
leading.between _ 1*pt       -- some numeric value --
```

Editor-specific:

```
leading.betweenLines _ single
leading.above _ double
```

mapping:

```
single = 2*pt
double = 4*pt
```

Different names used for standard names: supply a binding to the standard name from the editor-specific name using a quoted expression so that it is only evaluated when needed

in a right hand context:

Standard:

```
leading          _    [above_10*pt   between_2*pt  below_0*pt ]
```

Editor-specific:

```
Space            _    [BetweenLines_single  BeforePara_double  AfterPara_single]
```

mapping:

```
leading.between  _    'Space.BetweenLines'
leading.above    _    'Space.BeforePara'
leading.below    _    'Space.AfterPara'
```

Different concepts used for standard ones: supply a binding to the standard attribute names from the editor-specific concepts using quoted expressions so that they are only evaluated when needed in right hand contexts:

Standard:

```
leading _ [above_10*pt   between_2*pt  below_0*pt ]
```

Editor-specific:

```
Spacing _ [fontSize_10  on_14  leading_1]          -- all units assumed to be pts --
```

mapping:

```
leading.between  _    'pt*Spacing.on Spacing.fontSize'        -- value is in pts --
leading.above    _    'pt * Spacing.leading+Spacing.on Spacing.fontSize'
leading.below    _    0
```

In general, one can use the facilities of the base language to write essentially arbitrary programs that can, by being quoted, be bound to a standard identifier to cause the appropriate value to be computed based on editor-specific information put in the document by the editor that produced it. Moreover, since the mappings provided by editor X can be overridden in any subtree of the document, an editor that does not "understand" some subtree of a document produced by another editor Y can simply leave that subtree intact when producing an edited version of the original script except to ensure that that subtree's root node's first expression is an invocation of "Y", which will cause Y's editor-specific mappings to obtain in that subtree.

### **3.2**. **Standard External Environment**

It will be important to provide for a standard external environment for rendering scripts so that standard definitions need not be carried along with every script that uses them. The external environment will contain definitions for units (inch, pt, etc.), various "styles" (para, figure, etc.), and useful abbreviations (italic, bold, etc.).

#### *3.2.1*. *Units*

The Interscript standard assumes that distances are in meters and angles are in degrees. Using the language and the following constants defined in the standard external environment, a script can readily express distances and meters in other, possibly more convenient units:

```
meter=1.0                                -- IN TERMS OF METERS --
mica=1.E-5*meter        -- mica          = 1.E-5 --
inch=.0254*meter        -- inch          = .0254 --
pt=.013836*inch         -- pt            = .00035143 --
pica=12*pt              -- pica          = .00421752 --
tenPitch=inch/10        -- tenPitch      = .00254 --
twelvePitch=inch/12     -- twelvePitch   = .00211667 --
```

```
degree=1.0                      -- ANGLES ARE IN DEGREES --
pi=3.14159265
radian=180*degree/pi            -- = 57.29577951 --
```

## 4. Pragmatics

Private encodings and private representations
Conversion efficiency
Implementation considerations

## APPENDIX A

## GLOSSARY

An *italicized* word in a definition is defined in this glossary.

**abbreviation** An *invocation* used to shorten a *script*, rather than to indicate structure

**attribute** A component of an *environment*, identified by its *name*, which is bound to a *value*

**base language** The part of the *Interscript* language that is independent of the semantics of particular *properties* and *attributes*

**base semantics** The semantic rules that govern how *scripts* in the *base language* are elaborated to determine their *contents, environments,* and *labels*

**binding** The operation of associating a *value* with a *name* to add an *attribute* to an *environment*; also the resulting association

**binding mode** A *value* may be bound to an *identifier* as *const, var, local,* or *persistent*

**Boolean** An enumerated *primitive type* (F, T) used to control *selection* and as *primitive values*

**const binding** A *binding* of an *attribute* that prevents its being rebound in any contained *scope*

**contents** The *vector* of *values* denoted by a *node* of a *script*

**definition** Another name for a *const binding*

**document** The *rendition* of a *script* in a representation suitable for some editor

**dominant structure** The tree structure of a *document* corresponding to the *node* structure of its *script*

**editor-specific name** A non-standard *name* used by a specific editor in *scripts* it generates; an editor may use editor-specific terms without interfering with the interchangeability of a script if it provides *definitions* of the standard names in terms of its editor-specific names

**elaborate** (verb) To develop the semantics of a *script* or a *node* of a script according to the *Interscript* semantic rules. This is a left-to-right, depth-first processing of the script

**encoding** A particular representation of *scripts*

**environment** A *value* consisting of a set of *attributes*

**expression** A syntactic form denoting a *value*

**external environment** A standard *environment* relative to which an entire *script* is *elaborated*

**fidelity** The extent to which a *transcription* or *rendition* preserves *contents*, form, and structure

**hexInt** A component of an *intSequence* formed from a pair of letters in the set {A,B,. . .,O,P}, representing an *integer* 0 .. 255

**hierarchical name** A *name* containing at least one period, whose prefix unambiguously denotes the naming authority that assigned its meaning

**identifier** A sequence of letters used to identify an *attribute*

**integer** A mathematical integer in a limited range; one of the *primitive types*

**interchange encoding** A standard *encoding* of *scripts*

**Interscript** The current name of this basis for an editable document standard

**intSequence** An abbreviated notation for sequences of small *integers*

**invocation** The appearance of a *name* in an *expression*, except as the *attribute* of a *binding*

**label** A *tag,* or a *source*, a *target*, or a *link introduction* placed in a *node*

**link** The cross product of a *source* and a *target*; in general, a link is a set of (source, target) pairs; in the special case when there is exactly one source and one target, a link behaves like a directed arc between a pair of *nodes*

**link introduction** The appearance of id@! in a *node*, where id is the main *identifier* of a *link*

**literal** A representation of a *value* of a *primitive type* in a *script*

**local binding** A *binding* of a *value* to a *name*, causing the current *environment* to be updated with the new *attribute*; any outer binding's *scope* will resume at the end of the innermost containing *node*

**name** A sequence of *identifiers* internally separated by periods; e.g., a.b.c

**nested environment** The initial *environment* of a *node* contained in another node

**NIL** A name for the empty *value*; it does not lengthen a *vector* or *node* in which it appears

**node** Everything between a matched pair of {}s in a *script*; this generally represents a branch point in a *document's dominant structure*

**Null** Identifies the empty *environment*; the *value* it associates with any *identifier* is *NIL*

**Outer** A standard *attribute* of every *environment,* whose value is the environment just prior to the start of the current *node*

**Outermost** The standard outer *environment* for an entire *script*; the *value* of an *identifier* in Outermost is the *universal* consisting of the same letters in upper case

**persistent binding** A kind of *binding* within the *scope* of a *var binding* that acquires the *scope* of the *var binding*, and hence may endure beyond the end of the innermost containing *node*

**primitive type** *Boolean*, *Integer*, *Real*, *String*, or *Universal*

**primitive value** A *literal* or a *node*, *vector*, or *environment* containing only primitive values

**private encoding** One of a number of non-standard *encodings* of a script

**property** Each *tag* on a *node labels* it with a *property*; the *properties* of a *node* determine how it may be viewed and edited

**quoted expression** A *value* which is an *expression* bracketted by single quotes ("'"); the expression is evaluated in each *environment* in which the *identifier* to which it is *bound* is *invoked*

**real** A floating point number

**rendition** The process of converting from a *script* to a *document*; also the result of that process

**scope** The region of the *script* in which *invocations* of the *attribute* named in a *binding* yield its *value*; the *scope* starts textually at the end of the *binding*, and generally terminates at the end of the innermost containing *node*

**script** An *Interscript* program; the interchangeable result of *transcribing* a *document*

**selection** A conditional form in a *script* that denotes one of two *expressions*, depending on the value of a *Boolean* expression in the current *environment*

**source** The set of *nodes labelled* with *link@*

**string** A *literal* which is a *vector* of characters bracketed by "<>", e.g., <This is a string!>

**style** A *quoted expression* to be invoked in a *node* to modify the node's *environment*, *labels*, or *contents*

**Sub** A standard component of each *environment*, which is invoked to initialize *nested environments*

**SUBSCRIPT** A function that can be used to extract a value from a *vector*, e.g. SUBSCRIPT[(a b <str>), 3] is the value <str>

**tag** A *universal name labelling* a *node* using the syntax universal$; the *properties* of a node correspond to the set of tags labelling it

**target** The set of *nodes labelled* with *link*!

**transcription** The process of converting from a *document* to a *script*; also the result of that process

**transparency** A characteristic of *scripts* that allows an editor to identify the *nodes* of a script that it understands and thereby enables it to operate on those nodes without disturbing the ones that it doesn't understand

**Units** A set of *definitions* relating various typographical and scientifc units to the *Interscript* standard units, meters; e.g., inch=.0254 pt=.013836*inch

**universal** A *name* whose first *identifier* is all uppercase; a universal name can be used at the top level in the *external environment*, e.g., XEROX.fonts.Helvetica

**value** A *primitive value*, *node*, *vector*, *environment*, *universal*, or *quoted expression*

**var binding** A *binding* that is intended to be superseded by *persistent bindings* within its *scope*; useful for maintaining such things as running figure numbers

**vector** An ordered sequence of *values* that may be *subscripted*

## APPENDIX B

### ARBITRARY CHOICES

"One of the primary purposes of a standard
is to be definitive about otherwise arbitrary choices."

There are many places in this proposal where we have made an arbitrary choice for definiteness. It will be important that the ultimate standard make *some* choice on these points; it matters little whether it is the same as ours. To forestall profitless debate on these points, we have tried to list some of the choices that we believe can be easily changed at a later date:

*Encoding choices:*

The choice of representations for literals (we generally followed Interpress here).

The selection of particular characters for particular kinds of bracketting, and for particular operators.

The choice of infix and functional notation for the interchange encoding (as opposed, e.g., to Polish postfix).

The choice of particular identifiers for basic concepts.

*Linguistic choices:*

The choice of a particular set of basic operators for the language.

The particular set of primitive data types (we followed Interpress its set seems about as small as will suffice).

The choice of particular syntactic sugars for common linguistic forms.

# APPENDIX C

## RELATION TO OTHER STANDARDS

## APPENDIX D

## 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 Horning, May 4, 1982 5:16 PM, Fold in Truth Copy changes, add Appendix B

Edited by Mitchell, May 10, 1982 5:40 PM, changed "Interdoc" to "Interscript", "rendering" to "internalizing", and "transcribing" to "externalizing" plus various edits necessitated by these substitutions.