File: Interscript-2.bravo
Last edited by Mitchell, August 31, 1982 4:58 PM

2. The Language Basis: Syntax and Semantics

2.1. Grammar

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

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

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

parentheses ( ) are used purely for grouping.

script	::=	header node trailer
header	::=	"Interscript/Interchange/1.0 "
trailer	::=	"EndScript"
item	::=	content | binding | label
content	::=	term | node
term	::=	primary | primary op term
op	::=	"+" | " " | "*" | "/"
primary	::=	literal | invocation | indirection | application | selection | vector
literal	::=	Boolean | integer | real | string | universal
invocation	::=	name
name	::=	id ("." id)*
indirection	::=	name "%"
application	::=	(name | universal) "[" item* "]"
universal	::=	ucID
selection	::=	"(" term "|" item* "|" item* ")"
vector	::=	"(" item* ")"
node	::=	"{" item* "}"
binding	::=	localBind | globalBind
localBind	::=	name "_" rhs
globalBind	::=	(name | universal) ":=" rhs
rhs	::=	content | op term | "'" item* "'" | "[" item* "|" binding* "]"
label	::=	tag | link
tag	::=	universal "$"
link	::=	"LINKS" id | "^" name | name ":"

2.2. Discussion of Features

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

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

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

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

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

invocation ::= id

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

invocation ::= name "." id

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

indirection ::= name "%"

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

universal ::= ucID

Universals are identifiers that are written entirely in upper case letters. They are presumed to be defined externally, so they are not looked up in the environment (with one exception see the discussion of tags below).

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

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

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

Neither form of application changes the environment of succeeding expressions because item* is evaluated in a free-standing environment that is thrown away.

selection ::= "(" term "|" item1* "|" item2* ")"

This is a standard conditional item sequence, using syntax borrowed from Algol 68. The value and effect are those of item1* if the term evaluates to "T" in the current environment, those of item2* if it evaluates to "F".

vector ::= "(" item* ")"

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

node ::= "{" item* "}"

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

item* ::= ""

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

item* ::= item1 item*

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

localBind ::= name "_" rhs

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

globalBind ::= (name | universal) ":=" rhs

This adds a single binding to the outermost environment X. It makes sense to bind something to a universal only if the universal is a tag name (see tag below).

binding ::= name mode op term

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

mode ::= "_" | ":="

A value can be bound to a name either locally ("_") in the environment of the node in which the binding appears, or globally (":=") in the environment of the root node of a script.

rhs ::= "'" item* "'"

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

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

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

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

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

tag ::= universal "$"

This gives the containing node the property denoted by the universal. It also looks for a binding to the universal in X, the outermost environment; if one exists, it is invoked in the context of the current environment. This gives an easy way to attach a tag to a node and provide a set of defaults associated with the tag.

link ::= "LINKS" id

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

link ::= "^" name

This identifies the immediately containing node as a source of the link name (like a reference to the set of nodes which are link targets).

link ::= name ":"

This identifies the immediately containing node as a target of each of the links that is a prefix of name. For example, the link target "$id_1.id_2...id_n$:" would make the node containing it a target in the link sets for id_1, $id_1.id_2$, ..., $id_1.id_2...id_n$.

2.3. Safety Rules for Low-capability Editors

Interscript claims to make it possible for editors to manipulate the parts of documents they understand without harming parts they do not. This section develops a set of conservative rules for editor treatment of script nodes created by other editors.

We first need to define some terms. The implementor of an editor is said to *understand a tag*, T, if

(1) she knows the set of attributes and contents that are relevant to T, and

(2) she knows all the invariants among attributes that must be maintained for a node with tag T.

An editing system is said to *understand a tag*, T, if

(1) it is able to provide some rendition (display) of a node with tag T; and

(2) it allows insertion or deletion of direct subnodes of that node.

An editing system is said to *implement a tag* if

(1) it understands T; and

(2) it is able to alter a node with tag T.

Finally, an editing system is said to *fully implement a tag* if it is capable of changing *any* attribute relevant to T or *any* contents of a node with tag T.

With these definitions, we can now give some conservative rules for editors in treating parts of documents corresponding to nodes in a script:

It's OK for an editor to display a node if

it understands at least one of its tags.

It's OK for an editor to edit within a node if

it implements *all* of its tags, and either

(a) doesn't remove any of them, or

(b) also understands *all* tags of its parent.

It's OK for an editor to copy a node if

it understands *all* the tags of the node's new parent,

no labels are moved outside their scope, and

the two environments have the same bindings for all attributes that the editor either

doesn't understand, or

knows can't be relevant anywhere in the node or its subnodes.

It's OK for an editor to delete a node if

it understands *all* the tags of its parent.

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

2.4. **Encodings**

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

The script for a document can be encoded in many different ways. This section gives the rules for designing encodings. The purpose of these rules is to ensure that information is not lost or added by conversions from one encoding to another. There are two types of encodings: a single interchange encoding and many possible private encodings.

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

2.4.1. The interchange encoding

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

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

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

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

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

There are several reasons for this choice:

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

Control characters may be removed or inserted freely by software that receives the

interchange encoding. In this way, the receiving software can adhere to any restrictions imposed by its operating system.

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

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

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

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

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

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

Following the versionId is a node constituting the body of the script which is in turn followed by the trailer of a script, "ENDSCRIPT". The body of the script contains values encoded as follows.

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

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

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

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

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

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

String. A text vector usually contains integers that are interpreted as character codes. Often these codes lie in the range 32 to 126 inclusive, which are the numbers assigned to the characters of the interchange set by ISO 646. It is convenient to encode an element of such a vector by the character whose ISO code is the desired value. Such a string can be encoded by surrounding the characters with "<" and ">", thus "<Hello!>". If the string contains elements outside the allowed range (i.e., if the value is less than 32 or greater than 126) or the value 62 or 35 (the ISO codes for the characters ">" and "#"), those elements must be represented as integers inside "#" brackets, as described above. The two-character encoding of small integers is designed to make escape sequences compact. Thus "<Hello!>", "<Hello#CB#>", and "<Hel#GMGP#!>" are all equivalent.

Universal names. A universal is encoded by giving a name that begins with an uppercase letter followed by zero or more uppercase letters or digits, followed by a delimiter. E.g., "TEXT ", "XEROX860 ".

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

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

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

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

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

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

2.4.2. Normalization

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

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

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

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

Comments are omitted.

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

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

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

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

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

2.4.3. Level restriction

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

2.4.4. Private encodings

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

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

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

NP, the normalization function for P; see 2.4.2.

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

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

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

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

NPI=NIoCPI

NIP=NPoCIP

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

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

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

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

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

3. **Higher-Level Issues**

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

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

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

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

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

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

Symbolic values used instead of numbers:

supply standard values for the symbolic values:

Standard:	lineLeading _ 1*pt	-- some numeric value --
Editor-specific:	lineLeading _ single	
mapping:	single = 2*pt	

Different names used for standard names:

supply a binding to the standard name from the editor-specific name using a quoted expression so that it is only evaluated when needed in a right hand context:

Standard:	lineLeading _ 2*pt
Editor-specific:	lineSpace _ single
mapping:	lineLeading _ 'lineSpace'

Different concepts used for standard ones:

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

Standard:	lineLeading _ 2*pt	
Editor-specific:	lineSpacing _ [fontSize_10 on_14 leading_1]	-- lineSpacing units assumed to be pts --
mapping:	lineLeading _ 'pt*Spacing.on Spacing.fontSize'	-- compute result in standard units --

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

3.2. **Standard External Environment**

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

3.2.1. *Units*

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

```
meter=1.0                          -- IN TERMS OF METERS --
mica=1.E 5*meter        -- mica          = 1.E 5
inch=2540*mica          -- inch          = 2540 --
pt=.013836*inch         -- pt            = 35.143 --
pica=12*pt              -- pica          = 421.752 --
tenPitch=inch/10        -- tenPitch      = 254 --
twelvePitch=inch/12     -- twelvePitch   = 211.667 --

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