The Image Understanding Environment: Data Exchange

J.L. Mundy and R. Welty and Keith Price and IUE Committee*
mundy@crd.ge.com
G.E. Corporate Research and Development
Schenectady, NY 12309

Abstract

A major activity of the IUE committee is the design of a data exchange standard for IU algorithm results. The exchange standard is formulated according to object-oriented design principles and is based on the class hierarchy of the IUE specification. This paper provides an overview of the exchange format.

Introduction

As the design of the IUE progressed, it became clear that the concepts for IU data structures and operations could be applied to the formulation of a data exchange standard for IU research and application results. Such a standard is badly needed since two application-oriented programs are now underway at DARPA which involve the cooperation of a large number of research groups. One of these projects, RADIUS\(^1\) involves a number of university and other research institutions who are developing IU algorithms to support site modeling and image analysis. The second project is the Unmanned Ground Vehicle(UGV) project which is focussed on autonomous navigation and reconnaissance. It is clear that both of these projects can benefit from the capability to exchange detailed results of algorithms such as image segmentation, feature grouping and model matching.

The IUE data exchange standard based on an object-oriented representation of the main structures used in IU research and applications. The primary emphasis is on the relationship between image signal data and geometric structures. Much of IU research is involved with grouping and matching of geometry derived from images. Another major area of processing and representation is associated with the derivation of camera parameters associated with camera calibration and camera motion.

The design of the exchange format is based on these design principles: character based formats (ASCII) will be used for simplicity, object descriptions will be stored in Lisp-like lists, and the format facilitates the transfer of data between different systems.

Character format The first design choice eliminates binary formats, which may be necessary for efficient storage of some objects, but simplifies transportability between different systems. Note that this format is not primarily for the storage of image data, but for the storage of more geometric and relational IU object data.

LISP-like syntax The second principle implies only that parentheses (or another suitably defined macro characters and reserved words) surround the data. Otherwise the format is relatively free-form. Since Lisp has a simple syntax, this assumption provides a small set of delimiters to break the data and an easy mechanism to read the data in Lisp. For the C++ implementation, the parsing will be straightforward and through the use of a few key words, the format can be efficiently parsed by Lex and Yacc parsing mechanisms.

Free form output By expecting the user to have relative freedom in the output sequencing, we are not required to analyze the data to find relatively efficient formats. The user will specify what objects are to be saved, the order of the objects and the set of object slots that are included. This approach simplifies the output process, but does require the user to insure that all the required object instances are defined before used by other classes. The design assumes that the format conversion is a single pass operation.

Portability The final principle requires a format that is easily read and written in either Lisp or C, one that is not dependent on the host machine, and one that can be transformed into other internal representations independent of the Image Understanding Environment. The syntax is also very similar to the class construction styles used in both C++ and CLOS, so the action routines of the parser do not require much reconfiguration of the data to form class constructors.

Relation to Other Standards

There are many standards for binary image data file formats, such as NITF, TIFF and even as ASCII such as
Postscript. The Programmers Image Kernel (PIK) standard provides additional representations for image processing operations as well as representation for various image data types. There are also standards for the representation of CAD geometry such as the Initial Graphics Exchange Specification (IGES). Some aspects of IU are also captured by standards associated with the exchange of geographic data such as the DOD Vector Product Format (VPF) Standard and the Standard Interchange Format (SIF) used to represent simulation database entities. More recently, standards are emerging for the representation of product design information under the Product Data Exchange Using STEP (PDES) program of the Department of Commerce. STEP is an international standard for the representation of product geometry and functionality. In addition, some aspects of product definition and test requirements are being addressed by the DOD Computer-aided Acquisition and Logistic Support (CALS) program.

These existing standards do not cover the breadth of mathematical and physical concepts inherent in IU algorithms. For example, in the case of image segmentation, there are many attributes which must be defined, such as edge strength, or edge orientation. In order for different research groups to use each other’s results these attributes must be defined according to a standard naming convention and associated mathematical definition. As another example, IU algorithms depend on many types of grouping operations, some quite unique to IU, such as the Hough transform and are not supported by other exchange formats.

Finally, IU is a rapidly evolving discipline and it is necessary to have an easily extensible standard and the same time maintain the compatibility of existing data. The IUE object-oriented design approach enables this flexibility though inheritance and class definitions which can be provided in the exchange file itself.

In the remaining sections, we provide a summary of the ideas behind the development of the standard and provide the syntax for the current version of the file format.

Core Exchange Data Structures

The initial scope of the data exchange format is based on the core data structures in the IUE. The following summarizes the classes to be supported in the initial release of the standard.

1. Image Data
   (a) 8 and 16 bit intensity data
   (b) 8-bit, 3-channel color data
   (c) multi-channel land sat data
   (d) range with registered intensity data

2. Spatial Object
   (a) basic spatial object
   (b) 2d and 3d point sets
   (c) 2d and 3d implicit and parametric lines
   (d) 2d and 3d implicit and parametric planes
   (e) 2d polygonal topology (e.g. vertex, edge, face)

3. Transforms
   (a) 2d and 3d Euclidean
   (b) 3D quaternion
   (c) 4x4 Projective
   (d) 4x3 Image Transform
   (e) UTM and lat-long earth coordinates

4. Spatial Indices
   (a) 2D grid
   (b) 2D quadtree
   (c) 2D r-Tree
   (d) 2D Hough array

5. Image Features
   (a) edge
   (b) pixel chain
   (c) edge chain
   (d) segmentation line segment
   (e) connected line segments
   (f) image region

6. Sensors
   (a) perspective camera
   (b) stereo pair
   (c) moving linear array camera
   (d) range camera

These structures represent only a portion of the IU design, but have been selected as an initial implementation goal and are likely to provide maximum utility to the RADIUS and UGV projects mentioned in the introduction.

An Example

The following example is taken from the IUE specification document which includes examples for data exchange. The specification is mainly concerned with naming and definition of region attributes.

2d-image-region

Description A 2d-image-region is a connected set of image pixels, registered with a set of images by operations such histogram segmentation, region-growing, model surface projection. Sometimes attributes and operations involving regions are based upon the set of points which comprise the region (i.e., compactness, Euler number), some operations and attributes are based upon the image values at these locations (i.e., average intensity in the image area corresponding to the region). This concept can be generalized for voxel processing in analogy to the class block.

Superclasses
image-feature 2d-unordered-pointset face connected-image-pointset
Pseudo Slots (Attributes)

1-chns: list(image-1-chain)
Multiple interior boundaries as a list of usually
image-pixel-chains. These chains can be 4 connected
or 8-connected. The boundary is usually
composed of several pixel-chains which intersect at
image vertices.

area integer
The size of the region in image pixels a method of
face specialized for discrete pixel regions.

number-of-holes: integer
The number of holes in the region.

minimum-bounding-rectangle:
2d-aligned-rectangle-neighborhood

centroid: point
The position of the centroid of the region.

scatter-matrix-of-pixel-positions:
vector[2](vector[2](float))
Provides covariance of x and y coordinates of pixels
in the region.

compactness: float
Ratio of perimeter to area.

adjacent-regions: list(2d-image-region)
A list of regions which share a boundary with self.
The shared boundary descriptions are contained in the
influences of each region.

intensity-distribution: vector[2](float)
A distribution(assumed gaussian) with two slots,
mean and variance. These are floats with the values
computed using all the points in the region and
the corresponding intensity image. If the nature of
the image is unknown, then this is the mean of its
values. For other, known, image types such as red,
infrared, range, etc. other attributes will be used,
but they have the general same form.

red-distribution: vector[2](float)
Distribution for the red component.

green-distribution: vector[2](float)
Distribution for the green component.

blue-distribution: vector[2](float)
Distribution for the blue component.

xxx-distribution: vector[2](float)
Distribution for the XXX component. Since these
are implemented as pseudoslots, any number of such
distributions can be specified according to application
requirements. The general name for the different
distributions is <band-name>-distribution for the
variety of image bands.

An example of the data exchange format for region A in
the figure.

(make 2d-image-region 'reg-A')
(slot 1-chns (list
 (make image-1-chain 'chn-a-b-A')
 (slot edges (list
 (make 2d-pixel-chain 'pchn-a-b'))
 (slot v0 (make 2d-vertex 'vert-a'))
 (slot p (vector 2 integer 1 11 13)))
 (slot v1 (make 2d-vertex 'vert-b'))
 (slot p (vector 2 integer 13 9)))
 (slot n 24)
 (slot chain-code-sequence
 (vector 23 integer 4 4 4 4 3 2
 4 3 1 3 1 2 1 0 0 0 0 7 7 7 6 7)
 0)
 (slot dir (vector 2 integer 1 1))
 (slot closed-p true)
)
 (make image-1-chain 'ic-c-c-A')
 (slot edges (list
 (make 2d-pixel-chain 'pchn-c-c')
 (slot v0 (make 2d-vertex 'vert-c'))
 (slot p (vector 2 integer 7 7)))
 (slot v1 (use 'vert-c'))
 (slot n 14)
 (slot chain-code-sequence
 (vector 13 integer 6 5 7 6 0 0 0
 1 2 3 4 4)
 0)
 (slot dir (vector 1 integer 1))
 (slot closed-p true)
)
 (slot nghbrhood
 (make 2d-image-pixel-neighborhood 'n-5'
 (slot num-nghbrs 7))
 (slot number-of-holes 1)
 (slot adjacent-regions (list 'reg-B'))
 (slot intensity-distribution
 (vector 2 float 135.2 3.4))
)

The Exchange Format

Basic syntax
At the most basic level, IUE Exchange Format looks
somewhat like a lisp file; the format is designed to be
readable by most lisp readers without much difficulty,
should this be necessary. A prototype C parser generat-
ed by the standard Unix(TM) utilities Lex and Yacc
is available upon request.

File Organization
A file in IUE Exchange Format consists of an IUE Ex-
change Format version identifier, followed by a series of
IUE Class instance descriptions, default slot value settings, and references. The general model is that the content an individual IUE file will correspond to one hierarchy, e.g. one site or one building. IUE files may reference external objects by identifying their IUE files; consistent with the notion that the file/hierarchy relationship is one-to-one. Since an IUE file will normally contain a flattened series of IUE Class Instance descriptions and references to those descriptions, the convention has been adapted that the final IUE Class Instance description or reference in a file will be considered the root of the hierarchy. The standard algorithm for traversing a network of IUE Class Instances will invariably attempt to make the root object the last one referenced in the file.

File Identification

An IUE File Identifier is constrained to appear at the very beginning of the file, with no spaces or newline characters embedded. This is so that a file sniffer may depend on the first 20 characters of the file being "(IUE-Exchange-Format-Version )". Case must be strictly adhered to in this particular instance, again, so that file sniffers may be as simple and fast as possible. It will probably be adequate for file sniffers to examine the first four characters "(IUE)" in most instances.

Comments

Comments are introduced by a semi-colon (;) character, as in Common Lisp, and run to the end of line character, again as in Common Lisp.

Sequence numbers and Class instance names

When an IUE file writer describes an IUE Class instance (using a 'make' clause), it is assigned a unique positive integer. Most writers will probably start with one and increment the number for each Class instance described, but as long as the integers are unique, and all integers referenced are defined somewhere in the file, there is no other requirement. For files generated by humans, Class instances may have names instead of numbers, in which case they are not assigned integers from the sequence. These names must be unique within the context of the file, and have no meaning outside of that context; an IUE file reader may discard them once a file has been read. In a human-generated IUE file, both class instance numbers and names may be used.

Using an object

The (use ...) clause is the standard method for referencing an IUE class instance. A use clause may refer to an integer sequence number (described in the previous paragraph), an IUE Class instance name, or an external object (using an external clause.) The object need not have been defined at the time that a use clause refers to that object; in such cases an IUE file reader will place the reference on the list of presently unresolved references, which are to be cleaned up by the time that the file has been completely processed. A standard algorithm for forward reference handling is provided in an Appendix.

Examples of (use ...) clauses:

(use 12)
(use "foo-bar edge")
External objects

An (external ...) clause may be used inside a (use ...) clause to include files. Any file included must be in the same directory as the file containing the (external ...) clause. The last Class instance listed in the external IUE file by either a (make ...) clause or a (use ...) clause will be the one referenced by the (use (external ...)) clause.

Example of an (external ...) clause:

(use 23 (external "my-cube.iue"))

In this example, the file my-cube.iue is processed and the last IUE Class instance made or used in the file is returned, and assigned sequence number 23. This sequence number may be used elsewhere in the file - but it must not appear in any (make ...) clause in the same file.

It is expected that IUE file readers will keep a list of files and the last objects referenced by them; this way, when an external reference is made, a check can be made to see if the file has already been read; otherwise multiple copies of objects might be created.

Making an object

A (make ...) clause consists of the reserved word 'make', followed by an identifier corresponding to a Class name from the IUE class hierarchy, followed by either a sequence number or a string, and finally the slots and attributes for this IUE object and their contents. Attributes are items not provided for in the IUE standard that an IUE user may wish to attach to IUE objects. Examples of (make ...) clauses and slots appear in the next section.

Slots

Slots may contain a variety of data types; these include simple types like bit, int, and float, more complex types such as string and vector, and may contain objects or lists of objects.

A slot clause consists of the word 'slot', followed by the name of the slot and by the content of the slot. If a slot refers to an IUE Class instance, then a use clause will be emitted referencing the object. If the object does not exist, then in a human-generated IUE file, a make clause may be inserted describing the object. A human writing an IUE file may recursively descend through the structure, writing make clauses as IUE Class instances are referenced for the first time. A program generating an IUE file is required to generate a flattened file description in which make clauses are never nested; this is done because extremely large, deeply nested files may otherwise impose unreasonable memory management demands on IUE file readers.

Example of a simple Make clause for a 3D Vertex located at [45.3, 23.8, 3.4] (the vector data type will be detailed later in this document):

(make 3D-vertex 52
(slot p
 (vector 3 float 45.3 23.8 3.4)
))

Attributes

Attributes are very similar to slots, but being user defined items, they are not described in the IUE document. It is the responsibility of the IUE user to insure that any attributes can be properly written and read.

Attributes also differ from slots in that since they are not described in the IUE document, their data type is not known until they are encountered during input; for this reason, the attribute type is included in an Attribute clause.

Example of a nested Make clause for a 3D edge with 2 previously undescribed vertices:

(make edge "my edge"
(slot v1
(make 3D-vertex 35
(slot p (vector 3 float 3.45 -2.34 7.3298)))
(slot v2
(make 3D-vertex 36
(slot p (vector 3 float 5.732 3.21 -2.3)))
(attribute edge-name
    string "NE Edge of object Foo")
)

Example of the same description, flattened as required for machine input, and using the hard slots inferiors and superiors, as is more appropriate with machine generated IUE files:

(make 3D-vertex 2
(slot p
 (vector 3 float 3.45 -2.34 7.3298))
(slot superiors (list (use 1)))
)

(make 3D-vertex 3
(slot p (vector 3 float 5.732 3.21 -2.3))
(slot superiors (list (use 1)))
)

(make edge 1
(slot inferiors
    (list (use 2) (use 3))
(attribute edge-name
    string "NE Edge of object Foo")
)

It is recommended that lower level objects be created immediately before higher level objects, in order to keep the list of unresolved references reasonably small.

Example of a 2D 1-chain\(^2\) which contains 3 edges:

(make 2D-vertex 3
(slot p (vector 2 float 1.0 2.3))
(slot superiors (list (use 2)))
)

\(^2\)A 1-chain is a sequence of connected line segments.
(make 2D-vertex 4
  (slot p (vector 2 float 2.0 2.3))
  (slot superiors (list (use 2) (use 5)))
)(make 2D-vertex 6
  (slot p (vector 2 float 7.3 8.2))
  (slot superiors (list (use 7) (use 5)))
)(make 2D-vertex 8
  (slot p (vector 2 float -2.0 9.3))
  (slot superiors (list (use 7)))
)

(make edge 2
  (slot inferiors (list (use 3) (use 4)))
  (slot superiors (list (use 1)))
)(make edge 5
  (slot inferiors (list (use 4) (use 6)))
  (slot superiors (list (use 1)))
)(make edge 7
  (slot inferiors (list (use 6) (use 8)))
  (slot superiors (list (use 1)))
)

(make 1-chain 1
  (slot inferiors (list
    (use edge 2) (use edge 5)
    (use edge 7)))
)

Vector types

As may be inferred from the previous examples, the vector clause is used to describe homogeneous sequences of class instances; an array representation is presumed. A vector clause begins with the word vector. The second item in a vector clause is the number of elements in the vector; the third is the data type of the vector. Vector elements are constrained to be of the same type as specified by the vector type, or in the case of bit types, the elements must be integers.

IUE Exchange Format provides only 1D vectors and vectors of vectors; matrices are represented by vectors of similar vectors (which are generally similar in both length and data type.) It is possible to represent matrices of arbitrary size and dimensionality using this mechanism without extension to the grammar for the IUE Exchange Format.

Example of a vector representation of an 1024x1024x8bit array:

(vector 1024 vector
  (vector 1024 bit8
    10 11 10 14 14 15 ... )
  (vector 1024 bit8
    11 11 10 14 15 16 ... )
  ... )

Note that decimal integers are being used to represent bit values; a special syntax for bit values is not particularly necessary.

Lists

A list clause consists of the word list, followed by a sequence of simple types, vectors, and IUE Class instances. Lists of simple types will always be homogeneous. Lists of objects are always constrained so that all objects share a common superclass. These restrictions are intended to ease C++ implementation.

Characters and Strings

A simple data type for single characters has been intentionally omitted; this is because the Lisp and the C/C++ worlds have decidedly different notions of what is appropriate syntax. A character string containing exactly one character is more than adequate for representation of a single character.

String is intentionally limited to printable ascii characters; by implication, strings may not presently contain end-of-line characters or tabs. Strings may not contain double quote characters. Strings are written between double quote (" ) characters.

Default Slot Values

Default slot values may be specified for classes and subclasses using the default-slot-value clause. These defaults will be used whenever the class is instantiated and a slot value is not explicitly provided; the defaults may be changed with a new default slot value form at the top level in an IUE file. To set a default neighborhood 3d-ordered-point-sets, one would use a form such as the following:

(make 3d-linesegment-neighborhood 23
  (slot span 3.3))
(default-slot-value
  3d-ordered-pointset
  3d-neighborhood (use 23))

Floats

The syntax for floats is a subset of those of Common Lisp, C, and C++, thus permitting it to be parsed by the standard tools of any of those languages. It is expected that IUE floats will always be double precision floats.

Reserved Words

The number of reserved words has been kept to a minimum and the grammar designed so that changes to the IUE hierarchy will not necessarily force changes to the grammar for the exchange format; in particular, IUE class names and slot names are not reserved words.

User-defined IUE classes

A restricted form of class description is provided so that IUE users may describe their extensions to the IUE hierarchy. Such descriptions will be limited to class inheritance and slot definitions; no provision will be made for transmitting code fragments. An example follows:

3A 3d-linesegment-neighborhood is a linesegment joining a pointset in a one-dimensional sequence. Points are not considered connected if the span distance is exceeded.
A Lisp system can, of course, create such classes on the fly during IUE file input. A C++ system will have to take extra steps; a preprocessor will have to locate the class descriptions in the IUE file and emit a C++ header file fragment. The person(s) managing the IUE system at the destination site will be responsible for integrating this C++ code fragment with their system.

New IUE Classes must be defined before they are referenced.

Appendix 1

Output Algorithm

Using this algorithm to write an IUE class instance will produce a properly ordered flat file with the class instance for the IUE Class Instance selected appearing last in the file, which is proper organization for (external ...) clauses. Infinite loops due to circular references are avoided as objects which have sequence numbers assigned already have either been written or are on the stack waiting to be written, and thus do not need to be revisited. The algorithm is depth-first in character.

Data Structure required:

sequence number hash table – key is IUE-Class-Instance, datum is sequence number
method Output-Object( IUE-Class-Instance, output-stream)
1: [check for previous visitation] if object already has sequence number stored in hash table then return
2: [assign sequence number] obtain sequence number and place in sequence number hash table
3: [for all slots] if slot contains object, list of objects, or vector of objects then for each sub-object recursively invoke Output-Object on sub-object
4: [for desired attributes] if attribute contains object, list of objects, or vector of objects then for each sub-object recursively invoke Output-Object on sub-object
5: [write instance header] “(make-class-name sequence-number ...”
6: [for all slots write] “(slot-name slot-value)”
7: [for desired attributes write] “(attribute attribute-name attribute-type att-value)”
8: [write instance close] “)”

Input Algorithm

This reader will handle both flat and deep representations of data structures, correctly restoring circular references. If names are to be handled as well as sequence numbers, then a C++ implementation will need to double up hash tables (this is not necessary in a Common Lisp/CLOS implementation.) Implementation will be somewhat different in a Lex/Yacc driven implementation, but details of the restoration of the circular references will be identical.

This algorithm presumes that an implementation can support ‘empty shell’ class instances, whose slots have not yet been filled in. If an implementation cannot support such IUE Class Instances, but can create an inferior object without knowledge of its superiors (e.g., create edges given vertices but not yet knowing 1-chains), then it will be necessary to provide an intermediate ‘storage class’ in which to stash Class information such as slot contents, until the object description has been read in allowing the IUE Class Instance to be created. Such a two stage process may necessitate doubled hash tables for storage of intermediate information, and cause some processing steps to be slightly delayed. Such a two stage process is used in the Yacc/Lex prototype reader for Geometer Jr.

There are many implementation details such as handling slots which contain lists of references (both resolved and unresolved) that are not handled; some creativity may be required of the implementor, although there are no insurmountable problems (just a few irritating ones.)

Data structures required:

sequence number hash table – key is sequence number, datum is empty-shell IUE Class Instance
unresolved reference hash table – key is sequence number of as yet undefined instance; datum is a list of references in the form (sequence number, slot-name)
call Function Read-Object for each (make ...) in the input stream:

Function Read-Object( input-stream) returns IUE-Class-Instance

R1: [make instance] create appropriate shell of a IUE Class Instance based on IUE class type from (make ...) clause
R2: [record sequence number] put Class Instance shell and sequence number in sequence number hash table
R3: [for all slots and attributes] if make clause encountered then recursively invoke Read-Object on it, and set slot value to return value else if a sequence number is in the sequence number hash table then set slot value else put current sequence number, slot name, and sequence number of undefined object on the appropriate list in the unresolved reference table
R4: [check for references that can now be resolved] if sequence number for newly created IUE Class Instance appears in unresolved reference hash table, then fill the slots in the appropriate class instances and remove the associated triple from the table.
R5: [return] newly created IUE Class Instance as result
Appendix 2

IUE Exchange Format grammar:

\[ \langle \text{IUE-Exchange-Format-File} \rangle ::= \langle \text{IUE-File-Identifier} \rangle \{ \langle \text{top-object} \rangle \} \]

\[ \langle \text{digit} \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \]

\[ \langle \text{letter} \rangle ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z \]

\[ \langle \text{standard-type} \rangle ::= \text{vector} | \text{slot} | \text{float} | \text{integer} | \text{bit32} | \text{bit24} | \text{bit16} | \text{bit8} | \text{bit} | \text{type} \]

\[ \langle \text{reserved-words} \rangle ::= \text{make} | \text{use} | \text{list} | \text{slot} | \text{t} | \text{nil} | \langle \text{standard-type} \rangle | \langle \text{IUE-Exchange-Format-Version} \rangle | \langle \text{default-slot-value} \rangle | \langle \text{slots} \rangle | \langle \text{inherits} \rangle | \langle \text{external} \rangle \]

\[ \langle \text{digit-sequence} \rangle ::= \langle \text{digit} \rangle \{ \langle \text{digit} \rangle \} \]

\[ \langle \text{sign} \rangle ::= + | - \]

\[ \langle \text{float-exponent} \rangle ::= e \langle \text{digit-sequence} \rangle | E \langle \text{digit-sequence} \rangle \]

\[ \langle \text{dotted-digits} \rangle ::= \langle \text{digit-sequence} \rangle . \langle \text{digit-sequence} \rangle | \langle \text{digit-sequence} \rangle . | . \langle \text{digit-sequence} \rangle \]

\[ \langle \text{unsigned-float} \rangle ::= \langle \text{dotted-digits} \rangle | \langle \text{digit-sequence} \rangle < \text{float-exponent} > | \langle \text{dotted-digits} \rangle < \text{float-exponent} > \]

\[ \langle \text{float} \rangle ::= \langle \text{sign} \rangle \langle \text{unsigned-float} \rangle | \langle \text{unsigned-float} \rangle \]

\[ \langle \text{string} \rangle ::= \langle \text{double-quote} \rangle \langle \text{printable-ascii-characters} \rangle \langle \text{double-quote} \rangle \]

\[ \langle \text{integer} \rangle ::= \langle \text{digit-sequence} \rangle | \langle \text{sign} \rangle \langle \text{digit-sequence} \rangle \]

\[ \langle \text{label} \rangle ::= \langle \text{integer} \rangle | \langle \text{string} \rangle \]

\[ \langle \text{identifier} \rangle ::= \langle \text{letter} \rangle \{ \langle \text{identifier-char} \rangle \} | \langle \text{digit} \rangle \{ \langle \text{hyphen-or-digit} \rangle \} \langle \text{letter} \rangle \{ \langle \text{identifier-char} \rangle \} \]

\[ \langle \text{identifier-or-type} \rangle ::= \langle \text{identifier} \rangle | \langle \text{standard-type} \rangle \]

\[ \langle \text{identifier-char} \rangle ::= \langle \text{letter} \rangle | \langle \text{digit} \rangle | - \]

\[ \langle \text{hyphen-or-digit} \rangle ::= - | \langle \text{digit} \rangle \]

\[ \langle \text{IUE-File-Identifier} \rangle ::= ( \langle \text{IUE-Exchange-Format-Version} \rangle < \text{digit-sequence} > | < \text{digit-sequence} > ) \]

\[ \langle \text{make-or-use} \rangle ::= \langle \text{make} \rangle | \langle \text{use} \rangle | \langle \text{nil} \rangle \]
<obj-list> ::= { <make-or-use> }
<top-object> ::= <make-or-use> | <default-slot-value> | <class-definition>
<list> ::= ( list <list-tail> )
<list-tail> ::= <obj-list> | <int-list> | <float-list>
 | <vector-list> | <list-of-lists>
<int-list> ::= <integer> { <integer> }
<float-list> ::= <float> { <float> }
<string-list> ::= <string> { <string> }
<vector-list> ::= <vector> { <vector> }
<list-of-lists> ::= <list> { <list> }
<vector> ::= ( vector <element-count> <vector-tail> )
<vector-tail> ::= integer <int-list>
 | float <float-list>
 | string <string-list>
 | vector <vector-list>
 | list <list-of-lists>
 | identifier <obj-list>
<element-count> ::= <int>
<class-definition> ::= ( class <class-name> <class-inheritance> <class-slots> )
<class-name> ::= <identifier>
<class-inheritance> ::= ( inherits { <class-name> } )
<class-slots> ::= ( slots { <slot-name> <identifier-or-type> } )
<make> ::= ( make <identifier> <label> <slots-and-attributes> )
<use> ::= ( use <label> )
 | ( use <external> )
<external> ::= ( external <string> )
<default-slot-value> ::= ( default-slot-value <identifier> <identifier>
 | <slot-value> )
<slots-and-attributes> ::= { <slot-or-attribute> }
<slot-or-attribute> ::= <slot> | <attribute>
<slot-descriptor> ::= ( slot <identifier> <slot-value> )
<slot-value> ::= <string> | <integer> | <float> | t | <list>
 | <vector> | <make-or-use> | <identifier-or-type>
<attribute-descriptor> ::= ( attribute <identifier> <identifier-or-type>
 | <slot-value> )