A GRAPHICS INTERCHANGE PROTOCOL AND FORMAT, INCLUDING SYSTEM TOOLS

Kevin Weiler[1]
General Electric Company
Corporate Research and Development
Schenectady, New York

Greg Glass
Institute of Building Sciences
Carnegie-Mellon University
Pittsburgh, Pennsylvania

ABSTRACT

Most research groups over a period of time develop and acquire enough different programs and devices that they are in a position to appreciate the advantages of some common means of communicating information between them. In a small dynamic research environment, however, fixing everyone to a single inflexible standard is usually unacceptable because changes in programs and devices occur at a rapid rate, quickly outdating the standard.

We've attempted to resolve these two conflicting concerns by staging a flexible interchange format system into two distinct levels. The bottom level, the *protocol level*, is essentially the underlying language in which the format is written. The top level, the *format level*, is where the interchange format is actually expressed. The strength of this division is twofold: uniformity of syntax, allowing high-level parsing tools to be built; and flexibility of semantics, allowing a format which can evolve along with the user community in the least painful way.

Detailed descriptions of the protocol, a typical format, and the system tools expected to be provided with an implementation of the protocol are presented.

1. INTRODUCTION

Most research groups over a period of time develop and acquire enough different programs and devices that they begin to appreciate the advantages of some common means of communicating information between them. Some of the benefits include the reduced time and effort required to create the interfaces, since it is necessary to interface each program or device to the standard just once rather than having to interface it to every other existing and newly developed program and device. An additional advantage is the increase in overall synergy as new applications begin to piggyback on older applications and device drivers. For example, filter programs, such as graphics transformation and display, become possible and more generally useful.

In a small dynamic research environment, however, fixing everyone to one standard can also be anathema: new research and applications require new structures the old standard can't accommodate, new devices have new features and requirements that don't fit into the standard, or the standard itself may have been developed before there was a clear understanding of what it was required to do, and it is simply inadequate. Often, as a matter of sheer practicality, standards are designed as a least common denominator of what might actually be needed. Standards are perceived, often with justification, as simply too rigid to respond to changing needs and devices.

We've attempted to resolve these two conflicting concerns by staging an interchange format system into two distinct levels. The

[1]Research on this topic was initiated while at the Institute of Building Sciences, Carnegie-Mellon University, and completed while at Three Space.

result is an interchange format system that is flexible enough to grow with the user community, aided by the provision of high-level tools with which application programmers can build interfaces easily.

The bottom level of this two-level system, the *protocol,* is essentially the underlying language in which the format is written. It is fixed and is not intended to change. More interestingly, although it enforces a strong but general syntax, it has almost no semantics of its own.

The top level, the *format,* is where the standard is actually expressed. While its syntax has been predefined by the underlying protocol (and therefore all format syntax is a subset of the protocol syntax), virtually all semantics of the standard are defined at the format level.

The strength of this division is twofold—uniformity and flexibility—uniformity of syntax, and flexibility of semantics. A strong uniform syntax at the bottom level allows us to build high-level tools for users of whatever format is developed on top of the protocol. Flexibility in the format is obtained because no restrictions on the semantic meaning of the format are enforced from the underlying level. Two conflicting demands have been resolved by separation of the system into two parts, allowing each demand to be resolved in separate parts with little interaction with the other.

Using these principles, we have designed and implemented a system along with a series of tools that makes it easy to write new interfaces to existing local standards using the same protocol, as well as to update existing interfaces to both new and changing local standards.

2. STANDARDS

Don't let the foregoing fool you; national and international interchange standards are quite necessary and useful for the exchange of information between remote sites that otherwise have little contact with each other and have little other basis for agreement, as well as for communication between programs and devices that are used often but rarely changed.

Standards, such as the SIGGRAPH CORE Metafile proposed standard for graphic information [1] and the IGES ANSI standard for geometric modeling and drawing information [2], by political necessity must carve out a relatively narrow area of interest and/or deliver a fairly rigid set of formatting rules. The necessity of such fixed standards, and the difficulty of arriving at them in the first place, cannot be overestimated.

For the dynamic, local environment of interest here, however, such standards limit both capability and flexibility. For these environments, where short-term and experimental interfaces are a daily need, a flexible interchange format system that can evolve based on local concerns is required. This paper is intended to address this special need.

3. DESIGN CRITERIA

In addition to flexibility and the provision of high-level interfacing tools, several additional design criteria for an interchange format system were felt important enough to influence the design of the system.

We wanted the formats developed to be human-readable, not simply long lists of numbers. People should be able to use the same format for testing their programs as they might use for the actual interface to other programs. Thus it was stipulated the formats must use only printable characters and allow character names to appear. Text editors could then be used for manual entry and modification of data formatted in the protocol. It would also ease transfer of information over networks and other media capable of handling textual data.

We were concerned mainly about the conciseness of the formats, and not so much about the speed with which they could be processed. The technique is intended as a visible experimental interface and not a high-speed interface between heavily used and relatively unchanging programs.

The syntax needed to be flexible enough to allow hierarchical structuring of data, though we didn't feel the concept of unnamed pointers was human-readable enough to be of major concern in the design.

And, of course, we wanted it all to be relatively simple, and not blossom into another full-scale language development effort.

4. THE PROTOCOL

The protocol itself consists primarily of only one syntactic structure which is general enough to allow the assignment of a wide variety of semantic meaning at the format level. This structure, called a "context," is simply a name optionally followed by a bracketed context called a "context body" which may contain an arbitrary list of other contexts (see Figure 1).

Syntax

context ::= name ["<" { context } ">"] .

Examples

object
object < atribute1 attribute2 >

Figure 1. The protocol context syntax.

There are also two minor semantic features associated with the protocol itself. The first is the ability to recognize (and ignore) comments. The second is a simple named macro definition, deletion, and recall capability, which can also be used for file inclusion.

The complete syntactic and semantic definition of the protocol is given in Figure 2 and Appendix I.

Since the syntax of the protocol is rigid, it becomes feasible to provide a set of high-level accessing tools, which make the task of developing interpreters for the format simpler. In fact, that task becomes primarily a matter related to the semantics of what is going on rather than getting bogged down in syntax, since the syntax is already taken care of by the tools provided with the protocol. Only three primitive procedures are necessary to access formats written in the protocol: one for initialization and two for accessing. Users can build many other tools on top of these primitives, depending on the regularity of the formats that are used. These system tools are described in a later section.

5. FORMAT

A variety of semantics can be defined at the format level because of the generality of the underlying syntax. Data objects can be represented as single names, as names with attributes, and even as complex hierarchically structured objects. The macro capability built into the protocol also allows data abstraction. The format can also define procedural structures, including conditionals, compound statements, and even a limited emulation of procedure calls with parameters.

Strings

```
delimiter        ::=  <space> | <newline> | <pagemark> |
                      <tab> .
punctuation      ::=  "<" | ">" | "{" | "}" | "“" .
stringmodifier   ::=  "#" | "!" | "@" .
otherchar        ::=  <any printable char except
                      delimiters or punctuation> .
nonmodifierchar::=    <any otherchar except stringmodifier> .
quotedchar       ::=  "“" <any printable char> .
string           ::=  ( nonmodifierchar | quotedchar )
                      { otherchar | quotedchar } .
```

Comments

```
comment          ::=  "{"
                      { <any char except unquoted matching "}"> }
                      "}" .
```

Data Group/File Manipulation

```
filename         ::=  string .
groupname        ::=  string .
creation         ::=  "#" groupname creationbody .
creationbody     ::=  "<"
                      { <any char except unquoted matching ">"> }
                      ">" .
deletion         ::=  "!" groupname .
inclusion        ::=  "@" ( groupname | filename ) .
```

Contexts

```
contextname      ::=  string.
context          ::=  contextname [ contextbody ] .
contextbody      ::=  "<" { context } ">" .
```

The Protocol

```
protocol         ::=  { context } .
```

Figure 2. The Protocol Syntax described in a modified BNF. It is informal in that some nonterminals (enclosed in unquoted angle brackets) are described qualitatively for conciseness or clarity.

5.1 Format Design

Designing a format to serve as a communication standard is as much of an art as any other form of language design. Creating the design involves balancing many factors, including readability, conciseness, generality, modularity, probability of accidental syntax errors, ease of error checking and recovery, and others.

Care in the design of the format can have big payoffs. It can ensure compatibility with future formats and minimize changes required. For example, rather than making coordinate locations an integral part of the format for each data type we were interested in (such as polygons, lines, points, etc.), we decided that coordinate lists, though used by many other datatypes, should have a recognizable format of their own regardless of where they appear. A graphics matrix transform system thus would not need to know that it was busy transforming a point, a line, or a polygon. And it could automatically transform a bezier curve when that syntax became defined even though it had never seen one before and had not been designed with one in mind. Filter programs such as the transform system normally pass through to the output all information they read in, modifying only those things they recognize, which in this case are only the coordinate values.

Along these same lines, if the format design maintains a close correspondence between the protocol context syntax and the semantic units of the format, it is possible to provide general purpose applications (such as the transformation system mentioned above) that can perform its task regardless of the unknown data types and attributes it may encounter. This allows new elements to be designed after the implementation of the general purpose applications, without necessarily requiring those applications to be updated. Error checking and recovery are also easier when there is a close correspondence of the format semantic units and the protocol context syntax because of the ability to skip over portions of the input not understood by the application.

5.2 Format Semantics

The basic protocol syntax available to the format designer, while simple, has a great deal of flexibility in terms of the wide range of semantics that can easily be associated with it. Both data and procedural semantics can be assigned to the variations on the protocol syntax by the choice of syntax and semantics specified by the format.

5.2.1 Data Semantics

All constructs, including data constructs, are represented as character strings in the protocol. The simplest format possible would be a single string, such as a name or numeric value (see Figure 3a). Syntactically, this is just a context consisting of a context name without the context body. If the context body is included, the named data entity could have zero or more occurrences of other contexts within its context body. This could be used to associate attributes with the named data (Figure 3b). By expanding these "attributes" into full contexts with their own context bodies, a hierarchical data structure can be represented (Figure 3c). Data abstraction is also possible with the capabilities of the protocol syntax. By defining a data object context or list of contexts as a macro with the data group creation capability, an arbitrarily complex object can be repeatedly referenced by a single name (Figure 3d).

(a) *Single Name*

```
object
10.47E91
```

(b) *Name with Attributes*

```
object < big red heavy >
```

(c) *Hierarchically Structured Objects*

```
whole object <
  partone <
    subpart
    subpart
    >
  parttwo <
    subpart <
      part
      part
      >
    subpart < part >
    >
  >
```

(d) *Data Abstraction*

Definition:

```
#abstraction < many things large and small >
```

Use:

```
@abstraction
```

Figure 3. Example of possible data semantics formats.

5.2.2 Procedural Semantics

Procedural semantics can also be assigned to the protocol syntax by a format definition. Combinations of data and procedural semantics in a format could be used to create a syntax similar to a limited computer language.

One of the more obvious uses of the context syntax in a procedural fashion is to use the context as a grouping mechanism for

compound statements (Figure 4a). Conditionals can also be provided in a variety of ways (Figure 4b shows one). Contexts can also be used to create special procedural context scopes in which both data and other procedural constructs might take on special meanings (Figure 4c). Though somewhat clumsy, limited procedures with parameters can also be represented by use of the macro capabilities of the protocol (Figure 4d).

```
(a) Compound Statement
        Group   <Stmt1
                 Stmt2
                 Stmt3
                >

(b) Conditionals
        if < condition < a or b >
             then      < Stmt1 >
             else      < Stmt2 >
           >

(c) Special Contexts Affecting Contents
        display < line1
                  line2
                  poly1
                  poly2
                >

(d) Limited Parametric Procedure Definition
        Definition
          {procedure to add two parameters}
            #adder        <
              { param list: #param1 <first addend>
                            #param2 <second addend> }

            write < expr < @param1 + @param2 > >
            >

        Use
          #param1         <2>
          #param2         <17>
          @adder
            {limitation: nested procedures calls may
            not use parameters of the same names}
```

Figure 4. Examples of possible procedural semantics.

5.3 A Format Design

For our own local environments we decided to start simple and designed a format to encode some of the kinds of graphic data we wanted to ship between various programs and devices. We knew that the format would be tested, and probably modified and added to on the basis of what was found more desirable in our local environments. We wound up, therefore, designing a family of formats rather than just a single format.

A simplified description of the format we started with for two-dimensional graphic data is contained in Figure 5. It uses the special coordinate syntax described in Section 5.1 so that it is easier to build system filters as described in that section. The description of the syntax is followed by a short sample of graphic data written in the format.

Figure 6 describes the syntax of a format that controls the operation of a graphic transformation system filter. The filter, unless otherwise instructed, will copy data from input to output. If the tranformation command described is encountered, the process will continue with the exceptions that any coordinate information found in the transform context body will be transformed, and the transform command itself will not appear in the output. The only thing an implementation of this system filter has to be able to do is recognize its control syntax, recognize and transform coordinate information, and copy from its input to its output. It is not neces-

Syntax

```
numeral      ::= "0"|"1"|"2"|"3"|"4"|
                 "5"|"6"|"7"|"8"|"9" .
coord        ::= [ "+"|"-" ]
                 ( ( numeral { numeral } [ "." { numeral } ] )|
                 ( "." numeral { numeral } ) )
                 [ ( "E"|"e" ) [ "+"|"-" ] numeral { numeral } ] .

2dcoord      ::= "2d" "<" coord coord ">" .
2dcoords     ::= "2d" "<" { coord coord } ">" .
2dlinecoords ::= "2d" "<" { coord coord coord coord } ">" .

attributes   ::= { <user defined context for optional special
                 properties such as color, texture, etc.> } .

2dpoints     ::= "points" "<" [ attributes ]
                 2dcoords [ attributes ] ">" .
2dlines      ::= "line" "<" [ attributes ]
                 2dlinecoords [ attributes ] ">" .
2dlineseq    ::= "lineseq" "<" [ attributes ]
                 2dcoords [ attributes ] ">" .
2dpolygon    ::= "poly" "<" [ attributes ]
                 2dcoords [ attributes ]
                 { "hole" "<" [ attributes ] 2dcoords [ attributes ] ">" }
                 [ attributes ] ">" .
2dtext       ::= "text" "<" [ attributes ] 2dcoord [ attributes ]
                 "string" "<" contextbody ">" [ attributes ] ">" .
```

Sample Data in the Format

```
                                        {a polygon}
poly < red
       2d    <   20   20
                 20  -20
                -20  -20
                -20   20
              >
     >
                                        {label it}
text < italics
       2d    <   25   25 >
       string < this is a square here >
     >
                                        {underline label}
line < 2d < 23 23 128 23 > >
```

Figure 5. The format syntax definition in modified BNF designed as a minimal format for the interchange of two-dimensional graphic data.

Syntax

```
transform ::= "transform" "<" [ "trans" "<" coord coord coord ">" ]
                              [ "rotate" "<" coord coord coord ">" ]
                              [ "scale" "<" coord coord coord ">" ]
                              <context body which contains data
                               including coords to be transformed>
                          ">"
```

Sample Data

```
#square  <poly<red 2d<1 1   1 -1   -1 -1   -1 1>>>

transform <   trans <10 20 30>
              rotate <45 90 15>
              scale <100 200 33.33>

              @square

              transform   <   trans <12.0 24.45 36.77>
                              scale <  100 100 100 >

                              @square
                          >
          >
```

Figure 6. The format syntax definition described in a modified BNF.

sary for it to be able to recognize anything else, such as the data types used (polygons, lines, points, etc.). A short example of data in the format follows the syntax.

6. SYSTEM TOOLS

Since the general syntax class of all formats is specified by the protocol syntax, a good part of the effort of developing interpreters for a specific format can be provided by system tools at the protocol level. These same tools can be used in interpreters for all formats based on the protocol, and can reduce the implementation effort required for interfaces to only those areas specifically related to the semantics intended by a specific format. The basic set of primitive system tools specified here for developing format interpreters operate somewhere between the level of a typical compiler scanner and a full parser. They not only isolate individual symbols in the input, but also parse the general protocol syntax and identify the scope of context bodies. These primitive tools can also be used as the basis of even higher level tools which may be restricted to parsing syntax limited to the syntax of a specific format.

6.1 Primitives

Only three primitive procedures are necessary to access data written in formats using the protocol; one is used for initialization and two are used for accessing the data. These routines are described below along with a description of their parameter lists in a Pascal syntax.

procedure source (name: filespec);

This procedure performs all initialization of the protocol tools and specifies the initial source file or device from which the protocol data stream is obtained. It is called only once at the start of interpretation.

function context (s: string): boolean;

This function returns the next context name remaining on the current context level. The context level is the nesting depth of context bodies. If there is an unread context name at this level, *context* will be true and will return the string containing the new name, then advance its position in the incoming data stream. *Context* will only return names on the current context level; it will not increase the level automatically for a context that has a context body. It will therefore skip over a context body if necessary to obtain the next context name on the current level. *Context* will return false the first time there are no remaining context names on the current level (it has reached the end of a context body or has hit the end of the input stream specified at initialization). If *context* returns false it will also *decrease* the context level automatically so that the *next* call to *context* will return the next context name (if any) at the level above the level of the exhausted context body. It therefore moves over the end of the context body delimiter, decreasing the context nesting level. At the end of the original input source, *context* will continuously return false.

function down: boolean;

Down can be called after any successful call to *context*. It will return true if there is a context body associated with the most recent context name found by *context*. It will return false if there is no context body associated with the most recent context name. If true, *down* will also *increase* the context level nesting; that is, it will move over the beginning of context body delimiter so that the next call to *context* will return the first item in the context body associated with the context name of the previous call to *context*. The new context level will not be decreased until *context* calls exhaust all of the context names in the context body.

These three primitive routines are sufficient to parse all protocol syntax. *Source* is used to initialize the process and specify the initial data source. *Context* returns all context names at the current level, but goes up a level if it has exhausted all context names on the current level. *Down* determines if there is a context body associated with the current context name, and goes down a level if there is one.

6.2 Additional Tools

Many more high level tools based on the primitives, while not mandatory, can aid the development of a parser for interpreters. Some are general purpose, but many restrict themselves to a specific format syntax and can operate as a high level parsing tool. Increasingly higher level tools are built on top of one another until a tool which parses all productions in the format syntax is produced. The following three routines are a few of the general purpose tools we have found useful in our environment.

function cton(number: string; var r: real): boolean;

The *cton* function tries to convert a given input string into a numeric value if possible. If the string is successfully interpreted as a number *cton* is true and returns the value in *r;* otherwise *cton* is false.

function compare(a: arrayofstrings; s: string): index;

If the string *s* is found in the array of strings *arrayofstrings, compare* will be the value of the array index of the matching string; otherwise it will be the invalid index value 0. This is a convenient tool for parsing command names. Even higher level functions can be built which perform both calls to *context* and *compare* as a single function.

function getcoords(var x,y,z : real): boolean;

This function is a variation on *cton;* it is a higher level tool which collects three numeric values at a time by repeatedly calling *context* and then *cton.* It is to be used in situations where a three-dimensional coordinate is expected. If the three values are successfully found at the current context level and converted to numbers, then *getcoords* is true and the values are returned in *x, y, z;* otherwise *getcoords* is false.

6.3 Example of a Parser

The example in Figure 7 illustrates how a parser for a very simple format can be constructed. The hypothetical format of the example is intended to be used to create line drawings on a display and therefore consists only of five display instructions: three datatypes for polygons, lines, and points; and two commands to clear the screen and to wait for a fixed period of time. The parser shown interprets data in this format and displays the results. The example is intentionally simplistic and is only intended to illustrate how primitives and additional tools can be used when building a parser (as well as to illustrate how the primitives work). A production-quality parser would include more extensive error checking and recovery to guarantee robustness and give the user more information about the format syntax (or semantic!) errors.

7. CONCLUSION

Research environments have special need for graphic communication standards which are flexible enough to meet changing local needs, and which are backed up by enough high-level tools to

Sample Program

```
                                {name array declaration}
const NONE = 0 ;    POLY = 3 ;
      CLEAR = 1 ;   LINE = 4 ;
      WAIT = 2 ;    POINT = 5 ;
var   commands: array [CLEAR..POINT] of string;

                                {initialization of name array}
procedure init ;
begin
  commands[POLY]  := 'POLY  ' ;
  commands[LINE]  := 'LINE  ' ;
  commands[POINT] := 'POINT ' ;
  commands[CLEAR] := 'CLEAR ' ;
  commands[WAIT]  := 'WAIT  ' ;
end { init } ;

                                {actual parser}
procedure pictureparser (file: filename);

  var s: string;
      r, x,y,z, x1,y1,z1, x2,y2,z2 : real;

begin
                        {initialize}
  source (file);
  init;
                        {loop on input}
  while context(s) do
    case compare(commands,s) of
      NONE    :  writeln('ERROR - data not formatted correctly');
      CLEAR   :  clearscreen;
      WAIT    :  wait;
      POINT   :  if down then while getcoords(x1,y1,z1) do
                                drawpoint(x1, y1, z1) ;
      LINE    :  if down then
                   while (getcoords(x1,y1,z1) and
                         (getcoords(x2,y2,z2)) do
                    drawline (x1,y1,z1,x2,y2,z2);
      POLY    :  if down then
                   if getcoords(x,y,z) then begin
                    x1 := x; y1 := y; z1 := z1;
                    while getcoords (z2,y2,z2) do begin
                       drawline(x1,y1,z1,x2,y2,z2);
                       x1 := x2; y1 := y2; z1 := z2;
                       end;
                    drawline (x1,y1,z1,x,y,z)
                    end;
      end

end {parser};
```

Sample Data

```
CLEAR                           {clear the screen}
POLY                            {draw a square}
      <   100   100
          100  -100
         -100  -100
         -100   100
      >
LINE                            {add "legs" to square}
      <   100  -100   100  -150
         -100  -100  -100  -150
      >
WAIT                            {let user watch it}
CLEAR                           {clear screen for next picture}
```

Figure 7. Example of a parser for a simple format.

minimize the effort required to interface various programs and devices. We've attempted to address this problem by staging a standard format system into two levels, allowing high-level accessing tools to be built because of the uniformity of syntax on the lower level, and allowing flexibility because of the independence of the format semantics defined on the higher level.

ACKNOWLEDGMENTS

Thanks are due to the members of the CAD-Graphics Laboratory of IBS at CMU for many constructive suggestions during the development of this system.

BIBLIOGRAPHY

[1] "Status Report of the Graphic Standards Planning Committee," *Computer Graphics,* Vol 13, No. 3, August 1979.

[2] "Draft Proposed American National Standard Engineering Drawing and Related Documentation Practices Digital Representation for Communication of Product Definition Data," IGES Y14.26M Response Committee, June 1981.

Appendix I

GRAPHIC INTERCHANGE PROTOCOL NOTES

PROTOCOL SYNTAX NOTES

1. Delimiters and punctuation may appear in strings only if they are quoted by the quote character. Thus the quote character can only be used directly by quoting itself, i.e. ''. Unless it is desired to invoke the data group manipulation semantics, a stringmodifier character must be quoted if it appears at the beginning of a string. Note also that punctuation and delimiters may not be quoted or they will be treated as strings. It is recommended that the <newline> and <pagemark> characters not be quoted as the effect may be implementation dependent.
2. From the definition of "string" it is implicit that one or more delimiters may appear anywhere in the syntax for the purpose of separating adjacent string definitions from one another. These delimiters are also allowed but not required where a punctuation character would normally appear and fulfill this function also. Delimiters are NOT allowed between a stringmodifier character and its associated string.
3. "comment"'s, "creation"'s, "deletion"'s, and "inclusion"'s may appear anywhere in the syntax that a delimiter could appear, as long as they are distinguishable from adjacent strings by appropriate separation with additional delimiters as required.
4. There is an implementation-defined limit on the length of strings. This limit must be at least 256 characters (quotes are not counted). The format of strings passed to the user must be fully described for each language/system in which the protocol system is implemented.
5. There is an implementation-defined limit on the maximum length of "creationbody"'s. This limit must be at least 1024 characters.
6. Note that both comments and creation bodies are only ended by MATCHING delimiters, meaning that "{", "}" and "<", ">" punctuation pairs respectively may be nested inside. Comments may be nested inside one another, for example, but every "{" encountered must have its matching "}" before the comment is ended.
7. Only printable characters are allowed in the protocol. This eases manual checking and editing of the protocol. Nonprinting characters (as well as binary values) can be implemented in the format using this protocol if required.

PROTOCOL SEMANTICS NOTES

The protocol defined here is accessed through a standard set of interpretation procedures provided to users as defined elsewhere in this document. Users must always use the set of accessing procedures provided to gain access to data using the protocol.

There are four semantic functions provided by the the protocol: comments, and data group creation, deletion, and inclusion (as well as file inclusion). Their actions are invisible to client programs using the accessing procedures provided to read data written in the protocol. The semantics are described below.

1. The "creation" syntax associates the given name with the given group of data within the "creationbody". This name is valid for the entire body of the "context" in which the "creation" was encountered or the entire interpretation if it was encountered outside of any "contextbody". This data may later be repeatedly retrieved and used with the "inclusion" facility. Implementations of the protocol accessing tools may or may not have these data groups stored in operating system files of the specified name. Any operating system files possibly created by implementations of the "creation" command will be automatically deleted at the end of the "contextbody" in which the "creation" was encountered or at the end of interpretation if the "creation" occured outside of any "contextbody". In this case all local conventions regarding filenames apply; the user must therefore be aware of the effect of creating and deleting files with the specified names. In all implementations the names must be remembered by the protocol implementation since there is a limited scope for each named group. There is therefore an implementation defined limit, no less than sixteen, to the number of maximum "creation" groups currently accessable across all currently valid "contextbody"'s. Note that all characters between the "<" and ">" of the "contextbody" are simply transcribed into the named group storage area and are not interpreted by the system. Thus no "inclusion"'s, "creation"'s, or "deletion"'s are performed inside of the "creationbody" at the time of the creation of the named group; they can only be performed when the data is later requested for "inclusion" by name. Retrieval of groups by name via "inclusion" follows a stack ordered name search related to context level. Thus the definition of a group with a name identical to a group defined on a preceding level temporarily supersedes the previous definition until the context in which the second definition occurred is ended. Repeated definition on the same context level using the same name simply associates the name with the most recent definition.
2. The "deletion" syntax, when encountered, will cause the named group created with a similar "creation" to be immediately forgotten, and if implemented as files, the named file to be deleted. If "creation" syntax is not implemented with files, then no file deletion will be allowed. Note that group names are always automatically forgotten when the "contextbody" in which the "creation" of the group occcured ends. "deletion" performs this function explicitly and immediately.
3. The "inclusion" syntax, when encountered, will cause the named group or operating system file to be included into the character stream being interpreted at that point. "inclusion"'s may be nested up to an implementation-defined limit, which may not be less than four. Named groups of data created by "creation" syntax may or may not correspond to operating system files depending on the implementation. If "creation" does not use operating system files in its implementation, names of groups created by "creation" have precedence over operating system filenames in the case of identical names when using the "inclusion" syntax.
4. The "comment" syntax, when encountered, is entirely ignored and is not accessible to the user through the accessing routines.
5. Any actions implied by the "creation", "deletion", "inclusion", or "comment" syntax are taken by the interpretation system provided for the protocol. The client program using the accessing tools provided with the protocol need not and cannot be directly aware of file accesses or input redirection possibly caused by "inclusion"'s, of the file creation and deletion possibly caused by "creation"'s and "deletion"'s, or of the "comments" passed over by the protocol interpretation system provided.