11-1991

A Reconstruction of Context-Dependent Document Processing In SGML

Allen Brown Jr.

T. Wakayama

Howard A. Blair
Syracuse University, School of Computer and Information Science, blair@top.cis.syr.edu

SU-CIS-91-37

A Reconstruction of Context -Dependent Document Processing in SGML

A. L. Brown, Jr., T. Wakayama, and H. Blair

November 1991

School of Computer and Information Science
Syracuse University
Suite 4-116, Center for Science and Technology
Syracuse, New York 13244-4100

A Reconstruction of Context-Dependent Document Processing in SGML

Allen L. Brown, Jr., Toshiro Wakayama and Howard A. Blair

Xerox Corporation, Webster Research Center, Webster, New York, U.S.A.

ABSTRACT: SGML achieves a certain degree of context-dependent document processing through *attributes* and *linking*. These mechanisms are deficient in several respects. To address these deficiencies we propose augmenting SGML's `LINK` and `ATTLIST` constructs with two new mechanisms, *coordination* and (rule-based) *attribution*. The latter can be used to specify the *result* of context-dependent processing in a uniform fashion while considerably increasing SGML's expressive power. We illustrate this enhanced power by sketching a specification of (the result of) document layout that can be encoded in SGML augmented with coordination and attribution.

1 Introduction

Our research enterprise is to develop a fully articulated foundation for *document representation*. To that end we wish to understand the informal and intuitive models of structured documents, and codify that understanding in a document description language that can be interpreted by computers. Specifically, we seek a *declarative*, constraint-based description language for structured documents that:

- has a precise formal semantics;
- separates the specifications of logical structure, form, and content;
- is fully expressive of the constructs typical of traditional (procedural) document description languages that are mainly concerned with document processing applications (*e.g.* interchange, layout, and recognition).

While the first objective above has not been a particular technical concern of the designers of SGML, our formalism [BW91] could easily be used to give SGML a formal semantics. The second objective is a central concern of the designers of SGML as is evidenced by the following

> *...Generalized markup is based on two novel postulates:*
>
> *a. Markup should describe a document's structure and other attributes rather than specify processing to be performed on it, as descriptive markup need be done only once and will suffice for all future processing.*

> *b. Markup should be rigorous so that the techniques available for processing rigorously-defined objects like programs and data bases can be used for processing documents as well.*
>
> C.F. Goldfarb [Gol90]

With respect to our last objective above, the designers of SGML would clearly like such document processing descriptions to be possible, though they see such descriptions to be outside the scope of (SGML) document specification. SGML, as the designers themselves suggest [Gol90, §D.3 and §D.6], is in general insufficiently expressive to specify such processing. Indeed, the designers hint at a "rule-based" language [Gol90, p. 100] using SGML syntax to that end.[1] Our task here is to develop an illustrative fragment of a document description language incorporating rule-based constructs that will enable the rigorous and direct specification of document processing in addition to specifying document structure.[2] This language remains subject to a strictly enforced design principle: While the specification of the logical structure of a document can affect document processing, the latter should not affect the former. For example, a journal article's logical structure should independent of whether it is to be ultimately subjected to layout processing in the ACM style or the IEEE style.

The document representation that we favor has a number of components:

- Regular right-hand part grammars (RRHPGs) and parse trees: RRHPGs are used to specify structural constraints. A DTD is a (concrete) version of such a structural constraint specification (a source grammar) for a class of source documents. We also use such constraints (in a result grammar) to specify the structure of the result of a particular application (*e.g.* layout processing), *as well as* the structural correspondences between constituents of the source document and that result, *e.g.* a coordination between logical structure and layout structure. Parse trees describe concrete instances of structures compatible with an associated constraining grammar. Thus an SGML document instance is, in effect, a parse tree compatible with the grammar given by its DTD. Other parse trees, compatible with other mentioned grammars, will be created in the course of satisfying a result specification.

- Rule-based attribution and attributes: Attributes are associated with every grammar (just as they are associated with DTDs). The specification of the

[1] The proposed document style semantics and specification language [Adl91] is a related attempt to associate document formatting semantics with an SGML specified document.

[2] More precisely–and in accord with our declarative inclinations–we declaratively specify what the *result* of document processing is to be.

values that those attributes may assume is given by *definite clauses* [Llo87, SS86] (*i.e.* Prolog-like facts and rules).

- Coordination and embedding: Coordination grammars constrain structural correspondences between structural elements specified by source and result grammars respectively. An embedding is an instance of such a correspondence.
- Preference: Though the components mentioned above are *sufficient* to specify any computationally effective result, many such results are best specified as optimizations according to particular preference criteria. While the language of preference is beyond the scope of the present paper, this form of specification is nonetheless central to our overall program of document representation.

The remainder of the paper unfolds as follows: We first give a characterization of (the results of) context-dependent document processing in general and its particular articulation within SGML. We then describe rule-based attribution and coordination in the context of *abstract* syntax, using layout as our illustrative document processing application. Finally, we present our proposed embedding of attribution and coordination in a *concrete* SGML syntax.

2 Context-Dependency in Document Processing

In this section, we discuss what context-dependency is in document processing and why we need it. To this end, we first present document abstractions that underly our document representation model, and underly others such as SGML and ODA. We then show that the idea of context-dependent document processing is a natural consequence of these abstractions, and that it is also a key idea in specifying such processing. Finally, we show that the document representation model based on these abstractions offers a framework in which various types of context-dependent document processing can be classified.

2.1 A Document Representation Model

One of the most prominent properties of the electronic document is its amenability to flexible processing of various kinds. For instance, it can be processed for effective visual presentation (layout processing). It can be processed for answering queries (query processing). It can be manipulated for extracting and re-organizing a specific view of a document (view processing). In order to maximally exploit the potential of electronic documents, modern document representation languages support the following abstractions:

- the (processing) abstraction of a uniform *source* representation of documents which is free from any specific processing considerations. Hence a document so represented can be used for a variety of processing applications. This includes, for instance, a class of layout processing such as the ACM-style processing and the IEEE-style processing.

- the (structure) abstraction of a common organization of documents so that processing can be specified for a class of documents, not just for an individual document instance.

The processing abstraction as described above immediately implies that:

- the processing specification is external to the source document, and hence it must have its own representation;

- if the source document and its processing have different representations, there must be a way of associating the two when the processing of a document is demanded.

The structure abstraction, on the other hand, insists that there be a way of specifying such associations for a class of documents via their common structure, not just for individual document instances. Note that in this model, a document is not an instance of "tagged" content (*e.g.*, a document with markup tags inserted), but it comes with an abstracted, external structure (*e.g.*, a structurally organized set of markup tags) which has its own formal realization (*e.g.*, document type definition in SGML). Thus, document processing can now be specified with respect to such an externally defined formal structure. Most document representation languages, including SGML, fail to take full advantage of their structure abstractions.

In our formal model of documents, the external structure common to a class of documents is specified by an *attribute grammar* (see section 4) A document instance of this class is a *parse tree* derived by the grammar. As to the language for specifying processing, we use the same formal system based on attribute grammar. As we commented earlier, our specification of document processing is declarative, *i.e.*, we only specify the desired outcome of processing, which can be viewed as yet another document. This uniformity in formal notation for representing documents and (declaratively) their processing gives a *closure* property to our formal system of document processing: *i.e.*, a document after a processing is another document, which in turn can receive another processing, and so on.

Thus, in our model of document representation (see figure 1), a *source document* consists of a *source grammar* and a *source document instance*. When processing is applied to a source document instance, we get a *result document instance*, whose

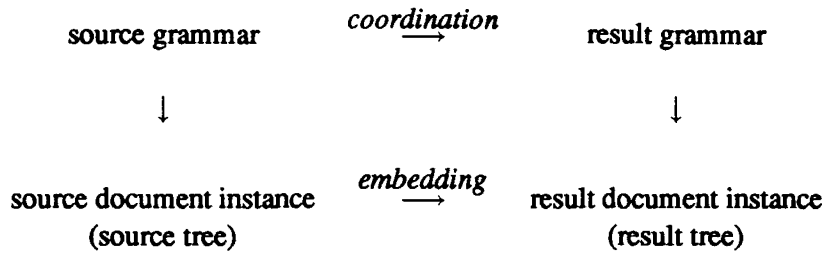


Figure 1: Document representation model.

structural properties are given by a *result grammar*. Note that the document processing captured in a result grammar is *generic, parameterized* processing in the sense that it can take any source instance of the corresponding source grammar. Note also that this generality of processing specification is a natural consequence of the two abstractions discussed above: the processing abstraction demands that there be a separate description of processing, and the structure abstraction insists that this description be general enough to accommodate a class of source document instances. However, a naive adherence to these abstractions might easily result in a document processing system which is blindly insensitive to peculiarities of individual source instances. Our notion of context-dependent document processing makes generic document processing maximally sensitive to individual differences of source instances. Coordination and (rule-based) attribution are the two main mechanisms to implement such processing: coordination offers a way of referring to nodes of the source tree, and attribution expresses attribute dependency among *nearby* nodes where the nearness metric is relative to paths in source, result and coordination trees.

2.2 A Characterization of Context-Dependent Document Processing

The model of document representation introduced above views document processing of a specific source instance[3] as the generation of result trees from the source instance, the pair of source and result grammars, and the coordination defined for the pair. Note that document processing is, therefore, a generalization of parsing. The usual parsing in single-grammar settings is document processing where,

- the result grammar is sufficiently trivial that source trees are simply unstructured strings;
- the coordination is empty.

[3]Unless otherwise noted, the term *document processing* refers to the processing of a specific source instance, not the generic processing specified in the result grammar.

In this case, if the result grammar is context-free, the expansion of a nonterminal node in each step of parsing is determined by the implicit linear order of the input string and does not depend on any of its *nearby* nodes. The general setting of parsing in document processing, on the other hand, is strongly *context-dependent*[4] (although in a well-controlled way) in the following sense:

- the input to document processing is not simply a string but is an attributed parse tree, where each step of parsing in general depends on both the subtree structure, designated by the coordination, of this source tree, and on the portion of the input string spanned by the subtree;
- each step of parsing must compute the attributes of the node to be expanded, but these attributes may depend on attributes of the nearby nodes.

We will refer to a collection of nodes, either from the source tree or from the result tree, on which each step of parsing depends as a *scope* of context-dependency. Figure 2 (the first column) classifies various scopes of context-dependency. Since these scopes have natural manifestations at a more abstract level as grammar objects, such as nonterminals and productions, we have included this correspondence in the figure (the second column).

The first three rows are the cases where scopes are within a result tree. In SGML, this type of context-dependency is called *processing state dependency*.[5] In this class, the smallest scope is a single nonterminal (*e.g.*, ⟨`word-block`⟩). In this case, its various attribute values (*e.g.*, font attribute values such as italic) must be directly provided, and can not be sensitive to attribute values of its nearby nonterminals. However, ⟨`word-block`⟩ instances in a paragraph of the main body (⟨`para-block`⟩) may have attribute values different from those of ⟨`word-block`⟩ instances in a figure caption (⟨`caption-block`⟩). If the result grammar has a production of the form,

$$\langle \texttt{caption-block} \rangle \longrightarrow \cdots \langle \texttt{word-block} \rangle^{+} \cdots$$

attribute values of ⟨`word-block`⟩ can be specified relative to this production. Moreover, they can be computed by using attribute values of ⟨`caption-block`⟩ (the second row of figure 2). This mechanism is absent in SGML, despite the fact that SGML has a formal, external representation of a common structure of a class of documents as a DTD. SGML fails to take advantage of structural relationships

[4]The term *context-sensitive* has a specific technical meaning in formal language theory. Hence we have chosen to use the term 'context-dependent' in the instances where the SGML community would use 'context-sensitive'.

[5]Consistent with the notation detailed in section 4, we indicate nonterminal symbols of a grammar by descriptive terms embraced by ⟨⟩.

a source tree ($\mathcal{T}_{source}$) and a result tree ($\mathcal{T}_{result}$)	a source grammar ($\mathcal{G}_{source}$) a result grammar ($\mathcal{G}_{result}$)
a node of $\mathcal{T}_{result}$	a nonterminal of $\mathcal{G}_{result}$
a height-one subtree of $\mathcal{T}_{result}$	a production of $\mathcal{G}_{result}$
a height-n subtree of $\mathcal{T}_{result}$	a chain of productions of $\mathcal{G}_{result}$
+ a node of $\mathcal{T}_{source}$	+ a nonterminal of $\mathcal{G}_{source}$
+ a height-one subtree of $\mathcal{T}_{source}$	+ a production of $\mathcal{G}_{source}$
+ a height-n subtree of $\mathcal{T}_{source}$	+ a chain of productions of $\mathcal{G}_{source}$

Figure 2: A classification of *scopes* of context-dependency in document processing.

of nonterminals (*e.g.*, how ⟨`word-block`⟩ is related to ⟨`caption-block`⟩) in specifying their attribute values. The third row of figure 2 shows a natural extension of this idea, *i.e.*, the scope of context-dependency may be a chain of productions as opposed to a single production. For instance, ⟨`word-block`⟩ may not be related to ⟨`para-block`⟩ in a single production but through a chain of productions.

A scope in figure 2 preceded by '+' (the last three rows) is a scope in the source tree which augments a scope in the result tree. A combination of a pair, one from the first three rows and the other from the last three rows, defines a scope of context-dependency extended by the coordination. We will see two examples of such coordination in later sections: syntactic coordination (section 5) and semantic coordination (section 6). Syntactic coordination specifies scopes of context-dependency in the source tree, but no attribute values pass from the source scopes to the result tree. However, the identification of a source scope itself may require examination of the attributes within the source scope. When attribute dependency extends from the result tree to the source tree, we have the case of *source attribute dependency*, which can be accommodated by semantic coordination.

Implicit in the above characterization of context-dependency is another dimension of characterization, *i.e.*, the complexity of attribute computation. In the simplest case, an attribute of a nonterminal can be specified directly (*e.g.*, the font of ⟨`word-box`⟩ is italic). The specification can also be rule-based (*e.g.*, the font of a ⟨`word-box`⟩ instance is italic *if* the font of the preceding ⟨`word-box`⟩ instance is Roman). A rule-based specification can also be parametrized (*e.g.*, the font of a ⟨`word-box`⟩ instance is twice that of the preceding ⟨`word-box`⟩ instance.

3 Context-dependent Document Processing in SGML

In this section, we assume that the reader is familiar with elementary SGML constructs such as *element declarations* and *attribute declarations*. Suppose we have the following element declaration,

```
<!ELEMENT book -- (font?,body,rear?) +(quote)>
```

and that we want to specify the font of `quote` elements context-dependently, *i.e.*, italic when they appear in the `front` part of the `book` while bold italic when they appear in the `body` part. In SGML, this may be accomplished as in figure 3. Note that the `font` attribute of the `quote` element is directly specified, there is no

```
⟨! LINK #INITIAL
        front  #USELINK q-style1
        body   #USELINK q-style2  ⟩

⟨! LINK q-style1
        quote [font=italic ] ⟩

⟨! LINK q-style2
        quote [font=bold-italic ] ⟩
```

Figure 3: Syntactic context-dependency.

rule-based dependency expressed.

Now suppose that we want to determine the font of `quote` elements according to the values (`direct` or `indirect`) of their source attribute `q-style`. Figure 4 illustrates how this may be accomplished. Note that those phrases in double quotes

```
⟨! LINK q-style2
        quote [usage = "q-type EQ direct"
              font=boldital ]
        quote [usage = "q-type EQ indirect"
              font=italic ] ⟩
```

Figure 4: Semantic dependency (source attribute dependency).

are *outside* the DTD language of SGML, and must be interpreted by the processing application. Hence, the semantics of this `!LINK` declaration's assignment of values to attributes becomes application-dependent.

The example of figure 5 shows that layout processing of an element depends on layout processing of some other elements. This last example leaves the details of associating values with attributes to an unspecified (to SGML parsing) application. The main message to be gleaned from the examples of this section is that document processing context-dependencies of extended scope or complexity are specified externally (to SGML). We aim to *internalize* such specifications.

```
⟨! LINK q-style1
        quote [usage = "curfont EQ italic"
              font=boldital ]
        quote [usage = "curfont NE italic"
              font=italic ] ⟩
```

Figure 5: Semantic dependency (processing state dependency).

4 Context-Dependency via Attribute Grammars

An attribute grammar scheme (AGS) provides a computational mechanism for specifying families of ordered, finite, labelled trees, and thereby characterizes a class of hierarchically structured documents. An AGS consists principally of a finite set (a grammar) of *regular right-hand part* (RRHP) productions [Cha87] over a finite vocabulary of terminal and nonterminal symbols, and (for each production) an associated set of definite clause schemes presented in Prolog-like syntax. The clauses serve to define attributes[6] associated with the nonterminal symbols of the grammar. We introduce AGS's by way of the example in figure 6 which consists of productions describing a document in terms of pages, pages in terms of lines, lines in terms of words, and words in terms of individual vocabulary items. Thus, we are specifying context-dependent document processing in the single grammar sens of section 2.2. The notation of the figure indicates nonterminal and terminal symbols in productions by embracing them in (respectively) ⟨⟩ and []. As usual [Har78], the left-hand side of a production is a nonterminal symbol, and, in the case

[6] In contrast to classical attribute grammars (AG's) [DJL88], AGS's define attributes in a *relational* rather than a *functional* fashion. That is, attributes are relations between items (nodes of parse trees in our formalism) and values rather than functions from such nodes to values.

of the grammar of the figure, the start symbol is ⟨doc⟩.[7] The right-hand sides of the productions are regular expressions over (terminal and nonterminal) symbols. Thus a ⟨doc⟩ may expand to a finite, non-empty sequence of ⟨page⟩s; a ⟨page⟩ may expand to a finite, non-empty sequence of ⟨line⟩s; a ⟨line⟩ may expand to a finite non-empty sequence of ⟨word⟩s; and a ⟨word⟩ may expand to one of [the], [quick], *etc.* Symbols of the grammar are superscripted with scheme variables (in parentheses). These variables will play a role in the definitions of attributes.

The clause schemes associated with the productions of an AGS mention two kinds of predicate symbols, *attributes* and *free* predicates. The two types are distinguished by the fact that the first term of an attribute predicate in a clause scheme is always written in an italic font, *e.g.* $lmt(y)$ in pguard. Attributes are defined by the collections of clause schemes associated with the productions of an AGS, while free predicates (like plus in the figure) are defined by an interpreter associated with the document processing system making use of the AGS. The italicized terms in the attributes are *schematic position expressions*, that is, expressions over position variables. These will be replaced by the nodes of a parse tree according to a discipline that we will describe below. Each parse tree of the ⟨doc⟩ grammar will induce a logic program as a consequence of that replacement process.

In figure 7 we present a parse tree of the ⟨doc⟩ grammar for the string of symbols [the] [quick] [brown] [fox] [jumps] [over] [the] [lazy] [dog]. We have labelled each node of the tree both with its associated grammar symbol and a unique position atom (d_0, d_1, *etc*). Each node of the parse tree corresponds to a symbol in the head of some production of the grammar. Thus the node will be identified with the position variable associated with that production's head symbol. Each node of the grammar that is an immediate child of some other node will also correspond to an instance of a symbol in the body of the production of the grammar in which the parent node figures as the head. Thus the child node will also be identified with the position variable of the corresponding body symbol. For the parse tree at hand, d_0 corresponds to the head symbol of the ⟨doc⟩ production and is thus identified with the position variable, x, that superscripts ⟨doc⟩. The nodes d_1 and d_2 correspond to (distinct) instances of the symbol ⟨page⟩ and are thus identified (each in turn) with the variable y. Thus from the first production of the grammar we get the clause substitution instances enumerated in figure 8. This set of clauses is still not fully instantiated as we find therein position expressions like $lmt(d_1)$ and $lt(d_2)$, the first denoting the leftmost sibling of d_1 and the second denoting the left sibling of d_2 (both with respect to their parent, d_0); similarly, rt and rmt respectively denote

[7] We will typically associate a grammar (AGS) with its start symbol. Thus we speak of the ⟨doc⟩ grammar (AGS).

right and rightmost siblings. Since the leftmost sibling of d_1 is d_1 and the left sibling of d_2 is also d_1, we replace the position expressions by the nodes denoted and get the set of clauses of figure 9. Notice that there is no clause derived from the clause that contained the position expression $lt(d_1)$ since there is no node left of d_1. If we carry out the instantiation process for the node d_1 of the parse tree with respect to the clause schemes attached to the ⟨page⟩ production for which those nodes correspond to head symbols, we obtain the clause instances of figure 10. We continue in this vein to generate a complete set of clause instances corresponding to the parse tree of figure 7. The resulting set not only fails to instantiate clause schemes referring to nonexistent parse tree nodes (like $lt(d_1)$, but also suppresses duplicate instances of clauses.

An instance, then, of the class of documents defined by an AGS is a parse tree according to that AGS together with the logic program induced by that tree. If t is a parse tree, we denote the generated (according to some particular AGS that will be clear from context) logic program as π_t. We will say that a parse t, having root node $\hat{t}$, is *admissible* if guard($\hat{t}$) is a logical consequence of the program π_t, that is, if a Prolog interpreter would respond "yes" to the query ?- guard($\hat{t}$). Since the grammar of figure 6 is highly ambiguous, there are many possible parse trees of the string [the]...[dog]. However, there is precisely one parse tree that is admissible for fixed values of the constants maxct (the maximum number of ⟨line⟩s on a ⟨page⟩) and maxln (the maximum number of characters on a ⟨line⟩), taken to have respective values of 2 and 15. Moreover, for the single admissible parse tree corresponding to a particular content string, each ⟨page⟩ node of the parse tree will have a lnct attribute whose value will be maxct unless the ⟨page⟩ node in question is the last ⟨page⟩ of the ⟨doc⟩ument, in which case the value will be nonnegative and at most maxct. Similarly, each ⟨line⟩ node of the parse tree will have a lnln attribute whose value will be maxln unless the ⟨line⟩ node in question is the last ⟨line⟩ node descendant of its parent ⟨page⟩. In the latter case the value will be nonnegative and at most maxln. Thus, the admissibility of a parse tree hinges not only on its syntactic structure but also on the context established by the values assumed by the attributes of the nodes of that tree. The attributes have been constrained in such a way as to compute the first-fit line breaking of the content string in the case of the unique admissible parse tree. To see how this comes to pass we need to examine the clause schemes of figure 6 more closely.

The parse tree rooted at the node d_0 is admissible just in case guard(d_0) is true, or just in case the pguard attribute of d_1 is t where d_1 is the leftmost ⟨page⟩ child of d_0. A necessary condition for the pguard attribute of a ⟨page⟩ node's being t is that the guard of that node be true and that the pguard attribute of each of its right sibling ⟨page⟩ nodes be t. Another necessary condition is that the ⟨page⟩

node's `lnct` attribute be the maximum line count (`maxct`) and that the value of the `wdlln` attribute of that node–the length of its leftmost ⟨`word`⟩ child–be less than the value of the ⟨`page`⟩ node's `wdrem` attribute. Now for `guard` to be true of a ⟨`page`⟩ node it is necessary for the values of the `pguard` attribute of each of the node's ⟨`word`⟩ children to have the value `t`. Moreover, the value of the `wdrem` attribute (the number of characters remaining on the last line) of such a child node must have a value numerically less than the value of the `wdlln` attribute of the first ⟨`line`⟩ child of that ⟨`page`⟩ node. These conditions are the essentials of a set of first-fit constraints.

5 Context-Dependency via Syntactic Coordination

As discussed in section 2, syntactic coordination is a way of specifying source scopes. In this section, we will illustrate this simpler form of coordination through an example (figure 11). Since there is no transfer of attribute values from the source tree to the result tree, we have chosen an example involving no attributes, and hence our notations in the example are accordingly simplified (*e.g.*, no position variables are shown).

In this example (see 11), the source document is a bilingual manual (⟨`man`⟩) consisting of the English part (⟨`esec`⟩) and the Japanese part (⟨`jsec`⟩). Each part has the usual paragraph structure. The finer structure below paragraph is not a concern for this example, so we elide it. The result document (⟨`man-lay`⟩) is a sequence of pages (⟨`pg`⟩), and each page has the left column (⟨`lc`⟩) and the right column (⟨`rc`⟩). Again, we are not concerned with internal column structures. The coordination, *i.e.*, the ⟨`esec`⟩-coordination grammar and the ⟨`jsec`⟩-coordination grammar, enforces that the English part (the Japanese part) of the manual be displayed in left (right) columns.

Figure 12 shows a pair of source and result (partial) trees. A part of the embedding of the source tree in the result tree is also shown. This node-to-node correspondence of the embedding is induced by the coordination together with the essential properties of the notion of embedding (*e.g.*, maximal preservation of the left-to-right order of the source tree, and preservation of the ancestral relation of the source tree). [Toshiro: need to indicate how word order might have shifted content in ⟨`man-lay`⟩ relative to ⟨`man`⟩.] Thus, for instance, the scope of context-dependency for the node e_4 is the subtree rooted at the node d_1, whereas that of the node e_5 is the subtree rooted at d_2. The scope of e_6, on the other hand, is that part of the subtree rooted at d_1 which is left unconsumed by the node e_4, and so on.

6 Context-Dependency via Semantic Coordination

The type of context-dependency we will discuss in this section is known as *source attribute dependency* in SGML. For instance, the font attribute (a result attribute) of some words might depend on some source attributes of those words, and might not be completely determined by result attribute values of nearby nodes. The example in figure 13 illustrates this point.

The source grammar in this example simply describes a sentence structure with quotation. The quotation ⟨qt⟩ has a quotation-type attribute `q-type`, whose value is to be specified by the user (the attribution clause associated with the first production). The quotation ⟨qt⟩ is in turn a sequence of ⟨word⟩'s, each of which inherits the `q-type` attribute from ⟨qt⟩. The result grammar specifies that a ⟨line⟩ is a sequence of ⟨box⟩'s, and that the `font` attribute of ⟨box⟩ depends on another attribute `q-type` of ⟨box⟩. The coordination states that a ⟨box⟩ is a container of ⟨word⟩, and that each ⟨box⟩ object inherits its `q-type` attribute value from its corresponding ⟨word⟩ object.

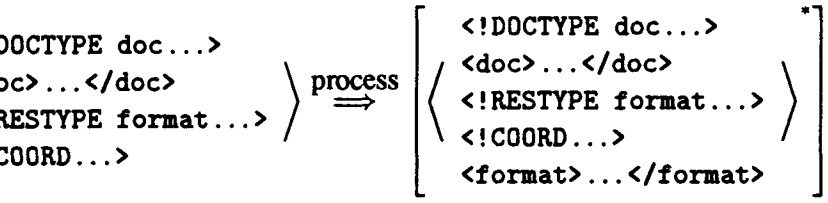
Figure 14 shows a pair of source and result trees, together with the node-to-node correspondence. It also shows the logic programs induced by those trees and by the ⟨word⟩-coordination tree.

7 Introducing Attribution and Coordination into SGML

Until this point our exploration of facilities to support context-dependent document processing has been couched in terms of abstract syntactic descriptions of documents. In this section we shall suggest a particular way of incorporating the facilities of coordination and rule-based attribution into a concrete SGML syntax. Our aim here is not to precisely specify an extension to SGML's formal syntax, but rather to give examples of the new facilities in a form compatible with current SGML syntax. Nonetheless, we believe it entirely straightforward to specify formally such an extension and are proceeding to do so to support the archival document representation needs of various research projects we have under way. We shall use the "direct quote" example of section 6 as the subject matter to illustrate concrete instances of the suggested extensions.

From the perspective of context-dependent document processing an SGML document has three essential components: the document type definition (DTD), the link process definition (LPD) and the document instance (DI). These items typically appear in an SGML document in the order given. It also happens that the DTD and DI are the only required components of an SGML document. We introduce three new optional components: the result type definition (RTD), the coordination definition (CD), and the result instance (RI). Mixing SGML and ODA

[WCGH+86] terminology we think of a source document consisting of a DTD, DI, RTD, and CD as an *editable, processable source document*, and the product resulting from processing processing an *editable, processable result document sequence*.[8] Schematically document processing may then be represented as



In figure 15 we present in concrete "extended" SGML syntax the specification given abstractly in figures 13 and 14. The figure represents an editable, processable result document. It is partitioned into segments bounded by SGML comments of the form `<!-- beginning ... -->` and `<!-- end ... -->`. The document exclusive of the RI portion constitutes an editable, processable source document. The DTD, RTD and CD portions of the document correspond to the abstract source AGS $\mathcal{G}_{\langle \texttt{sen} \rangle}$, result AGS $\mathcal{G}_{\langle \texttt{line} \rangle}$ and coordination $\{\mathcal{G}_{\langle \texttt{word} \rangle}\}$ of figure 13. The DI and RI portions are, in effect, linear encodings of abstract ⟨sen⟩ and ⟨line⟩ parse trees. The `!DOCTYPE` component presented is an extension of the standard SGML declaration of that sort. Each of the components `!DOCTYPE`, `!RESTYPE` and `!CTYPE` can be thought of as the same type of declaration applied to different descriptive purposes, namely source, result and coordination specifications. As their functional roles should already be clear from the descriptions of their abstract equivalents in section 6, we will turn immediately to explaining the nature of their descriptive augmentation with respect to ordinary SGML DTD's.

The first thing to notice is that an `!ATTLIST` component admits a new kind of value field, `ATOM`, and a new kind of default field, `#RULED`. The former says that the value of the attribute being defined is to be a Prolog atom, and the value (by default) is actually to be computed by a Prolog rule. Now observe that `!ELEMENT` declarations rather than merely mentioning other `!ELEMENT` identifiers in their content models pair such identifiers with Prolog (position) variables. These variables will be instantiated in the course of parsing (be it a DI according to a DTD or an RI according to an RTD) in exactly the way described in section 4. The `!ELEMENT` definition may also contain a sequence of Prolog rules, introduced by `!RULES`. These will actually specify values for attributes associated with the declared `!ELEMENT` by the mentioning `!ATTLIST` when those attribute values have the `#RULED` default. As in the abstract presentation, we have `guard` attributes that

[8] LPD's may also occur in the source document and result document sequence. Strictly speaking, LPD's are functionally redundant with respect to the new facilities being proposed.

determine the admissibility of parse (sub)trees (or alternatively admissibility of substructures of of DI and RI structures). By default every `!ELEMENT` has a `guard` defining rule even if not explicitly mentioned (as it is in the case of that for the `word` `!ELEMENT` declaration). When not explicitly mentioned, as in the case of the `sen` `!ELEMENT` declaration, it is presumed to be the fact `guard(X0)`. where the variable mentioned is always the position variable associated with the `!ELEMENT` being declared.

The DI (`<sen>...</sen>`) and RI (`<line>...</line>`) are simply linear encodings of the abstract parse trees decorated with attribute value pairs. Attributes are mentioned in DI's and RI's either because they are explicitly given or because they are implicitly established via mentioned defaults. Thus the `qt` tag has an attribute value because a default of (`direct`) was indicated. The `box` tags having defaults acquire them via rules contained in the RTD and CD.

8 Conclusions

Our informal and intuitive model of structured documents is based on the processing and structure abstractions. The processing abstraction demands separate representations for source documents and their processing. The structure abstraction insists that these representations encompass a class of documents via their common structure. The formal encoding of these observations results in a model of document representation consisting of a pair of source and result grammars, a pair of source and result trees, and the coordination defined for the pair of grammars. The document processing in this model is then the generation of result trees given the rest. The idea of context-dependent document processing is to make the generic processing captured (declaratively) in the result grammar maximally sensitive to individual differences of source instances. The main mechanisms to implement such context-dependency are (rule-based) attribution and coordination.

SGML clearly supports the two abstractions, at least conceptually. But its formal encoding of these abstractions is regrettably weak, owing to its particularly rudimentary mechanisms that correspond to coordination (linking) and attribution (direct specification of attribute values). We have shown how these SGML constructs fall short of realizing context-dependent processing, and how coordination and full attribution extend their corresponding SGML constructs. Finally, we have shown that our coordination and attribution can be straightforwardly incorporated into the SGML-like syntax.

References

[**Adl91**] Sharon Adler. Iso/iec cd 10179, information technology - text and office systems document style semantics and specification language (DSSL). Technical Report DIS 10179, International Standards Organization (ISO), Geneva, 1991.

[**BW91**] Allen L. Brown, Jr. and Toshiro Wakayama. Assigning meaning to markup: A study of the logical foundations of document representation. Manuscript to be submitted for publication, 1991.

[**Cha87**] Nigel P. Chapman. *LR Parsing: Theory and Practice.* Cambridge University Press, Cambridge, 1987.

[**DJL88**] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *Attribute Grammars: Definitions, Systems and Bibliography*, volume 323 of *Lecture Notes in Computer Science.* Springer-Verlag, New York, 1988.

[**Gol90**] Charles F. Goldfarb. *The SGML Handbook.* Oxford University Press, Oxford, 1990.

[**Har78**] Michael A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, Reading, Massachusetts, 1978.

[**Llo87**] John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, second edition, 1987.

[**SS86**] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* MIT Press, Cambridge, Massachusetts, 1986.

[**WCGH+86**] Henri C. Weisz, Ian R. Campbell-Grant, Roy Hunter, Roy Pierce, L.J. Zeckendorf, and Barry J. Woods. Information processing, text and office systems, office document architecture (ODA) and interchange format. Technical Report DIS 8613, International Standards Organization (ISO), Geneva, 1986.

$\langle \texttt{doc} \rangle^{(x)} \longrightarrow (\langle \texttt{page} \rangle^{(y)})^+$
  $\texttt{guard}(x) :- \texttt{pguard}(lmt(y), \texttt{t}).$
  $\texttt{pgct}(lmt(y), 1).$
  $\texttt{pgct}(y, \texttt{N}) :- \texttt{pgct}(lt(y), \texttt{M}), \texttt{plus}(\texttt{M}, 1, \texttt{N}).$
  $\texttt{pguard}(rmt(y), \texttt{t}) :- \texttt{guard}(rmt(y)).$
  $\texttt{pguard}(y, \texttt{t}) :- \texttt{pguard}(rt(y), \texttt{t}), \texttt{guard}(y),$
    $\texttt{lnct}(y, \texttt{maxct}), \texttt{wdrem}(y, \texttt{R}),$
    $\texttt{wd1ln}(rt(y), \texttt{L}), \texttt{R} < \texttt{L}.$
$\langle \texttt{page} \rangle^{(x)} \longrightarrow (\langle \texttt{line} \rangle^{(y)})^+$
  $\texttt{guard}(x) :- \texttt{pguard}(lmt(y), \texttt{t}).$
  $\texttt{lnct}(lmt(y), 1).$
  $\texttt{lnct}(y, \texttt{N}) :- \texttt{lnct}(lt(y), \texttt{M}), \texttt{plus}(\texttt{M}, 1, \texttt{N}).$
  $\texttt{pguard}(rmt(y), \texttt{t}) :- \texttt{guard}(rmt(y)).$
  $\texttt{pguard}(y, \texttt{t}) :- \texttt{pguard}(rt(y), \texttt{t}), \texttt{guard}(y),$
    $\texttt{wdrem}(y, \texttt{U}), \texttt{wd1ln}(rt(y), \texttt{V}),$
    $\texttt{U} < \texttt{V}.$
  $\texttt{wd1ln}(x, \texttt{L}) :- \texttt{wd1ln}(lmt(y), \texttt{L}).$
  $\texttt{wdrem}(x, \texttt{R}) :- \texttt{wdrem}(rmt(y), \texttt{R}).$
$\langle \texttt{line} \rangle^{(x)} \longrightarrow (\langle \texttt{word} \rangle^{(y)})^+$
  $\texttt{guard}(x).$
  $\texttt{wdrem}(x, \texttt{M}) :- \texttt{lnln}(x, \texttt{L}), \texttt{minus}(\texttt{maxln}, \texttt{L}, \texttt{M}).$
  $\texttt{lnln}(x, \texttt{L}) :- \texttt{lnln}(rmt(y), \texttt{L}).$
  $\texttt{lnln}(lmt(y), \texttt{L}) :- \texttt{ln}(lmt(y), \texttt{L}).$
  $\texttt{lnln}(y, \texttt{L}) :- \texttt{lnln}(lt(y), \texttt{M}), \texttt{ln}(y, \texttt{N}), \texttt{plus}(\texttt{M}, \texttt{N}, \texttt{L}).$
  $\texttt{wd1ln}(x, \texttt{U}) :- \texttt{ln}(lmt(y), \texttt{U}).$
$\langle \texttt{word} \rangle^{(x)} \longrightarrow [\texttt{the}]$
  $\texttt{ln}(x, 3).$
  $\texttt{guard}(x).$
$\langle \texttt{word} \rangle^{(x)} \longrightarrow [\texttt{quick}]$
  $\texttt{ln}(x, 5).$
  $\texttt{guard}(x).$
    $\vdots$
$\langle \texttt{word} \rangle^{(x)} \longrightarrow [\texttt{dog}]$
  $\texttt{ln}(x, 5).$
  $\texttt{guard}(x).$

Figure 6: First-fit document layout.

Figure 7: First-fit parse tree.

```
guard(d0) :- pguard(lmt(d1), t).
guard(d0) :- pguard(lmt(d2), t).
pgct(lmt(d1), 1).
pgct(lmt(d2), 1).
pgct(d1, N) :- pgct(lt(d1), M), plus(M, 1, N).
pgct(d2, N) :- pgct(lt(d2), M), plus(M, 1, N).
pguard(rmt(d1), t) :- guard(rmt(d1)).
pguard(rmt(d2), t) :- guard(rmt(d2)).
pguard(d1, t) :- pguard(rt(d1), t), guard(d1),
      lnct(d1, maxct), wdrem(d1, R),
      wd1ln(rt(d1), L), R < L.
pguard(d2, t) :- pguard(rt(d2), t), guard(d2),
      lnct(d2, maxct), wdrem(d2, R),
      wd1ln(rt(d2), L), R < L.
```

Figure 8: Partially instantiated clauses of the doc production.

```
guard(d0) :- pguard(d1, t).
pgct(d1, 1).
pgct(d2, N) :- pgct(d1, M), plus(M, 1, N).
pguard(d2, t) :- guard(d2).
pguard(d1, t) :- pguard(d2, t), guard(d1),
      lnct(d1, maxct), wdrem(d1, R),
      wd1ln(d2, L), R < L.
```

Figure 9: Fully instantiated clauses of the doc production.

```
guard(d1) :- pguard(d3, t).
lnct(d3, 1).
lnct(d4, N) :- lnct(d3, M), plus(M, 1, N).
pguard(d4, t) :- guard(d4).
pguard(d3, t) :- pguard(d4, t), guard(d3),
      wdrem(d3, U), wdlln(d4, V),
      U < V.
wdlln(d1, L) :- wdlln(d3, L).
wdrem(d1, R) :- wdrem(d4, R).
```

Figure 10: The clauses induced by the leftmost ⟨page⟩ node.

G_1: Source AGS
⟨man⟩ ⟶ ⟨esec⟩⟨jsec⟩
⟨esec⟩ ⟶ ⟨epara⟩$^+$
⟨jsec⟩ ⟶ ⟨jpara⟩$^+$
⋮

G_2: Result AGS
⟨man-lay⟩ ⟶ ⟨pg⟩$^+$
⟨pg⟩ ⟶ ⟨lc⟩⟨rc⟩
⋮

$G_{\langle esec \rangle}$: ⟨esec⟩-coordination
⟨esec⟩ ⟶ [lc]$^+$

$G_{\langle jsec \rangle}$: ⟨jsec⟩-coordination
⟨jsec⟩ ⟶ [rc]$^+$

Figure 11: Syntactic coordination.

See attachment

FIGURE HERE

Figure 12: ⟨man⟩ result tree coordinated with ⟨man-lay⟩ source tree.

$\mathcal{G}_1$: Source AGS

$\langle \texttt{sen} \rangle^{(x_0)} \longrightarrow (\langle \texttt{word} \rangle^{(x_1)})^+ \langle \texttt{qt} \rangle^{(x_2)} (\langle \texttt{word} \rangle^{(x_1)})^+$
  $\texttt{q-type}(x_2,\texttt{Y})\texttt{:-user-q-type}(x_2,\texttt{Y})$
$\langle \texttt{qt} \rangle^{(x_0)} \longrightarrow (\langle \texttt{word} \rangle^{(x_1)})^+$
  $\texttt{q-type}(x_1,\texttt{Y}) :- \texttt{q-type}(x_0,\texttt{Y})$

$\mathcal{G}_2$: Result AGS

$\langle \texttt{line} \rangle^{(x_0)} \longrightarrow ([\texttt{box}]^{(x_1)})^+$
  $\texttt{font}(x_1,\texttt{boldital}) :- \texttt{q-type}(x_1,\texttt{direct})$
  $\texttt{font}(x_1,\texttt{ital}) :- \texttt{q-type}(x_1,\texttt{indirect})$

$\mathcal{G}_{\langle \texttt{word} \rangle}$: $\langle$word$\rangle$-Coordination AGS

$\langle \texttt{word} \rangle^{(x_0)} \longrightarrow [\texttt{box}]^{(x_1)}$
  $\texttt{q-type}(x_1,\texttt{Y}) :- \texttt{q-type}(x_0,\texttt{Y})$

Figure 13: Semantic coordination.

See attachment

FIGURE HERE

Figure 14: $\langle$line$\rangle$ result tree coordinated with $\langle$sen$\rangle$ source tree.

```
<!-- beginning of DTD -->
<!DOCTYPE sen
 [<!ELEMENT (sen:X0) -- ((word:X1)+,(qt:X2),(word:X3)+)>
  <!ELEMENT (qt:X0) -- (word:X1)+
      <!RULES [q-type(X1,Y) :- q-type(X0,Y).]>>
  <!ELEMENT (word:X0) -- #PCDATA)
      <!RULES [guard(X0) :- q-type(X0,Y).]>>
  <!ATTLIST qt q-type (direct|indirect) direct>
  <!ATTLIST word q-type ATOM #RULED>]>
<!-- end of DTD -->
<!-- beginning of DI -->
<sen><word>the</word><word>quick</word><word>brown</word><word>fox</word>
     <qt q-type = direct><word>jumps</word><word>over</word></qt>
     <word>the</word><word>lazy</word><word>dog</word></sen>
<!-- end of DI -->
<!-- beginning of RTD -->
<!RESTYPE line
 [<!ELEMENT (line:X0) -- (box:X1)+
      <!RULES
       [font(X1,boldital) :- q-type(X1,direct).
        font(X1,ital) :- q-type(X1,indirect).]>
  <!ELEMENT (box:X0) -- #PCDATA>
  <!ATTLIST box font ATOM #RULED
                q-type ATOM #RULED>]>
<!-- end of RTD -->
<!-- beginning of CD -->
<!COORD
 [<!CTYPE word
   [<!ELEMENT (word:X0) -- (box:X1)
        <!RULES
         [q-type(X1,Y) :- q-type(X0,y).]>
<!-- end of CD -->
<!-- beginning of RI -->
<line><box>the</box><box>quick</box><box>brown</box><box>fox</box>
      <box font = boldital>jumps</box><box font = boldital>over</box>
      <box>the</box><box>lazy</box><box>dog</box>
<!-- end of RI -->
```

Figure 15: The "XSGML" specification of quotations and their presentation.

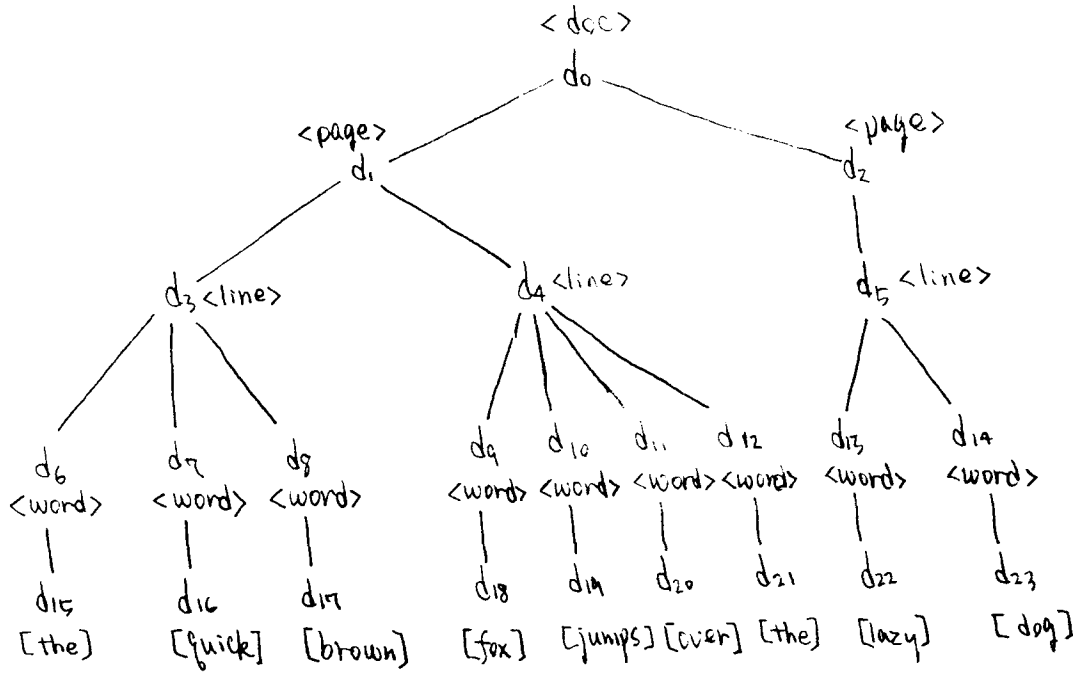

<doc>
d0
<page>
d1
<page>
d2
d3 <line>
d4 <line>
d5 <line>
d6 <word>
d7 <word>
d8 <word>
d9 <word>
d10 <word>
d11 <word>
d12 <word>
d13 <word>
d14 <word>
d15 [the]
d16 [quick]
d17 [brown]
d18 [fox]
d19 [jumps]
d20 [over]
d21 [the]
d22 [lazy]
d23 [dog]

Fig 4.

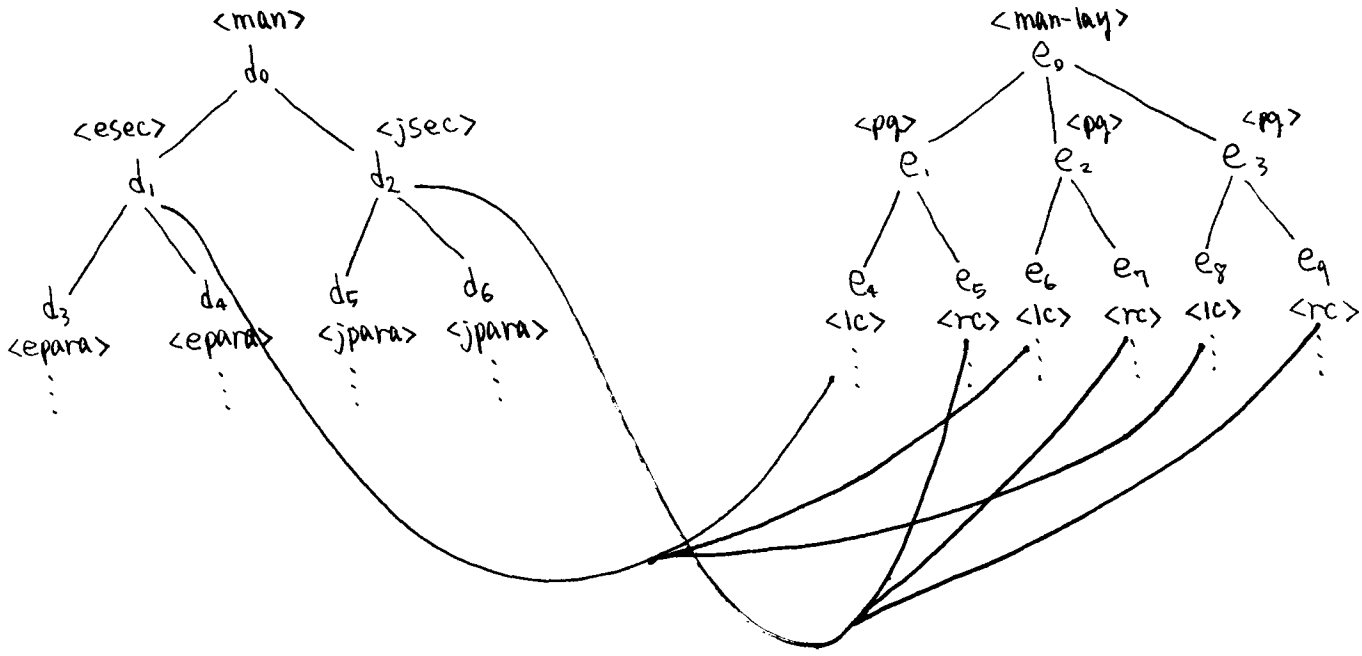
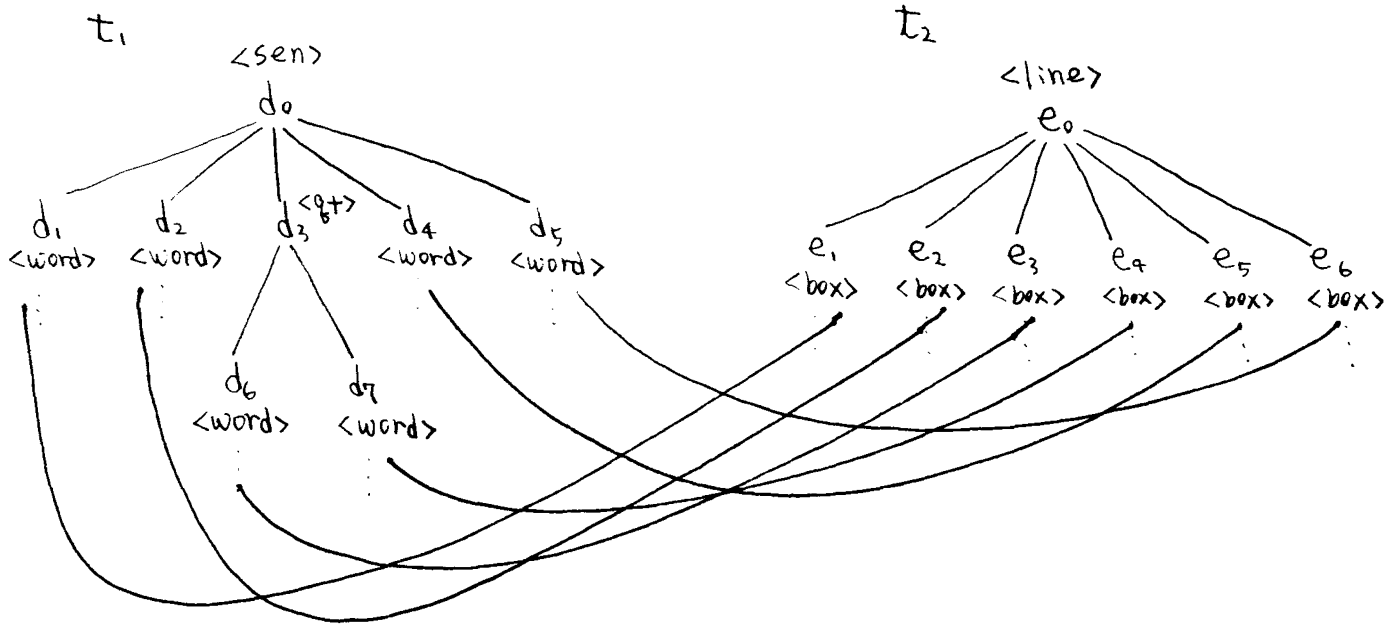

<man>
d0
<esec>
d1
<jsec>
d2
d3 <epara>
d4 <epara>
d5 <jpara>
d6 <jpara>
<man-lay>
e0
<pg>
e1
<pg>
e2
<pg>
e3
e4 <lc>
e5 <rc>
e6 <lc>
e7 <rc>
e8 <lc>
e9 <rc>

Fig. 12



π_{t_1}

```
g-type(d3, Y) :- user-g-type(d3, Y)

g-type(d6, Y) :- g-type(d3, Y)
g-type(d7, Y) :- g-type(d3, Y)
```

π_{t_2}

```
font(e1, boldital) :- g-type(e1, direct)
font(e1, ital) :- g-type(e1, indirect)

font(e2, boldital) :- g-type(e2, direct)
font(e2, ital) :- g-type(e2, indirect)

        :

font(e6, boldital) :- g-type(e6, direct)
font(e6, ital) :- g-type(e6, indirect)
```

Coordination DB

```
g-type(e3, Y) :- g-type(d6, Y)
g-type(e4, Y) :- g-type(d7, Y)
```

User DB (example)

```
user-g-type(d3, direct).
```

Fig. 14