Grammars++ for Modelling Information in Text

Airi Salminen[1] and Frank Wm. Tompa[2]

ABSTRACT

Grammars provide a convenient means to describe the set of valid strings in a language, and thus they seem natural for describing the set of valid instances in a text database. It is well-known that a given language can be described by many grammars, and similarly text database designers have a choice of grammar for specifying valid documents. This flexibility can be exploited to provide information modelling capability by designing productions in the grammar to represent entities and relationships of interest to the database applications. Additional constraints can be specified by attaching predicates to selected non-terminals in the grammar.

In this paper, we formalize and illustrate the use of extended grammars for text databases. When used for database definition, grammars can provide the functionality that users have come to expect of database schemas. Extended grammars can also be used to specify database manipulation, including query, update, view definition, and index specification.

1. Introduction to text databases

As electronic text repositories grow, there is an increasing need to manage the text as a database. This, in turn, necessitates a model of the information stored in order that database operators can be used effectively for querying, transforming, updating, and validating the text. A data model describing a text database will provide view designers and end-users the capability to direct their attention to relevant information fragments, to formulate meaningful queries, and to specify the amount of context to include with extracted data.

Unlike conventional databases, the data in a text database is not intended to represent an enterprise directly (Figure 1.1). Instead it represents a collection of documents, which, in turn, captures the information embodying the enterprise. What distinguishes a text database from an alternative database is that the data model must represent the text that exists, rather than an idealized version of the real world.

---

[1] Department of Computer Science and Information Systems, University of Jyväskylä, Finland.

[2] Department of Computer Science, University of Waterloo, Canada.

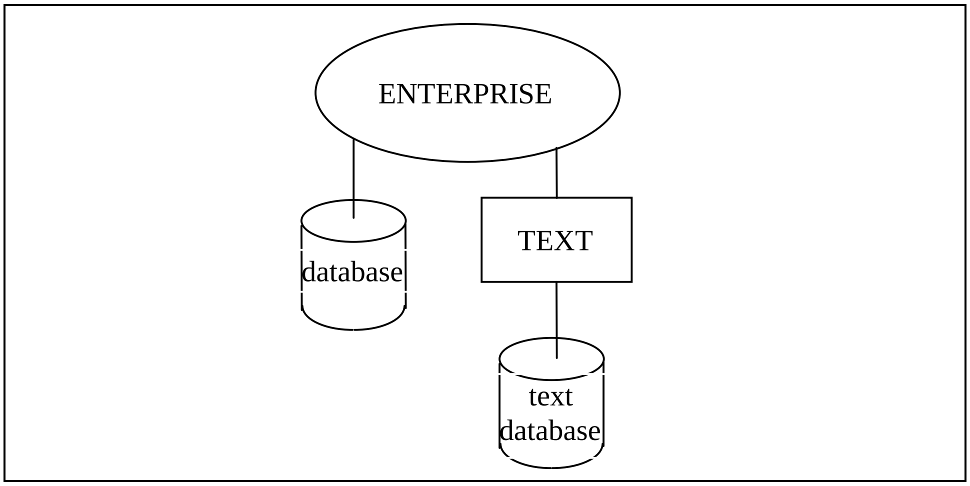

Figure 1.1. Contrast between text database and conventional database.

Consider the related definitions of "congress" and "conference" from the *Oxford English Dictionary*:

conference *sb*. 6

> A formal meeting for consultation or discussion; *e.g.* between the representatives of different sovereign states, the two Houses of Parliament or of Congress, the representatives of societies, parties, etc.

congress *sb*. 6. a.

> A formal meeting or assembly of delegates or representatives for the discussion or settlement of some question; *spec.* (in politics) of envoys, deputies, or plenipotentiaries representing sovereign states, or of sovereigns themselves, for the settlement of international affairs. Also an annual or periodical meeting or series of meetings of some association or society, or of persons engaged in special studies, as *Church Congress*, the name of annual meetings of the Church of England for discussion; *Social Science Congress, Congress of Orientalists*, etc.

Database experts cannot discard the text of the electronic *Oxford English Dictionary*, replacing it by a normalized word list with stylized and abstracted definitions, and expect to maintain the full information content. Similarly the text database maintaining a collection of laws and statutes cannot be replaced by some other database that captures its spirit but not its letter.

Whereas database modelling traditionally involves the identification of entities and relationships [55], text modelling is based on the rich history provided by the field of formal languages where text strings are characterized by grammars [4]. The information needs for users of a text collection can be extremely diverse, some in terms of external entities and relationships and others in terms of the text itself. Therefore effective use of text databases relies on the ability to carry out both modelling tasks simultaneously.

For example, consider a collection of newspaper articles in electronic form. A researcher of politics might be interested in news articles published between given dates and talking about relationships between Canada and Finland. A linguist might be interested in which way some word is used by the writers of articles, in which sections it is used, and when the use of the word first appeared. A sports editor might want to find the Olympic marathon winners and their records. A journalism researcher might want to find how many AP newswire items are used in various newspapers. For large collections, the retrieval system should offer powerful specification capabilities for these users to describe the portion of text in which they expect the needed information to be found such that not too much extra reading is required.

A model for semi-structured information in text databases[3] should comprise text, structure, and operations. Text consists of successive symbols of an alphabet and may constitute sentences of some natural language. The structure relates text pieces by some criterion to other text pieces or to some other types of data. The operations must be able to access and manipulate individual natural language elements ("words"), the sequential placement of those elements, the repeated occurrence of any elements, the structural units containing those elements, and the relationships among those structural units. Operations may be used to define constraints on text, to specify subsets to be retrieved, to arrange the results of retrieval and provide browsing facilities over those results, and to update instances of text.

In order to impose some constraints on collections of documents, or to recognize and exploit existing commonality, models have been devised to provide structure definitions. Such definitions, predominantly in the form of context-free grammars, serve the role of schema declarations for a text database. For example, SGML and XML offer such a data definition language for structured text [7, 22]. Text models based on grammars have richer data modelling capabilities than is possible without the support of data definition languages.

This paper introduces constraining grammars, which are specified by adding boolean conditions to the productions of a grammar. We show that constraining grammars extend the modelling capability of simple context-free grammars. They can serve as a uniform means to specify text operations, both for the purposes of validity checking as part of data definition and for data access as part of data manipulation. This paradigm has been effective in text filtering [33], specifying a full-text retrieval language [50], defining index structures [49, 53] and defining hypertext access to structured text [51].

The remainder of the paper is organized as follows. We discuss related work in Section 2. Section 3 provides a brief overview of the use of a grammar to describe text in terms of meaningful entities as well as merely delineating a set of strings. Sections 4 and 5 extend grammars to provide for the specification of further constraints by attaching predicates to non-terminal symbols. In Section 6 the extended grammar facilities are used as the basis of a fuller data manipulation language. We conclude with some comments on expressiveness and

---

[3] We use the term "text databases" to refer to databases containing primarily text, although they may also include some highly formatted data (such as tables) or multimedia data (*cf.* SGML).

efficiency. Throughout the paper, concepts and facilities are illustrated in terms of simple document databases.

2. Related work

Loeffen surveys the data types and operations supported by several text models and systems [35]. The most common text models treat text as a linear stream of characters with a superimposed structure denoting logical and/or physical segments (*cf.* [3, 12, 13, 24, 30, 39]). A more detailed analysis of the structuring mechanisms and query operations of some text models is summarized in Navarro and Baeza-Yates' taxonomy [39]. Sacks-Davis *et al.* characterize a classification of queries needed in SGML conformant database systems [47], including the classic capabilities of information retrieval systems embodied in text indexing and ranking. Kuikka presents another survey, which in addition covers text transformations and the production of structured documents by syntax-directed editing [31]. In this section we examine a few proposals in greater detail.

2.1. Extending the relational and object-oriented models

One approach in text modelling has been to extend the relational or object-oriented models to support textual objects and text operations. This approach has been taken, for example, in the TDM model [19] and in Atlas [48], extending the relational data model to encompass nested relations. In this approach, the data structuring is always systematic and explicitly defined by a schema. The use of the models to manage information stored in files, for example, as SGML documents, requires the specification of a transformation from the files to the database. Similarly, SGML has been mapped to the O_2 database management system [13], and the O_2 data model and query processor extended by new operations to manage hierarchic data efficiently [2]. OSQL was similarly extended to text as well [57].

A benefit of such an approach is that it supports integrated management of information stored in documents and in traditional databases. Furthermore, conventional database operations are made available for manipulating the data in documents. From the point of view of text and document management, the major problem of this approach is that it requires parallel understanding of two data models. The information structures are defined by document architectures whereas the operations are defined in the conventional database model. This also complicates the design of user interfaces for document management.

A different approach to the integration of text and relational data is to extend the set of operators in SQL to accommodate structured text [6, 8, 18]. It is not based on transformations between documents and relational databases, but on the coexistence of data repositories for two types of data and the definition of a combined query language for them. Unlike in other approaches, the text remains intact as the authoritative repository of text data. Although the text matching and extraction operators are now more closely aligned to the text model, users must still understand the relational data model as well to formulate complete queries.

2.2 Models for semistructured information

The problems caused by the management of irregularly structured information [1] has motivated the design of the Lore database management system [38]. Lore's data model is the Object Exchange Model (OEM), in which the data is represented by directed graphs with labelled edges, and a similar model underlies the UnQL language [10]. There is no fixed schema in a Lore database, which makes it suitable for unconstrained variability, such as for data stored in the World Wide Web. Queries in the Lorel query language are expressed by path expressions supporting specification of patterns for semistructured information. As a novel feature Lore includes an external data manager which enables Lore to import data from external sources on demand during query execution.

Text may be considered to be similarly semistructured, and some schema-less text models have been proposed. Such models superimpose arbitrarily overlapping collections of regions over the linear stream of characters in a text, and do not insist that the regions form a systematic structuring of the text. As examples of this approach, consider the GCL text algebra [11, 14, 17], the list-structure algebra [15], the PAT text algebra [24, 50] and the partial order model [44]. Models for "hypertexts" extend the hierarchic structure to more general graphs (as in Lore), allow for multiple hierarchies [39], or add the graph structure as another layer to the hierarchic structure [20].

Such implicit text structuring has its benefits: the models require simple collections of operations, a single query can be used to retrieve information from heterogeneously structured documents, structures can be flexibly changed without concern about constraints defined in a schema, and lightweight implementations are possible. A problem is that the design of semantically meaningful operations may be difficult.

2.3 Grammar-based approaches

As for more conventional databases, explicit, schema-like data definitions can help users in several ways. They provide bases to formulate

(a) constraints for data input and thus for validity checking,

(b) queries and text editing operations,

(c) meaningful views,

(d) text transformations,

(e) query optimization strategies, and

(f) presentations of documents and of query results for subsequent browsing or other processing.

This functionality can be partially supported in schema-less databases by dynamic "structural summaries" as provided for Lore databases through DataGuides [23].

Several models incorporating grammar-based data definition capabilities have been proposed [25, 26, 36]. In fact, some of these models are hybrid models: text structure is defined by a grammar but the text operations may also be applied to text having no explicit grammar. Grammar-based approaches are also common for text transformation systems [5, 16, 21, 27, 32, 37, 42]. The purpose of our model is to extend the capabilities offered by context-free grammars as schemas. Compared to other approaches, the important and novel feature in our model is that it offers a uniform way for specifying all tasks (a) through (f) listed above.

Our model is intended to support the specification of template-based management of hierarchic documents without binding the model to a specific system or document notation. Hypertext structures can also be defined as another layer, as described elsewhere [49, 51, 53], or by means of HyTime, HTML, or XLink. We do not propose a fixed collection of operations but rather an extensible framework, within which boolean operations can be defined to provide specification capabilities. Extensibility is especially important in multimedia environments: figures, video, and audio, for example, have important properties, which should be available to the users in specifying criteria for some needed information.

3. Context-free grammars and parse trees as models of text

Context-free grammars are well-suited for describing typical features of text: hierarchic structure, order, optionality, alternatives, and recursive structures. The basic notions of such grammars are well-known (see for example, [4]) and briefly, but rigorously, reviewed here to introduce the terminology and notation we require.

Definition. A *context-free grammar* is a 4-tuple (A, N, P, s), where

A is a set of *terminal symbols*,

N is a set of *non-terminal symbols*,

P is a set of *productions*, and

s is a distinguished non-terminal in N, called the *start symbol*.

Given a grammar G, the notation $G(t)$ denotes the identical grammar but having start symbol t in place of s.

The productions in P are of the form $t ::= \alpha$, where the left side is a non-terminal symbol and a production with left side t is called a *t-production*. We adopt an extended form of context free grammars, known as regular right part grammars [34], to model common text structures more directly. In such grammars, the right side is a regular expression over $A \cup N$, representing a set of *variants* that can be obtained by the following procedure:

1. Starting with α, insert parentheses as needed to reflect the decreasing precedence of operations as shown:

iteration and optionality	unary (postfix) operations	+ * ?
catenation	binary operation	(no explicit operator)
alternation	binary (infix) operation	|

2. Repeatedly process, as follows, until all explicit operators are deleted.

(i) An iteration denoted (β)+ is replaced by one or more copies of its operand (*i.e.*, β, ββ, βββ, etc.).

(ii) An iteration (β)* is replaced by zero or more copies of its operand.

(iii) An optional value (β)? is either replaced by its operand or deleted.

(iv) An alternation (β_1 | β_2) is replaced by one of its operands (which may be empty).

3. Finally, delete all remaining parentheses.

Note that if the right side of a production has no explicit operators and no parentheses, then the only variant is the right side itself.

The example grammar shown in Figure 2.1 represents the text structure for a collection of papers. The start symbol of the grammar is Papers, and the structure for a single paper follows the example used by Macleod [36].

(1) Papers ::= Paper*

(2) Paper ::= Front Body Back

(3) Front ::= Title Author+ Location Abstract

(4) Abstract ::= Paragraph+

(5) Paragraph ::= Sentence+

(6) Body ::= Section+

(7) Section ::= SectionHeading (Paragraph+ | Paragraph* SubSection+)

(8) SubSection ::= SectionHeading Paragraph+

(9) Back ::= Citation+

(10) Citation ::= Author* Title Source Year Pages?

Figure 2.1. Productions describing the structure of a collection of papers.

We restrict grammars for text databases such that each non-terminal symbol appears exactly once as the left side of a production (*i.e.*, for each non-terminal *t*, there is exactly one *t*-production); this restriction incurs no loss in specification power. For convenience, we assume an implicit non-terminal symbol Word, and a corresponding production for which the right side is an iteration over individual terminal symbols as alternatives. Terminal symbols explicitly specified in the examples are enclosed in quotation marks. Furthermore, for each non-terminal t other than Word, in the absence of an explicit production for t, we assume by default the production t ::= Word+.

A context-free grammar defines a formal language, *i.e.*, a set of strings, by specifying the symbols that can be used in the strings and the ways these symbols can be arranged. In text databases, the strings *together with their structure*, in the form of derivation or parse trees, constitute the instances of interest [25, 26].

Definition (adapted from Aho and Ullman [4] for regular right part grammars). A labelled ordered tree D is a *parse tree* for a context-free grammar $G = (A, N, P, s)$ if

(1) The root of the tree is labelled s.

(2) If $D_1, ..., D_k$ ($k \geq 1$) are the subtrees of the direct descendants of the root, and the root of D_i is labelled X_i, then $X_1 ... X_k$ is a variant of the right side of the s-production in P. D_i must be a parse tree for $G(X_i) = (A, N, P, X_i)$ if X_i is a non-terminal, and D_i is a single node labelled X_i if X_i is a terminal.

(3) Alternatively, if the root has no descendants, then a variant of the right side of the s-production in P is empty.

The *language generated by G* is the set of strings that result from concatenating the terminal symbols that label leaves of a parse tree for G in left-to-right order.

Figure 2.2 shows a parse tree corresponding to the grammar in Figure 2.1.

In our work we are not concerned with the way a parse tree is created. It may be created by parsing a given string, by syntax-directed editing, or by a transformation from an earlier parse tree. Neither are we concerned about ambiguity in the grammar (*cf*. [9]). However, we do require an unambiguous correspondence between non-terminal symbols in the grammar and the nodes of a corresponding parse tree defined as follows.

Because we have exactly one production in P for each non-terminal in N, any node labelled by t in the parse tree corresponds to the occurrence of t on the left side of the t-production. We define the function $\mathcal{L}$ to map a node in a parse tree to the non-terminal on the left side of the corresponding production in P. The children of a node labelled by t must have labels that match a variant of the right side of the t-production, α. This induces a correspondence between the child nodes and occurrences of non-terminal symbols in α (via the procedure producing the variant from α). To resolve ambiguity, we number repeated occurrences of non-terminal symbols in α, starting with 1, and use integer superscripts in the node labels of associated parse trees to indicate which occurrence corresponds to each node (with default 1). We define the function $\mathcal{R}$ to map a non-root node to the occurrence of a non-terminal symbol on the right side of some production in P. Finally we define the inverse function *Nodes* to map occurrences of non-terminal symbols in the productions of P to sets of corresponding nodes in a parse tree: $Nodes(v) = \{n \mid \mathcal{L}(n) = v \vee \mathcal{R}(n) = v\}$[4].

[4] For simplicity, we omit explicit mention of the grammar and parse tree in denoting the functions.

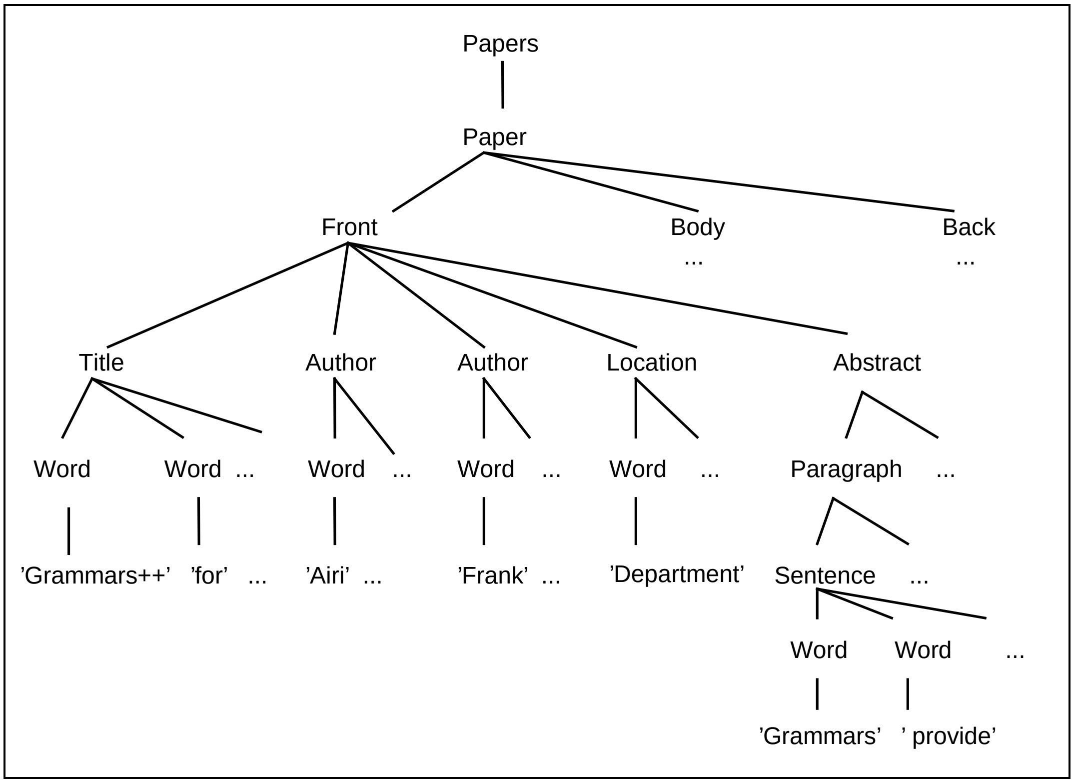

Figure 2.2. A parse tree corresponding to the grammar in Figure 2.1.

It is well known that a language can be generated by many different grammars. We capitalize on the flexibility in choice of grammar to define a text database that models meaningful entities as strings and trees. Therefore grammar design is concerned not only with describing valid strings but also characterizing meaningful structures. Because of parallels with data types in other database applications, we call the non-terminal symbols in N *text types*, and henceforth we use *non-terminals* to denote occurrences of non-terminal symbols in P. We define the function *Type* to map non-terminals in P to corresponding text types in N.

Let $G = (A, N, P, s)$ be a context-free grammar, t a text type in N, Y a parse tree for G and x a node in Y labelled by t. The subtree X with node x as its root is the *content* of x, and the string produced by concatenating the terminal symbols of X (from left to right) is the *value* of x. Thus, each text type represents simultaneously a set of subtrees (the contents of nodes labelled by t) and a set of substrings (the values of those nodes) in one parse tree, as well as the space of contents and values in all possible parse trees for the grammar.

Given a grammar G, not all substrings in the language generated by G correspond to entities in the data model represented by G. For example, Macleod defines an alternative type of paper [36] with production

MyType ::= Title Abstract Section* Citation+

He has chosen for such papers not to represent the units Front, Body, and Back explicitly. The substrings formed by concatenating the title and abstract or by concatenating multiple sections or multiple citations do not correspond to entities in the data model; there are no subtrees corresponding to these strings.

Similarly not all subtrees in a parse tree for G correspond to entities in the data model represented by G. Consider the following prototypical forms of production:

(1) $t ::= t_1$

(2) $t ::= t_1\ \ t_2\ \ \ldots\ \ t_n$

(3) $t ::= t_1+$

(4) $t ::= t_1*$

(5) $t ::= t_1 \mid t_2 \mid \ldots \mid \mathrm{t}_n$

(6) $t ::= t_1?$

Forms (2) through (4) are used to characterize aggregations of text entities: either corresponding to compound objects or to ordered collections of homogeneous objects. Forms (1) and (5) characterize renaming of entities, as does form (6) in the event that the instance is not empty. We therefore distinguish between nodes that represent distinct entities of a text database and those that rename entities, as follows:

> Let p and c be nodes labelled by text types in a parse tree, such that c is a child of p, and let the corresponding production in the grammar be $t ::= \alpha$. The node c is regarded as renaming an entity, not as a separate entity itself, if it is the only child of p and it corresponds to a variant of α obtained without processing iterations (*i.e.*, the label on c is obtained from α directly or through processing operators for alternation and optionality only). Nodes such as c are called *renaming nodes*; and all other nodes labelled by text types are called *parts*.

We define the function *Part* to map nodes labelled by text types to their corresponding parts, as follows: If node x is a part, then $Part(x) = x$. Otherwise, node x is a renaming node, and $Part(x) = Part(y)$, where y is the parent of x.

For example, consider the production

Publication ::= Book | Periodical

All Book or Periodical instances that correspond to the right side of this production are also instances of Publication. The nodes corresponding to either non-terminal on the right side of the production serve as renaming nodes of the parts corresponding to the left side of the production.

Definition. Let v be a non-terminal in a production of a grammar and $t = Type(v)$ be its text type. If x is a part in a corresponding parse tree, such that $Nodes(v)$ includes either x or a renaming node of x, then part x is said to be *of type t*.

Observation. In the presence of renaming nodes, parts may be of more than one type.

Definitions. If x and x' are two parts, not necessarily distinct, such that x' is a node in the content of x, we say that part x *contains* part x', that x' is *contained in x*, and that x' is *in the context of x*. Part x' is *properly contained in x* if x' is not x itself. Part x' is a *direct component of x* if it is properly contained in x and not properly contained in any part x'' which is also properly contained in x.

Observations. If node x is a descendant of node y, then either $Part(x) = Part(y)$ or $Part(x)$ is properly contained in $Part(y)$. In the absence of renaming nodes, the children of a part represent the direct components of that part.

4. Properties

In Section 3, grammars were used to confine text instances to fit into certain structures. For example, Author must conform to a given syntax and can only appear within certain contexts within Papers. In this section, we introduce the basis of a more powerful constraining mechanism that can be used to improve our modelling capacity by further limiting matching instances.

In structured text, information is captured in the types of parts, in the values of parts, and in the structural relationships among parts as reflected by their contents. Text operations should provide the functionality to test information in any of the three categories.

To begin, we define a boolean function, or *property*, for each of the text types t of a grammar that tests if a part of a parse tree is of type t. For example, in Figure 2.2, the property Papers is true for the root node and the property Paper is true for its only child; the property Front is not true for either part.

In general, properties can be arbitrary predicates that may be applied to any part, and they will be used to define text operations that behave as constraints. The properties testing the values and contents of parts are written in the form $t\{q\}$. In such a property, q specifies additional constraints that must be met. Regardless of those constraints, the property $t\{q\}$ is false for all parts that are not of type t.

Figure 4.1 shows the properties we define in this section and use in later examples. In the properties, t is a text type defined by a context free grammar, r is a character string, p is a property, n_1 and n_2 are integers, and q_i is a symbol sequence (a string of types, character strings, numbers, and operator symbols) such that $t\{q_i\}$ is a property. The properties (P2) through (P4) test a part independently of its context, properties (P5) and (P6) test a part with respect to its context, and properties (P7) through (P9) combine constraints with boolean operations. We do not claim that these properties are the only ones that should be defined, but they form a solid basis for a fully-developed language.

In Section 5 we show how properties may be used in a data definition language to obtain constraining productions from productions of a grammar, and we introduce a notation for defining the context within which a property is to be tested. We describe in Section 6 how the notation can also be used as a data manipulation language.

(P1)	t	part is of type t
(P2)	$t\{ =r \}$	… and value matches string r
(P3)	$t\{ p \}$	… and contains part with property p
(P4)	$t\{ r \}$	… and contains part with value r
(P5)	$t\{ n_1 .. n_2 \}$	… and lies in range n_1 .. n_2
(P6)	$t\{ =p \}$	… and value equals another part with property p
(P7)	$t\{ \neg q_1 \}$	… and property negation
(P8)	$t\{ q_1 \,\&\, q_2 \,\&\, ... \,\&\, q_n \}$	… and property conjunction
(P9)	$t\{ q_1 \mid q_2 \mid ... \mid q_n \}$	… and property disjunction

Figure 4.1. Universal properties.

4.1. Properties testing a part independently of context

In all text search languages there are capabilities to specify conditions to be met by character string values of textual parts. The property $t\{ =r \}$ tests whether the value of a part matches the character string r (as well as being of type t). Thus, for example, the property Title{="Mind Your Grammar"} tests whether a part is a title having the value specified. In all examples in this paper, we restrict this property to string equality, but any string pattern language (such as regular expressions) could be encoded within r to provide more general functionality.

For testing the containment hierarchy we will use the property $t\{p\}$ where p is itself a property. Such a property is true for a part x if it contains a part y (possibly x itself) for which property p is true.[5] For example, consider a parse tree conforming to the following grammar:

[5] In our earlier papers the property $t\{p\}$ was written as t\{contains p\}. In fact we previously defined three other properties for testing the containment hierarchy: t\{is p\}, t\{in p\}, and t\{where p\}, all of which have been useful [49, 50, 51]. We have since introduced filters (see Section 5), a mechanism to build combinations of conditions that allows us to reduce the set of universal properties without loss of expressivity.

Staff ::= Employee+

(E) Employee ::= Name Address Phone?

Name ::= Forename+ Surname

The property Employee{Phone} is true for an employee who has a phone. The property Employee{Surname{="Jones"}} is true for an employee with surname Jones.

The property $t\{r\}$ tests both the content and the value of a part. The property is true for a part x if it contains a part (not necessarily distinct from x) whose value matches r. Continuing with the above example, Employee{"Jones"} is true for an employee with forename or surname Jones, or one having the address 1728 Jones Street, for example (since Jones is then the value of a contained Word-part). In fact, since all text types are ultimately sequences of words, Employee{"Jones"} holds for any employee containing the word Jones in any sub-field. Note, however, that the property Employee{"David Jones"} is true for an employee named David Jones (the string matches the subpart Name), but not an employee named David Lee Jones, James David Jones, nor David Jones Smith, nor one living at 1728 David Jones Street: the test is for contained *parts* represented in the text model, not for arbitrary sequences of words.

4.2. Properties testing a part with respect to its context

Text defined by a grammar is an ordered hierarchy and the order of parts may be an important criterion to test. The property $t\{n_1 .. n_2\}$is defined for testing the position of a part with respect to other parts of the same type. The property is defined for a part x as follows:

> Given a context defined by a part c, assume there are m parts of type t enumerated in preorder and let x be the i^{th} such part. If n_1 and n_2 are both positive, the property $t\{n_1 .. n_2\}$ is true if $n_1 \leq i \leq n_2$. If n_1 and n_2 are both negative, the property is true if $n_1 \leq i - m - 1 \leq n_2$. If n_1 is positive and n_2 is negative, the property is true if $n_1 \leq i$ and $i - m - 1 \leq n_2$. For all other parts, the property is false. The notation $t\{n\}$ is short for $t\{n..n\}$.

For example, Figure 2.1 defines SubSection to be

SubSection ::= SectionHeading Paragraph+

Within the context of a single subsection, the property Paragraph{1..5} is true for each of the first five paragraphs, the property Paragraph{-1} is true for the last paragraph, and the property Paragraph{2..-3} is true for all but the first and the last two paragraphs. By altering the context (as described in Section 5) these same properties would test the location of a paragraph relative to a section, a paper, or any other containing part.

The property $t\{ =p \}$ provides a general capability to compare the values of parts.[6] This mirrors the functionality provided in relational databases by the equijoin operation (to compare attribute values among relations) and in hypertexts (following a cross-reference by

---

[6] In our earlier papers the property was written as $t\{$value equals part $p\}$.

matching a source to a target). The property is true for a part if its value is equal to another *distinct* part (within the given context) for which property p is true.

For example, consider a parse tree defined by the productions (E) above. Within the context of a single employee, the property Surname{=Forename} is true for those Surname parts that have a value that occurs also as a forename for the same employee (*e.g.*, Lee William Lee). Within the context of the whole staff, Surname{=Surname} is true for all Surname parts such that some *other* employee shares the same surname. Again, the mechanism for specifying a context is described in Section 5.

This part-to-part matching operation is not expressible by context-free grammars nor available in most text manipulation languages. Its expressive power adds considerably to a model's ability to specify validity constraints and retrieval requests.

4.3. Boolean combinations of constraints

If a text type t admits constraints $\{q_1\}$, ..., $\{q_n\}$, then new properties may be created with logical constructors ¬ (not), & (and) , and | (or). The property $t\{\neg q_1\}$ is satisfied by a t part if $t\{q_1\}$ is not. The property $t\{q_1 \,\&\, q_2 \,\&\, ... \,\&\, q_n\}$ is satisfied if all properties $t\{q_1\}$, ..., $t\{q_n\}$ are satisfied, and the property $t\{q_1 \mid q_2 \mid ... \mid q_n\}$ is satisfied if at least one property $t\{q_1\}$, ..., $t\{q_n\}$ is satisfied. For example, the property

Employee{Surname{="Smith"} & ¬Phone}

tests whether a given part is an Employee with surname Smith and containing no component of type Phone.

4.4. Special properties

So far we have defined predicates obtainable from the types defined in *any* grammar. In addition to such universal properties, additional properties may be defined so that they are available for specific needs. For example, more flexible pattern matching might be useful for information retrieval applications, part matching based on relations other than equality might be useful for some database applications, and being able to meet any k of n conditions might be useful for expressing partial matches. A property-definition capability allows application designers to define either grammar-specific or type-specific operators that attach semantics to the types.

As for universal properties, if a special property is defined via constraint $\{q\}$ to be used for a grammar G, it must be defined such that for any specific type t in G, the property $t\{q\}$ is defined (yielding the value true or false). To avoid ambiguity, each new property must be syntactically distinct from the universally defined properties and from other special properties.

For example, suppose we wish to define properties for arithmetically comparing the values of parts considered as numbers. The grammars for which the properties are defined contain the following productions:

```
Digit ::= ′0′ | ′1′ | ′2′ | ′3′ | ′4′ | ′5′ | ′6′ | ′7′ | ′8′ | ′9′
Number ::= Digit+
```

Using any suitable language for defining abstract data types, we may now define the constraint $\{< n\}$, where n is a value of type Number. With such a definition, for any type t, for any Number value n, and for any t part, the property $t\{< n\}$ yields true iff the value of the part is a Number value and its arithmetic value is less than n. For example, the values of Year parts could be tested by the operation Year{<1990}.

5. Constraining grammars and filters

We need to express schemas effectively to improve our ability to model stored information and users' information needs. Grammars provide the basis of our approach, allowing us to specify many structural constraints through productions. In Section 5.1 we introduce constraining grammars to extend the modelling power by using properties in place of non-terminals in productions. Constraining grammars define the context within which properties are tested and admit non-context-free constraints. In Section 5.2, we introduce transient text types to link sequences of constraining grammars to form filters, which provide a mechanism for building complex matching and transformation specifications.

5.1 Constraining grammars

We start with a model of text defined by a grammar as described in Section 3; thus text entities are identified. We next refine the model by attaching properties (as described in Section 4) to add further constraints to those already captured by productions in the grammar. We assume an interface through which productions of the base grammar can be readily extracted and annotated, perhaps through a cut-and-paste or drag-and-drop GUI or through a syntax-directed editing environment [33].

A constraining grammar includes constraining productions obtained from the productions of a given base grammar. The start symbol of the constraining grammar restricts the evaluation context for the properties in the constraining productions. In addition, productions in a constraining grammar may contain annotations that provide names for parts matched by the grammar.

Definition. Given a base grammar $G = (A, N, P, s)$, a text type t from N, and a set of symbols D such that $D \cap (A \cup N) = \emptyset$, a *constraining grammar for t* is a 4-tuple $G'(t) = (G, D, P', t)$, where the productions of P' are constraining productions (see below) obtained from the productions of P with annotations that are symbols drawn from D. The text type t is called the *context type* of the constraining grammar.

Definition. A *constraining production* is obtained from a production of a base grammar G as follows: if t is a non-terminal in the original production and $t\{q\}$ is a property defined for the grammar, then t may be replaced by $t\{q\}$. Furthermore, using set D as defined above, we may annotate any non-terminal using a symbol drawn from D: such an annotation is written like a constraint (included in braces after the type name) but preceded by two colons.

For example, given the production:

```
Paper ::= Front Body Back
```

we can specify the constraining production:

```
Paper{:: related} ::= Front{"grammars" :: summary} Body Back{Author{"Aho"}}
```

(This identifies a paper and its front matter if the front matter uses the word grammars and the back matter includes a citation by Aho.)

Given a parse tree corresponding to some grammar, we must define how parts in the tree are matched to a constraining form of the grammar. We start by defining the role of constraining productions.

Definition. Let x be a part of type t in a parse tree corresponding to base grammar G, and let ρ be a constraining production obtained from the t-production in G by replacing non-terminals t_i with properties $t_i\{q\}$. Given a part c to serve as a context, *part x satisfies production ρ in context c* if

(i) part c contains part x,

(ii) the property on the left side of ρ is true for x within context c, and

(iii) for each node y such that y is a child of x, if $\mathcal{R}(y) = t_i$ then $t_i\{q\}$ is true for the part $Part(y)$ within context c.

Note that we start with a parse tree in which every node's correspondence to the base grammar G is already established. Therefore, satisfaction is defined such that the property on the left side concerns x and the properties on the right side concern direct components of x (or x itself if the only child of x is a renaming node). As a special case, since any type is itself a property (P1), an unmodified t-production from the base grammar is also a constraining t-production (having no additional constraints) that is satisfied by any part of type t in any context.

If part x satisfies production ρ and ρ includes an annotation d for some non-terminal v, then for each node y in $Nodes(v)$ such that $y = x$ or y is a child of x, we say that *d selects* part $Part(y)$. Thus we establish (candidate) associations between annotations in a constraining grammar and parts in a parse tree.

As an example consider a grammar having only one explicit production:

```
Section ::= Heading Paragraph* Section*
```

Figure 5.1 shows a subtree S in a parse tree corresponding to that grammar. There are three parts of type Section in S: the three nodes labelled by Section.

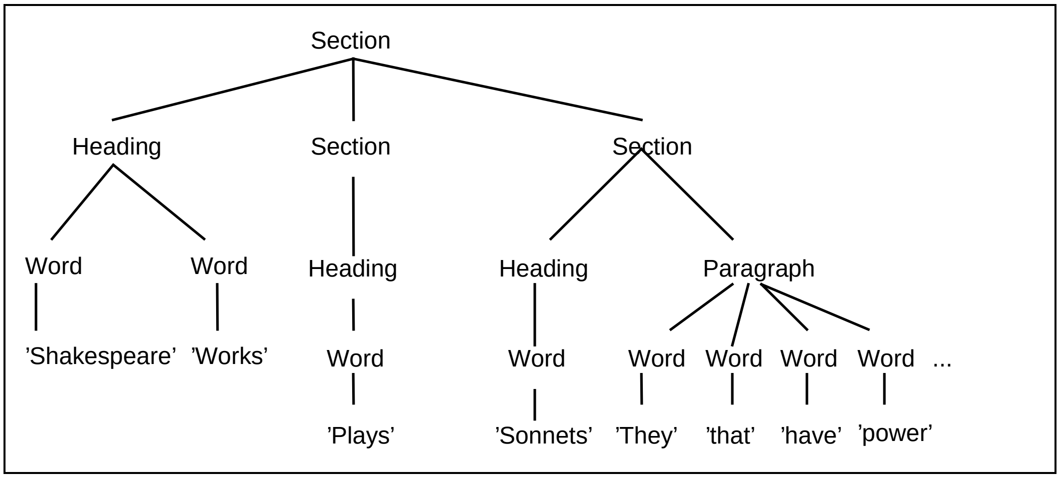

Figure 5.1. A subtree in a parse tree corresponding to a section.

Consider now the following constraining production:

Section{"Sonnets" & "power"} ::= Heading{::title} Paragraph* Section*

A section in some context *c* satisfies the constraining production if it contains the words Sonnets and power somewhere in the section. Within the context of the outermost section in Figure 5.1, that section and its second subsection satisfy the production since they both contain the words Sonnets and power. As a result, the annotation title selects the two parts corresponding to the headings of those two sections. Alternatively, the constraining production

Section{"power"} ::= Heading{"Sonnets"} Paragraph* Section*

is satisfied by the second subsection only, since it is the only one for which the Heading part contains the word Sonnets. Attaching conditions within an iteration constrains all the corresponding parts. Thus

Section ::= Heading Paragraph* Section{“power”}*

requires that every subsection include the word power. The outermost section in Figure 5.1 does not satisfy this constraining production, since its first subsection does not include the word; however both inner sections satisfy the production vacuously.

We write productions of a constraining grammar inside a box, with the start symbol of the grammar (which identifies which parts will serve as the evaluation contexts of the properties in the constraining productions) indicated outside the top left corner of the box. Each production of the base grammar is used for writing zero or more constraining productions. For example, consider again the grammar in Figure 2.1. We can define a constraining grammar as

Paper

```
Paper{"SGML"} ::= Front{Location{"Canada"}::Summary} Body Back
Section{1 :: Start} ::= SectionHeading (Paragraph+ | Paragraph* SubSection+)
Section ::= SectionHeading{"grammar"} (Paragraph+ | Paragraph* SubSection+)
SubSection ::= SectionHeading{"constraint"} Paragraph+
```

The context type for the constraining grammar is Paper, as indicated outside the box, and thus all parts of type Paper may serve as contexts. A paper that contains the word SGML anywhere and the word Canada within its location satisfies the first production. The first section of any paper satisfies the second production. Any section having the word grammar in its section heading satisfies the third production. Finally, any subsection that includes the word constraint in its heading satisfies the fourth production.

By including these four productions in a single constraining grammar, we specify additionally that annotations Summary and Start are to identify parts that are selected by all four productions taken together.

Definition. Given a part c of type t, a constraining grammar $G'(t)$, and a production ρ in $G'(t)$, a *candidate match point x for ρ in context c* is a part contained in c that satisfies ρ in context c, and, in addition, if there are constraining t'-productions in the grammar and there is a part x' of type t' such that c contains x' and x' contains x, then the part x' must satisfy at least one of the t'-productions in context c.

Thus, for example, candidate match points for the second constraining production above are restricted to be in papers that satisfy the first constraining production. Similarly, to be candidate match points, sections satisfying the third production must also be within papers satisfying the first one, and subsections satisfying the fourth production must be contained within candidate match points for either the second or the third constraining production. In this way, we can specify a tree pattern to operate against data conforming to the base grammar (*cf.* [8, 29]).

Definition. Let $G'(t)$ be a constraining grammar for type t obtained from a base grammar G, and let c be a part of type t in a parse tree for G. Part *c matches $G'(t)$* if c contains at least one candidate match point for each production of $G'(t)$. Furthermore, let d be the annotation attached to non-terminal t' in constraining production ρ, and let c contain a part x in $Nodes(t')$. Part *x matches d in $G'(t)$* if c matches $G'(t)$ and either

(i) x is a candidate match point for ρ in context c, or

(ii) t' is on the right side of ρ and x is a direct component of a candidate match point for ρ in context c.

Using these definitions consider again the above example. A paper matches the constraining grammar if

(i) it contains the word SGML anywhere and the word Canada within its location,

(ii) one of its sections includes the word grammar in its section heading, and
(iii) a subsection that includes the word constraint in its heading is contained in either the first section or in some section with the word grammar in its section heading.

Furthermore, the part corresponding to such a paper's front matter matches the annotation Summary, and the part corresponding to the first section of such a paper matches the annotation Start.

We may want instead to allow a paper in which *any* subsection contains the word constraint in its heading, even if it is in a section that is not first and whose heading does not contain the word grammar. In this case, we have to add another Section-production to the constraining grammar to create the possibility for a path from such a subsection to the root of the paper. Such a constraining grammar would then be as follows:

Paper

```
Paper{"SGML"} ::= Front{Location{"Canada"}::Summary} Body Back
Section{1 :: Start} ::= SectionHeading (Paragraph+ | Paragraph* SubSection+)
Section ::= SectionHeading{"grammar"} (Paragraph+ | Paragraph* SubSection+)
Section ::= SectionHeading (Paragraph+ | Paragraph* SubSection+)
SubSection ::= SectionHeading{"constraint"} Paragraph+
```

Because the third Section-production has no additional constraints, any section in a paper is a candidate match point for the production.

The remaining examples show how selected specifications described by Macleod [36] are written as constraining grammars.

Example 5.1:

SmithList **gets document where** (′Smith′ **in** Author)

Paper

```
Paper{::SmithList} ::= Front{Author{“Smith”}} Body Back
```

The property Author{"Smith"} must hold in the front matter of a paper for the document to be selected. Note that the following formulation is *not* equivalent:

Paper

```
Paper{::SmithList} ::= Front Body Back
Front ::= Title Author{“Smith”}+ Location Abstract
```

as this requires *every* author of the paper to contain Smith.

Example 5.2:

DbPaper **gets document having** SubSection **where** (′database′ **in** SectionHeading) **and having** SubSection **where** (′text′ **in** SubSection)

The query specifies the documents where the word database occurs in the heading of a subsection and the word text occurs in a subsection.

Paper

```
Paper {:: DbPaper}::= Front Body Back
SubSection ::= SectionHeading{"database"} Paragraph+
SubSection{"text"} ::= SectionHeading Paragraph+
```

By including two constrained productions for subsections, the constraints are independently applied. Nevertheless both conditions must be satisfied for a paper to match the constrained grammar (and thus the annotation DbPaper).

Example 5.3:

SList **gets** SubSection **having** Paragraph **where** (′database′ **in** Paragraph) **of** Section **where** (′retrieval′ **in** SectionHeading)

This query specifies the subsections where the word database occurs in an included paragraph and they are contained in sections having the word retrieval in *any* contained heading.

Section

```
SubSection{Paragraph{"database"} :: SList} ::= SectionHeading Paragraph+
SubSection ::= SectionHeading Paragraph+
SectionHeading{"retrieval"} ::= Word+
```

The context type restricts all candidate match points to occur within a single section. The first production identifies subsections that include the word database in any paragraph. The word retrieval may occur in the heading of the section itself or in the heading of a subsection of the section. The presence of the second unconstrained production allows the heading to be attached to any subsection. (Because there is no constraining production for Section, the matched heading may also be directly attached to the section itself.)

5.2 Building filters

To allow more flexibility in our specifications, we introduce filters which consist of one or more interconnected constraining grammars. In a compound filter (*i.e.*, a filter having more than one constraining grammar) we may build up conditions such that the constraints written in each grammar remain simple; this facilitates readability and reusability. Compound filters are required when there is a need to combine conditions evaluated in different contexts. They also simplify writing disjunctive conditions, and they are needed to specify complicated structures where one type name has several occurrences on the right side of productions. With a compound filter we are able to specify different kinds of constraints concerning different occurrences of the same type, or different constraints for different parts corresponding to the same non-terminal.

When one or more constraining grammars are used in filters, annotations appearing in the constraining grammars are regarded as declarations of transient text types. Annotations can be associated to parse trees by adding transient types as additional labels during the application of the filters. Therefore we extend our notions related to text types to cover transient types in addition to the base types. Extended definitions for types, text entities, and properties are given as follows:

Definition. Let $G = (A, N, P, s)$ be a context-free grammar and D a set of symbols distinct from the symbols of $A \cup N$. The symbols of D are called *transient* (*text*) *types*, the symbols of N are called *base (text) types*, and the symbols of $N \cup D$ together are called (*text*) *types*. An *annotated parse tree* for G and D is a parse tree Y for G where some nodes are annotated with additional node labels taken from D. The *parts* of Y, the *parts and values of the types* in N, and the function *Nodes*, are defined as in Section 3. A part of Y is a *part of a transient type* t in D if it is labelled by t. Transient types have as values the values of the parts on which the type appears as an annotation. An annotated tree derived from a parse tree by labelling the parts matching annotations in a filter F is called a *parse tree annotated by* F. When there is no ambiguity, we call an annotated parse tree simply a parse tree.

The universal properties defined in Section 4 are extended such that the types occurring in them may be either base types or transient types. Thus in a filter, constraints may refer to base and transient types.

Definition. Let G be a grammar and D a set of transient types. A *filter* $F = \langle F_1, ..., F_n \rangle$ ($n \geq 1$) is a sequence of constraining grammars for G and D such that the types used to constrain each component F_i are either base types defined in the grammar or transient types defined by the annotations in the filter $\langle F_1, ..., F_{i-1} \rangle$. Given a parse tree Y, a part x in Y *matches* d in F if

(i) d is an annotation in F_1 and x matches d in F_1, or

(ii) d is an annotation in F_i ($i > 1$) and, after annotating Y by the filter $\langle F_1, ..., F_{i-1} \rangle$, x matches d in F_i.

Example 5.4:

Suppose we want to check the following constraints for a paper defined using the grammar of Figure 2.1:

(a) No author's name can be repeated in the authors list.

(b) The location must contain the word Canada.

(c) The abstract may contain no more than three paragraphs.

(d) There may be no more than five subsections in any section.

(e) There may be no more than ten citations.

These conditions are defined by the following compound filter:

Front

```
Front{::FrontOK} ::= Title Author{¬ = Author}+ Location{"Canada"} Abstract{ ¬ Paragraph{4} }
```

Section

```
Section{¬ SubSection{6}::SectOK} ::= SectionHeading (Paragraph+ | Paragraph* SubSection+)
```

Paper

```
Paper{:: PaperOK} ::= Front{FrontOK} Body Back{ ¬ Citation{11} }
Body ::= Section{ SectOK }+
```

The first three conditions are checked by the first constraining grammar, using the context of the front matter for one paper. Although the simple constraint on location can be done in any context, the check against other authors and the enumeration of paragraphs must be evaluated within the part corresponding to the front of a single paper. Condition (d) must be evaluated on a section-by-section basis so that the counting of subsections is correct. Note that the second constraining grammar is equivalent to:

Section

```
Section{::SectOK} ::= SectionHeading (Paragraph+ | Paragraph* SubSection{1..5}+)
```

The final constraining grammar in the filter identifies papers that meet all five conditions. Because citations occur only in the back matter of a paper, their enumeration within a paper is identical to that within the back matter taken separately. In this case, the final constraining grammar is *not* equivalent to

Paper

```
Paper{:: PaperOK} ::= Front{FrontOK} Body{SectOK} Back{ ¬ Citation{11} }
```

which states merely that the body should contain at least one valid section.

6. Filters for data manipulation

In the previous section, we introduced the notion of a filter as a mechanism for specifying a set of parts in a given parse tree. When a filter is used to check the validity of a text database or an individual document, it can operate as a boolean operation returning the value true if the database or document matches an annotation in the final constraining grammar of the filter. In this section we elaborate the use of filters for data retrieval, transformation, update, view definition, and hypertext creation. Given a parse tree, these applications yield other parse trees, thus providing a set of operations with the property of closure. In Section 6.1 we consider text retrieval. In Section 6.2 we discuss transformations and show that retrieval may be regarded as a special case of transformation. In Section 6.3 we discuss update and view definition also as transformation operations applied to persistent data. Finally, in Section 6.4 we show how filters can be used for creating hypertexts.

6.1 Retrieval

If *d* is an annotation of a non-terminal *t* in a filter, then the parts matching *d* are not only parts of the transient type *d* in the annotated tree, but also parts of the base type *t* in the original parse tree. The retrieval operation may be defined as an operation that, given a parse tree for a grammar *G*, returns a parse tree for the grammar G'(Output), where $G' = G \cup$ {Output ::= $(t_1|t_2|\ldots|t_n)^*$} and t_i are non-terminals in the base grammar. In the resulting tree, the subtrees of the root consist of all subtrees *Y* taken from the input tree such that the root of *Y* is a part matching *d*.

Example 6.1:

To retrieve the front sections of papers for which one of the paper's authors is also a cited author, we write the following specification based on a compound filter composed of two grammars:

Citation

```
Author{:: RefAuthor} ::= Word+
```

Paper

```
Paper  ::= Front{Author{ = RefAuthor} :: SelfRef } Body Back
```

Output: SelfRef

The first constraining grammar annotates the authors in the citations, thus distinguishing the corresponding parts from the authors of the papers themselves. Using the second grammar, the front matter of a paper matches the annotation SelfRef if one of the authors has the same value as a cited author within the context of that paper. The output line specifies that parts matching SelfRef are to be retrieved. Thus the grammar for the result contains the production Output ::= Front*, the output parse tree has root labelled by Output, and the subtrees of the root are copies of subtrees in the argument tree with roots labelled by Front and matching the annotation SelfRef in the filter.

6.2 Transformations

Many text operations can be described as parse tree transformations. Specifying parse tree transformations based on grammar transformations was introduced by Pratt [42] and by Aho and Ullman [4], with additional operators defined by others, including Furuta and Stotts [21], Kilpeläinen *et al*. [30], Kuikka and Penttonen [32], Mamrak *et al.* [37], and in DSSSL [27]. In this approach, a text transformation is described by a pair of grammars, input grammar and output grammar, and an associated algorithm that defines how a parse tree for the input grammar is transformed into a parse tree for the output grammar. A similar idea for defining text transformations is also included in the p-string model [25], where a production may be used to specify a tree transformation. In SGML document databases, as application requirements and document specifications evolve, text transformations are often needed to change from an old DTD to a new one so that old documents can be managed together with the new documents within one DTD.

The filter notation described in this paper may be used to capture the capabilities of such text transformation specifications. In our framework, a transformation is a two-stage process specified by a pair of filters: the input filter specifies the parts to be transformed, and the output filter describes the new structure to be assembled.

The result of a transformation is a tree with root labelled by Output and subtrees labelled by the context type of the first constraining grammar in the output filter, which implies the start symbol(s) of the grammar describing those subtrees. Note that retrieval, as described in the previous section, is a special case in which there are no constraining productions in the output filter.

For more elaborate transformations in place of simple selections, we include productions in the output filter. We wish to allow selection of some parts, structural changes such as the removal or interchange of non-terminals, and the insertion of strings of terminal symbols. Thus we must define output filters to allow productions that are modified from the productions of the base grammar. In place of constraining grammars as defined in Section 5, a grammar for an output filter is defined as follows:

Definition. Let $F = <F_1, ..., F_n>$ be an input filter defined over grammar $G = (A, N, P, s)$ and annotations D. Given a text type t from $N \cup D$ and a set of terminal symbols A' such that $A' \cap (A \cup N \cup D) = \emptyset$, an *output grammar for t* is a 4-tuple $G'(t) = (A \cup A', N \cup D, P', t)$, where the productions of P' have the following characteristics:

(i) There is at most one t'-production for every non-terminal t' in N.

(ii) In every use of iteration, optionality, or alternation, each argument must be either a simple non-terminal or the concatenation of a non-terminal with terminal symbols.

The text type t is called the *context type* of the output grammar. An output filter is a sequence of output grammars.

For example, assume we wish to display the front parts of papers containing the word SGML, but we wish to see the authors followed by title and abstract, and we wish to preserve only the first and last paragraphs of the abstract. We can define a compound input filter consisting of two constraining grammars (so that only paragraphs from the abstract are selected), and a simple output filter that assembles the required text fragments:

Example 6.2:

Paper

Paper{"SGML"} ::= Front{::FrontMatter} Body Back

Front

Front{FrontMatter} ::= Title Author+ Location Abstract
Paragraph{1:: FirstPara} ::= Sentence+
Paragraph{-1 & ¬1:: LastPara} ::= Sentence+

Output: FrontMatter

Front ::= ′SGML paper′ Author+ Title Abstract
Abstract ::= FirstPara LastPara?

The result of a transformation defined by a simple output filter is a parse tree for a grammar with root symbol Output and productions including all those defined in the base and output grammars, as well as the production

Output ::= t*

if the context type t is a non-terminal, or

Output ::= (t_1 | t_2 | … | t_n)*

if the context type t is an annotation matched by parts that are labelled by non-terminals t_i. The output tree is populated by data drawn from the original text.

In Example 6.2, having matched the input filter, the output tree will contain a subtree for the front matter for each selected paper. The subtrees of each node labelled Front will include a single terminal node, followed by subtrees copied from all Author parts *within the context of the given front matter*, followed by a subtree matching the original title, followed by a revised subtree for the abstract. The revised subtree for the abstract will have a copy of the part selected by FirstPara within the context of the same front matter followed by the part selected by LastPara if it exists within that same context.

If an output filter includes more than one output grammar, the second and subsequent grammars specify transformations to the parse tree created by the previous grammar in the filter. For each such grammar, the context type identifies matching nodes in the intermediate parse tree and restricts manipulations to subtrees forming the contents of those nodes.

To illustrate this, we use a simple input filter and a compound output filter to form tables of contents for papers mentioning SGML:

Example 6.3:

Paper

```
Paper{"SGML" ::Selected} ::= Front Body Back
Front ::= Title Author{::Writer}+ Location Abstract
Section ::= SectionHeading{::Head} (Paragraph+ | Paragraph* SubSection+)
SubSection ::= SectionHeading{::Head} Paragraph+
```

Output: Selected

```
Paper ::= Title  Writer+  Section+
```

Section

```
Section ::= Head+
```

Within the context of a paper containing the word SGML, the paper itself, its authors, and each section and subsection heading are annotated. The next constraining grammar begins the output filter. Each selected paper is replaced by its unique title, all its authors (but not the authors in citations), and all its sections (maintaining the order in which they occur in the input). Then, *within the context of a single selected section*, the section's content is replaced by all headings that occur under that section, again in the order that they occur. Note that if the same Section-production were included in the first output grammar, each transformed section would include all the headings gathered from context of the whole paper.

6.3 Update and views

If a transformation producing an output tree from the argument tree is applied *in situ* to persistent data, then it can be regarded as having specified an update. In syntax directed text editors, the update operations could be specified by corresponding output grammars, applied to a chosen part (for example, a document). We can specify an update in the form of a transformation as above, but instead of creating a new structure with root labelled by Output, the identified subtrees are replaced in the argument tree.

Example 6.4:

For example, the following pair of filters might be used to delete all but the first three sections of the paper with title "A Peek at SGML":

Paper

Paper {::Peek} ::= Front {Title {="A Peek at SGML"} } Body Back
Section{1..3 :: KeepSect} ::= SectionHeading (Paragraph+ | Paragraph* SubSection+)

Update: Peek

Paper ::= Front Body Back
Body ::= KeepSect+

The semantics behind the transformation algorithm are that parts of the text are not altered except as required to meet the structural conditions as specified by the update filter. Thus, for example, the order of the first three sections is preserved by the transformation.

The transformation might change the grammar as well as the content of the database. If the structure of a subset of the parts of a given type *t* is changed, the *t*-production must be extended in the resulting grammar. For example, an update where the front parts of selected papers are changed to contain title and abstract only can be specified as follows:

Paper

Paper{"SGML"} ::= Front{::FrontMatter} Body Back

Update: FrontMatter

Front ::= Title Abstract

If this update is applied, the result is a single tree for Papers that replaces some of the Front parts in the argument tree. The grammar for the updated tree is obtained from the grammar for the argument tree by replacing the Front-production by

Front ::= Title Author+ Location Abstract | Title Abstract

In addition to queries and updates, there is also a need to define views, which may be similarly specified by filters. As for relational databases, in using a filter for a view definition, the resulting tree may be either a virtual structure (similar to that produced by a query) or materialized (similar to an update). Information retrieval from databases of documents defined by varying grammars could be described without actual text transformations if unmaterialized views are defined as wrappers around the information sources to create a uniform collection [14, 43]. On the other hand, indexing of a text may be considered to be a special case of creating a materialized view: a transformation performed at the time that a text is loaded into a document database. The text modelling technique described in this paper has been used to specify text indexing by grammar transformations [49, 50, 53].

6.4 Specifying hypertexts

In structured text, some parts in the hierarchic structure may reference other parts, and these references may be implemented as links in a hypertext retrieval system. The hypertext structure may also be regarded as an alternative structure, defined over the same text as the hierarchic structure, but creating non-hierarchic networks [44]. Such a hypertext structure may exist statically, or it may be created dynamically in response to a user request [45, 54, 56]. Simple properties as defined in Section 4 have been shown to be useful for specifying such hypertext structures [49, 51, 53].

Example 6.5:

Salminen *et al.* define a hypertext over the text of the Canadian Patent Reporter, a collection of court case descriptions organized in three series [49]. Among the grammar rules, they define the following productions:

```
CPR ::= case+
…
section ::= … | citations | … | decision | … | summary | …
citations ::= CPR_citation txt*
CPR_citation ::= first_series | second_series | third_series
reference ::= first_series | second_series | third_series
```

where references may appear in decision or summary sections. They then specify constraints to ensure that citations are uniquely defined and references are valid; in our revised notation this would be expressed as a filter:

CPR

```
citations ::= CPR_citation(¬= CPR_citation) txt*
reference(= CPR_citation) ::= first_series | second_series | third_series
```

Based on such a filter, they define navigating operations that, given a reference-part within the context of a particular case, can retrieve the case containing the matching CPR_citation-part.

7. Conclusion

In this paper we have described an information modelling facility using structured text. It offers a framework in which the semantics of data definition and of data manipulation operations may be precisely defined.

The structure of text is defined by a context-free grammar, and the structured text instance is given by a parse tree. Thus models of text are distinguished by the grammars chosen to represent valid collections. For a grammar and a parse tree, we define the notions of text types and parts and of content and value equality for parts, as required of a data model [46]. Other equivalence relations can be developed in a similar manner. For example, some applications might use an equivalence relation in which terminal symbols that occur as siblings of parts are ignored: if HTML text is parsed such that tags are regarded as unnamed delimiters, then their inclusion or omission would not affect a comparison for equality.

A second contribution is the notion of a filter for specifying a set of parts from a parse tree. A simple filter is an annotated constraining grammar; a compound filter consists of several such grammars in a chain. Annotations in a filter define transient text types that may be used to reference parts from constraining grammars further along the filter chain. We characterize how operations based on filters could be defined for data validation, data retrieval, text transformation, text update, view definition, and for creating hypertext structures.

7.1. Expressive power and efficiency

Navarro and Baeza-Yates compare the structuring power and efficiency of some text retrieval models [39]. Within their framework, our data definition facility can be classified as hierarchical, explicitly and statically single-structured, and strongly structure-bound. It is intended to model text having a regular, but not necessarily simple, structure. An important feature in our model is its capability to support the definition and manipulation of hierarchic information in a uniform manner. Defining filters using multiple grammars over the same text or using several databases with heterogeneous grammars is a natural extension of the model to be studied in the future.

In our data manipulation facility, query answers are nested. Navarro and Baeza-Yates identify four additional features of query languages to compare text models: text matching, set manipulation, inclusion relationships, and distances. The properties we define in Section 4 provide simple text matching of words and phrases; approximate matching, including string pattern matching, and testing for distance between words can be similarly defined. We define several properties for testing inclusion relationships as well. Unlike the other systems reviewed by Navarro and Baeza-Yates, however, we also define a property to test for repetition of words or phrases (P6 in Figure 4.1), which can be used to define additional constraints on a text database or to support hypertext linking, among other applications. The functionality of set manipulation is provided through logical connectives to combine properties and through the filtering mechanism described in Section 5.

Neven and Van den Bussche characterize the expressiveness of structured document query languages more formally [41]. In particular they show that boolean attribute grammars can specify exactly those queries expressible in monadic second-order logic and that relation-valued attribute grammars have a greater expressivity, characterized by linearly-bounded iterations of first-order formulas. Turning to our framework, filters can be shown to be definable by first-order predicate calculus over expressions involving values of type node and of type string. Thus they cannot describe all queries expressible by attribute grammars, even if attributes in those grammars are restricted to be boolean-valued only.[7] On the other hand, the filter property to test for repetition of words or phrases cannot be simulated by a boolean attribute grammar, so these two formalisms are incomparable.

As a result of their more limited expressivity, however, query filters can be implemented to run as efficiently as more conventional database systems rather than taking time proportional

---

[7] The same inability is true of SQL [40].

to the size of the database as is required of most attribute grammars [40]. A prototype text retrieval system has been implemented using SICStus Prolog and Tcl [33]. Most of the languages described in Section 2 (including DSSSL [27], Lorel [38], extensions of SQL [8], or the language of proximal nodes [39]) could be used as implementation languages for filters more generally.

7.2 Further work

Text is often divided into multiple concurrent hierarchic structures, such as logical sections and physical pages [44, 52]. Having shown how to manage text defined by one grammar, how can the model be extended to cover text defined by several grammars simultaneously?

Relaxing the restrictions on productions in output grammars for transformation filters is an interesting area for further study. We require that parts of the text are not altered except as required to meet the constraints specified by an output or update filter. One approach to defining a transform is to associate costs with various atomic operations and to choose a minimal cost change that meets the output constraints [28]. Under what conditions is there a unique minimal cost change such that the update semantics seems natural? What limitations does this impose on the expressivity of transformations, and thus on the set of possible views?

A major problem in developing systems for structured text is to define flexible, simple, and effective user interfaces. A primary motivation for developing an approach based on constraining grammars was to create a framework that supports the development of template-based user interfaces. Query or transformation templates are created directly from the productions of a base grammar, upon which the user may then impose constraints to the non-terminals. The template-based specification capability has been implemented in a prototype system [33], but this has not been subject to systematic end-user evaluation. The design and evaluation of user interfaces for flexible and powerful structured text environments is a challenging area for continued study.

Acknowledgements

This paper evolved from numerous collaborations with Gaston Gonnet, Eila Kuikka, Darrell Raymond, Jean Tague-Sutcliffe, and Carolyn Watters, to whom we are indebted. Frank Neven suggested the characterization of the expressivity of filters. Financial support from the Academy of Finland and the Natural Sciences and Engineering Research Council of Canada is gratefully acknowledged.

We dedicate our paper to the memory of Jean Tague-Sutcliffe.

References

1. S. Abiteboul. Querying Semistructured Data, Proc. 6th Int. Conf. On Database Theory (ICDT'97), Delphi, Greece (January 1997), Lecture Notes in Computer Science 1186, Springer-Verlag, 1-18.

2. S. Abiteboul, S. Cluet, and T. Milo. Querying and updating the file. In Proc. 19th Int. Conf. on Very Large Data Bases, pp. 73-84 (1993).

3. S. Abiteboul, S. Cluet, and T. Milo. Correspondence and translation for heterogeneous data. Proc. 6th Int. Conf. On Database Theory (ICDT'97): 351-363(1997).

4. A.V. Aho and J.D. Ullman. The Theory of Parsing, Translation, and Compiling, Vol. 1: Parsing. Prentice-Hall, Inc. (1972).

5. G.E. Blake, T. Bray, and F.W. Tompa. Shortening the OED: experience with a grammar-defined database. ACM Trans. on Information Systems, **10** (3): 213-232 (1992).

6. G.E. Blake, M.P. Consens, P. Kilpeläinen, P.-Å. Larson, T. Snider, and F.W. Tompa. Text / relational database management systems: harmonizing SQL and SGML. In Proc. Applications of Databases (ADB-94), pp. 267-280, Vadstena, Sweden (1994).

7. T. Bray, J. Paoli, and C. M. Sperberg-McQueen (eds.). Extensible Markup Language (XML) 1.0. W3C Recommendation 10-February-1998. REC-xml-19980210. 36pp.

8. L.J. Brown, M. Consens, I.J. Davis, C.R. Palmer, and F.W. Tompa. A structured text ADT for object-relational databases, TAPOS (to appear 1998).

9. A. Brüggemann-Klein. Unambiguity of Extended Regular Expressions in SGML Document Grammars. Proc. 1st European Symposium on Algorithms (ESA'93): 73-84 (1993).

10. P. Buneman, S.B. Davidson, G.G. Hillebrand, D. Suciu, A Query Language and Optimization Techniques for Unstructured Data, Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data (Sigmod 96), Montreal, Canada (June 1996), SIGMOD Record **25**,2, pp. 505-516.

11. F.J. Burkowski. An algebra for hierarchically organized text-dominated databases Information Processing & Management, **28** (3): 333-348 (1992).

12. S.S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data, SIGMOD Record, **25** (2): 493-504 (1996).

13. V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In Proc. 1994 ACM SIGMOD Int. Conf. on Management of Data, SIGMOD Record, **23** (2): 313-324 (1994).

14. C.L.A. Clarke, G.V. Cormack, and F.J. Burkowski. Schema-independent retrieval from heterogeneous structured text. In Proc. 4th Ann. Symp. on Document Analysis and Information Retrieval: 279-290 (1995).

15. L.S. Colby, L.V. Saxton, and D. Van Gucht. Concepts for modeling and querying list-structured data. Information Processing & Management, **30** (5): 687-707 (1994).

16. D.D. Cowan, E.W. Mackie, G.M. Pianosi, and G. de V. Smit. Rita – an editor and user interface for manipulating structured documents. Electronic Publishing, Origination, Dissemination, and Design, **4** (3): 125-150 (1991).

17. Dao, T., Sacks-Davis, R., and Thom, J.A., Indexing structured text for queries on containment relationships, Proc.7th Australasian Database Conf., Melbourne, Australia, Jan. 1996, 82-91.

18. I.J., Davis. Adding structured text to SQL/MM Part 2: Full Text. Dept. of Computer Science, Univ. of Waterloo, SQL/MM Change Proposal LHR-24, CAC WG3 N334R2 (1996).

19. B.C. Desai, P. Goyal, and F. Sadri. A data model for use with formatted and textual data. J. American Society for Information Science, **37** (3): 158-165 (1986).

20. M. Fuller, E. Mackie, R. Sacks-Davis, and R. Wilkinson. Structured answers for a large structured document collection. In Proc. 16th Ann. Int. ACM/SIGIR Conf. on Research and Development in Information Retrieval, pp. 204-213, ACM Press (1993).

21. R. Furuta and P.D. Stotts. Specifying structured document transformations. In Proc. Int. Conf. on Electronic Publishing, Document Manipulation and Typography, pp. 109-120, Cambridge University Press (1988).

22. C.F. Goldfarb. The SGML Handbook. Oxford University Press, Oxford, UK (1990).

23. R. Goldman and J. Widom: DataGuides: enabling query formulation and optimization in semistructured databases. VLDB 1997: 436-445.

24. G.H. Gonnet. Examples of PAT® applied to the Oxford English Dictionary. Technical Report OED-87-02, UW Centre for the New Oxford English Dictionary (1987).

25. G.H. Gonnet and F.W. Tompa. Mind your grammar: a new approach to modelling text. In Proc. 13th Int. Conf. on Very Large Data Bases, pp. 339-346 (1987).

26. M. Gyssens, J. Paredaens, and D. Van Gucht. A grammar-based approach towards unifying hierarchical data models, In Proc. 1989 ACM SIGMOD Int. Conf. on Management of Data, SIGMOD Record **18** (2): 263-272 (1989).

27. ISO IEC 10179:1996, Information technology - Processing languages - Document Style Semantics and Specification Language (DSSSL). International Organization for Standardization (1996).

28. A.M. Keller. The role of semantics in translating view updates. Computer **19**, 63-73 (1986).

29. P. Kilpeläinen. Tree matching problems with application to structured text databases. Ph.D. Thesis, University of Helsinki, Helsinki, Finland (1992).

30. P. Kilpeläinen, G. Lindén, H. Mannila and E. Nikunen, A structured document database system. In Proc. Int. Conf. on Electronic Publishing, Document Manipulation & Typography, pp. 139-151, Cambridge University Press (1990).

31. E. Kuikka. Processing of structured documents using a syntax-directed approach. Ph.D. Thesis, Kuopio University Publications C, Natural and Environmental Sciences 53, Department of Computer Science and Applied Mathematics, University of Kuopio, Kuopio, Finland (1996).

32. E. Kuikka and M. Penttonen. Transformation of structured documents with the use of grammar. Electronic Publishing, Origination, Dissemination, and Design **6** (4): 373-383 (1993).

33. E. Kuikka and A. Salminen. Filtering structured documents in the SYNDOC environment. Electronic Publishing, **8** (2&3): 181-193 (1995).

34. W.R. Lalonde. Regular right part grammars and their parsers. Comm. ACM, **20** (10): 731-741 (1977).

35. A. Loeffen. Text databases: A survey of text models and systems. SIGMOD Record, **23** (1): 97-106 (1994).

36. I.A. Macleod. A query language for retrieving information from hierarchic text structures. The Computer Journal, **34** (3): 254-264 (1991).

37. S. Mamrak, C.S. O'Connell, and J. Barnes. The Integrated Chameleon Architecture. Prentice-Hall (1994).

38. J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Barnes. Lore: A database management system for semistructured data. SIGMOD Record **26** (3): 54-66 (1997).

39. G. Navarro and R. Baeza-Yates. Proximal nodes: A model to query document databases by content and structure. Trans. Info. Sys. **15** (4): 400-435 (1997).

40. F. Neven and J. Van den Bussche. On implementing structured document query facilities on top of a DOOD. Proc. 5th Int. Conf. on Deductive and Object-Oriented Databases: 351-367 (1997).

41. F. Neven and J. Van den Bussche. Expressiveness of structured document query languages based on attribute grammars. Proc. 17th ACM Symp. on Principles of Database Systems: 11-17 (1998).

42. T.W. Pratt. Pair grammars, graph languages and string-to-graph translations. J. Computer and System Sciences **5**, 560-595 (1971).

43. D. Quass, A. Rajaraman, Y. Sagiv, J.D. Ullman, and J. Widom. Querying semistructured heterogeneous information. Proc. 4th Int. Conf. on Deductive and Object-Oriented Databases: 319-344 (1995).

44. D.R. Raymond. Partial Order Databases. Ph.D. Thesis, Dept. Of Computer Science, University of Waterloo, Ontario (1996).

45. D.R. Raymond and F.W. Tompa. Hypertext and the Oxford English Dictionary. Comm. ACM, **31** (7): 871-879 (1988).

46. D.R. Raymond, F.W. Tompa, and D. Wood. From data representation to data model: meta-semantic issues in the evolution of SGML. Computer Standards & Interfaces **18**, 25-36 (1996).

47. R. Sacks-Davis, T. Arnold-Moore, and J. Zobel. Database systems for structured documents, IEICE Transactions on Information and Systems, **E78-D** (11): 1335-1342 (1995).

48. R. Sacks-Davis, A. Kent, K. Ramamohanarao, J. Thom, and J. Zobel. Atlas: a nested relational database system for text applications, IEEE Transactions on Knowledge and Data Engineering **7** (3): 454-470 (1995).

49. A. Salminen, J. Tague-Sutcliffe, and C. McClellan. From text to hypertext by indexing. ACM Trans. on Information Systems, **13** (1): 69-99 (1995).

50. A. Salminen and F.W. Tompa. PAT expressions: an algebra for text search. Acta Linguistica Hungarica, **41** (1-4): 277-306 (1992-1993).

51. A. Salminen and C. Watters. A two-level structure for textual databases to support hypertext access. J. American Society for Information Science, **43** (6): 432-447 (1992).

52. C.M. Sperberg-McQueen and L. Burnard, eds., Guidelines for Electronic Text Encoding and Interchange. Chicago and Oxford, ALLC-ACH-ACL Text Encoding Initiative (1994).

53. J. Tague, A. Salminen, and C. McClellan. Complete formal model for information retrieval systems. In Proc. 14th Ann. Int. ACM/SIGIR Conf. on Research and Development in Information Retrieval, pp. 14-20, ACM Press (1991).

54. F.W. Tompa, G.E. Blake, and D.R. Raymond. Hypertext by link-resolving components. In Proc. 5th ACM Conf. on Hypertext, pp. 118-130, Seattle (1993).

55. D. Tsichritzis and F. Lochovsky. Data Models. Prentice-Hall, Inc. (1982).

56. C.R. Watters and M.A. Shepherd. Transient hypergraph-based model for data access. ACM Trans. on Inf. Syst., **8** (2): 77-102 (1990).

57. T.W. Yan and J. Annevelink, Integrating a structured-text retrieval system with an object-oriented database system. Proc. 20th Int. Conf. on Very Large Data Bases (VLDB'94), Santiago de Chile, pp. 740-749, Morgan Kaufmann (1994).