Grammars++ for Modelling Information in Text

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

November 1, 1996

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

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

provide view designers and end-users the capability to direct their attention to relevant information fragments, to formulate meaningful queries, and to specify the appropriate 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. 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, as depicted in Figure 1.1.

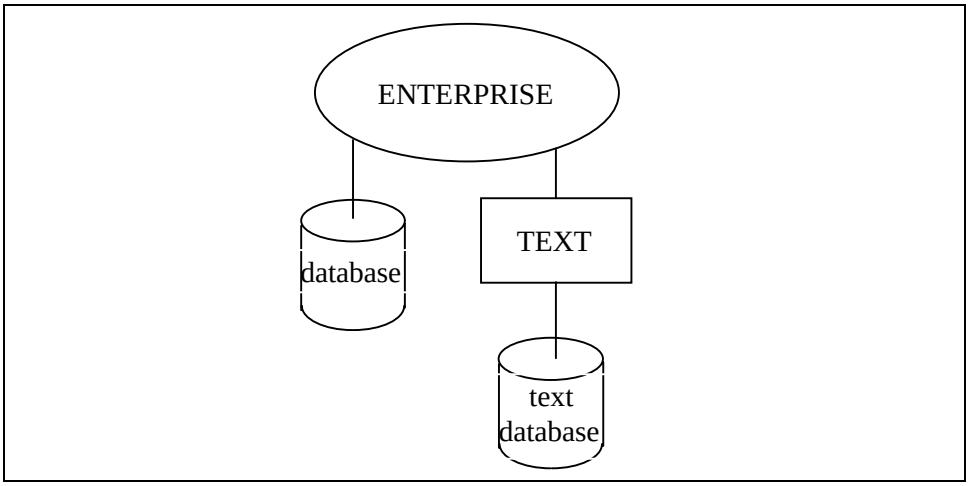

Figure 1.1. Contrast between text database and conventional database.

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 (see, for example, Tsichritzis & Lochovsky (1982)), the field of formal languages has a rich history of modelling text strings with grammars (see, for example, Aho & Ullman (1972)). 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. The designer of a publishing system might want to know how many different glyphs are required for printing articles. 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.

The most common models for documents treat text as a linear stream of characters with a superimposed hierarchic structure denoting logical and/or physical segments. Such a vision of text corresponds to a tree structure, and text operations are described in terms of tree matching and manipulation. Models for "hypertexts" extend this approach to more general graphs. More flexible 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 text algebra of Burkowski (1992) and extended by Clarke, Cormack and Burkowski (1995), the list-structure algebra of Colby, Saxton and Van Gucht (1994), the PAT text algebra described by Gonnet (1987) and Salminen and Tompa (1992), and the partial order model proposed by Raymond (1996).

In order to impose some constraints on collections of documents, or to recognize and exploit existing commonalities, models have been devised to include structure definitions separately from the texts themselves. Such definitions, predominantly in the form of context-free grammars, serve the role of schema declarations for a text database. For example, SGML offers such a data definition language for structured text (Goldfarb 1990, Burnard & Sperberg-McQueen 1994). Text models based on grammars have richer data modelling capabilities than is possible without the support of data definition languages.

Several models combining grammar-based data definition capabilities with operational capabilities on trees have also been introduced; consider, for example, the models of Gonnet and Tompa (1987), Furuta and Stotts (1988), Gyssens, Paredaens and Van Gucht (1989), Macleod (1990, 1991), Christophides (1994), and Blake *et al.* (1994). 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. The p-string model (Gonnet & Tompa 1987, Blake, Bray & Tompa 1992), the model of Gyssens *et al*. (1989),

and the structured text extensions to SQL (Blake *et al.* 1995, Davis 1996) are such hybrid models.

This paper introduces constrained grammars, which are specified by adding boolean conditions to the productions of a grammar. We show that constrained grammars extend the modelling capability of simple context-free grammars. Such constrained grammars can serve as a 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 defining multiple views, specifying data conversion, specifying a full-text retrieval language (Salminen & Tompa 1992), defining index structures (Salminen *et al.* 1995), and defining hypertext access to structured text (Salminen & Watters 1992).

Section 2 provides a brief overview of the use of a grammar to describe text and introduces notation and basic vocabulary used in the remainder of the paper. Section 3 then describes how grammars can be used to model entities as well merely delineating a set of strings. The following section extends grammars to provide for the specification of further constraints by attaching predicates to non-terminal symbols. In Section 5 we show how constrained grammars provide a data definition language for describing text database schemas as well as providing a simple query language. Similarly, in the following section the extended grammar facilities are used as the basis of a fuller data manipulation language. Throughout the paper, concepts and facilities are illustrated in terms of a simple document database.

2. Context-free grammars, strings, derivations, and parse trees

Formal grammars are designed for describing typical features of text: hierarchic structure, order, optionality, alternatives, recursive structures. Among formal grammars, the context-free grammars, or some specializations of them, are most commonly used for specifying structured text. The basic notions of formal grammars are well-known (see for example, Aho & Ullman 1972) and briefly, but rigorously, reviewed here to introduce terminology and notation.

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

(a) *A* is a set of *terminal symbols*,
(b) *N* is a set of *non-terminal symbols*,
(c) *P* is a set of *productions*, and
(d) *s* is a distinguished non-terminal in *N*, called the *start symbol*.

The productions are of the form $t ::= \alpha$, where *t* is called the left side and α the right side of the production. The left side is a non-terminal symbol, the right side may contain non-terminal symbols, terminal symbols, and metasymbols (α may also be empty). The metasymbols are used as operators to indicate iteration, alternatives and optionality. Iteration is denoted by * (zero or more times) and + (one or more times), optionality by square brackets [and], and | separates alternatives. Parentheses (and) are used for grouping, i.e., to show the operand of an operator and the order in which operators are applied. Terminal symbols are written between the characters ' and '. A production whose left side is *t* is called a *t-production*.

Example 2.1
A familiar text structure is a collection of papers. Throughout this paper we will use the example grammar shown in Figure 2.1, where the structure for a paper has been defined following the example used by Macleod (1991). The start symbol of the grammar is Papers. Following Macleod, we suppose that unspecified non-terminal symbols in the sample grammar represent word sequences: for each unspecified non-terminal *t*, there is an implicit production *t* ::= Word+, where the production for Word produces a terminal symbol.

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 (the terminal symbols), and the ways these symbols can be combined to build a legal string in the language. Any such string can be derived from the start symbol *s* of the grammar as follows. Let *G* = (*A*, *N*, *P*, *s*) be a context-free grammar.

```
(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+
```

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

Definition. A *derivation* is a sequence $ß_1 => ... => ß_n$ where

(a) $ß_1 = s$,
(b) $ß_n$ is a string of terminal symbols,
(c) $ß_i$, $1 < i < n$, is a string consisting of terminal and non-terminal symbols,
(d) each $ß_i$, $1 < i \leq n$, is derived from $ß_{i-1}$ by replacing one non-terminal symbol occurrence t in $ß_{i-1}$ by a *variant* of the right side of a t-production. The variant is produced as follows:

(i) An iteration indicated by + is replaced by one or more of its operands.
(ii) An iteration indicated by * is replaced by zero or more of its operands.
(iii) An alternatives is replaced by one of its operands.
(iv) An optional value is either replaced by its operand or deleted.
(v) These replacements are conducted until all metasymbols are deleted.

Note that if the right side of the t-production has no metasymbols, then the variant is the same as the right side.

A *language generated by G* is the set of terminal symbol strings producible by such derivations.

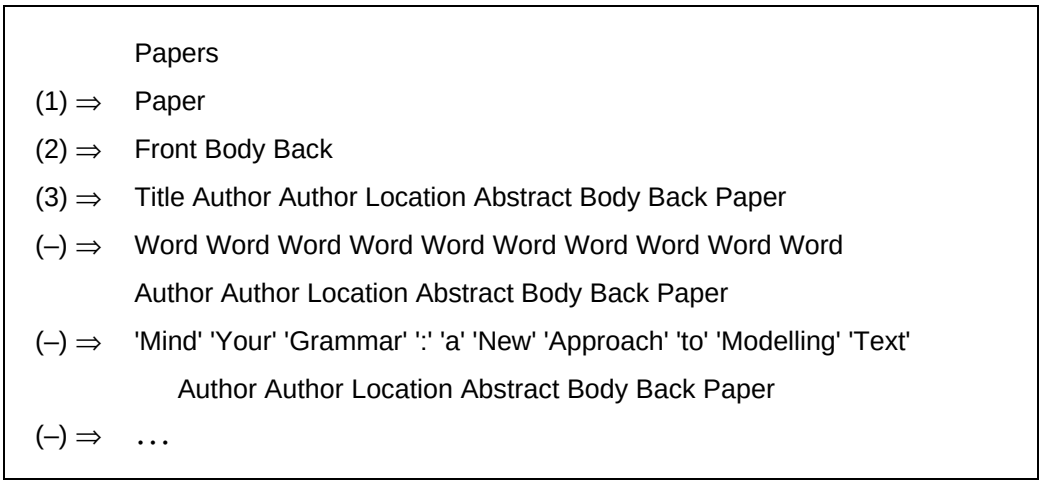

Figure 2.2. A derivation using the grammar in Figure 2.1.

Example 2.2.
Figure 2.2 shows a portion of a derivation using the grammar in Figure 2.1. The derivation starts from the symbol Papers. In each step of the derivation, the leftmost non-terminal symbol has been replaced by a variant of the right side of the corresponding production. The productions used are indicated by their numbers before the symbols ⇒. The derivation begins by an application of production (1). The variant of the right side of the production is Paper. The Word symbols have been derived by using the implicit production Title ::= Word+ which is omitted from Figure 2.1.

The relationship between a string in a language and a grammar defining that language can be represented by a derivation tree, or equivalently a parse tree (see Aho & Ullman 1972). In the tree, each parent with its children corresponds to an application of some production in the derivation. Figure 2.3 shows a parse tree using the grammar in Figure 2.1 and corresponding to the derivation shown in Figure 2.2. The nodes of the tree are labelled by non-terminal and terminal symbols.

Definition. 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 by *s*.

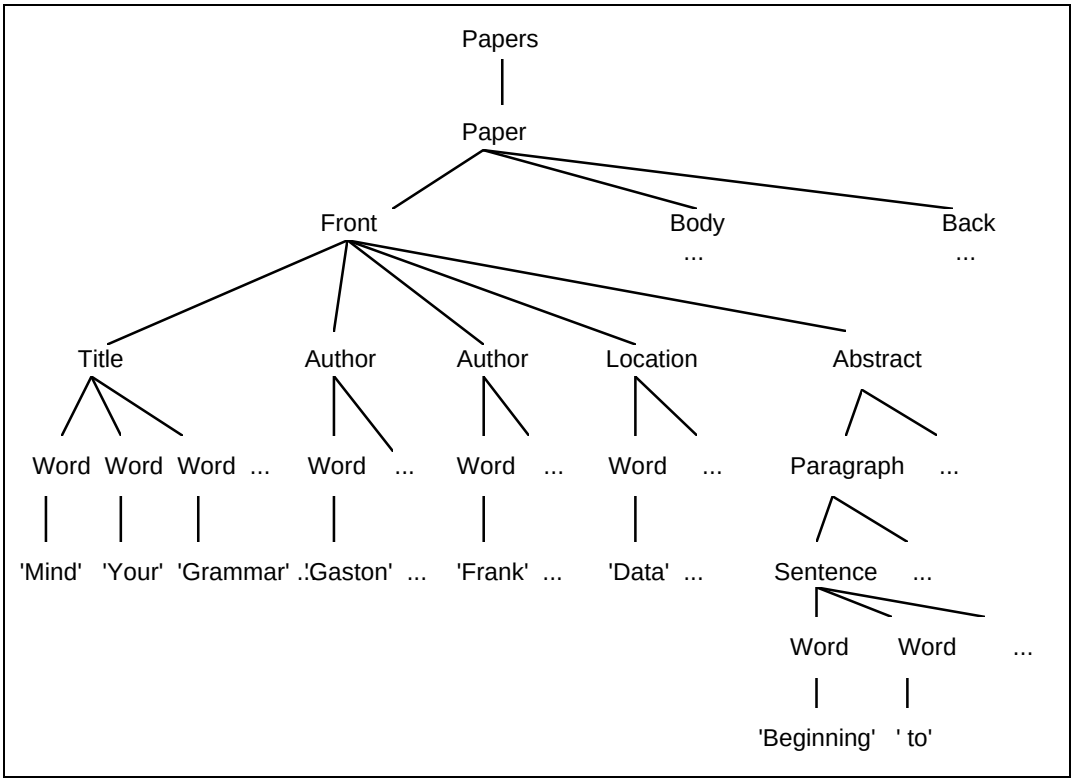

Figure 2.3. A parse tree corresponding to the derivation in Figure 2.2.

(2) If D_1, ..., D_k ($k \geq 1$) are subtrees that are 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 an 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 by X_i if X_i is a terminal.

(3) Alternatively, if the root has no descendants then there is in G an s-production whose right side has a variant that is empty.

In many cases a parse tree is created by parsing a character string: given a string find some derivation using the given grammar. The components of the hierarchic structure are often marked so that the parser is able to identify the begin and end of meaningful substrings representing elements of the hierarchic structure. For example, a parse tree may be created from an SGML document in which begin and end tags show the parts of the hierarchic structure. On the other hand, a parse

tree may also be created by a syntax-directed editor: text to be edited is associated with a given grammar, and the operations provided by the editor manipulate a parse tree (see, for example, Cowan *et al.* 1991). No special indication for the hierarchic structure is then needed in the character strings, as the structure is maintained implicitly as the string evolves. The third approach to create a parse tree for a string, given a grammar, is to derive it as a text transformation from an existing parse tree corresponding to a related grammar.

In many text management systems, structured text is created in two phases. First a hierarchic structure is identified from a character string by parsing. Then indexing is applied to the text, and as changes are applied to the original text during indexing, a revised parse tree is generated. For example, the grammar of Figure 2.1 is meant to describe indexed text for which the original text might consist of a collection of SGML documents. Elsewhere we have shown how indexing of text may be described by text transformations (Salminen & Tompa 1992 and Salminen *et al.* 1995).

3. Text types and parts

The grammar in Figure 2.1 specifies the strings belonging to the language defined to be a collection of papers. It is well-known that a language can be generated by many different grammars, and several normal forms for grammars have been described for the purposes of formal language theory. However, we wish to regard a collection of papers as a text database modelling meaningful entities, and therefore grammar design is concerned not only with describing valid strings but also characterizing meaningful structures. In particular, the text entities defined by a grammar $G = (A, N, P, s)$ are of two kinds: character strings and hierarchic structures. Each non-terminal symbol in N represents simultaneously a set of possible character strings, a set of nodes (together with corresponding subtrees) in all possible parse trees for the grammar, and, for a given parsed string, a set of specific substrings and a set of nodes in the corresponding parse tree. Because of the parallels with data types in other database applications, we call the non-terminal symbols in *N text types*. To be

able to consider a production with *t* on its left side as a mechanism for completely defining a type *t*, we restrict grammars for text databases such that each non-terminal symbol appears only once as the left side of a production.

Let $G = (A, N, P, s)$ be a context-free grammar, *t* a text type in *N*, and *Y* a parse tree for *G*. We define the values and parts of type *t* as follows:

Definition. The *values of type t* (or *t values*) are the terminal symbol strings in the language generated by the grammar $G(t) = (A, N, P, t)$, i.e., the grammar derived from *G* by choosing *t* as the start symbol. In *Y*, a node *x* labelled by a non-terminal symbol is a *part* if it is not a singleton child of its parent (see below). The subtree *X* with *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*.

A single child of a parent is regarded as renaming a part, not as a separate part itself, and such nodes are called *renaming nodes* of the part. A part is a *part of type t* (or a *t part*) if *t* is the label of *x* or the label of a renaming node of *x*. 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′* and *x′* is *contained in x*. Part *x′* is *properly contained in x* if *x′* is not *x*. Part *x′* is *directly contained in x* if it is properly contained in *x* and not properly contained in any *x″* which is also properly contained in *x*.

If the grammar allows the derivation of an empty string from a non-terminal symbol *t*, then a leaf node may be labelled by non-terminal *t*. In such a case, the *t* part has an empty string as its value. For example, the grammar in Figure 2.1 has the production

(1) Papers ::= Paper*.

Thus there may be an empty collection of papers, and respectively a parse tree consisting of the root node alone. The parse tree would then consist of one part of type Papers with an empty value.

In the parse tree of Figure 2.3, the nodes labelled by Papers, Front, Title, and Author are examples of parts. The Papers part directly contains the Front part, which directly contains the Title and the two Author parts. The node labelled by Paper is not a part: it just renames the part labelled by Papers, both denoting the complete text. Thus the root of the tree is a part of two types: Papers and Paper; in the real world being modelled, it represents both the collection of papers and the only paper in that collection. Note that, as a corollary of these definitions, if part *x* contains another part *x′* and they have identical values, then either *x* contains some other part *x″* having an empty value or the two parts *x* and *x′* are themselves identical.

Our approach also uses grammar productions to specify operations on text. To determine the targets of such operations within a given parse tree, we define a correspondence between text type occurrences in the grammar and nodes of the parse tree. Because we have exactly one production for each text type, a type *t* on the left side of a production corresponds to any node in the parse tree labelled by *t*. If the node has any children, they are produced by applying the *t*-production, using one of the variants of its right side, which induces a correspondence between the child nodes and type occurrences appearing on the right sides of productions. To avoid ambiguity, if a type occurs more than once on the right side of a production, integer superscripts in the node labels of associated parse trees will be used to indicate which type occurrence corresponds to each node (and a node label with no superscript behaves as if it had the integer superscript 1).

Since the correspondence between type occurrences in a grammar and nodes of a parse tree is now unambiguous, the correspondence between parts of a type *t* in a parse tree and occurrences of *t* in the grammar is also unambiguous.

Definition. Let $G = (A, N, P, s)$ be a context-free grammar, *t* a type occurrence in *P*, and *Y* a parse tree for *G*. A part *x* of type *t* in *Y* is a *part corresponding to the type occurrence t* if either *x* or a renaming node of *x* is a node corresponding to that occurrence.

Consider the parse tree in Figure 2.3 and the productions

(1) Papers ::= Paper*

(2) Paper ::= Front Body Back

The parts corresponding to the Papers occurrence in the first production consist of the root node. The root part also corresponds to the Paper occurrence in the first production as well as in the second production.

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. Furthermore, in choosing the grammar in Figure 2.1, the text database designer has opted not to include an entity type Authors as part of the domain of discourse.

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. We describe in Sections 5 and 6 how the mechanism can be used in a data definition language and in a data manipulation language.

For each of the text types *t* of a grammar we can define a logical operation, or *property*, that tests if a part of a parse tree is a part of type *t*. For example, in Figure 2.3, the properties Papers and Paper are true for the same part, whereas the property Front is not true for that part. In general, properties can be arbitrary predicates that may be applied to any part.

Properties will be used to define text operations that behave as constraints for text type occurrences. They are expressed in a form that can be written as part of the productions of a grammar. In Section 4.1 we discuss properties that may be

defined for the types of any grammar, and in Section 4.2 we indicate properties that may be separately defined for some specific grammars. All properties are described as if they were tested in the context of a complete database. In Section 5 we then show how properties may be used to obtain constrained productions from productions of a grammar, and we introduce a mechanism for restricting the context in which a property is to be tested.

4.1. Universal properties

In structured text, information is captured in the names of parts, in the values of parts, and in the structural relationships among parts. Text operations should provide the functionality to test information in any of the three categories. For testing the name of a part there is the family of properties t_i, where t_i is any type of the grammar. As indicated above, the property t is true for any part of type t and false otherwise. The properties testing the values of parts as well as the properties testing the structural relationships of parts are written in the form $t\{q\}$. In such a property, q specifies an additional constraint for parts of type t. Regardless of the constraint, the property $t\{q\}$ is false for all parts that are not of the type t. The constraint q is written such that it is a string of types, character strings, numbers, and operator symbols.

(P1)	t
(P2)	$t\{ =r \}$
(P3)	$t\{ =p \}$
(P4)	$t\{ r \}$
(P5)	$t\{ p \}$
(P6)	$t\{ n_1 .. n_2 \}$
(P7)	$t\{ \neg q_1 \}$
(P8)	$t\{ q_1 \,\&\, q_2 \,\&\, ... \,\&\, q_n \}$
(P9)	$t\{ q_1 \mid q_2 \mid ... \mid q_n \}$

Figure 4.1. Universal properties.

Figure 4.1 shows the properties we define in this section and use in later examples. In the properties, t is a text type of 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 such that $t\{q_i\}$ is a property. The properties (P2) and (P3) test the value of a part, properties (P4) through (P6) test structural relationships among parts, and the properties (P7) through (P9) combine different 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.

4.1.1. Properties testing the value of a part

In all text retrieval languages there are capabilities to specify conditions to be met by character string values of textual parts. The property $t\{ =r \}$ simply tests if the value of a part is equal to the character string r. In many retrieval environments more fuzzy string properties are needed. Sophisticated properties may be defined as special properties to meet specific needs for grammars, as discussed in Section 4.2. As general properties, however, we might easily define additional properties such as “begins with the given string,” “contains the given string,” or “ends with the given string.”

In relational databases the join operation is used to compare attribute values among relations, and in hypertexts following a cross-reference requires matching a source to a target. The property $t\{ =p \}$ provides a general capability in text databases to compare the values of parts.[3] The property is true for a part if its value is equal to another distinct part which matches the property p. For example, consider a parse tree defined by the following grammar:

```
                Staff ::= Employee+
(E)             Employee ::= Name Address [Phone]
                Name ::= Surname FirstName+
```

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

The property

(P) Surname{ =FirstName }

is true for those Surname parts that have a value that occurs as a first name somewhere in the text. In this example the whole parse tree serves as the context for the comparison. In many cases, however, it is important to be able to restrict the context to a subtree of the parse tree. For example, we might be interested in employees having a matching first name and surname. Without restricting the context for property evaluation, property (P) alone would not be correct for testing this feature. Later we will define means by which the comparison of the values may be restricted within a specified context.

4.1.2. Properties testing the structural relationships among parts

Structural relationships include the containment hierarchy of parts and the ordering of parts with respect to other parts.

For testing the containment hierarchy we will use only the properties $t\{p\}$ and $t\{r\}$. The property $t\{p\}$ is true for a part x if it contains a part y (possibly x itself) for which p is true.[4] For example, consider again a parse tree defined by the productions (E) above. 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 structure and value. The property is true for a part if it contains an atomic part whose value is r. For example, using Figure 2.1, the property Abstract{"SGML"} is true for an abstract that contains the word SGML; because all atomic parts in Figure 2.1 are of type word, Abstract{"SGML text"} would always return false (assuming the phrase SGML text is two words) and Abstract{="SGML text"} would return true only for two-word abstracts having precisely those words.

[4] In our earlier papers the property $t\{p\}$ was written as $t\{\text{contains } p\}$.

In earlier papers we used four kinds of properties for testing the containment hierarchy: $t\{\text{is } p\}$, $t\{\text{in } p\}$, $t\{\text{contains } p\}$, and $t\{\text{where } p\}$, all of which have been useful (Salminen *et al.* 1995, Salminen & Tompa 1992, Salminen & Watters 1992). Although conditions combining properties that test both the content of a part and its surrounding context are commonly required, these may become quite complicated to specify simultaneously. In Section 5 we introduce a mechanism to build combinations of conditions that eliminates the need to define primitive properties that test the context surrounding a part. As a result, we have reduced the set of universal properties.

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 among an ordered set of sibling parts. The semantics of the property is defined as follows:

Suppose the argument is the i^{th} of m sibling parts corresponding to a single type occurrence t. 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 $-n_2 \leq m-i+1 \leq -n_1$. If n_1 is positive and n_2 is negative, the property is true if $i > n_1$ and $m-i+1 > -n_2$. The notation $t\{n\}$ is short for $t\{n..n\}$ and the notation $t\{n..\}$ is short for $t\{n..-1\}$. For example, Figure 2.1 defines SubSection to be

SubSection ::= SectionHeading Paragraph+

The property Paragraph{1..5} is true for the first five paragraphs, the property Paragraph{-1} is true for the last paragraph, and the property Paragraph{5..} is true for all paragraphs starting from the fifth.

The property $t\{n_1 .. n_2\}$as defined above is not the only possibility for a property testing the position of a part. In some applications, it might be useful to have a property that tests the position of a part of a type among all parts of the type within a specified context, regardless of whether the parts are siblings. By restricting the context we might then search, for example, for the first five

sentences of a paragraph or the first five sentences of a section, or the first five sentences of a paper.

4.1.3. 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 of the 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 of the properties $t\{q_1\}$, ..., $t\{q_n\}$ is satisfied. For example, the property Surname{ ¬ =Surname} tests whether a given part is a Surname having a value not equal to any other Surname part.

4.2. 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. This ability allows us to define either grammar-specific or type-specific operators giving formal meaning to the types.

As for universal properties, if a special property $t\{q\}$ is 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+
```

We may now define the property *t*{< *n*}, where *n* is a value of type Number. For any type *t'* of a grammar containing the productions, 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*. Another way to define the property is to restrict its application to parts of type Number. The property would then yield false to parts of all other types. For example, the values of Year parts could be tested by the operation if a grammar contains the production

```
Year ::= Number
```

but not if it contains the production

```
Year ::= Digit Digit
```

As mentioned in the previous section, testing values of parts by the character string equality property *t*{ =*r* } is not very often useful as a criterion in text queries. In many cases we need more flexible properties for testing if the value of a part matches some character string. For a specific kind of grammar we can define the property *t*{matches *r*} where *r* is a character string. The property may be defined in different ways for different types *t*, but in that case it has to be noted that a part may be a part of several types at the same time. For example, we might define the matching property for the parts of the grammar in Figure 2.1 such that the property *t*{matches *r*} is true if *r* is the prefix of the value of a *t* part, after both *r* and the value of the part are normalized in the same way. The normalization is then defined in different ways for different types. For example, for the type Author the normalization could consist of the following steps: (a) if the first character is "O" followed by upper case letter, then replace it by "O ", (b) replace all upper case letters by corresponding lower case letters, (c) replace hyphens and apostrophes by spaces, (d) replace the prefix "mac" by "mc ", and finally (e) replace the multi-space sequences by one space. Then, for example,

```
Author{matches "OBrien"}
```

would be true for an author part whose value is "O'Brien A T" since the normalized test string "o brien" matches a normalized prefix. Similarly,

```
Author{matches "mac  Connell"}
```

would be true for author parts with value "McConnell H" or "MacConnelly John".

A complete text database system should include a facility for specifying user-defined properties, likely as part of a general abstract datatype mechanism.

5. Specifying conditions for parts by constrained grammars

We need to express constraints effectively to improve our ability to model information. Grammars provide the basis of our approach, allowing us to specify constraints through productions. In Section 5.1 we extend the modelling power by using properties in place of non-terminals in productions. Then, in Section 5.2, the notion of a constrained grammar and the matching of a part and a constrained grammar is defined. Constrained grammars are used to define the context within properties are tested and to specify non-context-free constraints. In the final section, we introduce the notions of filter and transient text type, which together provide a mechanism to build complex matching criteria.

5.1 Constrained productions

Definition. A *constrained production* is obtained from a production of a *base grammar G* by adding constraints to type occurrences. If t is a type occurrence in the original production and $t\{q\}$ is a property defined for the grammar, then in the constrained production t may be replaced by $t\{q\}$. If x is a part of a parse tree corresponding to base grammar G and x corresponds to a type occurrence t in G then *x matches a constrained production* obtained from the t-production in G if

(i) The property on the left side of the production is true for x.

(ii) If x corresponds to a constrained type occurrence $t_i\{q\}$ on the right side of the production then $t_i\{q\}$ is true for x.

(iii) If x' is a part directly contained in x and x' corresponds to a constrained type occurrence $t_i\{q\}$ on the right side of the production, then $t_i\{q\}$ is true for x'.

Note that we start with a parse tree in which every node's correspondence to the base grammar *G* is already established. Therefore, the matching is defined such that the property on the left side concerns *x*, the properties on the right side concern direct components of *x*, or *x* itself if *x* is a part corresponding to a type on the right side as well. As a special case, since any type is itself a property (P1), an unmodified *t*-production from the base grammar is also a constrained *t*-production (having no additional constraints). Any part of type *t* matches such a production.

As an example consider the production

```
Section ::= SectionHeading (Paragraph* | Paragraph* Section+)
```

in a grammar for an article. Figure 5.1 shows a subtree in a parse tree for an article collection. In the subtree, there are three parts of type Section: the three nodes labelled by Section. Two of the Section parts are contained in the outermost Section part, and the first of the subsections consists of a section heading only. Consider the following constrained production:

```
Section{"Sonnets" & "power"} ::=
        SectionHeading{"Shakespeare"} (Paragraph* | Paragraph* Section+)
```

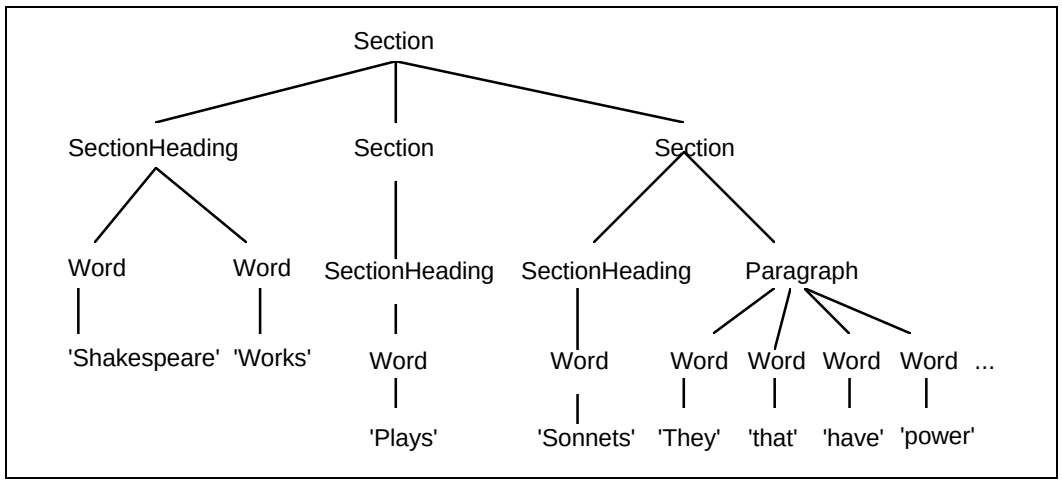

Figure 5.1. A subtree in a parse tree for an article.

A section matches the production if it contains the words Sonnets and power somewhere in the section, and its heading contains the word Shakespeare. Only the outermost section in Figure 5.1 matches the production: although the words Sonnets and power occur in the second subsection, the title of the subsection does not contain the word Shakespeare. Similarly, the constrained production

```
Section  ::= SectionHeading{ Section } (Paragraph* | Paragraph* Section+)
```

matches the first subsection only, since it is the only one for which the SectionHeading part contains a Section part (namely itself). The constrained production

```
Section{ Paragraph {2} }  ::= SectionHeading (Paragraph* | Paragraph* Section+)
```

is matched by no part, since no section has a second paragraph.

By associating the constraints with specific type occurrences in the productions, we are able to restrict a given condition to the parts corresponding to that specific occurrence. Suppose a grammar contains the production

```
Authors ::= Author Author*
```

which makes a distinction between the first author and the rest of the authors. Consider the constrained production

```
Authors ::= Author{="Doe"} Author{1..2}*
```

In a parse tree corresponding to the base grammar, either a part *x* corresponding to Authors has only one author, in which case *x* is also an Author part, or *x* has a directly contained part corresponding to the first Author type occurrence in the production and one or more directly contained parts corresponding to the second Author type occurrence in that production. Thus, a part *x* of type Authors matches the constrained production if the first (or only) author is Doe and if there are at most two other authors. Note that the value of the positional property $t\{n_1..n_2\}$ depends on the position of the parts corresponding to an individual type

occurrence in the production only, not other siblings even if they have the same name. Adding a constraint to an iterated type occurrence means that all of the corresponding parts must obey the constraint. Therefore, for successful matching, the property Author{1..2} must be true for all authors corresponding to the second Author occurrence in *x*, and thus at most two authors are accepted after Doe.

5.2 Constrained grammars

A constrained grammar consists of constrained productions obtained from the productions of a given base grammar. The start symbol of the constrained grammar restricts the evaluation context for the properties in the constrained productions. In addition, productions in a constrained grammar may contain annotations, which provide names for sub-parts matched by the grammar.

An annotation in a constrained production is written as part of a property, within the braces following the type name and separated from any constraints by two colons.

TypeName {*constraints* :: *annotation*}

For example, the annotated production

```
Front ::= Title{:: CFTitle}  Author+  Location{"Canada" | "Finland" }
                                Abstract{"grammar"  & "SGML" :: CFAbstract}
```

includes two annotations: CFTitle and CFAbstract. The intent of the production is to specify the title and abstract of a paper where the location contains the word Canada or Finland, and the abstract contains the words grammar and SGML.

Formally, annotations on a constrained production obtained from a grammar *G* are drawn from a set of symbols, *D*, distinct from the type names of *G*.

Definition. A *constrained grammar* obtained from a base grammar $G = (A, N, P, s)$ is a grammar $G'(t) = (A, N, P', t)$, where the productions of P' are constrained productions obtained from the productions of P, annotations in P' are distinct symbols drawn from a set of symbols D such that $D \cap (A \cup N) = \varnothing$, and t is a text type in N. $G'(t)$ is called a *constrained grammar for t,* and t is called the *context type* of the constrained grammar.

Unlike the restriction for base grammars, a constrained grammar may contain more than one constrained production for a given production of the base grammar. However, no annotation symbol may be repeated within a constrained grammar.

We write productions of a constrained grammar inside a box, with the start symbol of the grammar (and thus the evaluation context of the properties in the constrained productions) indicated outside the top left corner of the box. For example, consider again the simple grammar from Section 4.1:

```
Staff ::= Employee+
Employee ::= Name Address [Phone]
Name ::= Surname FirstName+
```

We can define a constrained grammar for Employee as

Employee

```
Name {:: EchoName} ::= Surname{ = FirstName}  FirstName+
```

In the absence of specifying a context, the property Surname{= FirstName} is true for a Surname part having a value equal to *any* FirstName part anywhere in the text database. A constrained grammar with *t* as the start symbol limits the context to a single *t* part: it is used to test parts contained in a *t* part and the evaluation context of properties in the constrained productions is restricted to the corresponding subtree that is a parse tree for the grammar $G(t)$. Thus, the constrained grammar above associates the annotation EchoName with a part of

type Name if, within the context of a employee, the surname is identical to one of the first names, such as for William Halse Rivers Rivers.

If x is a part and X its content, we denote by $X(t)$ the parse tree for $G(t)$ whose root is x or a renaming node of x labelled by t. $X(t)$ is the parse tree within which we test whether a part matches a constrained production. Note that if t labels a renaming node of x, the evaluation context $X(t)$ excludes one or more nodes starting from the root of X. For example, if the staff grammar above is G, the properties in the constrained grammar for Employee are tested in a context which is a parse tree for the grammar G(Employee). If there is only one employee, the part of type Employee is the node labelled Staff, which is outside the tree whose root is labelled by Employee. Nodes outside the context do not affect the truth values of the properties tested, and therefore the values remain the same whether or not an employee is the sole employee in the staff database.

A constrained grammar that includes several productions is used to test conditions through hierarchical application of the constrained productions, as explained below, rather than testing the productions independently of each other. Given the grammar in Figure 2.1 as the base grammar, we might define the following constrained grammar for Paper:

Paper

```
Front ::= Title{"grammar" :: ATitle}  Author+ Location{"Canada"} Abstract
Section(::ASection} ::= SectionHeading{"SGML"} (Paragraph+ | Paragraph*
SubSection+)
Paragraph{"filter"}   ::= Sentence+
```

In this example, since Paper is chosen as the start symbol, the properties in the productions are tested within a paper. The first production indicates that the title of the paper contains the word grammar, the word Canada occurs in the Location part, and the name ATitle is to be associated with the title of a paper conforming to the constrained grammar. The second production indicates that a section heading contains the word SGML and that the name ASection is to be associated with the corresponding section of a paper conforming to the constrained

grammar. Finally, the third production identifies paragraphs that contain the word filter. However, these productions are not used independently of one another: the constraints for a paragraph apply only to those appearing within parts matching one of the other productions.

The matching of a part *x* and elements of a constrained grammar for *t* is defined by using the notion of *matching points* for the productions of the constrained grammar.

Definition. Given a part *x* of type *t*, a constrained grammar *G'*(*t*), and a production *p* in *G'*(*t*), a *matching point y for p* is a part contained in *x* that, using the evaluation context of *x*, matches *p*, and, in addition, if there are constrained *t'*-productions in the grammar and there is a part *x'* of type *t'* such that *y* is contained in *x'* and *x'* is contained in *x*, then the part *x'* must match at least one of the *t'*-productions within the evaluation context of *x*.

Definition. Let *G'*(*t*) be a constrained grammar for type *t* obtained from a base grammar *G*, and let *x* be a part of type *t* in a parse tree for *G*. Part *x matches G'*(*t*) if *x* contains at least one matching point for each production of *G'*(*t*).

Definition. Let *G* be a base grammar, *G'*(*t*) a constrained grammar, *d* the annotation attached to a type occurrence *t'* in the constrained grammar, and *x* a part corresponding to that same type occurrence *t'* in a parse tree for *G*. Part *x matches d in G'*(*t*) if either

(i) *t'* is a type occurrence on the left side of a production and *x* is a matching point for the production in a part that matches *G'*(*t*), or
(ii) *t'* is a type occurrence on the right side of a production and *x* is a matching point for the production in a part which matches *G'*(*t*), or
(iii) *t'* is a type occurrence on the right side of a production and *x* is a part directly contained in a matching point for the production in a part which matches *G'*(*t*).

Using these definitions consider again the above example. Note from Figure 2.1 that the contents for Front parts and for Section parts must be disjoint. A matching point for the first production is the front part of a paper whose title contains the word grammar, and the word Canada occurs in the Location part of the paper. A matching point for the Section-production is a section of a paper if its heading contains the word SGML. Finally, a matching point for the Paragraph-production has two conditions: it must be a paragraph whose heading contains the word filter; and, because Paragraph parts must be contained within Front or Section parts, it must be contained in a matching point for the Section-production or for the Front-production. A part of type Paper matches the constrained grammar if it contains a matching point for each of the three productions. In such a matching paper, the title matches ATitle by condition (iii) and the section with heading including SGML matches ASection by condition (i). Note that condition (ii) applies only when t' is the label of a renaming node of x.

If we want instead to allow a paper in which *any* paragraph contains the word filter, even if it is in a section whose heading does not contain the word SGML, then we have to add another Section-production to the constrained grammar to create the possibility for a path from such a paragraph to the root of the paper. Such a constrained grammar would be as follows:

Paper

```
Front ::= Title{"grammar" :: ATitle}  Author+ Location{"Canada"} Abstract
Section{:: ASection} ::= SectionHeading{"SGML"} (Paragraph+ | Paragraph* SubSection+)
Section ::= SectionHeading (Paragraph+ | Paragraph* SubSection+)
Paragraph{"filter"}   ::= Sentence+
```

The second Section-production has no constraints, and thus any section in a paper is a matching point for the production.

Example 5.1

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 constrained grammar:

Paper

```
Front ::= Title Author{¬ = Author}+ Location{"Canada"} Abstract
Abstract ::= Paragraph{1..3}+
Body ::= Section{ ¬ SubSection{6} }+
Back ::= Citation{1..10}+
```

Choosing Paper as the start symbol means that all properties are tested in the context of a single paper. Conditions (a) and (b) are tested by the first production. The property Author{¬ = Author} tests that the value of an Author part is not equal to the value of another Author part of the same paper. Because of the context, no comparison with the authors of other papers is made. Conditions (c), (d) and (e) are tested by the second, third, and fourth productions, respectively. Condition (d) is indicated in the Body-production by specifying that no section of the body may include a sixth subsection. Note that replacing this by the constrained production

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

is not equivalent: placing such a constraint in a Section-production specifies merely that there exists a section with no more than five subsections.

Example 5.2
Suppose we want to specify the front part and the first section in a paper

(a) which contains the word SGML, and
(b) where the location contains the word Canada, and

(c) some section heading contains the word grammar, and

(d) the word grammar occurs in the heading of some subsection.

The conditions are expressed as follows:

Paper

```
Paper{"SGML"} ::= Front{::FrontMatter} Body Back
Front ::= Title Author+ Location{"Canada" } Abstract
Section{1 :: FirstSection} ::= SectionHeading (Paragraph+ | Paragraph* SubSection+)
Section ::= SectionHeading{"grammar"} (Paragraph+ | Paragraph* SubSection+)
Section ::= SectionHeading(Paragraph+ | Paragraph* SubSection+)
SubSection ::= SectionHeading{"grammar"} Paragraph+
```

Three Section-productions are required: the first for indicating the first section, the second for indicating a section whose section heading contains the word grammar, and finally the third without any constraints to allow the specified subsections to occur in any section. The grammar has two annotations: FrontMatter and FirstSection. A part matches the annotation FrontMatter if it is the front part in a paper described by the conditions (a) - (d). Similarly, a part matches the annotation FirstSection if it is the first section in a paper described by the same conditions.

The remaining examples show how specifications described by Macleod (1991) are written as constrained grammars.

Example 5.3

DbPaper **gets document having** SubSection **where** ('database' **in** SectionHeading)

The query specifies the documents where the word database occurs in the heading of a subsection. This requires constraining the SectionHeading in the SubSection-production:

Paper

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

Example 5.4

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

The query specifies the subsections where the word database occurs in a paragraph, from sections which contain the word retrieval in a section heading. The word retrieval may occur in the heading of the section itself, or in the heading of a subsection of the section. The corresponding specification with a constrained grammar would be done by adding appropriate constraints to the SubSection- and SectionHeading-productions:

Section

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

The SubSection-production without constraints is included to allow the word retrieval to occur in the heading of any subsection, not only the subsection with the word database. If we wish to restrict the query to require the word retrieval to occur in the heading of the outermost section itself, then the constraint should be placed in the Section-production:

Section

```
Section ::= SectionHeading{"retrieval"} (Paragraph+ | Paragraph* SubSection+)
SubSection{Paragraph{"database"} :: SList} ::= SectionHeading Paragraph+
```

There is no longer any need for the SubSection-production without constraints, since there are no conditions defined for parts that may occur inside any subsection.

Example 5.5

In the query language described by Macleod (1991) some constraints concerning the positions of parts among an ordered set of parts may be given by the locators

first, **second**, **third**, and **last**. Thus the first, second, third, and last position (but not other positions) may be used as a criterion for data access. The query

FirstSection **gets first** Section **of** Papers **where** ('database' **in** SectionHeading)

retrieves the first section of any paper having database in a section heading. The corresponding constrained grammar is as follows:

Body

```
Section{1 :: FirstSection} ::= SectionHeading (Paragraph+ | Paragraph* SubSection+)
SectionHeading{"database"} ::= Word+
```

The context is defined to be Body so as to be able to compare the positions of sections. The word database may occur either in the heading of the first section or in the heading of any of its subsections.

Our capabilities to specify restrictions concerning the position of parts among an ordered set of parts are flexible. Using the various forms of the property $t\{n_1 .. n_2\}$, we are able to refer to parts in a specific position, between specific positions, or before or after a specific position. For example, the last two subsections of the fifth section are specified by

Body

```
Section{5} ::= SectionHeading (Paragraph+ | Paragraph* SubSection+)
SubSection{ -2..-1 :: LastTwo} ::= SectionHeading Paragraph+
```

The following constrained grammar matches the front of a paper where the body contains the word multimedia and the back matter contains at least 80 citations.

Paper

```
Paper ::= Front{:: SurveyPaper} Body{"multimedia"} Back{Citation{80}}
```

Because citations only appear within the back matter of a paper, the constraint concerning the number of citations may be associated with the type Paper in

place of the type Back. Thus, the same parts are specified by the constrained grammar

Paper

```
Paper{Citation{80}} ::= Front{:: SurveyPaper} Body{"multimedia"} Back
```

5.3 Building filters

To get more flexibility into our specifications, we will introduce filters which consist of one or more interconnected constrained grammars. In a compound filter (i.e., a filter having more than one constrained grammar) we may build up conditions such that the constraints written in each grammar remain simple; this facilitates readability and reusability. This is particularly apparent in writing disjunctive conditions, which would be quite complicated if expressed within one constrained grammar. More significantly, compound filters also increase the modelling capability: they are needed, for example, 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 type occurrence. Finally, compound filters are required when there is a need to combine conditions evaluated in different contexts.

When one or more constrained grammars are used in filters, annotations appearing in the constrained grammars are regarded as declarations of transient text types. These can be associated to parse trees by adding transient types as annotations during the application of the filters. Therefore we extend our notions related to text types to cover transient types in addition to the previous 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 *base (text) types*, and the symbols of $N \cup D$ together (*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 *parts corresponding to a type occurrence* in a production, 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* $F = <F_1, ..., F_n>$ ($n \geq 1$) is a sequence of constrained grammars for *G* and *D* such that all annotations in the grammars are distinct and 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 $<F_1, ..., F_{i-1}>$. 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 *x* matches *d* in F_i, after annotating *Y* by the filter $<F_1, ..., F_{i-1}>$.

Example 5.6
When a context-free grammar is used as a database schema, an important principle in schema design is to define clear partitioning hierarchies. In such a definition it may be difficult to restrict the values of types to the correct values, even though the definition of the values could be easily done by a context-free grammar. In our approach, we may use the capabilities of context-free grammars to define a generic partitioning hierarchy; and the restrictions not expressed by the BNF-productions can be specified by filters. Let the base grammar for a database of newspaper articles contain the following productions:

```
ArticleDatabase ::= Article+
Article ::= Date DayOfWeek [Authors] Section [Title] Text
DayOfWeek ::= Saturday | Sunday | Workday
Section ::= News | Business | Sports | Entertainment | Home
```

We might want to test some given constraints for each article at the time it is added to the database, or we might need to check some particular constraints against all articles of an existing database. Suppose we want to check that all of the following conditions hold for each of the articles of the database:

(a) The entertainment articles are published on Saturdays and Sundays only.
(b) The home articles are published on Sundays only.
(c) All news articles have a title.

The corresponding compound filter is as follows:

Article

```
Article{Entertainment:: EntA} ::=
                    Date DayOfWeek{ Saturday | Sunday } [Authors] Section [Title] Text
Article{Home :: HomeA} ::= Date DayOfWeek{Sunday} [Authors] Section [Title] Text
Article{News & Title :: NewsA} ::= Date DayOfWeek [Authors] Section  [Title] Text
```

Article

```
Article{ Business | Sports | EntA  | HomeA  |  NewsA :: CorrectA } ::=
                                    Date DayOfWeek [Authors] Section [Title] Text
```

ArticleDatabase

```
ArticleDatabase{:: CorrectDatabase} ::= Article{CorrectA}+
```

With this filter, we specify the database constraints by first subtyping articles according to the sections in which they occur so as to specify the restrictions according to subtype. A correct entertainment article will be labelled by the name EntA, a correct home article by the name HomeA, and a correct news article by the name NewsA; no restrictions are specified for business or sports articles. In

a second constrained grammar we then specify that a correct article, called CorrectA, is either any business or sports article, or an article of one of the types specified in the first grammar. Finally, in a third grammar we define a correct database to consist of correct articles only. The database matches the annotation CorrectDatabase in the compound filter iff all of the articles in the database are correct articles.

Example 5.7

Suppose we extend the grammar shown in Figure 2.1 by the production:

```
Citation ::= Author* Title Publisher Year [Pages]
```

Note that the grammar also includes the production:

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

Thus in a paper defined by the extended grammar, an Author part may appear in a paper either in a Front part or in a Citation part. To specify those papers for which one of the paper's authors is also a cited author (for the same paper), we write the following compound filter:

Citation

```
Citation ::= Author{:: RefAuthor}* Title Publisher Year [Pages]
```

Paper

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

The first constrained grammar annotates the authors in the citations, thus distinguishing the corresponding parts from the authors of the paper itself. The application of the filter causes all authors in all citations in all papers to be annotated by RefAuthor. The second grammar uses the annotation from the first one as a transient type. A paper *x* matches the annotation SelfRef if one of the authors has the same value as a cited author *within the context of x*.

6. Filter application areas

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. We have shown examples of the use of filters for checking the validity of data and for data retrieval. 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 constrained 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, the uses of filters for retrieval, transformation, update, and view definition 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 type occurrence *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 grammar $G' = G \cup \{\text{Output ::= t*}\}$. 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 labelled by *t* and it is either a part matching *d* or a renaming node of a part matching *d*. For example, consider a compound filter composed of two grammars as follows. The first constrained grammar specifies the papers where the word SGML appears somewhere in the paper and the word Canada occurs in the location. The annotation FrontMatter specifies the front parts of such papers. In the second part of the filter, the annotation FirstPara is bound to the first paragraphs in the abstracts of such papers.

Paper

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

```
Front ::= Title Author+ Location{"Canada" } Abstract
```

Front

```
Front{FrontMatter} ::= Title Author+ Location Abstract
Paragraph{1 :: FirstPara} ::= Sentence+
```

This filter may be used to retrieve the front of the papers by

Output: FrontMatter

and the first paragraphs of the abstracts of the specified papers by

Output: FirstPara

In the first case, the grammar for the result contains the production Output ::= Front*. The resulting subtrees of the root labelled by Output are subtrees of the argument tree with root labelled by Front, and they are parse trees for our sample grammar when Front is taken as the start symbol. In the second case, the grammar for the result contains the production Output ::= Paragraph*, and the resulting subtrees are rooted by a node labelled by Paragraph. If the abstract from which the paragraph is taken contains more than one paragraph, then the root of each subtree is a part in the original tree; if the abstract consists of one paragraph only, then the node labelled by Paragraph is a renaming node of the Abstract part in the original tree (and a renaming node of the Output part in the resulting tree). In either event, the resulting subtrees are parse trees for the sample grammar taking Paragraph as the start symbol.

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 (1971) and by Aho and Ullman (1972), with additional operators defined by Furuta and Stotts (1988), Kilpeläinen *et al*. (1990), Kuikka and Penttonen (1993), Mamrak *et al.* (1994), and in DSSSL (ISO 1996). 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 (Gonnet & Tompa 1987) 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 filters described in this paper may be used to generalize the capabilities of such text transformation specifications, which allow removal of parts by the removal of text type occurrences from an output grammar, the transfer of parts to new positions indicated by a text type, and deletion or insertion of terminal symbols in a parse tree to produce a new parse tree. In our framework, a transformation is 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.

As for retrieval, the result of a transformation is a tree with root labelled by Output and subtrees labelled by the context type of the output filter, which also specifies the start symbol of the grammar describing those subtrees. Constraints in the output filter may only include transient types defined by the input filter, indicating which parts, as annotated by the given type names, participate in the transformed result. As an extension of the filters described in Section 5, the output filter can include productions that are modified from the productions of the base grammar. In the output productions we might allow, for example, structural changes such as the removal or interchange of text type occurrences and the insertion of strings of terminal symbols.

For example, the following pair of filters, using a compound input filter and a simple output filter, might be used to display the front parts of papers containing the word SGML such that only the title and the two first paragraphs of the abstract are printed, each specified part starting a new line.

Input: Paper

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

Front

```
Front{FrontMatter} ::= Title Author+ Location Abstract
Paragraph{1..2 :: FirstTwo} ::= Sentence+
```

Output: Front

```
Front{FrontMatter}::= '\n' Title '\n' Abstract
Abstract ::= ('\n' Paragraph{FirstTwo} ) +
```

The result of the example transformation is a parse tree for the grammar derived from the input grammar by adding the production:

```
Output ::= Front*
```

and replacing the productions for Front and Abstract by the productions:

```
Front ::= '\n' Title '\n' Abstract
Abstract ::= ('\n' Paragraph)+
```

The resulting tree is populated by data drawn from the original text.

Indexing requires a special case of text transformation typically performed at the time that a text is stored into a document database. The text modelling technique described in this paper has been used to specify text indexing by grammar transformations (Tague *et al.* 1991, Salminen & Tompa 1992, Salminen *et al.* 1995). In subsequent document retrieval, the external representation of retrieved documents may also be regarded as a transformation of the documents in the database. For example, in the output some specific layout features may be added or some parts may be excluded. This application of text transformations has been described, for example, by Kilpeläinen *et al.* (1990), Salminen and Watters (1992), and Kuikka and Penttonen (1993).

Finally, retrieval can also be seen as a special case of text transformation, with an implicitly specified output filter. For example, the retrieval of the front parts of selected papers could be expressed as the following transformation:

Input: Paper

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

Output: Front

```
Front{FrontMatter}::= Title Author+ Location Abstract
```

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 productions, 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.

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”:

Input: Paper

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

Output: Paper

```
Paper {::Peek} ::= Front Body Back
Body ::= Section{::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.

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:

Input: Paper

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

Output: Papers

```
Front{FrontMatter}::= Title  Abstract
```

The result is a single tree 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

For persistent databases, there is also a need to define views. Using text transformations, views may be defined 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 views are defined to create a uniform collection (Clarke *et al.* 1995, Quass *et al.* 1995). By giving detailed definitions for the operations applied to persistent data, the model provides a framework within which to analyze the issues concerning text update, view update, materialized views, and consistency and redundancy control as has been done in the context of relational databases (see, for example, Brodnik & Tompa 1993, Raymond *et al.* 1996).

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. On the other hand, 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 (Raymond 1996). The hypertext structure may exist statically as an alternative structure, or it may be created dynamically in response to a user request (Raymond & Tompa 1988, Watters & Shepherd 1991, Tompa *et al.* 1993). The modelling facilities of this paper have been used to describe a framework for specifying dynamic hypertext structures by Tague *et al*. (1991), Salminen and Watters (1992), and Salminen *et al*. (1995). In those papers, the parts chosen to form the hypertext structures were specified by simple properties. Having the extended power of a filter as defined in this paper, we have much more expressive capabilities to specify a set of parts for a hypertext structure.

7. Conclusion

In this paper we have described an information modelling facility using structured text. It offers a framework in which the semantics of operations may be clearly defined. In a model, the structure of text is defined by a context-free grammar and the structured text instance is given by a parse tree. For a grammar and a parse tree, we defined the notions of text types and parts. These definitions give us the possibility to define the notions of content and value equality for parts. Starting from this framework, other forms of equivalence relation may be developed in a similar manner. For example, an equivalence relation in which the unnamed character strings between named parts are ignored might be useful in some applications: if SGML text is parsed such that the tags are regarded as unnamed delimiters, then their inclusion or omission would not affect a comparison for equality. The notions of equivalence give a basis for defining consistency, as required of a data model (Raymond *et al*. 1996).

A major contribution is the notion of a filter for specifying a set of parts from a parse tree. A simple filter is an annotated constrained 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 constrained grammars further along the filter chain. We characterized how operations based on filters could be defined for data validation, data retrieval, text transformations, text update, view definitions, and for creating hypertext structures. We also showed that retrieval, update and view definitions may be considered to be special cases of transformations.

The detailed definition of the transformation operation is an interesting area for further study. We have asserted that parts of the text are not altered except as required to meet the constraints specified by an output filter. One approach to defining the transform is to associate costs with various atomic operations and to choose a minimal cost change that meets the output constraints (Keller 1986). Under what conditions is there a unique minimal cost change such that the update semantics seems natural? If a sequence of update operations are specified, incorporating the notion of versions would also be useful.

Text is often divided into multiple concurrent hierarchic structures, such as logical sections and physical pages (Sperberg-McQueen & Burnard 1994, Raymond 1996). Having shown how to manage text defined by one grammar, the model can be extended to cover text defined by several grammars simultaneously.

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 constrained 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. The user may then add constraints to the type occurrences, using a two-dimensional template (Kuikka & Salminen 1996). The template-based specification capability has been implemented in a prototype system, which have not yet 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 the future 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. 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

Aho, A.V. and Ullman, J.D., *The Theory of Parsing, Translation, and Compiling, Vol. 1 & Vol. 2*, Prentice-Hall, Inc., 1972.

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

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

Blake, G.E., Consens, M.P., Davis, I.J., Kilpeläinen, P., Kuikka, E., Larson, P.-A., Snider, T., and Tompa, F.W., Text / relational database management systems: overview and proposed SQL extensions, Univ. of Waterloo, Dept. of Computer Science Technical Report, CS-95-25 (June 1995).

Brodnik, A., and Tompa, F.W., A new architecture to support database updates through views, Univ. of Waterloo Centre for the New OED Technical Report, OED-93-02 (July 1993), 36 pp.

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

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

Clarke, C.L.A., Cormack, G.V., and Burkowski, F.J., Schema-independent retrieval from heterogeneous structured text, *Proc. 4th Ann. Symp. on Document Analysis and Information Retrieval*, Las Vegas, April 1995.

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

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

Davis, I.J., 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, February 12, 1996, 42 pp.

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

Goldfarb, C.F., *The SGML Handbook*. Oxford University Press, Oxford, UK, 1990.

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

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

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

ISO IEC 10179:1996, Information technology - Text and office systems - Document Style Semantics and Specification Language (DSSSL), International Organization for Standardization, 1996.

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

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

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

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

Kuikka, E. and Salminen, A., Two-dimensional filters for structured text, *Information Processing & Management* (accepted 1996).

Macleod, I.A., Storage and retrieval of structured documents, *Information Processing & Management* 26, 2 (1990), 197-208.

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

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

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

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

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

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

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

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

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

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

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

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

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

Tsichritzis, D. and Lochovsky, F., *Data Models*, Prentice-Hall, Inc., 1982.