The Languages of Thot

Vincent Quint

Translated from French by Ethan Munson

Version of September 15, 2000

Contents

The document model of Thot

All of the services which Thot provides to the user are based on the system's internal document representation. This representation is itself derived from the document model which underlies Thot. The model is presented here, prior to the description of the languages which permit the generic specification of documents.

The logical structure of documents

The document model of Thot is primarily designed to allow the user to operate on those entities which s/he has in mind when s/he works on a document. The model makes no assumptions about the nature of these entities. It is essentially these logical entities, such as paragraphs, sections, chapters, notes, titles, and cross–references which give a document its logical structure.

Because of this model, the author can divide the document into chapters, giving each one a title. The content of these chapters can be further divided into sections, subsections, etc. The text is organized into successive paragraphs, according to the content. In the writing phase, the lines, pages, margins, spacing, fonts, and character styles are not very important. In fact, if the system requires documents to be written in these terms, it gets in the way. So, Thot's model is primarily based on the logical aspect of documents. The creation of a model of this type essentially requires the definition :

- of the entities which can appear in the documents,
- and the relations between these entities.

The choice of entities to include in the model can be subtle. Some documents require chapters, while others only need various levels of sections. Certain documents contain appendices, others don't. In different documents the same logical entity may go by different names (e.g. ''Conclusion'' and ''Summary''). Certain entities which are absolutely necessary in some documents, such as clauses in a contract or the address of the recipient in a letter, are useless in most other cases.

The differences between documents result from more than just the entities that appear in them, but also from the relationships between these entities and the ways that they are linked. In certain documents, notes are spread throughout the document, for example at the bottom of the page containing the cross–reference to them, while in other documents they are collected at the end of each chapter or even at the end of the work. As another example, the introduction of some documents can contain many sections, while in other documents, the introduction is restricted to be a short sequence of paragraphs.

All of this makes it unlikely that a single model can describe any document at a relatively high level. It is obviously tempting to make up a list of widely used entities, such as chapters, sections, paragraphs, and titles, and then map all other entities onto the available choices. In this way, an introduction can be supported as a chapter and a contract clause supported as a paragraph or section. However, in trying to widen the range of usage of certain entities, their meaning can be lost and the power of the model reduced. In addition, while this widening partially solves the problem of choosing entities, it does not solve the problem of their organization: when a chapter must be composed of sections, how does one indicate that an introduction has none when it is merely another chapter? One solution is to include introductions in the list of supported entities. But then, how does one distinguish those introductions which are allowed to have sections from those which are not. Perhaps this could be done by defining two types of introduction. Clearly, this approach risks an infinite expansion of the list of widely used entities.

Generic and specific structures

Thus, it is apparently impossible to construct an exhaustive inventory of all those entities which are necessary and sufficient to precisely describe any document. It also seems impossible to specify all possible arrangements of these entities in a document. This is why Thot uses a *meta−model* instead, which permits the description of numerous *models*, each one describing a *class* of documents.

A *class* is a set of documents having very similar structure. Thus, the collection of research reports published by a laboratory constitutes a class; the set of commercial proposals by the sales department of a company constitutes another class; the set of articles published by a journal constitutes a third class. Clearly, it is not possible to enumerate every possible document class. It is also clear that new document classes must be created to satisfy new needs and applications.

To give a more rigorous definition of classes, we must introduce the ideas of *generic structure* and *specific structure*. Each document has a *specific structure* which organizes the various parts which comprise it. We illustrate this with the help of a simple example comparing two reports, A and B (see Figure). The report A contains an introduction followed by three chapters and a conclusion. The first chapter contains two sections, the second, three sections. That is the *specific* structure of document A. Similarly, the structure of document B is: an introduction, two chapters, a conclusion; Chapter 1 has three sections while Chapter 2 has four. The specific structures of these two documents are thus different.

```
        Report A                  Report B
             Introduction               Introduction
```

Chapter 1
    Section 1.1
    Section 1.2
Chapter 2
    Section 2.1
    Section 2.2
    Section 2.3
Chapter 3
Conclusion
	Chapter 1
	    Section 1.1
	    Section 1.2
	    Section 1.3
	Chapter 2
	    Section 2.1
	    Section 2.2
	    Section 2.3
	    Section 2.4
	Conclusion

Two specific structures

The *generic structure* defines the ways in which specific structures can be constructed. It specifies how to generate specific structures. The reports A and B, though different, are constructed in accordance with the same generic structure, which specifies that a report contains an introduction followed by a variable number of chapters and a conclusion, with each chapter containing a variable number of sections.

There is a one–to–one correspondence between a class and a generic structure: all the documents of a class are constructed in accordance with the same generic structure. Hence the definition of the class: a class is a set of documents whose specific structure is constructed in accordance with the same generic structure. A class is characterized by its generic structure.

Thus, a generic structure can be considered to be a model at the level which interests us, but only for one class of documents. When the definition is limited to a single class of documents, it is possible to define a model which does a good job of representing the documents of the class, including the necessary entities and unencumbered by useless entities. The description of the organization of the documents in the class can then be sufficiently precise.

Logical structure and physical structure

Generic structures only describe the *logical* organization of documents, not their *physical* presentation on a screen or on sheets of paper. However, for a document to be displayed or printed, its graphic presentation must be taken into account.

An examination of current printed documents shows that the details of presentation essentially serve to bring out their logical structure. Outside of some particular domains, notably advertising, the presentation is rarely independent of the logical organization of the text. Moreover, the art of typography consists of enhancing the organization of the text being set, without catching the eye of the reader with overly pronounced effects. Thus, italic and boldface type are used to emphasize words or expressions which have greater significance than the rest of the text: keywords, new ideas, citations, book titles, etc. Other effects highlight the organization of the text: vertical space, margin changes, page breaks, centering, eventually combined with the changes in the shapes or weight of the characters. These effects serve to indicate the transitions between paragraphs, sections, or chapters: an object's level in the logical structure of the document is shown by the markedness of the effects.

Since the model permits the description of all of the logical structure of the document, the presentation can be derived from the model without being submerged in the document itself. It suffices to use the logical structure of the document to make the desired changes in its presentation: changes in type size, type style, spacing, margin, centering, etc.

Just as one cannot define a unique generic logical structure for all document classes, one cannot define universal presentation rules which can be applied to all document classes. For certain types of documents the chapter titles will be centered on the page and printed in large, bold type. For other documents, the same chapter titles will be printed in small, italic type and aligned on the left margin.

Therefore, it is necessary to base the presentation specifications for documents on their class. Such a specification can be very fine−grained, because the presentation is expressed in terms of the entities defined in the generic logical structure of the class. Thus, it is possible to specify a different presentation for the chapter titles and the section titles, and similarly to specify titles for the sections according to their level in the section hierarchy. The set of rules which specify the presentation of all the elements defined in a generic logical structure is called a *generic presentation*.

There are several advantages derived from having a presentation linked to the generic structure and described by a generic presentation. Homogeneity is the first. Since every document in a class corresponds to the same generic logical structure, a homogenous presentation for different documents of the same class can be assured by applying the same generic presentation to all documents of the class. Homogeneity of presentation can also be found among the entities of a single document: every section heading will be presented in the same way, the first line of every paragraph of the same type will have the same indentation, etc.

Another advantage of this approach to presentation is that it facilitates changes to the graphical aspect of documents. A change to the generic presentation rules attached to each type of entity will alter the presentation of the entire document, and will do so homogenously. In this case, the internal homogeneity of the class is no longer assured, but the way to control it is simple. It suffices to adopt a single generic presentation for the entire class.

If the presentation of the class does not have to be homogenous, then the appearance of the document can be adapted to the way it will be used or to the device used to render it. This quality is sufficient to allow the existence of many generic presentations for the same document class. By applying one or the other of these presentations to it, the document can be seen under different graphical aspects. It must be emphasized that this type of modification of the presentation is not a change to the document itself (in its specific logical structure or its content), but only in its appearance at the time of editing or printing.

Document structures and object structures

So far, we have only discussed the global structure of documents and have not considered the contents found in that structure. We could limit ourselves to purely textual contents by assuming that a title or a paragraph contains a simple linear text. But this model would be too restrictive. In fact, certain documents contain not only text, but also contain tables, diagrams, photographs, mathematical formulas, and program fragments. The model must permit the representation of such *objects*.

Just as with the whole of the document, the model takes into account the logical structure of objects of this type. Some are clearly structured, others are less so. Logical structure can be recognized in mathematical formulas, in tables, and in certain types of diagrams. On the other hand, it is difficult to define the structure of a photograph or of some drawings. But in any case, it does not seem possible to define one unique structure which can represent every one of these types of objects. The approach taken in the definition of meta–structure and document classes also applies to objects. Object classes can be defined which put together objects of similar type, constructed from the same generic logical structure.

Thus, a mathematical class can be defined and have a generic logical structure associated with it. But even if a single generic structure can represent a sufficient variety of mathematical formulas, for other objects with less rigorous structure, multiple classes must be defined. As for documents, using multiple classes assures that the model can describe the full range of objects to be presented. It also permits the system to support objects which were not initially anticipated. Moreover, this comment applies equally to mathematics: different classes of formulas can be described depending on the domain of mathematics being described.

Since objects have the same level of logical representation as documents, they gain the same advantages. In particular, it is possible to define the presentation separately from the objects themselves and attach it to the class. Thus, as for documents, objects of the same type have a uniform presentation and the presentation of every object in a given class can be changed simply by changing the generic presentation of the class. Another advantage of using this document model is that the system does not bother the user with the details of presentation, but rather allows the user to concentrate on the logical aspect of the document and the objects.

It is clear that the documents in a class do not necessarily use the same classes of objects: one technical report will contain tables while another report will have no tables but will use mathematical formulas. The usable object classes are not always mentioned in a limiting way in the generic logical structure of documents. Rather, they can be chosen freely from a large set, independent of the document class.

Thus, the object classes will be made commonplace and usable in every document. The notion of ‘‘object’’ can be enlarged to include not only non–textual elements, but also certain types of textual elements which can appear in practically every document, whatever their class. Among these textual elements, one can mention enumerations, descriptions, examples, quotations, even paragraphs.

Thus, the document model is not a single, general model describing every type of document in one place. Rather, it is a meta–model which can be used to describe many different models each of which represents either a class of similar documents or a class of similar objects which every document can include.

The S language

Document meta–structure

Since the concept of meta–structure is well suited to the task of describing documents at a high level of abstraction, this meta–structure must be precisely defined. Toward that end this section first presents the basic elements from which documents and structured objects are composed and then specifies the ways in which these basic elements are assembled into structures representing complete documents and objects.

The basic types

At the lowest level of a document’s structure, the first atom considered is the character. However, since characters are seldom isolated, usually appearing as part of a linear sequence, and in order to reduce the complexity of the document structure, *character strings* are used as atoms and consecutive characters belonging to the same structural element are grouped in the same character string.

If the structure of a document is not refined to go down to the level of words or phrases, the contents of a simple paragraph can be considered to be a single character string. On the other hand, the title of a chapter, the title of the first section of that chapter, and the text of the first paragraph of that section constitute three different character strings, because they belong to distinct structural elements.

If, instead, a very fine–grained representation for the structure of a document is sought, character strings could be defined to contain only a single word, or even just a single character. This is the case, for example, in programs, for which one wants to retain a structure very close to the syntax of the programming language. In this case, an assignment statement initializing a simple variable to zero would be composed of two structural elements, the identifier of the variable (a short character string) and the assigned value (a string of a single character, ‘0’).

The character string is not the only atom necessary for representing those documents that interest us. It suffices for purely textual documents, but as soon as the non–textual objects which we have considered arise, there must be other atoms; the number of objects which are to be represented determines the number of types of atoms that are necessary.

Primitive *graphical elements* are used for tables and figures of different types. These elements are simple geometric shapes like horizontal or vertical lines, which are sufficient for tables, or even oblique lines, arrows, rectangles, circles, polygons, and curves for use in figures. From these elements and character strings, graphical objects and tables can be constructed.

Photographs, though having very little structure, must still appear in documents. They are supported by *picture* elements, which are represented as matrices of pixels.

Finally, mathematical notations require certain elements which are simultaneously characters and graphical elements, the *symbols*. By way of example, radicals, integration signs, or even large parentheses are examples of this type of atom. The size of each of these symbols is determined by its environment, that is to say, by the expression to which it is attached.

To summarize, the primitive elements which are used in the construction of documents and structured objects are:

- character strings,
- graphical elements,
- pictures,
- and mathematical symbols.

Constructed elements

A document is evidently formed from primitive elements. But the model of Thot also proposes higher level elements. Thus, in a document composed of several chapters, each chapter is an element, and in the chapters each section is also an element, and so on. A document is thus an organized set of elements.

In a document there are different sorts of elements. Each element has a *type* which indicates the role of the element within the document as a whole. Thus, we have, for example, the chapter and section types. The document is made up of typed elements: elements of the type chapter and elements of the type section, among others, but also character string elements and graphical elements: the primitive elements are typed elements just as well. At the other extreme, the document itself is also considered to be a typed element.

The important difference between the primitive elements and the other elements of the document is that the primitive elements are atoms (they cannot be decomposed), whereas the others, called *constructed elements*, are composed of other elements, which can either be primitive elements or constructed elements. A constructed element of type chapter (or more simply, ‘‘a chapter’’) is composed of sections, which are also constructed elements. A paragraph, a constructed element, can be made up of character strings, which are primitive elements, and of equations, which are constructed elements.

A document is also a constructed element. This is an important point. In particular, it allows a document to be treated as part of another document, and conversely, permits a part of a document to be treated as a complete document. Thus, an article presented in a journal is treated by its author as a document in itself, while the editor of the journal considers it to be part of an issue. A table or a figure appearing in a document can be extracted and treated as a complete document, for example to prepare transparencies for a conference.

These thoughts about types and constructed elements apply just as well to objects as they do to documents. A table is a constructed element made up of other constructed elements, rows and columns. A row is formed of cells, which are also constructed elements which contain primitive elements (character strings) and/or constructed elements like equations.

Logical structure constructors

Having defined the primitive elements and the constructed elements, it is now time to define the types of organization which allow the building of structures. For this, we rely on the notion of the *constructor*. A constructor defines a way of assembling certain elements in a structure. It resides at the level of the meta–structure: it does not describe the existing relations in a given structure, but rather defines how elements are assembled to build a structure that conforms to a model.

In defining the overall organization of documents, the first two constructors considered are the aggregate and the list.

Aggregate and List

The *aggregate* constructor is used to define constructed element types which are collections of a given number of other elements. These collections may or may not be ordered. The elements may be either constructed or primitive and are specified by their type. A report (that is, a constructed element of the report type) has an aggregate structure. It is formed from a title, an author's name, an introduction, a body, and a conclusion, making it a collection of five element types. This type of constructor is found in practically every document, and generally at several levels in a document.

The *list* constructor is used to define constructed elements which are ordered sequences of elements (constructed or primitive) having the same type. The minimum and maximum numbers of elements for the sequence can be specified in the list constructor or the number of elements can be left unconstrained. The body of a report is a list of chapters and is typically required to contain a minimum of two chapters (is a chapter useful if it is the only one in the report?) The chapter itself can contain a list of sections, each section containing a list of paragraphs. In the same way as the aggregate, the list is a very frequently used constructor in every type of document. However, these two constructors are not sufficient to describe every document structure; thus other constructors supplement them.

Choice, Schema, and Unit

The *choice* constructor is used to define the structure of an element type for which one alternative is chosen from several possibilities. Thus, a paragraph can be either a simple text paragraph, or an enumeration, or a citation.

The choice constructor indicates the complete list of possible options, which can be too restrictive in certain cases, the paragraph being one such case. Two constructors, *unit* and *schema*, address this inconvenience. They allow more freedom in the choice of an element type. If a paragraph is defined by a schema constructor, it is possible to put in the place of a paragraph a table, an equation, a drawing or any other object defined by another generic logical structure. It is also possible to define a paragraph as a sequence of units, which could be character strings, symbols, or pictures. The choice constructor alone defines a generic logical structure that is relatively constrained; in contrast, using units and schemas, a very open structure can be defined.

The *schema* constructor represents an object defined by a generic logical structure chosen freely from among those available.

The *unit* constructor represents an element whose type can be either a primitive type or an element type defined as a unit in the generic logical structure of the document, or in another generic logical structure used in the document. Such an element may be used in document objects constructed according to other generic structures.

Thus, for example, if a cross–reference to a footnote is defined in the generic logical structure ‘‘Article’’ as a unit, a table (an object defined by another generic structure) can contain cross–references to footnotes, when they appear in an article. In another type of document, a table defined by the same generic structure can contain other types of elements, depending on the type of document into which the table is inserted. All that is needed is to declare, in the generic structure for tables, that the contents of cells are units. In this way, the generic structure of objects is divided up between different types of documents which are able to adapt themselves to the environment into which they are inserted.

Reference and Inclusion

The *reference* is used to define document elements that are cross–references to other elements, such as a section, a chapter, a bibliographic citation, or a figure. The reference is bi–directional. It can be used to access both the element being cross–referenced and each of the elements which make use of the cross–reference.

References can be either *internal* or *external*. That is, they can designate elements which appear in the same document or in another document.

The *inclusion* constructor is a special type of reference. Like the reference, it is an internal or external bidirectional link, but it is not a cross–reference. This link represents the ''live'' inclusion of the designated element; it accesses the most recent version of that element and not a ''dead'' copy, fixed in the state in which it was found at the moment the copy was made. As soon as an element is modified, all of its inclusions are automatically brought up to date. It must be noted that, in addition to inclusion, Thot permits the creation of ''dead'' copies.

There are three types of inclusions: inclusions with full expansion, inclusions with partial expansion, and inclusions without expansion. During editing, inclusions without expansion are represented on the screen by the name of the included document, in a special color, while inclusions with expansion (full or partial) are represented by a copy (full or partial) of the included element (also in a special color). The on–screen representation of a partial inclusion is a ''skeleton'' image of the included document.

Inclusion with complete expansion can be used to include parts of the same document or of other documents. Thus, it can be either an internal or an external link. It can be used to include certain bibliographic entries of a scientific article in another article, or to copy part of a mathematical formula into another formula of the same document, thus assuring that both copies will remain synchronized.

Inclusion without expansion or with partial expansion is used to include complete documents. It is always an external link. It is used primarily to divide very large documents into sub–documents that are easier to manipulate, especially when there are many authors. So, a book can include some chapters, where each chapter is a different document which can be edited separately. When viewing the book on the screen, it might be desirable to see only the titles of the chapters and sections. This can be achieved using inclusion with partial expansion.

During printing, inclusions without expansion or with partial expansion can be represented either as they were shown on the screen or by a complete (and up–to–date) copy of the included element or document.

The inclusion constructor, whatever its type, respects the generic structure: only those elements authorized by the generic structure can be included at a given position in a document.

Mark pairs

It is often useful to delimit certain parts of a document independently from the logical structure.

For example, one might wish to attach some information (in the form of an attribute) or a particular treatment to a group of words or a set of consecutive paragraphs. *Mark pairs* are used to do this.

Mark pairs are elements which are always paired and are terminals in the logical structure of the document. Their position in the structure of the document is defined in the generic structure. It is important to note that when the terminals of a mark pair are *extensions* (see the next section), they can be used quite freely.

Restrictions and Extensions

The primitive types and the constructors presented so far permit the definition of the logical structure of documents and objects in a rigorous way. But this definition can be very cumbersome in certain cases, notably when trying to constrain or extend the authorized element types in a particular context. *Restrictions* and *extensions* are used to cope with these cases.

A restriction associates with a particular element type *A*, a list of those element types which elements of type *A* may not contain, even if the definition of type *A* and those of its components authorize them otherwise. This simplifies the writing of generic logical structures and allows limitations to be placed, when necessary, on the choices offered by the schema and unit constructors.

Extensions are the inverse of restrictions. They identify a list of element types whose presence *is* permitted, even if its definition and those of its components do not authorize them otherwise.

Summary

Thus, four constructors are used to construct a document:

- the aggregate constructor (ordered or not),
- the list constructor,
- the choice constructor and its extensions, the unit and schema constructors,
- the reference constructor and its variant, the inclusion.

These constructors are also sufficient for objects. Thus, these constructors provide a homogenous meta–model which can describe both the organization of the document as a whole and that of the various types of objects which it contains. After presenting the description language for generic structures, we will present several examples which illustrate the appropriateness of the model.

The first three constructors (aggregate, list and choice) lead to tree–like structures for documents and objects, the objects being simply the subtrees of the tree of a document (or even of other objects' subtrees). The reference constructor introduces other, non–hierarchical, relations which augment those of the tree: when a paragraph makes reference to a chapter or a section, that relation leaves the purely tree−like structure. Moreover, external reference and inclusion constructors permit the establishment of links between different documents, thus creating a hypertext structure.

Associated Elements

Thanks to the list, aggregate and choice constructors, the organization of the document is specified rigorously, using constructed and primitive elements. But a document is made up of more than just its elements; it clearly also contains links between them. There exist elements whose position in the document's structure is not determinable. This is notably the case for figures and notes. A figure can be designated at many points in the same document and its place in the physical document can vary over the life of the document without any effect on the meaning or clarity of the document. At one time, it can be placed at the end of the document along with all other figures. At another time, it can appear at the top of the page which follows the first mention of the figure. The figures can be dispersed throughout the document or can be grouped together. The situation is similar for notes, which can be printed at the bottom of the page on which they are mentioned or assembled together at the end of the chapter or even the end of the work. Of course, this brings up questions of the physical position of elements in documents that are broken into pages, but this reflects the structural instability of these elements. They cannot be treated the same way as elements like paragraphs or sections, whose position in the structure is directly linked to the semantics of the document.

Those elements whose position in the structure of the document is not fixed, even though they are definitely part of the document, are called *associated elements*. Associated elements are themselves structures, which is to say that their content can be organized logically by the constructors from primitive and constructed elements.

It can happen that the associated elements are totally disconnected from the structure of the document, as in a commentary or appraisal of the entire work. But more often, the associated elements are linked to the content of the document by references. This is generally the case for notes and figures, among others.

Thus, associated elements introduce a new use for the reference constructor. It not only serves to create links between elements of the principal structure of the document, but also serves to link the associated elements to the primary structure.

Attributes

There remain logical aspects of documents that are not entirely described by the structure. Certain types of semantic information, which are not stated explicitly in the text, must also be taken into account. In particular, such information is shown by typographic effects which do not correspond to a change between structural elements. In fact, certain titles are set in bold or italic or are printed in a different typeface from the rest of the text in order to mark them as structurally distinct. But these same effects frequently appear in the middle of continuous text (e.g. in the interior of a paragraph). In this case, there is no change between structural elements; the effect serves to highlight a word, expression, or phrase. The notion of an *attribute* is used to express this type of information.

An attribute is a piece of information attached to a structural element which augments the type of the element and clarifies its function in the document. Keywords, foreign language words, and titles of other works can all be represented by character strings with attached attributes. Attributes may also be attached to constructed elements. Thus, an attribute indicating the language can be attached to a single word or to a large part of a document.

In fact, an attribute can be any piece of information which is linked to a part of a document and which can be used by agents which work on the document. For example, the language in which the document is written determines the set of characters used by an editor or formatter. It also determines the algorithm or hyphenation dictionary to be used. The attribute ''keyword'' facilitates the work of an information retrieval system. The attribute ''index word'' allows a formatter to automatically construct an index at the end of the document.

As with the types of constructed elements, the attributes and the values they can take are defined separately in each generic logical structure, not in the meta–model, according to the needs of the document class or the nature of the object.

Many types of attributes are offered: numeric, textual, references, and enumerations:

- *Numeric attributes* can take integer values (negative, positive, or null).
- *Textual attributes* have as their values character strings.
- *Reference attributes* designate an element of the logical structure.
- *Enumeration attributes* can take one value from a limited list of possible values, each value being a name.

In a generic structure, there is a distinction between *global attributes* and *local attributes*. A global attribute can be applied to every element type defined in the generic structure where it is specified. In contrast, a local attribute can only be applied to certain types of elements, even only a single type. The ''language'' attribute presented above is an example of a global attribute. An example of a local attribute is the rank of an author (principal author of the document or secondary author): this attribute can only be applied sensibly to an element of the ''author'' type.

Attributes can be assigned to the elements which make up the document in many different ways. The author can freely and dynamically place them on any part of the document in order to attach supplementary information of his/her choice. However, attributes may only be assigned in accordance with the rules of the generic structure; in particular, local attributes can only be assigned to those element types for which they are defined.

In the generic structure, certain local attributes can be made mandatory for certain element types. In this case, Thot automatically associates the attribute with the elements of this type and it requires the user to provide a value for this attribute.

Attributes can also be automatically assigned, with a given value, by every application processing the document in order to systematically add a piece of information to certain predefined elements of the document. By way of example, in a report containing a French abstract and an English abstract, each of the two abstracts is defined as a sequence of paragraphs. The first abstract has a value of ''French'' for the ''language'' attribute while the second abstract's ''language'' attribute has a value of ''English''.

In the case of mark pairs, attributes are logically associated with the pair as a whole, but are actually attached to the first mark.

Discussion of the model

The notions of attribute, constructor, structured element, and associated element are used in the definition of generic logical structures of documents and objects. The problem is to assemble them to form generic structures. In fact, many types of elements and attributes can be found in a variety of generic structures. Rather than redefine them for each structure in which they appear, it is best to share them between structures. The object classes already fill this sharing function. If a mathematical class is defined, its formulas can be used in many different document classes, without redefining the structure of each class. This problem arises not only for the objects considered here; it also arises for the commonplace textual elements found in many document classes. This is the reason why the notion of object is so broad and why paragraphs and enumerations are also considered to be objects. These object classes not only permit the sharing of the structures of elements, but also of the attributes defined in the generic structures.

Structure, such as that presented here, can appear very rigid, and it is possible to imagine that a document editing system based on this model could prove very constraining to the user. This is, in fact, a common criticism of syntax–directed editors. This defect can be avoided with Thot, primarily for three reasons:

- the generic structures are not fixed in the model itself,
- the model takes the dynamics of documents into account,
- the constructors offer great flexibility.

When the generic structure of a document is not predefined, but rather is constructed specifically for each document class, it can be carefully adapted to the current needs. In cases where the generic structure is inadequate for a particular document of the class, it is always possible either to create a new class with a generic structure well suited to the new case or to extend the generic structure of the existing class to take into account the specifics of the document which poses the problem. These two solutions can also be applied to objects whose structures prove to be poorly designed.

The model is sufficiently flexible to take into account all the phases of the life of the document. When a generic structure specifies that a report must contain a title, an abstract, an introduction, at least two chapters, and a conclusion, this means only that a report, *upon completion*, will have to contain all of these elements. When the author begins writing, none of these elements is present. Thot uses this model. Therefore, it tolerates documents which do not conform strictly to the generic structure of their class; it also considers the generic logical structure to be a way of helping the user in the construction of a complex document.

In contrast, other applications may reject a document which does not conform strictly to its generic structure. This is, for example, what is done by compilers which refuse to generate code for a program which is not syntactically correct. This might also occur when using a document application for a report which does not have an abstract or title.

The constructors of the document model bring a great flexibility to the generic structures. A choice constructor (and even more, a unit or schema constructor) can represent several, very different elements. The list constructor permits the addition of more elements of the same type. Used together, these two constructors permit any series of elements of different types. Of course, this flexibility can be reduced wherever necessary since a generic structure can limit the choices or the number of elements in a list.

Another difficulty linked to the use of structure in the document model resides in the choice of the level of the structure. The structure of a discussion could be extracted from the text itself via linguistic analysis. Some studies are exploring this approach, but the model of Thot excludes this type of structure. It only takes into account the logical structure provided explicitly by the author.

However, the level of structure of the model is not imposed. Each generic structure defines its own level of structure, adapted to the document class or object and to the ways in which it will be processed. If it will only be edited and printed, a relatively simple structure suffices. If more specialized processing will be applied to it, the structure must represent the element types on which this processing must act. By way of example, a simple structure is sufficient for printing formulas, but a more complex structure is required to perform symbolic or numeric calculations on the mathematical expressions. The document model of Thot allows both types of structure.

The definition language for generic structures

Generic structures, which form the basis of the document model of Thot, are specified using a special language. This definition language, called S, is described in this section.

Each generic structure, which defines a class of documents or objects, is specified by a file, written in the S language, which is called a *structure schema*. Structure schemas are compiled into tables, called structure tables, which are used by Thot and which determine its behavior.

Writing Conventions

The grammar of S, like those of the languages P and T presented later, is described using the meta−language M, derived from the Backus−Naur Form (BNF).

In this meta−language each rule of the grammar is composed of a grammar symbol followed by an equals sign ('=') and the right part of the rule. The equals sign plays the same role as the traditional '::=' of BNF: it indicates that the right part defines the symbol of the left part. In the right part,

concatenation
: is shown by the juxtaposition of symbols;

character strings
: between apostrophes ' represent terminal symbols, that is, keywords in the language defined. Keywords are written here in upper−case letters, but can be written in any combination of upper and lower−case letters. For example, the keyword `DEFPRES` of S can also be written as `defpres` or `DefPres`.

material between brackets
: ('[' and ']') is optional;

material between angle brackets
: ('<' and '>') can be repeated many times or omitted;

the slash
: ('/') indicates an alternative, a choice between the options separated by the slash character;

the period
: marks the end of a rule;

text between braces
: ('{' and '}') is simply a comment.

The M meta−language also uses the concepts of identifiers, strings, and integers:

`NAME`
: represents an identifier, a sequence of letters (upper or lower−case), digits, and underline characters ('_'), beginning with a letter. Also considered a letter is the sequence of characters '`\nnn`' where the letter `n` represents the ISO Latin−1 code of the letter in octal. It is thus possible to use accented letters in identifiers. The maximum length of identifiers is fixed by the compiler. It is normally 31 characters.
: Unlike keywords, upper and lower−case letters are distinct in identifiers. Thus, `Title`,

TITLE, and title are considered different identifiers.

STRING
: represents a string. This is a string of characters delimited by apostrophes. If an apostrophe must appear in a string, it is doubled. As with identifiers, strings can contain characters represented by their octal code (after a backslash). As with apostrophes, if a backslash must appear in a string, it is doubled.

NUMBER
: represents a positive integer or zero (without a sign), or said another way, a sequence of decimal digits.

The M language can be used to define itself as follows:

```
{ Any text between braces is a comment. }
Grammar       = Rule < Rule > 'END' .
                 { The < and > signs indicate zero }
                 { or more repetitions. }
                 { END marks the end of the grammar. }
Rule          = Ident '=' RightPart '.' .
                 { The period indicates the end of a rule }
RightPart     = RtTerminal / RtIntermed .
                 { The slash indicates a choice }
RtTerminal    ='NAME' / 'STRING' / 'NUMBER' .
                 { Right part of a terminal rule }
RtIntermed    = Possibility < '/' Possibility > .
                 { Right part of an intermediate rule }
Possibility   = ElemOpt < ElemOpt > .
ElemOpt       = Element / '[' Element < Element > ']' /
               '<' Element < Element > '>'  .
                 { Brackets delimit optional parts }
Element       = Ident / KeyWord .
Ident         = NAME .
                 { Identifier, sequence of characters }
KeyWord       = STRING .
                 { Character string delimited by apostrophes }
END
```

Extension schemas

A structure schema defines the generic logical structure of a class of documents or objects, independent of the operations which can be performed on the documents. However, certain applications may require particular information to be represented by the structure for the documents that they operate on. Thus a document version manager will need to indicate in the document the parts which belong to one version or another. An indexing system will add highly−structured index tables as well as the links between these tables and the rest of the document.

Thus, many applications need to extend the generic structure of the documents on which they operate to introduce new attributes, associated elements or element types. These additions are specific to each application and must be able to be applied to any generic structure: users will want to manage versions or construct indices for many types of documents. Extension schemas fulfill this role: they define attributes, elements, associated elements, units, etc., but they can only be used jointly with a structure schema that they complete. Otherwise, structure schemas can always be used without these extensions when the corresponding applications are not available.

The general organization of structure schemas

Every structure schema begins with the keyword `STRUCTURE` and ends with the keyword `END`. The keyword `STRUCTURE` is followed by the keyword `EXTENSION` in the case where the schema defines an extension, then by the name of the generic structure which the schema defines (the name of the document or object class). The name of the structure is followed by a semicolon.

In the case of a complete schema (that is, a schema which is not an extension), the definition of the name of the structure is followed by the declarations of the default presentation schema, the global attributes, the parameters, the structure rules, the associated elements, the units, the skeleton elements and the exceptions. Only the definition of the structure rules is required. Each series of declarations begins with a keyword: `DEFPRES`, `ATTR`, `PARAM`, `STRUCT`, `ASSOC`, `UNITS`, `EXPORT`, `EXCEPT`.

In the case of an extension schema, there are neither parameters nor skeleton elements and the `STRUCT` section is optional, while that section is required in a schema that is not an extension. On the other hand, extension schemas can contain an `EXTENS` section, which must not appear in a schema which is not an extension; this section defines the complements to attach to the rules found in the schema to which the extension will be added. The sections `ATTR`, `STRUCT`, `ASSOC`, and `UNITS` define new attributes, new elements, new associated elements, and new units which add their definitions to the principal schema.

```
StructSchema ='STRUCTURE' ElemID ';'
               'DEFPRES' PresID ';'
             [ 'ATTR' AttrSeq ]
             [ 'PARAM' RulesSeq ]
               'STRUCT' RulesSeq
             [ 'ASSOC' RulesSeq ]
             [ 'UNITS' RulesSeq ]
             [ 'EXPORT' SkeletonSeq ]
             [ 'EXCEPT' ExceptSeq ]
              'END' .
ElemID        = NAME .
```

or

```
ExtensSchema ='STRUCTURE' 'EXTENSION' ElemID ';'
               'DEFPRES' PresID ';'
             [ 'ATTR' AttrSeq ]
             [ 'STRUCT' RulesSeq ]
             [ 'EXTENS' ExtensRuleSeq ]
             [ 'ASSOC' RulesSeq ]
             [ 'UNITS' RulesSeq ]
             [ 'EXCEPT' ExceptSeq ]
```

```
                 'END' .
ElemID         = NAME .
```

The default presentation

It was shown above that many different presentations are possible for documents and objects of the same class. The structure schema defines a preferred presentation for the class, called the *default presentation*. Like generic structures, presentations are described by programs, called *presentation schemas*, which are written in a specific language, P, presented later in this document. The name appearing after the keyword `DEFPRES` is the name of the default presentation schema. When a new document is created, Thot will use this presentation schema by default, but the user remains free to choose another if s/he wishes.

```
PresID = NAME .
```

Global Attributes

If the generic structure includes global attributes of its own, they are declared after the keyword `ATTR`. Each global attribute is defined by its name, followed by an equals sign and the definition of its type. The declaration of a global attribute is terminated by a semi–colon.

For attributes of the numeric, textual, or reference types, the type is indicated by a keyword, `INTEGER`, `TEXT`, or `REFERENCE` respectively.

In the case of a reference attribute, the keyword `REFERENCE` is followed by the type of the referenced element in parentheses. It can refer to any type at all, specified by using the keyword `ANY`, or to a specific type. In the latter case, the element type designated by the reference can be defined either in the `STRUCT` section of the same structure schema or in the `STRUCT` section of another structure schema. When the type is defined in another schema, the element type is followed by the name of the structure schema (within parentheses) in which it is defined. The name of the designated element type can be preceded by the keyword `First` or `Second`, but only in the case where the type is defined as a pair. These keywords indicate whether the attribute must designate the first mark of the pair or the second. If the reference refers to a pair and neither of these two keywords is present, then the first mark is used.

In the case of an enumeration attribute, the equals sign is followed by the list of names representing the possible values of the attribute, the names being separated from each other by commas. An enumeration attribute has at least one possible value; the maximum number of values is defined by the compiler for the S language.

```
AttrSeq   = Attribute < Attribute > .
Attribute = AttrID '=' AttrType  ';' .
AttrType  = 'INTEGER' / 'TEXT' /
            'REFERENCE' '(' RefType ')' /
            ValueSeq .
RefType   = 'ANY' / [ FirstSec ] ElemID [ ExtStruct ] .
FirstSec  = 'First' / 'Second' .
```

```
ExtStruct = '(' ElemID ')' .
ValueSeq  = AttrVal < ',' AttrVal > .
AttrID    = NAME .
AttrVal   = NAME .
```

There is a predefined global text attribute, the *language*, which is automatically added to every Thot structure schema. This attribute allows Thot to perform certain actions, such as hyphenation and spell–checking, which cannot be performed without knowing the language in which each part of the document is written. This attribute can be used just like any explicitly declared attribute: the system acts as if every structure schema contains

```
ATTR
   Language = TEXT;
```

Example:

The following specification defines the global enumeration attribute WordType.

```
ATTR
   WordType = Definition, IndexWord, DocumentTitle;
```

Parameters

A parameter is a document element which can appear many times in the document, but always has the same value. This value can only be modified in a controlled way by certain applications. For example, in an advertising circular, the name of the recipient may appear in the address part and in the text of the circular. If the recipient's name were a parameter, it might only be able to be changed by a ''mail–merge'' application.

Parameters are not needed for every document class, but if the schema includes parameters they are declared after the keyword `PARAM`. Each parameter declaration is made in the same way as a structure element declaration.

During editing, Thot permits the insertion of parameters wherever the structure schema allows; it also permits the removal of parameters which are already in the document but does not allow the modification of the parameter's content in any way. The content is generated automatically by Thot during the creation of the parameter, based on the value of the parameter in the document.

Structured elements

The rules for defining structured elements are required, except in an extension schema: they constitute the core of a structure schema, since they define the structure of the different types of elements that occur in a document or object of the class defined by the schema.

The first structure rule after the keyword `STRUCT` must define the structure of the class whose name appears in the first instruction (`STRUCTURE`) of the schema. This is the root rule of the schema, defining the root of the document tree or object tree.

The remaining rules may be placed in any order, since the language permits the definition of element types before or after their use, or even in the same instruction in which they are used. This last case allows the definition of recursive structures.

Each rule is composed of a name (the name of the element type whose structure is being defined) followed by an equals sign and a structure definition.

If any local attributes are associated with the element type defined by the rule, they appear between parentheses after the type name and before the equals sign. The parentheses contain, first, the keyword `ATTR`, then the list of local attributes, separated by commas. Each local attribute is composed of the name of the attribute followed by an equals sign and the definition of the attribute's type, just as in the definition of global attributes. The name of the attribute can be preceded by an exclamation point to indicate that the attribute must always be present for this element type. The same attribute, identified by its name, can be defined as a local attribute for multiple element types. In this case, the equals sign and definition of the attribute type need only appear in the first occurrence of the attribute. It should be noted that global attributes cannot also be defined as local attributes.

If any extensions are defined for this element type, a plus sign follows the structure definition and the names of the extension element types appear between parentheses after the plus. If there are multiple extensions, they are separated by commas. These types can either be defined in the same schema, defined in other schemas, or they may be base types identified by the keywords `TEXT`, `GRAPHICS`, `SYMBOL`, or `PICTURE`.

Restrictions are indicated in the same manner as extensions, but they are introduced by a minus sign and they come after the extensions, or if there are no extensions, after the structure definition.

If the values of attributes must be attached systematically to this element type, they are introduced by the keyword `WITH` and declared in the form of a list of fixed–value attributes. When such definitions of fixed attribute values appear, they are always the last part of the rule.

The rule is terminated by a semicolon.

```
RuleSeq        = Rule < Rule > .
Rule            = ElemID [ LocAttrSeq ] '=' DefWithAttr ';'.
LocAttrSeq     = '(' 'ATTR' LocAttr < ';' LocAttr > ')' .
LocAttr        = [ '!' ] AttrID [ '=' AttrType ] .
DefWithAttr    = Definition
                 [ '+' '(' ExtensionSeq ')' ]
                 [ '-' '(' RestrictSeq ')' ]
                 [ 'WITH' FixedAttrSeq ] .
ExtensionSeq  = ExtensionElem < ',' ExtensionElem > .
ExtensionElem = ElemID / 'TEXT' / 'GRAPHICS' /
                'SYMBOL' / 'PICTURE' .
RestrictSeq    = RestrictElem < ',' RestrictElem > .
```

```
RestrictElem  = ElemID / 'TEXT' / 'GRAPHICS' /
                'SYMBOL' / 'PICTURE' .
```

The list of fixed−value attributes is composed of a sequence of attribute−value pairs separated by commas. Each pair contains the name of the attribute and the fixed value for this element type, the two being separated by an equals sign. If the sign is preceded by a question mark the given value is only an initial value that may be modified later rather than a value fixed for all time. Reference attributes are an exception to this norm. They cannot be assigned a fixed value, but when the name of such an attribute appears this indicates that this element type must have a valid value for the attribute. For the other attribute types, the fixed value is indicated by a signed integer (numeric attributes), a character string between apostrophes (textual attributes) or the name of a value (enumeration attributes).

Fixed−value attributes can either be global or local to the element type for which they are fixed, but they must be declared before they are used.

```
FixedAttrSeq     = FixedAttr < ',' FixedAttr > .
FixedAttr        = AttrID [ FixedOrModifVal ] .
FixedOrModifVal = [ '?' ] '=' FixedValue .
FixedValue       = [ '-' ] NumValue / TextVal / AttrVal .
NumValue         = NUMBER .
TextVal          = STRING .
```

Structure definitions

The structure of an element type can be a simple base type or a constructed type.

For constructed types, it is frequently the case that similar structures appear in many places in a document. For example the contents of the abstract, of the introduction, and of a section can have the same structure, that of a sequence of paragraphs. In this case, a single, common structure can be defined (the paragraph sequence in this example), and the schema is written to indicate that each element type possesses this structure, as follows:

```
Abstract            = Paragraph_sequence;
Introduction        = Paragraph_sequence;
Section_contents    = Paragraph_sequence;
```

The equals sign means ‘‘has the same structure as’’.

If the element type defined is a simple base type, this is indicated by one of the keywords `TEXT`, `GRAPHICS`, `SYMBOL`, or `PICTURE`. If some local attributes must be associated with a base type, the keyword of the base type is followed by the declaration of the local attributes using the syntax presented above.

In the case of an open choice, the type is indicated by the keyword `UNIT` for units or the keyword `NATURE` for objects having a structure defined by any other schema.

A unit represents one of the two following categories:

- a base type: text, graphical element, symbol, picture,

- an element whose type is chosen from among the types defined as units in the `UNITS` section of the document's structure schema. It can also be chosen from among the types defined as units in the `UNITS` section of the structure schemas that defines the ancestors of the element to which the rule is applied.

Before the creation of an element defined as a unit, Thot asks the user to choose between the categories of elements.

Thus, the contents of a paragraph can be specified as a sequence of units, which will permit the inclusion in the paragraphs of character strings, symbols, and various elements, such as cross–references, if these are defined as units.

A schema object (keyword `NATURE`) represents an object defined by a structure schema freely chosen from among the available schemas; in the case the element type is defined by the first rule (the root rule) of the chosen schema.

If the element type defined is a constructed type, the list, aggregate, choice, and reference constructors are used. In this case the definition begins with a keyword identifying the constructor. This keyword is followed by a syntax specific to each constructor.

The local attribute definitions appear after the name of the element type being defined, if this element type has local attributes.

```
Definition = BaseType [ LocAttrSeq ] / Constr / Element .
BaseType   = 'TEXT' / 'GRAPHICS' / 'SYMBOL' / 'PICTURE' /
             'UNIT' / 'NATURE' .
Element    = ElemID [ ExtOrDef ] .
ExtOrDef   = 'EXTERN' / 'INCLUDED' /
              [ LocAttrSeq ] '=' Definition .
Constr     = 'LIST' [ '[' min '..' max ']' ] 'OF'
                     '(' DefWithAttr ')' /
             'BEGIN' DefOptSeq 'END' /
             'AGGREGATE' DefOptSeq 'END' /
             'CASE' 'OF' DefSeq 'END' /
             'REFERENCE' '(' RefType ')' /
             'PAIR' .
```

List

The list constructor permits the definition of an element type composed of a list of elements, all of the same type. A list definition begins with the `LIST` keyword followed by an optional range, the keyword `OF`, and the definition, between parentheses, of the element type which must compose the list. The optional range is composed of the minimum and maximum number of elements for the list separated by two periods and enclosed by brackets. If the range is not present, the number of list elements is unconstrained. When only one of the two bounds of the range is unconstrained, it is represented by a star ('*') character. Even when both bounds are unconstrained, they can be specified by `[*..*]`, but it is simpler not to specify any bound.

```
'LIST' [ '[' min '..' max ']' ]
```

```
            'OF' '(' DefWithAttr ')'
min      = Integer / '*' .
max      = Integer / '*' .
Integer = NUMBER .
```

Before the document is edited, Thot creates the minimum number of elements for the list. If no minimum was given, it creates a single element. If a maximum number of elements is given and that number is attained, the editor refuses to create new elements for the list.

Example:

The following two instructions define the body of a document as a sequence of at least two chapters and the contents of a section as a sequence of paragraphs. A single paragraph can be the entire contents of a section.

```
Body             = LIST [2..*] OF (Chapter);
Section_contents = LIST OF (Paragraph);
```

Aggregate

The aggregate constructor is used to define an element type as a collection of sub–elements, each having a fixed type. The collection may be ordered or unordered. The elements composing the collection are called *components*. In the definition of an aggregate, a keyword indicates whether or not the aggregate is ordered: `BEGIN` for an ordered aggregate, `AGGREGATE` for an unordered aggregate. This keyword is followed by the list of component type definitions which is terminated by the `END` keyword. The component type definitions are separated by commas.

Before creating an aggregate, the Thot editor creates all the aggregate's components in the order they appear in the structure schema, even for unordered aggregates. However, unlike ordered aggregates, the components of an unordered aggregate may be rearranged using operations of the Thot editor. The exceptions to the rule are any components whose name was preceded by a question mark character ('?'). These components, which are optional, can be created by explicit request, possibly at the time the aggregate is created, but they are not created automatically *prior* to the creation of the aggregate.

```
             'BEGIN' DefOptSeq 'END'
DefOptSeq = DefOpt ';' < DefOpt ';' > .
DefOpt    = [ '?' ] DefWithAttr .
```

Example:

In a bilingual document, each paragraph has an English version and a French version. In certain cases, the translator wants to add a marginal note, but this note is present in very few paragraphs. Thus, it must not be created systematically for every paragraph. A bilingual paragraph of this type is declared:

```
Bilingual_paragraph = BEGIN
                      French_paragraph  = TEXT;
                      English_paragraph = TEXT;
                      ? Note            = TEXT;
```

```
END;
```

Choice

The choice constructor permits the definition of an element type which is chosen from among a set of possible types. The keywords CASE and OF are followed by a list of definitions of possible types, which are separated by semicolons and terminated by the END keyword.

```
             'CASE' 'OF' DefSeq 'END'
DefSeq = DefWithAttr ';' < DefWithAttr ';' > .
```

Before the creation of an element defined as a choice, the Thot editor presents the list of possible types for the element to the user. The user has only to select the element type that s/he wants to create from this list.

The order of the type declarations is important. It determines the order of the list presented to the user before the creation of the element. Also, when a Choice element is being created automatically, the first type in the list is used. In fact, using the Thot editor, when an empty Choice element is selected, it is possible to select this element and to enter its text from keyboard. In this case, the editor uses the first element type which can contain an atom of the character string type.

The two special cases of the choice constructor, the *schema* and the *unit* are discussed elsewhere.

> **Example:**
>
> It is common in documents to treat a variety of objects as if they were ordinary paragraphs. Thus, a ''Paragraph'' might actually be composed of a block of text (an ordinary paragraph), or a mathematical formula whose structure is defined by another structure schema named Math, or a table, also defined by another structure schema. Here is a definition of such a paragraph:

```
Paragraph = CASE OF
              Simple_text = TEXT;
              Formula     = Math;
              Table_para  = Table;
              END;
```

Reference

Like all elements in Thot, references are typed. An element type defined as a reference is a cross–reference to an element of some other given type. The keyword REFERENCE is followed by the name of a type enclosed in parentheses. When the type which is being cross–referenced is defined in another structure schema, the type name is itself followed by the name of the external structure schema in which it is defined.

When the designated element type is a mark pair, it can be preceded by a FIRST or SECOND keyword. These keywords indicate whether the reference points to the first or second mark of the pair. If the reference points to a pair and neither of these two keywords is present, the reference is considered to point to the first mark of the pair.

There is an exception to the principle of typed references: it is possible to define a reference which designates an element of any type, which can either be in the same document or another document. In this case, it suffices to put the keyword `ANY` in the parentheses which indicate the referenced element type.

```
            'REFERENCE' '(' RefType ')'
   RefType = 'ANY' / [ FirstSec ] ElemID [ ExtStruct ] .
```

When defining an inclusion, the `REFERENCE` keyword is not used. Inclusions with complete expansion are not declared as such in the structure schemas, since any element defined in a structure schema can be replaced by an element of the same type. Instead, inclusions without expansion or with partial expansion must be declared explicitly whenever they will include a complete object (and not a part of an object). In this case, the object type to be included (that is, the name of its structure schema) is followed by a keyword: `EXTERN` for inclusion without expansion and `INCLUDED` for partial expansion.

Before creating a cross–reference or an inclusion, the Thot editor asks the user to choose, from the document images displayed, the referenced or included element.

Example:

If the types Note and Section are defined in the Article structure schema, it is possible to define, in the same structure schema, a reference to a note and a reference to a section in this manner:

```
Ref_note     = REFERENCE (Note);
Ref_section = REFERENCE (Section);
```

It is also possible to define the generic structure of a collection of articles, which include (with partial expansion) objects of the Article class and which possess an introduction which may include cross–references to sections of the included articles. In the Collection structure schema, the definitions are:

```
Collection = BEGIN
              Collection_title = TEXT;
              Introduction = LIST OF (Elem = CASE OF
                                              TEXT;
                                              Ref_sect;
                                              END);
              Body = LIST OF (Article INCLUDED);
             END;
Ref_sect    = REFERENCE (Section (Article));
```

Here we define a Folder document class which has a title and includes documents of different types, particularly Folders:

```
Folder    = BEGIN
```

```
            Folder_title    = TEXT;
            Folder_contents = LIST OF (Document);
            END;

Document = CASE OF
              Article EXTERN;
              Collection EXTERN;
              Folder EXTERN;
              END;
```

Under this definition, Folder represents either an aggregate which contains a folder title and the list of included documents or an included folder. To resolve this ambiguity, in the P language, the placement of a star character in front of the type name (here, Folder) indicates an included document.

Mark pairs

Like other elements, mark pairs are typed. The two marks of the pair have the same type, but there exist two predefined subtypes which apply to all mark pairs: the first mark of the pair (called `First` in the P and T languages) and the second mark (called `Second`).

In the S language, a mark pair is noted simply by the `PAIR` keyword.

In the Thot editor, marks are always moved or destroyed together. The two marks of a pair have the same identifier, unique within the document, which permits intertwining mark pairs without risk of ambiguity.

Imports

Because of schema constructors, it is possible, before editing a document, to use classes defined by other structure schemas whenever they are needed. It is also possible to assign specific document classes to certain element types. In this case, these classes are simply designated by their name. In fact, if a type name is not defined in the structure schema, it is assumed that it specifies a structure defined by another structure schema.

Example:

If the types Math and Table don't appear in the left part of a structure rule in the schema, the following two rules indicate that a formula has the structure of an object defined by the structure schema Math and that a table element has the structure of an object defined by the Table schema.

```
Formula    = Math;
Table_elem = Table;
```

Extension rules

The `EXTENS` section, which can only appear in an extension schema, defines complements to the rules in the primary schema (i.e. the structure schema to which the extension schema will be applied). More precisely, this section permits the addition to an existing type of local attributes, extensions, restrictions and fixed−value attributes.

These additions can be applied to the root rule of the primary schema, designated by the keyword `Root`, or to any other explicitly named rule.

Extension rules are separated from each other by a semicolon and each extension rule has the same syntax as a structure rule, but the part which defines the constructor is absent.

```
ExtenRuleSeq = ExtensRule ';' < ExtensRule ';' > .
ExtensRule   = RootOrElem [ LocAttrSeq ]
               [ '+' '(' ExtensionSeq ')' ]
               [ '-' '(' RestrictSeq ')' ]
               [ 'WITH' FixedAttrSeq ] .
RootOrElem   = 'Root' / ElemID .
```

Associated elements

If associated elements are necessary, they must be declared in a specific section of the structure schema, introduced by the keyword `ASSOC`. Each associated element type is specified like any other structured element. However, these types must not appear in any other element types of the schema, except in `REFERENCE` rules.

Units

The `UNITS` section of the structure schema contains the declarations of the element types which can be used in the external objects making up parts of the document or in objects of the class defined by the schema. As with associated elements, these element types are defined just like other structured element types. They can be used in the other element types of the schema, but they can also be used in any other rule of the schema.

Example:

If references to notes are declared as units:

```
UNITS
   Ref_note = REFERENCE (Note);
```

then it is possible to use references to notes in a cell of a table, even when `Table` is an external structure schema. The `Table` schema must declare a cell to be a sequence of units, which can then be base element types (text, for example) or references to notes in the document.

```
Cell = LIST OF (UNITS);
```

Skeleton elements

When editing a document which contains or must contain external references to several other documents, it may be necessary to load a large number of documents, simply to see the parts designated by the external references of the document while editing, or to access the source of included elements. In this case, the external documents are not modified and it is only necessary to see the elements of these documents which could be referenced. Because of this, the editor will suggest that the documents be loaded in ‘‘skeleton’’ form. This form contains only the elements of the document explicitly mentioned in the `EXPORT` section of their structure schema and, for these elements, only the part of the contents specified in that section. This form has the advantage of being very compact, thus requiring very few resources from the editor. This is also the skeleton form which constitutes the expanded form of inclusions with partial expansion.

Skeleton elements must be declared explicitly in the `EXPORT` section of the structure schema that defines them. This section begins with the keyword `EXPORT` followed by a comma–separated list of the element types which must appear in the skeleton form and ending with a semicolon. These types must have been previously declared in the schema.

For each skeleton element type, the part of the contents which is loaded by the editor, and therefore displayable, can be specified by putting the keyword `WITH` and the name of the contained element type to be loaded after the name of the skeleton element type. In this case only that named element, among all the elements contained in the exportable element type, will be loaded. If the `WITH` is absent, the entire contents of the skeleton element will be loaded by the editor. If instead, it is better that the skeleton form not load the contents of a particular element type, the keyword `WITH` must be followed by the word `Nothing`.

```
             [ 'EXPORT' SkeletonSeq ]

SkeletonSeq = SkelElem < ',' SkelElem > ';' .
SkelElem    = ElemID [ 'WITH' Contents ] .
Contents    = 'Nothing' / ElemID [ ExtStruct ] .
```

Example:

Suppose that, in documents of the article class, the element types Article_title, Figure, Section, Paragraph, and Biblio should appear in the skeleton form in order to make it easier to create external references to them from other documents. When loading an article in its skeleton form, all of these element types will be loaded except for paragraphs, but only the article title will be loaded in its entirety. For figures, the caption will be loaded, while for sections, the title will be loaded, and for bibliographic entries, only the title that they contain will be loaded. Note that bibliographic elements are defined in another structure schema, RefBib. To produce this result, the following declarations should be placed in the Article structure schema:

```
EXPORT
   Article_title,
   Figure With Caption,
   Section With Section_title,
   Paragraph With Nothing,
   Biblio With Biblio_title(RefBib);
```

Exceptions

The behavior of the Thot editor and the actions that it performs are determined by the structure schemas. These actions are applied to all document and object types in accordance with their generic structure. For certain object types, such as tables and graphics, these actions are not sufficient or are poorly adapted and some special actions must be added to or substituted for certain standard actions. These special actions are called *exceptions*.

Exceptions only inhibit or modify certain standard actions, but they can be used freely in every structure schema.

Each structure schema can contain a section defining exceptions. It begins with the keyword `EXCEPT` and is composed of a sequence of exception declarations, separated by semicolons. Each declaration of an exception begins with the name of an element type or attribute followed by a colon. This indicates the element type or attribute to which the following exceptions apply. When the given element type name is a mark pair, and only in this case, the type name can be preceded by the keyword `First` or `Second`, to indicate if the exceptions which follow are associated with the first mark of the pair or the second. In the absence of this keyword, the first mark is used.

When placed in an extension schema, the keyword `EXTERN` indicates that the type name which follows is found in the principal schema (the schema being extended by the extension schema). The exceptions are indicated by a name. They are separated by semicolons.

```
               [ 'EXCEPT' ExceptSeq ]

ExceptSeq      = Except ';' < Except ';' > .
Except         = [ 'EXTERN' ] [ FirstSec ] ExcTypeOrAttr
                 ':' ExcValSeq .
ExcTypeOrAttr = ElemID / AttrID .
ExcValSeq      = ExcValue < ',' ExcValue > .
ExcValue       ='NoCut' / 'NoCreate' / 'NoHMove' /
                 'NoVMove' / 'NoHResize' / 'NoVResize' /
                 'NoMove' / 'NoResize' / 'MoveResize' /
                 'NewWidth' / 'NewHeight' / 'NewHPos' /
                 'NewVPos' / 'Invisible' /
                 'NoSelect' / 'NoSpellCheck' /
                 'Hidden' / 'ActiveRef' /
                 'ImportLine' / 'ImportParagraph' /
                 'NoPaginate' / 'ParagraphBreak' /
                 'PageBreak' / 'PageBreakAllowed' / 'PageBreakPlace' /
                 'PageBreakRepetition' / 'PageBreakRepBefore' /
                 'HighlightChildren' / 'ExtendedSelection' /
```

```
                    'ReturnCreateNL' / 'ReturnCreateWithin' /
                    'IsDraw' / 'IsTable' /
                    'IsRow' / 'IsColHead' / 'IsCell' /
                    'NewPercentWidth' / 'ColRef' / 'ColSpan' /
                    'RowSpan' / 'SaveDocument' / 'Shadow' .
```

The following are the available exceptions:

`NoCut`
: This exception can only be applied to element types. Elements of a type to which this exception is applied cannot be deleted by the editor.

`NoCreate`
: This exception can only be applied to element types. Elements of a type to which this exception is applied cannot be created by ordinary commands for creating new elements. These elements are usually created by special actions associated with other exceptions.

`NoHMove`
: This exception can only be applied to element types. Elements of a type to which this exception is applied cannot be moved horizontally with the mouse. Their children elements cannot be moved either.

`NoVMove`
: This exception can only be applied to element types. Elements of a type to which this exception is applied cannot be moved vertically with the mouse. Their children elements cannot be moved either.

`NoMove`
: This exception can only be applied to element types. Elements of a type to which this exception is applied cannot be moved in any direction with the mouse. Their children elements cannot be moved either.

`NoHResize`
: This exception can only be applied to element types. Elements of a type to which this exception is applied cannot be resized horizontally with the mouse. Their children elements cannot be resized either.

`NoVResize`
: This exception can only be applied to element types. Elements of a type to which this exception is applied cannot be resized vertically with the mouse. Their children elements cannot be resized either.

`NoResize`
: This exception can only be applied to element types. Elements of a type to which this exception is applied cannot be resized in any direction with the mouse. Their children elements cannot be resized either.

`MoveResize`
: This exception can only be applied to element types. Elements of a type to which this exception is applied can be moved and resized in any direction with the mouse, even if one of their ancestor element has an exception that prevents moving or resizing. Their children elements can also be resized or moved.

`NewWidth`
: This exception can only be applied to numeric attributes. If the width of an element

which has this attribute is modified with the mouse, the value of the new width will be assigned to the attribute.

`NewHeight`
: This exception can only be applied to numeric attributes. If the height of an element which has this attribute is modified with the mouse, the value of the new height will be assigned to the attribute.

`NewHPos`
: This exception can only be applied to numeric attributes. If the horizontal position of an element which has this attribute is modified with the mouse, the value of the new horizontal position will be assigned to the attribute.

`NewVPos`
: This exception can only be applied to numeric attributes. If the vertical position of an element which has this attribute is modified with the mouse, the value of the new vertical position will be assigned to the attribute.

`Invisible`
: This exception can only be applied to attributes, but can be applied to all attribute types. It indicates that the attribute must not be seen by the user and that its value must not be changed directly. This exception is usually used when another exception manipulates the value of an attribute.

`NoSelect`
: This exception can only be applied to element types. Elements of a type to which this exception is applied cannot be selected directly with the mouse, but they can be selected by other methods provided by the editor.

NoSpellCheck
: This exception can only be applied to element types. Elements of a type to which this exception is applied are not taken into account by the spell checker.

`Hidden`
: This exception can only be applied to element types. It indicates that elements of this type, although present in the document's structure, must not be shown to the user of the editor. In particular, the creation menus must not propose this type and the selection message must not pick it.

`ActiveRef`
: This exception can only be applied to attributes of the reference type. It indicates that when the user of the editor makes a double click on an element which possesses a reference attribute having this exception, the element designated by the reference attribute will be selected.

`ImportLine`
: This exception can only be applied to element types. It indicates that elements of this type should receive the content of imported text files. An element is created for each line of the imported file. A structure schema cannot contain several exceptions `ImportLine` and, if it contains one, it should not contain any exception `ImportParagraph`.

`ImportParagraph`

This exception can only be applied to element types. It indicates that elements of this type should receive the content of imported text files. An element is created for each paragraph of the imported file. A paragraph is a sequence of lines without any empty line. A structure schema cannot contain several exceptions `ImportParagraph` and, if it contains one, it should not contain any exception `ImportLine`.

`NoPaginate`
: This exception can only be applied to the root element, i.e. the name that appear after the keyword `STRUCTURE` at the beginning of the structure schema. It indicates that the editor should not allow the user to paginate documents of that type.

`ParagraphBreak`
: This exception can only be applied to element types. When the caret is within an element of a type to which this exception is applied, it is that element that will be split when the user hits the Return key.

`ReturnCreateNL`
: This exception can only be applied to element types. When the caret is within an element of a type to which this exception is applied, the Return key simply inserts a New line character (code \212) at the current position. The Return key does not create a new element; it does not split the current element either.

`ReturnCreateWithin`
: This exception can only be applied to element types. When the caret is within an element of a type to which this exception is applied, the Return key will create a new element within that element, not a sibling after that element.

`HighlightChildren`
: This exception can only be applied to element types. Elements of a type to which this exception is applied are not highlighted themselves when they are selected, but all their children are highlighted instead.

`ExtendedSelection`
: This exception can only be applied to element types. The selection extension command (middle button of the mouse) only add the clicked element (if it has that exception) to the current selection, without selecting other elements between the current selection and the clicked element.

IsDraw, IsTable, IsColHead, IsRow, IsCell
: These exceptions can only be applied to element types. Elements of a type to which these exceptions are applied are identified as Draws, Tables, Colheads, Rows or Cells and specific processing are applied to them.

ColRef
: This exception can only be applied to attributes of the reference type. It indicates that this attribute refers to the column head (see exception IsColHead) which the element belongs to.

ColSpan, RowSpan
: These exceptions can only be applied to numeric attributes of cells. They indicate that

attribute values give how many columns or rows the element spans.

Shadow
: This exception can only be applied to element types. Text of elements of a type to which this exception is applied are displayed and printed as a set of stars ('*').

Example:

Consider a structure schema for object−style graphics which defines the Graphic_object element type with the associated Height and Weight numeric attributes. Suppose that we want documents of this class to have the following qualities:

- Whenever the width or height of an object is changed using the mouse, the new values are stored in the object's Width and Height attributes.
- The user should not be able to change the values of the Width and Height attributes via the Attributes menu of the Thot editor.

The following exceptions will produce this effect.

```
STRUCT
...
   Graphics_object (ATTR Height = Integer; Width = Integer)
        = GRAPHICS with Height ?= 10, Width ?= 10;
...
EXCEPT
   Height: NewHeight, Invisible;
   Width: NewWidth, Invisible;
```

Some examples

In order to illustrate the principles of the document model and the syntax of the S language, this section presents two examples of structure schemas. One defines a class of documents, the other defines a class of objects.

A class of documents: articles

This example shows a possible structure for articles published in a journal. Text between braces is comments.

```
STRUCTURE Article;  { This schema defines the Article class }
DEFPRES ArticleP;   { The default presentation schema is
                      ArticleP }
ATTR                { Global attribute definitions }
   WordType = Definition, IndexWord, DocumentTitle;
   { A single global attribute is defined, with three values }
STRUCT              { Definition of the generic structure }
   Article = BEGIN  { The Article class has an aggregate
```

```
                 structure }
           Title = BEGIN   { The title is an aggregate }
                   French_title =
                        Text WITH Language='Fran\347ais';
                   English_title =
                        Text WITH Language='English';
                   END;
           Authors =
             LIST OF (Author
                (ATTR Author_type=principal,secondary)
               { The Author type has a local attribute }
               = BEGIN
                 Author_name = Text;
                 Info = Paragraphs ;
                 { Paragraphs is defined later }
                 Address    = Text;
                 END
               );
           Keywords = Text;
           { The journal's editor introduces the article
             with a short introduction, in French and
             in English }
           Introduction =
               BEGIN
               French_intr  = Paragraphs WITH
                              Language='Fran\347ais';
               English_intr = Paragraphs WITH
                              Language='English';
               END;
           Body = Sections; { Sections are defined later }
                 { Appendixes are only created on demand }
         ? Appendices =
               LIST OF (Appendix =
                        BEGIN
                        Appendix_Title    = Text;
                        Appendix_Contents = Paragraphs;
                        END
                       );
           END;      { End of the Article aggregate }

Sections = LIST [2..*] OF (
               Section = { At least 2 sections }
               BEGIN
               Section_title   = Text;
               Section_contents =
                 BEGIN
                 Paragraphs;
                 Sections; { Sections at a lower level }
                 END;
               END
               );

Paragraphs = LIST OF (Paragraph = CASE OF
                           Enumeration =
                               LIST [2..*] OF
                                   (Item = Paragraphs);
                           Isolated_formula = Formula;
                           LIST OF (UNIT);
```

```
                                END
                           );

ASSOC          { Associated elements definitions }

   Figure = BEGIN
            Figure_caption  = Text;
            Illustration   = NATURE;
            END;

   Biblio_citation = CASE OF
                        Ref_Article =
                           BEGIN
                           Authors_Bib   = Text;
                           Article_Title = Text;
                           Journal       = Text;
                           Page_Numbers  = Text;
                           Date          = Text;
                           END;
                        Ref_Livre =
                           BEGIN
                           Authors_Bib; { Defined above }
                           Book_Title   = Text;
                           Editor       = Text;
                           Date;        { Defined above }
                           END;
                       END;

   Note =  Paragraphs - (Ref_note);

UNITS       { Elements which can be used in objects }

   Ref_note    = REFERENCE (Note);
   Ref_biblio  = REFERENCE (Biblio_citation);
   Ref_figure  = REFERENCE (Figure);
   Ref_formula = REFERENCE (Isolated_formula);

EXPORT      { Skeleton elements }

   Title,
   Figure with Figure_caption,
   Section With Section_title;

END            { End of the structure schema }
```

This schema is very complete since it defines both paragraphs and bibliographic citations. These element types could just as well be defined in other structure schemas, as is the case with the `Formula` class. All sorts of other elements can be inserted into an article, since a paragraph can contain any type of unit. Similarly, figures can be any class of document or object that the user chooses.

Generally, an article doesn't contain appendices, but it is possible to add them on explicit request: this is the effect of the question mark before the word Appendices.

The Figure, Biblio_citation and Note elements are associated elements. Thus, they are only used in `REFERENCE` statements.

Various types of cross–references can be put in paragraphs. They can also be placed the objects which are part of the article, since the cross–references are defined as units (`UNITS`).

There is a single restriction to prevent the creation of Ref_note elements within notes.

It is worth noting that the S language permits the definition of recursive structures like sections: a section can contain other sections (which are thus at the next lower level of the document tree). Paragraphs are also recursive elements, since a paragraph can contain an enumeration in which each element (`Item`) is composed of paragraphs.

A class of objects: mathematical formulas

The example below defines the `Formula` class which is used in Article documents. This class represents mathematical formulas with a rather simple structure, but sufficient to produce a correct rendition on the screen or printer. To support more elaborate operations (formal or numeric calculations), a finer structure should be defined. This class doesn't use any other class and doesn't define any associated elements or units.

```
STRUCTURE Formula;
DEFPRES FormulaP;

ATTR
   String_type = Function_name, Variable_name;

STRUCT
   Formula      = Expression;
   Expression   = LIST OF (Construction);
   Construction = CASE OF
                  TEXT;         { Simple character string }
                  Index    = Expression;
                  Exponent = Expression;
                  Fraction =
                        BEGIN
                        Numerator   = Expression;
                        Denominator = Expression;
                        END;
                  Root =
                        BEGIN
                      ? Order = TEXT;
                        Root_Contents = Expression;
                        END;
                  Integral =
                        BEGIN
                        Integration_Symbol = SYMBOL;
                        Lower_Bound        = Expression;
                        Upper_Bound        = Expression;
                        END;
                  Triple =
                        BEGIN
                        Princ_Expression = Expression;
                        Lower_Expression = Expression;
```

```
                            Upper_Expression = Expression;
                            END;
                     Column = LIST [2..*] OF
                                 (Element = Expression);
                     Parentheses_Block =
                            BEGIN
                            Opening  = SYMBOL;
                            Contents = Expression;
                            Closing  = SYMBOL;
                            END;
                     END;       { End of Choice Constructor }
END                             { End of Structure Schema }
```

This schema defines a single global attribute which allows functions and variables to be distinguished. In the presentation schema, this attribute can be used to choose between roman (for functions) and italic characters (for variables).

A formula's structure is that of a mathematical expression, which is itself a sequence of mathematical constructions. A mathematical construction can be either a simple character string, an index, an exponent, a fraction, a root, etc. Each of these mathematical constructions has a sensible structure which generally includes one or more expressions, thus making the formula class's structure definition recursive.

In most cases, the roots which appear in the formulas are square roots and their order (2) is not specified. This is why the Order component is marked optional by a question mark. When explicitly requested, it is possible to add an order to a root, for example for cube roots (order = 3).

An integral is formed by an integration symbol, chosen by the user (simple integral, double, curvilinear, etc.), and two bounds. A more fine−grained schema would add components for the integrand and the integration variable. Similarly, the Block_Parentheses construction leaves the choice of opening and closing symbols to the user. They can be brackets, braces, parentheses, etc.

The P Language

Document presentation

Because of the model adopted for Thot, the presentation of documents is clearly separated from their structure and content. After having presented the logical structure of documents, we now detail the principles implemented for their presentation. The concept of *presentation* encompasses what is often called the page layout, the composition, or the document style. It is the set of operations which display the document on the screen or print it on paper. Like logical structure, document presentation is defined generically with the help of a language, called P.

Two levels of presentation

The link between structure and presentation is clear: the logical organization of a document is used to carry out its presentation, since the purpose of the presentation is to make evident the organization of the document. But the presentation is equally dependent on the device used to render the document. Certain presentation effects, notably changes of font or character set, cannot be performed on all printers or on all screens. This is why Thot uses a two−level approach, where the presentation is first described in abstract terms, without taking into account each particular device, and then the presentation is realized within the constraints of a given device.

Thus, presentation is only described as a function of the structure of the documents and the image that would be produced on an idealized device. For this reason, presentation descriptions do not refer to any device characteristics: they describe *abstract presentations* which can be concretized on different devices.

A presentation description also defines a *generic presentation*, since it describes the appearance of a class of documents or objects. This generic presentation must also be applied to document and object instances, each conforming to its generic logical structure, but with all the allowances that were called to mind above: missing elements, constructed elements with other logical structures, etc.

In order to preserve the homogeneity between documents and objects, presentation is described with a single set of tools which support the layout of a large document as well as the composition of objects like a graphical figure or mathematical formula. This unity of presentation description tools contrasts with the traditional approach, which focuses more on documents than objects and thus is based on the usual typographic conventions, such as the placement of margins, indentations, vertical spaces, line lengths, justification, font changes, etc.

Boxes

To assure the homogeneity of tools, all presentation in Thot, for documents as well as for the objects which they contain, is based on the notion of the *box*, such as was implemented in TEX.

Corresponding to each element of the document is a box, which is the rectangle enclosing the element on the display device (screen or sheet of paper); the outline of this rectangle is not visible, except when a `ShowBox` rule applies to the element. The sides of the box are parallel to the sides of the screen or the sheet of paper. By way of example, a box is associated with a character string, a line of text, a page, a paragraph, a title, a mathematical formula, or a table cell.

Whatever element it corresponds to, each box possesses four sides and four axes, which we designate as follows (see figure):

`Top`
: the upper side,

`Bottom`
: the lower side,

`Left`
: the left side,

`Right`
: the right side,

`VMiddle`
: the vertical axis passing through the center of the box,

`HMiddle`
: the horizontal axis passing through the center of the box,

`VRef`
: the vertical reference axis,

`HRef`
: the horizontal reference axis.

```
         Left    VRef   VMiddle         Right
                   :       :
     Top    -----------------------------
           |       :       :             |
           |       :       :             |
           |       :       :             |
           |       :       :             |
           |       :       :             |
HMiddle  ..|.............................|..
           |       :       :             |
           |       :       :             |
    HRef ..|.............................|..
           |       :       :             |
           |       :       :             |
   Bottom   -----------------------------
                   :       :
```

The sides and axes of boxes

The principal role of boxes is to set the extent and position of the images of the different elements of a document with respect to each other on the reproduction device. This is done by defining relations between the boxes of different elements which give relative extents and positions to these boxes.

There are three types of boxes:

- boxes corresponding to structural elements of the document,
- presentation boxes,
- page layout boxes.

Boxes corresponding to structural elements of the document are those which linked to each of the elements (base or structured) of the logical structure of the document. Such a box contains all the contents of the element to which it corresponds (there is an exception: see rules `VertOverflow` and `HorizOverflow`). These boxes form a tree–like structure, identical to that of the structural elements to which they correspond. This tree expresses the inclusion relationships between the boxes: a box includes all the boxes of its subtree. On the other hand, there are no predefined rules for the relative positions of the included boxes. If they are at the same level, they can overlap, be contiguous, or be disjoint. The rules expressed in the generic presentation specify their relative positions.

Presentation boxes represent elements which are not found in the logical structure of the document but which are added to meet the needs of presentation. These boxes are linked to the elements of the logical structure that are best suited to bringing them out. For example, they are used to add the character string ‘‘Summary:’’ before the summary in the presentation of a report or to represent the fraction bar in a formula, or also to make the title of a field in a form appear. These elements have no role in the logical structure of the document: the presence of a Summary element in the document does not require the creation of another structural object to hold the word ‘‘Summary’’. Similarly, if a Fraction element contains both a Numerator element and a Denominator element, the fraction bar has no purpose structurally. On the other hand, these elements of the presentation are important for the reader of the reproduced document or for the user of an editor. This is why they must appear in the document’s image. It is the generic presentation which specifies the presentation boxes to add by indicating their content (a base element for which the value is specified) and the position that they must take in the tree of boxes. During editing, these boxes cannot be modified by the user.

Page layout boxes are boxes created implicitly by the page layout rules. These rules indicate how the contents of a structured element must be broken into lines and pages. In contrast to presentation boxes, these line and page boxes do not depend on the logical structure of the document, but rather on the physical constraints of the output devices: character size, height and width of the window on the screen or of the sheet of paper.

Views and visibility

One of the operations that one might wish to perform on a document is to view it is different ways. For this reason, it is possible to define several *views* for the same document, or better yet, for all documents of the same class. A view is not a different presentation of the document, but rather a filter which only allows the display of certain parts of the document. For example, it might be desirable to see only the titles of chapters and sections in order to be able to move rapidly through the document. Such a view could be called a ‘‘table of contents’’. It might also be desirable to see only the mathematical formulas of a document in order to avoid being distracted by the non–mathematical aspects of the document. A ‘‘mathematics’’ view could provide this service.

Views, like presentation, are based on the generic logical structure. Each document class, and each generic presentation, can be provided with views which are particularly useful for that class or presentation. For each view, the *visibility* of elements is defined, indicated whether or not the elements must be presented to the user. The visibility is calculated as a function of the type of the elements or their hierarchical position in the structure of the document. Thus, for a table of contents, all the ''Chapter Title'' and ''Section Title'' elements are made visible. However, the hierarchical level could be used to make the section titles invisible below a certain threshold level. By varying this threshold, the granularity of the view can be varied. In the ''mathematics'' view, only Formula elements would be made visible, no matter what their hierarchical level.

Because views are especially useful for producing a synthetic image of the document, it is necessary to adapt the presentation of the elements to the view in which they appear. For example, it is inappropriate to have a page break before every chapter title in the table of contents. Thus, generic presentations take into account the possible views and permit each element type's presentation to vary according the view in which its image appears.

Views are also used, when editing documents, to display the associated elements. So, in addition to the primary view of the document, there can be a ''notes'' view and a ''figures'' view which contain, respectively, the associated elements of the Note and Figure types. In this way, it is possible to see simultaneously the text which refers to these elements and the elements themselves, even if they will be separated when printed.

Pages

Presentation schemas can be defined which display the document as a long scroll, without page breaks. This type of schema is particularly well–suited to the initial phase of work on a document, where jumps from page to page would hinder composing and reading the document on a screen. In this case, the associated elements (such as notes), which are normally displayed in the page footer, are presented in a separate window. But, once the document is written, it may be desirable to display the document on the screen in the same manner in which it will be printed. So, the presentation schema must define pages.

The P language permits the specification of the dimensions of pages as well as their composition. It is possible to generate running titles, page numbers, zones at the bottom of the page for notes, etc. The editor follows this model and inserts page break marks in the document which are used during printing, insuring that the pages on paper are the same as on the screen.

Once a document has been edited with a presentation schema defining pages, it contains page marks. But it is always possible to edit the document using a schema without pages. In this case, the page marks are simply ignored by the editor. They are considered again as soon as a schema with pages is used. Thus, the user is free to choose between schemas with and without pages.

Thot treats the page break, rather than the page itself, as a box. This page break box contains all the elements of one page's footer, a rule marking the edge of this page, and all the elements of the next page's header. The elements of the header and footer can be running titles, page number, associated elements (notes, for example), etc. All these elements, as well as their content and graphical appearance, are defined by the generic presentation.

Numbering

Many elements are numbered in documents: pages, chapters, sections, formulas, theorems, notes, figures, bibliographic references, exercises, examples, lemmas, etc. Because Thot has a notion of logical structure, all of these numbers (with the exception of pages) are redundant with information implicit in the logical structure of the document. Such numbers are simply a way to make the structure of the document more visible. So, they are part of the document's presentation and are calculated by the editor from the logical structure. The structure does not contain numbers as such; it only defines relative structural positions between elements, which serve as ordering relations on these elements.

If the structure schema defines the body of a document as a sequence of at least two chapters:

```
Body = LIST [2..*] OF Chapter;
```

the sequence defined by the list constructor is ordered and each chapter can be assigned a number based on its rank in the Body list. Therefore, all elements contained in lists a the structure of a document can be numbered, but they are not the only ones. The tree structure induced by the aggregate, list, and choice constructors (excluding references) defines a total order on the elements of the document's primary structure. So, it is possible to define a numbering which uses this order, filtering elements according to their type so that only certain element types are taken into account in the numbering. In this way, it possible to number all the theorems and lemmas of a chapter in the same sequence of numbers, even when they are not part of the same list constructor and appear at different levels of the document's tree. By changing the filter, they can be numbered separately: one sequence of numbers for theorems, another for the lemmas.

Associated elements pose a special problem, since they are not part of the document's primary structure, but are attached only by references, which violate the total order of the document. Then, these associated elements are frequently numbered, precisely because the number is an effective way to visualize the reference. In order to resolve this problem, Thot implicitly defines a list constructor for each type of associated element, gathering together (and ordering) these elements. Thus, the associated elements can be numbered by type.

Since they are calculated from the document's logical structure and only for the needs of the presentation, numbers are presentation elements, described by presentation boxes, just like the fraction bar or the word ``Summary''. Nevertheless, numbers differ from these other boxes because their content varies from instance to instance, even though they are of the same type, whereas all fraction bars are horizontal lines and the same word ‘‘Summary’’ appears at the head of every document’s summary.

Presentation parameters

The principal parameters which determine document presentation are the *positions* and *dimensions* of boxes, the *font*, the *style*, the *size*, the *underlining* and the *color* of their content. From these parameters, and some others of less importance, it is possible to represent the usual typographic parameters for the textual parts of the document. These same parameters can be used to describe the geometry of the non−textual elements, even though they are two−dimensional elements unlike the text, which is linear.

As we have already seen, the positions of the boxes always respect the rule of enclosure: a box in the tree encloses all the boxes of the next lower level which are attached to it. The positional parameters permit the specification of the position of each box in relation to the enclosing box or to its sibling boxes (boxes directly attached to the same enclosing box in the tree of boxes).

The presentation parameters also provide control over the dimensions of the boxes. The dimensions of a box can depend either on its content or on its context (its sibling boxes and the enclosing box). Each dimension (height or width) can be defined independently of the other.

Because of the position and dimension parameters, it is possible to do the same things that are normally done in typography by changing margins, line lengths, and vertical or horizontal skips. This approach can also align or center elements and groups of elements.

In contrast to the position and dimension parameters, the font, style, size, underlining, and color do not concern the box itself (the rectangle delimiting the element), but its content. These parameters indicate the typographic attributes which must be applied to the text contained in the box, and by extension, to all base elements.

For text, the font parameter is used to change the family of characters (Times, Helvetica, Courier, etc.); the style is used to obtain italic or roman, bold or light characters; the size determines the point size of the characters; underlining defines the type and thickness of the lines drawn above, below, or through the characters.

For graphics, the line style parameter can be either solid, dotted, or dashed; the line thickness parameter controls the width of the lines; the fill pattern parameter determines how closed geometric figures must be filled.

While some of the parameters which determine the appearance of a box’s contents make sense only for one content type (text or graphic), other parameters apply to all content types: these are the color parameters. These indicate the color of lines and the background color.

Presentation description language

A generic presentation defines the values of presentation parameters (or the way to calculate those values) for a generic structure, or more precisely, for all the element types and all the global and local attributes defined in that generic structure. This definition of the presentation parameters is made with the P language. A program written in this language, that is a generic presentation expressed in P, is call a *presentation schema*. This section describes the syntax and semantics of the language, using the same meta–language as was used for the definition of the S language.

Recall that it is possible to write many different presentation schemas for the same class of documents or objects. This allows users to choose for a document the graphical appearance which best suits their type of work or their personal taste.

The organization of a presentation schema

A presentation schema begins with the word `PRESENTATION` and ends with the word `END`. The word `PRESENTATION` is followed by the name of the generic structure to which the presentation will be applied. This name must be the same as that which follows the keyword `STRUCTURE` in the structure schema associated with the presentation schema.

After this declaration of the name of the structure, the following sections appear (in order):

- Declarations of
  - all views,
  - printed views,
  - counters,
  - presentation constants,
  - variables,
- default presentation rules,
- presentation box and page layout box definitions,
- presentation rules for structured elements,
- presentation rules for attributes,
- rules for transmitting values to attributes of included documents.

Each of these sections is introduced by a keyword which is followed by a sequence of declarations. Every section is optional.

```
SchemaPres ='PRESENTATION' ElemID ';'
          [ 'VIEWS' ViewSeq ]
          [ 'PRINT' PrintViewSeq ]
          [ 'COUNTERS' CounterSeq ]
          [ 'CONST' ConstSeq ]
          [ 'VAR' VarSeq ]
          [ 'DEFAULT' ViewRuleSeq ]
```

```
                [ 'BOXES' BoxSeq ]
                [ 'RULES' PresentSeq ]
                [ 'ATTRIBUTES' PresAttrSeq ]
                [ 'TRANSMIT' TransmitSeq ]
                  'END' .
ElemID         = NAME .
```

Views

Each of the possible views must be declared in the presentation schema. As has already been described, the presentation rules for an element type can vary according to the view in which the element appears. The name of the view is used to designate the view to which the presentation rules apply (see the `IN` instruction). The definition of the view's contents are dispersed throughout the presentation rules attached to the different element types and attributes. The `VIEWS` section is simply a sequence of view names separated by commas and terminated by a semi–colon.

One of the view names (and only one) can be followed by the keyword `EXPORT`. This keyword identifies the view which presents the members of the document class in skeleton form. The graphical appearance and the content of this view is defined just as with other views, but it is useless to specify presentation rules concerning this view for the elements which are not loaded in the skeleton form.

It is not necessary to declare any views; in this case there is a single unnamed view. If many views are declared, the first view listed is considered the principal view. The principal view is the one to which all rules that are not preceded by an indication of a view will apply (see the instruction `IN`).

The principal view is the the one which the editor presents on the screen when the user asks to create or edit a document. Thus, it makes sense to put the most frequently used view at the head of the list. But if the structure schema contains skeleton elements and is loaded in its skeleton form, the view whose name is followed by the keyword `EXPORT` will be opened and no other views can be opened.

```
                   'VIEWS' ViewSeq
ViewSeq          = ViewDeclaration
                     < ',' ViewDeclaration > ';' .
ViewDeclaration = ViewID [ 'EXPORT' ] .
ViewID           = NAME .
```

Example:

When editing a report, it might be useful have views of the table of contents and of the mathematical formulas, in addition to the principal view which shows the document in its entirety. To achieve this, a presentation schema for the Report class would have the following `VIEWS` section:

```
VIEWS
```

```
     Full_text, Table_of_contents, Formulas;
```

The contents of these views are specified in the presentation rules of the schema.

Print Views

When editing a document, each view is presented in a different window. In addition to the views specified by the `VIEWS` instruction, the user can display the associated elements with one window for each type of associated element.

When printing a document, it is possible to print any number of views, chosen from among all the views which the editor can display (views in the strict sense or associated elements). Print views, as well as the order in which they must be printed, are indicated by the `PRINT` instruction. It appears after the `VIEWS` instruction and is formed of the keyword `PRINT` followed by the ordered list of print view names. The print view names are separated by commas and followed by a semi−colon. A print view name is either a view name declared in the `VIEWS` instruction or the name of an associated element type (with an ‘‘s’’ added to the end). The associated element must have been declared in the `ASSOC` section of the structure schema.

```
                'PRINT' PrintViewSeq
PrintViewSeq = PrintView < ',' PrintView > ';' .
PrintView    = ViewID / ElemID .
```

If the `PRINT` instruction is absent, the printing program will print only the principal view (the first view specified by the `VIEWS` instruction or the single, unnamed view when there is no `VIEWS` instruction).

Example:

Consider a Report presentation using the view declarations from the preceding example. Suppose we want to print the full text and table of contents views, but not the Formulas view, which is only useful when editing. In addition, suppose that we also want to print the bibliographic citations, which are associated elements (of type `Citation`). A sensible printing order would be to print the full text then the bibliography and finally the table of contents. To obtain this result when printing, the presentation schema would say:

```
PRINT
     Full_text, Citations, Table_of_contents;
```

Counters

A presentation has a *counter* for each type of number in the presentation. All counters, and therefore all types of numbers, used in the schema must be declared after the `COUNTERS` keyword.

Each counter declaration is composed of a name identifying the counter followed by a colon and the counting function to be applied to the counter. The counter declaration ends with a semi–colon.

The counting function indicates how the counter values will be calculated. Three types of counting functions are available. The first type is used to count the elements of a list or aggregate: it assigns to the counter the rank of the element in the list or aggregate. More precisely, the function

```
RANK OF ElemID [ LevelAsc ] [ INIT AttrID ]
        [ 'REINIT' AttrID ]
```

indicates that when an element creates, by a creation rule (see the `Create` instructions), a presentation box containing the counter value, this value is the rank of the creating element, if it is of type `ElemID`, otherwise the rank of the first element of type `ElemID` which encloses the creating element in the logical structure of the document.

The type name can be preceded by a star in the special case where the structure schema defines an element of whose `ElemID` is the same as that of an inclusion without expansion or with partial expansion. To resolve this ambiguity, the `ElemID` alone refers to the type defined in the structure schema while the `ElemID` preceded by a star refers to the included type.

The type name `ElemID` can be followed by an integer. That number represents the relative level, among the ancestors of the creating element, of the element whose rank is asked. If that relative level n is unsigned, the n^{th} element of type `ElemID` encountered when travelling the logical structure from the root to the creating element is taken into account. If the relative level is negative, the logical structure is travelled in the other direction, from the creating element to the root.

The function can end with the keyword `INIT` followed by the name of a numeric attribute (and only a numeric attribute). Then, the rank of the first element of the list or aggregate is considered to be the value of this attribute, rather than the default value of 1, and the rank of the other elements is shifted accordingly. The attribute which determines the initial value is searched on the element itself and on its ancestors.

The function can end with the keyword `REINIT` followed by the name of a numeric attribute (and only a numeric attribute). Then, if an element to be counted has this attribute, the counter value for this element is the attribute value and the following elements are numbered starting from this value.

When the `RANK` function is written

```
RANK OF Page [ ViewID ] [ INIT AttrID ]
```

(`Page`is a keyword of the P language), the counter takes as its value the number of the page on which the element which creates the presentation box containing the number appears. This is done as if the pages of the document form a list for each view. The counter only takes into account the pages of the relevant view, that is the view displaying the presentation box whose contents take the value of the number. However, if the keyword `Page` is followed by the name of a view (between parentheses), it is the pages of that view that are taken into account. As in the preceding form, the `RANK` function applied to pages can end with the `INIT` keyword followed by the name of a numeric attribute which sets the value of the first page's number. This attribute must be a local attribute of the document itself, and not of one of its components.

The second counting function is used to count the occurrences of a certain element type in a specified context. The instruction

```
SET n ON Type1 ADD m ON Type2 [ INIT AttrID ]
```

says that when the document is traversed from beginning to end (in the order induced by the logical structure), the counter is assigned the value `n` each time an element of type `Type1` is encountered, no matter what the current value of the counter, and the value `m` is added to the current value of the counter each time an element of type `Type2` is encountered.

As with the `RANK` function, the type names can be preceded by a star to resolve the ambiguity of included elements.

If the function ends with the keyword `INIT` followed by the name of an attribute and if the document possesses this attribute, the value of this attribute is used in place of `n`. The attribute must be numeric. It is searched on the element itself and on its ancestors.

This function can also be used with the `Page` keyword in the place of `Type1` or `Type2`. In the first case, the counter is reinitialized on each page with the value `n`, while in the second case, it is incremented by `m` on each page. As with the preceding counting function, the word `Page` can be followed by a name between parentheses. In this case, the name specifies a view whose pages are taken into account.

The definition of a counter can contain several `SET` functions and several `ADD` functions, each with a different value. The total number of counting functions must not be greater than 6.

The third counting function is used to count the elements of a certain type encountered when travelling from the creating element to the root of the logical structure. The creating element is included if it is of that type. That function is written

```
RLEVEL OF Type
```

where `Type` represents the type of the elements to be counted.

The formal definition of counter declarations is:

```
                'COUNTERS' CounterSeq
CounterSeq   = Counter < Counter > .
Counter      = CounterID ':' CounterFunc ';' .
CounterID    = NAME .
CounterFunc  = 'RANK' 'OF' TypeOrPage [ SLevelAsc ]
                [ 'INIT' AttrID ] [ 'REINIT' AttrID ] /
```

```
                 SetFunction < SetFunction >
                 AddFunction < AddFunction >
                 [ 'INIT' AttrID ] /
                 'RLEVEL' 'OF' ElemID .
SLevelAsc     = [ '-' ] LevelAsc .
LevelAsc      =  NUMBER .
SetFunction   = 'SET' CounterValue 'ON' TypeOrPage .
AddFunction   = 'ADD' CounterValue 'ON' TypeOrPage .
TypeOrPage    = 'Page' [ '(' ViewID ')' ] /
                [ '*' ] ElemID .
CounterValue = NUMBER .
```

Example:

If the body of a chapter is defined as a sequence of sections in the structure schema:

```
Chapter_body = LIST OF (Section =
                          BEGIN
                          Section_Title = Text;
                          Section_Body  = Paragraphs;
                          END
                       );
```

the section counter is declared:

```
SectionCtr : RANK OF Section;
```

and the display of the section number before the section title is obtained by a `CreateBefore` rule attached the `Section_Title` type, which creates a presentation box whose content is the value of the `SectionCtr` counter (see the `Content` instruction).

In order to number the formulas separately within each chapter, the formula counter is declared:

```
FormulaCtr : SET 0 ON Chapter ADD 1 ON Formula;
```

and the display of the formula number in the right margin, alongside each formula, is obtained by a `CreateAfter` instruction attached to the `Formula` type, which creates a presentation box whose content is the value of the `FormulaCtr` counter.

To number the page chapter by chapter, with the first page of each chapter having the number 1, the counter definition would be

```
ChapterPageCtr : SET 0 ON Chapter ADD 1 ON Page;
```

If there is also a chapter counter

```
ChapterCtr : RANK OF Chapter;
```

the content of a presentation box created at the top of each page could be defined as:

```
Content : (VALUE(ChapterCtr, URoman) TEXT '-'
           VALUE(ChapterPageCtr, Arabic));
```

Thus, the presentation box contains the number of the chapter in upper–case roman numerals followed by a hyphen and the number of the page within the chapter in arabic numerals.

Example:

To count tables and figures together in a document of the chapter type, a counter could be defined using:

```
CommonCtr : SET 0 ON Chapter ADD 1 ON Table
            ADD 1 ON Figure;
```

Presentation constants

Presentation constants are used in the definition of the content of presentation boxes. This content is used in variable definitions and in the `Content` rule. The only presentation constants which can be used are character strings, mathematical symbols, graphical elements, and pictures, that is to say, base elements.

Constants can be defined directly in the variables or presentation boxes (`Content` rule) which use them. But it is only necessary them to declare once, in the constant declaration section, even though they are used in many variables or boxes. Thus, each declared constant has a name, which allows it to be designated whenever it is used, a type (one of the four base types) and a value (a character string or a single character for mathematical symbols and graphical elements).

The constant declarations appear after the keyword `CONST`. Each declaration is composed of the name of the constant, an equals sign, a keyword representing its type (`Text`, `Symbol`, `Graphics` or `Picture`) and the string representing its value. A semi–colon terminates each declaration.

In the case of a character string, the keyword `Text` can be followed by the name of an alphabet (for example, `Greek` or `Latin`) in which the constant's text should be expressed. If the alphabet name is absent, the Latin alphabet is used. When the alphabet name is present, only the first letter of the alphabet name is interpreted. Thus, the words `Greek` and `Grec` designate the same alphabet. In current versions of Thot, only the Greek and Latin alphabets are available.

```
             'CONST' ConstSeq
ConstSeq   = Const < Const > .
Const      = ConstID '=' ConstType ConstValue ';' .
ConstID    = NAME .
ConstType  ='Text' [ Alphabet ] / 'Symbol' /
             'Graphics' / 'Picture' .
ConstValue = STRING .
Alphabet   = NAME .
```

For character strings in the Latin alphabet (ISO Latin–1 character set), characters having codes higher than 127 (decimal) are represented by their code in octal.

In the case of a symbol or graphical element, the value only contains a single character, between apostrophes, which indicates the form of the element which must be drawn in the box whose content is the constant. The symbol or graphical element takes the dimensions of the box, which are determined by the `Height` and `Width` rules. See table of codes for the symbols and graphical elements.

Example:

The constants ''Summary:'' and fraction bar, which were described earlier, are declared:

```
CONST
     SummaryConst = Text 'Summary:';
     Bar          = Graphics 'h';
```

Variables

Variables permit the definition of computed content for presentation boxes. A variable is associated with a presentation box by a `Content` rule; but before being used in a `Content` rule, a variable can be defined in the `VAR` section. It is also possible to define a variable at the time of its use in a `Content` rule, as can be done with a constant.

A variable has a name and a value which is a character string resulting from the concatenation of the values of a sequence of functions. Each variable declaration is composed of the variable name followed by a colon and the sequence of functions which produces its value, separated by spaces. Each declaration is terminated by a semi–colon.

```
               'VAR' VarSeq
VarSeq       = Variable < Variable > .
Variable     = VarID ':' FunctionSeq ';' .
VarID        = NAME .
FunctionSeq = Function < Function > .
```

Several functions are available. The first two return, in the form of a character string, the current date. `DATE` returns the date in English, while `FDATE` returns the date in french.

Two other functions, `DocName` and `DirName`, return the document name and the directory where the document is stored.

Function `ElemName` returns the type of the element which created the presentation box whose contents are the variable.

Another function simply returns the value of a presentation constant. For any constant declared in the `CONST` section, it is sufficient to give the name of the constant. Otherwise, the type and value of the constant must be given, using the same form as in a constant declaration. If the constant is not of type text, (types `Symbol`, `Graphics` or `Picture`), it must be alone in the variable definition; only constants of type `Text` can be mixed with other functions.

It is also possible to obtain the value of an attribute, simply by mentioning the attribute's name.

The value of this function is the value of the attribute for the element which created the presentation box whose contents are the variable. If the creating element does not have the indicated attribute, the value is an empty string. In the case of a numeric attribute, the attribute is translated into a decimal number in arabic numerals. If another form is desired, the `VALUE` function must be used.

The last available function returns, as a character string, the value of a counter, an attribute or a page number. This value can be presented in different styles. The keyword `VALUE` is followed (between parentheses) by the name of the counter, the name of the attribute, or the keyword `PageNumber` and the desired style, the two parameters being separated by a comma. The style is a keyword which indicates whether the value should be presented in arabic numerals (`Arabic`), lower–case roman numerals (`LRoman`), or upper–case roman numerals (`URoman`), or by an upper–case letter (`Uppercase`) or lower–case letter (`Lowercase`).

For a page counter, the keyword `PageNumber` can be followed, between parentheses, by the name of the view from which to obtain the page number. By default, the first view declared in the `VIEWS` section is used. The value obtained is the number of the page on which is found the element that is using the variable in a `Content` rule.

For an ordinary counter, the name of the counter can be preceded by the keyword `MaxRangeVal` or `MinRangeVal`. These keywords mean that the value returned by the function is the maximum (minimum resp.) value taken by the counter in the whole document, not the value for the element concerned by the function.

```
Function      = 'DATE' / 'FDATE' /
                'DocName' / 'DirName' /
                'ElemName' / 'AttributeName' /
                 ConstID / ConstType ConstValue /
                 AttrID /
                'VALUE' '(' PageAttrCtr ','
                            CounterStyle ')' .
PageAttrCtr   = 'PageNumber' [ '(' ViewID ')' ] /
                 [ MinMax ] CounterID / AttrID .
CounterStyle  = 'Arabic' / 'LRoman' / 'URoman' /
                'Uppercase' / 'Lowercase' .
MinMax        = 'MaxRangeVal' / 'MinRangeVal' .
```

Example:

To make today's date appear at the top of the first page of a report, a `CREATE` rule associated with the Report_Title element type generates a presentation box whose content (specified by the `Content` rule of that presentation box) is the variable:

```
VAR
     Todays_date : TEXT 'Version of ' DATE;
```

To produce, before each section title, the chapter number (in upper–case roman numerals) followed by the section number (in arabic numerals), two counters must be defined:

```
COUNTERS
     ChapterCtr : RANK OF Chapter;
     SectionCtr : RANK OF Section;
```

and the Section_Title element must create a presentation box whose content is the variable

```
VAR
     SectionNum : VALUE (ChapterCtr, URoman) TEXT '-'
                  VALUE (SectionCtr, Arabic);
```

In order to make the page number on which each section begins appear in the table of contents view next to the section title, each Section_Title element must create a presentation box, visible only in the table of contents view, whose content is the variable:

```
VAR
     TitlePageNume :
            VALUE (PageNumber(Full_text), Arabic);
```

Default presentation rules

In order to avoid having to specify, for each element type defined in the structure schema, values for every one of the numerous presentation parameters, the presentation schema allows the definition of a set of default presentation rules. These rules apply to all the boxes of the elements defined in the structure schema and to the presentation boxes and page layout boxes defined in the presentation schema. Only rules which differ from these default need to be specified in other sections of the presentation schema.

For the primary view, the default rules can define every presentation parameter, but not the presentation functions or the linebreaking conditions (the `NoBreak1`, `NoBreak2`, and `Gather` rules).

In a presentation schema, the default presentation rules section is optional; in this case, the `DEFAULT` keyword is also absent and the following rules are considered to be the default rules:

```
Visibility:         Enclosing =;
VertRef:            * . Left;
HorizRef:           Enclosed . HRef;
Height:             Enclosed . Height;
Width:              Enclosed . Width;
VertPos:            Top = Previous . Bottom;
HorizPos:           Left = Enclosing . Left;
MarginTop:          0;
MarginRight:        0;
MarginBottom:       0;
```

```
MarginLeft:         0;
PaddingTop:         0;
PaddingRight:       0;
PaddingBottom:      0;
PaddingLeft:        0;
BorderTopWidth:     0;
BorderRightWidth:   0;
BorderBottomWidth:  0;
BorderLeftWidth:    0;
BorderTopColor:     Foreground;
BorderRightColor:   Foreground;
BorderBottomColor:  Foreground;
BorderLeftColor:    Foreground;
BorderTopStyle:     None;
BorderRightStyle:   None;
BorderBottomStyle:  None;
BorderLeftStyle:    None;
VertOverflow:       No;
HorizOverflow:      No;
Size:               Enclosing =;
Style:              Enclosing =;
Weight:             Enclosing =;
Font:               Enclosing =;
Underline:          Enclosing =;
Thickness:          Enclosing =;
Indent:             Enclosing =;
LineSpacing:        Enclosing =;
Adjust:             Enclosing =;
Justify:            Enclosing =;
Hyphenate:          Enclosing =;
PageBreak:          Yes;
LineBreak:          Yes;
InLine:             Yes;
Depth:              0;
LineStyle:          Enclosing =;
LineWeight:         Enclosing =;
FillPattern:        Enclosing =;
Background:         Enclosing =;
Foreground:         Enclosing =;
```

If other values are desired for the default rules, they must be defined explicitly in the default rules section. In fact, it is only necessary to define those default rules which differ from the ones above, since the rules above will be used whenever a rule is not explicitly named.

Default rules for views other than the primary view can also be specified. Otherwise, the default rules for the primary views are applied to the other views.

Default rules are expressed in the same way as explicit rules for document elements.

Presentation and page layout boxes

The presentation process uses elements which are not part of the logical structure of the document, such as pages (which are the page layout boxes) or alternatively, rules, numbers, or words introducing certain parts of the document, such as ‘‘Summary’’, ‘‘Appendices’’, ‘‘Bibliography’’, etc. (which are presentation boxes).

After the word BOXES, each presentation or page layout box is defined by its name and a sequence of presentation rules which indicate how they must be displayed. These rules are the same as those which define the boxes associated with element of the logical structure of the document, with a single exception, the `Content` rule which is used only to specify the content of presentation boxes. The content of boxes associated with elements of the document structure is defined in each document or object and thus is not specified in the presentation schema, which applies to all documents or objects of a class.

Among the rules which define a presentation box, certain ones can refer to another presentation box (for example, in their positional rules). If the designated box is defined after the box which designates it, a FORWARD instruction followed by the name of the designated box must appear before the designation.

```
          'BOXES' BoxSeq
BoxSeq = Box < Box > .
Box    ='FORWARD' BoxID ';' /
          BoxID ':' ViewRuleSeq .
BoxID  = NAME .
```

Presentation of structured elements

After the words RULES, the presentation schema gives the presentation rules that apply to the elements whose types are defined in the structure schema. Only those rules which differ from the default must be specified in the RULES section.

The rule definitions for each element type are composed of the name of the element type (as specified in the structure schema) followed by a colon and the set of rules specific to that type.

The type name can be preceded by a star in the special case where the structure schema defines an inclusion without expansion (or with partial expansion) of a type with the same name as an element of defined in the structure schema.

In the case where the element is a mark pair, but only in this case, the type name can be preceded by the keywords `First` or `Second`. These keywords indicate whether the rules that follow apply to the first or second mark of the pair.

```
             'RULES' PresentSeq
PresentSeq = Present < Present > .
Present    = [ '*' ] [ FirstSec ] ElemID ':'
             ViewRuleSeq .
FirstSec   = 'First' / 'Second' .
```

A presentation schema can define presentation rules for base elements, which are defined implicitly in the structure schemas. In the English version of the presentation schema compiler, the base type names are the same as in the S language, but they are terminated by the _UNIT suffix: `TEXT_UNIT`, `PICTURE_UNIT`, `SYMBOL_UNIT`, `GRAPHICS_UNIT`. The base type names are written in upper–case letters.

Logical attribute presentation

After the keyword `ATTRIBUTES`, all attributes which are to have some effect on the presentation of the element to which they are attached must be mentioned, along with the corresponding presentation rules. This is true for both global attributes (which can be attached to all element types) and local attributes (which can only be attached to certain element types).

Also mentioned in this section are attributes which imply an effect on elements in the subtree of the element to which they are attached. The presentation of these descendants can be modified as a function of the value of the attribute which they inherit, just as if it was attached to them directly.

The specification for each attribute includes the attribute's name, followed by an optional value specification and, after a colon, a set of rules. The set of rules must contain at least one rule.

When there is no value specification, the rules are applied to all elements which carry the attribute, no matter what their value. When the rules must only apply when the attribute has certain values, these values must be specified. Thus, the same attribute can appear in the `ATTRIBUTES` section several times, with each appearance having a different value specification. However, reference attributes never have a value specification and, as a result, can only appear once in the `ATTRIBUTES` section.

To specify that the presentation rules apply to some of the descendants of the element having the attribute, the name of the affected element type is given, between parentheses, after the attribute name. This way, the presentation rules for the attribute will be applied to the element having the attribute, if it is of the given type, and to all of its descendants of the given type. In the case where this type is a mark pair, but only in this case, the type name can be preceded by the keywords `First` or `Second`. These keywords indicate whether the rules that follow apply to the first or second mark of the pair. If the rule must apply to several different element types, the specification must be repeated for each element type.

The specification of values for which the presentation rules will be applied varies according to the type of the attribute:

numeric attribute

: If the rules are to apply for one value of the attribute, then the attribute name is followed by an equals sign and this value. If the rules are to apply for all values less than (or greater than) a threshold value, non–inclusive, the attribute name followed by a '<' sign (or a '>' sign, respectively) and the threshold value. If the rules must apply to a range of values, the attribute name is followed by the word '`IN`' and the two bounds of the range, enclosed in brackets and separated by two periods ('..'). In the case of ranges, the values of the bounds are included in the range.

The threshold value in the comparisons can be the value of an attribute attached to an ancestor element. In this case, the attribute name is given instead of a constant value.

It is also possible to write rules which apply only when a comparison between two different attributes of the element's ancestors is true. In this case, the first attribute name is followed by a comparison keyword and the name of the second attribute. The comparison keywords are `EQUAL` (simple equality), `LESS` (non–inclusive less than), and `GREATER` (non–inclusive greater than).

text attribute
: If the rules are to apply for one value of the attribute, then the attribute name is followed by an equals sign and this value.

reference attribute
: There is never a value specification; the rules apply no matter what element is designated by the attribute.

enumerated attribute
: If the rules are to apply for one value of the attribute, then the attribute name is followed by an equals sign and this value.

The order in which the rules associated with a numeric attribute are defined is important. When multiple sets of rules can be applied, the first set declared is the one used.

Rules for attributes have priority over both default rules and rules associated with element types. The attribute rules apply to the element to which the attribute is attached. It is the rules which apply to the surrounding elements (and especially to the descendants) which determine the effect of the attribute rules on the environment (and especially on the terminal elements of the structure).

```
                 'ATTRIBUTES' PresAttrSeq
PresAttrSeq   = PresAttr < PresAttr > .
PresAttr      = AttrID [ '(' [ FirstSec ] ElemID ')' ]
                 [ AttrRelation ] ':' ViewRuleSeq .
AttrID        = NAME .
AttrRelation ='=' AttrVal /
                 '>' [ '-' ] MinValue /
                 '<' [ '-' ] MaxValue /
                 'IN' '[' [ '-' ] LowerBound '..'
                 [ '-' ] UpperBound ']' /
                 'GREATER' AttrID /
                 'EQUAL' AttrID /
                 'LESS' AttrID .
AttrVal       = [ '-' ] EqualNum / EqualText /
                 AttrValue .
MinValue      = NUMBER .
MaxValue      = NUMBER .
LowerBound    = NUMBER .
UpperBound    = NUMBER.
EqualNum      = NUMBER .
```

```
EqualText     = STRING .
AttrValue     = NAME .
```

In presentation rules associated with a numeric attribute (and only in such rules), the attribute name can be used in place of a numeric value. In this case, the value of the attribute is used in the application of the rule. Thus, the attribute can represent a relation between the size of two boxes, the height and width of a box, the height of an area where page breaks are prohibited, the distance between two boxes, the position of the reference axis of a box, the interline spacing, the indentation of the first line, the visibility, the depth (z–order), or the character set.

The presentation rules associated with reference attributes, it is possible to use the element designated by the attribute as a reference box in a positional or extent rule. This element is represented in the position or extent rule by the keyword `Referred`.

Example:

In all structure schemas, there is a global Language attribute defined as follows:

```
ATTR
     Language = TEXT;
```

The following rules would make French text be displayed in roman characters and English text be displayed in italics:

```
ATTRIBUTES
     Language = 'French' :
                Style : Roman;
     Language = 'English' :
                Style : Italics;
```

Using these rules, when the user puts the Language attribute with the value 'English' on the summary of a document, every character string (terminal elements) contained in the summary are displayed in italics. See the `Style` rule.

Example:

A numeric attribute representing the importance of the part of the document to which it is attached can be defined:

```
ATTR
     Importance = INTEGER;
```

In the presentation schema, the importance of an element is reflected in the choice of character size, using the following rules.

```
ATTRIBUTES
     Importance < 2 :
              Size : 1;
     Importance IN [2..4] :
              Size : Importance;
     Importance = 10 :
              Size : 5;
     Importance > 4 :
              Size : 4;
```

Thus, the character size corresponds to the value of the Importance attribute; its value is

- the value of the Importance attribute when the value is between 2 and 4 (inclusive),
- 1, when the value of the Importance attribute is less than 2,
- 4, when the value of the Importance attribute is greater than 4,
- 5, when the value of the Importance attribute is 10.

The last case (value 5) must be defined before the case which handles all Importance values greater than 4, because the two rules are not disjoint and the first one defined will have priority. Otherwise, when the Importance attribute has value 10, the font size will be 4.

Example:

Suppose the structure defines a list element which can have an attribute defining the type of list (numbered or not):

```
STRUCT
     list (ATTR list_type = enumeration, dash)
          = LIST OF (list_item = TEXT);
```

Then, the presentation schema could use the attribute placed on the list element to put either a dash or a number before the each element of the list:

```
ATTRIBUTES
    list_type (list_item) = enumeration :
          CreateBefore (NumberBox);
    list_type (list_item) = dash :
          CreateBefore (DashBox);
```

Example:

Suppose that two attributes are defined in the structure schema. The first is a numeric global attribute called ‘‘version’’. The other is a local attribute defined on the root of the document called ‘‘Document_version’’:

```
STRUCTURE Document
ATTR
     version = INTEGER;
STRUCT
     Document (ATTR Document_version = INTEGER) =
         BEGIN
         SomeElement ;
         ...
         SomeOtherElement ;
         END ;
...
```

These attributes can be used in the presentation schema to place change bars in the margin next to elements whose version attribute has a value equal to the

Document_version attribute of the root and to place a star in margin of elements whose version attribute is less than the value of the root's Document_version attribute:

```
ATTRIBUTES
     version EQUAL Document_version :
         CreateBefore (ChangeBarBox) ;
     version LESS Document_version :
         CreateBefore (StarBox) ;
```

Value transmission rules

The last section of a presentation schema, which is optional, serves to defines the way in which a document transmits certain values to its sub–documents. A sub–document is an document included without expansion or with partial expansion. The primary document can transmit to its sub–documents the values of certain counters or the textual content of certain of its elements, as a function of their type.

The sub–documents receive these values in attributes which must be defined in their structure schema as local attributes of the root element. The types of these attributes must correspond to the type of the value which they receive: numeric attributes for receiving the value of a counter, textual attributes for receiving the content of an element.

In the structure schema of the primary document, there appears at the end, after the `TRANSMIT` keyword, a sequence of transmission rules. Each rule begins with the name of the counter to transmit or of the element type whose textual content will be transmitted. This name is followed by the keyword `To` and the name of the attribute of the sub–document to which the value is transmitted. The sub–document class is indicated between parentheses after the name of the attribute. The transmission rule ends with a semicolon.

```
TransmitSeq    =  Transmit < Transmit > .
Transmit       =  TypeOrCounter 'To' ExternAttr
                  '(' ElemID ')' ';' .
TypeOrCounter =  CounterID / ElemID .
ExternAttr    =  NAME .
```

Example:

Consider a Book document class which includes instances of the Chapter document class. These classes might have the following schemas:

```
STRUCTURE Book
STRUCT
   Book = BEGIN
          Title = Text;
          Body  = LIST OF (Chapter INCLUDED);
          END;
   ...

STRUCTURE Chapter
```

```
STRUCT
   Chapter (ATTR FirstPageNum = Integer;
                 ChapterNum = Integer;
                 CurrentTitle   = Text) =
         BEGIN
         ChapterTitle = Text;
         ...
         END;
   ...
```

Then the presentation schema for books could define chapter and page counters. The following transmission rules in the book presentation schema would transmit values for the three attributes defined at the root of each chapter sub–document.

```
PRESENTATION Book;
VIEWS
   Full_text;
COUNTERS
   ChapterCtr: Rank of Chapter;
   PageCtr: Rank of Page(Full_text);
...
TRANSMIT
   PageCtr TO FirstPageNum(Chapter);
   ChapterCtr TO ChapterNum(Chapter);
   Title TO CurrentTitle(Chapter);
END
```

Thus, each chapter included in a book can number its pages as a function of the number of pages preceding it in the book, can make the chapter's number appear before the number of each of its sections, or can place the title of the book at the top of each page.

Presentation rules

Whether defining the appearance of a presentation or page layout box, an element type, or an attribute value, the set of presentation rules that apply is always defined in the same way.

Normally, a set of presentation rules is placed between the keywords `BEGIN` and `END`, the keyword `END` being followed by a semicolon. The first section of this block defines the rules that apply to the primary view, if the default rules are not completely suitable. Next comes the rules which apply to specific other views, with a rule sequence for each view for which the default rules are not satisfactory. If the default rules are suitable for the non–primary views, there will not be any specific rules for these views. If there is only one rule which applies to all views then the keywords `BEGIN` and `END` need not appear.

For each view, it is only necessary to specify those rules which differ from the default rules for the view, so that for certain views (or even all views), there may be no specific rules.

The specific rules for a non–primary view are introduced by the `IN` keyword, followed by the view name. The rules for that view follow, delimited by the keywords `BEGIN` and `END`, or without these two keywords when there is only one rule.

Note: the view name which follows the `IN` keyword must not be the name of the primary view, since the rules for that view are found before the rules for the other views.

Within each block concerning a view, other blocks can appear, delimited by the same keywords `BEGIN` and `END`. Each of these blocks gathers the presentation rules that apply, for a given view, only when a given condition is satisfied. Each block is preceded by a condition introduced by the `IF` keyword. If such a conditional block contains only one rule, the keywords `BEGIN` and `END` can be omitted.

Although the syntax allows any presentation rule to appear in a conditional block, only creation rules are allowed after any condition; other rules are allowed only after conditions `Within` and ElemID. In addition, the following rules cannot be conditional: `PageBreak, LineBreak, Inline, Gather`.

For a given view, the rules that apply without any condition must appear before the first conditional block. If some rules apply only when none of the specified condition holds, they are grouped in a block preceded by the keyword `Otherwise`, and that block must appear after the last conditionnal block concerning the same view.

```
ViewRuleSeq  = 'BEGIN' < RulesAndCond > < ViewRules >
               'END' ';' /
               ViewRules / CondRules / Rule .
RulesAndCond = CondRules / Rule .
ViewRules    = 'IN' ViewID CondRuleSeq .
CondRuleSeq  = 'BEGIN' < RulesAndCond > 'END' ';' /
               CondRules / Rule .
CondRules    = CondRule < CondRule >
               [ 'Otherwise' RuleSeq ] .
CondRule     = 'IF' ConditionSeq RuleSeq .
RulesSeq     = 'BEGIN' Rule < Rule > 'END' ';' /
               Rule .
```

Example:

The following rules for a report's title make the title visible in the primary view and invisible in the table of contents and in the formula views (see the `Visibility` rule).

```
Title : BEGIN
        Visibility : 1;
        ...     {Other rules for the primary view}
        IN Table_of_contents
           Visibility : 0;
        IN Formulas
           Visibility : 0;
        END;
```

Conditions applying to presentation rules

Many conditions can be applied to presentation rules. Conditions allow certain presentation rules to apply only in certain cases. These conditions can be based on the structural position of the element. They can be based on whether the element has references, and what type of references, whether the element has attributes, whether the element is empty or not. They can also be based on the value of a counter.

It is possible to specify several conditions which must all be true for the rules to apply.

A set of conditions is specified by the `IF` keyword. This keyword is followed by the sequence of conditions, separated by the `AND` keyword. Each condition is specified by a keyword which defines the condition type. In some cases, the keyword is followed by other data, which specify the condition more precisely.

An elementary condition can be negative; it is then preceded by the `NOT` keyword.

When the presentation rule(s) controlled by the condition apply to a reference element or a reference attribute, an elementary condition can also apply to element referred by this reference. The `Target` keyword is used for that purpose. It must appear before the keyword defining the condition type.

```
CondRule       ='IF' ConditionSeq RuleSeq .
ConditionSeq   = Condition < 'AND' Condition > .
Condition      = [ 'NOT' ] [ 'Target' ] ConditionElem .
ConditionElem  ='First' / 'Last' /
                 [ 'Immediately' ] 'Within' [ NumParent ]
                                   ElemID [ ExtStruct ] /
                ElemID /
                'Referred' / 'FirstRef' / 'LastRef' /
                'ExternalRef' / 'InternalRef' / 'CopyRef' /
                'AnyAttributes' / 'FirstAttr' / 'LastAttr' /
                'UserPage' / 'StartPage' / 'ComputedPage' /
                'Empty' /
                '(' [ MinMax ] CounterName CounterCond ')' /
                 CondPage '(' CounterID ')' .
NumParent      = [ GreaterLess ] NParent .
GreaterLess    ='>' / '<' .
NParent        = NUMBER.
ExtStruct      ='(' ElemID ')' .
CounterCond    ='<' MaxCtrVal / '>' MinCtrVal /
                '=' EqCtrVal /
                'IN' '[' ['-'] MinCtrBound '.' '.'
                 ['-'] MaxCtrBound ']' .
PageCond       ='Even' / 'Odd' / 'One' .
MaxCtrVal      = NUMBER .
MinCtrVal      = NUMBER .
EqCtrVal       = NUMBER .
MaxCtrBound    = NUMBER .
MinCtrBound    = NUMBER .
```

Conditions based on the logical position of the element

The condition can be on the position of the element in the document's logical structure tree. It is possible to test whether the element is the first (`First`) or last (`Last`) among its siblings or if it is not the first (`NOT First`) or not the last (`NOT Last`). These conditions can be associated only with creation rules.

It is also possible to test if the element is contained in an element of a given type (`Within`) or if it is not (`NOT Within`). The type is indicated after the keyword `Within`. If that element type is defined in a structure schema which is not the one which corresponds to the presentation schema, the type name of this element must be followed, between parentheses, by the name of the structure schema which defines it.

If the keyword `Within` is preceded by `Immediately`, the condition is satisfied only if the *parent* element has the type indicated. If the word `Immediately` is missing, the condition is satisfied if any *ancestor* has the type indicated.

An integer *n* can appear between the keyword `Within` and the type. It specifies the number of ancestors of the indicated type that must be present for the condition to be satisfied. If the keyword `Immediately` is also present, the *n* immediate ancestors of the element must have the indicated type. The integer *n* must be positive or zero. It can be preceded by `<` or `>` to indicate a maximum or minimum number of ancestors. If these symbols are missing, the condition is satisfied only if it exists exactly *n* ancestors. When this number is missing, it is equivalent to > 0.

If the condition applies to presentation rules associated with an attribute, in the `ATTRIBUTES` section of the presentation schema, the condition can be simply an element name. Presentation rules are then executed only if the attribute is attached to an element of that type. The keyword `NOT` before the element name indicates that the presentation rules must be executed only if the element is not of the type indicated.

Conditions on references

References may be taken into account in conditions, which can be based on the fact that the element, or one of its ancestors, is designated by a at least one reference (`Referred`) or by none (`NOT Referred`).

If the element or attribute to which the condition is attached is a reference, the condition can be based on the fact that it acts as the first reference to the designated element (`FirstRef`), or as the last (`LastRef`), or as a reference to an element located in another document (`ExternalRef`) or in the same document (`InternalRef`).

The condition can also be based on the fact that the element is an inclusion. This is noted

(`CopyRef`).

Like all conditions, conditions on references can be inverted by the `NOT` keyword. These conditions can be associated only with creation rules.

Conditions on logical attributes

The condition can be based on the presence or absence of attributes associated with the element, no matter what the attributes or their values. The `AnyAttributes` keyword expresses this condition.

If the condition appears in the presentation rules of an attribute, the `FirstAttr` and `LastAttr` keywords can be used to indicate that the rules must only be applied if this attribute is the first attribute for the element or if it is the last (respectively). These conditions can also be inverted by the `NOT` keyword. These conditions can be associated only with creation rules.

It is also possible to apply certain presentation rules only when the element being processed or one of its ancestors has a certain attribute, perhaps with a certain value. This can be done in the `ATTRIBUTES` section.

Conditions on page breaks

The page break base type (and only this type) can use the following conditions: `ComputedPage`, `StartPage`, and `UserPage`. The `ComputedPage` condition indicates that the presentation rule(s) should apply if the page break was created automatically by Thot; the `StartPage` condition is true if the page break is generated before the element by the `Page` rule; and the `UserPage` condition applies if the page break was inserted by the user.

These conditions can be associated only with creation rules.

Conditions on the element's content

The condition can be based on whether or not the element is empty. An element which has no children or whose leaves are all empty is considered to be empty itself. This condition is expressed by the `Empty` keyword, optionally preceded by the `NOT` keyword. This condition can be associated only with creation rules.

Conditions on counters

Presentation rules can apply when the counter's value is one, is even or odd, is equal, greater than or less than a given value or falls in a range of values. This is particularly useful for creating header and footer boxes. These conditions can be associated only with creation rules.

To compare the value of a counter to a given value, a comparison is given between parentheses.

The comparison is composed of the counter name followed by an equals, greater than, or less than sign and the value to which the counter will be compared. A test for whether or not a counter's value falls in a range also appears within parentheses. In this case, the counter name is followed by the `IN` keyword and the range definition within brackets. The `Even`, `Odd` and `One` are used to test a counter's value and are followed by the counter name between parentheses.

The list of possible conditions on counters is:

`Even (Counter)`
: the box is created only if the counter has an even value.

`Odd (Counter)`
: the box is created only if the counter has an odd value.

`One (Counter)`
: the box is created only the counter's value is 1.

`NOT One (Counter)`
: the box is created, unless the counter's value is 1.

`(Counter < Value)`
: the box is created only if the counter's value is less than Value.

`(Counter > Value)`
: the box is created only if the counter's value is greater than Value.

`(Counter = Value)`
: the box is created only if the counter's value is equal to Value.

`NOT (Counter = Value)`
: the is created only if the counter's value is different than Value.

`(Counter IN [MinValue..MaxValue])`
: the box is created only if the counter's value falls in the range bounded by MinValue and MaxValue (inclusive).

`NOT (Counter IN [MinValue..MaxValue])`
: the box is created only if the value of the counter does not fall in the range bounded by MinValue and MaxValue (inclusive).

Note: the `NOT Even` and `NOT Odd` conditions are syntactically correct but can be expressed more simply by `Odd` and `Even`, respectively.

A presentation rule

A presentation rule defines either a presentation parameter or presentation function. The parameters are:

- the position of the vertical and horizontal reference axes of the box,
- the position of the box in relation to other boxes,
- the height or width of the box, with overflow exceptions,
- the margins, padding and borders of the box,
- the characteristics of the lines contained in the box: linespacing, indentation of the first line, justification, hyphenation,

- the conditions for breaking the box across pages,
- the characteristics of the characters contained in the box: size, font, style, underlining,
- the depth of the box among overlapping boxes (often called stacking order),
- the characteristics of graphic elements contained in the box: style and thickness of lines, fill pattern for closed objects,
- the colors in text, graphics, pictures, and symbols contained in the box are displayed or printed,
- for presentation boxes only, the contents of the box.

The presentation functions are:

- the creation of a presentation box
- the line−breaking or page−breaking style,
- the copying of another box,
- the display of the box background and border,
- the display of a background picture and its aspect.

For each box and each view, every presentation parameter is defined once and only once, either explicitly or by the default rules. In contrast, presentation functions are not obligatory and can appear many times for the same element. for example an element can create many presentation boxes. Another element may not use any presentation functions.

Each rule defining a presentation parameter begins with a keyword followed by a colon. The keyword indicates the parameter which is the subject of the rule. After the keyword and the colon, the remainder of the rule varies. All rules are terminated by a semicolon.

```
Rule      = PresParam ';' / PresFunc ';' .
PresParam ='VertRef' ':'        PositionHoriz /
           'HorizRef' ':'       PositionVert /
           'VertPos' ':'        VPos /
           'HorizPos' ':'       HPos /
           'Height' ':'         Dimension /
           'Width' ':'          Dimension /
           'VertOverflow' ':'   Boolean /
           'HorizOverflow' ':'  Boolean /
           'MarginTop' ':'      MarginWidth /
           'MarginRight' ':'    MarginWidth /
           'MarginBottom' ':'   MarginWidth /
           'MarginLeft' ':'     MarginWidth /
           'PaddingTop' ':'     PaddingWidth /
           'PaddingRight' ':'   PaddingWidth /
           'PaddingBottom' ':'  PaddingWidth /
           'PaddingLeft' ':'    PaddingWidth /
           'BorderTopWidth' ':' BorderWidth /
           'BorderRightWidth' ':' BorderWidth /
           'BorderBottomWidth' ':' BorderWidth /
           'BorderLeftWidth' ':' BorderWidth /
           'BorderTopColor' ':' BorderColor /
           'BorderRightColor' ':' BorderColor /
           'BorderBottomColor' ':' BorderColor /
           'BorderLeftColor' ':' BorderColor /
```

```
                'BorderTopStyle' ':' BorderStyle /
                'BorderRightStyle' ':' BorderStyle /
                'BorderBottomStyle' ':' BorderStyle /
                'BorderLeftStyle' ':' BorderStyle .
                'LineSpacing' ':'   DistanceInherit /
                'Indent' ':'        DistanceInherit /
                'Adjust' ':'        AdjustInherit /
                'Justify' ':'       BoolInherit /
                'Hyphenate' ':'     BoolInherit /
                'PageBreak' ':'     Boolean /
                'LineBreak' ':'     Boolean /
                'InLine' ':'        Boolean /
                'NoBreak1' ':'      AbsDist /
                'NoBreak2' ':'      AbsDist /
                'Gather' ':'        Boolean /
                'Visibility' ':'    NumberInherit /
                'Size'  ':'         SizeInherit /
                'Font' ':'          NameInherit /
                'Style' ':'         StyleInherit /
                'Weight' ':'        WeightInherit /
                'Underline' ':'     UnderLineInherit /
                'Thickness' ':'     ThicknessInherit /
                'Depth' ':'         NumberInherit /
                'LineStyle' ':'     LineStyleInherit /
                'LineWeight' ':'    DistanceInherit /
                'FillPattern' ':'   NameInherit /
                'Background' ':'    NameInherit /
                'Foreground' ':'    NameInherit /
                'Content' ':'       VarConst .
     PresFunc = Creation '(' BoxID ')' /
                'Line' /
                'NoLine' /
                'Page' '(' BoxID ')' /
                'Copy' '(' BoxTypeToCopy ')' /
                'ShowBox' /
                'BackgroundPicture' ':' FileName /
                'PictureMode' ':'   PictMode /
```

Box axes

The position of the middle axes `VMiddle` and `HMiddle` in relation to their box is always calculated automatically as a function of the height and width of the box and is not specified by the presentation rules. In the presentation schema, these middle axes are used only to position their box with respect to another by specifying the distance between the middle axis and an axis or a side of another box (see the relative position).

The reference axes of a box are also used to position their box in relation to another, but in contrast to the middle axes, the presentation schema must make their position explicit, either in relation to a side or the middle axis of the box itself, or in relation to an axis of an enclosed box.

Only boxes of base elements have predefined reference axes. For character string boxes, the horizontal reference axis is the baseline of the characters (the line which passes immediately under the upper–case letters, ignoring the letter Q) and the vertical reference axis is at the left edge of the first character of the string.

The positions of a box's reference axes are defined by the `VertRef` and `HorizRef` rules which specify the distance between the reference axis and an axis or parallel side of the same box or of an enclosed box.

```
'VertRef'  ':' PositionHoriz
'HorizRef' ':' PositionVert
```

Example:

If, in the structure schema for mathematical formulas, the fraction element is defined by

```
Fraction = BEGIN
            Numerator   = Expression;
            Denominator = Expression;
           END;
```

then the horizontal reference axis of the fraction can be positioned on top of the denominator by the rule:

```
Fraction :
     BEGIN
     HorizRef : Enclosed Denominator . Top;
     ...
     END;
```

To put the horizontal reference axis of a column at its middle:

```
Column :
     BEGIN
     HorizRef : * . HMiddle;
     ...
     END;
```

Distance units

Some distances and dimensions appear in many rules of a presentation schema, especially in position rules (`VertPos, HorizPos`), in extent rules for boxes (`Height, Width`), in rules defining lines (`LineSpacing, Indent`), in rules controlling pagination (`NoBreak1, NoBreak2`) and in rules specifying the thickness of strokes (`LineWeight`).

In all these rules, the distance or extent can be expressed

- either in relative units, which depend on the size of the characters in the current font: height of the element's font or height of the letter 'x',
- or in absolute units: centimeter, millimeter, inch, typographer's point, pica or pixel.

Units can be chosen freely. Thus, it is possible to use relative units in one rule, centimeters in the next rule, and typographer's points in another.

Absolute units are used to set rigid rules for the appearance of documents. In contrast, relative units allow changes of scale. The editor lets the value of relative units be changed dynamically. Such changes affect every box using relative units simultaneously and in the same proportion. Changing the value of the relative units affects the size of the characters and graphical elements, and the size of the boxes and the distances between them.

A distance or extent is specified by a number, which may be followed by one or more spaces and a units keyword. When there is no units keyword, the number specifies the number of relative units, where a relative unit is the height of a character in the current font (an em). When the number is followed by a units keyword, the keyword indicates the type of absolute units:

- `em`: height of the element's font,
- `ex`: height of the letter 'x',
- `cm`: centimeter,
- `mm`: millimeter,
- `in`: inch (1 in = 2.54 cm),
- `pt`: point (1 pt = 1/72 in),
- `pc`: pica (1 pc = 12 pt),
- `px`: pixel.

Whatever the chosen unit, relative or absolute, the number is not necessarily an integer and may be expressed in fixed point notation (using the American convention of a period to express the decimal point).

If the distance appears in a presentation rule for a numeric attribute, the number can be replaced by the name of an attribute. In this case, the value of the attribute is used. Obviously, the attribute name cannot be followed by a decimal point and a fractional part, but it can be followed a units keyword. However, the choice of units is limited to em, ex, pt and px.

```
Distance       = [ Sign ] AbsDist .
Sign           ='+' / '-' .
AbsDist        = IntegerOrAttr [ '.' DecimalPart ]
                 [ Unit ].
IntegerOrAttr  = IntegerPart / AttrID .
IntegerPart    = NUMBER .
DecimalPart    = NUMBER .
Unit           ='em' / 'ex' / 'cm' / 'mm' / 'in' / 'pt' /
                 'pc' / 'px' / '%' .
```

Example:

The following rules specify that a box has a height of 10.5 centimeters and a width of 5.3 ems:

```
Height : 10.5 cm;
Width  : 5.3;
```

Relative positions

The positioning of boxes uses the eight axes and sides, the sides generally being used to define the juxtapositioning (vertical or horizontal) of boxes, the middle axes being used to define centering, and the reference axes being used for alignment.

Two rules allow a box to be placed relative to other boxes. The `VertPos` rule positions the box vertically. The `HorizPos` rule positions the box horizontally. It is possible that a box's position could be entirely determined by other boxes positioned relative to it. In this case, the position is implicit and the word `nil` can be used to specify that no position rule is needed. Otherwise, an explicit rule must be given by indicating the axis or side which defines the position of the box, followed by an equals sign and the distance between between this axis or side and a parallel axis or side of another box, called the reference box. The box for which the rule is written will be positioned relative to the reference box.

```
                'VertPos' ':' VPos
                'HorizPos' ':' HPos
HPos        = 'nil' / VertAxis '=' HorizPosition
                [ 'UserSpecified' ].
VPos        = 'nil' / HorizAxis '=' VertPosition
                [ 'UserSpecified' ].
VertAxis  = 'Left' / 'VMiddle' / 'VRef' / 'Right' .
HorizAxis = 'Top' / 'HMiddle' / 'HRef' / 'Bottom' .
```

The reference box is an adjacent box: enclosing, enclosed or adjacent. When a rule is associated with a reference type attribute (and only in this case), it can be a box of the element designated by the attribute. The reference box can be either a presentation box previously defined in the `BOXES` section of the schema and created by a creation function, or the box associated with a structured element.

The structural position of the reference box (relative to the box for which the rule is being written) is indicated by a keyword: `Enclosing`, `Enclosed`, or, for sibling boxes, `Previous` or `Next`. The reference attributes, or presentation boxes created by a reference attribute, the `Referred` keyword may be used to designate the element which the reference points to. The keyword `Creator` can be used in rules for presentation boxes to designate the box of the element which created the presentation box. Finally, the `Root` keyword can be used to designate the root of the document.

When the keyword is ambiguous, it is followed by the name of an element type or presentation box which resolves the ambiguity (the `Creator` and `Root` keywords are never ambiguous). If this name is not given, then the first box encountered is used as the reference box. It is also possible to use just the name of an element type or presentation box without an initial keyword. In this case, a sibling having that name will be used. If the name is preceded by the keyword `NOT`, then the reference box will be the first box whose type is not the named one.

In place of the box or type name, the keywords `AnyElem`, `AnyBox` and `ElemWithAttr` can be used, representing respectively, any structured element box, any presentation box, and the box corresponding to any structured element having a particular attribute, whatever its value; the name of this attribute must follow the keyword `ElemWithAttr`.

A type name may be preceded by a star in order to resolve the ambiguity in the special case where the structure schema defines an inclusion without expansion (or with partial expansion) of the same type as an element of the scheme. For mark pairs (and only for mark pairs) the type name *must* be preceded by the `First` or `Second` keyword, which indicates which of the two marks of the pair should be used as the reference box.

The star character ('`*`') used alone designates the box to which the rule applies (in this case, it is obviously useless to specify the type of the reference box).

The keywords `Enclosing` and `Enclosed` can be used no matter what constructor defines the type to which the rule applies. When applied to the element which represents the entire document, `Enclosing` designates the window or page in which the document's image is displayed for the view to which the rule applies. A box or type name without a keyword is used for aggregate elements and designates another element of the same aggregate. It can also be used to designate a presentation or page layout box. The keywords `Previous` and `Next` are primarily used to denote list elements, but can also be used to denote elements of an aggregate.

In the position rule, the structural position relative to the reference box is followed, after a period, by the name of an axis or side. The rule specifies its node's position as being some distance from this axis or side of the reference box. If this distance is zero, then the distance does not appear in the rule. Otherwise, it does appear as a positive or negative number (the sign is required for negative numbers). The sign takes into account the orientation of the coordinate axes: for top to bottom for the vertical axis and from left to right for the horizontal axis. Thus, a negative distance in a vertical position indicates that the side or axis specified in the rule is above the side or axis of the reference box.

The distance can be followed by the `UserSpecified` keyword (even if the distance is nil and does not appear, the `UserSpecified` keyword can be used). It indicates that when the element to which the rule applies is being created, the editor will ask the user to specify the distance himself, using the mouse. In this case, the distance specified in the rule is a default distance which is suggested to the user but can be modified. The `UserSpecified` keyword can be used either in the vertical position rule, the horizontal position rule, or both.

```
VertPosition  = Reference '.' HorizAxis [ Distance ] .
HorizPosition = Reference '.' VertAxis [ Distance ] .
Reference     ='Enclosing' [ BoxTypeNot ] /
                'Enclosed' [ BoxTypeNot ] /
                'Previous' [ BoxTypeNot ] /
                'Next' [ BoxTypeNot ] /
                'Referred' [ BoxTypeNot ] /
                'Creator' /
                'Root' /
```

```
                 '*' /
                  BoxOrType .
BoxOrType      = BoxID /
                  [ '*' ] [ FirstSec ] ElemID /
                 'AnyElem' / 'AnyBox' /
                 'ElemWithAttr' AttrID .
BoxTypeNot     = [ 'NOT' ] BoxOrType .
```

Example:

If a report is defined by the following structure schema:

```
Report = BEGIN
         Title   = Text;
         Summary = Text;
         Keywords = Text;
         ...
         END;
```

then the presentation schema could contain the rules:

```
Report  : BEGIN
          VertPos   : Top = Enclosing . Top;
          HorizPos  : Left = Enclosing . Left;
          ...
          END;
```

These rules place the report in the upper left corner of the enclosing box, which is the window in which the document is being edited.

```
Title  :  BEGIN
          VertPos   : Top = Enclosing . Top + 1;
          HorizPos  : VMiddle = Enclosing . VMiddle;
          ...
          END;
```

The top of the title is one line (a line has the height of the characters of the title) from the top of the report, which is also the top of the editing window. The title is centered horizontally in the window (see figure).

```
Summary  : BEGIN
           VertPos   : Top = Title . Bottom + 1.5;
           HorizPos  : Left = Enclosing . Left + 2 cm;
           ...
           END;
```

The top of the summary is place a line and a half below the bottom of the title and is shifted two centimeters from the side of the window.

Example:

Suppose there is a Design logical structure which contains graphical elements:

```
Design = LIST OF (ElemGraph = GRAPHICS);
```

The following rules allow the user to freely choose the position of each element when it is created:

```
ElemGraph =
   BEGIN
   VertPos : Top = Enclosing . Top + 1 cm UserSpecified;
   HorizPos: Left = Enclosing . Left UserSpecified;
   ...
   END;
```

Thus, when a graphical element is created, its default placement is at the left of the window and 1 cm from the top, but the user can move it immediately, simply by moving the mouse.

Box extents

The extents (height and width) of each box are defined by the two rules `Height` and `Width`. There are three types of extents: fixed, relative, and elastic.

Fixed extents

A fixed dimension sets the height or width of the box independently of all other boxes. It is expressed in distance units. The extent can be followed by the `UserSpecified` keyword which indicates that when the element to which the rule applies is being created, the editor will ask the user to specify the extent himself, using the mouse. In this case, the extent specified in the rule is a default extent which is suggested to the user but can be modified. The `UserSpecified` keyword can be used either in the `Height` rule, the `Width` rule, or both.

A fixed extent rule can be ended by the `Min` keyword, which signifies that the indicated value is a minimum, and that, if the contents of the box require it, a larger extent is possible.

```
             'Height' ':' Dimension
             'Width' ':' Dimension
Dimension = AbsDist [ 'UserSpecified' ]  [ 'Min' ] /
             ...
```

Example:

Continuing with the previous example, it is possible to allow the user to choose the size of each graphical element as it is created:

```
ElemGraph : BEGIN
             Width :  2 cm UserSpecified;
             Height : 1 cm UserSpecified;
             ...
             END;
```

Thus, when a graphical element is create, it is drawn by default with a width of 2 cm and a height of 1 cm, but the user is free to resize it immediately with the mouse.

```
Summary :  BEGIN
           Height : 5 cm Min;
           ...
```

```
                END;
    Keywords  : BEGIN
                VertPos : Top = Summary . Bottom;
                ...
                END;
```

Relative extents

A relative extent determines the extent as a function of the extent of another box, just as a relative position places a box in relation to another. The reference box in an extent rule is designated using the same syntax as is used in a relative position rule. It is followed by a period and a `Height` or `Width` keyword, depending on the extent being referred to. Next comes the relation between the extent being defined and the extent of the reference box. This relation can be either a percentage or a difference.

A percentage is indicated by a star (the multiplication symbol) followed by the numeric percentage value (which may be greater than or less than 100) and the percent ('%') character. A difference is simply indicated by a signed difference.

If the rule appears in the presentation rules of a numeric attribute, the percentage value can be replaced by the name of the attribute. This attribute is then used as a percentage. The attribute can also be used as part of a difference.

Just as with a fixed extent, a relative extent rule can end with the `Min` keyword, which signifies that the extent is a minimum and that, if the contents of the box require it, a larger extent is possible.

A special case of relative extent rules is:

```
Height : Enclosed . Height;
```

or

```
Width  : Enclosed . Width;
```

which specifies that the box has a height (or width) such that it encloses all the boxes which it contains, excluding boxes having a rule `VertOverflow: Yes;` or `HorizOverflow: Yes;`.

Note: character strings (type `TEXT_UNIT`) generally must use the sum of the widths of the characters which compose them as their width, which is expressed by the rule:

```
TEXT_UNIT :
     Width : Enclosed . Width;
```

If this rule is not the default `Width` rule, it must be given explicitly in the `RULES` section which defines the presentation rules of the logical elements.

```
                   'Height' ':' Extent
                   'Width' ':' Extent
```

```
Extent       = Reference '.' HeightWidth [ Relation ]
               [ 'Min' ] / ...
HeightWidth ='Height' / 'Width' .
Relation    ='*' ExtentAttr '%' / Distance .
ExtentAttr  = ExtentVal / AttrID .
ExtentVal   = NUMBER .
```

Example:

Completing the above example, it is possible to specify that the report takes its width from the editing window and its height from the size of its contents (this can obviously be greater than that of the window):

```
Report :   BEGIN
           Width : Enclosing . Width;
           Height : Enclosed . Height;
           ...
           END;
```

Then, the following rules make the title occupy 60% of the width of the report (which is that of the window) and is broken into centered lines of this width (see the `Line` rule).

```
Title :    BEGIN
           Width : Enclosing . Width * 60%;
           Height : Enclosed . Height;
           Line;
           Adjust : VMiddle;
           ...
           END;
```

The summary occupy the entire width of the window, with the exception of a 2 cm margin reserved by the horizontal position rule:

```
Summary :  BEGIN
           Width : Enclosing . Width - 2 cm;
           Height : Enclosed . Height;
           ...
           END;
```

This set of rules, plus the position rules given above, produce the layout of boxes shown in the followingfigure.

```
-------------------------------------------------------------
| Window and Report           ^                             |
|                             | 1 line                      |
|                             v                             |
|           ------------------------------------            |
|           |                                  |            |
:    20%    :               Title              :    20%     :
:<--------->:                                  :<--------->:
:           :                60%               :            :
:           :<-------------------------------->:            :
|           |                                  |            |
|           ------------------------------------            |
|                             ^                             |
```

```
|                                 | 1.5 line                       |
|                                 |                                |
|                                 v                                |
|          -------------------------------------------------------|
|  2 cm   |                                                        |
|<------->|                   Summary                              |
:         :                                                        :
```

Box position and extent

Elastic extents

The last type of extent is the elastic extent. Either one or both extents can be elastic. A box has an elastic extent when two opposite sides are linked by distance constraints to two sides or axes of other boxes.

One of the sides of the elastic box is linked by a position rule (`VertPos` or `HorizPos`) to a neighboring box. The other side is link to another box by a `Height` or `Width` rule, which takes the same form as the position rule. For the elastic box itself, the notions of sides (left or right, top or bottom) are fuzzy, since the movement of either one of the two reference boxes can, for example, make the left side of the elastic box move to the right of its right side. This is not important. The only requirement is that the two sides of the elastic box used in the position and extent rule are opposite sides of the box.

```
         'Height' ':' Extent
         'Width' ':' Extent
Extent = HPos / VPos / ...
```

Example:

Suppose we want to draw an elastic arrow or line between the middle of the bottom side of box A and the upper left corner of box B. To do this, we would define a graphics box whose upper left corner coincides with the middle of the bottom side of A (a position rule) and whose lower right corner coincides with with the upper left corner of B (dimension rules):

```
LinkedBox :
   BEGIN
   VertPos  : Top = A .Bottom;
   HorizPos : Left = A . VMiddle;
   Height   : Bottom = B . Top;
   Width    : Right = B . Left;
   END;
```

Example:

The element SectionTitle creates a presentation box called SectionNum which contains the number of the section. Suppose we want to align the SectionNum and SectionTitle horizontally, have the SectionNum take its width from its contents (the section number), have the SectionTitle box begin 0.5 cm to the right of the

SectionNum box and end at the right edge of its enclosing box. This would make the SectionTitle box elastic, since its width is defined by the position of its left and right sides. The following rules produce this effect:

```
SectionNum :
   BEGIN
   HorizPos : Left = Enclosing . Left;
   Width : Enclosed . Width;
   ...
   END;

SectionTitle :
   BEGIN
   HorizPos : Left = SectionNum . Right + 0.5 cm;
   Width : Right = Enclosing . Right;
   ...
   END;
```

Overflow

A boxes corresponding to a structural element normally contain all boxes corresponding to the elements of its subtree. However, in some cases, it could be necessary to allow a box to jut out from its parent box. Two presentation rules indicate that such an overflow is allowed, one for horizontal overflow, one for vertical overflow.

Each of these rules is expressed by a keyword followed by a colon and the keyword `Yes` or `No`.

```
               'VertOverflow' ':' Boolean /
               'HorizOverflow' ':' Boolean .
     Boolean = 'Yes' / 'No' .
```

@@@@@ ******

Explain Margin, Padding and Border. Refer to CSS2.

```
                'MarginTop' ':' MarginWidth /
                'MarginRight' ':' MarginWidth /
                'MarginBottom' ':' MarginWidth /
                'MarginLeft' ':' MarginWidth /
                'PaddingTop' ':' PaddingWidth /
                'PaddingRight' ':' PaddingWidth /
                'PaddingBottom' ':' PaddingWidth /
                'PaddingLeft' ':' PaddingWidth /
                'BorderTopWidth' ':' BorderWidth /
                'BorderRightWidth' ':' BorderWidth /
                'BorderBottomWidth' ':' BorderWidth /
                'BorderLeftWidth' ':' BorderWidth /
                'BorderTopColor' ':' BorderColor /
                'BorderRightColor' ':' BorderColor /
                'BorderBottomColor' ':' BorderColor /
                'BorderLeftColor' ':' BorderColor /
                'BorderTopStyle' ':' BorderStyle /
                'BorderRightStyle' ':' BorderStyle /
                'BorderBottomStyle' ':' BorderStyle /
```

```
                 'BorderLeftStyle' ':' BorderStyle .

MarginWidth    = InheritParent / 'Auto' / Distance .
PaddingWidth   = InheritParent / Distance .
BorderWidth    = InheritParent / 'Thin' / 'Medium' / 'Thick' / Distance .
BorderColor    = InheritParent / 'Transparent' / 'Foreground' /
                 ColorName .
BorderStyle    = InheritParent /
                 'None' / 'Hidden' / 'Dotted' / 'Dashed' / 'Solid' /
                 'Double' / 'Groove' / 'Ridge' / 'Inset' / 'Outset' .
InheritParent = 'Enclosing' '=' / 'Creator' '=' .
ColorName      =  NAME .
```

@@@@@ ******

Inheritance

A presentation parameter can be defined by reference to the same parameter of another box in the tree of boxes. These structural links are expressed by kinship. The reference box can be that of the element immediately above in the structure (`Enclosing`), two levels above (`GrandFather`), immediately below (`Enclosed`) or immediately before (`Previous`). In the case of a presentation box, and only in that case, the reference box may be the element which created the presentation box (`Creator`).

Kinship is expressed in terms of the logical structure of the document and not in terms of the tree of boxes. The presentation box cannot transmit any of their parameters by inheritance; only structured element boxes can do so. As an example, consider an element B which follows an element A in the logical structure. The element B creates a presentation box P in front of itself, using the `CreateBefore` rule (see the creation rules). If element B's box inherits its character style using the `Previous` kinship operation, it gets its character style from A's box, not from P's box. Inheritance works differently for positions and extents, which can refer to presentation boxes.

The inherited parameter value can be the same as that of the reference box. This is indicated by an equals sign. However, for numeric parameters, a different value can be obtained by adding or subtracting a number from the reference box's parameter value. Addition is indicated by a plus sign before the number, while subtraction is specified with a minus sign. The value of a parameter can also be given a maximum (if the sign is a plus) or minimum (if the sign is a minus).

If the rule is being applied to a numeric attribute, the number to add or subtract can be replaced by the attribute name. The value of a maximum or minimum may also be replaced by an attribute name. In these cases, the value of the attribute is used.

```
Inheritance     = Kinship  InheritedValue .
Kinship         ='Enclosing' / 'GrandFather' / 'Enclosed' /
                 'Previous' / 'Creator' .
InheritedValue ='+' PosIntAttr [ 'Max' maximumA ] /
                 '-' NegIntAttr [ 'Min' minimumA ] /
                 '=' .
```

```
PosIntAttr      = PosInt / AttrID .
PosInt          = NUMBER .
NegIntAttr      = NegInt / AttrID .
NegInt          = NUMBER .
maximumA        = maximum / AttrID .
maximum         = NUMBER .
minimumA        = minimum / AttrID .
minimum         = NUMBER .
```

The parameters which can be obtained by inheritance are justification, hyphenation, interline spacing, character font (font family), font style, font size, visibility, indentation, underlining, alignment of text, stacking order of objects, the style and thickness of lines, fill pattern and the colors of lines and characters.

Line breaking

The `Line` rule specifies that the contents of the box should be broken into lines: the boxes included in the box to which this rule is attached are displayed one after the other, from left to right, with their horizontal reference axes aligned so that they form a series of lines. The length of these lines is equal to the width of the box to which the `Line` rule is attached.

When an included box overflows the current line, it is either carried forward to the next line, cur, or left the way it is. The `LineBreak` rule is used to allow or prevent the breaking of included boxes. If the included box is not breakable but is longer than the space remaining on the line, it is left as is. When a character string box is breakable, the line is broken between words or, if necessary, by hyphenating a word. When a compound box is breakable, the box is transparent in regard to line breaking. The boxes included in the compound box are treated just like included boxes which have the `LineBreak` rule. Thus, it is possible to traverse a complete subtree of boxes to line break the text leaves of a complex structure.

The relative position rules of the included boxes are ignored, since the boxes will be placed according to the line breaking rules.

The `Line` rule does not have a parameter. The characteristics of the lines that will be constructed are determined by the `LineSpacing`, `Indent`, `Adjust`, `Justify`, and `Hyphenate` rules. Moreover, the `Inline` rule permits the exclusion of certain elements from the line breaking process.

When the `Line` rule appears in the rules sequence of a non–primary view, it applies only to that view, but when the `Line` rule appears in the rules sequence of the primary view, it also applies to the other views by default, except for those views which explicitly invoke the `NoLine` rule. Thus, the `NoLine` rule can be used in a non–primary view to override the primary view's `Line` rule. The `NoLine` rule must not be used with the primary view because the absence of the `Line` rule has the same effect. Like the `Line` rule, the `NoLine` rule does not take any parameters.

```
               'Line'
               'NoLine'
```

Line spacing

The `LineSpacing` rule defines the line spacing to be used in the line breaking process. The line spacing is the distance between the baselines (horizontal reference axis) of the successive lines produced by the `Line` rule. The value of the line spacing can be specified as a constant or by inheritance. It is expressed in any of the available distance units.

Inheritance allows the value to be obtained from a relative in the structure tree, either without change (an equals sign appears after the inheritance keyword), with a positive difference (a plus sign), or a negative difference (a minus sign). When the rule uses a difference, the value of the difference follows the sign and is expressed as a distance.

```
                 'LineSpacing' ':' DistOrInherit
DistOrInherit =  Kinship InheritedDist / Distance .
InheritedDist = '=' / '+' AbsDist / '-' AbsDist .
```

When the line spacing value (or its difference from another element) is expressed in relative units, it changes with the size of the characters. Thus, when a larger font is chosen for a part of the document, the line spacing of that part expands proportionally. In contrast, when the line spacing value is expressed in absolute units (centimeters, inches, typographer's points), it is independent of the characters, which permits the maintenance of a consistent line spacing, whatever the character font. Either approach can be taken, depending on the desired effect.

First line indentation

The `Indent` rule is used to specify the indentation of the first line of the elements broken into lines by the `Line` function. The indentation determines how far the first line of the element is shifted with respect to the other lines of the same element. It can be specified as a constant or by inheritance. The constant value is a positive integer (shifted to the right; the sign is optional), a negative integer (shifted to the left) or zero (no shift). All available units can be used.

Indentation can be defined for any box, regardless of whether the box is line broken, and transmitted by inheritance to elements that are line broken. The size of the indentation is specified in the same manner as the line spacing.

```
               'Indent' ':' DistOrInherit
```

Alignment

The alignment style of the lines constructed during line breaking is defined by the `Adjust` rule. The alignment value can be a constant or inherited. A constant value is specified by a keyword:

- `Left`: at the left edge,

- `Right`: at the right edge,
- `VMiddle`: centered
- `LeftWithDots`: at the left edge with a dotted line filling out the last line up to the right edge of the line breaking box.

An inherited value can only be the same as that of the reference box and is specified by a kinship keyword followed by an equals sign.

```
                   'Adjust' ':' AlignOrInherit
AlignOrInherit = Kinship '=' / Alignment .
Alignment      = 'Left' / 'Right' / 'VMiddle' /
                   'LeftWithDots' .
```

Justification

The `Justify` rule indicates whether the lines contained in the box and produced by a `Line` rule should be extended horizontally to occupy the entire width of their enclosing box. The first and last lines are treated specially: the position of the beginning of the first line is fixed by the `Indent` rule and last line is not extended. The justification parameter defined by this rule takes a boolean value, which can be a constant or inherited. A constant boolean value is expressed by either the `Yes` or the `No` keyword. An inherited value can only be the same as that of the reference box and is specified by a kinship keyword followed by an equals sign.

```
               'Justify' ':' BoolInherit
BoolInherit = Boolean / Kinship '=' .
Boolean     ='Yes' / 'No' .
```

When the lines are justified, the alignment parameter specified in the `Adjust` rule has no influence, other than on the last line produced. This occurs because, when the other are extended to the limits of the box, the alignment style is no longer perceptible.

> **Example:**
>
> An important use of inheritance is to vary the characteristics of lines for an element type (for example, Paragraph) according to the enclosing environment (for example, Summary or Section), and thus obtain different line breaking styles for the same elements when they appear in different environments. The following rules specify that paragraphs inherit their alignment, justification, and line spacing:

```
Paragraph :
   BEGIN
   Justify : Enclosing = ;
   LineSpacing : Enclosing = ;
   Adjust : Enclosing =;
   Line;
```

```
   END;
```

If the alignment, justification, and line spacing of the Section and Summary elements is fixed:

```
Section :
   BEGIN
   Adjust : Left;
   Justify : Yes;
   LineSpacing : 1;
   END;
Summary :
   BEGIN
   Adjust : VMiddle;
   Justify : No;
   LineSpacing : 1.3;
   END;
```

then the paragraphs appearing in sections are justified with a simple line spacing while those appearing in summaries are centered and not justified and have a larger line spacing. These are nevertheless the very same type of paragraph defined in the logical structure schema.

Hyphenation

The `Hyphenate` rule indicates whether or not words should be broken by hyphenation at the end of lines. It affects the lines produced by the `Line` rule and contained in the box carrying the `Hyphenate` rule.

The hyphenation parameter takes a boolean value, which can be either constant or inherited. A constant boolean value is expressed by either the `Yes` or the `No` keyword. An inherited value can only be the same as that of the reference box and is specified by a kinship keyword followed by an equals sign.

```
                'Hyphenate' ':' BoolInherit
BoolInherit = Boolean / Kinship '=' .
Boolean     = 'Yes' / 'No' .
```

Avoiding line breaking

The `InLine` rule is used to specify that a box that would otherwise participate in line breaking asked for by the `Line` rule of an enclosing box, instead avoids the line breaking process and positions itself according to the `HorizPos` and `VertPos` rules that apply to it. When the `InLine` rule applies to a box which would not be line broken, it has no effect.

The rule is expressed by the `InLine` keyword followed by a colon and the keyword `Yes`, if the box should participate in line breaking, or the keyword `No`, if it should not. This is the only form possible: this rule cannot be inherited. Moreover, it can only appear in the rules of the primary view and applies to all views defined in the presentation schema.

```
               'InLine' ':' Boolean .
```

```
Boolean = 'Yes' / 'No' .
```

Example:

Suppose the structure schema defines a logical attribute called `New` which is used to identify the passages in a document which were recently modified. It would be nice to have the presentation schema make a bar appear in the left margin next to each passage having the `New` attribute. A new passage can be an entire element, such as a paragraph or section, or it can be some words in the middle of a paragraph. To produce the desired effect, the `New` attribute is given a creation rule which generates a `VerticalBar` presentation box.

When the `New` attribute is attached to a character string which is inside a line broken element (inside a paragraph, for example), the bar is one of the elements which participates in line breaking and it is placed normally in the current line, at the end of the character string which has the attribute. To avoid this, the `InLine` rule is used in the following way:

```
BOXES
  VerticalBar:
      BEGIN
      Content: Graphics 'l';
      HorizPos: Left = Root . Left;
      VertPos: Top = Creator . Top;
      Height: Bottom = Creator . Bottom;
      Width: 1 pt;
      InLine: No;
      ...
      END;
...
ATTRIBUTES
  Nouveau:
      BEGIN
      CreateAfter(VerticalBar);
      END;
```

Page breaking and line breaking conditions

Pages are constructed by the editor in accordance with the model specified by a `Page` rule. The page model describes only the composition of the pages but does not give any rules for breaking different element types across pages. Now, it is possible that certain elements must not be cut by page breaks, while others can be cut anywhere. The `PageBreak`, `NoBreak1`, and `NoBreak2` rules are used to specify the conditions under which each element type can be cut.

The `PageBreak` rule is used to indicate whether or not the box can be cut during the construction of pages. If cutting is authorized, the box can be cut, with one part appearing at the bottom of a page and the other part appearing at the top of the next page. The rule is formed by the `PageBreak` keyword followed by a colon and a constant boolean value (`Yes` or `No`). This is the only form possible: this rule cannot be inherited. Moreover, it can only appear in the rules of the primary view and applies to all views defined in the presentation schema.

Whether objects can be cut by line breaks can be controlled in a similar way using the `LineBreak` rule. This rule allows the specification of whether or not the box can be cut during the construction of lines. If cutting is authorized, the box can be cut, with one part appearing at the end of a line and the other part appearing at the beginning of the next line. The rule is formed by the `LineBreak` keyword followed by a colon and a constant boolean value (`Yes` or `No`). This is the only form possible: this rule cannot be inherited. Moreover, it can only appear in the rules of the primary view and applies to all views defined in the presentation schema.

```
               'PageBreak' ':' Boolean .
               'LineBreak' ':' Boolean .
     Boolean = 'Yes' / 'No' .
```

When a box can be cut by a page break, it is possible that a page break will fall an inappropriate spot, creating, for example, a widow or orphan, or separating the title of a section from the first paragraph of the section. The `NoBreak1` and `NoBreak2` rules are used to avoid this. They specify that the box of the element to which they apply cannot be cut within a certain zone at the top (`NoBreak1` rule) or at the bottom (`NoBreak2` rule). These two rules specify the height of the zones in which page breaks are prohibited.

The `NoBreak1` and `NoBreak2` rules give the height of the zone in which page breaking is prohibited. The height is given as a constant value using any of the available units, absolute or relative. The value may not be inherited.

```
                   'NoBreak1' ':' AbsDist .
                   'NoBreak2' ':' AbsDist .
```

Example:

The following rules prevent widows and orphans in a paragraph:

```
Paragraph :
   BEGIN
   NoBreak1 : 2;
   NoBreak2 : 2;
   END;
```

This rule prevents a section title from becoming separated from the first paragraph of the section by prohibiting page breaks at the beginning of the section rule:

```
Section :
   NoBreak1 : 1.5 cm;
```

Finally, this rule prevents a figure from being page broken in any way:

```
Figure :
   PageBreak : No;
```

The Thot editor constructs the document images displayed on the screen dynamically. As the user moves in the document or makes the document scroll in a window, the editor constructs the image to be displayed in little bits, filling the gaps which are produced in the course of these operations. It stops filling in the image when an element reaches the edge of the window in which the gap appears. If the appearance of the document is complex, it is possible that the image in incomplete, even though the edge of the window was reached. For example, an element might need to be presented to the side of the last element displayed, but its image was not constructed. The user will not know whether the element is really absent or if its image has simply not been constructed.

The `Gather` rule is used to remedy this problem. When the rule `Gather : Yes;` is associated with an element type, the image of such elements is constructed as a block by the editor: it is never split up.

The `Gather` rule may not appear in the default rules. Elements which do not have the `Gather` rule are considered susceptible to being split up during display. Thus, it is not necessary to use the `Gather : No;` form. This rule must be used prudently and only for those elements which truly need it. If used incorrectly, it can pointlessly increase the size of the image constructed by the editor and lead to excessive memory consumption by the editor.

Like the `PageBreak` and `LineBreak` rules, the `Gather` rule can only appear in rules of the primary view and applies to all views defined in the presentation schema.

```
                 'Gather' ':' Boolean .
```

Visibility

The visibility parameter is used to control which elements should or should not be displayed, based on context. An element can have different visibilities in different views. If an element's visibility is zero for a view, that element is not displayed in that view and does not occupy any space (its extents are zero).

Visibility takes non–negative integer values (positive or zero). If values greater than 1 are used, they allow the user to choose a degree of visibility and, thus, to see only those boxes whose visibility parameter exceeds a certain threshold. This gives the user control over the granularity of the displayed pictures.

The visibility parameter can be defined as a constant or by inheritance. If defined by inheritance, it cannot be based on the value of the next or previous box. Visibility can only be inherited from above.

If it is a numeric attribute's presentation rule, the visibility can be specified by the attribute's name, in which case the value of the attribute is used.

```
                 'Visibility' ':' NumberInherit
   NumberInherit = Integer / AttrID / Inheritance .
   Integer       = NUMBER .
```

Example:

Suppose that only `Formula` elements should be displayed in the `MathView` view. Then, the default rules should include:

```
DEFAULT
      IN MathView Visibility:0;
```

which makes all elements invisible in the `MathView` view. However, the `Formula` element also has a `Visibility` rule:

```
Formula :
      IN MathView Visibility:5;
```

which makes formulas, and only formulas, visible.

Character style parameters

Four parameters are used to determine which characters are used to display text. They are size, font, style, and underlining.

Character size

The size parameter has two effects. First, it is used to specify the actual size and distance units for boxes defined in relative units. Second, it defines the size of the characters contained in the box.

As a distance or length, the size can be expressed in abstract or absolute units. It can also be inherited. If it is not inherited, it is expressed simply as an integer followed by the `pt` keyword, which indicates that the size is expressed in typographer's points. The absence of the `pt` keyword indicates that it is in abstract units in which the value 1 represents the smallest size while the value 16 is the largest size. The relationship between these abstract sizes and the real character sizes is controlled by a table which can be modified statically or even dynamically during the execution of the Thot editor.

If the `Size` rule is a numeric attribute's presentation rule, the value of the size parameter can be specified by the attribute's name, in which case the value of the attribute is used.

Note: the only unit available for defining an absolute size is the typographer's point. Centimeters and inches may not be used.

If the size is inherited, the rule must specify the relative from which to inherit and any percentage or difference from that relative's value. A percentage is indicated by a `*` followed by the value of the percentage and a `%`. A difference can be expressed in either typographer's points or in abstract units and the maximum or minimum size can be specified, but without specifying the unit: it is the same as was specified for the difference.

In a numeric attribute's presentation rule, the difference in size or the percentage can be indicated by the attribute's name instead of the numeric value, which means that the attribute's value should be used as the difference or percentage. The attribute can also be used as the minimum or maximum size for a difference.

```
                   'Size' ':' SizeInherit
SizeInherit      = SizeAttr [ 'pt' ] /
                   Kinship InheritedSize .
InheritedSize    ='+' SizeAttr [ 'pt' ]
                   [ 'Max' MaxSizeAttr ] /
                  '-' SizeAttr [ 'pt' ]
                   [ 'Min' MinSizeAttr ] /
                  '*' PercentSizeAttr '%' /
                  '=' .
SizeAttr         = Size / AttrID .
Size             = NUMBER .
MaxSizeAttr      = MaxSize / AttrID .
MaxSize          = NUMBER .
MinSizeAttr      = MinSize / AttrID .
MinSize          = NUMBER .
PercentSizeAttr  = PercentSize / AttrID .
PercentSize      = NUMBER .
```

Example:

The rule

```
Size : Enclosing - 2 pt Min 7;
```

states that the character size is 2 points less than that of the enclosing box, but that it may not be less than 7 points, whatever the enclosing box's value.

The following rules make the text of a report be displayed with medium-sized characters (for example, size 5), while the title is displayed with larger characters and the summary is displayed with smaller characters:

```
Report :
     Size : 5;
Title :
     Size : Enclosing + 2;
Summary :
     Size : Enclosing - 1;
```

Thus, the character sizes in the entire document can be changed by changing the size parameter of the Report element, while preserving the relationships between the sizes of the different elements.

Font and character style

The `Font` rule determines the font family to be used to display the characters contained in the box, while the `Style` rule determines their style and the `Weight` rule determines their weight. Thot recognizes three character fonts (Times, Helvetica, and Courier), three styles: Roman, Italics, and Oblique, and two weights: Normal and Bold.

The font family, style and weight can be specified by a named constant or can be inherited. For the name of the font family only the first character is used.

Only identical inheritance is allowed: the box takes the same font, style or weight as the box from which it inherits. This is indicated by an equals sign after the kinship specification.

```
                'Font' ':'          NameInherit /
                'Style' ':'         StyleInherit /
                'Weight' ':'        WeightInherit /

NameInherit    = Kinship '=' / FontName .
FontName       = NAME .
StyleInherit   = Kinship '=' /
                 'Roman' / 'Italics' / 'Oblique' .
WeightInherit  = Kinship '=' /
                 'Normal' / 'Bold' .
```

Example:

To specify that the summary uses the font family of the rest of the document, but in the italic style, the following rules are used:

```
Summary :
   BEGIN
   Font : Enclosing =;
   Style : Italics;
   END;
```

Underlining

The `Underline` rule is used to specify if the characters contained in a box should have lines drawn on or near them. There are four underlining styles: `Underlined`, `Overlined`, `CrossedOut`, and `NoUnderline`. The `Thickness` rule specifies the thickness of the line, `Thin` or `Thick`.

As with font family and style, only identical inheritance is allowed: the box has the same underlining type as the box from which it inherits the value. This is indicated by an equals sign after the kinship specification.

```
                   'Underline' ':' UnderLineInherit /
                   'Thickness' ':' ThicknessInherit /

UnderLineInherit = Kinship '=' / 'NoUnderline' /
                   'Underlined' /
                   'Overlined' / 'CrossedOut' .
ThicknessInherit = Kinship '=' / 'Thick' / 'Thin' .
```

Stacking order

The `Depth` rule is used to define the stacking order of terminal boxes when multiple boxes at least partially overlap. This rule defines how the depth parameter, which is zero or a positive integer, is calculated. The depth parameter has a value for all boxes. For terminal boxes in the structure and for presentation boxes, the depth value is used during display and printing: the boxes with the lowest value overlap those with higher depths. For non–terminal boxes, the depth is not interpreted during display, but it is used to calculate the depth of terminal boxes by inheritance.

Like most other rules, the depth rule is defined in the default rules of each presentation schema. Thus, there is always a depth value, even when it is not necessary because there is no overlapping. To avoid useless operations, a zero value can be given to the depth parameter, which signifies that overlapping is never a problem.

The depth rule has the same form as the visibility rule. It can be defined by inheritance or by a constant numeric value. When the rule is attached to a numeric attribute, it can take the value of that attribute.

```
               'Depth' ':' NumberInherit
```

Example:

For a purely textual document, in which overlapping never poses a problem, a single default `Depth` rule in the presentation schema is sufficient:

```
DEFAULT
    Depth : 0;
    ...
```

To make the text of examples appear on a light blue background, a presentation box is defined:

```
BOXES
   BlueBG :
      BEGIN
      Content : Graphics 'R';
      Background : LightBlue3;
      FillPattern: backgroundcolor;
      Depth : 2;
      ...
      END;
```

and is created by the `Example` element, which has the rules:

```
RULES
   Example :
      BEGIN
      CreateFirst (BlueBG);
      Depth : 1;
      ...
      END;
```

In this way, the text of an example (if it inherits its depth from its ancestor) will be superimposed on a light blue background, and not the reverse).

Line style

The `LineStyle` rule determines the style of line which should be used to draw all the elements contained in the box and the box itself, if it has a `ShowBox` rule. The line style can be indicated by a name (`Solid`, `Dashed`, `Dotted`) or it can be inherited. All elements of the graphic base type are affected by this rule, but it can be attached to any box and transmitted by inheritance to the graphic elements. The border of elements having a `ShowBox` rule is drawn according to the line style specified by this rule.

Only identical inheritance is allowed: the box takes the same line style as the box from which it inherits. This is indicated by an equals sign after the kinship specification.

```
                 'LineStyle' ':' LineStyleInherit
LineStyleInherit = Kinship '=' /
                 'Solid' / 'Dashed' / 'Dotted' .
```

Example:

To specify that, in Figures, the graphical parts should be drawn in solid lines, the Figure element is given a rule using the `Solid` name:

```
Figure :
   LineStyle : Solid;
```

and the elements composing figures are given an inheritance rule:

```
   LineStyle : Enclosing =;
```

Line thickness

The `LineWeight` rule determines the thickness of the lines of all graphical elements which appear in the box, no matter what their line style. Line thickness can be specified by a constant value or by inheritance. A constant value is a positive number followed by an optional unit specification (which is absent when using relative units). All available distance units can be used. Line thickness is expressed in the same way as line spacing.

```
                 'LineWeight' ':' DistOrInherit
```

All elements of the graphic base type are affected by this rule, but it can be attached to any box and transmitted by inheritance to the graphic elements. The border of element having a `ShowBox` rule is also drawn according to the thickness specified by this rule.

Example:

To specify that, in Figures, the graphical parts should be drawn with lines 0.3 pt thick, the Figure element is given this rule:

```
Figure :
   LineWeight : 0.3 pt;
```

and the elements composing figures are given an inheritance rule:

```
LineWeight : Enclosing =;
```

Fill pattern

The `FillPattern` rule determines the pattern used to fill closed graphical elements (circles, rectangles, etc.) which appear in the box. This rule also specifies the pattern used to fill the box associated with elements having a `ShowBox` rule. This pattern can be indicated by a named constant or by inheritance. The named constant identifies one of the patterns available in Thot. The names of the available patterns are: nopattern, foregroundcolor, backgroundcolor, gray1, gray2, gray3, gray4, gray5, gray6, gray7, horiz1, horiz2, horiz3, vert1, vert2, vert3, left1, left2, left3, right1, right2, right3, square1, square2, square3, lozenge, brick, tile, sea, basket.

Like the other rules peculiar to graphics, `LineStyle` and `LineWeight`, only elements of the graphic base type are affected by the `FillPattern` rule, but the rule can be attached to any box and transmitted by inheritance to the graphic elements. As with the other rules specific to graphics, only identical inheritance is allowed.

The `FillPattern` rule can also be used to determine whether or not text characters, symbols and pictures should be colored. For these element types (text, symbols, and pictures), the only valid values are `nopattern`, `foregroundcolor`, and `backgroundcolor`. When `FillPattern` has the value `backgroundcolor`, text characters, symbols, and bitmaps are given the color specified by the `Background` rule which applies to these elements. When `FillPattern` has the value `foregroundcolor`, these same elements are given the color specified by the `Foreground` rule which applies to these elements. In all other case, text characters are not colored.

```
'FillPattern' ':' NameInherit
```

Example:

To specify that, in Figures, the closed graphical elements should be filled with a pattern resembling a brick wall, the Figure element is given this rule:

```
Figure :
   FillPattern : brick;
```

and the elements composing figures are given an inheritance rule:

```
FillPattern : Enclosing =;
```

Colors

The `Foreground` and `Background` rules determine the foreground and background colors of the base elements which appear in the box. They also control the color of boxes associated with elements having a `ShowBox` rule. These colors can be specified with a named constant or by inheritance. The named constants specify one of the available colors in Thot. The available color names can be found in the file `thot.color`.

The color rules affect the same way all base elements and elements having a `ShowBox` rule, no matter what their type (text, graphics, pictures, symbols). The color rules can be associated with any box and can be transmitted by inheritance to the base elements or the elements having a `ShowBox` rule. Like the preceding rules, only inheritance of the same value is allowed.

```
'Foreground' ':' NameInherit
'Background' ':' NameInherit
```

Note: text colors only appear for text elements whose fill pattern does not prevent the use of color.

> **Example:**
>
> To specify that, in Figures, everything must be drawn in blue on a background of yellow, the Figure element is given these rules:

```
Figure :
   BEGIN
   Foreground : Blue;
   Background : Yellow;
   Fillpattern : backgroundcolor;
   END;
```

> and the elements composing figures are given inheritance rules:

```
   Foreground : Enclosing =;
   Background : Enclosing =;
   FillPattern : Enclosing =;
```

Background color and border

Boxes associated with structural elements are normally not visible, but it is possible to draw their border and/or to paint their area when it is needed. This is achieved by associating the `ShowBox` rule with the concerned element. This rule has no parameter and no value. It is simply written `Showbox;`. It is not inherited nor transmitted to any other element. It applies only to the element with which it is associated.

```
'ShowBox'
```

When an element has a `ShowBox` rule, the border is drawn only if the `LineWeight` rule that applies to that element has a non–zero value (this value can be inherited). The color, style and thickness of the border are defined by the `Foreground`, `LineStyle`, and `LineWeight` rules that apply to the element.

When an element has a `ShowBox` rule, the background of this element is paint only if the value of the `FillPattern` rule that applies to that element is not `nopattern`. The pattern and color(s) of the background are defined by the `FillPattern`, `Background`, and `Foreground` rules that apply to the element.

Background pictures

The `BackgroundPicture` rule allows to display a picture as the background of an element. It has a single parameter, the file name of the picture. This is a string delimited by single quotes. If the first character in this string is '/', it is considered as an absolute path, otherwise the file is searched for along the schema directory path. This file may contain a picture in any format accepted by Thot (xbm, xpm, gif, jpeg, png, etc.)

The `BackgroundPicture` and `PictureMode` rules apply only to the element with which they are associated. They are not inherited nor transmitted to children elements.

The background picture has not always the same size as the element's box. There are diffrent ways to fill the element box with the picture. This is specified by the `PictureMode` rule, which should be associated to the same element. This rule may take one of the following values:

`NormalSize`
: The picture is centered in the box, and clipped if it is too large.

`Scale`
: The picture is zoomed to fit the box size.

`RepeatX`
: The picture is repeated horizontally to fit the box width.

`RepeatY`
: The picture is repeated vertically to fit the box height.

`RepeatXY`
: The picture is repeated both horizontally and vertically to fill the box.

If an element has a `BackgroundPicture` rule and no `PictureMode` rule, the `NormalSize` value is assumed.

```
               'BackgroundPicture' ':' FileName /
               'PictureMode' ':' PictMode .

FileName = STRING .
PictMode = 'NormalSize' / 'Scale' / 'RepeatXY' / 'RepeatX' / 'RepeatY' .
```

The `BackgroundPicture` and `PictureMode` rules apply only to the element with which they are associated. They are not inherited nor transmitted to children elements.

Presentation box content

The `Content` rule applies to presentation boxes. It indicates the content given to a box. This content is either a variable's value or a constant value. In the special case of header or footer boxes, the content can also be a structured element type.

If the content is a constant, it can be specified, as in a variable declaration, either by the name of a constant declared in the `CONST` section or by direct specification of the type and value of the box's content.

Similarly, if it is a variable, the name of a variable declared in `VAR` section can be given or the variable may be defined within parentheses. The content inside the parentheses has the same syntax as a variable declaration.

When the content is a structured element type, the name of the element type is given after the colon. In this case, the box's content is all elements of the named type which are designated by references which are part of the page on which the header or footer with this `Content` rule appears. Only associated elements can appear in a `Content` rule and the structure must provide references to these elements. Moreover, the box whose content they are must be a header or footer box generated by a page box of the primary view.

```
               'Content' ':' VarConst
     VarConst = ConstID / ConstType ConstValue /
                VarID / '(' FunctionSeq ')' /
                ElemID .
```

A presentation box can have only one `Content` rule, which means that the content of a presentation box cannot vary from view to view. However, such an effect can be achieved by creating several presentation boxes, each with different content and visible in different views.

The `Content` rule also applies to elements defined as references in the structure schema. In this case, the content defined by the rule must be a constant. It is this content which appears on the screen or paper to represent references of the type to which the rule applies. A reference can have a `Content` rule or a `Copy` rule for each view. If neither of these rules appears, the reference is displayed as `[*]`, which is equivalent to the rule:

```
Content: Text '[*]';
```

Example:

The content of the presentation box created to make the chapter number and section number appear before each section title can be defined by:

```
BOXES
     SectionNumBox :
          BEGIN
          Content : NumSection;
          ...
          END;
```

if the `NumSection` variable has been defined in the variable definition section of the presentation schema. Otherwise the `Content` would be written:

```
BOXES
     SectionNumBox :
          BEGIN
          Content : (VALUE (ChapterCtr, Roman) TEXT '.'
                     VALUE (SectionCtr, Arabic));
          ...
          END;
```

To specify that a page footer should contain all elements of the `Note` type are referred to in the page, the following rule is written:

```
BOXES
     NotesFooterBox :
          BEGIN
          Content : Note;
          ...
          END;
```

`Note` is defined as an associated element in the structure schema and NotesFooterBox is created by a page box of the primary view.

Presentation box creation

A creation rule specifies that a presentation box should be created when an element of the type to which the rule is attached appears in the document.

A keyword specifies the position, relative to the creating box, at which the created box will be placed in the structure:

`CreateFirst`
: specifies that the box should be created as the first box of the next lower level, before any already existing boxes, and only if the beginning of the creating element is visible;

`CreateLast`
: specifies that the box should be created as the last box of the next lower level, after any existing boxes, and only if the end of the creating element is visible;

`CreateBefore`
: specifies that the box should be created before the creating box, on the same level as the creating box, and only if the beginning of the creating element is visible;

`CreateAfter`
: specifies that the box should be created after the creating box, on the same level as the creating box, and only if the beginning of the creating element is visible;

`CreateEnclosing`
: specifies that the box should be created at the upper level relatively to the creating box, and that it must contain that creating box and all presentation boxes created by the same creating box.

This keyword can be followed by the `Repeated` keyword to indicate that the box must be created for each part of the creating element. These parts result from the division of the element by page breaks or column changes. If the `Repeated` keyword is missing, the box is only created for the first part of the creating element (`CreateFirst` and `CreateBefore` rules) or for the last part (`CreateLast` and `CreateAfter` rules).

The type of presentation to be created is specified at the end of the rule between parentheses.

Creation rules cannot appear in the default presentation rules. The boxes being created should have a `Content` rule which indicates their content.

Creation rules can only appear in the block of rules for the primary view; creation is provoked by a document element for all views. However, for each view, the presentation box is only created if the creating element is itself a box in the view. Moreover, the visibility parameter of the presentation box can be adjusted to control the creation of the box on a view−by−view basis.

```
                   Creation '(' BoxID ')'
Creation        = Create [ 'Repeated' ] .
Create          ='CreateFirst' / 'CreateLast' /
                  'CreateBefore' / 'CreateAfter' /
                  'CreateEnclosing' .
```

Example:

Let us define an object type, called Table, which is composed of a sequence of columns, all having the same fixed width, where the columns are separated by vertical lines. There is a line to the left of the first column and one to the right of the last. Each column has a variable number of cells, placed one on top of the other and separated by horizontal lines. There are no horizontal lines above the first cell or below the last cell. The text contained in each cell is broken into lines and these lines are centered horizontally in the cell. The logical structure of this object is defined by:

```
Table    = LIST OF (Column);
Column   = LIST OF (Cell = Text);
```

```
|                |                |                |
|  xx xxxx xxxx  |x xxxx xxx xxxxx|  x xxx x xxx   |
| xxx xxx xxxx x |   x xx x xxx   | xxxxx xxxx xx  |
|   xxxxx xxxx   |----------------|  xxx xxxxx x   |
| xxxxx xxx xxxx | xxxx xx xx xxx |     xx xx      |
| xxx xxxx x xxx |  xxxx x xxx x  |----------------|
|----------------| xxx xxxx xxxxx |  xxxxx xxxxx   |
| xxx xxx xxxxxx |----------------| xxx xxxx xxxx  |
|  xxxx xxxx xx  |  xxxx xx x xx  |  xxx xx x xx   |
|----------------| xxx xxxxx xxxx | xxxx xxxx xxx  |
| xxxxx xxx xxxx |  xxxx xx x xx  |   xxxxx xxx    |
|xxxx xx x xxxxxx| xxxx xx xxxxxx |  xxxxx xxxxx   |
```

The design of a table

The presentation of the table should resemble the design of the above figure. It is defined by the following presentation schema fragment:

```
BOXES
     VertLine : BEGIN
                Width : 0.3 cm;
                Height : Enclosing . Height;
                VertPos : Top = Enclosing . Top;
                HorizPos : Left = Previous . Right;
                Content : Graphics 'v';
                END;

     HorizLine: BEGIN
                Width : Enclosing . Width;
                Height : 0.3 cm;
```

```
                   VertPos : Top = Previous . Bottom;
                   HorizPos : Left = Enclosing . Left;
                   Content : Graphics 'h';
                   END;

RULES
     Column    : BEGIN
                   CreateBefore (VertLine);
                   IF LAST CreateAfter (VertLine);
                   Width : 2.8 cm;
                   Height : Enclosed . Height;
                   VertPos : Top = Enclosing . Top;
                   HorizPos : Left = Previous . Right;
                   END;

     Cell      : BEGIN
                   IF NOT FIRST CreateBefore (HorizLine);
                   Width : Enclosing . Width;
                   Height : Enclosed . Height;
                   VertPos : Top = Previous . Bottom;
                   HorizPos : Left = Enclosing . Left;
                   Line;
                   Adjust : VMiddle;
                   END;
```

It is useful to note that the horizontal position rule of the first vertical line will not be applied, since there is no preceding box. In this case, the box is simply placed on the left side of the enclosing box.

Page layout

The page models specified in the `Page` rule are defined by boxes declared in the `BOXES` section of the presentation schema. Pages are not described as frames which will be filled by the document's text, but as element are inserted in the flow of the document and which mark the page breaks. Each of these page break elements contains presentation boxes which represent the footer boxes of a page followed by header boxes of the next page. The page box itself is the simple line which separates two pages on the screen. Both the footer and header boxes placed themselves with respect to this page box, with the footer being placed above it and the header boxes being placed above it.

The boxes created by a page box are headers and footers and can only place themselves vertically with respect to the page box itself (which is in fact the separation between two pages). Besides, it is their vertical position rule which determines whether they are header or footer boxes. Header and footer boxes must have an explicit vertical position rule (they must not use the default rule).

Footer boxes must have an absolute height or inherit the height of their contents:

```
Height : Enclosed . Height;
```

A page box must have height and width rules and these two rules must be specified with constant values, expressed in centimeters, inches, or typographer's points. These two rules are interpreted in a special way for page boxes: they determine the width of the page and the vertical distance between two page separators, which is the height of the page and its header and footer together.

A page box should also have vertical and horizontal position rules and these two rules should specify the position on the sheet of paper of the rectangle enclosing the page's contents. These two rules must position the upper left corner of the enclosing rectangle in relation to the upper left corner of the sheet of paper, considered to be the enclosing element. In both rules, distances must be expressed in fixed units: centimeters (`cm`), inches (`in`), or typographer's points (`pt`). Thus, rules similar to the following should be found in the rules for a page box:

```
BOXES
   ThePage :
      BEGIN
      VertPos : Top = Enclosing . Top + 3 cm;
      HorizPos : Left = Enclosing . Left + 2.5 cm;
      Width : 16 cm;
      Height : 22.5 cm;
      END;
```

When a document must be page broken, the page models to be constructed are defined in the `BOXES` section of the presentation schema by declaring page boxes and header and footer boxes. Also, the `Page` rule is used to specify to which parts of the document and to which views each model should be applied.

The `Page` rule has only one parameter, given between parentheses after the `Page` keyword. This parameter is the name of the box which must serve as the model for page construction. When a `Page` rule is attached to an element type, each time such an element appears in a document, a page break takes place and the page model indicated in the rule is applied to all following pages, until reaching the next element which has a `Page` rule.

The `Page` rule applies to only one view; if it appears in the primary view's block of rules, a `Page` rule applies only to that view. Thus, different page models can be defined for the full document and for its table of contents, which is another view of the same document. Some views can be specified with pages, and other views of the same document can be specified without pages.

```
                   'Page' '(' BoxID ')'
```

Box copies

The `Copy` rule can be used for an element which is defined as a reference in the structure schema. In this case, the rule specifies, between parenthesis, the name of the box (declared in the `BOXES` section) which must be produced when this reference appears in the structure of a document. The box produced is a copy (same contents, but possible different presentation) of the box type indicated by the parameter between parentheses, and which is in the element designated by the reference. The name of a box can be replaced by type name. Then what is copied is the contents of the element of this type which is inside the referenced element.

Whether a box name or type name is given, it may be followed by the name of a structure schema between parentheses. This signifies that the box or type is defined in the indicated structure schema and not in the structure schema with which the rule's presentation schema is associated.

The `Copy` rule can also be applied to a presentation box. If the presentation box was created by a reference attribute, the rule is applied as in the case of a reference element: the contents of the box having the `Copy` rule are based on the element designated by the reference attribute. For other presentation boxes, the `Copy` rule takes a type name parameter which can be followed, between parentheses, by the name of the structure schema in which the type is defined, if it is not defined in the same schema. The contents of the box which has this rule are a copy of the element of this type which is in the element creating the presentation box, or by default, the box of this type which precedes the presentation box. This last facility is used, for example, to define the running titles in headers or footers.

```
                 'Copy' '(' BoxTypeToCopy ')' .
BoxTypeToCopy = BoxID [ ExtStruct ] /
                 ElemID [ ExtStruct ] .
ExtStruct      = '(' ElemID ')' .
```

Like the creation rules, the `Copy` rule cannot appear in the default presentation rules. Moreover, this rule can only appear in the primary view's block of rules; the copy rule is applied to all views.

Example:

If the following definitions are in the structure schema:

```
Body = LIST OF (Chapter =
                     BEGIN
                     ChapterTitle = Text;
                     ChapterBody = SectionSeq;
                     END);
RefChapter = REFERENCE (Chapter);
```

then the following presentation rules (among many other rules in the presentation schema) can be specified:

```
COUNTERS
   ChapterCtr : RANK OF Chapter;
BOXES
   ChapterNumber :
      BEGIN
      Content : (VALUE (ChapterCtr, URoman));
      ...
      END;
```

```
RULES
   Chapter :
      BEGIN
      CreateFirst (ChapterNumber);
      ...
      END;
   RefChapter :
      BEGIN
      Copy (ChapterNumber);
      ...
      END;
```

which makes the number of the chapter designated by the reference appear in uppercase roman numerals, in place of the reference to a chapter itself. Alternatively, the chapter title can be made to appear in place of the reference by writing this `Copy`rule:

```
      Copy (ChapterTitle);
```

To define a header box, named `RunningTitle`, which contains the title of the current chapter, the box's contents are defined in this way:

```
BOXES
   RunningTitle :
      Copy (ChapterTitle);
```

The T language

Document translation

Because of its document model, Thot can produce documents in a high–level abstract form. This form, called the *canonical form* is specific to Thot; it is well suited to the editor's manipulations, but it does not necessarily suit other operations which might be applied to documents. Because of this, the Thot editor offers the choice of saving documents in its own form (the canonical form) or a format defined by the user. In the latter case, the Thot document is transformed by the translation program. This facility can also be used to export documents from Thot to systems using other formalisms.

Translation principles

Document translation allows the export of documents to other systems which do not accept Thot's canonical form. Translation can be used to export document to source–based formatters like TEX, LATEX, and `troff`. It can also be used to translate documents into interchange formats like SGML or HTML. To allow the widest range of possible exports, Thot does not limit the choice of translations, but rather allows the user to define the formalisms into which documents can be translated.

For each document or object class, a set of translation rules can be defined, specifying how the canonical form should be transformed into a given formalism. These translation rules are grouped into *translation schemas*, each schema containing the rules necessary to translate a generic logical structure (document or object structure) into a particular formalism. The same generic logical structure can have several different translation schemas, each defining translation rules for a different formalism.

Like presentation schemas, translation schemas are generic. Thus, they apply to an entire object or document class and permit translation of all documents or objects of that class.

Translation procedure

The translator works on the specific logical structure of the document being translated. It traverses the primary tree of this logical structure in pre–order and, at each node encountered, it applies the corresponding translation rules defined in the translation schema. Translation can be associated:

- with element types defined in the structure schema,
- with global or local attributes defined in the structure schema,
- with specific presentation rules,
- with the content of the leaves of the structure (characters, symbols and graphical elements)

Thus, for each node, the translator applies all rules associated with the element type, all rules associated with each attribute (local or global) carried by the element, and if the element is a leaf of the tree, it also applies translation rules for characters, symbols, or graphical elements, depending on the type of the leaf.

Rules associated with the content of leaves are different from all other rules: they specify only how to translate character strings, symbols, and graphical elements. All other rules, whether associated with element types, with specific presentation rules or with attributes, are treated similarly. These rules primarily allow:

- generation of a text constant or variable before or after the contents of an element,
- modification of the order in which elements appear after translation,
- removal of an element in the translated document,
- and writing messages on the user's terminal during translation.

Translation definition language

Translation schemas are written in a custom language, called T, which is described in the rest of this chapter. The grammar of T is specified using the same meta–language as was used for the S and P languages and the translation schemas are written using the same conventions as the structure and presentation schemas. In particular, the keywords of the T language (the stings between apostrophes in the following syntax rules) can be written in any combination of upper–case and lower–case letters, but identifiers created by the programmer must always be written in the same way.

Organization of a translation schema

A translation schema is begun by the `TRANSLATION` keyword and is terminated by the `END` keyword. The `TRANSLATION` keyword is followed by the name of the generic structure for which a translation is being defined and a semicolon. This name must be identical to the name which appears after the `STRUCTURE` keyword in the corresponding structure schema.

After this declaration of the structure, the following material appears in order:

- the length of lines produced by the translation,
- the character delimiting the end of the line,
- the character string which the translator will insert if it must line–break the translated text,
- declarations of
  - buffers,
  - counters,
  - constants,
  - variables,
- translation rules associated with element types,
- translation rules associated with attributes,
- translation rules associated with specific presentation rules,
- translation rules associated with characters strings, symbols and graphical elements.

Each of these sections is introduced by a keyword followed by a sequence of declarations. All of these sections are optional, expect for the translation rules associated with element types. Many `TEXTTRANSLATE` sections can appear, each defining the rules for translating character strings of a particular alphabet.

```
TransSchema ='TRANSLATION' ElemID ';'
            [ 'LINELENGTH' LineLength ';' ]
            [ 'LINEEND' CHARACTER ';' ]
            [ 'LINEENDINSERT' STRING ';' ]
            [ 'BUFFERS' BufferSeq ]
            [ 'COUNTERS' CounterSeq ]
            [ 'CONST' ConstSeq ]
            [ 'VAR' VariableSeq ]
              'RULES' ElemSeq
            [ 'ATTRIBUTES' AttrSeq ]
            [ 'PRESENTATION' PresSeq ]
            < 'TEXTTRANSLATE' TextTransSeq >
            [ 'SYMBTRANSLATE' TransSeq ]
            [ 'GRAPHTRANSLATE' TransSeq ]
```

```
                 'END' .
```

Line length

If a `LINELENGTH` instruction is present after the structure declaration, the translator divides the text it produces into lines, each line having a length less than or equal to the integer which follows the `LINELENGTH` keyword. This maximum line length is expressed as a number of characters. The end of the line is marked by the character defined by the `LINEEND` instruction. When the translator breaks the lines on a space character in generated text, this space will be replaced by the character string defined by the `LINEENDINSERT` instruction.

If the `LINEEND` instruction is not defined then the linefeed character (octal code 12) is used as the default line end character. If the `LINEENDINSERT` instruction is not defined, the linefeed character is inserted at the end of the produced lines. If there is no `LINELENGTH` instruction, the translated text is not divided into lines. Otherwise, if the translation rules generate line end marks, these marks remain in the translated text, but the length of the lines is not controlled by the translator.

```
LineLength = NUMBER .
```

Example:

To limit the lines produced by the translator to a length of 80 characters, the following rule is written at the beginning of the translation schema.

```
LineLength 80;
```

Buffers

A buffer is a unit of memory managed by the translator, which can either contain text read from the terminal during the translation (see the `Read` rule), or the name of the last picture (bit–map) encountered by the translator in its traversal of the document. Remember the pictures are stored in files that are separate for the document files and that the canonical form contains only the names of the files in which the pictures are found.

Thus, there are two types of buffers: buffers for reading from the terminal (filled by the `Read` rule) and the buffer of picture names (containing the name of the last picture encountered). A translation schema can use either type, one or several read buffers and one (and only one) picture name buffer.

If any buffers are used, the `BUFFERS` keyword must be present, followed by declarations of every buffer used in the translation schema. Each buffer declaration is composed only of the name of the buffer, chosen freely by the programmer. The picture name buffer is identified by the `Picture` keyword, between parentheses, following the buffer name. The `Picture` keyword may only appear once. Each buffer declaration is terminated by a semicolon.

```
BufferSeq = Buffer < Buffer > .
```

```
Buffer     = BufferID [ '(' 'Picture' ')' ] ';' .
BufferID   = NAME .
```

Example:

The following buffer declarations create a picture name buffer named `pictureName` and a read buffer named `DestName`:

```
BUFFERS
     pictureName (Picture); DestName;
```

Counters

Certain translation rules generate text that varies according to the context of the element to which the rules apply. Variable text is defined either in the `VAR` section of the translation schema or in the rule itself (see the `Create` and `Write` rules). Both types of definition rely on counters for the calculation of variable material.

There are two types of counter: counters whose value is explicitely computed by applying `Set` and `Add` rules, and counters whose value is computed by a function associated with the counter. Those functions allow the same calculations as can be used in presentation schemas. As in a presentation schema, counters must be defined in the `COUNTERS` section of the translation schema before they are used.

When counters are used in a translation schema, the `COUNTERS` keyword is followed by the declarations of every counter used. Each declaration is composed of the counter's name possibly followed by a colon and the counting function to be used for the counter. The declaration is terminated by a semi–colon. If the counter is explicitely computed by `Set` and `Add` rules, no counting function is indicated. If a counting function is indicated, `Set` and `Add` rules cannot be applied to that counter.

The counting function indicates how the counter's value will be computed. Three functions are available: `Rank`, `Rlevel`, and `Set`.

- `Rank of ElemID` indicates that the counter's value is the rank of the element of type `ElemID` which encloses the element for which the counter is being evaluated. For the purposes of this function, an element of type `ElemID` is considered to enclose itself. This function is primarily used when the element of type `ElemID` is part of an aggregate or list, in which case the counter's value is the element's rank in its list or aggregate. Note that, unlike the `Rank` function for presentation schemas, the `Page` keyword cannot be used in place of the `ElemID`.
  The type name `ElemID` can be followed by an integer. That number represents the relative level, among the ancestors of the concerned element, of the element whose rank is asked. If that relative level n is unsigned, the n^{th} element of type `ElemID` encountered when travelling the logical structure from the root to the concerned element is taken into

account. If the relative level is negative, the logical structure is travelled in the other direction, from the concerned element to the root.

- `Rlevel of ElemID` indicates that the counter's values is the relative level in the tree of the element for which the counter is being evaluated. The counter counts the number of elements of type `ElemID` which are found on the path between the root of the document's logical structure tree and the element (inclusive).
- `Set n on Type1 Add m on Type2` indicates that the counter's value is calculated as follows: in traversing the document from the beginning to the element for which the counter is being evaluated, the counter is set to the value `n` each time a `Type1` element is encountered and is incremented by the amount `m` each time a `Type2` element is encountered. The initial value `n` and the increment `m` are integers.

As in a presentation schema, the `Rank` and `Set` functions can be modified by a numeric attribute which changes their initial value. This is indicated by the `Init` keyword followed by the numeric attribute's name. The `Set` function takes the value of the attribute instead of the `InitValue` (`n`). For the `Rank` function, the value of the attribute is considered to be the rank of the first element of the list (rather than the normal value of 1). Subsequent items in the list have their ranks shifted accordingly. In both cases, the attribute must be numeric and must be a local attribute of the root of the document itself.

```
CounterSeq  = Counter < Counter > .
Counter     = CounterID [ ':' CounterFunc ] ';' .
CounterID   = NAME .
CounterFunc = 'Rank' 'of' ElemID [ SLevelAsc ]
              [ 'Init' AttrID ] /
              'Rlevel' 'of' ElemID /
              'Set' InitValue 'On' ElemID
                    'Add' Increment 'On' ElemID
                    [ 'Init' AttrID ] .
SLevelAsc   = [ '-' ] LevelAsc .
LevelAsc    =  NUMBER .
InitValue   = NUMBER .
Increment   = NUMBER .
ElemID      = NAME .
AttrID      = NAME .
```

Example:

If the body of a chapter is defined in the structure schema by:

```
Chapter_Body = LIST OF
          (Section = BEGIN
                     Section_Title = Text;
                     Section_Body  = BEGIN
                                     Paragraphs;
                                     Section;
                                     END;
                     END
```

```
        );
```

(sections are defined recursively), a counter can be defined giving the number of a section within its level in the hierarchy:

```
COUNTERS
   SectionNumber : Rank of Section;
```

A counter holding the hierarchic level of a section:

```
   SectionLevel : Rlevel of Section;
```

A counter which sequentially numbers all the document's sections, whatever their hierarchic level:

```
   UniqueSectNum : Set 0 on Document Add 1 on Section;
```

Constants

A common feature of translation rules is the generation of constant text. This text can be defined in the rule that generates it (see for example the `Create` and `Write` rules); but it can also be defined once in the constant declaration section and used many times in different rules. The latter option is preferable when the same text is used in several rules or several variables.

The `CONST` keyword begins the constant declaration section of the translation schema. It must be omitted if no constants are declared. Each constant declaration is composed of the constant name, an equals sign, and the constant's value, which is a character string between apostrophes. A constant declaration is terminated by a semicolon.

```
ConstSeq   = Const < Const > .
Const      = ConstID '=' ConstValue ';' .
ConstID    = NAME .
ConstValue = STRING .
```

Example:

The following rule assigns the name `TxtLevel` to the character string ‘‘Level’’:

```
CONST
     TxtLevel = 'Level';
```

Variables

Variables allow to define variable text which is generated by the `Create` and `Write` rules. They are also used to define file names which are used in the `Create`, `ChangeMainFile`, `RemoveFile`, and `Indent` rules. Variables can be defined either in the `VAR` section of the translation schema or directly in the rules which use them. Variables that define file names must be declared in the `VAR` section, and when the same variable is used several times in the translation schema, it makes sense to define it globally in the `VAR` section. This section is only present if at least one variable is defined globally.

After the `VAR` keyword, each global variable is defined by its name, a colon separator and a sequence of functions (at least one function). Each variable definition is terminated by a semicolon. Functions determine the different parts which together give the value of the variable. The value is obtained by concatenating the strings produced by each of the functions. Seven types of functions are available. Each variable definition may use any number of functions of each type.

- The function `Value(Counter)` returns a string representing the value taken by the counter when it is evaluated for the element in whose rule the variable is used. The counter must have been declared in the `COUNTERS` section of the translation schema. When the counter is expressed in arabic numerals, the counter name can be followed by a colon and an integer indicating a minimum length (number of characters) for the string; if the counter's value is normally expressed with fewer characters than the required minimum, zeroes are added to the front of the string to achieve the minimum length.
  By default, the counter value is written in arabic digits. If another representation of that value is needed, the counter name must be followed by a comma and one of the following keywords:
  - `Arabic`: arabic numerals (default value),
  - `LRoman`: lower–case roman numerals,
  - `URoman`: upper–case roman numerals,
  - `Uppercase`: upper–case letter,
  - `Lowercase`: lower–case letter.
- The function `FileDir`, without parameter, returns a string representing the name of the directory of the output file that has been given as a parameter to the translation program. The string includes a character '/' at the end.
- The function `FileName`, without parameter, returns a string representing the name of the output file that has been given as a parameter to the translation program. The file extension (the character string that terminate the file name, after a dot) is not part of that string.
- The function `Extension`, without parameter, returns a string representing the extension of the file name. That string is empty if the file name that has been given as a parameter to the translation program has no extension. If there is an extension, its first character is a dot.
- The function `DocumentName`, without parameter, returns a string representing the name of the document being translated.
- The function `DocumentDir`, without parameter, returns a string representing the directory containing the document being translated.
- The function formed by the name of a constant returns that constant's value.
- The function formed by a character string between apostrophes returns that string.
- The function formed by the name of a buffer returns the contents of that buffer. If the named buffer is the picture buffer, then the name of the last picture encountered is

returned. Otherwise, the buffer is a read buffer and the value returned is text previously read from the terminal. If the buffer is empty (no picture has been encountered or the `Read` rule has not been executed for the buffer), then the empty string is returned.

- The function formed by an attribute name takes the value of the indicated attribute for the element to which the variable applies. If the element does not have that attribute, then the element's ancestor are searched toward the root of the tree. If one of the ancestors does have the attribute then its value is used. If no ancestors have the attribute, then the value of the function is the empty string.

```
VariableSeq = Variable < Variable > .
Variable    = VarID ':' Function < Function > ';' .
VarID       = NAME .
Function    ='Value' '(' CounterID [ ':' Length ]
                          [ ',' CounterStyle ] ')' /
             'FileDir' / 'FileName' / 'Extension' /
             'DocumentName' / 'DocumentDir' /
              ConstID / CharString /
              BufferID / AttrID .
Length      = NUMBER .
CounterStyle= 'Arabic' / 'LRoman' / 'URoman' /
              'Uppercase' / 'Lowercase' .
CharString  = STRING .
```

Example:

To create, at the beginning of each section of the translated document, text composed of the string ''Section'' followed by the section number, the following variable definition might be used:

```
VAR
     SectionVar : 'Section' Value(SectionNumber);
```

(see the definition of `SectionNumber`).

The following variable definition can be used to create, at the beginning of each section, the text ''Level'' followed by the hierarchical level of the section. It used the constant defined above.

```
     LevelVar : TxtLevel Value(SectionLevel);
```

(see the definitions of `SectionLevel` and of `TxtLevel`).

To generate the translation of each section in a different file (see rule `ChangeMainFile`), the name of these files might be defined by the following variable:

```
     VarOutpuFile : FileName Value(SectionNumber)
                    Extension;
```

If `output.txt` is the name of the output file specified when starting the translation program, translated sections are written in files `output1.txt`, `output2.txt`, etc.

Translating structure elements

The `RULES` keyword introduces the translation rules which will be applied to the various structured element types. Translation rules can be specified for each element type defined in the structure schema, including the base types defined implicitly, whose names are `TEXT_UNIT`, `PICTURE_UNIT`, `SYMBOL_UNIT`, `GRAPHIC_UNIT` and `PAGE_UNIT`. But it is not necessary to specify rules for every defined type.

If there are no translation rules for an element type, the elements that it contains (and which may have rules themselves) will still be translated, but the translator will produce nothing for the element itself. To make the translator completely ignore the content of an element the `Remove` rule must be used.

The translation rules for an element type defined in the structure schema are written using the name of the type followed by a colon and the list of applicable rules. When the element type is a mark pair, but only in this case, the type name must be preceded by the `First` or `Second` keyword. This keyword indicates whether the rules that follow apply to the first or second mark of the pair.

The list of rules can take several forms. It may be a simple non–conditional rule. It can also be formed by a condition followed by one or more simple rules. Or it can be a block of rules beginning with the `BEGIN` keyword and ending with the `END` keyword and a semicolon. This block of rules can contain one or more simple rules and/or one or more conditions, each followed by one or more simple rules.

```
ElemSeq        = TransType < TransType > .
TransType      = [ FirstSec ] ElemID ':' RuleSeq .
FirstSec       = 'First' / 'Second' .
RuleSeq        = Rule / 'BEGIN' < Rule > 'END' ';' .
Rule           = SimpleRule / ConditionBlock .
ConditionBlock = 'IF' ConditionSeq SimpleRuleSeq .
SimpleRuleSeq  = 'BEGIN' < SimpleRule > 'END' ';' /
                 SimpleRule .
```

Conditional rules

In a translation schema, the translation rules are either associated with element types or with attribute values or with a specific presentation. They are applied by the translator each time an element of the corresponding type is encountered in the translated document or each time the attribute value is carried by an element or also, each time the specific translation is attached to an element. This systematic application of the rules can be relaxed: it is possible to add a condition to one or more rules, so that these rules are only applied when the condition is true.

A condition begins with the keyword `IF`, followed by a sequence of elementary conditions. Elementary conditions are separated from each other by the `AND` keyword. If there is only one elementary condition, this keyword is absent. The rules are only applied if all the elementary conditions are true. The elementary condition can be negative; it is then preceded by the `NOT` keyword.

When the translation rule(s) controlled by the condition apply to a reference element or a reference attribute, an elementary condition can also apply to element referred by this reference. The `Target` keyword is used for that purpose. It must appear before the keyword defining the condition type.

Depending on their type, some conditions may apply either to the element with which they are associated, or to one of its ancestor. In the case of an ancestor, the key word `Ancestor` must be used, followed by

- either an integer which represents the number of levels in the tree between the element and the ancestor of interest,
- or the type name of the ancestor of interest. If that type is defined in a separate structure schema, the name of that schema must follow between parentheses.

There is a special case for the parent element, which can be simply written `Parent` instead of `Ancestor 1`.

Only conditions `First`, `Last`, `Referred`, `Within`, `Attributes`, `Presentation`, `Comment` and those concerning an attribute or a specific presentation can apply to an ancestor. Conditions `Defined`, `FirstRef`, `LastRef`, `ExternalRef`, `Alphabet`, `FirstAttr`, `LastAttr`, `ComputedPage`, `StartPage`, `UserPage`, `ReminderPage`, `Empty` cannot be preceded by keywords `Parent` or `Ancestor`.

In condition `Referred` and in the condition that applies to a named attribute, a symbol `'*'` can indicate that the condition is related only to the element itself. If this symbol is not present, not only the element is considered, but also its ancestor, at any level.

The form of an elementary condition varies according to the type of condition.

Conditions based on the logical position of the element

The condition can be on the position of the element in the document's logical structure tree. It is possible to test whether the element is the first (`First`) or last (`Last`) among its siblings or if it is not the first (`NOT First`) or not the last (`NOT Last`).

It is also possible to test if the element is contained in an element of a given type (`Within`) or if it is not (`NOT Within`). If that element type is defined in a structure schema which is not the one which corresponds to the translation schema, the type name of this element must be followed, between parentheses, by the name of the structure schema which defines it.

If the keyword `Within` is preceded by `Immediately`, the condition is satisfied only if the *parent* element has the type indicated. If the word `Immediately` is missing, the condition is satisfied if any *ancestor* has the type indicated.

An integer *n* can appear between the keyword `Within` and the type. It specifies the number of ancestors of the indicated type that must be present for the condition to be satisfied. If the keyword `Immediately` is also present, the *n* immediate ancestors of the element must have the indicated type. The integer *n* must be positive or zero. It can be preceded by `<` or `>` to indicate a maximum or minimum number of ancestors. If these symbols are missing, the condition is satisfied only if it exists exactly *n* ancestors. When this number is missing, it is equivalent to > 0.

If the condition applies to translation rules associated with an attribute, i.e. if it is in the `ATTRIBUTES` section of the presentation schema, the condition can be simply an element name. Translation rules are then executed only if the attribute is attached to an element of that type. The keyword `NOT` before the element name indicates that the translation rules must be executed only if the element is not of the type indicated.

Conditions on references

References may be taken into account in conditions, which can be based on the fact that the element, or one of its ancestors (unless symbol `*` is present), is designated by a at least one reference (`Referred`) or by none (`NOT Referred`). If the element or attribute to which the condition is attached is a reference, the condition can be based on the fact that it acts as the first reference to the designated element (`FirstRef`), or as the last (`LastRef`), or as a reference to an element located in another document (`ExternalRef`). Like all conditions, conditions on references can be inverted by the `NOT` keyword.

Conditions on the parameters

Elements which are parameters can be given a particular condition which is based on whether or not the parameter is given a value in the document (`Defined` or `NOT Defined`, respectively).

Conditions on the alphabets

The character string base type (and only this type) can use the condition `Alphabet = a` which indicates that the translation rule(s) should only apply if the alphabet of the character string is the one whose name appears after the equals sign (or is not, if there is a preceding `NOT` keyword). This condition cannot be applied to translation rules of an attribute.

In the current implementation of Thot, the available alphabets are the `Latin` alphabet and the `Greek` alphabet.

Conditions on page breaks

The page break base type (and only this type) can use the following conditions: `ComputedPage`, `StartPage`, `UserPage`, and `ReminderPage`. The `ComputedPage` condition indicates that the translation rule(s) should apply if the page break was created automatically by Thot; the `StartPage` condition is true if the page break is generated before the element by the `Page` rule of the P language; the `UserPage` condition applies if the page break was inserted by the user; and the `ReminderPage` is applied if the page break is a reminder of page breaking.

Conditions on the element's content

The condition can be based on whether or not the element is empty. An element which has no children or whose leaves are all empty is considered to be empty itself. This condition is expressed by the `Empty` keyword, optionally preceded by the `NOT` keyword.

Conditions on the presence of comments

The condition can be based on the presence or absence of comments associated with the translated element. This condition is expressed by the keyword `Comment`, optionally preceded by the keyword `NOT`.

Conditions on the presence of specific presentation rules

The condition can be based on the presence or absence of specific presentation rules associated with the translated element, whatever the rules, their value or their number. This condition is expressed by the keyword `Presentation`, optionally preceded by the `NOT` keyword.

Conditions on the presence of logical attributes

In the same way, the condition can be based on the presence or absence of attributes associated with the translated elements, no matter what the attributes or their values. The `Attributes` keyword expresses this condition.

Conditions on logical attributes

If the condition appears in the translation rules of an attribute, the `FirstAttr` and `LastAttr` keywords can be used to indicate that the rules must only be applied if this attribute is the first attribute for the translated element or if it is the last (respectively). These conditions can also be inverted by the `NOT` keyword.

Another type of condition can only be applied to the translation rules when the element being processed (or one of its ancestors if symbol `*` is missing) has a certain attribute, perhaps with a certain value or, in contrast, when the element does not have this attribute with this value. The condition is specified by writing the name of the attribute after the keyword `IF` or `AND`. The `NOT` keyword can be used to invert the condition. If the translation rules must be applied to any element which has this attribute (or does not have it, if the condition is inverted) no matter what the attribute's value, the condition is complete. If, in contrast, the condition applies to one or more values of the attribute, these are indicated after the name of the attribute, except for reference attributes which do not have values.

The representation of the values of an attribute in a condition depends on the attribute's type. For attributes with enumerated or textual types, the value (a name or character string between apostrophes, respectively) is simply preceded by an equals sign. For numeric attributes, the condition can be based on a single value or on a range of values. In the case of a unique value, this value (an integer) is simply preceded by an equals sign. Conditions based on ranges of values have several forms:

- all values less than a given value (the value is preceded by a ''less than'' sign).
- all values greater than a given value (the value is preceded by a ''greater than'' sign).
- all values falling in an interval, bounds included. The range of values is then specified `IN [`Minimum `..`Maximum`]`, where Minimum and Maximum are integers.

All numeric values may be negative. The integer is simply preceded by a minus sign.

Both local and global attributes can be used in conditions.

Conditions on specific presentation rules

It is possible to apply translation rules only when the element being processed has or does not have a specific presentation rule, possibly with a certain value. The condition is specified by writing the name of the presentation rule after the keyword `IF` or `AND`. The `NOT` keyword can be used to invert the condition. If the translation rules must be applied to any element which has this presentation rule (or does not have it, if the condition is inverted) no matter what the rule's value, the condition is complete. If, in contrast, the condition applies to one or more values of the rule, these are indicated after the name of the attribute.

The representation of presentation rule values in a condition is similar to that for attribute values. The representation of these values depend on the type of the presentation rule. There are three categories of presentation rules:

- those taking numeric values (`Size, Indent, LineSpacing, LineWeight`),
- those with values taken from a predefined list (`Adjust, Justify, Hyphenate, Style, Weight, Font, UnderLine, Thickness, LineStyle`),
- those whose value is a name (`FillPattern, Background, Foreground`).

For presentation rules which take numeric values, the condition can take a unique value or a range of values. In the case of a unique value, this value (an integer) is simply preceded by an equals sign. Conditions based on ranges of values have several forms:

- all values less than a given value (the value is preceded by a ''less than'' sign).
- all values greater than a given value (the value is preceded by a ''greater than'' sign).
- all values falling in an interval, bounds included. The range of values is then specified `IN [`Minimum `..`Maximum`]`, where Minimum and Maximum are integers.

Values for the `Indent`rule may be negative. The integer is then simply preceded by a minus sign and represents how far the first line starts to the left of the other lines.

For presentation rules whose values are taken from predefined lists, the value which satisfies the condition is indicated by an equals sign followed by the name of the value.

For presentation rule whose values are names, the value which satisfies the condition is indicated by the equals sign followed by the value's name. The names of fill patterns (the `FillPattern` rule) and of colors (the `Foreground` and `Background` rules) known to Thot are the same as in the P language.

The syntax of conditions based on the specific presentation is the same as the syntax used to express the translation of specific presentation rules.

When a condition has only one rule, the condition is simply followed by that rule. If it has several rules, they are placed after the condition between the keywords `BEGIN` and `END`.

```
ConditionSeq = Condition [ 'AND' Condition ] .
Condition    = [ 'NOT' ] [ 'Target' ] Cond .
Cond         = CondElem / CondAscend .
CondElem     ='FirstRef' / 'LastRef' /
               'ExternalRef' /
               'Defined' /
               'Alphabet' '=' Alphabet /
               'ComputedPage' / 'StartPage' /
               'UserPage' / 'ReminderPage' /
               'Empty' /
                ElemID /
               'FirstAttr' / 'LastAttr' .
CondAscend   = [ Ascend ] CondOnAscend .
Ascend       = '*' / 'Parent' / 'Ancestor' LevelOrType .
LevelOrType  = CondRelLevel / ElemID [ ExtStruct ] .
CondRelLevel = NUMBER .
CondOnAscend ='First' / 'Last' /
               'Referred' /
                 [ 'Immediately' ] 'Within' [ NumParent ]
                                  ElemID [ ExtStruct ] /
               'Attributes' /
                AttrID [ RelatAttr ] /
               'Presentation' /
                PresRule /
               'Comment' .
NumParent    = [ GreaterLess ] NParent .
```

```
GreaterLess   = '>' / '<' .
NParent       = NUMBER.
ExtStruct     = '(' ElemID ')' .
Alphabet      = NAME .
RelatAttr     ='=' Value /
               '>' [ '-' ] Minimum /
               '<' [ '-' ] Maximum /
               'IN' '[' [ '-' ] MinInterval '..'
                        [ '-' ] MaxInterval ']' .
Value         = [ '-' ] IntegerVal / TextVal / AttrValue .
Minimum       = NUMBER .
Maximum       = NUMBER .
MinInterval   = NUMBER .
MaxInterval   = NUMBER .
IntegerVal    = NUMBER .
TextVal       = STRING .
AttrValue     = NAME .
```

Example:

Suppose that after each element of type Section_Title it is useful to produce the text `\label{SectX}` where `X` represents the section number, but only if the section is designated by one or more references in the document. The following conditional rule produces this effect:

```
RULES
  Section_Title :
    IF Referred
       Create ('\label{Sect' Value(UniqueSectNum) '}\12') After;
```

(the declaration of the `UniqueSectNum` counter is given above). The string `\12` represents a line break.

Example:

Suppose that for elements of the Elmnt type it would be useful to produce a character indicating the value of the numeric attribute Level associated with the element: an ''A'' for all values of Level less than 3, a ''B'' for values between 3 and 10 and a ''C'' for values greater than 10. This can be achieved by writing the following rules for the Elmnt type:

```
RULES
  Elmnt :
    BEGIN
    IF Level < 3
      Create 'A';
    IF Level IN [3..10]
      Create 'B';
    IF Level > 10
      Create 'C';
    END;
```

Translation rules

Fifteen types of translation rules can be associated with element types and attribute values. They are the `Create`, `Write`, `Read`, `Include`, `Get`, `Copy`, `Use`, `Remove`, `NoTranslation`, `NoLineBreak`, `ChangeMainFile`, `RemoveFile`, `Set`, `Add`, `Indent`, rules. Each rule has its own syntax, although they are all based on very similar models.

```
SimpleRule = 'Create' [ 'IN' VarID ] Object
                    [ Position ] ';' /
             'Write' Object [ Position ] ';' /
             'Read' BufferID [ Position ] ';' /
             'Include' File [ Position ] ';' /
             'Get' [ RelPosition ] ElemID
                    [ ExtStruct ]
                    [ Position ] ';' /
             'Copy' [ RelPosition ] ElemID
                    [ ExtStruct ]
                    [ Position ] ';' /
             'Use' TrSchema [ 'For' ElemID ] ';' /
             'Remove' ';' /
             'NoTranslation' ';' /
             'NoLineBreak' ';' /
             'ChangeMainFile' VarID [ Position ] ';' /
             'RemoveFile' VarID [ Position ] ';' /
             'Set' CounterID InitValue [ Position ] ';' /
             'Add' CounterID Increment [ Position ] ';' /
             'Indent' [ 'IN' VarID ] Indent [ Position ] ';' .
```

The `Create` rule

The most frequently used rule is undoubtedly the `Create` rule, which generates fixed or variable text (called an *object*) in the output file. The generated text can be made to appear either before or after the content of the element to which the rule applies. The rule begins with the `Create` keyword, followed by a specifier for the object and a keyword (`Before` or `After`) indicating the position of the generated text (before or after the element's content). If the position is not indicated, the object will be generated before the element's content. This rule, like all translation rules, is terminated by a semicolon.

The `Create` keyword can be followed by the `IN` keyword and by the name of a variable. This means that the text generated by the rule must not be written in the main output file, but in the file whose name is specified by the variable.

This allows the translation program to generate text in different files during the same run. These files do not need to be explicitely declared or opened. They do not need to be closed either, but if they contain temporary data, they can be removed (see the `RemoveFile` rule). As soon as the translation program executes a `Create` rule for a file that is not yet open, it opens the file. These files are closed when the translation is finished.

```
           'Create' [ 'IN' VarID ] Object
                    [ Position ] ';'
Object   = ConstID / CharString /
           BufferID /
```

```
             VarID /
            '(' Function < Function > ')' /
             [ 'Translated' ] AttrID /
            'Value' /
            'Content' /
            'Comment' /
            'Attributes' /
            'Presentation' /
            'RefId' /
            'PairId' /
            'FileDir' /
            'FileName' /
            'Extension' /
            'DocumentName' /
            'DocumentDir' /
             [ 'Referred' ] ReferredObject .
Position ='After' / 'Before' .

ReferredObject = VarID /
             ElemID [ ExtStruct ] /
            'RefId' /
            'DocumentName' /
            'DocumentDir' .
```

The object to be generated can be:

- a constant string, specified by its name if it is declared in the schema's `CONST` section, or given directly as a value between apostrophes;
- the contents of a buffer, designated by the name of the buffer;
- a variable, designated by its name if it is declared in the translation schema's `VAR` section, or given directly between parentheses. The text generated is the value of that variable evaluated for the element to which the rule applies.
- the value of an attribute, if the element being translated has this attribute. The attribute is specified by its name. If it's a text attribute, it can be preceded by the `Translated` keyword, which causes the attribute value to be recoded using the text translation table defined by section `TEXTTRANSLATE`;
- the value of a specific presentation rule. This object can only be generated if the translation rule is for a specific presentation rule. It is specified by the `Value` keyword;
- the element's content. That is, the content of the leaves of the subtree of the translated element. This is specified by the `Content` keyword;
- the comment attached to the element. When the element doesn't have a comment, nothing is generated. This is indicated by the `Comment` keyword;
- the translation of all attributes of the element (which is primarily used to apply the attribute translation rules before those of the element type). This is specified by the `Attributes` keyword.
- the translation of all of the element's specific presentation rules (which is primarily used to apply the translation rules for the specific presentation rules before those of the element or its attributes). This option is specified by the `Presentation` keyword;

- The value of the reference's identifier.
  Thot associates a unique identifier with each element in a document. This identifier (called *reference's identifier* or *label*) is a character string containing the letter 'L' followed by digits. Thot uses it in references for identifying the referred element.
  The `RefId` keyword produces the reference's identifier of the element to which the translation rule is applied, or the reference's identifier of its first ancestor that is referred by a reference or that can be referred by a reference.
- the value of a mark pair's unique identifier. This may only be used for mark pairs and is indicated by the `PairId` keyword.
- the directory containing the file being generated (this string includes an ending '/', if it is not empty). This is indicated by the `FileDir` keyword.
- the name of the file being generated (only the name, without the directory and without the extension). This is indicated by the `FileName` keyword.
- the extension of the file being generated (this string starts with a dot, if it is not empty). This is indicated by the `Extension` keyword.
- the name of the document being translated. This is indicated by the `DocumentName` keyword.
- the directory containing the document being translated. This is indicated by the `DocumentDir` keyword.

When the rule applies to a reference (an element or an attribute defined as a reference in the structure schema), it can generate a text related to the element referred by that reference. The rule name is then followed by the `Referred` keyword and a specification of the object to be generated for the referred element. This specification can be:

- the name of a variable. The rule generates the value of that variable, computed for the referred element.
- an element type. The rule generates the translation of the element of that type, which is in the subtree of the referred element. If this element is not defined in the structure schema which corresponds to the translation schema (that is, an object defined in another schema), the element's type name must be followed by the name of its structure schema between parentheses.
- the `RefId` keyword. The rule generates the reference's identifier of the referred element.
- the `DocumentName` keyword. The rule generates the name of the document to which the referred element belongs.
- the `DocumentDir` keyword. The rule generates the name of the directory that contains the document of the referred element.

The `Write` rule

The `Write` has the same syntax as the `Create` rule. It also produces the same effect, but the generated text is displayed on the user's terminal during the translation of the document, instead of being produced in the translated document. This is useful for helping the user keep track of the progress of the translation and for prompting the user on the terminal for input required by the `Read` rule.

```
'Write' Object [ Position ] ';'
```

Notice: if the translator is launched by the editor (by the ''Save as'' command), messages produced by the `Write` rule are not displayed.

Example:

To make the translator display the number of each section being translated on the user's terminal, the following rule is specified for the `Section` element type:

```
Section : BEGIN
           Write VarSection;
          ...
          END;
```

(see above for the definition of the `VarSection` variable).

To display text on the terminal before issuing a read operation with the `Read` rule, the following rule is used:

```
BEGIN
Write 'Enter the name of the destination: ';
...
END;
```

The `Read` rule

The `Read` rule reads text from the terminal during the translation of the document and saves the text read in one of the buffers declared in the `BUFFERS` section of the schema. The buffer to be used is indicated by its name, after the `READ` keyword. This name can be followed, as in the `Create` and `Write` rules, by a keyword indicating if the read operation must be performed `Before` or `After` the translation of the element's content. If this keyword is absent, the read operation is done beforehand. The text is read into the buffer and remains there until a rule using the same buffer – possibly the same rule – is applied.

```
'Read' BufferID [ Position ] ';'
```

Example:

The following set of rules tells the user that the translator is waiting for the entry of some text, reads this text into a buffer and copies the text into the translated document.

```
BEGIN
Write 'Enter the name of the destination: ';
```

```
Read DestName;
Create DestName;
...
END;
```

(see above the definition of `DestName`).

The `Include` rule

The `Include` rule, like the `Create` rule, is used to produce text in the translated document. It inserts constant text which is not defined in the translation schema, but is instead taken from a file. The file's name is specified after the `Include` keyword, either directly as a character string between apostrophes or as the name of one of the buffers declared in the `BUFFERS` section of the schema. In the latter case, the buffer is assumed to contain the file's name. This can be used when the included file's name is known only at the moment of translation. This only requires that the `Include` rule is preceded by a `Read` rule which puts the name of the file desired by the user into the buffer.

Like the other rules, it is possible to specify whether the inclusion will occur before or after the element's content, with the default being before. The file inclusion is only done at the moment of translation, not during the compilation of the translation schema. Thus, the file to be included need not exist during the compilation, but it must be accessible at the time of translation. Its contents can also be modified between two translations, thus producing different results, even if neither the document or the translation schema are modified.

During translation, the file to be included is searched for along the schema directory path (indicated by the environment variable `THOTSCH`). The file name is normally only composed of a simple name, without specification of a complete file path. However, if the filename starts with a '/', it is considered as an absolute path.

```
            'Include' File [ Position ] ';'
File     = FileName / BufferID .
FileName = STRING .
```

Example:

Suppose that it is desirable to print documents of the Article class with a formatter which requires a number of declarations and definitions at the beginning of the file. The `Include`rule can be used to achieve this. All the declarations and definitions a replaced in a file called `DeclarArt` and then the `Article` element type is given the following rule:

```
Article : BEGIN
          Include 'DeclarArt' Before;
          ...
```

```
END;
```

The `Get` rule

The `Get` rule is used to change the order in which the elements appear in the translated document. More precisely, it produces the translation of a specified element before or after the translation of the content of the element to which the rule applies. The `Before` and `After` keywords are placed at the end of the rule to specify whether the operation should be performed before or after translation of the rule's element (the default is before). The type of the element to be moved must be specified after the `Get` keyword, optionally preceded by a keyword indicating where the element will be found in the logical structure of the document:

`Included`
: The element to be moved is the first element of the indicated type which is found inside the element to which the rule applies.

`Referred`
: This keyword can only be used if the rule applies to a reference element. The element to be moved is either the element designated by the reference (if that element is of the specified type), or the first element of the desired type contained within the element designated by the reference.

no keyword
: If the element to be moved is an associated element, defined in the `ASSOC` section of the structure schema, all associated elements of this type which have not been translated yet are then translated. Certain elements may in fact have already been translated by a `Get Referred` rule.
: If the element to be moved is not an associated element, the translator takes the first element of the indicated type from among the siblings of the rule's element. This is primarily used to change the order of the components of an aggregate.

If the element to be moved is defined in a structure schema which is not the one which corresponds to the translation schema (in the case of an included object with a different schema), the type name of this element must be followed, between parentheses, by the name of the structure schema which defines it.

```
                 'Get' [ RelPosition ] ElemID
                       [ ExtStruct ]
                       [ Position ] ';' /
     RelPosition = 'Included' / 'Referred' .
     ExtStruct   = '(' ElemID ')' .
```

The `Get` rule has no effect if the element which it is supposed to move has already been translated. Thus, the element will not be duplicated. It is generally best to associate the rule with the first element which will be encountered by the translator in its traversal of the document. Suppose an aggregate has two elements `A` and `B`, with `A` appearing first in the logical structure. To permute these two elements, a `Get B before` rule should be associated with the `A` element type, not the inverse. Similarly, a rule of the form `Get Included X After`, even though syntactically correct, makes no sense since, by the time it will be applied, after the translation of the contents of the element to which it is attached, the `X` element will already have been translated.

The `Get` rule is the only way to obtain the translation of the associated elements. In fact, the translator only traverses the primary tree of the document and thus does not translate the associated elements, except when the translation is explicitly required by a `Get Referred Type` or `Get Type` rule where `Type` is an associated element type.

Example:

The structure schema defined figures as associated element which are composed of some content and a caption. Moreover, it is possible to make references to figures, using elements of the `RefFigure` type:

```
      ...
      RefFigure = REFERENCE(Figure);
ASSOC
     Figure    = BEGIN
                   Content = NATURE;
                   Caption = Text;
                  END;
      ...
```

Suppose it would be useful to make a figure appear in the translated document at the place in the text where the first reference to the figure is made. If some figures are not referenced, then they would appear at the end of the document. Also, each figure's caption should appear before the content. The following rules in the translation schema will produce this result:

```
Article :    BEGIN
             ...
             Get Figures After;
             END;
RefFigure : BEGIN
             If FirstRef Get Referred Figure;
             ...
             END;
Content :    BEGIN
             Get Caption Before;
             ...
             END;
```

The `Copy` rule

Like the `Get` rule, the `Copy` rule generates the translation of a specified element, but it acts even if the element has already been translated and it allows to copy it or to translate it later. Both rules have the same syntax.

```
'Copy' [ RelPosition ] ElemID
```

```
                 [ ExtStruct ] [ Position ] ';'
```

The `Use` rule

The `Use` rule specifies the translation schema to be applied to objects of a certain class that are part of the document. This rule only appears in the rules for the root element of the document (the first type defined after the `STRUCT` keyword in the structure schema) or the rules of an element defined by an external structure (by another structure schema). Also, the `Use` rule cannot be conditional.

If the rule is applied to an element defined by an external structure, the `Use` keyword is simply followed by the name of the translation schema to be used for element constructed according to that external structure. If the rule is applied to the document's root element, it is formed by the `Use` keyword followed by the translation schema's name, the `For` keyword and the name of the external structure to which the indicated translation schema should be applied.

```
                'Use' TrSchema [ 'For' ElemID ] ';'
     TrSchema = NAME .
```

If no `Use` rule defines the translation schema to be used for an external structure which appears in a document, the translator asks the user, during the translation process, which schema should be used. Thus, it is not necessary to give the translation schema a `Use` rule for every external structure used, especially when the choice of translation schemas is to be left to the user.

Notice: if the translator is launched by the editor (by the ''Save as'' command), prompts are not displayed.

Example:

The `Article` structure schema uses the `Formula` external structure, defined by another structure schema, for mathematical formulas:

```
STRUCTURE Article;
   ...
STRUCT
   Article = ...
   ...
   Formula_in_text  = Formula;
   Isolated_formula = Formula;
   ...
END
```

Suppose that it would be useful to use the `FormulaT` translation schema for the formulas of an article. This can be expressed in two different ways in the `Article` class translation schema, using the rules:

```
RULES
    Article :
       Use FormulaT for Formula;
```

or:

```
RULES
    ...
    Formula :
       Use FormulaT;
```

The `Remove` rule

The `Remove` rule indicates that nothing should be generated, in the translated document, for the content of the element to which the rule applies. The content of that element is simply ignored by the translator. This does not prevent the generation of text for the element itself, using the `Create` or `Include` rules, for example.

The `Remove` rule is simply written with the `Remove` keyword. It is terminated, like all rules, by a semicolon.

```
'Remove' ';'
```

The `NoTranslation` rule

The `NoTranslation` rule indicates to the translator that it must not translate the content of the leaves of the element to which it applies. In contrast to the `Remove` rule, it does not suppress the content of the element, but it inhibits the translation of character strings, symbols, and graphical elements contained in the element. These are retrieved so that after the translation of the document, the rules of the `TEXTTRANSLATE`, `SYMBTRANSLATE` and `GRAPHTRANSLATE` sections will not be applied to them.

The `NoTranslation` rule is written with the `NoTranslation` keyword followed by a semicolon.

```
'NoTranslation' ';'
```

The `NoLineBreak` rule

The `NoLineBreak` rule indicates to the translator that it must not generate additional line breaks in the output produced for the element to which it applies. This is as if it was an instruction `LINELENGTH 0;` at the beginning of the translation schema, but only for the current element.

The `NoLineBreak` rule is written with the `NoLineBreak` keyword followed by a semicolon.

```
'NoLineBreak' ';'
```

The `ChangeMainFile` rule

When the translation program starts, it opens a main output file, whose name is given as a parameter of the translator. All `Create` rules without explicit indication of the output file write sequentially in this file. When a `ChangeMainFile` rule is executed, the main output file is closed and it is replaced by a new one, whose name is specified in the `ChangeMainFile` rule. The `Create` rules without indication of the output file that are then executed write in this new file. Several `ChangeMainFile` rules can be executed during the same translation, for dividing the main output into several files.

This rule is written with the `ChangeMainFile` keyword followed by the name of a variable that specifies the name of the new main file. The keyword `Before` or `After` can be placed at the end of the rule to specify whether the operation should be performed before or after translation of the rule's element (the default is before). This rule, like all translation rules, is terminated by a semicolon.

```
'ChangeMainFile' VarID [ Position ] ';'
```

Example:

To generate the translation of each section in a different file, the following rule can be associated with type `Section`. That rule uses the `VarOutpuFile` variable defined above.

```
Section:
     ChangeMainFile VarOutpuFile Before;
```

If `output.txt` is the name of the output file specified when starting the translation program, translated sections are written in files `output1.txt`, `output2.txt`, etc.

The `RemoveFile` rule

Files may be used for storing temporary data that are no longer needed when the translation of a document is complete. These files may be removed by the `RemoveFile` rule.

This rule is written with the `RemoveFile` keyword followed by the name of a variable that specifies the name of the file to be removed. The keyword `Before` or `After` can be placed at the end of the rule to specify whether the operation should be performed before or after translation of the rule's element (the default is before). This rule, like all translation rules, is terminated by a semicolon.

```
'RemoveFile' VarID [ Position ] ';'
```

The `Set` and `Add` rules

The `Set` and `Add` rules are used for modifying the value of counters that have no counting function. Only this type of counter can be used in the `Set` and `Add` rules.

Both rules have the same syntax: after the keyword `Set` or `Add` appear the counter name and the value to assign to the counter (`Set` rule) or the value to be added to the counter (`Add` rule). The keyword `Before` or `After` can follow that value to indicate when the rule must be applied: before or after the element's content is translated. By default, `Before` is assumed. A semicolon terminates the rule.

```
               'Set' CounterID InitValue [ Position ] ';' /
               'Add' CounterID Increment [ Position ] ';'
```

The `Indent` rule

The `Indent` rule is used to modify the value of text indentation in the output files.

Each time the translator creates a new line in an output file, it generates a variable number of space characters at the beginning of the new line. By default, the number of these characters (the indentation value) is 0. It can be changed with the `Indent` rule.

In its simple form, the rule begins with the `Indent` keyword, followed by the indentation sign (optional) and value and a keyword `Before` or `After` indicating that the indentation should be changed before or after the element's content is generated. If the position is not indicated, the indentation is changed before the element's content is generated. This rule, like all translation rules, is terminated by a semicolon.

The indentation value is indicated by an integer, which is the number of space characters to be generated at the beginning of each new line. A sign (`+` or `-`) can appear before the integer to indicate that the value is relative: the current value of indentation is incremented (if sign is `+`) or decremented (if sign is `-`) by the specified value.

Keyword `Suspend` or `Resume` can appear instead of the (possibly signed) identation value. `Suspend` means that the new indentation value to be used is zero until another `Indent` rule is executed and changes the indentation value. `Resume` means that the indentation value that was used before the last `Indent Suspend` was executed becomes the new value. Only one `Suspend` can be used before a `Resume`; `Supend-Resume` pairs can not be nested.

Like the `Create` rule, the `Indent` keyword can be followed by the `IN` keyword and by the name of a variable. This means that the rule must not change indentation in the main output file, but in the file whose name is specified by the variable (by default, indentation is changed in the main output file).

```
               'Indent' [ 'IN' VarID ] Indent [ Position ] ';' .

Indent         = 'Suspend' / 'Resume' / [ IndentSign ] IndentValue .
IndentSign     = '+' / '-' .
IndentValue    = NUMBER .
```

Rule application order

The translator translates the elements which comprise the document in the order induced by the tree structure, except when the `Get` rule is used to change the order of translation. For each element, the translator first applies the rules specified for the element's type that must be applied before translation of the element's content (rules ending with the `Before` keyword or which have no position keyword). If several rules meet these criteria, the translator applies them in the order in where they appear in the translation schema.

It then applies all rules for the attributes which the element has and which must be applied before the translation of the element's content (rules ending with the `Before` keyword or which have no position keyword). For one attribute value, the translator applies the rules in the order in which they are defined in the translation schema.

The same procedure is followed with translation rules for specific presentations.

Next, the element's content is translated, as long as a `Remove` rule does not apply.

In the next step, the translator applies rules for the specific presentation of the element that are to be applied after translation of the content (rules which end with the `After` keyword). The rules for each type of presentation rule or each value are applied in the order in which the translation appear in the schema.

Then, the same procedure is followed for translation rules for attributes of the element.

Finally, the translator applies rules for the element which must be applied after translation of the element's content. These rules are applied in the order that they appear in the translation schema. When the translation of an element is done, the translator procedes to translate the following element.

This order can be changed with the `Attributes` and `Presentation` options of the `Create` rule.

Translation of logical attributes

After the rules for the element types, the translation schema defines rules for attribute values. This section begins with the `ATTRIBUTES` keyword and is composed of a sequence of rule blocks each preceded by an attribute name and an optional value or value range.

If the attribute's name appears alone before the rule block, the rule are applied to all element which have the attribute, no matter what value the attribute has. In this case, the attribute name is followed by a colon before the beginning of the rule block.

The attribute's name can be followed by the name of an element type between parentheses. This says, as in presentation schemas, that the rule block which follows applies not to the element which has the attribute, but to its descendants of the type indicated between the parentheses.

If values are given after the attribute name (or after the name of the element type), the rules are applied only when the attribute has the indicated values. The same attribute can appear several times, with different values and different translation rules. Attribute values are indicated in the same way as in conditions and are followed by a colon before the block of rules.

The rule block associated with an attribute is either a simple rule or a sequence of rules delimited by the `BEGIN` and `END` keywords. Note that rules associated with attribute values cannot be conditional.

Translation rules are not required for all attributes (or their values) defined in a structure schema. Only those attributes for which a particular action must be performed by the translator must have such rules. The rules that can be used are those described above, from `Create` to `NoTranslation`.

```
AttrSeq        = TransAttr < TransAttr > .
TransAttr      = AttrID [ '(' ElemID ')' ]
                 [ RelatAttr ] ':' RuleSeq .
AttrID         = NAME .
ElemID         = NAME .
```

Example:

The structure defined the ''Language'' attribute which can take the values ''French'' and ''English''. To have the French parts of the original document removed and prevent the translation of the leaves of the English parts, the following rules would be used:

```
ATTRIBUTES
   Language=French :
      Remove;
   Language=English :
      NoTranslation;
```

Translation of specific presentations

After the rules for attributes, the translation schema defines rules for the specific presentation. This section begins with the `PRESENTATION` keyword and is composed of a sequence of translation rule blocks each preceded by a presentation rule name, optionally accompanied by a part which depends on the particular presentation rule.

Each of these translation rule blocks is applied when the translator operates on an element which has a specific presentation rule of the type indicated at the head of the block. Depending on the type of the specific presentation rule, it is possible to specify values of the presentation rule for which the translation rule block should be applied.

There are three categories of the presentation rules:

- rules taking numeric values: `Size`, `Indent`, `LineSpacing`, `LineWeight`,
- rules whose values are taken from a predefined list (i.e. whose type is an enumeration): `Adjust, Justify, Hyphenate,/TT>, Style, Weight, Font, UnderLine, Thickness, LineStyle,`
- rules whose value is a name: `FillPattern`, `Background`, `Foreground`.

For presentation rules of the first category, the values which provoke application of the translation rules are indicated in the same manner as for numeric attributes. This can be either a unique value or range of values. For a unique value, the value (an integer) is simply preceded by an equals sign. Value ranges can be specified in one of three ways:

- all values less than a given value (this value is preceded by a ‘‘less than’’ sign ’<’),
- all values greater than a given value (this value is preceded by a‘ ‘greater than’’ sign ’>’),
- all values falling in an interval, bounds included. The range of values is then specified `IN [`Minimum`..`Maximum`]`, where Minimum and Maximum are integers.

All numeric values can be negative, in which case the integer is preceded by a minus sign. All values must be given in typographers points.

For presentation rules whose values are taken from a predefined list, the value which provokes application of the translation rules is simply indicated by the equals sign followed by the name of the value.

For presentation rules whose values are names, the value which provokes the application of translation rules is simply indicated by the equals sign followed by the name of the value. The names of the fill patterns (the `FillPattern` rule) and of the colors (the `Foreground` and `Background` rules) used in Thot are the same as in the P language.

```
PresSeq         = PresTrans < PresTrans > .
PresTrans       = PresRule ':' RuleSeq .
PresRule        = 'Size' [ PresRelation ] /
                  'Indent' [ PresRelation ] /
                  'LineSpacing' [ PresRelation ] /
                  'Adjust' [ '=' AdjustVal ] /
                  'Justify' [ '=' BoolVal ] /
                  'Hyphenate' [ '=' BoolVal ] /
                  'Style' [ '=' StyleVal ] /
                  'Weight' [ '=' WeightVal ] /
                  'Font' [ '=' FontVal ] /
                  'UnderLine' [ '=' UnderLineVal ] /
                  'Thickness' [ '=' ThicknessVal ] /
                  'LineStyle' [ '=' LineStyleVal ] /
                  'LineWeight' [ PresRelation ] /
                  'FillPattern' [ '=' Pattern ] /
                  'Background' [ '=' Color ] /
                  'Foreground' [ '=' Color ] .

PresRelation    = '=' PresValue /
                  '>' [ '-' ] PresMinimum /
                  '<' [ '-' ] PresMaximum /
                  'IN' '[' [ '-' ] PresIntervalMin '..'
                          [ '-' ] PresIntervalMax ']' .
AdjustVal       = 'Left' / 'Right' / 'VMiddle' /
                  'LeftWithDots' .
BoolVal         = 'Yes' / 'No' .
StyleVal        = 'Roman' / 'Italics' / 'Oblique' .
WeightVal       = 'Normal' / 'Bold' .
FontVal         = 'Times' / 'Helvetica' / 'Courier' .
```

```
UnderLineVal    = 'NoUnderline' / 'UnderLined' /
                  'OverLined' / 'CrossedOut' .
ThicknessVal    = 'Thick' / 'Thin' .
LineStyleVal    = 'Solid' / 'Dashed' / 'Dotted' .
Pattern         = NAME .
Color           = NAME .
PresMinimum     = NUMBER .
PresMaximum     = NUMBER .
PresIntervalMin= NUMBER .
PresIntervalMax= NUMBER .
PresValue       = [ '-' ] PresVal .
PresVal         = NUMBER .
```

The translation rules associated with specific presentation rules can use the value of the specific presentation rule that causes them to be applied. This behavior is designated by the keyword `Value`. For numerically−valued presentation rules, the numeric value is produced. For other presentation rules, the name of the value is produced.

It should be noted that modifications to the layout of the document's elements that are made using the combination of the control key and a mouse button will have no effect on the translation of the document.

Example:

Suppose that it is desirable to use the same font sizes as in the specific presentation, but the font size must be between 10 and 18 typographer's points. If font size is set in the translated document by the string `pointsize=n` where `n` is the font size in typographer's points then the following rules will suffice:

```
PRESENTATION
   Size < 10 :
        Create 'pointsize=10';
   Size in [10..18] :
        BEGIN
        Create 'pointsize=';
        Create Value;
        END;
   Size > 18 :
        Create 'pointsize=18';
```

Recoding of characters, symbols and graphics

The coding of characters, graphical elements and symbols as defined in Thot does not necessarily correspond to what is required by an application to which a Thot document must be exported. Because of this the translator can recode these terminal elements of the documents structure. The last sections of a translation schema are intended for this purpose, each specifying the recoding rules for one type of terminal element.

The recoding rules for character strings are grouped by alphabets. There is a group of rules for each alphabet of the Thot document that must be translated. Each such group of rules begins with the `TEXTTRANSLATE` keyword, followed by the specification of the alphabet to translate and the recoding rules, between the BEGIN and END keywords unless there is only one recoding rule for the alphabet. The specification of the alphabet is not required: by default it is assumed to the Latin alphabet (the ISO Latin−1 character set).

Each recoding rule is formed by a source string between apostrophes and a target string, also between apostrophes, the two strings being separated by the arrow symbol (->), formed by the ''minus'' and ''greater than'' characters. The rule is terminated by a semi−colon.

```
TextTransSeq = [ Alphabet ] TransSeq .
Alphabet     = NAME .
TransSeq     ='BEGIN' < Translation > 'END' ';' /
               Translation .
Translation  = Source [ '->' Target ] ';' .
Source       = STRING .
Target       = STRING .
```

One such rule signifies that when the source string appears in a text leaf of the document being translated, the translator must replace it, in the translated document, with the target string. The source string and the target string can have different lengths and the target string can be empty. In this last case, the translator simply suppresses every occurrence of the source string in the translated document.

For a given alphabet, the order of the rules is not important and has no significance because the T language compiler reorders the rules in ways that speed up the translator's work. The total number of recoding rules is limited by the compiler as is the maximum length of the source and target strings.

The recoding rules for symbols and graphical elements are written in the same manner as the recoding rules for character strings. They are preceded, respectively, by the SYMBTRANSLATE and GRAPHTRANSLATE and so not require a specification of the alphabet. Their source string is limited to one character, since, in Thot, each symbol and each graphical element is represented by a single character. The symbol and graphical element codes are defined along with the non−standard character codes.

> **Example:**
>
> In a translation schema producing documents destined for use with the LATEX formatter, the Latin characters''é'' (octal code 351 in Thot) and ''è'' (octal code 350 in Thot) must be converted to their representation in LATEX:

```
TEXTTRANSLATE Latin
     BEGIN
     '\350' -> '\`{e}';     { e grave }
     '\351' -> '\''{e}';    { e acute }
     END;
```

Language grammars

This chapter gives the complete grammars of the languages of Thot. The grammars were presented and described in the preceding chapters, which also specify the semantics of the languages. This section gives only the syntax.

The M meta–language

The language grammars are all expressed in the same formalism, the M meta–language, which is defined in this section.

```
{ Any text between braces is a comment. }
Grammar       = Rule < Rule > 'END' .
                { The < and > signs indicate zero }
                { or more repetitions. }
                { END marks the end of the grammar. }
Rule          = Ident '=' RightPart '.' .
                { The period indicates the end of a rule }
RightPart     = RtTerminal / RtIntermed .
                { The slash indicates a choice }
RtTerminal    ='NAME' / 'STRING' / 'NUMBER' .
                { Right part of a terminal rule }
RtIntermed    = Possibility < '/' Possibility > .
                { Right part of an intermediate rule }
Possibility   = ElemOpt < ElemOpt > .
ElemOpt       = Element / '[' Element < Element > ']' /
               '<' Element < Element > '>'  .
                { Brackets delimit optional parts }
Element       = Ident / KeyWord .
Ident         = NAME .
                { Identifier, sequence of characters
KeyWord       = STRING .
                { Character string delimited by apostrophes }
END
```

The S language

The S language is used to write structure schemas, which contain the generic logical structures of document and object classes. It is described here in the M meta–language.

```
StructSchema    = 'STRUCTURE' [ 'EXTENSION' ] ElemID ';'
                  'DEFPRES' PresID ';'
                [ 'ATTR' AttrSeq ]
                [ 'PARAM' RulesSeq ]
                [ 'STRUCT' RulesSeq ]
                [ 'EXTENS' ExtensRuleSeq ]
                [ 'ASSOC' RulesSeq ]
                [ 'UNITS' RulesSeq ]
                [ 'EXPORT' SkeletonSeq ]
                [ 'EXCEPT' ExceptSeq ]
                  'END' .

ElemID          = NAME .
PresID          = NAME .

AttrSeq         = Attribute < Attribute > .
```

```
Attribute       = AttrID '=' AttrType ';' .
AttrType        = 'INTEGER' / 'TEXT' /
                  'REFERENCE' '(' RefType ')' /
                  ValueSeq .
RefType         = 'ANY' /
                  [ FirstSec ] ElemID [ ExtStruct ] .
ValueSeq        = AttrVal < ',' AttrVal > .
AttrID          = NAME .
FirstSec        = 'First' / 'Second' .
ExtStruct       = '(' ElemID ')' .
AttrVal         = NAME .

RulesSeq        = Rule < Rule > .
Rule            = ElemID [ LocAttrSeq ] '='
                  DefWithAttr ';' .
LocAttrSeq      = '(' 'ATTR' LocalAttr
                       < ';' LocalAttr > ')' .
LocalAttr       = [ '!' ] AttrID [ '=' AttrType ] .
DefWithAttr     = Definition
                  [ '+' '(' ExtensionSeq ')' ]
                  [ '-' '(' RestrictSeq ')' ]
                  [ 'WITH' FixedAttrSeq ] .
ExtensionSeq    = ExtensionElem < ',' ExtensionElem > .
ExtensionElem   = ElemID / 'TEXT' / 'GRAPHICS' /
                  'SYMBOL' / 'PICTURE' .
RestrictSeq     = RestrictElem < ',' RestrictElem > .
RestrictElem    = ElemID / 'TEXT' / 'GRAPHICS' /
                  'SYMBOL' / 'PICTURE' .
FixedAttrSeq    = FixedAttr < ',' FixedAttr > .
FixedAttr       = AttrID [ FixedOrModifVal ] .
FixedOrModifVal= [ '?' ] '=' FixedValue .
FixedValue      = [ '-' ] NumValue / TextValue / AttrVal .
NumValue        = NUMBER .
TextValue       = STRING .

Definition      = BaseType [ LocAttrSeq ] / Constr /
                  Element .
BaseType        = 'TEXT' / 'GRAPHICS' / 'SYMBOL' /
                  'PICTURE' / 'UNIT' / 'NATURE' .
Element         = ElemID [ ExtOrDef ] .
ExtOrDef        = 'EXTERN' / 'INCLUDED' /
                  [ LocAttrSeq ] '=' Definition .

Constr          = 'LIST' [ '[' min '..' max ']' ] 'OF'
                         '(' DefWithAttr ')' /
                  'BEGIN' DefOptSeq 'END' /
                  'AGGREGATE' DefOptSeq 'END' /
                  'CASE' 'OF' DefSeq 'END' /
                  'REFERENCE' '(' RefType ')' /
                  'PAIR' .

min             = Integer / '*' .
max             = Integer / '*' .
Integer         = NUMBER .

DefOptSeq       = DefOpt ';' < DefOpt ';' > .
DefOpt          = [ '?' ] DefWithAttr .
```

```
DefSeq          = DefWithAttr ';' < DefWithAttr ';' > .

SkeletonSeq     = SkeletonElem < ',' SkeletonElem > ';' .
SkeletonElem    = ElemID [ 'WITH' Contents ] .
Contents        = 'Nothing' / ElemID [ ExtStruct ] .

ExceptSeq       = Except ';' < Except ';' > .
Except          = [ 'EXTERN' ] [ FirstSec ] ExcTypeOrAttr ':'
                  ExcValSeq .
ExcTypeOrAttr   = ElemID / AttrID .
ExcValSeq       = ExcValue < ',' ExcValue > .
ExcValue        = 'NoCut' / 'NoCreate' /
                  'NoHMove' / 'NoVMove' / 'NoMove' /
                  'NoHResize' / 'NoVResize' / 'NoResize' /
                  'MoveResize' /
                  'NewWidth' / 'NewHeight' /
                  'NewHPos' / 'NewVPos' /
                  'Invisible' / 'NoSelect' /
                  'Hidden' / 'ActiveRef' /
                  'ImportLine' / 'ImportParagraph' /
                  'NoPaginate' / 'ParagraphBreak' /
                  'HighlightChildren' / 'ExtendedSelection' /
                  'ReturnCreateNL' / 'ReturnCreateWithin' .

ExtensRuleSeq   = ExtensRule ';' < ExtensRule ';' > .
ExtensRule      = RootOrElem [ LocAttrSeq ]
                  [ '+' '(' ExtensionSeq ')' ]
                  [ '-' '(' RestrictSeq ')' ]
                  [ 'WITH' FixedAttrSeq ] .
RootOrElem      = 'Root' / ElemID .

END
```

The P language

The P language is used to write presentation schemas, which define the graphical presentation rules to be applied to different classes of documents and objects. It is described here in the M meta–language.

```
PresSchema       = 'PRESENTATION' ElemID ';'
                 [ 'VIEWS' ViewSeq ]
                 [ 'PRINT' PrintViewSeq ]
                 [ 'COUNTERS' CounterSeq ]
                 [ 'CONST' ConstSeq ]
                 [ 'VAR' VarSeq ]
                 [ 'DEFAULT' ViewRuleSeq ]
                 [ 'BOXES' BoxSeq ]
                 [ 'RULES' PresentSeq ]
                 [ 'ATTRIBUTES' PresAttrSeq ]
                 [ 'TRANSMIT' TransmitSeq ]
                   'END' .

ElemID           = NAME .

ViewSeq          = ViewDeclaration
                   < ',' ViewDeclaration > ';' .
```

```
ViewDeclaration = ViewID [ 'EXPORT' ] .
ViewID          = NAME .

PrintViewSeq    = PrintView < ',' PrintView > ';' .
PrintView       = ViewID / ElemID .

CounterSeq      = Counter < Counter > .
Counter         = CounterID ':' CounterFunc ';' .
CounterID       = NAME .
CounterFunc     = 'RANK' 'OF' TypeOrPage [ SLevelAsc ]
                  [ 'INIT' AttrID ] [ 'REINIT' AttrID ] /
                  SetFunction < SetFunction >
                  AddFunction < AddFunction >
                  [ 'INIT' AttrID ] /
                  'RLEVEL' 'OF' ElemID .
SLevelAsc       = [ '-' ] LevelAsc .
LevelAsc        = NUMBER .
SetFunction     = 'SET' CounterValue 'ON' TypeOrPage .
AddFunction     = 'ADD' CounterValue 'ON' TypeOrPage .
TypeOrPage      = 'Page' [ '(' ViewID ')' ] /
                  [ '*' ] ElemID .
CounterValue    = NUMBER .

ConstSeq        = Const < Const > .
Const           = ConstID '=' ConstType ConstValue ';' .
ConstID         = NAME .
ConstType       = 'Text' [ Alphabet ] / 'Symbol' /
                  'Graphics' / 'Picture' .
ConstValue      = STRING .
Alphabet        = NAME .

VarSeq          = Variable < Variable > .
Variable        = VarID ':' FunctionSeq ';' .
VarID           = NAME .
FunctionSeq     = Function < Function > .
Function        = 'DATE' / 'FDATE' /
                  'DocName' / 'DirName' /
                  'ElemName' / 'AttributeName' /
                  ConstID / ConstType ConstValue /
                  AttrID /
                  'VALUE' '(' PageAttrCtr ','
                  CounterStyle ')' .
PageAttrCtr     = 'PageNumber' [ '(' ViewID ')' ] /
                  [ MinMax ] CounterID / AttrID .
CounterStyle    = 'Arabic' / 'LRoman' / 'URoman' /
                  'Uppercase' / 'Lowercase' .
MinMax          = 'MaxRangeVal' / 'MinRangeVal' .

BoxSeq          = Box < Box > .
Box             = 'FORWARD' BoxID ';' /
                  BoxID ':' ViewRuleSeq .
BoxID           = NAME .

PresentSeq      = Present < Present > .
Present         = [ '*' ] [ FirstSec ] ElemID ':'
                  ViewRuleSeq .
FirstSec        = 'First' / 'Second' .
```

```
PresAttrSeq      = PresAttr < PresAttr > .
PresAttr         = AttrID [ '(' [ FirstSec ] ElemID ')' ]
                    [ AttrRelation ] ':' ViewRuleSeq .
AttrID           = NAME .
AttrRelation     = '=' AttrVal /
                   '>' [ '-' ] MinValue /
                   '<' [ '-' ] MaxValue /
                   'IN' '[' [ '-' ] LowerBound '..'
                   [ '-' ] UpperBound ']' /
                   'GREATER' AttrID /
                   'EQUAL' AttrID /
                   'LESS' AttrID .
AttrVal          = [ '-' ] EqualNum / EqualText / AttrValue .
MinValue         = NUMBER .
MaxValue         = NUMBER .
LowerBound       = NUMBER .
UpperBound       = NUMBER.
EqualNum         = NUMBER .
EqualText        = STRING .
AttrValue        = NAME .

ViewRuleSeq      = 'BEGIN' < RulesAndCond > < ViewRules >
                   'END' ';' /
                   ViewRules / CondRules / Rule .
RulesAndCond     = CondRules / Rule .
ViewRules        = 'IN' ViewID CondRuleSeq .
CondRuleSeq      = 'BEGIN' < RulesAndCond > 'END' ';' /
                   CondRules / Rule .
CondRules        = CondRule < CondRule >
                   [ 'Otherwise' RuleSeq ] .
CondRule         = 'IF' ConditionSeq RuleSeq .
RulesSeq         = 'BEGIN' Rule < Rule > 'END' ';' / Rule .

ConditionSeq     = Condition < 'AND' Condition > .
Condition        = [ 'NOT' ] [ 'Target' ] ConditionElem .
ConditionElem    = 'First' / 'Last' /
                   [ 'Immediately' ] 'Within' [ NumParent ]
                                      ElemID [ ExtStruct ] /
                    ElemID /
                   'Referred' / 'FirstRef' / 'LastRef' /
                   'ExternalRef' / 'InternalRef' / 'CopyRef' /
                   'AnyAttributes' / 'FirstAttr' / 'LastAttr' /
                   'UserPage' / 'StartPage' / 'ComputedPage' /
                   'Empty' /
                   '(' [ MinMax ] CounterName CounterCond ')' /
                   CondPage '(' CounterID ')' .
NumParent        = [ GreaterLess ] NParent .
GreaterLess      = '>' / '<' .
NParent          = NUMBER.
CounterCond      = '<' MaxCtrVal / '>' MinCtrVal /
                   '=' EqCtrVal /
                   'IN' '[' ['-'] MinCtrBound '..'
                   ['-'] MaxCtrBound ']' .
PageCond         = 'Even' / 'Odd' / 'One' .
MaxCtrVal        = NUMBER .
MinCtrVal        = NUMBER .
EqCtrVal         = NUMBER .
MaxCtrBound      = NUMBER .
```

```
MinCtrBound      = NUMBER .

Rule             = PresParam ';' / PresFunc ';' .
PresParam        = 'VertRef' ':' HorizPosition /
                   'HorizRef' ':' VertPosition /
                   'VertPos' ':' VPos /
                   'HorizPos' ':' HPos /
                   'Height' ':' Extent /
                   'Width' ':' Extent /
                   'VertOverflow' ':' Boolean /
                   'HorizOverflow' ':' Boolean /
                   'MarginTop' ':' MarginWidth /
                   'MarginRight' ':' MarginWidth /
                   'MarginBottom' ':' MarginWidth /
                   'MarginLeft' ':' MarginWidth /
                   'PaddingTop' ':' PaddingWidth /
                   'PaddingRight' ':' PaddingWidth /
                   'PaddingBottom' ':' PaddingWidth /
                   'PaddingLeft' ':' PaddingWidth /
                   'BorderTopWidth' ':' BorderWidth /
                   'BorderRightWidth' ':' BorderWidth /
                   'BorderBottomWidth' ':' BorderWidth /
                   'BorderLeftWidth' ':' BorderWidth /
                   'BorderTopColor' ':' BorderColor /
                   'BorderRightColor' ':' BorderColor /
                   'BorderBottomColor' ':' BorderColor /
                   'BorderLeftColor' ':' BorderColor /
                   'BorderTopStyle' ':' BorderStyle /
                   'BorderRightStyle' ':' BorderStyle /
                   'BorderBottomStyle' ':' BorderStyle /
                   'BorderLeftStyle' ':' BorderStyle .
                   'LineSpacing' ':' DistOrInherit /
                   'Indent' ':' DistOrInherit /
                   'Adjust' ':' AlignOrInherit /
                   'Justify' ':' BoolInherit /
                   'Hyphenate' ':' BoolInherit /
                   'PageBreak' ':' Boolean /
                   'LineBreak' ':' Boolean /
                   'InLine' ':' Boolean /
                   'NoBreak1' ':' AbsDist /
                   'NoBreak2' ':' AbsDist /
                   'Gather' ':' Boolean /
                   'Visibility' ':' NumberInherit /
                   'Size'  ':' SizeInherit /
                   'Font' ':' NameInherit /
                   'Style' ':' StyleInherit /
                   'Weight' ':' WeightInherit /
                   'Underline' ':' UnderLineInherit /
                   'Thickness' ':' ThicknessInherit /
                   'Depth' ':' NumberInherit /
                   'LineStyle' ':' LineStyleInherit /
                   'LineWeight' ':' DistOrInherit /
                   'FillPattern' ':' NameInherit /
                   'Background' ':' NameInherit /
                   'Foreground' ':' NameInherit /
                   'Content' ':' VarConst .
PresFunc         = Creation '(' BoxID ')' /
                   'Line' /
```

```
                   'NoLine' /
                   'Page' '(' BoxID ')' /
                   'Copy' '(' BoxTypeToCopy ')' /
                   'ShowBox' /
                   'BackgroundPicture' ':' FileName /
                   'PictureMode' ':' PictMode .

BoxTypeToCopy    = BoxID [ ExtStruct ] /
                    ElemID [ ExtStruct ] .
ExtStruct        = '(' ElemID ')' .

Distance         = [ Sign ] AbsDist .
Sign             = '+' / '-' .
AbsDist          = IntegerOrAttr [ '.' DecimalPart ]
                   [ Unit ] .
IntegerOrAttr    = IntegerPart / AttrID .
IntegerPart      = NUMBER .
DecimalPart      = NUMBER .
Unit             = 'em' / 'ex' / 'cm' / 'mm' / 'in' / 'pt' /
                   'pc' / 'px' / '%' .

HPos             = 'nil' / VertAxis '=' HorizPosition
                   [ 'UserSpecified' ] .
VPos             = 'nil' / HorizAxis '=' VertPosition
                   [ 'UserSpecified' ] .
VertAxis         = 'Left' / 'VMiddle' / 'VRef' / 'Right' .
HorizAxis        = 'Top' / 'HMiddle' / 'HRef' / 'Bottom' .

VertPosition     = Reference '.' HorizAxis [ Distance ] .
HorizPosition    = Reference '.' VertAxis [ Distance ] .
Reference        = 'Enclosing' [ BoxTypeNot ] /
                   'Enclosed' [ BoxTypeNot ] /
                   'Previous' [ BoxTypeNot ] /
                   'Next' [ BoxTypeNot ] /
                   'Referred' [ BoxTypeNot ] /
                   'Creator' /
                   'Root' /
                   '*' /
                   BoxOrType .
BoxOrType        = BoxID /
                   [ '*' ] [ FirstSec ] ElemID /
                   'AnyElem' / 'AnyBox' /
                   'ElemWithAttr' AttrID .
BoxTypeNot       = [ 'NOT' ] BoxOrType .

Extent           = Reference '.' HeightWidth
                   [ Relation ] [ 'Min' ] /
                   AbsDist [ 'UserSpecified' ] [ 'Min' ] /
                   HPos / VPos .
HeightWidth      = 'Height' / 'Width' .
Relation         = '*' ExtentAttr '%' / Distance .
ExtentAttr       = ExtentVal / AttrID .
ExtentVal        = NUMBER .

MarginWidth      = InheritParent / 'Auto' / Distance .
PaddingWidth     = InheritParent / Distance .
BorderWidth      = InheritParent / 'Thin' / 'Medium' / 'Thick' / Distance .
BorderColor      = InheritParent / 'Transparent' / 'Foreground' /
```

```
                     ColorName .
BorderStyle        = InheritParent /
                     'None' / 'Hidden' / 'Dotted' / 'Dashed' / 'Solid' /
                     'Double' / 'Groove' / 'Ridge' / 'Inset' / 'Outset' .
InheritParent      = 'Enclosing' '=' / 'Creator' '=' .
ColorName          = NAME .

Inheritance        = Kinship  InheritedValue .
Kinship             = 'Enclosing' / 'GrandFather'/ 'Enclosed' /
                     'Previous' / 'Creator' .
InheritedValue     = '+' PosIntAttr [ 'Max' maximumA ] /
                     '-' NegIntAttr [ 'Min' minimumA ] /
                     '=' .
PosIntAttr         = PosInt / AttrID .
PosInt             = NUMBER .
NegIntAttr         = NegInt / AttrID .
NegInt             = NUMBER .
maximumA           = maximum / AttrID .
maximum            = NUMBER .
minimumA           = minimum / AttrID .
minimum            = NUMBER .

AlignOrInherit     = Kinship '=' / Alignment .
Alignment           = 'Left' / 'Right' / 'VMiddle' /
                     'LeftWithDots' .

DistOrInherit      = Kinship InheritedDist / Distance .
InheritedDist      = '=' / '+' AbsDist / '-' AbsDist .

BoolInherit        = Boolean / Kinship '=' .
Boolean            = 'Yes' / 'No' .

NumberInherit      = Integer / AttrID / Inheritance .
Integer            = NUMBER .

LineStyleInherit= Kinship '=' / 'Solid' / 'Dashed' /
                     'Dotted' .

SizeInherit        = SizeAttr [ 'pt' ] / Kinship InheritedSize .
InheritedSize      = '+' SizeAttr [ 'pt' ]
                          [ 'Max' MaxSizeAttr ] /
                     '-' SizeAttr [ 'pt' ]
                          [ 'Min' MinSizeAttr ] /
                     '*' PercentSizeAttr '%' /
                     '=' .
SizeAttr           = Size / AttrID .
Size               = NUMBER .
MaxSizeAttr        = MaxSize / AttrID .
MaxSize            = NUMBER .
MinSizeAttr        = MinSize / AttrID .
MinSize            = NUMBER .
PercentSizeAttr = PercentSize / AttrID .
PercentSize        = NUMBER .

NameInherit        = Kinship '=' / FontName .
FontName           = NAME .
StyleInherit       = Kinship '=' /
                     'Roman' / 'Italics' / 'Oblique' .
```

```
WeightInherit    = Kinship '=' /
                   'Normal' / 'Bold' .
UnderLineInherit= Kinship '=' /
                   'NoUnderline' / 'Underlined' /
                   'Overlined' / 'CrossedOut' .
ThicknessInherit= Kinship '=' / 'Thick' / 'Thin' .

FileName =         STRING .
PictMode =         'NormalSize' / 'Scale' /
                   'RepeatXY' / 'RepeatX' / 'RepeatY' .

VarConst         = ConstID / ConstType ConstValue /
                   VarID / '(' FunctionSeq ')' /
                   ElemID .

Creation         = Create [ 'Repeated' ] .
Create           = 'CreateFirst' / 'CreateLast' /
                   'CreateBefore' / 'CreateAfter' /
                   'CreateEnclosing' .

TransmitSeq      = Transmit < Transmit > .
Transmit         = TypeOrCounter 'To' ExternAttr
                   '(' ElemID ')' ';' .
TypeOrCounter    = CounterID / ElemID .
ExternAttr       = NAME .

END
```

The T language

```
TransSchema    = 'TRANSLATION' ElemID ';'
               [ 'LINELENGTH' LineLength ';' ]
               [ 'LINEEND' CHARACTER ';' ]
               [ 'LINEENDINSERT' STRING ';' ]
               [ 'BUFFERS' BufferSeq ]
               [ 'COUNTERS' CounterSeq ]
               [ 'CONST' ConstSeq ]
               [ 'VAR' VariableSeq ]
                 'RULES' ElemSeq
               [ 'ATTRIBUTES' AttrSeq ]
               [ 'PRESENTATION' PresSeq ]
               < 'TEXTTRANSLATE' TextTransSeq >
               [ 'SYMBTRANSLATE' TransSeq ]
               [ 'GRAPHTRANSLATE' TransSeq ]
                 'END' .

LineLength     = NUMBER .

BufferSeq      = Buffer < Buffer > .
Buffer         = BufferID [ '(' 'Picture' ')' ] ';' .
BufferID       = NAME .

CounterSeq     = Counter < Counter > .
Counter        = CounterID [ ':' CounterFunc ] ';' .
CounterID      = NAME .
```

```
CounterFunc    = 'Rank' 'of' ElemID [ SLevelAsc ]
                 [ 'Init' AttrID ] /
                 'Rlevel' 'of' ElemID /
                 'Set' InitValue 'On' ElemID
                       'Add' Increment 'On' ElemID
                       [ 'Init' AttrID ] .
SLevelAsc      = [ '-' ] LevelAsc .
LevelAsc       =  NUMBER .
InitValue      = NUMBER .
Increment      = NUMBER .
ElemID         = NAME .
AttrID         = NAME .

ConstSeq       = Const < Const > .
Const          = ConstID '=' ConstValue ';' .
ConstID        = NAME .
ConstValue     = STRING .

VariableSeq    = Variable < Variable > .
Variable       = VarID ':' Function < Function > ';' .
VarID          = NAME .
Function       = 'Value' '(' CounterID [ ':' Length ]
                           [ ',' CounterStyle ]  ')' /
                 'FileDir' / 'FileName' / 'Extension' /
                 'DocumentName' / 'DocumentDir' /
                 ConstID / CharString /
                 BufferID / AttrID .
Length         = NUMBER .
CounterStyle=    'Arabic' / 'LRoman' / 'URoman' /
                 'Uppercase' / 'Lowercase' .
CharString     = STRING .

ElemSeq        = TransType < TransType > .
TransType      = [ FirstSec ] ElemID ':' RuleSeq .
FirstSec       = 'First' / 'Second' .
RuleSeq        = Rule / 'BEGIN' < Rule > 'END' ';' .
Rule           = SimpleRule / ConditionBlock .
ConditionBlock= 'IF' ConditionSeq SimpleRuleSeq .
SimpleRuleSeq = 'BEGIN' < SimpleRule > 'END' ';' /
                 SimpleRule .

ConditionSeq  = Condition [ 'AND' Condition ] .
Condition      = [ 'NOT' ] [ 'Target' ] Cond .
Cond           = CondElem / CondAscend .
CondElem       = 'FirstRef' / 'LastRef' /
                 'ExternalRef' /
                 'Defined' /
                 'Alphabet' '=' Alphabet /
                 'ComputedPage' / 'StartPage' /
                 'UserPage' / 'ReminderPage' /
                 'Empty' /
                 ElemID /
                 'FirstAttr' / 'LastAttr' .
CondAscend     = [ Ascend ] CondOnAscend .
Ascend         = '*' / 'Parent' / 'Ancestor' LevelOrType .
LevelOrType    = CondRelLevel / ElemID [ ExtStruct ] .
CondRelLevel  = NUMBER .
CondOnAscend  = 'First' / 'Last' /
```

```
                  'Referred' /
                  [ 'Immediately' ] 'Within' [ NumParent ]
                                    ElemID [ ExtStruct ] /
                  'Attributes' /
                  AttrID [ RelatAttr ] /
                  'Presentation' /
                  PresRule /
                  'Comment' .
NumParent       = [ GreaterLess ] NParent .
GreaterLess     = '>' / '<' .
NParent         = NUMBER.
Alphabet        = NAME .
RelatAttr       = '=' Value /
                   '>' [ '-' ] Minimum /
                   '<' [ '-' ] Maximum /
                   'IN' '[' [ '-' ] MinInterval '..'
                            [ '-' ] MaxInterval ']' .
Value           = [ '-' ] IntegerVal / TextVal / AttrValue .
Minimum         = NUMBER .
Maximum         = NUMBER .
MinInterval     = NUMBER .
MaxInterval     = NUMBER .
IntegerVal      = NUMBER .
TextVal         = STRING .
AttrValue       = NAME .

SimpleRule      = 'Create' [ 'IN' VarID ] Object
                         [ Position ] ';' /
                  'Write' Object [ Position ] ';' /
                  'Read' BufferID [ Position ] ';' /
                  'Include' File [ Position ] ';' /
                  'Get'   [ RelPosition ] ElemID
                          [ ExtStruct ]
                          [ Position ] ';' /
                  'Copy'  [ RelPosition ] ElemID
                          [ ExtStruct ]
                          [ Position ] ';' /
                  'Use' TrSchema [ 'For' ElemID ] ';' /
                  'Remove' ';' /
                  'NoTranslation' ';' /
                  'NoLineBreak' ';' /
                  'ChangeMainFile' VarID [ Position ] ';' /
                  'RemoveFile' VarID [ Position ] ';' /
                  'Set' CounterID InitValue [ Position ] ';' /
                  'Add' CounterID Increment [ Position ] ';' /
                  'Indent' [ 'IN' VarID ] Indent [ Position ] ';' .

Indent          = 'Suspend' / 'Resume' / [ IndentSign ] IndentValue .
IndentSign      = '+' / '-' .
IndentValue     = NUMBER .

Object          = ConstID / CharString /
                  BufferID /
                  VarID /
                  '(' Function < Function > ')' /
                   [ 'Translated' ] AttrID /
                  'Value' /
                  'Content' /
```

```
                 'Comment' /
                 'Attributes' /
                 'Presentation' /
                 'RefId' /
                 'PairId' /
                 'FileDir' / 'FileName' / 'Extension' /
                 'DocumentName' / 'DocumentDir' /
                 [ 'Referred' ] ReferredObject .
Position       = 'After' / 'Before' .

ReferredObject= VarID /
                 ElemID [ ExtStruct ] /
                 'RefId' /
                 'DocumentName' / 'DocumentDir' .

File           = FileName / BufferID .
FileName       = STRING .

RelPosition    = 'Included' / 'Referred' .
ExtStruct      = '(' ElemID ')' .

TrSchema       = NAME .

AttrSeq        = TransAttr < TransAttr > .
TransAttr      = AttrID [ '(' ElemID ')' ]
                 [ RelatAttr ] ':' RuleSeq .

PresSeq        = PresTrans < PresTrans > .
PresTrans      = PresRule ':' RuleSeq .
PresRule       = 'Size' [ PresRelation ] /
                 'Indent' [ PresRelation ] /
                 'LineSpacing' [ PresRelation ] /
                 'Adjust' [ '=' AdjustVal ] /
                 'Justify' [ '=' BoolVal ] /
                 'Hyphenate' [ '=' BoolVal ] /
                 'Style' [ '=' StyleVal ] /
                 'Weight' [ '=' WeightVal ] /
                 'Font' [ '=' FontVal ] /
                 'UnderLine' [ '=' UnderLineVal ] /
                 'Thickness' [ '=' ThicknessVal ] /
                 'LineStyle' [ '=' LineStyleVal ] /
                 'LineWeight' [ PresRelation ] /
                 'FillPattern' [ '=' Pattern ] /
                 'Background' [ '=' Color ] /
                 'Foreground' [ '=' Color ] .

PresRelation   = '=' PresValue /
                 '>' [ '-' ] PresMinimum /
                 '<' [ '-' ] PresMaximum /
                 'IN' '[' [ '-' ] PresIntervalMin '..'
                          [ '-' ] PresIntervalMax ']' .
AdjustVal      = 'Left' / 'Right' / 'VMiddle' /
                 'LeftWithDots' .
BoolVal        = 'Yes' / 'No' .
StyleVal       = 'Roman' / 'Italics' / 'Oblique' .
WeightVal      = 'Normal' / 'Bold' .
FontVal        = 'Times' / 'Helvetica' / 'Courier' .
UnderLineVal   = 'NoUnderline' / 'UnderLined' /
```

```
                    'OverLined' / 'CrossedOut' .
ThicknessVal   = 'Thick' / 'Thin' .
LineStyleVal   = 'Solid' / 'Dashed' / 'Dotted' .
Pattern        = NAME .
Color          = NAME .
PresMinimum    = NUMBER .
PresMaximum    = NUMBER .
PresIntervalMin= NUMBER .
PresIntervalMax= NUMBER .
PresValue      = [ '-' ] PresVal .
PresVal        = NUMBER .

TextTransSeq   = [ Alphabet ] TransSeq .
Alphabet       = NAME .
TransSeq       = 'BEGIN' < Translation > 'END' ';' /
                 Translation .
Translation    = Source [ '->' Target ] ';' .
Source         = STRING .
Target         = STRING .
```

Character coding

Characters

The characters of the Latin alphabet follow the encoding defined in the ISO 8859−1 (ISO Latin−1) standard. The characters of the Greek alphabet follow the encoding defined by Adobe for its Symbol font (Adobe FontSpecific).

Characters whose octal code is greater than 0200 are written in the form of their octal code preceded by a backslash character (‘‘\’’). For example, the French word 'Résumé' is written `R\351sum\351`.

To the ISO 8859−1 encoding four characters with the following codes have been added:
`212`: line break
`240`: sticky space
`201`: thin space
`202`: en space

The `212` character is a ‘‘line break’’ character which forces a line break. The `240` character is a ‘‘sticky space’’, which cannot be replaced by a line break.

Symbols

The table below gives the codes for the symbols of Thot. Symbols can be used in presentation schemas constants and in transcoding rules of translation schemas. Each symbol is represented by a single character.

- `r`: a radical
- `i`: a simple integral
- `c`: a curvilinear integral
- `d`: a double integral
- `t`: a triple integral
- `S`: the summation symbol
- `P`: the product symbol
- `U`: the union symbol
- `I`: the intersection symbol
- `>`: a right arrow
- `<`: a left arrow
- `^`: an up arrow
- `V`: a down arrow
- `(`: an opening parenthesis
- `)`: a closing parenthesis
- `{`: an opening brace
- `}`: a closing brace
- `[`: an opening bracket
- `]`: a closing bracket

Graphical elements

The table below gives the codes for the graphical elements of Thot. These elements can be used in presentation schemas constants and in transcoding rules of translation schemas. Each graphical element is represented by a single character.

- a: a circle
- `A`: an open curve with an arrow head at the end
- `b`: a horizontal line along the lower side of the box
- `B`: an open curve
- `c`: an ellipse inscribed in the box
- `C`: a rectangle with rounded corners
- `D`: an open curve with two arrow heads
- `e`: The northwest/southeast diagonal of the box with an arrowhead at the bottom
- `E`: The southwest/northeast diagonal of the box with an arrowhead at the top
- `F`: an open curve with an arrow head at start
- `g`: a line from the origin of the box to its opposite corner
- `h`: a horizontal line as wide as the box and placed in its middle
- `l`: a vertical line on the left side of the box
- `L`: a lozenge

- `M`: an open broken line with two arrow heads
- `N`: an open broken line with an arrow head at start
- `o`: The southwest/northeast diagonal of the box with an arrowhead at the bottom
- `O`: The northwest/southeast diagonal of the box with an arrowhead at the top
- `p`: a polygon
- `P`: a rectangle with round corners and a horizontal bar at the top
- `Q`: an ellipse with a horizontal bar at the top
- `r`: a vertical line on the right side of the box
- `R`: a rectangle which is the shape of the box
- `s`: a closed curve
- `S`: an open broken line
- `t`: a horizontal line along the upper side of the box
- `U`: an open broken line with an arrow head at the end
- `v`: a vertical line as tall as the box and placed in its middle
- `V`: a down arrow as tall as the box and in its middle
- w: a segment (2 points)
- `W`: the upper right corner
- x: a segment (2 points) with an arrow head at the end
- `X`: the lower right corner
- y: a segment (2 points) with an arrow head at the end
- `Y`: the lower left corner
- z: a segment (2 points) with an arrow head at the end
- `Z`: the upper left corner
- `space`: a transparent element
- `^`: an up arrow as tall as the box and in its middle
- `>`: a right arrow as long as the box's width and in its middle
- `>`: a left arrow as long as the box's width and in its middle
- `/`: The southwest/northeast diagonal of the box
- `\`: the northwest/southeast diagonal of the box
