XEROX

Xerox System Integration Standard

INTERSCRIPT

XSIS 000000
March 1984

Xerox Corporation
Stamford, Connecticut 06904

Notice

This *Xerox System Integration Standard* describes Interscript - A Proposal for a Standard for the Interchange of Editable Documents.

1. This standard includes subject matter relating to patent(s) of Xerox Corporation. No license under such patent(s) is granted by implication, estoppel, or otherwise, as a result of publication of this specification.

2. This standard is furnished for informational purposes only. Xerox does not warrant or represent that this standard or any products made in conformance with it will work in the intended manner or be compatible with other products in a network system. Xerox does not assume any responsibility or liability for any errors or inaccuracies that this document may contain, nor have any liabilities or obligations for any damages, including but not limited to special, indirect, or consequential damages, arising out of or in connection with the use of this document in any way.

3. No representations or warranties are made that this specification, or anything made in accordance with it, is or will be free of any proprietary rights of third parties.

Preface

This document is one of a family of publications that describes the network protocols underlying Xerox Network Systems.

Xerox Network Systems comprise a variety of digital processors interconnected by means of a variety of transmission media. System elements communicate both to transmit information between users and to economically share resources. For system elements to communicate with one another, certain standard protocols must be observed.

Comments and suggestions on this document and its use are encouraged. Please address communications to:

Xerox Corporation
Office Systems Division
Network Systems Administration Office
3333 Coyote Hill Road
Palo Alto, California 94304

Table of contents

1 **Introduction** . . . 1

1.1 Rationale for an interchange standard . . . 1
1.2 Properties that any interchange standard must have . . . 2
1.2.1 Encoding efficiency . . . 2
1.2.2 Open-ended representation . . . 2
1.2.3 Document content and form . . . 2
1.2.4 Document structure . . . 3
1.2.5 Transcription fidelity . . . 3
1.2.6 Script comprehension . . . 3
1.2.7 Regeneration . . . 4
1.3 What the Interscript standard does not do . . . 4
1.3.1 Interscript is not a file format . . . 4
1.3.2 Interscript is not a standard for editing . . . 4
1.3.3 Combining documents is not an interchange function . . . 4
1.3.4 Interscript does not overlap with other standards . . . 5
1.4 Concepts and guiding principles . . . 5
1.4.1 Layers . . . 5
1.4.2 Externalization and internalization . . . 5
1.4.3 Content, form, value, and structure . . . 5

2 **The language basis: syntax and semantics** . . . 7

2.1 Grammar . . . 7
2.2 Abstract grammar for Interscript . . . 8
2.3 Formal semantics for Interscript . . . 10
2.3.1 Semantics organized according to the abstract grammar . . . 10
2.3.2 Definitions of semantics as a Mesa function . . . 14
2.3.3 Notes for semantics . . . 20
2.3.4 Equivalence of scripts . . . 20

3 **Tags, node invariants, and safe editing** . . . 21

3.1 Tags, types, and node invariants . . . 21
3.1.1 Defining tags . . . 21
3.1.2 Defining types . . . 22

3.2 The invariant associated with a node . . . 24
3.3 Safety rules for editors . . . 25

4 Standard document . . . 27

4.1 Introduction . . . 27
4.2 Overview . . . 28
4.2.1 Basic text . . . 28
4.2.2 Layout . . . 28
4.2.3 Tag definition . . . 28
4.2.4 Publication encoding . . . 28
4.3 The document as an entity . . . 29
4.4 Boxes and Layout . . . 29
4.4.1 Box fundamentals . . . 29
4.4.2 Measures . . . 30
4.4.3 Box nodes . . . 31
4.4.4 The page node . . . 33
4.5 Naming and labelling noides. . . . 33
4.5.1 Labelling . . . 33
4.5.2 Intra-document references via labels . . . 34
4.5.3 User-sensible naming . . . 35
4.6 Basic text content . . . 35
4.6.1 Characters . . . 36
4.6.2 Special characters . . . 37
4.6.3 Text . . . 37
4.6.4 A text encoding-notation . . . 38
4.6.5 Text fields . . . 38
4.6.6 Text value . . . 39
4.7 Pouring constructions . . . 40
4.7.1 Pouring, templates, and molds . . . 40
4.7.2 Paragraph . . . 42
4.7.2.1 Layout within the paragraph. . . . 43
4.7.2.2 The line node . . . 43
4.7.2.3 Tabs . . . 44
4.7.3 Fill. . . . 44
4.7.4 Value replication . . . 45
4.7.5 Anchors . . . 45
4.7.6 Penalty node . . . 46
4.8 Tables . . . 46
4.9 Inked boses. . . . 49
4.10 Interpress graphics. . . . 49
4.11 Non-Interscript editing. . . . 50
4.12 Styles . . . 50

5 Layout . . . 53

5.1 Introduction . . . 53
5.2 Overview . . . 53
5.3 Boxes and measure arithmetic.. . . . 53
5.4 Layout with boxes: "Solid" layout. . . . 53
5.4.1 Solid page layout. . . . 54

5.4.2 Coordinate system 54
5.4.3 Recursive page layout. 55
5.4.4 Measure synthesis 55
5.4.5 Fixing a box's dimensions and locations. 57
5.5 Layout with templates: "Pouring". 59
5.5.1 Pour node. 59
5.5.2 Pouring streams. 59
5.5.3 Best pours and good pours. 61
5.5.4 Local minima. 62
5.5.5 Fences.. 62
5.5.6 Multi-level pours. 63
5.5.7 Pour "leftovers". 63
5.6 Figures. 64

Appendices 67

B Example of Script Evaluation 67
P Paragraph reducesTo 71
Q Line reducesTo 73

Glossary 75

1

Introduction

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

1.1 Rationale for an interchange standard

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

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

This N(N-1) problem can be reduced to 2(N-1) by noticing that we could write N-1 conversion routines to go from F1 (format for editor1) to F2,. . .,FN, and another N-1 routines to convert from F2,. . .,FN to F1. Except when converting from or to F1, this scheme requires two conversions to go from Fi to Fj (j=i). This is a minor drawback. Choosing which editor should be editor1 is the critical issue, however, since the capabilities of that editor will determine how general a class of documents can be interchanged among the editors.

This presents a truly difficult problem in the case that there is no single functionally dominant editor1 in the set. If the pivotal editor1 doesn't incorporate all of the structures, formats, and content types used by editor2,. . .,editorN, then it will not be possible to faithfully convert documents containing them. Even if there were a single, functionally dominant editor, 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, this standard has been developed by examining, in general, what information editors need and how that

information can be organized to represent general documents. It provides an external representation that is capable of conveying the content, form, and structure of editable documents. That external representation has only one purpose: to enable the interchange of documents among different editors. It must be easy to convert between real editors' formats and this interchange encoding.

When represented by this interchange encoding, we call a document a script and reserve the term document for the representation that an editing system uses to enable editing it. Using a standard interchange encoding has the additional advantage that much of the input and output conversion algorithms will be common to all conforming editors. For example, when a new version of an existing editor is released, the only differences in the new version's conversion routines will be in the areas in which its internal document format has changed from its previous form; this represents a significant saving of programming.

1.2 Properties that any interchange standard must have

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

1.2.1 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 space-efficient.

Similarly, the cost in time 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.

1.2.2 Open-ended representation

Scripts must be capable of describing virtually all editable documents, including those containing formatted text, synthetic graphics, scanned images, animated 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). Thus, a standard must be capable of incremental extension and any extension must have the same guarantees and be able to employ the same mechanisms as the most basic parts of the standard.

For the same reasons, the standard must not be tied to particular hardware or to a file format since documents will be stored and transmitted using a variety of media.

1.2.3 Document content and form

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

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

1.2.4 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. This standard exploits 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.

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 Transcription fidelity

It must be possible to convert any document from any editor's private format to a script and reconvert it back to the same editor's private format with no observable effect on the document's content, form, or structure. This characteristic is called transcription fidelity, and is a sine qua non for an interchange encoding; if it is not possible to accomplish this, the interchange encoding or the conversion routines (or both) must be defective. It must, of course, also be possible to test that an editor does transcribe scripts faithfully, which in turn requires that it be possible to test if two scripts are equivalent (section 2.3.4).

1.2.6 Script comprehension

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

At a minimum, this requires that a script contain information by which an editor can easily determine whether or not it understands a component well enough to edit it, and that it be able to interpret the effect that components which it does not understand have on the ones it does. For example, if an editor does not understand figures, it might 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.

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

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

1.3 What the Interscript standard does not do

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

1.3.1 Interscript is not a file format

This standard is not concerned with how scripts are held in files on various media (floppy disks, hard disks, tapes, etc.), or with how they are transmitted over communications media (local area network, 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: that is the purview of editors and their associated private document formats. A script is intended to be internalized before being edited. This might be done by the editor, by a utility program on the editing workstation, or by a completely separate service.

1.3.3 Combining documents is not an interchange function

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

1.3.4 Interscript does not overlap with other standards

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

1.4 Concepts and guiding principles

1.4.1 Layers

The Interscript standard is presented in layers:

Layer 0 defines the *syntax of the base language* for scripts; parsing reveals the dominant structure of the documents they represent (sections 2.1-2.2).

Layer 1 defines the *semantics of the base language*, particularly the treatment of bindings and environments (section 2.3, chapter 3).

Layer 2 defines the *semantics of properties and attributes* that are expected to have a uniform interpretation across all editors (chapters 4-5).

1.4.2 Externalization and internalization

A *script* represents a document in the Interscript format. Its sole purpose is to enable the interchange of documents among editors in a manner that is independent of any one editor.

A script is *not* the editable form of a document. The editable form is created by an editor by *internalizing* a script according to the rules (semantics) of Interscript. The reverse operation of converting a document in an editor's internal, editable format to a valid script is called *externalization*.

It is important that any document prepared by any editor can be externalized as a script that will then be (re)internalized by the editor without "loss of information". Ease of internalization requires that the Interscript base language contain only relatively few (and simple) constructs. This apparent paradox has been resolved 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 is "compiled" to convert the script to the private representation of a particular editor, ready for further editing. The Interscript language has been designed so that internalizing scripts into typical editors' representations can be performed in a single pass over the script by maintaining a few simple data structures.

1.4.3 Content, form, value, and structure

Most editors deal with both the *content* of a document (or piece of a document), and its *form*. The former is thought of as "what" is in the document, the latter as "how" it is to be

viewed; e.g., "ABC" has a sequence of character codes as its contents; its format may include font and position information. Interscript maintains this distinction.

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

Content: may be represented by structures built from character strings, numbers, Booleans, identifiers, and nodes, which are structured objects containing simpler ones.

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

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

The most important, and most frequent, structuring mechanism between values is logical adjacency (sequentiality), which is represented by simply putting them one after another in the script.

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 primary tree structure and used to control the inheritance of attributes.

Structures recorded for form use explicit *indirection* by means of names. Interscript allows expressions composed of literals, identifiers, and operators, and permits the use of identifiers to represent expressions.

2

The language basis: syntax and semantics

This chapter defines the Interscript Base Language. Two versions of its syntax, a concrete one and an abstract one, are supplied. The concrete grammar defines the publication encoding for Interscript, which is solely intended for communicating Interscript concepts among people and is used for all examples of Interscript in this standard (chapter 7 defines the actual encoding to be used for representing scripts for editing systems rather than humans).

Section 2.3 defines the semantics of the base language. These semantics have two primary functions: (1) to provide a rigorous definition for equivalence of scripts, and (2) to show conceptually what information in any script an Interscript implementation must represent in order to be able to internalize and externalize a script correctly.

2.1 Grammar

Our notation is basically BNF with terminals quoted and parentheses used for grouping.

```
script      ::= header node trailer
header      ::= "INTERSCRIPT/INTERCHANGE/1.0 "
trailer     ::= "ENDSCRIPT"
node        ::= "{" items "}"
items       ::= empty | items item
item        ::= tag | indirection | binding | sBinding | term | openedNode | scope
tag         ::= primary "$"
openedNode  ::= (term | name "%") "|"
scope       ::= "[" items "]"
binding     ::= name "←" term
sBinding    ::= name "%←" ( term | indirection | "'" term "'" )
term        ::= primary | term op primary
op          ::= " + " | "-" | "*" | "/" | "!" | "LT" | "EQ"
primary     ::= literal | invocation | node | "(" term ")"
literal     ::= name | number | string
name        ::= id | name "." id
invocation  ::= primary " ↑ "
indirection ::= name "%"
```

This small grammar defines a representation language for scripts that is small enough to admit of a precise denotational semantics (section 2.3), yet powerful enough to include facilities for strong typing and data-type extensibility.

This surface syntax is intended only for use as a means of communications about Interscript. It is a *publication encoding* and is not necessarily intended as the actual encoding to be used for interchanging documents among editingsystems. Any such encoding should pay attention both to the structure of this language and to the requirements for an easily decoded, robust, and space-efficient representation.

2.2 Abstract grammar for Interscript

To associate a precise semantics with this language, it is easier to work from an abstract syntax for it, ignoring surface punctuation and treating the parsed form of some external encoding of a script. The following grammar serves this purpose as well as indicating the reason for each of the parts of the language (in small print, like this). However, the small print should be viewed as hints about the semantics; in any conflict between the small print and the actual semantics, the semantics is to be considered the correct version:

script ::= node

An entire script is represented as a single high level node with subnodes to carry the various parts of the logical and layout structures.

node ::= items

items ::= empty | items item

item ::= tag | binding | sBinding | term | indirection | openedNode | scope

A node is the primary structured data type of the language and is used to represent both logical and layout structures. The items that make up a node may be tags, values, bindings, and expressions giving rise to tags, values, and bindings. A node with values but no bindings behaves like a vector; a node with bindings but no values behaves like a record; values and bindings may be intermixed to capture both the content of a piece of a document and the bindings associated with that content (such as what font a character is to be displayed in).

tag ::= primary

A node can have zero or more tags. Each tag is associated with an invariant that all nodes with that tag must satisfy. This invariant includes what attributes (bindings) are relevant to the node, what types of contents are allowed for the node, and any relations that must hold among the attributes of the node. For example, a TEXT node is only allowed to have character strings or PSEUDOCHAR nodes as contents, and font is a relevant attribute of a TEXT node. An editor can tell whether or not it has code to deal explicitly with a node simply by looking at the node's tags.

binding ::= name term

A binding associates a value (the result of elaborating the term) with a name. As bindings are encountered during internalization, they are appended to the current environment. Whenever the value associated with

a name is needed, the environment is scanned from most-recent to least-recent for a binding to that name (of course, this is a logical description, not necessarily how one would implement environments).

sBinding :: = name sRhs

A structured binding is just like a normal binding except that the binding must be kept with the node in which it occurs, even if it is not a relevant attribute of (a tag of) that node. Only names that are structurally bound can be used in an indirection (see below) or can be opened (see openedNode below). An intended use of sBindings is for associating style information with a name so that it can be applied to nodes while still maintaining the inherent indirection that would allow the style to be changed and have the effect of the change propagate to all the nodes using it.

sRhs :: = term | indirection | quoted

A structured binding can associate a single value, another name, or a quoted expression with a name. to get the effect of structurally binding a set of items to a name, one can bind a node value to the name and then access the items using the openedNode mechanism; e.g., list %← {1 1 2 3 5 8} list%| will result in the sequence 1 1 2 3 5 8 becoming items at the same level as the list%|.

quoted :: = term

A quoted expression must be a syntactically correct expression (this is not a macro facility!). It will be evaluated in the environment that obtains when used in an indirection (see below).

term :: = primary | term op primary

op :: = ADD | SUB | MUL | DIV | SUBSCRIPT | LT | EQ

primary :: = literal | invocation | node | term

Interscript expressions are primitive with a minimal spanning set of operators. Since scripts are expected to be written and read by editing systems rather than by people, this is adequate. An invocation replaces an occurrence of a name by its value. A node is a value like any other.

literal :: = name | number | string

A name standing by itself is atomic: it simply stands for itself, nothing more (an invocation is used if the name is to be evaluated to produce a value). Numbers may be either Fortran-like reals or integers. Strings are characters in the ISO 646 subset bracketted by the double quote character (").

name :: = id | name id

A name is either a simple identifier (e.g., a, b, simpleId) or it is a qualified name (e.g., a.position.x, font.face.style). If the name is to be evaluated (for an invocation, an indirection, or an openedNode), then it is evaluated from left to right. All the identifiers except the last in a name must evaluate to node values, which will then be treated as environments in which to continue the name evaluation process. For example, in a.position.x, a must be bound to a node value and must contain a binding for position, which must also be a node value containing in turn a binding for x.

invocation :: = primary

When a name is invoked, it is evaluated as described under the name rule above, and the resultant value is used in place of the invocation.

indirection :: = name

An indirection behaves just like an invocation for the purposes of evaluation. In addition, the fact that the resulting value came from an indirection must be remembered in case of changes to the binding for the name or for correct externalization of the document as a script.

openedNode :: = primary | name

A node value can be bound to a name and then later "opened" in some context to place the items of the node into the environment in which it is opened. This can be used to introduce default values for tags, values, or bindings anywhere in a script. If the form name%| is used instead of name ↑ |, then the semantics demand that the name of the node opened be maintained as well as the result of opening it. Such a *structured open* can be used to associate styles with parts of a document while guaranteeing that any change to the style can be reflected in all the places that the style is used.

scope :: = items

A list of items can be put in scope brackets "[...]" to ensure that any computation in the scope cannot affect the surrounding environment except by maintaining any bindings that are structural (see sBinding). Any contents that result from evaluating the scope are considered part of the surrounding environment (just as if they had not been surrounded by scope brackets).

2.3 Formal semantics for Interscript

The semantics for the base language are defined by a function, S, whose domain is the set of parse trees corresponding to the abstract grammar of section 2.2. S is written in a variant of the Mesa programming language [Mesa] which has had added to it the ability to define and manipulate lists of values (provided all the elements of a given list have the same type in the Mesa sense).

A type describing a list of items of type *T* can be defined as "LIST OF *T*" (in these semantics, *T* will always take the form "REF *T'*" for some type *T'*). A list element can be created by the operation

CONS: PROC[first, rest: LIST OF *T*] RETURNS[LIST OF *T*]

where *T* may be any type. Given a list *L*, the first element can be accessed as *L*.first, and the tail of the list as *L*.rest.

2.3.1 Semantics organized according to the abstract grammar

The semantics of the base language are described by the function S:

S: PROC[ε: *Env, nt: NonTerminal*] RETURNS[vs: Vs]

where a NonTerminal represents a node of an abstract parse tree (not defined here).

```
Vs: TYPE = LIST OF V

V: TYPE = REF VRec
```

Conceptually, a *VRec* is a triple consisting of a mark, *m*, some contents, *c*, and sometimes extra information, *x*, which is primarily for recording a name (e.g., for a binding). To get the maximum benefit of type checking the definition of *S* without encumbering the definition with a lot of extraneous type coercions, a *Vrec* is defined to contain a set of fields, *c*α, one for each type of content (*c*α is one of *cV*, *cVs*, *cNum*, *cStr*, *cId*, *cTerm*). For similar reasons, the *x* component may be one of *xId*, *xName*, or *xEnv*. In any given *Vrec* at most one of the *c* fields and at most one of the *x* fields will actually be used; e.g., a tuple representing the number 3.14 has the form *VRec[m: num, cNum:* 3.14], and a tuple representing the name *a.b.c* as an atom has the form *VRec[m: atom, xName*: CONS[c, CONS[b, CONS[a, NIL]]]]. Here are the complete definitions of V*R*ec and *Mark*:

```
VRec: TYPE = RECORD[m: Mark,
    cVs: Vs←NIL, cV: V←NIL, cNum: Number←0.0, cStr: STRING←NIL, cId: Id←NIL,
    cTerm: NonTerminal←NIL,
    xId: Id←NIL, xName: Name←NIL, xEnv: Env←NIL];

Mark: TYPE = {atom, num, string, node,      -- base values--
    tag, bind, scope, bindStruc, evalStruc, onodeStruc, quotedTerm, vOfQ,
    evalSentinel};
```

S is defined in terms of various subsidiary semantic functions, which divide naturally into two groups: those that require an environment as a parameter and those that do not. The type signatures of the functions in the two groups are

(1) functions with an environment parameter:

```
MkNode: PROC[ε: Env, vs: Vs] RETURNS [V -- node--]
MkBinding: PROC[ε: Env, v: V, n: Name, kind: Mark] RETURNS [V]
Lookup: PROC[ε: Env, n: Name] RETURNS [V]
EvalName: PROC[ε: Env, n: Name] RETURNS [V]
RelBindings: PROC[ε: Env, vs: Vs --tag items only--] RETURNS [Vs]
```

(2) functions independent of environment (dealing only with tuples or lists of tuples):

```
Apply: PROC[op: Op, v1: V, v2: V] RETURNS [result: V]
Bindings: PROC[v: V] RETURNS [Env]
Contents: PROC[vs: Vs] RETURNS [Vs]
Tags: PROC[vs: Vs] RETURNS [Vs --tag items only--
Items: PROC[v: V] RETURNS [Vs]
MkScope: PROC[vs: Vs] RETURNS [Vs]
```

S is written in a "distributed" manner in the table below. It associates the parts of *S* with non-terminals of the abstract grammar. If *S* were written as a single function, it would look like a large case statement with one arm for each non-terminal of the abstract grammar and the lines under the column heading ***S[ε, Alternative]*** as the code for the arms of the case statement.

ε denotes the *current environment* argument of S. An environment is simply a sequence of bindings just like those that are relevant to a node. Some of the semantic rules pass a modified ε to recursive application of S; see, e.g., the rule defining *items*. An environment can be augmented by one or more bindings, ***bind****, denoted by **ConcatVs[*bind**, ε]**.

Only the semantics for "interesting" alternatives of the grammar are given below. For any alternative ***lhs*** **::=** ***rhs*** whose semantics are not presented, its *value semantics* are **S[ε, *lhs*] = S[ε, *rhs*]**.

When S maps a nonterminal of the abstract grammar into its tuple form, it is said to be *elaborated*.

Lefthand Side	Alternative	S[ε, Alternative]
script	::= node	S[X, node]

The environment for the root node of an abstracted script is the external environment X.

```
node        ::= items       {t: Vs = S[ε, items]; RETURN[
                             MkNode[ConcatVs[Bindings[t], ε], t]
```

Make a node from the list of items using the environment obtained by appending bindings in the node with the current environment.

```
items       ::= empty       empty
              | items item  {t: VS = S[ε, items]; RETURN[
                             ConcatVs[t, S[ConcatVs[Bindings[t], ε], item]}
```

Recursively elaborate items and then elaborate the last item using the environment formed by appending the bindings in the preceding items to the current environment. Thus bindings affect things to their right in a list of items.

```
tag         ::= primary     MkTag[ε, S[ε, primary]]
```

Elaborate the primary. That should produce a (possibly qualified) name to use as the tag.

```
scope       ::= items       MkScope[S[ε, items]]   -- see MkScope below--

openedNode  ::= term         Items[S[ε, term]]
              | name         MkV[m: onodeStruc,
                             cVs: Items[EvalName[ε, S[NIL, name]]], name]
```

If a term (non-structural openedNode), then use the Items function to pull all the necessary elaborated items from the node value that the term must elaborate to. If a structural openedNode, embed the elaborated

items of the named node in a tuple marked as an onodeStruc in order to remember the name from which the items were obtained.

binding ::= name term **MkBinding[ε, S[ε, term], S[NIL, name], bind]**

Elaborate the term in the current environment, and make a (nonstructural) binding between the name and the resultant value. See MkBinding for details, especially for how values are bound to qualified names.

sBinding ::= name sRhs **MkBinding[ε, S[ε, sRhs], S[NIL, name], bindStruc]**

Elaborate the term in the current environment, and make a structural binding between the name and the resultant value. See MkBinding for details, especially for how values are bound to qualified names.

quoted ::= term **Mk_V[m: *quotedTerm*, cTerm: term]**

Simply make a *quotedTerm* tuple to hold the term for later elaboration.

term ::= term op primary **apply[op, S[ε, term], S[ε, primary]]**

Elaborate the term, then the primary, and then apply the op to the result.

literal ::= name **Mk_V[m: atom, xName: S[NIL, name]]**

| number **Mk_V[m: num, cNum: number]**

| string **Mk_V[m: string, cStr: string]**

A name as a literal is just made into an atom tuple. A numeric literal is stored as a num tuple. And, a string literal is stored as a string tuple.

name ::= id **MkAtom[id]**

| name id **CONS[MkAtom[id], S[NIL, name]]**

A qualified name is mapped into a list with the most highly qualified identifier first and the head, unqualified identifier last.

invocation ::= primary **EvalName[ε, S[ε, primary].xName]**

Elaborate the primary to obtain a name to invoke (stored as the xName component of the resultant tuple). Then use EvalName to lookup and evaluate the name.

indirection ::= name **Mk_V[m: evalStruc, c*V*: EvalName[ε, S[NIL, name]],**

xName: S[NIL, name]]

Use EvalName to lookup and evaluate the name and embed the resultant value in an evalStruc tuple along with the name from which it came.

2.3.2 Definitions of semantics as a Mesa function

Groundrule: Once constructed, no values or lists of values are ever modified (except in Lookup). Therefore, they can be handed around without regard for being changed in situ later.

```
S: PROC[ε: Env, nt: NonTerminal] RETURNS[vs: Vs] = {--TBD--};

NonTerminal: TYPE = REF --ParseNode--;

Error: ERROR[ec: {BoundsFault, InvalidTag, NotSingleton, UnboundId, WrongType}] =
CODE;

*******VALUES*******

V: TYPE = REF VRec;
Number: TYPE = REAL;
VRec: TYPE = RECORD[m: Mark,
    cVs: Vs←NIL, cV: V←NIL, cNum: Number←0.0, cStr: STRING←NIL, cId: Id←NIL, -- Contents
    cTerm: NonTerminal←NIL,          -- used for representing quoted terms--
    xId: Id←NIL, xName: Name←NIL, xEnv: Env←NIL];  -- eXtra information

Mark: TYPE = {atom, num, string, node,     -- base values--
    tag, bind, scope, bindStruc, evalStruc, onodeStruc, quotedTerm, vOfQ,
    evalSentinel};

MarkAsAtom: ARRAY Mark OF ATOM = [$atom, $num, $string, $node, $tag, $bind, $scope,
    $bindStruc, $evalStruc, $onodeStruc, $quotedTerm, $vOfQ, $evalSentinel];
Vs, Env: TYPE = LIST OF V;          -- for an Env, all items have mark bind or bindStruc--
X: Env ← NIL;          -- stand-in for the eXternal environment--
MkV: PROC[m: Mark, cVs: Vs←NIL, cV: V←NIL, cNum: Number←0.0, cStr: STRING←NIL,
    cId: Id←NIL, cTerm: NonTerminal←NIL, xId: Id←NIL, xName: Name←NIL, xEnv:
    Env←NIL] RETURNS[V] = {

        -- The basic function for making a tuple--

    RETURN[NEW[VRec ← [m: m, cVs: cVs, cV: cV, cNum: cNum, cStr: cStr, cId: cId, cTerm:
    cTerm, xId: xId, xName: xName, xEnv: xEnv]]]};

True: V = MkV[m: num, cNum: 1.0]; False: V = MkV[m: num, cNum: 0.0];

        --TRUE and FALSE as tuples--

MkVs: PROC[v1, v2, v3: V←NIL] RETURNS [vs: Vs] = {

        --Make a list of Vs from individual Vs--

    vs←NIL; IF v3#NIL THEN vs←CONS[v3, vs];
    IF v2#NIL THEN vs←CONS[v2, vs]; IF v1#NIL THEN vs←CONS[v1, vs] };

ConcatVs: PROC[v1, v2, v3: Vs←NIL] RETURNS [vs: Vs] = {
```

```
        --Concatenate a number of individual lists into one list (in order v1, v2, v3)--

    AppendToT: PROC[list: Vs] = {
        UNTIL list = NIL DO t.rest ← CONS[list.first, NIL]; t ← t.rest; list ← list.rest ENDLOOP};

    t: Vs ← (vs←CONS[NIL, NIL]);
    AppendToT[v1]; AppendToT[v2]; AppendToT[v3]};

NthItem: PROC[vs: Vs, n: INT] RETURNS [V] = {     -- indexing into a list --
    IF vs = NIL OR n<0 THEN Error[BoundsFault];
    RETURN[ IF n = 0 THEN vs.first ELSE NthItem[vs.rest, n-1] ]};

Name, Ids: TYPE = LIST OF Id; Id: TYPE = ATOM;

        -- A Name is never empty: it always comes from the parser --

MkName: PROC[a: ATOM, hd: Name←NIL] RETURNS[Name] = {RETURN[CONS[a, hd]]};

EqName: PROC[n1, n2: Name] RETURNS[BOOL] = {
    UNTIL n1 = NIL OR n2 = NIL DO
        IF n1.first#n2.first THEN RETURN[FALSE];
        n1←n1.rest; n2←n2.rest;
        ENDLOOP;
    RETURN[n1 = n2]};

*******REAL WORK*******

MkScope: PROC[vs: Vs] RETURNS [Vs] = {RETURN[

        Unless a scope contains some structural items, its contents simply become contents
        of the surrounding node.

    (IF HasStruc[vs] THEN MkVs[MkV[m: scope, cVs: Contents[vs]]] ELSE Contents[vs])]};

    HasStruc: PROC[vs: Vs] RETURNS [BOOL] = {

        Tests if a list of Vs has any structural items (binds, indirections, or structurally
        opened node).

        One: PROC[v: V] RETURNS [BOOL] = {
            RETURN[v.m = bindStruc OR v.m = evalStruc OR v.m = onodeStruc OR
            v.m = quotedTerm]};

        RETURN[ IF vs = NIL THEN FALSE ELSE (One[vs.first] OR HasStruc[vs.rest]) ]};

MkTag: PROC[ε: Env, n: Name] RETURNS [V -- tag --] = {

        For every valid tag in a script there must be a tag definition (section 3.1.1). That
        definition is given as a (structural) binding of a TAG$ node to the tag's name.
        capture that definition in the tuple representing the tag along with its name
        (remember these are value semantics, not an implementation: a real
        implementation would avoid copying the definition and point to it somehow).
```

```
    tagDef: -- node --V = EvalName[ε, n];
    IF NOT HasTag[tagDef.cVs, STAG] THEN Error[InvalidTag];
    RETURN[MkV[m: tag, cV: tagDef, xName: n]]};

MkNode: PROC[ε: Env, vs: Vs] RETURNS [V -- node --] = { -- [note 1] --
```

The canonical form for a node is a list of all its tags followed by its contents, followed by a sequence of the relevant bindings for each of the tags.

```
    RETURN[MkV[node, ConcatVs[Tags[vs], Contents[vs], RelBindings[ε, Tags[vs]]]]]};
```

The semantic procedures MkBinding, Lookup, EvalName, and Eval are a closely related set and provide the central definition of how names are used in Interscript and how bindings are handled.

```
MkBinding: PROC[ε: Env, v: V, n: Name, kind: Mark -- bind/bindStruc --]
RETURNS [-- bind/bindStruc -- V] = {
```

The only difference between a ←and a %← is that the tuple is marked as such.

```
IF n.rest = NIL THEN RETURN[MkV[m: kind, cV: v, xId: n.first]]
```

A binding to a simple id results in a simple tuple being made.

```
ELSE {t: V = EvalName[ε, n.rest]; IF t.m#node THEN Error[WrongType];
    RETURN[MkBinding[ε, MkNode[ε, ConcatVs[t.cVs, MkVs[MkV[m: kind, cV: v, xId:
        n.first]]]], n.rest, kind]]};
```

A binding to a qualified name is handled by copying the binding corresponding to the prefix of the name (it will be bound to a node value) with a new binding tuple added to the end of the node value. Any future use of EvalName for the name will find that binding first.

```
};

Lookup: PROC[ε: Env, n: Name] RETURNS [V] = {
```

looking up a simple identifier in an environment is a simple process of marching down the environment list until a binding to that identifier is reached or the end of the list is encountered.

```
    One: PROC[ε: Env, id: Id] RETURNS [V] = {    -- local procedure to handle simple ids --
        b: V ← IF e = NIL THEN NIL ELSE IF ε.first.xId = id THEN ε.first ELSE One[ε.rest, id];
        IF ε.first.m = evalSentinel AND b#NIL THEN ε.first.xEnv ←CONS[b, ε.first.xEnv];
```

see note in Eval to explain why the above statement hangs bindings off sentinels put into environments

```
        RETURN[b.cV]};
```

Looking up a qualified name requires that we first evaluate the prefix of the name without its last identifier using EvalName (which must yield a node value), then make an environment from the bindings of that prefix in which to look up the suffix identifier.

```
    RETURN[ IF n = NIL THEN NIL ELSE One[Bindings[EvalName[ε, n.rest]], n.first] ]};

EvalName: PROC[ε: Env, n: Name] RETURNS [V] = {

    Lookup the name and evaluate it (it might be a quoted term)

    RETURN[Eval[ε, Lookup[ε, n]]]};
    Eval: PROC[ε: Env, v: V] RETURNS [V] = {
        IF v.m = quotedTerm THEN {
            ePlus: Env = CONS[Mk_V[m: evalSentinel, xId: NIL], ε];

                We push an evalSentinel tuple on the top of the environment so that lookup
                can collect all the bindings needed to evaluate this quotedTerm. The
                environment that ends up hanging off the evalSentinel tuple is then tucked
                away in the vOfQ tuple that results from evaluating the quoted term. This
                information is needed to correctly externalize an indirection that caused a
                quoted term to be evaluated. If the bindings in the xEnv component of the
                vOfQ tuple match those extant when the document is externalized, then the
                editor can simply output the indirection that gave rise to the vOfQ tuple safe
                in the assurance that its value is unchanged.

            RETURN Mk_V[m: vOfQ, cV: S[ePlus, v.cTerm].first, xEnv: ePlus.first.xEnv]}
        ELSE RETURN[BaseVal[v]]};

BaseVal: PROC[v: V] RETURNS [V] = { RETURN[
    (IF v.m = evalStruc THEN BaseVal[v.cV.cV -- it must be a vOfQ--] ELSE v)]};

Op: TYPE = {ADD, SUB, MUL, DIV, LT, EQ, SUBSCRIPT};

IsInteger: PROC[v: V] RETURNS[BOOL] = { -- check that a number is actually an integer--
    IF v.m#num THEN Error[WrongType];
    RETURN[Real.Float[Real.Fix[v.cNum]] = v.cNum]};

Apply: PROC[op: Op, v1: V, v2: V] RETURNS [result: V] = {

    --apply the operator to the base values for v1 and v2 (i.e., evaluate any quoted terms
        that you encounter)--
    vv1: V = BaseVal[v1]; vv2: V = BaseVal[v2];
    SELECT op FROM
        ADD, SUB, MUL, DIV, LT = > {IF vv1.m#num OR vv2.m#num THEN Error[WrongType];
            RETURN[(SELECT op FROM

                ADD = > Mk_V[m: num, cNum: (vv1.cNum + vv2.cNum)],
                SUB = > Mk_V[m: num, cNum: (vv1.cNum-vv2.cNum)],
                MUL = > Mk_V[m: num, cNum: (vv1.cNum*vv2.cNum)],
                DIV = > Mk_V[m: num, cNum: (vv1.cNum/vv2.cNum)],
                LT  = > (IF vv1.cNum<vv2.cNum THEN True ELSE False),
```

```
            ENDCASE => NIL)]};
            EQ  => {IF v1.m#v2.m THEN RETURN[False];

                --Tuple marks have to be the same if they are to be equal --

                SELECT vv1.m FROM
                    node => RETURN[False];  -- two separate node values are never equal --
                    atom => RETURN[IF (vv1.cId = vv2.cId) THEN True ELSE False];
                    num => RETURN[IF (vv1.cNum = vv2.cNum) THEN True ELSE False];
                    string => RETURN[IF String.EqualString[vv1.cStr, vv2.cStr] THEN True
                        ELSE False];
                    ENDCASE => Error[WrongType]};
            SUBSCRIPT => -- use v2 as an index into the node v1--
                IF NOT IsInteger[vv2] OR vv1.m#node THEN Error[WrongType]
                ELSE RETURN[NthItem[Contents[Items[vv1]], Real.Fix[vv2.cNum]]];
            ENDCASE};

Tags: PROC[vs: Vs] RETURNS [Vs -- tag items only --] = {RETURN[(Sort[GetTags[vs]])]};
    GetTags: PROC[vs: Vs] RETURNS [Vs --tag items only--] = {
        One: PROC[v: V] RETURNS[Vs] = {RETURN[(IF v.m = tag THEN MkVs[v] ELSE NIL)]};
        RETURN[ IF vs = NIL THEN NIL ELSE ConcatVs[One[vs.first], GetTags[vs.rest]] ]};

    Sort: PROC[vs: Vs -- tag items only --] RETURNS [sVs: Vs -- tag items only --] = {-- [note 2]
    --};

Items: PROC[v: V] RETURNS [Vs] = {
    IF v.m = node THEN RETURN[RawItems[v.cVs]] ELSE Error[WrongType]};
    RawItems: PROC[vs: Vs] RETURNS[Vs] = {

        Pull all the basic items out of a list, burrowing into evalStruc, onodeStruc, or
        scope tuples to extract their contained items

        One: PROC[v: V] RETURNS[Vs] = {

            Local procedure to look at a single tuple and pull items out of it if it is an
            evalStruc, onodeStruc, or a scope. Otherwise the value is returned.

        RETURN[(SELECT v.m FROM
            evalStruc, onodeStruc => v.cVs,
            scope => RawItems[v.cVs],
            ENDCASE => MkVs[v])]};
    RETURN[IF vs = NIL THEN NIL ELSE ConcatVs[One[vs.first], RawItems[vs.rest]]]};

Bindings: PROC[v: V] RETURNS [Env] = {

    Make an environment from the bindings in the node v.

    IF v.m = node THEN RETURN[GetBindings[Items[v]]] ELSE Error[WrongType]};
    GetBindings: PROC[vs: Vs] RETURNS[ε: Env] = {
    One: PROC[v: V] RETURNS[ε: Env] = {     -- local procedure for catching bindings--
        RETURN[IF v.m = bind OR v.m = bindStruc THEN MkVs[v] ELSE NIL]};
    RETURN[IF vs = NIL THEN NIL ELSE ConcatVs[GetBindings[vs.rest], One[vs.first]]]};
```

```
RelBindings: PROC[ε: Env, vs: Vs -- tag items only --] RETURNS [Vs] = {

    --Find all the relevant bindings for the tags in vs and make an environment from
    them.--

  One: PROC[ε: Env, v: V] RETURNS [Vs] = {
    IF v.m#tag THEN Error[WrongType];
    RETURN[RefineBindings[ε, PullDefaults[Bindings[
          EvalName[ε, MkName[$attributes, v.xName]]]]]]};

  RETURN[IF vs = NIL THEN NIL ELSE ConcatVs[One[ε, vs.first], RelBindings[ε, vs.rest]]]};

RefineBindings: PROC[ε: Env, basis: Vs] RETURNS [Vs] = {

    For every identifier bound non-structurally in basis, place a binding in the result
    Vs. If there is a binding in the environment for the identifier use that; if not, use the
    one for it in basis (which provides a default value).

  One: PROC[ε: Env, b: V] RETURNS [Vs] = {

    IF b.m#bind AND b.m#bindStruc THEN Error[WrongType];
    RETURN[IF b.m = bindStruc THEN NIL
      ELSE MkVs[MkV[
          m: bind, cV: Lookup[ConcatVs[ε, MkVs[b]], MkName[b.xId]], xId:
          b.xId]]]};

  RETURN[IF basis = NIL THEN NIL
    ELSE ConcatVs[RefineBindings[ε, basis.rest], One[ε, basis.first]]]};

PullDefaults: PROC[ε: Env] RETURNS [Vs] = {

    For every binding in the environment return a binding that gives a default value for
    that identifier. Every identifier in the incoming environment is bound to a TYPE$
    node (because relBindings reaches into the attributes attribute for a TAG$ node to
    get these type definitions). The result Vs is not an environment, but is in the same
    left-to-right order as the bindings occuring in the attributes attribute for the TAG$
    node from which they came.

  One: PROC[b: V] RETURNS [Env] = {
    IF b.cV.m = node AND HasTag[Items[b.cV], $TYPE] THEN
      RETURN[MkVs[MkV
          [m: bind, cV: Lookup[Bindings[b.cV], MkName[$default]], xId: b.xId]]]

    ELSE Error[WrongType]};

  RETURN[IF ε = NIL THEN NIL ELSE ConcatVs[PullDefaults[ε.rest], One[ε.first]]]};
  HasTag: PROC[vs: Vs, id: Id] RETURNS[BOOL] = {

      Checks whether a given id occurs in the list of tags.

    RETURN[ IF vs.first.m = tag AND vs.first.cId = id THEN TRUE ELSE HasTag[vs.rest, id] ]};
```

```
Contents: PROC[vs: Vs] RETURNS [Vs] = {
```

The contents of a node are all the tuples in it except tags and non-structural bindings (it thus includes all other values, structural bindings, structured openedNodes, and indirections).

```
    One: PROC[v: V] RETURNS [Vs] = {
        RETURN[(IF v.m = tag OR v.m = bind THEN NIL ELSE Mk_Vs[v])]};
    RETURN[ IF vs = NIL THEN NIL ELSE ConcatVs[One[vs.first], Contents[vs.rest]] ]};
```

2.3.3 Notes for semantics

The semantics deal almost exclusively in immutable values, i.e., values which, once created are never changed. As a result, it copies values prodigiously. Any implementation of Interscript semantics should attempt to use pointers wherever possible to avoid extraneous copies and copying of values.

[1]: All the relevant attributes for each tag are listed at the end of a node, even if the node's tags' relevant attribute sets are not disjoint.

To place some bindings in a node without regard for using a specific node type indicated by tags, each binding must be a structural binding; e.g.,

```
abbrev ← {a%←5 b%←7}

...

{...abbrev^|...}
```

[2]: the only requirements on SORT are that it sort names correctly and eliminate any duplicates. More precisely, if N* is a list of names to be sorted and S* is the result of sorting N* and removing duplicates, then

$$(\forall i \in [0..card(S^*)-1)\ s_i < s_{i+1}) \wedge (\forall i \in [0..card(N^*))\ \exists s \in S^*\ \ni n_i = s)$$

2.3.4 Equivalence of scripts

A script s_1 is equivalent to a script s_2 if and only if $S[X,s_1] = S[X,s_2]$.

One way to test Interscript-conforming editors might be to develop a set of test scripts, and, for each script in the set, check it for equivalence with a version of the script obtained by internalizing and then externalizing the script using the editor in question.

3

Tags, node invariants, and safe editing

Interscript makes it possible for editors to manipulate the parts of documents they understand without altering parts they do not. This section describes the facilities that make this possible and a set of safety rules that will enable editors to preserve this property.

The first part of this section gives the details of how node tags are "defined" in the standard. The second part describes the invariant associated with a node and how it is specified. And the third part shows how a tag's definition can be used by an editor to treat nodes with that tag safely even if the editor does not specifically implement the tag.

3.1 Tags, types, and node invariants

This section uses the Interscript base language syntax and semantics to define precisely the properties of tags, which is accomplished by binding a TAG node to the tag's name. To make a tag definition precise, we will introduce the notion of *type*, which is not embedded in the base language, but rather is defined using it. Then it will actually be possible to define (recursively) the properties of the tag TAG, including the notion of invariants for nodes, which can be used by an editor to test whether an edit is legal, i.e., invariant-preserving.

3.1.1 Defining tags

For every tag *T* used in a script there must be a corresponding definition of the important properties of that tag. This is accomplished by (structurally) binding it to a TAG node, which is used to capture some of *T*'s properties so as to enable editors to operate on T nodes without necessarily having knowledge of *T* built into the editor. A TAG node defining *T* will have the following relevant attributes:

attributes: a TYPE node (see definition of TYPE below) with each relevant attribute of *T* bound to a TYPE node containing the *default* value for the relevant attribute, and its *type*, which is a predicate on the attribute's value in a given node. If no attributes are specified, the default is that the tag has no relevant attributes.

contentType: a TYPE node value specifying the type(s) of the contents allowed in a *T* node. The default type is *Any* (see definition of TYPE below).

nodeInvariant: a predicate expressible as an Interscript (quoted) expression that must be true of any *T* node. Its default value is *True*.

hasMoreInv: a Boolean indicating whether there is more invariant for *T* than can be expressed in *T*'s TAG node; if *hasMoreInv* is *False* then *T*'s invariant is completely captured by the information in its TAG definition. Its default value is *True*.

requiredTags: an untagged node listing as atoms all the tags other than *T* that are required to be on a *T* node (this is a way of coupling *T*'s invariant to the invariants for other tags). The default is that no other tags are required.

reducesTo: either a quoted expression of the form '{...}' that can be used to reduce a tag to an equivalent (but more basic) tag by treating the appearance of *T*$ as *T.reducesTo*%|, or NIL if *T* does not reduce to any other tag. The default is NIL.

tagOnly: a Boolean indicating that a node with this tag need only be viewed by its parent as a node consisting of just its tags for the purpose of checking the parent's invariant. The default is *True*.

3.1.2 Defining types

In order to define the *attributes* and *contentType* attributes of a TAG definition, we must be able to define types. This is done by having a standard tag, TYPE, which can be used to construct TYPE nodes to define types. The definition of the tag TYPE can be "described" in Interscript (even though such a definition would actually be circular since TAG uses it):

```
TYPE%←{TAG$
attributes
    code %←node
    tags
    union %←NIL  -- or a node with a list of TYPE$ nodes as contents
    predicate %←{Predicate↑ | default%←'1'}
    default %←{Any↑ | default←NIL --Minimal automatic default--} }
contentType ←None↑ }
```

That is, a TYPE node has the relevant attributes *code*, *tags*, *union*, *predicate*, and *default*, with the following interpretations:

code must be an atom and one of *atom*, *evalStruc*, *node*, *num*, *quotedTerm*, *string*, corresponding to a subset of the marks on *VRecs* in section 2.3.1. Or, *code* can have the value *Any*, meaning that the type being declared may correspond to any of the marks above.

tags is a list defining what tags a node of this type *must* possess.

union, if not NIL, is a node value containing as contents a list of TYPE nodes; in this case, the type being defined is viewed as having to satisfy one or more of the type definitions given in the union.

predicate gives an Interscript quoted expression which a value must satisfy in order to be a valid instance of the type. The default value ('1') always yields True.

default provides a convenient way of specifying a legal default value for a node of the type being defined.

A TYPE node has no contents (since it is basically a record with its relevant attributes as components), so its *contentType* has the type *None* (see below).

Note that the above definition of the tag TYPE is circular in that it "uses itself" to define the types of the relevant attributes *code*, *tags*, *union*, *predicate*, and *default*.

Here are a set of standard TYPE definitions for the basic Interscript data types and for types needed for defining tags and types:

```
Bool% ← {TYPE$ code←num predicate%←'(A ↑ EQ 0) + (A ↑ EQ 1)'}
Number% ← {TYPE$ code←num}
String% ← {TYPE$ code←string}
Atom% ← {TYPE$ code←atom default←0 -- no default --}
AtomList% ← {TYPE$ tags←{ATOMLIST} default←{}}  -- no default--
ATOMLIST% ← {TAG$ contentType←Atom ↑ }
Any% ← {TYPE$ code←Any}
None% ← {TYPE$ predicate%←'0'}  -- ever False, forbids any content--
Predicate% ← {TYPE$ code←quotedTerm}
Type% ← {TYPE$ tags←{TYPE} default←Any ↑ }
```

To test if a given value v can be considered to satisfy a type with definition T, one can simply invoke the following *HasType* procedure with the V representations of v and T. HasType requires that

(1) either $T.code$ is *Any* or $T.code$ matches the mark of the value v; and

(2) if $T.code$ is *node*, then v's tags must be a subset of the tags given in $T.tags$; and

(3) if T is a union type ($T.union \# NIL$), then *HasType*[v, subtype] must be true for at least one of the subtypes given in the list $T.union$; and

(4) v must satisfy $T.predicate$.

To help in understanding how *HasType* works, try testing the value of **Mk_V[m: num, cNum: 2]** against the type definition *Bool* above. *HasType* should return FALSE.

```
HasType: PROC[v: V, T: --node--V] RETURNS[r: BOOL ← FALSE] = {
    bt: Env = Bindings[T];
    IF NOT (EqName[Lookup[bt, MkName[$code]].xName, MkName[$Any]] OR
            EqName[Lookup[bt, MkName[$code]].xName,
            MkName[MarkAsAtom[v.m]]])
        THEN RETURN[FALSE];
    IF EqName[Lookup[bt, MkName[$code]].xName, MkName[$node]] THEN
```

```
        IF NOT Subset[Tags[v.cVs], Lookup[bt, MkName[$tags]].cVs] THEN
        RETURN[FALSE];
    FOR tt: Vs ← Lookup[bt, MkName[$union]].cVs, tt.rest UNTIL tt = NIL DO
        IF HasType[v, tt.first] THEN EXIT; -- can stop checking unions as soon as one
            works--
        REPEAT
            FINISHED => RETURN[FALSE]; -- v didn't satisfy any of the types in the union--
        ENDLOOP;
    RETURN[ ApplyPred[arg: v, pred: Lookup[bt, MkName[$predicate]]] ]};
ApplyPred: PROC[arg: V, pred: V] RETURNS[t: BOOL] = {
    predEnv: Env = CONS[MkBinding[X, arg, MkName[$A], bind], X]; -- just the
        external environment plus A←arg
    r: V = Eval[ε: predEnv, v: pred];
    RETURN[IF r.m#num THEN FALSE ELSE (r.cNum#0) ]};
Subset: PROC[sub, set: Vs--all atoms--] RETURNS[BOOL] = {
    One: PROC[a: V, set: Vs] RETURNS [BOOL] = { RETURN[
        (IF set = NIL THEN FALSE
        ELSE IF EqName[a.xName, set.first.xName] THEN TRUE ELSE One[a, set.rest])]};
    RETURN[IF sub = NIL THEN TRUE ELSE (One[sub.first, set] AND Subset[sub.rest,
        set])]};
```

3.2 The invariant associated with a node

A node in an internalized script is *valid* if it satisfies the complete *invariant* for the node. A node's complete invariant depends on what tags are on the node, the TAG definitions for those tags, and the actual contents of the node and bindings of the relevant attributes of the node's tags. The following function *NodeInvariant* evaluates as much of a node's complete invariant as is possible without recourse to any external invariants associated with the node's tags. *NodeInvariant* returns the value *no* if a node is invalid, *yes* if it is valid, and *checkExternalInvariant* if it is valid modulo the external invariants of any of its tags that have such.

```
NodeValidity: TYPE = {yes, no, checkExternalInvariant};

NodeInvariant: PROC[n: --node--V] RETURNS[r: NodeValidity←yes] = {
    IF n.m#node THEN Error[WrongType];
    FOR t: Vs ← Tags[n.cVs], t.rest UNTIL t = NIL DO
        SELECT TagCorrect[n, t.first.cV] FROM
            no => RETURN[no];
            checkExternalInvariant => r ← checkExternalInvariant;
            ENDCASE;
        ENDLOOP;
    RETURN[r]};

TagCorrect: PROC[n, tagType: --node--V] RETURNS[NodeValidity] = {
    bt: Env = Bindings[tagType];
    RETURN[ IF
        CheckAttributes[bs: Bindings[n],
                types: Bindings[Lookup[bt, MkName[$attributes]]]] AND
        CheckContents[Contents[n.cVs], Lookup[bt, MkName[$contentType]].cV] AND
```

```
        Subset[Lookup[bt, MkName[$requiredTags]].cVs, Tags[n.cVs]] AND
        ApplyPred[arg: Stripped[n], pred: Lookup[bt, MkName[$nodeInvariant]]]
            THEN (IF Lookup[bt, MkName[$hasMoreInv]].cNum = 1 THEN yes
            ELSE checkExternalInvariant)
        ELSE no]};

    CheckAttributes: PROC[bs, types: Env] RETURNS[BOOL] = {
        One: PROC[b: --bind/bindStruc-- V] RETURNS[BOOL] = {
            t: --bind/bindStruc-- V = Lookup[types, MkName[b.xId]];
            RETURN[IF t = NIL THEN TRUE ELSE HasType[b.cV, t.cV]]};
        RETURN[IF bs = NIL THEN TRUE
            ELSE (One[bs.first] AND CheckAttributes[bs.rest, types])]};

    CheckContents: PROC[vs: Vs, type: V] RETURNS[BOOL] = {
        One: PROC[c: V] RETURNS[BOOL] = {
            RETURN[IF c.m = bindStruc THEN TRUE ELSE HasType[c, type]]};
        RETURN[IF vs = NIL THEN TRUE ELSE (One[vs.first] AND CheckContents[vs.rest, type])]};

    Stripped: PROC[n: --node--V] RETURNS[--node--V] = {
        RETURN[Mk_V[m: node, cVs: Strip[Items[n]]]]};
        Strip: PROC[vs: Vs] RETURNS[svs: Vs] = {
            One: PROC[v: V] RETURNS[Vs] = {
                SELECT v.m FROM
                    bind = > RETURN[Mk_Vs[Mk_V[m: bind, cV: One[v.cV].first, xId: v.xId]]];
                    node = > RETURN[IF ShouldStrip[v] THEN Mk_Vs[Mk_V[m: node, cVs:
                            Tags[v.cVs]]] ELSE Mk_Vs[Stripped[v]]];
                    ENDCASE = > RETURN[Mk_Vs[v]]};
            RETURN[IF vs = NIL THEN NIL ELSE ConcatVs[One[vs.first], Strip[vs.rest]]]};
        ShouldStrip: PROC[v: --node--V] RETURNS[BOOL] = {
        all of v's tags must agree on tagOnly for it to be true
            tags: Vs = Tags[v.cVs];
            FOR tags: Vs ← Tags[v.cVs], tags.rest UNTIL tags = NIL DO
                tDef: V = tags.first.cV;
                btDef: Env = Bindings[tDef];
                IF Lookup[btDef, MkName[$tagOnly]].cNum#1 THEN RETURN[FALSE];
                ENDLOOP;
            RETURN[TRUE]};
```

3.3 Safety rules for editors

Using TAG definitions and the concept of *NodeInvariant*, we can give some conservative rules for editors in treating parts of documents that correspond to nodes in a script. These are simply observations about how a node's complete invariant can be affected by various editing operations.

Rendering a node

An editor may display any node at all since this is not an editing operation. If it does not implement any of the tags on a given node, the editor could still display the external form of the node in the Interscript publication format (since it has to be able to externalize the document as a script anyway). Of course, if an editor implements one or more of the tags

on a node, it can use those mechanisms to render the node and use the more basic, generic display mechanism of script externalization for the attributes or content that it does not directly implement.

Editing a node

All editing operations within a node can be composed from the following *atomic* operations:

(1) replacing a value (whether of an attribute, an item of content, or the value of a structured binding) by another value of the correct type;

(2) removing an item of content (includes structured bindings and subnodes)

(3) adding an item of content (includes structured bindings and subnodes)

(4) adding a tag to a node

(5) removing a tag from a node

In general, an atomic operation affects a subset of the tags on a node in the sense that it could potentially affect the invariants of those tags (e.g., if it changed the value of an attribute relevant to them). A *single-node* sequence of operations is one that affects exactly one node.

A sequence of atomic operations that maintains the invariants of all nodes affected by the edit is called *valid.*

What valid single-node operations can an editor perform on nodes that it does not directly implement? Here is a general, safe mechanism:

save a copy of the node somewhere;

perform the operation sequence;

if the node's invariant and its parent's invariant are true allow the edit, otherwise restore the original state.

4

Standard document constructs

The preceeding sections of this standard have defined the Interscript language and the form of a script. Those sections define Interscript in an abstract sense, just as a Pascal syntax defines the Pascal language.

Yet to be described are the *meanings* that are attached to particular language constructs. In the same sense that Pascal is not useful for summing until the syntactic operator "+" is associated with concrete operations in real arithmetic, Interscript is not useful for interchange until particular syntactic and semantic constructs are associated with concrete document characteristics such as text, figures, paragraphs, and pages.

4.1 Introduction

In addition to defining the formal Interscript language, the Interscript 83 standard defines a number of basic script constructions.

The goal of the standard document constructs is to maximize effective interchange amongst normal editors. The Interscript language provides total interchange at a basic machine–to–machine level, but to provide interchange at a user–to–user level, we need more.

What we need are constructs, defined in the standard, that encourage editors to formulate document scripts in their terms. Thus we can define the "paragraph" and encourage editors to use that construct in scripts. An editor could use its own "lines" and "breaks," say, and the script would still be completely transportable; but it would be less editable, since most other editors would tell their users that they didn't know how to edit "breaks."

These basic constructs neither widen nor narrow the syntax and semantics defined earlier in this standard, rather they identify several areas where document interchange is enabled or enhanced by identifying and agreeing upon a particular representation for a popular document construct, such as *text* or *paragraph*. This agreement aids the construction of editing programs. It also aids the capturing of *user intent* – e.g. distinguishing between wide paragraph margins and a wide page margin, which are different *intents* even though, in a particular instance, they may result in the same layout.

4.2 Overview

A document has two basic aspects: its *content*, typically text, and its *layout*. The basic constructs defined in this section address these aspects, and address them as separately as possible. Maintaining this separation is vital when interchanging editable documents, since a characteristic of many documents is that the text is not attached to particular pages, but rather "flows" from one to another. The pervasive effects of the dichotomy between content and layout are visible throughout this standard.

4.2.1 Basic text

Text, especially, is kept simple. We wish to allow a very basic editor to be able to interpret and alter the textual portions of a script, since we believe that it will be common for a low–cost workstation to be asked to display document text (e.g. reading mail) and to perform minor edits on text (e.g. correcting typos in a fancy document produced on an expensive workstation).

4.2.2 Layout

We have created one construct – the box node – to formulate all geometric layout. Every box node participates in layout according to the same rules; the common layout arrangements, e.g. the indented paragraph, are expressed in terms of boxes.

The details of how the various document nodes interact when a layout is actually made are deferred to section five.

4.2.3 Tag definition

Many of the subsections, below, define particular tags. Tags mark Interscript nodes, and nodes are used for two distinct purposes (although the syntax is the same):

> A node may exist to collect and associate a set of named values. This usage is similar to a Mesa RECORD: a node–instance is typically assigned to a variable.
>
> A node may be document content. As content, it will be free–standing and a member of the logical–structure hierarchy.

Each tag introduced is presented by a *formal definition*, which is set off with heavy rules. The tag is defined by giving its formal definition in terms of the TAG$ construct. Following the formal definition, the meanings of the relevant bindings are discussed. The phrase "*something* node" is equivalent to, and a convenient substitute for, the phrase "node containing a *something* tag"

4.2.4 Publication encoding

Within ths standard, we often exhibit Interscript fragments. We do so by using the Interscript *publication encoding*. This encoding of the basic Interscript syntactic tokens is defined to facilitate human communication and comprehension, not to serve for editor–to–editor communication; to facilitate the latter, a *machine encoding* is used.

The publication encoding is basically the characters of the BNF exhibited in section two. This basic is augmented by several *encoding-notations* added for ease of use. These encoding-notations *do not extend the syntax in any way;* they are only part of the publication encoding; their sole purpose is to make it simpler to write and communicate a script. They are described, within the section where they become useful, by stating the encoding byte sequence and the actual BNF that it represents. Currently, the only example of a publication encoding-notation is the "< >" encoding for characters, introduced in section 4.6.3.

4.3 The document as an entity

A document, considered as a unit of transmission and storage, has certain "properties" that in many software systems are considered as properties of the document-as-filed-entity. These "properties" include date-last-modified, author, etc. Since we wish to "carry" these properties along with the document through editing operations, it is appropriate that they be given within the script.

Content Tag: **DOCUMENT$**

```
DOCUMENT % ← { TAG$
    attributes ← {
        lastAlterationEditor ← Atom
        lastAlterationTime ← String — ISO or other format
        author, owner, editReason ← String }
    }
— the attributes will be augmented during standardization
```

A document node defines, via its bindings, various attributes of the document; attributes that are not the document's content, but rather identify "properties" of the document as a whole.

The scope of the document node's bindings define the portion of the script that they describe. Thus typically there will be one document node at the outermost level which contains all of the other nodes that comprise the document.

4.4 Boxes and layout

Document layout is described as an arrangment of *boxes* on document *pages*. Some layouts are straightforward, while others are complicated and involve subtle positioning requirements: keep the footnote on the page with the reference, the illustration must be on a right-hand page, etc.

This section describes box metrics; layout algorithms are described in section five.

4.4.1 Box fundamentals

The function of box nodes is to describe document layout by (conceptually) partitioning the (rectangular) document presentation medium into rectangular regions with edges that are parallel to the edges of the medium.

Box nodes are concerned with the *layout geometry* of the document. They are the *only* standard nodes that are. All geometry is expressed in units of *micas* where a mica is 0.01 millimeter. [A script can use multiplication to effect other units: e.g. express unit inches as 1*2540.]

Surprisingly, ISO standardization is not using the metric system, but defining geometry in terms of a "Basic Measurement Unit" which is variously proposed to be 1/240 inch, 1/600 inch, or 1/1200 inch. Perhaps Interscript should change a sentence, above, to read *"All geometry is expressed in units of 'Basic measurement Units' or* BMUs *where a BMU is <whatever ISO decides>"* **And perhaps Interscript should allow the size of the BMU to be defined in the script.**

We define a box's layout by defining its size and positioning. Size and position are defined separately for each coordinate. Sizes and positions are defined by a *number* when a document is actually rendered (laid out on a medium). When they are stated by a script, however, they are typically given with some leeway, and are then defined by a *measure*.

4.4.2 Measures

A measure is a triple of numbers [under, nominal, over] which suggest a distance but allow some leeway in the layout process, or the atom SYNTHESIZED which is a "stand-in" for such a triple.

A measure is a *numeric measure* if it is bound to a measure node (a triple of numbers) and not bound to the atom SYNTHESIZED.

A measure node is a record that supplies the nominal metric for a span extent, together with an "over" and "under" that indicate how much leeway the layout proces has to adjust the value. The exact semantics of "over" and "under" are defined in the layout discussions in section five.

Record Tag: **MEASURE$**

```
MEASURE %← { TAG$
    attributes ← {
        under ← Number ↑
        nominal ← Number ↑
        over ← Number ↑ }
    contentType ← { NIL }
    }
```

Sometimes a box's size or positioning is not specified in the script, but is computed during layout. In this case, the binding is made to a special atom instead of to a measure node:

> SYNTHESIZED: The measure does not have explicit metrics; rather the metric is computed during the layout. The typical usage is to let a box assume the size dictated by the layout requirements of its content.

4.4.3 Box nodes

A box node suggests an area and its positioning *relative to the enclosing box.* Recursively, a tree of box nodes suggests a complete document layout.

Content Tag: **BOX$**

```
BOX %← { TAG$
    attributes ← {
        xSpan ← SPAN
        ySpan ← SPAN
        coordinateRotation ← ONEOF {
            0ccw 90ccw 180ccw 270ccw }
        clips ← Boolean ↑ }
    }
```

The **xSpan** and **ySpan** bindings give the metrics of a box in the two (x, y) layout directions.

The **coordinateRotation** binding describes how the coordinates are rotated (in steps of counter–clockwise right angles), inside the box, relative to the coordinates that were used to lay out the box. Throughout this standard, in the interests of simplicity, *coordinate rotation is ignored in many discussions.* Algorithms are described as though the coordinate system does not change - e.g. the interior span that is parallel to an outside xSpan is assumed to be an inside xSpan, rather than a ySpan, or possibly the negative of an xSpan. The reader and implementor should provide logical coordinate rotation, as required, as the box–tree is traversed. A formal definition of page coordinates is given in section 5.4.2.

Boxes are always allowed to be placed so that they "stick out" beyond the area of their containing box. If the **clips** binding of a container is TRUE, then children's "stick out" portions *are not rendered* (imaged onto the display medium); if **clips** is FALSE, then "stick outs" are rendered.

Record Tag: **SPAN$**

```
SPAN %← { TAG$
    attributes ← {
        lowPartExtent ← MEASURE
        highPartExtent ← MEASURE
        fromLowContainer ← MEASURE
        fromHighContainer ← MEASURE
        fromLowSibling ← MEASURE
        fromHighSibling ← MEASURE }
    contentType ← { NIL }
    }
```

Figure 4.4–1 illustrates how the attributes of a span define the size and positioning of a box node.

In a tree of box nodes, all but the "leaf" boxes will have layout continuing within them. Such box nodes are given the **INSIDELAYOUT$** tag. Its attributes describe how layout is performed inside this box node.

Content Tag: **INSIDELAYOUT$**

```
INSIDELAYOUT %← { TAG$
    attributes ← {
        xLayout ← INSIDELAYOUTMETHOD
        yLayout ← INSIDELAYOUTMETHOD }
    requiredTags ← { BOX }
    contentType ← { NIL }
    }
```

Record Tag: **INSIDELAYOUTMETHOD$**

```
INSIDELAYOUTMETHOD %← { TAG$
    attributes ← {
        direction ← ONEOF { fixed up down onOrigins }
        siblingAdjacency ← ONEOF { parallel serial — TEX — } }
    contentType ← { NIL }
    }
```

Figure 4.4–2 illustrates some box layout features.

4.4.4 The page node

A page node collects together the information that make up a single document page.

Content Tag: **PAGE$**

```
PAGE% ← { TAG$
    attributes ← { — get these from Interpress — }
    requiredTags ← { BOX }
    contentType ← { NIL }
    hasMoreInvariant ← TRUE  — no ancestor can have PAGE tag
    }
— page attributes will be the appropriate set of Interpress
— attributes to define such things as desired paper color
```

The document contains some number of page nodes. Each page node is the root of a tree of box nodes. The page nodes, in script order, are the pages of the document. The tree of boxes rooted at a page node defines a page's layout and content, from the page box all the way down to character boxes. *The layout structure of the document is a sequence of box–trees, with the root of each tree being a page node.*

4.5 Naming and labelling nodes

It is convenient to be able to attach *naming information* to particular nodes within the document. This can serve two purposes:

to allow for intra–document references, which can augment the tree–like node hierarchy of the logical structure, and indicate non–hierarchical document connections

to augment nodes with user–sensible information so that the user can e presented with information bout ertain nodes.

This naming information is attached to a node by adding a tag to the node; the naming information is then an attribute of the additional tag. This is the standard way to augment a node: adding a tag extends the set of relevant bindings of he node, and thus adds capability.

4.5.1 Labelling

It is sometimes necessary to associate two nodes that are not connected via the document logical structure "tree" of nodes. Such a connection is created through *labels*.

The nodes to be connected (two or more) are given a *label* which is the same atom; an editor finds the correspondence by matching label–atoms. The label–atoms are not user sensible; unique labels can easily be created by an editor.

A label–atom is attached to a node by giving the node a LABEL tag:

Tag: **LABELS$**

```
LABELS%← { TAG$
    attributes ← {
        labels ← { Atom ↑  — can be more than one — }  }
    }
```

In this standard, the phrase "the node is labelled with the atom" is shorthand for "the node is given a LABEL$ tag and the script's bindings are arranged so the atom is among the elements of the current binding to the attribute *labels*"

4.5.2 Intra–document references via labels

There are several forms of intra–document references that are handled by the label mechanism. Listed below are the possible cases.

1. A set of (two or more) nodes with no distinguished memeber(s):

   A unique label–atom is chosen and each member of the set labelled with that atom.

2. A set of nodes where one node "points to" all of the other members:

   A unique label–atom is chosen; the atom is associated with the pointer–node via a special attribute of that node; the other members of the set are labelled with the atom.

3. A sequence of nodes, where the ordering matches script order:

   Handled just like case one, with the ordering recovered by script examination.

4. A sequence of nodes, where the ordering does not match script order:

   a) This may be handled like case one, with the ordering information supplied outside the labelling arrangements; for example, all of the nodes may have some other tag, and this tag may have a sequence–number attribute. Or

   b) Every member of the sequence except the initial member is given a unique label. The initial mamber of the set has an attribute (because it has some special tag) whose binding is a list of the label–atoms of the members of the sequence. Or

   c) If the members of the sequence all have some tag (this is typical) then one can generate atoms, but not use the LABEL tag, rather each member may have *from* and *to* attributes, and each (from to) pair shares one generated atom. This resembles a singley–threaded list, with matching atoms replacing pointers.

   One might consider an arrangement like c, above, which uses only LABEL tags and labels attributes: the initial node is given a label; each subsequent node (except the last) is given *two* label atoms, with the first matching a label of its predecessor and the second matching a label of its successor; the final node has

only a predecessor label. Such an arrangement, however, does not generalize well when nodes need to be members of more than one sequence or need to be labelled for some other purpose.

5. More complicated arrangement, such as collections of sub–sequences and directed graphs:

Are handled via extension of the methods suggested in #4, above.

In this standard, when we use the phrase "the node pointed to by ..." we mean an arrangement like #2, above, where the set has two members. Thus "the node pointed to by the *from* atom" means "the binding to *from* is an atom; there is exactly one node in the document which has a LABELS tag and the *from* atom among its bindings to *labels*; this node is meant; if there is not exactly one such node, it is a script error."

4.5.3 User–sensible naming

It is often convenient for an editor to attach a user–sensible "name" or "description" to a node. This may provide, e.g., an explanation of what the data–value should be, to aid editor prompting. We provide this by additional tags which can augment any node.

Tag: **USERNAME$**

```
USERNAME %← { TAG$
    attributes ← {
        name ← String ↑ — user–sensible name — }
    }
```

Tag: **USERDESCRIPTION$**

```
USERDESCRIPTION %← { TAG$
    attributes ← {
        description ← String ↑ — user–sensible description— }
    }
```

4.6 Basic text content

This section defines the nodes which comprise the basic content of a document.

A document script is often organized as a hierarchical arrangement of content: words within paragraphs within sections within chapters. The constructs in this section provide such an arrangement.

A script hierarchy which reflects the document's physical medium - lines within columns within pages - is possible with these constructs. It is more likely, however, that a script will be arranged as a content–tree and then rearranged into pages via an (editing) operation called *pouring*. Pouring is described in section five.

4.6.1 Characters

The basic unit of text is the character. Characters participate in layout, and, therefore, are (conceptually) boxes. *A character node can be viewed as a stand-in for a box node: the box node is the one obtained by looking up the "character" in a font.* This is similar to the view that the Interpress standard takes about characters.

Content Tag: **CHAR$**

```
CHAR %← { TAG$
    attributes ← {
        font ← FONT }
    contentType ← { Number ↑ }
    }
```

Each CHAR$ node is a single character. The content is a single number that identifies the character using an appropriate character numbering standard. The Interscript Standard is separate from any character set numbering standard. To facilitate interchange, editors clarly must agree on one character set. We suggest the Xerox Character Set Standard, which embraces and extends the current international standards.

The character node acts just like a box node. Conceptually, the font binding identifies an array of box node prototypes; the number in the character node is used to look up a box in the font array; the box node thus obtained is used in place of the character node.

It is obviously unreasonable to encode an entire document's text one letter at a time via character nodes. Later in this standard we discuss encoding notations, which allow common constructions, such as a sequence of character nodes, to be compactly expressed.

Record Tag: **FONT$**

```
FONT %← { TAG$
    attributes ← {
        name ← String ↑ — printers name of the font family
        points ← Number ↑ — body size of type in printer's points
        baselineOffset ← Number — baseline up-shift in points
        italic ← Boolean ↑ — whether italic form should be used
        boldness ← ONEOF { lighter regular darker }
        strikeout ← Boolean ↑ — whether 'struckout' version
        underlining ← ONEOF { none single double }
        — the metrics for strikeout and underline are
        — provided by the font design — }
    contentType ← { NIL }
    }
```

An Interscript font node identifies and describes the font that the characters come from. It contains the standard information that a printer would expect in a font description, plus several attributes that have been bundled into the logical "font" for convenience:

The *baselineOffset* supplies a vertical shift for subscripts etc. Conceptually, an ordinary font is augmented by a similar font where the baseline is one point higher, another where the baseline is two points higher, etc. Practically, this "font" will be synthesized as needed.

Real-world fonts have complicated arrangements of their "boldness." Rather than try to track this, we state that the boldness of the font is captured in the font name: the name is, e.g. "Futura Light." To aid editors in letting users "boldify" words, we provide the *boldness* enumerated. It lets the font node specify "one step lighter" or "one step darker."

Conceptually, an ordinary font is augmented by *strikeout* and *underline* versions of itself. For some fonts and some printing methodologies, this is likely to be true; otherwise the strikeout and underline will be artificially produced.

4.6.2 Special characters

One maywish to create a built-up structure and have it act like a character: create a logotype, for example, with a small bitmap frame, or build up a character like "$\frac{2}{3}$" with an equation.

Tag: **PSEUDOCHAR$**

```
PSEUDOCHAR %← { TAG$
    attributes ← { }
    requiredTags ← { BOX }
    }
```

The box defined by the pseudochar node will be treated just as if it was the conceptual box equivalent to a character node.

4.6.3 Text

Running text (as opposed to characters in isolation) has attributes that single characters lack. To provide for these attributes, we may imbed running text in a text node.

Content Tag: **TEXT$**

```
TEXT %← { TAG$
    attributes ← {
        dialect ← Number ↑
        — see appendix 00 for a numbering of languages — }
    contentType ← { textContent }
    }

textContent ← { union {
    {Type ↑ | Tags ← {CHAR}}
    {Type ↑ | Tags ← {PSEUDOCHAR}} }
— "textContent" abbreviation is used throughout this specification
```

The semantics of text nodes, and of character nodes, is deliberately kept simple. Thus simple editors can edit "text" without "understanding," e.g., anything relating to paragraph attributes.

4.6.4 A text encoding–notation

We define a compact way, in the publication encoding, to place character nodes into a text node. The publication encoding

```
{ TEXT$ <Cat> < and dog.> }
```

is a notation for

```
{ TEXT$ {CHAR$ LABELS$ 67} {CHAR$ LABELS$ 97}
{CHAR$ LABELS$ 116} {CHAR$ LABELS$ 40} {CHAR$ LABELS$ 97}
{CHAR$ LABELS$ 110} {CHAR$ LABELS$ 100} {CHAR$ LABELS$ 40}
{CHAR$ LABELS$ 100} {CHAR$ LABELS$ 111} {CHAR$ LABELS$ 103} }
```

where the character–set reresentation for "C" is 67, etc.

4.6.5 Text fields

A text field node exists within running text. It holds a value which is reduced to text for presentation.

Tag: **TEXTSHOW$**

```
TEXTSHOW% ← { TAG$
    attributes ← {
        type ← ONEOF { text amount date ... }
        value ← Any ↑
        picture ← String ↑ }  — a COBOL picture clause? —
    requiredTags ← { TEXT }
    contentType ← { textContent  — same as TEXT — }
    hasMoreInvariant ← TRUE  — value matches content —
    }
```

The TEXTSHOW tag acts as an augmentation of a text node. Thus an editor which understands text can display (but not edit) the content, which is the textual representation of the underlying value.

4.6.6 Text value

A text remote value node is an augmentation to a text field. It provides a mechanism for incorporating derived values ("see figure 2.3 on page 27") into text.

Tag: **TEXTREMOTEVALUE$**

```
TEXTREMOTEVALUE% ← { TAG$
    attributes ← {
        from ← Atom ↑
        value ← Any ↑ }
    hasMoreInvariant ← FALSE
    }
```

As an editing operation, typically during a pour, the **value** binding of the text field can be derived from the node pointed to by the from atom *by copying the binding of* **value**. The **value** can then, perhaps, be reduced to text via a TEXTSHOW tag.

Notice that a set of textremotevalue nodes can be placed within another node; each one capturing one remote value; the value binding of the container node can then be calculated via arithmetic on the imbedded values. This allows ordinary "fill in rules" to be expressed in Interscript.

When content, as opposed to a binding, is to be retrieved, we use a different tag:

Tag: **TEXTREMOTECONTENT$**

```
TEXTREMOTECONTENT% ← { TAG$
    attributes ← {
        from ← Atom ↑ }
    requiredTags ← { TEXT }
    contentType ← { textContent  — same as TEXT — }
    }
```

The text remote content tag is an augmentation of a text node. *As an editing operation,* typically during a pour, the content of the text field can be copied from the node pointed to by the *from* atom.

4.7 Pouring constructions

For some documents, it is appropriate to have a tree of box nodes defined for each page, and to place in a box node the exact document content (text, illustrations, ...) that belong there. This is a straightforward approach - it is called solid layout.

When a solid layout is inappropriate, we separate the description of the layout and the content. Instead of placing text (and other content) within the layout boxes that it belongs in, we split it out into a separate *logical structure.*

Then rendering must include some function which recombines the content and layout, so it can put the text in the right places on the page. We call non-solid layout *fluid,* and the act of combining the layout information and the logical structure elements *pouring.*

Layout is strongly affected by content, and we cannot, in general, produce a correct layout structure without knowing where the content falls. The font-size of the running text, for example, affects how many lines fit into a column. So the layout information cannot be precise, but rather must supply a template from which the correct layout of a particular content can be constructed.

"Pouring" itself is detailed in section five.

In this section we define, without a detailed explanation of their semantics, the nodes that control pouring.

4.7.1 Pouring, templates, and molds

A pour operation is controlled via a pour node. The content of the pour node is the logical structure elements to be poured. The bindings of the pour node control the pour operation.

Tag: **POUR$**

```
POUR %← { TAG$
    attributes ← {
        labelset ← { Atom ↑  — can be more than one —
            — supplies label-set for matching — }
        template ← Node  — a template or contains template(s)
        satisfactionThreshold ← Number ↑
        satisfactionForwardSearch ← Number ↑
        satisfactionUpwardSearch ← Number ↑ }
    }
```

The binding to template supplies the layout template that the content is "poured into."

Tag: **TEMPLATE$**

```
TEMPLATE %← { TAG$
    attributes ← {
        expresses ← ONEOF { sequence alternation repetition } }
    contentType ← Node ↑
    }
```

A template node supplies a recipe for the construction of various layout possibilities that are valid document layouts. A pour operation will examine this space of layout possibilities and choose one actual layout that is appropriate for the actual content.

Tag: **MOLD$**

```
MOLD %← { TAG$
    attributes ← { coercion ← QuotedExpression }
    }
```

The mold tag defines the nodes, within a template, that are "targets" for content to be poured into. Some nodes in the template will not be molds becase they are supplying layout geometry, e.g. non-leaf boxes. Other nodes in the template will not be molds becase they are supplying fixed page-content, e.g. fixed marginal rules.

Tag: **FENCE$**

```
FENCE %← { TAG$
    attributes ← { }
    contentType ← NIL
    }
```

A fence node defines a logical "break" for layout algorithms.

4.7.2 Paragraph

A paragraph node collects together the text and formatting information that make up a single document paragraph.

It is important to understand that a paragraph node is (just) a shorthand for a description of a particular layout arrangement, together with the paragraph's (text) content. *A paragraph node is a stand-in for a pour node.*

We standardize the paragraph node, instead of having all paragraph-like layout performed with pour nodes, so that:

1. The user-intent surrounding the "paragraph" is captured: the editor can recognize the user's paragraphs, e.g. to support a "select next paragpraph" key; the user's semantics of paragraph-margin, widow control, etc., are captured in the script and easily recognized by editors.

2. Paragraph-layout can be "hard-coded" into an editor. This may greatly speed up pagination.

Tag: **PARAGRAPH$**

```
PARAGRAPH %← { TAG$
    attributes ← {
        lineJustification ← Boolean ↑
        lineRaggedness ← ONEOF { atBeginning centered atEnd }
        avoidWidow ← Boolean ↑
        avoidOrphan ← Boolean ↑
        firstLineLeftMargin ← Number ↑
        leftMargin ← Number ↑
        rightMargin ← Number ↑
        preleading ← MEASURE — of the paragraph —
        postleading ← MEASURE — of the paragraph —
        interlineLeading: MEASURE
        aboveBaseline ← MEASURE — for TEX line layout —
        belowBaseline ← MEASURE — for TEX line layout —
        specialLineFormatting ← QuotedExpression
        specialLineFormattingThroughLineN ← Number ↑
        tabstops: { TABSTOP Nodes — how do you say this ?? — }
    }
    reducesTo ← — see appendix P — }
```

We expect that the paragraph attributes, above, will be extended and perhaps modified as this draft is finalized. There is a trade-off between making the paragraph simple (and forcing odd-ball cases to use a pour) and making the paragraph fancy enough to include all the "reasonable" cases.

4.7.2.1 Layout within the paragraph

Within the paragraph, layout remains boxlike. A paragraph is composed of *lines of text.* These lines are defined by *line node* boxes.

The line node boxes are layed out consecutively according to the layout direction inside their container. (Since the paragraph node is a pour node, the ultimate container of the line nodes will be a column or page; see section five.) Within a line, characters are laid out according to the layout direction inside the line node box. Thus the two layout directions define the writing–order of the paragraph text. For normal European text, the two layout directions will be top–to–bottom and left–to–right.

4.7.2.2 The line node

The line nodes define the text structure of the paragraph; one of the line layout boxes is probably defined with a repetition. They specify indenting, inter–line leading, special fonts for the initial line, etc.

In typical usage, line nodes do not appear in the script; only paragraph nodes appear, and the lines are derived from the paragraph information. For situations where the standard paragraph does not suffice to describe a desired paragraph layout, explicit line nodes can be used.

Tag: **LINE$**

```
LINE %← { TAG$
    attributes ← {
        lineJustification ← Boolean ↑
        lineRaggedness ← ONEOF { atBeginning centered atEnd }
        leftMargin ← Number ↑
        rightMargin ← Number ↑
        leading ← MEASURE — interline —
        aboveBaseline ← MEASURE
        belowBaseline ← MEASURE
        tabstops: { TABSTOP Nodes — how do you say this ?? — }
    reducesTo ← — see appendix Q — }
```

The line node has several attributes that are copied through from the paragraph node.

A line node is typically a *mold* for a pour operation, and therefore also has a **coercion** attribute. This attribute can supply a "late" modification of the font information. The actual font used to display the characters of running text is not defined by the **font** attribute of text directly, but rather by the binding to **font** possibly modified by the **coercion** quoted expression. A font coercion is useful when one wishes to alter the font of some text after the pour - say because one wishes the first two lines of a paragraph to be larger than the rest. This *coercion* of values during a *pour* is discussed in section 5.5.2.

4.7.2.3 Tabs

We provide a facility which supports typewriter-like "tabbing." Tabular material will normally use the table facility and not tabbing.

Record tag: **TABSTOP$**

```
TABSTOP %← { TAG$
    attributes ← {
        type ← ONEOF { left centered right aligned }
        alignedOn ← Number ↑  — a character —
        position ← Number ↑ }
```

An array of tabstops may be supplied for a paragraph. Tab nodes in the text then can move text-positioning to a tabstop.

Tag: **TAB$**

```
TAB %← { TAG$
    attributes ← { } }
```

A tab node will position text to the next tabstop, where "next" means next-by-count, not next-by-position. [The "next" tab stop may be on the next line. Editors that wish to disallow this should ensure that there are enough tabstops set.]

4.7.3 Fill

Often one wishes to terminate the "filling" of the currently-being-filled layout entity and continue with the "next" one. E.g. force a new line or a new page. The fill node aids in implementing this function.

Tag: **FILL$**

```
FILL% ← { TAG$
    attributes ← {
        container ← Atom ↑ }
    }
```

The fill tag simply augments the node it is placed on, and provides the relevant binding **container**. It is up to the molds of a pour operation to be arranged to provide the functionality of terminating the "filling" of the layout entity; this is further discussed in section five.

4.7.4 Value replication

Sometimes the desired effect of a "pour" is not simply to put each item of content into a place in a layout. Often one wishes to replicate certain values. A page heading, for example, will appear in each page node as the result of a pour, yet there is only one instance of the page heading in the script.

We address this need via the vacuum node.

Tag: **VACUUM$**

```
VACUUM %← { TAG$
    attributes ← { sources ← { Atom ↑ } }
    requiredTags ← { MOLD }
    }
```

When a vacuum node is encountered in a template, the pour operation is at some point, matching the pour label set, in the content. The content tree is scanned upward, examining the direct ancestors of the vacuum node and their immediate descendants, for the first instance of a vacuum source node whose binding to **sources** is the same as in the vacuum node. That is, a match with $(parent^i)_j$ for the smallest i, then the smallest j. The content of this node is used as the source of the pour into the vacuum node in the template. The pour may alter bindings and content in the target vacuum node, as normal. If no matching vacuum source node is found, the vacuum node is treated just like an ordinary mold that lacks matching liquid.

Tag: **VACUUMSOURCE$**

```
VACUUMSOURCE %← { TAG$
    attributes ← { sources ← { Atom ↑ } }
    }
```

4.7.5 Anchors

It is often convenient to synchronize streams of content. Footnotes and illustrations should be near their references. We achieve this synchrony in layout via "together" nodes.

Content Tag: **TOGETHER$**

```
TOGETHER %← { TAG$
    attributes ← {
        groupings ← { GROUPING%* } }
    contentType ← Node ↑
    }
```

Record Tag: **GROUPING$**

```
GROUPING %← { TAG$
    attributes ← {
        penalty ← Number ↑
        levelLabel ← Atom | NIL — nil = 'immediate ancestor' —
}
    contentType ← Node ↑
    }
```

The content nodes of the together node should be layed out "close" to one another. For each possible pair of nodes (if the together node contains m nodes, there are m*(m-1)/2 pairs), a layout incurs a penalty which is the sum of the **penalty** in each grouping times the logical "distance" between the layout placements. The "distance" is computed as follows:

If levelLabel ≠ NIL: In the solid layout, traverse the tree upward from one node until encountering a node with the **levelLabel**, and mark that node. Do the same for the other node in a pair. If either node of a pair does not have an ancestor with the levelLabel, then the "distance" is 0. Else if the two ancestor nodes are the same, then the "distance" is 0. Otherwise, traverse the solid layout tree in depth-first order, and count the noumber of **levelLabel**led nodes you encounter between the two marked nodes; the "distance" is encounters - 1.

If levelLabel = NIL: In the solid layout, find the immediate parent nodes of the pair of nodes. If the two parents are the same node, then the "distance" is 0. Otherwise, the "distance" is 1.

4.7.6 Penalty node

It is convenient to be able to arbitrarily "penalize" a particular layout. This allows a bias to be given to the choices within a regular expresion node.

Content Tag: **PENALTY$**

```
PENALTY %← { TAG$
    attributes ← { penalty ← Number ↑ }
    contentType ← { NIL }
    }
```

4.8 Tables

The function of a table node is to collect together the text, numbers, and formatting information that make up a simple document table.

We take the view that the table itself is a layout problem, with content-information to be poured – the table elements – and a "mold" – regular expressions that define the format of

the table. To ensure that the table node is understandable by the simple editor – to avoid the table being an "open ended" construct – we define it with a *reducesTo*.

This definition of the Interscript table is the one originally developed for the draft standard. It is ambitious and standardizes multi-page, split-column, feature-rich tables. It may be appropriate to, instead, initially standardize a simple, indivisible, rectangular-lattice table. The latter would be easier for editors to "understand" and, lacking the interaction with layout that multi-page tables posess, would simplify pouring.

Tag: **TABLE$**

```
TABLE %← { TAG$
    attributes ← {
        splittable ← Boolean ↑ — can it be split across pages
        heading ← — tableheading — Node ↑
        headingSubsequent ← — if split to 2nd page — Node ↑
        layoutOfRow ← Node ↑ }
    contentType ← { TABLEROW }
    reducesTo ← — see appendix T — }
```

A table node defines a table by defining "rows." The major repeating dimension of a table may repeat vertically or horizontally; we call the repeating major structure the "row" in either case. The table's container's layout coordinate system and **layoutProceeds** determines whether the "rows" repeat horizontally or vertically.

A table may have *column headings*; these are supplied separately from the rows, via a binding.

A table may be considered to have *row headings;* these are the initial column entries in each row. [The standard considers column headings to be "special" but does not consider row headings to be special.]

Since the table node is a pour node, it has a binding to **re** which gives the template which controls the layout of the table. The table carries its layout information and its table entries separately, and a *pour* is necessary to combine them.

Various information may be maintained, within a table, about the columns. This may include restrictions on table elements in the column, formatting information, and possibly information about *split columns*. All of this information is supplied by the binding to **layoutOfRow**.

The template in the **layoutOfRow** binding describes the layout of each row.

Row layout is made complicated by the presence of "split columns." Detail of "split columns" could perhaps be derived from the templates; this would be difficult in practice. To simplify "split column" desription, we standardize define that all "splitting" of logical columns is derived from the nesting of the COLUMNSPLIT nodes within the **layoutOfRow** node; this is the sole function of COLUMNSPLIT nodes.

Tag: **COLUMNSPLIT$**

```
COLUMNSPLIT %← { TAG$
    attributes ← { }
    }
```

It is important to note that the column-widths are fixed by the layout information contained within the node bound to **layoutOfRow** (and probably reflected in the tableheadings). *Altering the column widths is an editing operation.* A table node within a script has specified column widths; there is no "automatic" width adjustment provided by the script; an editor, of course, is free to edit the widths as the user edits the table elements.

An ordinary matrix-style (m by n) table has no split columns. Its **layoutOfRow** template will contain no COLUMNSPLIT nodes.

An ordinary matrix-style (m by n) table will contain *m* TABLEROW nodes. Each TABLEROW node will contain *n* TABLEELEMENT nodes.

Tag: **TABLEROW$**

```
TABLEROW% ← { TAG$
    attributes ← { }
    contentType ← { Node }
    reducesTo ←  — see Appendix T — }
```

A tablerow node defines a (logical) row of the table. Tablerow nodes exist (only) as content within a table node. There is one tablerow node of content for each logical row of the table instance.

Tag: **TABLEENTRY$**

```
TABLEENTRY% ← { TAG$
    attributes ← { matchNumber ← Integer ↑ }
    }
```

Tableentry nodes are one-to-one with the "leaf node" entries in the table. In the case of a simple table, there will be (number of columns) tableentry nodes within each tablerow node and thus (number of rows)*(number of columns) tableentry nodes in the entire table node. The actual *content* of the tableentry is the *content* of the table entry, e.g. the tableentry node may be a text node.

If the table is not an ordinary matrix-style one, e.g. it contains "split columns" with repeating groups, then the number of tableentry nodes per row can vary, and the actual layout of the rows can vary. There is still only one binding to **layoutOfRow**; the variability is is accomodated by having the row template describe a pour operation, with

regular–expression choices and alternations allowing for splits and repeats. It then becomes necessary to arrange for an appropriate match–up between tableentry nodes in the content and in the template. We control this via the **matchNumber** bindings of the tableentry nodes by stating: when choices and repetitions in the template allow for more than one logical row structure, the bindings to matchNumber in the template must be sufficient to resolve all ambiguity in the regular–expression. In particular:

> Within an alternation, all bindings to **matchNumber** must be different and none may be the atom MATCH.
>
> In the cae of a repetition, the bindings to **matchNumber** in the tableentry before the repetition (if any), the repeated tableentry, and the tableentry after the repetition (if any) must be different and none may be the atom MATCH.
>
> Also, none of the bindings to **matchNumber** *in the content* may be to MATCH.

4.9 Inked boxes

We allow a box node to be "filled" with a straightforward "ink." This allows for the simple definition of black lines within a layout.

Tag: **INKED$**

```
INKED %← { TAG$
    attributes ← { color ← ONEOF { Black ...} }
    requiredTags ← { BOX }
    }
```

4.10 Interpress graphics

We allow arbitrary images to be placed inside box nodes via an interpress node. This node contains a fragment of an Interpress master.

The coordinate system is that of the interior of the containing box: the box's origin and coordinate rotation are used, and the length transformation is the unit (micas to micas) one.

The Interpress content of the node is "executed" in the Interpress sense to obtain an image. This "execution" is done exactly as an Interpress *composed operator* is executed.

Tag: **INTERPRESS$**

```
INTERPRESS %← { TAG$
    attributes ← {
        version ← String ↑
        — exactly the Interpress heading character sequence — }
    requiredTags ← { }
    parentTags ← { BOX }
    contentType ← Integer ↑ — probably an encoded sequence —
    }
```

4.11 Non–Interscript editing

We enable a non–Interscript conforming editor to "edit" a script by allowing it to leave "warnings" in nodes whose invariants may not hold. This is particularly useful when a *user* wishes to edit a node that he understands but the workstation software does not; the workstation could allow him to directly edit Interscript constructs, much as a GML or Scribe user directly edits the underlying data–representation.

Tag: **CAUTION$**

```
CAUTION %← { TAG$
    }
```

The presence of this tag indicates that possibly non–Interscript conforming edits have been made to this node. Editors should exercise care. If an editor understands a node, it can, if it desires, acertain that its invariant is satisified and then remove the warning.

4.12 Styles

Many editors use "styles" in their user interface. Styles enable the user to associate a named property, e.g. *foreign word*, with text. The style then can cause the text to be presented in italic. This indirection has two advantages: it lets one have different properties (e.g. *foreign word* and *emphasis*) that render the same, yet can be searched for distinctly, and it allows the presentation of a document to be altered by substituting a new set of styles.

The concept of styles exists within the Interscript language via the "%" indirection operator. What the stylesheet standard document construct defines is the *user–sensible* styles of a document.

The "stylesheet" consists of the bindings to the environment **stylesheet**. Each element of that environment is itself an environment which defines a single style.

Each style, therefore, has an element name of the form **stylesheet.*name***, and that is how the style is referenced within the script.

A style–environment has three elements:

The **userIdentification** of the style: a text string which is the user's name for the style. The element name in the stylesheet cannot act as the user–sensible name, since it is restricted to the ISO 646 character set.

The **description** of the style: a text string can explain the style to the user. This has no effect on the script, but is useful for editors.

The **expansion** of the style: a node–value; its content is the expansion of the style. Thus the invokation of a style–sheet style within the script will be literally of the form **stylesheet.*name*.expansion%|**.

5

Layout

5.1 Introduction

The preceeding section defined several standard node types that encourage editors to formulate scripts in their terms, in order to maximize interchange.

A main thread within those nodes types was the concept of layout – the arranging of document content on pages. This section discusses layout in detail.

5.2 Overview

Layout uses the box node, defined in the previous section, as its "workhorse." All geometric layout is defined in terms of box nodes and their placement.

5.3 Boxes and measure arithmetic

The function of box nodes is to describe document layout by (conceptually) partitioning the document presentation medium into rectangular regions.

Recall from section four that box nodes suggest a desired layout via various *measures* which suggest box dimensions and box placement-within-parent via a triple [undersize, nominal, oversize] of numbers. [A measure may also be the atom SYNTHESIZED, which is a stand-in for a measure.]

In a final layout, a box has a position and dimensions on the rendering medium. The final position and size of the box may be a function, not only of its measures, but of the measures of other boxes that must be laid out with it.

5.4 Layout with boxes: "Solid" layout

A script may arrange boxes in pages to achieve layout, and place each piece of content (text, graphics, image ...) into the appropriate box. This strictly-nested layout method is termed "solid" layout. Solid layout is straightforward to describe and is suitable for rigid document description: it might be used for single-page documents such as posters, or for tightly-controlled layouts, such as a display of illustrations.

For documents where the text "flows" from page to page, solid layout is inappropriate, and a layout model where continuous text is "poured" into successive pages is required. Subsection 5.5 describes the additional features of "fluid" layout.

Note that an initial step in rendering a "flowing" document is to produce, from the "fluid" document, a "solid" document which contains the same content; this "solid" document can then be rendered.

5.4.1 Solid page layout

Given a page-tree of boxes to be laid out, we define a *page layout* as a specific arrangement of boxes on the rendering medium.

While a box in the script has various *measures* that describe its layout relative to its containing box, a box in a *page layout* has fewer interesting metrics. It has exactly three metrics in each layout direction:

a container-relative *position*.

a *lowpart* and a *highpart*. The sum of these defines the box's width or height. The lowpart defines the origin of the box's interior coordinates.

Formally, *laying out* a page-tree is defined as *associating a set of numbers* [xContained, yContained, xLowPart, xHighPart, yLowPart, yHighPart] with each box of the page-tree.

5.4.2 Coordinate system

The coordinate system that is in force within a layed out box is defined as follows [this definition is quite analogous to Interpress]:

The initial *medium* coordinate system has coordinates defined by

The two coordinate axes are named x and y. The rectangular image includes the origin and lies in the first quadrant. The units of measurement are micas, where a mica equals 10^{-5} meter. The coordinate system is chosen so that the y axis points "up" in the normal viewing orientation.

Within the page box, the coordinates are defined by a transformation which takes a point in the box's internal coordinates into the medium coordinates via the matrix multiplication:

$$[\, x_{new}, y_{new}, 1 \,] = [\, x_{old}, y_{old}, 1 \,]\ M$$

This transformation matrix M is:

cos(a)	sin(a)	0
-sin(a)	cos(a)	0
x	y	1

where

α is the page box's coordinateRotation, with 0ccw = 0° 90ccw = 90° 180ccw = 180° and 270ccw = 270° and

x = pageBox.xSpan.lowPart.nominal and

y = pageBox.ySpan.lowPart.nominal.

Within the box on a page, the coordinates are defined by a transformation which takes a point in the box's internal coordinates into the coordinates of the containing box. This transformation matrix is:

cos(α)	sin(α)	0
-sin(α)	cos(α)	0
x	y	1

where

α is the box's coordinateRotation, with 0ccw = 0° 90ccw = 90° 180ccw = 180° and 270ccw = 270° and

x = box.xSpan.fromLowContainer.nominal + box.xSpan.lowPart.nominal and

y = box.ySpan.fromLowContainer.nominal box.ySpan.lowPart.nominal.

5.4.3 Recursive page layout

If none of the boxes on a page have SYNTHESIZED measures, then the page layout can be performed top-down. Furthermore, the role of **layoutProceeds** insures that *the two directions on the page can be laid out independently.*

If some of the boxes on a page do have SYNTHESIZED measures, then these measures are first converted to ordinary measures. This can be done bottom-up.

The algorithms, below, define how top-down page layout is performed.

5.4.4 Measure synthesis

If a **lowPartExtent** or a **highPartExtent** of a box is bound to SYNTHESIZED, we need to know how to convert the atom to a numeric meaure. The algorithms below, define how SYNTHESIZED measures are converted to numeric measures.

In the expositions below, we define a *BoxNode* as a logical "nesting" of boxes. A BoxNode *is* an outermost box (with its box relevant bindings, content, etc.), *and* a set of contained boxes. BoxNodes are passed *by value* in the procedures defined below; this is probably not appropriate for an actual implementation, but aids the exposition. Within an outer box B, there are *B.kids* contained boxes, named B_k for $0 \leq k < B.kids$. It is sometimes useful to consider the contained boxes to be numbered "in reverse"; we facilitate this by defining $B_{k?boolean}$ to mean B_k if *boolean* is TRUE and $B_{(B.kids-1-k)}$ if *boolean* is FALSE.

Given a BoxNode B (a tree of box-nodes, rooted at B), a call to *MakeMeasuresNumericXandY*(B) will return a BoxNode, similar to the first, but with all its SYNTHESIZED measures replaced by the correct numeric ones.

```
procedure MakeMeasuresNumericXandY (b: BoxNode) returns (b': BoxNode);
  begin
  b ← MakeMeasuresNumericXorY (b, x);
  b' ← MakeMeasuresNumericXorY (b, y);
  end

    procedure MakeMeasuresNumericXorY (B: BoxNode, xy: SpanName)
    returns (B': BoxNode);
    begin
    how: LayoutMethod ← if xy = x then B.xLayout.direction else B.yLayout.direction;
    glue: GlueFunction ← if xy = x then B.xLayout.siblingAdjacency
     else B.yLayout.siblingAdjacency;
    b: array [0..B.kids] of Box;

    -- copy descendent nodes to local aray, reversing if layout direction is "down" --
    for j in [0..B.kids) do b_j ← B_j?how#down;

    -- use descendent nodes' measures to calculate this node's measures --
    for am: Measure in MeasureName do
    if B.am = SYNTHESIZED then -- leave alone if already numeric --
     begin
     if how = fixed then error
     if LAYOUTINSIDES not in B_j.tags then error -- synthesized but no internal layout --
     -- am = synth, how#fixed, am is lowPartExtent or highPartExtent --
     if how = onOrigins then
      begin
      bm: array [0..B.kids) of Measure;
      for k in [0..B.kids) do
      bm_k ← if am = l then b_k.xy.l + b_k.xy.cl else b_k.xy.h + b_k.xy.ch;
      b_j.am.nominal ← MAX 0≤n<B.kids (bm_i.nominal)
      b_j.am.over ← MAX 0≤n<B.kids (bm_i.over)
      b_j.am.under ← MAX 0≤n<B.kids (bm_i.under)
      end
    else -- how = up or how = down --
      begin
      bs: array [0..B.kids) of Span;
      bm: array [0..B.kids) of Measure;
      for k in [0..B.kids) do bs_k ← b_k.xy;
      bm ← Decompose (bs, glue);
      if am = lowPartExtent then b_j.am ← bm_0 else b_j.am ← Sum (bm ) - bm_0;
     end

    -- copy descendent nodes back --
    for j in [0..B.kids) do B_j?how#down ← b_j;
    B' ← B;
    end
```

```
procedure Decompose
 (b: array [0..n) of Span,
 function: procedure(Span,Span) returns (Measure) )
 returns (a: array [0..3*n) of Measure)
begin
a0 ← b0.cl;
for i in [0..n) do a(i + 1) ← bi.l; a(3i + 2) ← bi.h; a(3i) ← function(b(i-1), b(i)) end;
a(3i + 3) ← bn.ch;
end
```

5.4.5 Fixing a box's dimensions and locations

When the SYNTHESIZED measures have been replaced with numeric measures, a page can be layed out by *fixing* the box measures: (logically) replacing the box measure triples with single values.

If a particular box has fixed dimensions, the boxes contained directly within it can be fixed with the algorithms to be presented below. Since the page-box has fixed dimensions, a page's box-tree can be layed out from the top down.

The *FixMeasuresXandY* procedure, below, performs this operation. It *fixes* each box by setting the correct value into the box's *lowPart.e*, *highPart.e*, and *containeeLow.e* in both x and y.

```
procedure FixMeasuresXandY (b: BoxNode) returns (b': BoxNode);
   begin
   b ← FixmeasuresXorY (b, x);
   b' ← FixmeasuresXorY (b, y);
   end

      procedure FixMeasuresXorY (B: BoxNode, xy: SpanName)
      returns (B': BoxNode);
      begin
      s: Span ← B.xy;
      how: LayoutMethod ← if xy = x then B.xLayout.direction else B.yLayout.direction;
      glue: GlueFunction ← if xy = x then B.xLayout.siblingAdjacency
       else B.yLayout.siblingAdjacency;

      for j in [0..B.kids) do -- get measures of all descendents .. recursively --
        bj ← MakeMeasuresNumericXorY [bj, xy];

      if how = fixed then
       begin -- each child box is layed out as though it's the only one --
       for j in [0..B.kids) do
        begin
        spaceAvailable: number ← B.xy.l.e + B.xy.h.e;
        b, b': array [0..1) of Span;
        b0 ← Bj;
        b' ← DecomposeSetGlueRecompose [b, glue, spaceAvailable];
        Bj ← b'0;
        end
       end
```

```
else if how = onOrigins then
 begin
 spaceAvailable: number ← B.xy.l.e - Bi.xy.l.e;
 two: array [0..1] of Measure ← [Bi.xy.cl, B.xy.l];
 two': array [0..1] of Span ← Recompose [ SetGlue [two, spaceAvailable] ];
 Bi.xy.cl ← two'0; Bi.xy.l. ← two'1;
 FixMeasuresXorY [Bi, xy];
 end
else if how = up or how = down then
 begin
 spaceAvailable: number ← B.xy.l.e + B.xy.h.e;
 b, b': array [0..B.kids) of Span;
 for k in [0..B.kids) do bk ← Bk?how#down;
 b' ← DecomposeSetGlueRecompose [b, how.sibGlueF, spaceAvailable];
 for k in [0..B.kids) do Bk?how#down ← b'k;
 end

end

procedure DecomposeSetGlueRecompose
 (b: array [0..n) of Span,
  function: procedure(Span,Span) returns (Measure),
  spaceAvailable: number ) returns (b': array [0..n) of Span)
begin
a: array [0..3*n) of Measure ← Decompose [b, function];
d: array [0..3*n) of Dim ← SetGlue [a, spaceAvailable];
b' ← Recompose [d];
end

        procedure Decompose
         (b: array [0..n) of Span,
          function: procedure(Span,Span) returns (Measure),
          spaceAvailable: number ) returns (a: array [0..3*n) of Measure)
        begin
        a0 ← b0.cl;
        for i in [0..n) do a(i + 1) ← bi.l; a(3i + 2) ← bi.h; a(3i) ← function(b(i-1), b(i)) end;
        a(3i + 3) ← bn.ch;
        end

        procedure SetGlue
         (b: array [0..n) of Measure,
          spaceAvailable: number ) returns (d: array [0..3*n) of Number)
        begin
        m: Measure ← Sum[b];
        delta: Number ← s - m.e;
        g: Number ← if delta>0 then delta / (m.o-m.e) else delta / (m.e-m.u);
        for i in [0..n) do di ← bi.e + g*(if g>0 then (bi.o-bi.e) else (bi.e-bi.u))
        -- s now equals Sum[d] --
        end

        procedure Recompose
         ( d: array [0..3n) of Distance ) returns (b': array [0..n) of Span)
```

```
begin
s: Number ← Sum[d];
for i in [0..n) do
b'i.cl.e ← if i = 0 then d0 else b'(i-1).cl + d(3i-2) + d(3i-1) + d(3i);
b'i.l.cl.e ← d(3i + 1);
b'i.h.cl.e ← d(3i + 2);
-- note that ch, sl, and sh are meaningless in the recomposed spans --
-- note also that o and e are meaningless in fixed measures --
end
```

5.5 Layout with templates: "Pouring"

When a solid layout is inappropriate, we separate the description of the layout and the content. Instead of placing text (and other content) within the layout boxes that geometrically contain it on the printed page, we split it out into a separate *galley*.

Then rendering must include some function which recombines the content and layout, so it can put the text in the right places on the page. We call non-solid layout *fluid*, and the act of recombining layout and galley *pouring*.

Layout is strongly affected by content, and we cannot, in general, produce a correct layout structure without knowing where the content falls. The font-size of the running text, for example, affects how many line-boxes fit into a column. So the layout supplied with the galley cannot be a precise one, but rather must supply a *template* from which the correct layout of a particular content can be constructed.

Pouring is defined to be an editing operation. It is an operation which accepts one document (a script containing layout templates and a galley separate from the layout) and outputs a different document -- a solid-layout document which has no regular-expression nodes and with the former galley arranged within the layout boxes.

An editor may choose to make the document that is the result of pouring a transient one, by using it to render the document and then discarding it. Or an editor may make the solid-layout document available to its user, perhaps by emitting the script that corresponds to the "solid" document.

5.5.1 Pour node

A pour operation is performed under the control of, and within the scope of, a *pour* node. The pour node defines the *template* that controls the target layout. The pour node's content is the *galley nodes*. The pour node controls the pour by defining the *set of relevant labels* that will be used to match up nodes in the galley with nodes in the template.

5.5.2 Pouring streams

In order to perform a "pour," the editor must be able to match up layout and galley. Conceptually, the galley consists of a set of *streams* that will be poured into a set of *containers*. *Streams and containers are matched via labels.*

A pour operation involves a *template T*, a *galley G*, and a set of relevant *labels L*. Both the template and the galley are nodes.

In order to talk about the process of matching nodes from the template with nodes from the galley, it is convenient to number the nodes. We arrange to number the interesting nodes of the template and the galley in depth-first order, using the function Number (x: Node, p: Predicate) where p is a predicate on the node. The result of Number (x, p) is a tree shaped like x, [except that descendants of nodes satisfying p have been discarded] but with each node of that tree replaced with a triple [node, number, last]. The node value of the triple is exactly the original node; the number field indexes the nodes satisfying the predicate in depth-first order; the last field is just for use within the Number function procedure. A node not satisfying the predicate is numbered 0; when a node satisfies the predicate, its substructure is not numbered.

```
procedure Number (x: Node, p: Predicate, i: number) returns (nx: NumberTriple);
  begin
  if p(x) then nx ← [node: x, number: i + 1, last: i + 1]
  else if x.kids = 0 then nx ← [node: x, number: 0, last: i]
  else
   begin
    nx ← [node: nt, number: 0, last: i];
    for k in [0..x.kids) do
      begin
      nx.node_k ← Number (nx.node_k, p, nx.last);
       nx.last ← nx.node_k.last;
      end
   end
  end
```

If *nt* is a numbered tree, we write nt_k for the *node field* of the node with *number* = *k*.

The set of relevant labels L determines the *interesting* nodes: those containing at least one label from L. Each subset LS of L determines a sequence or *stream* of interesting nodes in the template and the galley: those containing all the labels in LS and no labels in L-LS. Thus if L = {red, green}, the streams are determined by {red}, {green}, and {red, green}. When we are considering a set of relevant labels L, we use symbols to denote a template and a galley which consist of the interesting nodes:

$$t^\circ \equiv Number\,(t, \lambda\, x \text{ IN } x.labels \cap L \neq \varnothing \wedge \text{MOLD\$} \in x.tags, 0)$$
$$g^\circ \equiv Number\,(t, \lambda\, x \text{ IN } x.labels \cap L \neq \varnothing \wedge , 0)$$

We also define the notion of the predecessor/successor relationship of nodes within a stream. A node *i* is the predecessor of a node *j* in some stream if and only if they are both in the stream, *i* preceeds *j*, and there is no node *k* which is in the stream and between them:

$$i \rightarrow_x j \equiv i<j \wedge (x_i.\text{labels} \cap L = x_j.\text{labels} \cap L \vee i=0) \wedge$$
$$(x_k.\text{labels} \cap l = x_j.\text{lables} \cap L \Rightarrow k<i \vee k>j)$$

We can now define a relation between node numbers which pairs interesting nodes in a tamplate and a galley which are in the same position in the same stream. Two such nodes *correspond* to one another. We write "↔" for correspond:

$$i \leftrightarrow_{t,g} j \equiv i=j=0 \vee (\exists k, l: k \rightarrow_{t^\circ} i \wedge l \rightarrow_{g^\circ} j \wedge k \leftrightarrow l]$$

A node *g* in a galley *satisfies* the corresponding node *t* in a template if one is the Atom MATCH, or both are nodes and *g* has every tag that *t* has, and for each attribute *a* of *t*, *g.a* satisfies *t.a*. When *g* satisfies *t*, we can define molded by *t* to be *g*, with every attribute replaced by the attribute of *t* which it satisfies.

A pour *fails* unless each stream from the template is the same length as the corresponding stream from the galley, and each interesting node of the galley satisfies the corresponding node of the template. If the pour does not fail, we say that the template *accepts* the galley.

$$t \angle^? g \equiv t^\circ.\text{last} \leftrightarrow_{t,g} g^\circ.\text{last} \wedge i \leftrightarrow_{t,g} j \Rightarrow g^\circ_j \text{ satisfies } t^\circ_i$$

If a pour does not fail, then we can define the result of the pour. It is the template with each interesting node *t* replaced by the corresponding node of the galley *g molded by t*. We write "t∠g" for the result of the pour.

G molded by t is defined to be g, modified as follows:

every MATCH attribute is replaced by the attribute of t which it satisfies

the quoted expression bound to **coerce** is executed in the environment of the result of the pour.

Often it is useful to have a less restrictive version of pouring. We say that a template *t* accepts a prefix of a galley *g* if there is some way of extending it to accept the whole galley, i.e. some other template *u* such that *tu* accepts *g*. In this case the result of the pour is just the part of *tu*∠*g* that comes from *t*. We use the "◭" character to indicate such partial pours:

$$t \blacktriangleleft^? g \equiv \exists u: tu \angle^? g$$
$$t \blacktriangleleft g \equiv x \in \exists u, y: [tu \angle^? g \wedge tu \angle g = xy] \text{ defined only if } t \blacktriangleleft^? g$$

We also define the portion of a galley that remains after a prefix of it has been poured. If a prefix of the galley *g* is poured into the template *t*, then the remainder is *g-t* defined by:

$$g\text{-}t \equiv \text{Purge}(g^\circ, t^\circ) \text{ defined only if } t \blacktriangleleft^? g$$

```
procedure Purge (g: NumberTree, t: NumberTree) returns (g': Node);
  begin
  if g.isLeaf then g' ← g.node
  else if g.number ≠ 0 ∧ ∃ i<t°.last ↔t,g g°.number THEN NIL then g' ← NIL
  else begin g' ← g.node; for k in [0..g'.kids) do g'k ← Purge ((g'k)°, t); end
  end
```

5.5.3 Best pours and good pours

Given a penalty function P and a set S of values, we can define the subset of *best* values as the subset whose penalty is as good as possible:

$$S^{best} = \{s \in S: (\forall s' \in S)\ P(s') \leq P(s)\}$$

This is most interesting when used for the results of pouring a galley into the (templates resulting from) a regular expression.

We would like to say that, in performing a pour, the editor must choose a *best* template in the above sense.

Stating that the editor must pick a template which minimizes total penalty (which is about what we did in section 4.4.2.4 for solid layout) is unreasonable, as that potentially involves searching a huge space for a global minimum, and there is no clear strategy.

We bound the editor's search space by defining the *valid local minimum*. An editor may choose to search further for a better layout, but is deemed successful with any layout that is a member of the regular expression choice-space and has a penalty that is a valid local minimum. Such a pour will be called a *good* one.

5.5.4 Local minima

When performing a trial "pour" into a *choice* or *repetition* regular expression, an editor may compute the penalty associated with the first alternate, then the second, etc. (considering the repetition as an infinite alternation of the form a | aa | aaa | aaaa ...).

If the best alternative has a penalty less than **satisfactionThreshold** then the regular expression succeeds *at this level.*

Finding the best alternative may involve examining an infinite number of alternatives. So we provide a search-bounding rule: an editor may choose the best from *among the subset it has examined* providing that has examined a compact subset from the left and:

> It has examined aa compact subset from the left and
>
> it has found an alternative with penalty less than **satisfactionThreshold** and
>
> either a) the n alternatives at the right of the examined subset are no better than the current best alternative and $n >$ **satisfactionForwardSearch** or b) some alternative to the right of the current best alternative is at least **satisfactionUpwardSearch** worse in penalty.

The above rule applies *at a particular regular expression.* Having made a choice at one level, that choice must be propagated upward in the regular expression tree, and a higher regular expression may prove unsatisfiable *owing to the choice made here.*

5.5.5 Fences

Another aid in performing a pour is provided by the *fence*. A fence is a fence-node appearing in a template. It does not participate in the pour directly (it is not a mold) but it serves to split the template into parts. If we indicate a fence by ||, then we can consider a template t which is broken by one or more fences:

$$t = t_1 \| t_2 \| \ldots \| t_n$$

Given the presence of the fences, we redefine an *good* pour into *t* to include the case where the pour into each part t_n is *good* and the galley is used up by the last pour.

> If $t = t_1 \| t_2 \| t_3 \ldots \| t_n$,
> and $t_1 \measuredangle^? g$, and $t_2 \measuredangle^? (g\text{-}t_1)$ and $t_3 \measuredangle^? (g\text{-}t_1\text{-}t_2) \ldots$ and $t_n \angle^? (g\text{-}t_1\text{-}t_2 \ldots \text{-}t_{n-1})$
> and those pours are *good*,
> then the pour $t \angle^? g$ is *good*
> and the result is exactly the concatenation of the pours
> $t_1 \measuredangle g \;\; t_2 \measuredangle (g\text{-}t_1) \;\; t_3 \measuredangle (g\text{-}t_1\text{-}t_2) \ldots t_n \angle (g\text{-}t_1\text{-}t_2 \ldots \text{-}t_{n-1})$.

5.5.6 Multi-level pours

Frequently, one our operation will cascade into another. Characters will be poured into lines, for example, and then the lines poured into columns.

This cannot be treated purely as one complete pour followed by another, since the two pours influence one another. In the above example, the lengths of the lines are not fixed until the second pour, and yet the line lengths obviously affect how many characters go in each line.

Thus the cascaded pours must be treated as one logical operation.

Given that the first pour results in a number of best/good results, we cannot, at that time, decide on one and continue. Rather we must, logically, use all the initial pours (where the template accepts the galley) as input to the second pour, and only choose after all pours have been completed.

5.5.7 Pour "leftovers"

Multi-level pours introduce another phenomenon: the "leftover" which is the labelled galley node which does not participate in the pour because its lable(s) is not relevent to the pour. In the case of a single pour, these nodes were effectively discarded, since they did not get into a template and the template was the result of the pour.

But in th multi-level pour case, we must consider the case where a node in the galley is labelled with a label which will be relevent to the second pour. We do not want to lose this node during the first pour. So we state:

> Nodes which belong to no stream of apour are appended, after the pour, to the result of the pour.

Note that this rule puts the irrelevent nodes *outside* the filled template. Thus, for example, a node which is not poured into a page during the final pour of a document will be effectively dropped, since it will not be on any page-image.

5.6 Figures

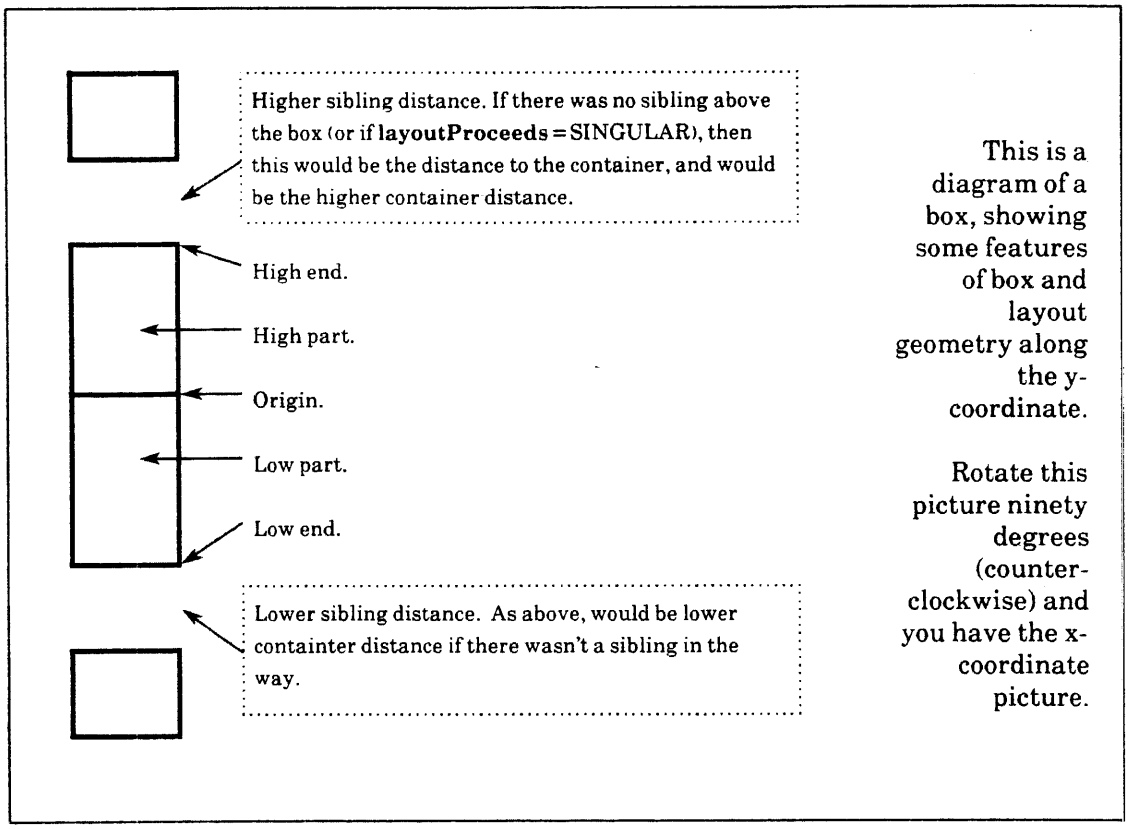
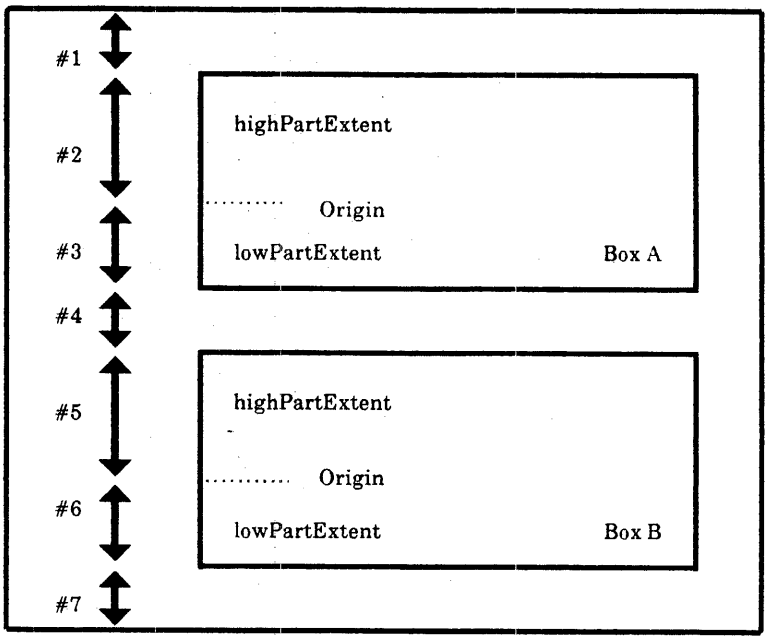


Figure 5-1: one-dimensional box metrics



The seven distances are related to the contained boxs' measures as follows:

#1 relates to BoxA.ySpan.highContainer.

#2 relates to BoxA.ySpan.highPartExtent .

#3 relates to BoxA.ySpan.lowPartExtent .

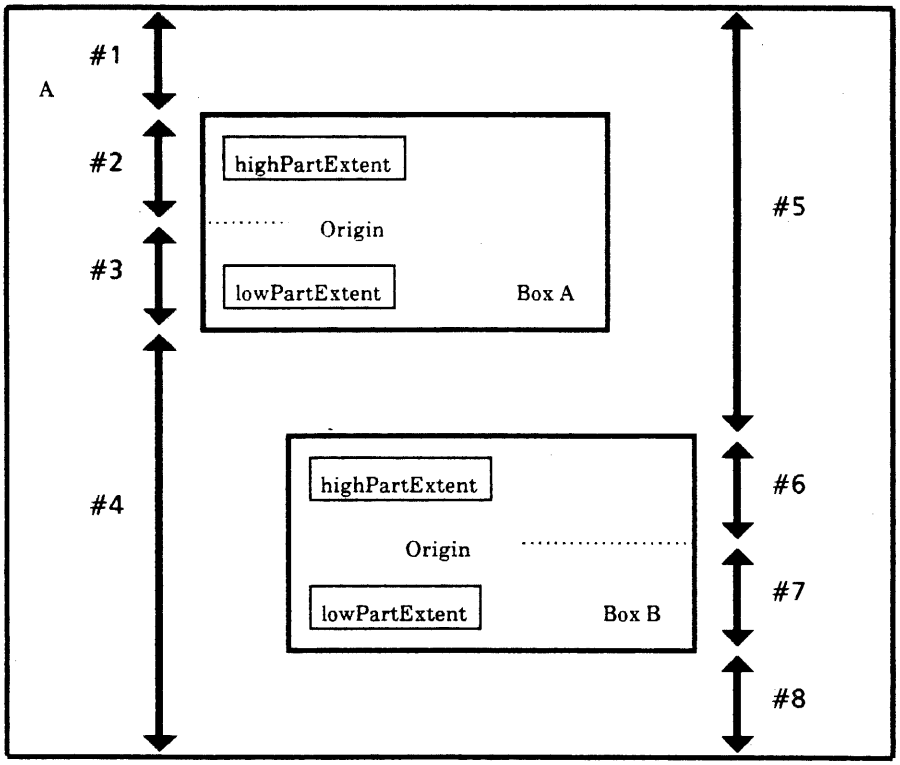
#4 relates to BoxA.ySapn.lowSibling and BoxB.ySpan.highSibling, with the actual measure used in the penalty calculation being derived by the method indicated by ContainingBox.ySpan.siblingAddition.

#5 relates to BoxB.ySpan.highPartExtent .

#6 relates to BoxB.ySpan.lowPartExtent .

#7 relates to BoxB.ySpan.lowContainer.

Figure 5-2: A layoutProceeds ≠ SINGULAR span



The eight distances are related to the contained boxs' measures as follows:

#1 relates to BoxA.ySpan.highContainer.
#2 relates to BoxA.ySpan.highPartExtent.
#3 relates to BoxA.ySpan.lowPartExtent.
#4 relates to from BoxA.ySpan.lowContainer.

#5 - #8 are similar but use Box B's measures.

Figure 5-3: A layoutProceeds = SINGULAR span

B

Appendix B
Example of Script Evaluation

This appendix contains a simple example of a script as it is

(a) written in the publication encoding,

(b) represented in terms of the abstract grammar, and

(c) represented in terms of value items.

In the process of going from (a) to (b) the script is parsed; in going from (b) to (c) the parse tree is evaluated using the semantics defined in section 2.3.

The source script

```
{aTag$
  relV1 ←0                    -- relV1 is a relevant attribute of an X node--
  v ←relV1 ↑ +5          -- v is not a relevant attribute of an X node--
  q%←          -- bind a quoted expression to q, which is not a relevant attribute--
     '{"FalseString" "TrueString"}!(relV1 ↑ LT v ↑ )' -- conditional expression--
  "content"                   -- simple string as content--
  q%           -- evaluate q (remembering that the result came from q)--
  }
```

For purposes of this example, assume that the tag X is defined as follows:

```
aTag%←{TAG$
  attributes ← {
     relV1%← Number ↑
     relV2%← {String ↑ | default←"relV2 default value"}}
  contentType ← String ↑
  reducesTo ← NIL}
```

Abstract version of the source script

```
script(
  node( items(items(items(items(items(items(
     item( tag(name(id(X))))                    -- aTag$--
     item( binding(name(id(relV1)) term(primary(number(0))))))     -- relV1 ←0--
     item( binding(              -- v ←relV1 ↑ +5--
        name(id(v))
```

```
            term(
                term(primary(invocation(name(id(relV1)))))
                op(ADD)
                primary( number(1))
                )
            ))
        item( sBinding(      -- q%←--
            name(id(q))
            sRhs(quoted(term(
                term(primary(
                    node(items(items( -- '{"FalseString" "TrueString"}!(relV1 ↑ LT
                            v ↑)'--
                        item(term(primary(literal(string("FalseString")))))
                        item(term(primary(literal(string("TrueString")))))
                        )))
                    ))
                op(SUBSCRIPT)
                primary(term(
                    term(primary(invocation(name(id(relV1)))))
                    op(LT)
                    primary(invocation(name(id(v))))
                    ))
                )))
            ))
        item( term(primary(literal(string("content"))))))    -- "content"--
        item( indirection(name(id(q))))      -- q%--
        )))))))
    )
```

Elaboration of abstract script's semantics

```
[m: node,
  cVs: LIST[
      [m: tag, xId: aTag],
      [m: bindStruc,
          cV: [m: quotedTerm,
              term(
                  term(primary(
                      node(items(items(
                          item(term(primary(literal(string("FalseString")))))
                          item(term(primary(literal(string("TrueString")))))
                          )))
                      op(SUBSCRIPT)
                      primary(term(
                          term(primary(invocation(name(id(relV1)))))
                          op(LT)
                          primary(invocation(name(id(v))))
                          ))
                      )],
              xId: q],
          [m: string, cStr: "content"],
```

```
    [m: evalStruc,
        [m: vOfQ,            -- note the carried-along environment--
            [m: string, cStr: "TrueString"],
            xEnv: LIST[[m: bind, cV: [m: num,0], xId: relV1] [m: bind, cV: [m: num,5],
                    xId: v]]
            ],
        xName: LIST[$q, NIL],
    [m: bind, cV: [m: num, 0], xId: relV1],        -- an explicit binding--
    [m: bind, cV: [m: string, "relV2 default value"], xId: relV2]      -- from aTag's
            definition--
    ]            -- end of cVs LIST
]              -- end of the node
```

Appendix P
Paragraph% reducesTo

```
[ To save space in what follows, I abbreviate as follows:
    { MEASURE$ under ← a nominal ← b  over ← c } = > { MEASURE$ a b c }
    { MEASURE$ under ← a nominal ← a  over ← a } = > { MEASURE$ a } ]

PARAGRAPH% ← { TAG$ ....
    .....
    reducesTo ← { POUR$ -- nested paras are not in yet --
        leftLineMargin ← firstLineLeftMargin
        rightLineMargin ← rightMargin
        topLineMeasure ← { MEASURE$ prelead }
        bottomLineMeasure ← { MEASURE$ leading }
        beginningPad ← NOT lineJustification AND
                (lineRaggedness = atBeginning OR lineRaggedness = centered)
        endingPad ← NOT lineJustification AND
                (lineRaggedness = atEnd OR lineRaggedness = centered)
        firstLineBox ← { LINE$ }
        leftLineMargin ← leftMargin
        topLineMeasure ← { MEASURE$ leading }
        middleLineBox ← { LINE$ }
        bottomLineMeasure ← { MEASURE$ postlead }
        beginningPad ← (lineRaggedness = atBeginning OR lineRaggedness = centered)
        endingPad ← (lineRaggedness = atEnd OR lineRaggedness = centered)
        lastLineBox ← { LINE$ }
        topLineMeasure ← { MEASURE$ prelead }
        firstAndLastLineBox ← { LINE$ }
        template ← { expresses ← alternation
            { -- single line -- firstAndLastLineBox }
            { -- two lines --
                { TOGETHER$ penalty ← widowControl * 1000 + orphanControl * 1000
                    firstLineBox lastLineBox }
            { -- three lines is ugly .. use a select on widow and orphan control --
                { -- select: zero if neither or both, one if only widow, two if only orphan --
                    { TOGETHER$ penalty ← widowControl * 1000
                        firstLineBox middleLineBox lastLineBox }
                    { { TOGETHER$ penalty ← widowControl * 1000
                        firstLineBox lineBox } lastLineBox }
                    { firstLineBox
```

```
                {TOGETHER$ penalty ← orphanControl * 1000
                lineBox lastLineBox } }
            }( (widowControl#orphanControl) *
                (widowControl + orphanControl + orphanControl ) ) }
    { -- four or more lines --
        { { TOGETHER$ penalty ← widowControl * 1000
            firstLineBox middleLineBox }
          { TEMPLATE$ expresses ← repetition middleLineBox }
        { TOGETHER$ penalty ← orphanControl * 1000
            middleLineBox lastLineBox } } } }
```

Appendix Q
Line% reducesTo

```
LINE% ← { TAG$ ....
    .....
    reducesTo ← { -- first we illustrate the non-tab case .. as it is easier to understand --
        stretchySpan ← { SPAN$
            lowPartExtent ← { MEASURE$ 0 0 99999 }
            highPartExtent ← { MEASURE$ 0 0 99999 }
            fromLowContainer ← { MEASURE$ 0 0 99999 }
            fromHighContainer ← { MEASURE$ 0 0 99999 }
            fromLowSibling ← { MEASURE$ 0 0 99999 }
            fromHighSibling ← { MEASURE$ 0 0 99999 } }
        stretchyBox ← { BOX$ xSpan ← stretchySpan ySpan ← stretchySpan }
        leftLineMeasure ← { MEASURE$ leftLineMargin }
        rightLineMeasure ← { MEASURE$ rightLineMargin } --
        { BOX$ LABEL$ POUR$ labels ← {uniqueBodyText}
            xSpan ← { SPAN$
                lowPartExtent ← SYNTHESIZED
                highPartExtent ← SYNTHESIZED
                fromLowContainer ← { MEASURE$ leftLineMargin }
                fromHighContainer ← { MEASURE$ rightLineMargin } }
            ySpan ← { SPAN$
                lowPartExtent ← belowBaseline highPartExtent ← aboveBaseline
                fromLowSibling ← { MEASURE$ topLineMargin }
                fromHighSibling ← { MEASURE$ bottomLineMargin } }
            coordinateRotation ← 0
            {NIL stretchyBox}(beginningPad) -- stretchy iff beginningPad -- }
            template ← { TEMPLATE$ expresses ← alternation
                { TEMPLATE$ expresses ← repetition { MOLD$ PSEUDOCHAR$ } }
                { TEMPLATE$ expresses ← sequence
                    { TEMPLATE$ expresses ← repetition { MOLD$ PSEUDOCHAR$ } }
                    { MOLD$ PSEUDOCHAR$ FILL$ container ← Line } } }
            { NIL stretchyBox } ( endingPad ) } }
```

```
LINE% ← { TAG$ ....
    .....
    reducesTo ← { -- this is the real case which handles tabs --
        stretchyMeasure ← { MEASURE$ under ← 0 nominal ← 0 over ← 99999 }
        stretchySpan ← { SPAN$
            lowPartExtent ← { MEASURE$ 0 0 99999 }
            highPartExtent ← { MEASURE$ 0 0 99999 }
            fromLowContainer ← { MEASURE$ 0 0 99999 }
            fromHighContainer ← { MEASURE$ 0 0 99999 }
            fromLowSibling ← { MEASURE$ 0 0 99999 }
            fromHighSibling ← { MEASURE$ 0 0 99999 } }
        stretchyBox ← { BOX$ xSpan ← stretchySpan ySpan ← stretchySpan }
        leftLineMeasure ← { MEASURE$ leftLineMargin }
        rightLineMeasure ← { MEASURE$ rightLineMargin }
        tabstopZeroBox ← { BOX$ POUR$ LABEL$ labels ← {uniqueBodyText}
            xSpan ← { SPAN$
                lowPartExtent ← SYNTHESIZED
                highPartExtent ← SYNTHESIZED
                fromLowContainer ← 0
                fromHighContainer ← 0 }
            ySpan ← { SPAN$
                lowPartExtent ← belowBaseline highPartExtent ← aboveBaseline }
            { NIL stretchyBox } (beginningPad )
            template ← { TEMPLATE$ expresses ← repetition
                { MOLD$ PSEUDOCHAR$ } }
            { NIL stretchyBox } (endingPad ) } }
        -- "tabstopBoxOne" needs to be repeated for each defined tabstop --
        -- the recursion to accomplish this would overly complicate an already --
        -- complicated draft example --
        tabstopOne ← tabstops!0
        -- this draft handles left, centered, and right tabstops but not aligned tabstops --
        tabstopOneBox ← { BOX$ POUR$ LABEL$ labels ← {uniqueBodyText}
            xSpan ← { SPAN$
                lowPartExtent ← { {MEASURE$ 0} {MEASURE$ 99999} }
                    ( tabstopOne.type = centered OR tabstopOne.type = right )
                highPartExtent ← {
                    {MEASURE$ 0 0 99999} {MEASURE$ 99999} {MEASURE$ 0} }
                    ( tabstopOne.type -- left=0 centered=1 right=2 -- )
                fromLowContainer ←
                    { MEASURE$ tabstopOne.position - lowPartExtent.nominal } }
                fromHighContainer ← { {MEASURE$ 0 0 99999} {MEASURE$ 0} }
                    ( tabstopOne.type = left ) -- if left, box must end at line end --
                fromHighContainer ←
                    { MEASURE$ tabstopOne.position - lowPartExtent.nominal } }
            ySpan ← { SPAN$
                lowPartExtent ← belowBaseline highPartExtent ← aboveBaseline }
            { NIL stretchyBox } ( tabstopOne.type = centered OR tabstopOne.type = right )
            template ← { TEMPLATE$ expresses ← sequence
                { MOLD$ TAB$ }
                { TEMPLATE$ expresses ← repetition { MOLD$ PSEUDOCHAR$ } } }
            { NIL stretchyBox } ( tabstopOne.type = left OR tabstopOne.type = centered )
        fillLineBox ← { BOX$ POUR$ LABEL$ labels ← {uniqueBodyText}
```

```
    xSpan ← { SPAN$
        lowPartExtent ← 0
        highPartExtent ← 0
        fromLowContainer ← 0 -- area not interesting -- }
    ySpan ← { SPAN$
        lowPartExtent ← 0 highPartExtent ← 0 }
    template ← { FILL$ container ← Line }
 { BOX$ INSIDELAYOUT$
-- this is the line box .. it contains a box for each tabstop's characters --
    clips ← TRUE
    xLayout ← {INSIDELAYOUTMETHOD$ direction ← fixed}
    xSpan ← { SPAN$
        lowPartExtent ← SYNTHESIZED
        highPartExtent ← SYNTHESIZED
        fromLowContainer ← leftLineMeasure
        fromHighContainer ← rightLineMeasure }
    ySpan ← { SPAN$
        lowPartExtent ← belowBaseline highPartExtent ← aboveBaseline
        fromLowSibling ← topLineMeasure
        fromHighSibling ← bottomLineMeasure }
    coordinateRotation ← 0
    tabstopZeroBox  tabstopOneBox  -- etc --  fillLineBox
    { NIL stretchyBox } ( endingPad ) }
```

Glossary

Italics indicate words defined in this glossary.

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

attribute: a (name, value) pair, identified by its name, which is bound to a value

atom: a name denoting only itself. Occurrences of the same atom in different parts of a script all denote the same primitive value

base language: the part of the *Interscript* language that is independent of the semantics of particular *tags*

base semantics: the semantic rules that govern how *scripts* in the *base language* are *elaborated* to determine their *contents* and *attributes*

binding: the operation of associating a *value* with a *name*; also the resulting association

contents: the vector of *values* denoted by a *node* of a *script*

document: the *internalization* 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*

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*. The encoding presented in this standard is the *publication encoding* for writing about Interscript.

environment: a *value* consisting of a sequence of *attributes*. An environment may be either *free-standing* or *nodal*. A free-standing environment is a structured value much like a record, with the components being the *attributes* of the environment. A nodal environment is associated with a *node* of a *script* and represents the *attributes* bound in that node.

expression: a syntactic form denoting a *value*

external environment: a standard *environment* in whose scope an entire *script* is *elaborated*. This environment is named *X*.

externalization: the process of converting from a *document* to a *script*.

fidelity: the extent to which an *externalization* or *internalization* preserves *contents*, form, and structure

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

indirection: the appearance of a *name%*, denoting the evaluation of *name* plus the necessity of remembering that the result came from evaluating *name*

internalization: the process of converting from a *script* to a *document*; also the result of that process

Interscript: the name of this editable document standard

invocation: the appearance of a *name* ↑ (see also *indirection*)

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

node: everything between a matched pair of {}s in a *script*; this may represent a branch point in a *document's dominant structure*, or it may simply be a structured value acting as a vector, a record-like value, or a mixture of the two.

number: the *Interscript primitive type* for representing numeric values.

primitive type: Boolean, *number*, *string*, or *atom*

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

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

publication encoding: the *encoding* for *scripts* used for examples to be read by people rather than editing systems.

qualified name: see *hierarchical name.*

quoted expression: a *value* which is an *expression* bracketed by single quotes (""""); the expression is evaluated by *invoking* the *name* (with or without *indirection*).

real: a floating point number

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 *externalizing* a *document*

string: a *literal* which is a sequence of characters bracketed by double quote marks, 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*

tag: an *atom labelling* a *node* using the syntax *atom*$; the properties of a *node* correspond to the set of tags labelling it

value: a *primitive value*, *node*, or *quoted expression*

X: the standard outer *environment* for an entire *script*