Syracuse University

SURFACE

Electrical Engineering and Computer Science - Technical Reports

College of Engineering and Computer Science

5-1990

A Logic Grammar Foundation for Document Representation and Document Layout

Allen Brown Jr.

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

Follow this and additional works at: https://surface.syr.edu/eecs_techreports

 Part of the Computer Sciences Commons

Recommended Citation

Brown, Allen Jr. and Blair, Howard A., "A Logic Grammar Foundation for Document Representation and Document Layout" (1990). *Electrical Engineering and Computer Science - Technical Reports*. 71.
https://surface.syr.edu/eecs_techreports/71

This Report is brought to you for free and open access by the College of Engineering and Computer Science at SURFACE. It has been accepted for inclusion in Electrical Engineering and Computer Science - Technical Reports by an authorized administrator of SURFACE. For more information, please contact surface@syr.edu.

SU-CIS-90-09

A Logic Grammar Foundation for Document Representation and Document Layout

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

May 1990

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

(315) 443-2368

A Logic Grammar Foundation for Document Representation and Document Layout

Allen L. Brown, Jr.*† and Howard A. Blair‡§

Abstract

We present a powerful grammar-based paradigm for electronic document markup: coordinated definite clause translation grammars. This markup is of a declarative character, being, in effect, a collection of constraints on the logical and physical structure of documents. To the best of our knowledge, coordinated grammars and their parsers can accommodate all of the descriptive and layout processing functionality enjoyed by extant electronic markup languages. We describe an operational prototype that demonstrates the feasibility of a syntax-directed basis for formalizing and realizing document layout.

1 Introduction

Our aim is to formulate an electronic markup language with an unambiguous formal semantics within which one can specify documents in a declarative fashion. We contrast our goal with the reality of popular electronic markup languages such as TEX, LATEX, Scribe, SGML and ODA. While the casual user's view of some of these markups (*e.g.* LATEX and Scribe) would *appear* to be declarative[1], the actual meanings of user issued directives are to be understood through underlying imperative languages. This is evident when a user needs to comprehend the "style" defining mechanisms of these markups.

The technical point of view that we have adopted regarding document representation and document processing is aggressively syntax-oriented. While our methods are related to both syntax-directed translation of programming languages and to syntax-directed natural language processing, our approach is novel in that it uses multiple grammars for the same document. Specifically, these grammars separately represent the logical and layout views of the document represented. The layout grammar is said to be *coordinated* with the logical grammar, and the two are allowed to interact through a narrowly defined interface.

In pursuit of our goal we have embarked upon a three-phased research program. In the recently concluded first phase we have formalized a representative fragment of the Office Document Architecture (ODA) by faithfully translating the ODA description of the specimen document of the published standard into coordinated attribute grammars. The particular class of attribute grammars that we employ is a variant of definite clause translation grammars. With respect to this translation, ODA layout processing has been formalized by giving a declarative (Prolog) description of the parsing and attribute evaluation processes. The main conclusion of the first phase of research is that we can achieve an electronic markup language of ODA-like descriptive power with a precise semantical characterization.

In the remainder of this essay we shall demonstrate that the syntax-directed methods we have developed in our first phase provide a natural and powerful framework for document representation and document processing. Our main vehicle for arriving at this conclusion is the embedding of ODA-like document representation/processing capabilities in a logic grammar framework. Presuming the reader to have some acquaintance with Prolog, we sketch a direct embedding of ODA structures and processing in that language. We introduce particular logic grammars: definite clause grammars and definite clause translation grammars (DCTG's). We illustrate our own variant of the latter by reducing to a DCTG a fragment of the above mentioned ODA specimen document. We then provide a comprehensive exposition of DCTG's and parsing applied to document representation and document processing by considering the detailed specification of the layout process (realized in our operational prototype) for a simple ODA-like document. We show how to pass

*Xerox Corporation, Webster Research Center, 800 Phillips Road, Webster, New York 14580

†abrown.wbst@xerox.com

‡Syracuse University, School of Computer and Information Science, Syracuse, New York 13244-4100

§blair@logiclab.cis.syr.edu

[1] In complete fairness to the designers of SGML, we should point out that it is clearly their *intent* to be declarative. The problem that they leave unresolved is the formal interpretation of their declarations.

from the definition of the document layout process based on *total* parsing that is declarative but impractically inefficient to one based on *partial* parsing that is equally declarative and potentially quite efficient. We briefly discuss the adaptations of efficient context free parsing and incremental attribute evaluation techniques that we plan for our second phase of research. We close by sketching the future phases of research that follow from our operational prototype and its associated semantical framework.

2 Overview of ODA

ODA [11] expresses a syntactically well-defined collection of *document constituents* of which the principal sorts are content portions (graphic characters, raster graphic elements and geometric graphic elements), logical objects and logical object classes, layout objects and layout object classes, and attributes. A logical (layout) structure is a tree-like arrangement of logical (layout) objects and object classes, with the trees' being "foliated" with content portion constituents.

The logical structure of an ODA document is a partitioning of the document's content based on meaning. In that context, logical object classes are elements of generic logical structure from which a set of logical objects with common characteristics may be derived (*e.g.* composite logical objects representing sections), while logical objects are elements of a document having specific interpretations (*e.g.* particular chapters, sections and paragraphs).

The layout structure of an ODA document is a partitioning of the document's content based on presentation. In that context, layout object classes are elements of a generic layout structure from which a set of layout objects with common characteristics may be derived (*e.g.* pages with common headers and footers), while layout objects are elements of a specific layout structure of a document having specific geometric properties (*e.g.* particular pages and blocks). An attribute is an element of a document constituent that has a name and a value, and that expresses a characteristic of that constituent or relationship with one or more other constituents (*e.g.* the "presentation style" attribute establishes the relationship between a basic component description and a presentation style). Document constituent attributes can be viewed as decorating the ODA structure trees in much the same way as semantical attributes decorate parse trees in the attribute grammar paradigm.

The ODA language allows the composition of the above-mentioned constituents into *document descriptions*, each of the latter being composed of a *document profile* and a *document body*. A document profile is a collection of predefined (and preinterpreted) attributes that apply globally to the document description. The document body consists of a generic logical structure, a generic layout structure, specific logical structure, specific layout structure, and style constituents. The last are predetermined (and preinterpreted) collections of attributes that explicitly and implicitly link logical constituents with layout constituents.

3 ODA documents as Prolog

We initially attempted to formalize ODA by a direct Prolog translation of ODA document constructs. We translated particular ODA document descriptions into particular Prolog fragments. Certain ODA constituents correspond to particular Prolog-defined predicates.[2] The real utility of ODA, however, comes only through the descriptive interpretations of various attributes (*e.g.* presentation style) and processes (*e.g.* document layout). The interpretation of these attributes is the main task of *document processing* as exemplified by the layout process. These interpretations are given in [11] in an informal fashion. The main task of formalizing ODA is to define these interpretations *rigorously*. They will turn out to be other Prolog fragments relative to which we define *each* (and every) Prolog translation of an ODA document description. Hereafter, we shall refer to the translation of an ODA structural document description into a logic program as the *data* description and to the Prolog interpretation of attributes (in the document processing context) as the *process* description. The latter rendering can be thought of as defining *interpreters* for various document processors.

The recipe for generating the data description is as follows: Generic (logical and layout) objects are represented as unary predicates. We may think of these generic objects as *types* whose *tokens* are specific (logical and layout) objects. In particular, tokens are individual terms in the Prolog language. Generic attributes, *i.e.* attributes of generic logical or layout objects, are represented as binary predicates. For example, the fact that the generic letter has a presentation style attribute with value '`letter_layout`' is represented by

```
presentation_style(X,Y) :- letter(X), letter_layout(Y).
```

which says that the `presentation_style` of the generic `letter` is the generic `letter_layout`. That a specific letter object `$Letter` had the specific presentation style `$Letter_Layout` would result from asserting the fact

```
presentation_style($Letter,$Letter_Layout).
```

We illustrate the use of the above recipe by a partial translation of the specimen letter whose ODA doc-

[2] In our various ODA translation efforts we have ignored the document profile and all of the details encoded in ODA's document content representations. To elucidate the core document layout process it suffices to view content portions of a document as atomic constructs with certain externally apparent attributes.

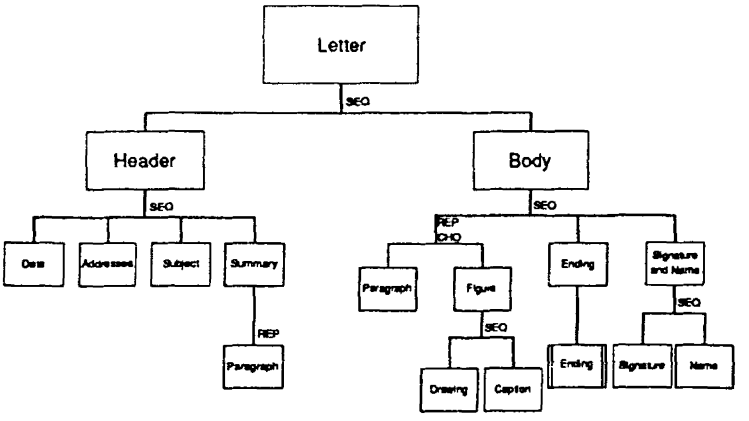

Figure 1: Specimen letter generic logical structure.

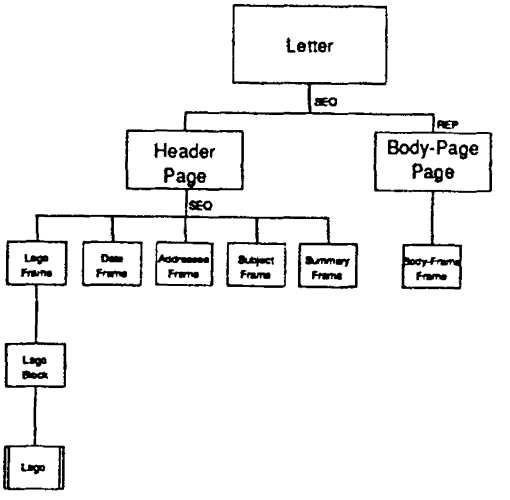

Figure 2: Specimen letter generic layout structure.

ument description is presented in Annex B of Part 2 of [11]. Specifically, our illustrations are drawn from sections B.5 and B.6 (respectively the "processable form document with generic logical structure [figure 1], and generic layout structure [figure 2]" and the "specific logical structure [figure 3] and specific layout structure [figure 4]"). We translate structural aspects of the first three entries of table B.4 (presenting the hierarchical object class descriptions of the specimen letter) of [11] as:

```
letter(X) :-
  sequence(X,[U,V]),
  header(U),
  body(V).
header(X) :-
  sequence(X,[T,U,V,W]),
  date(T),
  addressee(U),
  subject(V),
  summary(W).
summary([]).
summary([X|Y]) :-
  summary_paragraph(X),
  summary(Y).
```

The generic attributes associated with the logical object classes would be given by:

```
object_class(X,document_root) :- letter(X).
user_visible_name(X,"Letter") :- letter(X).
object_class(X,composite) :- header(X).
user_visible_name(X,"Header") :- header(X).
object_class(X,basic) :- date(X).
user_visible_name(X,"Date") :- date(X).
layout_style(X,Y) :- date(X),date_layout(Y).
offset(X,[trailing(710),right_hand(395)]) :- date(X).
content_architecture_class(X,processable_characters) :-
  date(X).
object_class(X,basic) :- addressee(X).
user_visible_name(X,"Addressee") :- addressee(X).
layout_style(X,Y) :- addressee(X),addressee_layout(Y).
content_architecture_class(X,processable_characters) :-
  addressee(X).
object_class(X,basic) :- subject(X).
user_visible_name(X,"Subject") :- subject(X).
```

```
layout_style(X,Y) :- subject(X),subject_layout(Y).
presentation_style(X,Y) :-
  subject(X),
  subject_presentation(Y).
line_spacing(X,300) :- subject(X).
content_architecture_class(X,processable_characters) :-
  subject(X).
object_class(X,composite) :- summary(X).
layout_style(X,Y) :- summary(X),summary_layout(Y).
user_visible_name(X,"Summary") :- summary(X).
object_class(X,basic) :- summary_paragraph(X).
user_visible_name(X,"Summary paragraph") :-
  summary_paragraph(X).
layout_style(X,Y) :-
  summary_paragraph(X),
  summary_paragraph_layout(Y).
offset(X,[left_hand(705)]) :- summary_paragraph(X).
alignment(X,justified) :- summary_paragraph(X).
presentation_style(X,Y) :-
  summary_paragraph(X),
  summary_paragraph_presentation(Y).
content_architecture_class(X,processable_characters) :-
  summary_paragraph(X).
```

The translation of a specific logical document description has the same object-oriented structure as does the original ODA. That is, each object represented in Table B.6 of [11] has a corresponding term, realized as a so-called *gensym* (constants beginning with $). These gensyms are created as a side effect of a Prolog query, whose details need not concern us here. The query results in the creation of the gensyms $Letter_1, $Header_2, $Body_3, $Body_paragraph_4, $Body_paragraph_5, *etc.*, as well as the assertion of ancillary facts about those gensyms, such as

```
object_type($Letter_1,letter).
subordinate($Header_2,$Letter_1).
object_type($Header_2,header).
subordinate($Body_3,body,$Header_2).
object_type($Body_3,body).
subordinate($Body_paragraph_4,$Body_3).
```

```
object_type(#Body_paragraph_4,body_paragraph).
content_portion(#Body_paragraph_4,#A).
object_type(#A,graphic_characters).
subordinate(#Body_paragraph_5,#Body_3).
object_type(#Body_paragraph_5,body_paragraph).
content_portion(#Body_paragraph_5,#B).
object_type(#B,graphic_characters).
```

effectively translating the descriptive content of Table B.6. The translations of the specimen letter's generic layout structure, generic attributes of the layout object classes, and the specific document layout structure parallels the translations of the analogous logical entities.

For the most part the above leaves the generic attributes entirely undefined. For example, what does it *mean* for the logical class `summary_paragraph` to have an `alignment` attribute of `justified` (*i.e.* `alignment(X,justified) :- summary_paragraph(X).`)? The real definition for this attribute lies in its intended interpreter, in this case the *layout process*. The ODA layout process includes the *document layout process* and the *content layout process*. These processes are concerned with the creation of a specific layout structures which can be used by the *imaging process* to present the ODA-specified document in human perceptible form in a presentation medium. The document layout process creates a specific layout structure in accordance with the generic layout structure and information derived from the specific logical structure, the generic logical structure and layout styles (if present).

The gist of ODA's layout model is as follows: Each ODA content portion is mapped on to one or more (layout) blocks (having geometric extents constrained by ODA layout attributes such as `alignment` having a value of `justified`) where the blocks may be generated "on the fly". The situation of multiple mapping arises when content layout permits a content object to be split since the (yet to be mapped) content cannot be fit into the space remaining in the containing frame. The content portions are totally ordered and it is in that order that blocks for content objects are generated. The order derives from the depth-first (pre-)ordering of the tree implicit in the specific logical description (figure 3). To capture the layout process we defined the Prolog predicate `layout_process(X,Y,U,V)` where `V` is the specific layout structure (produced by side-effect) resulting from laying out the specific logical structure `U` (an instance of the generic logical structure whose user visible name is `X`) according to the generic layout structure whose user visible name is `Y`. In particular the query `?- layout_process(letter,letter_layout,#Letter_1,V)` will result in binding `V` to `#Letter_layout_5`, as well as the creation of subordinate specific objects and the asserting of the facts about them in the fashion we sketched above. The `layout_process` predicate can be (and has been) fully Prolog-defined. The definition is fundamentally flawed in that layout occurs mainly by side-effect, and its definition mimics the traditional procedural style of document layout. In the sections that follow we address these flaws by substantially raising the level of abstraction of the logic programming account of ODA data descriptions. We thereby enable an declarative account of layout process description.

4 Logic grammars

Definite clause grammars are a version of context-free grammars [7] that have particularly straightforward translations into definite clauses (Prolog facts and rules) yielding parsers for those grammars. The following definite clause translation grammar[3] characterizes a fragment of English:

```
sentence ::= noun_phrase,verb_phrase.
noun_phrase ::= determiner,noun_phrase2.
noun_phrase ::= noun_phrase2.
noun_phrase2 ::= adjective,noun_phrase2.
noun_phrase2 ::= noun.
verb_phrase ::= verb.
verb_phrase ::= verb,noun_phrase.
determiner ::= [the].
determiner ::= [a].
adjective ::= [decorated].
noun ::= [pieplate].
noun ::= [surprise].
verb ::= [contains].
```

The expressions above (whose principal functor is `::=`) are productions of a (context-free) grammar wherein the (Prolog) constants bracketed by `[` and `]` are terminals, while the remaining constants are nonterminals. The first production states that a `sentence` is a `noun_phrase` followed by a `verb_phrase`. The last production states that a a verb consists of the word `contains`. The translation recipe from productions of a grammar to Prolog rules and facts (definite clauses) is roughly as follows: For each production

1. Replace `::=` with `:-` in any production free of terminals to its right;
2. Append to each nonterminal appearing in a production having no terminal on its right-hand-side the string `(Xi)` where `Xi` is any Prolog variable not previously used in the translation process;
3. If `X0,X1,...`, and `Xn` are the variables appearing (in their order of introduction) in a transformed production, append to the right-hand-side of the production (after the last transformed nonterminal) the string

   `append(X1,X2,L3), append(L3,X3,L4),...,`
   `append(Ln,Xn,X0)`; and

[3] This grammar must be augmented with a definition for the usual append predicate where `append(X,Y,Z)` holds just in case the list `Z` is the list `Y` appended to the list `X`.

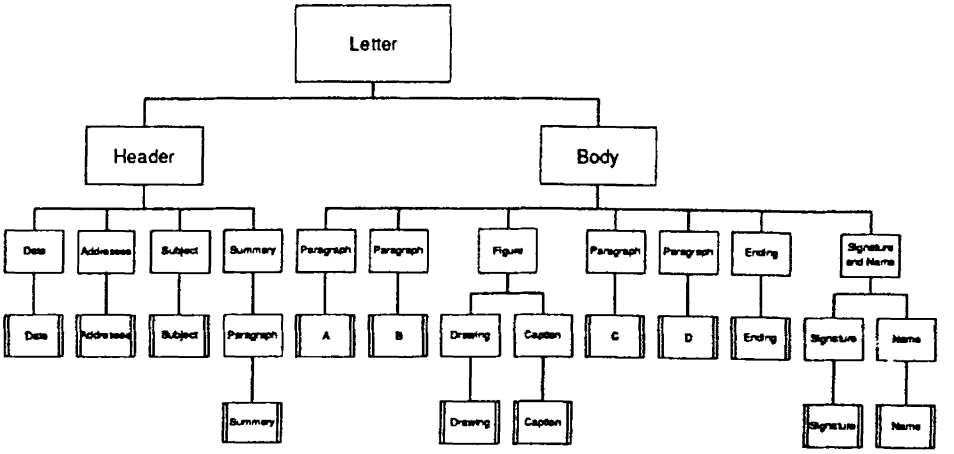

Figure 3: Specimen letter specific logical structure.

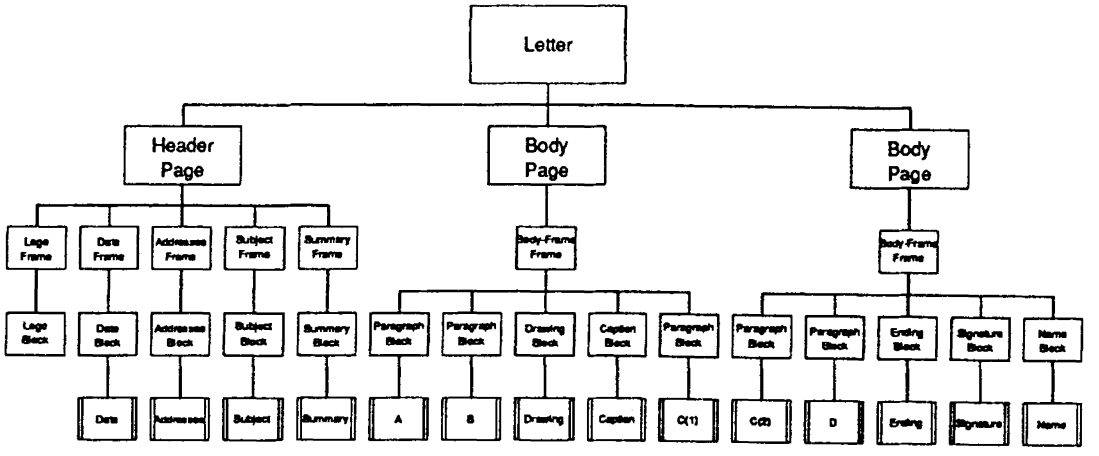

Figure 4: Specimen letter specific layout structure.

4. From any production having a terminal on its right-hand side, delete the `::=` symbol and append the string `(t)` to the nonterminal appearing on the left-hand-side of the production where t is the terminal expression appearing on the right-hand-side of the production.

A query (against the Prolog translation) of the form `?- sentence(S).` will generate all of the legal English sentences according to the given grammar. The query `?- sentence([the, decorated, pieplate, contains, a, surprise]).` will return `yes`, and `?- sentence([pieplates, contain, surprises]).` will return `no`.

There are many features of languages that are either inconvenient or impossible to capture by context-free grammars. Subject-verb agreement in English is such a feature. We extend our grammatical notation as illustrated above by the rewritten productions below to capture subject-verb agreement. Augmenting the grammar above with auxiliary (Prolog) variables can support context sensitivity. This methodology becomes unmanageable because it does not of itself encourage any particular structuring discipline. To address the problem, investigators of logic-based parsing formalisms [1, 10] have borrowed liberally from researchers in attribute grammars [3]. One result of this confluence of interests is the definite clause translation grammar. This particular logic grammar provides a logic programming setting with both the context-sensitive expressive power and structuring discipline of attribute grammars. In our particular version of DCTG's (a variant of that described in [1]) we present the following context-sensitive grammar to handle noun-verb agreement:

```
sentence ::=
    meta(seq([noun_phrase^^T1,verb_phrase^^T2])) <:>
  num(Num) ::- T1^^num(Num),T2^^num(Num).
noun_phrase ::=
    meta(seq([determiner^^T1,noun_phrase2^^T2])) <:>
```

```
  num(Num) ::- T1^^num(Num),T2^^num(Num).
noun_phrase ::= meta(seq([noun_phrase2^^T1])) <:>
  num(Num) ::- T1^^num(Num).
noun_phrase2 ::= meta(seq([adjective,noun_phrase2^^T2])) <:>
  num(Num) ::- T2^^num(Num).
noun_phrase2 ::= meta(seq([noun^^T1])) <:>
  num(Num) ::- T1^^num(Num).
verb_phrase ::= meta(seq([verb^^T1])) <:>
  num(Num) ::- T1^^num(Num).
verb_phrase ::= meta(seq([verb^^T1,noun_phrase^^T1])) <:>
  num(Num) ::- T1^^num(Num).
determiner ::= [the] <:>
  num(sing).
determiner ::= [the] <:>
  num(plur).
determiner ::= [a] <:>
  num(sing).
determiner ::= [some] <:>
  num(plur).
adjective ::= [decorated].
noun ::= [pieplate] <:>
  num(sing).
noun ::= [pieplates] <:>
  num(plur).
noun ::= [surprise] <:>
  num(sing).
noun ::= [surprises] <:>
  num(plur).
verb ::= [contains] <:>
  num(sing).
verb ::= [contain] <:>
  num(plur).
```

To understand the DCTG, consider the first rule of the grammar. This rule has two parts separated by the token <:>. The first part is a syntactic constraint indicating (just as before) that a `sentence` is composed of[4] a `noun_phrase` followed by a `verb_phrase`. Two Prolog variables, `T1` and `T2`, are introduced. In the course of parsing these will be bound respectively to the parse tree generated for `noun_phrase` and that generated for `verb_phrase`. The second part of the rule is zero or more (one in this case) *semantic* constraints. These semantic constraints govern the values that can be taken on by attributes associated with parse trees (and therefore with nonterminals). The parse trees associated with each of `sentence`, `noun_phrase` and `verb_phrase` have `num` attributes and the value of that attribute for a parse tree generated from `sentence` is constrained to be the same as the values of that attribute for the parse trees generated from `noun_phrase` and `verb_phrase`. The nonterminals of the grammar such as `determiner` that rewrite to terminals such as `[the]` have the values of their `num` attributes fixed at particular constants (either `sing` or `plur`). The translation of the DCTG yields `yes` on queries such as `?- sentence([some, pieplates, contain, a, surprise]).` and `no` on `?- sentence([some, pieplates, contains, a, surprise]).`.

An ODA document can be embedded in the DCTG formalism in the following way: Generic logical and generic layout structures will each be encoded as grammars. Nonterminals will correspond to generic objects and terminals will correspond to individual content portions. Attributes in the ODA sense will be directly mapped into attributes in the DCTG sense. Specific logical and layout structures are simply the parse trees generated by their respective grammars.

The layout structure grammar is *coordinated* with the logical structure grammar. Roughly speaking, this means that any "string" of content portions generated by the logical structure grammar is also generated by the layout structure grammar, and that certain subtrees of the parse tree of the logical structure grammar will correspond to subtrees of the parse tree of the layout structure grammar. The parse trees with respect to the two grammars for that string are distinct. The logical structure grammar for the fragment of the ODA specimen document's generic logical structure (with some of its attributes defined) that we presented in section 3 is as follows[5]:

```
letter ::= meta(seq([header,body])) <:>
  object_class(document_root),
  user_visible_name("Letter").
header ::= meta(seq([date,addresse,subject,summary])) <:>
  object_class(composite),
  user_visible_name("Header").
summary ::= meta(rep(summary_paragraph)) <:>
  object_class(composite),
  user_visible_name("Summary").
```

No parse tree variables appear in this DCTG because none of the nonterminals of this fragment has attributes dependent upon the attributes of other nonterminals appearing in the same production.

5 Representing and laying out a simple ODA-like document

A full recounting of our logic grammar/parsing treatment of the data/process descriptions of the ODA specimen letter would be inappropriately complex for a report of this length. Instead, we shall illustrate the essential details of our approach by appealing to an ODA representable document of considerably simpler structure. The generic logical structure of our simple document will consist of arbitrarily long sequences of para-

[4] The idea of describing a document as a grammar mimicing ODA structure diagrams such as figure 1 and treating document layout as attribute evaluation is not unique to ourselves (*e.g.* [4]). One novelty in our approach is to treat logical and layout structure as *distinct* but coupled (through their attributes) grammars.

[5] Simple concatenation of phrase structures is indicated in our DCTG's by use of the metasyntactic constructor `seq`, such constructors being introduced by the indicator `meta`. We also make use of the metasyntactic constructor `rep` indicating arbitrary repetition of the phrase structure(s) in its scope. Readers familiar with ODA should note the analogy with the ODA content generator operators `SEQ` and `REP`.

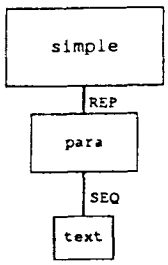

Figure 5: Simple generic logical structure.

graphs. (We shall take a paragraph to be a simple text portion.) Similarly, the generic layout structure for our simple document consists of arbitrarily long sequences of plates (pages), which in turn are arbitrarily long sequences of paragraph blocks. Below we present DCTG's `simple`[6] and `simple_layout` that correspond to the generic logical and layout document structures illustrated in (respectively) figures 5 and 6. We begin by declaring `simple_layout` and `para_block` to be the styles corresponding respectively to `simple` and `para`:

```
styles(simple_layout,simple).

styles(para_block,para).
```

Were `simple` and `para` ODA logical objects, these declarations would correspond to asserting that the values of the "`layout style`" attributes of these two objects respectively have as values the generic layout objects `simple_layout` and `para_block`.

Below is the DCTG representing the generic logical structure of the `simple` document:

```
simple^^T0 ::= meta(rep(para^^T1)),
    {T0^^content_interval(U,V)} <:>
  logical_type(root),
  (style(X) ::- styles(X,simple)),
  (countent(Z) ::- sum_countent_from(T1,Z)),
  (content_interval(M,N) ::-
     M is 1,
     sum_countent_from(T1,N)).
para ::= meta(seq([text^^T1])) <:>
  (style(X) ::- styles(X,para)),
  (countent(Z) ::- T1^^countent(Z)),
  (content_interval(M,N) ::-
     number(M),
     T1^^countent(U), N is (M + (U - 1))).
text ::= ["TEXT1"] <:>
  (countent(Z) ::- Z is 1),
  (content_interval(N,N) ::- number(N)).
text ::= ["TEXT2"] <:>
  (countent(Z) ::- Z is 1),
  (content_interval(N,N) ::- number(N)),
  layout_directive_req(apart).
text ::= ["TEXT3"] <:>
  (countent(Z) ::- Z is 1),
  (content_interval(N,N) ::- number(N)).
```

[6] We identify a grammar with its root nonterminal. Hence we speak of the `simple` DCTG.

The DCTG has root nonterminal `simple`, intermediate nonterminals `para` and `text`, and terminals "`TEXT1`", "`TEXT2`", and "`TEXT3`" (which expressions are Prolog text terms). The syntactic part of the `simple` DCTG production asserts that a `simple` is any nonempty finite sequence of `para`'s. (A formally more accurate view of the syntactic part of the first rule is that it is an abbreviation for the infinite collection of rules

```
simple ::= meta(seq([para^^T1])).
simple ::= meta(seq([para^^T1,para^^T2])).
simple ::= meta(seq([para^^T1,para^^T2,para^^T3])).
          ⋮     )
```

Similarly, `para` is syntactically specified to be a `text` and `text` is syntactically specified to be one of the three terminals, "`TEXT1`", "`TEXT2`", or "`TEXT3`".

In addition to the syntactic characterization of nonterminals provided by productions, there is the semantic characterization provided by *guards* and attributes. The guard of the `simple` production (the expression embraced by {}) guarantees that the production can be used successfully only if the `countent_interval` attribute can be given a value consisting of the ordered pair whose first and second components are the values of `U` and `V` respectively (*i.e.* the indices of the content portions spanned by `simple`). `simple` has attributes `logical_type`, `style`, `countent`, and `content_interval`. The first attribute asserts that `simple` names a logical structure grammar (*i.e.* a "root object" in the parlance of ODA). The second says that the `style` of `simple` is `X` if `X styles simple`, *i.e.* `X = simple_layout`. The `countent` attribute asserts that the number of content objects spanned by `simple` is the sum of the numbers of content objects spanned by each of the immediate descendants (*i.e.* the `para`'s) of `simple`. The `content_interval` attribute indicates that the interval of indices of content portions spanned by `simple` includes the first through last content portions.

The `style` attribute of `para` is `X` if `X styles para`, *i.e.* `X = para_block`. The `countent` attribute of `para` is the same as the `countent` attribute of the object (*i.e.* `text`) that is the immediate descendant of `para`. The `content_interval` attribute of `para` is the same as the `content_interval` attribute of the object (*i.e.* `text`) that is the immediate descendant of `para`.

Finally, the second occurrence of `text` has a layout directive request of `apart`. That is, the content associated with no previous logical object will be placed on the same layout object (a `plate` in this case) as the content associated with that occurrence of `text`. This attribution corresponds to the ODA layout directive "`new layout object`". This particular content object is to be placed in a layout object distinct from that which receives the previous (in the layout order) content object. The `countent` attributes of all the occurrences of `text` have values of `1`. The `content_interval` attributes of all

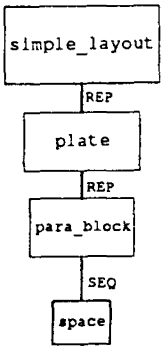

Figure 6: Simple generic layout structure.

the occurrences of text have values that are intervals of length 1 beginning at an index 1 greater than the upper bound of the previously indexed object (in the layout order).

Below is the DCTG representing the generic layout structure of the simple document:

```
simple_layout^^T0 ::= meta(rep(plate^^T1)),
    {T0^^content_interval(1,V)} <:>
  layout_type(root),
  (page_count(PC) ::- sum_page_count_from(T1,PC)),
  (out_trees(TTI,TT0) ::-
     styles(simple_layout,X),
     find1_node(node(X,STT,Sem),TTI,TT01,true),
     propagate_trees(T1,TT01,TT0)),
  (content(Z) ::- sum_content_from(T1,Z)),
  (content_interval(M,N) ::-
     number(M),
     sum_content_from(T1,U),
     N is (M + (U - 1))).
plate^^T0 ::=
    meta(rep(para_block^^T1)),
    {T0^^set_depth(X), sum_set_depth_from(T1,S), X >= S} <:>
  layout_type(page),
  (page_count(PC) ::- PC is 1),
  (set_depth(X) ::- X is 1.0),
  (out_trees(TTI,TT0) ::- propagate_trees(T1,TTI,TT0)),
  (content(Z) ::- sum_content_from(T1,Z)),
  (content_interval(M,N) ::-
     number(M),
     sum_content_from(T1,U),
     N is (M + (U - 1))),
  layout_directive_ack(apart,text).
para_block ::= meta(seq([space^^T1])) <:>
  layout_type(block),
  (out_trees(TTI,TT0) ::-
     styles(para_block,Y),
     find1_node(node(Y,STT,Sem),TTI,[T|TT0],true)),
  (content(Z) ::- T1^^content(Z)),
  (content_interval(M,N) ::-
     T1^^content(U), N is (M + (U - 1))),
  (set_depth(X) ::- T1^^set_depth(X)).
space ::= ["TEXT1"] <:>
  (set_depth(X) ::- X is 0.5),
  (content(Z) ::- Z is 1),
  (content_interval(N,N) ::- number(N)).
space ::= ["TEXT2"] <:>
  (set_depth(X) ::- X is 0.5),
  (content(Z) ::- Z is 1),
  (content_interval(N,N) ::- number(N)).
space ::= ["TEXT3"] <:>
  (set_depth(X) ::- X is 0.5),
  (content(Z) ::- Z is 1),
  (content_interval(N,N) ::- number(N)).
```

The guard of the simple_layout production guarantees that the production can be used successfully only if the content_interval attribute can be given a value consisting of the ordered pair whose first and second components are the values of U and V respectively. simple_layout has intermediate nonterminals plate, para_block, and space. The syntactic part of the simple_layout rule asserts that a simple_layout is any nonempty finite sequence of plate's, that a plate is any nonempty finite sequence of para_block's, that a para_block is a space, and that a space is one of the terminals "TEXT1", "TEXT2" and "TEXT3". Consistent with our earlier remarks that the layout structure grammar is coordinated with the logical structure grammar, the terminals of simple_layout subsume those of simple. Turning now to the semantic attributes of the simple_layout DCTG, we shall explain all but the out_trees attribute. We shall postpone its explanation until our description of the layout process.

simple_layout has a layout_type of root, indicating that simple_layout names a layout structure grammar. The value of the page_count attribute is constrained by its rule to be the reckoning of the number of layout objects spanned by simple_layout having a layout_type attribute with value page. The content attribute asserts that the number of content objects spanned by simple_layout is the sum of the numbers of content objects spanned by each of the immediate descendants (*i.e.* the plates's) of simple_layout. The content_interval attribute indicates the interval of indices of content objects spanned by simple_layout includes the first through last (in layout order) content objects.

plate has a layout_type attribute with value page and a page_count attribute with a fixed value of unity. (Naturally, an object of layout_type of page counts as a single page!) The value of the set_depth attribute of an object indicates an object's vertical extent, and, in the case of a plate, has a value of 1.0. The layout_directive_ack attribute for plate indicates the type and the source of layout_directive_requests to which a plate is willing to respond. In this case plate responds to apart requests from (logical) objects of type text. As the request indicates that requesting content object is to be placed in a layout object distinct from that which received the previous content object, the acknowledgement of the request leads to the creation of a new plate object to receive the requesting content object. The content attribute asserts that the number of content objects spanned by plate is the sum of the numbers of content objects spanned by each of the immediate descendants (*i.e.* the para_block's) of plate. The value of

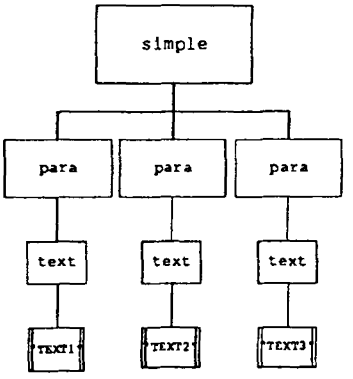

Figure 7: Simple specific logical structure.

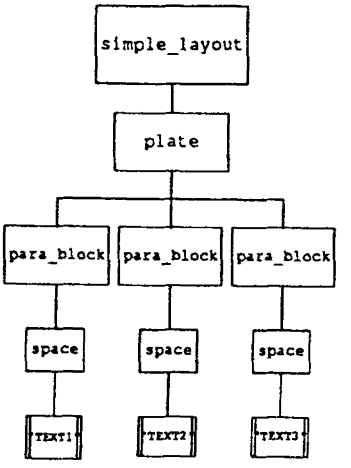

Figure 8: First simple specific layout structure.

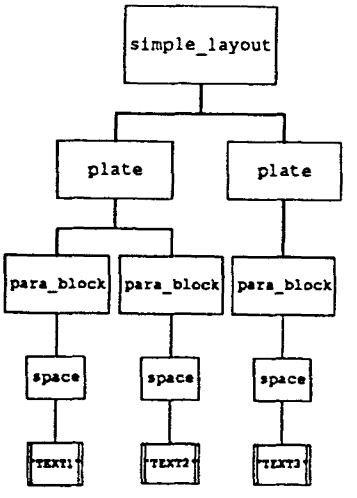

Figure 9: Second simple specific layout structure.

the `content_interval` attribute of `plate` is the pair consisting of the index of the first content object spanned by the left-most (in layout order) of the `plate`'s immediate descendants, and the index of the last content object spanned by the right-most (in layout order) of the `plate`'s immediate descendants. The guard of the `plate` rule admits only those applications of the production in which the sum of the `set_depth`'s of the `para_block`'s is no larger than the `set_depth` of the `plate`.

A `para_block` has a `layout_type` attribute with value `block` and "synthesizes" the value of its `set_depth` attribute from the value of the same attribute of the `space` object below. A `para_block`'s `content` attribute asserts that the number of content objects spanned by the `para_block` is the same as that spanned by its immediate descendant (*i.e.* the `space`). The value of the `content_interval` attribute of `para_block` is the pair consisting of the index of the first content object spanned (left-most in layout order) by the `para_block`'s immediate descendant, and the index of the last content object spanned (right-most in layout order) by the `plate`'s immediate descendant.

All three `space` objects have `set_depth` attributes with values of `0.5`. As indicated by the values of their `content` attributes, each spans precisely one content object. As a consequence, their `content_interval` attributes are unit intervals whose boundaries are the indices (in layout order) of the single content objects spanned.

Considering the input string of content portions, `["TEXT1","TEXT2","TEXT3"]`, the logical structure grammar `simple` (figure 5) yields only one context free parse, that of figure 7. On the same input list of content portions, the layout structure grammar `simple_layout` (figure 6) yields four context free parses, figures 8-11. The main task of the layout process is to "disambiguate" the latter parses by using the context-sensitive information provided primarily by the attributes in the layout structure grammar. A collection of preference criteria is applied to the set of context free parses (of the layout structure grammar on a particular input string of content portions). These criteria induce a preference ordering on the parses. With respect to that ordering the "best" parses are chosen. A parse tree P1 is preferred to a parse tree P2 if

1. P1 satisfies the guards (the literals embraced by {}) on all the productions used in its construction, but P2 does not;
2. P1 and P2 are unordered by the previous criterion, but P1 is coordinated with the parse of the input list according to the logical structure grammar while P2 is not;
3. P1 and P2 are unordered by the previous criteria, but P1 spans fewer page objects than does P2; and

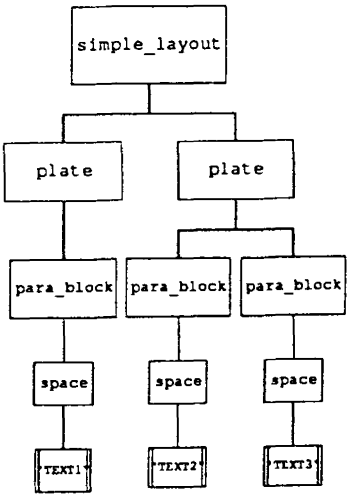

Figure 10: Third simple specific layout structure.

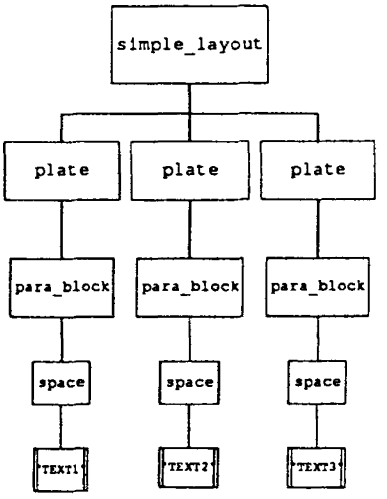

Figure 11: Fourth simple specific layout structure.

4. P1 and P2 are unordered by the previous criteria, but some content portion X appears on an earlier page in P1 than it does in P2, while no content portion before (in the layout ordering) X appears on an earlier page in P2 than it does in P1.

We define a Prolog predicate `layout`

```
layout(CG,FG,L,FTT) :-
  bagof(FT,
        (parse([CG],[CT|CTT],L,[]),
         parse([FG],[FT|FTT1],L,[]),
         FT^^out_trees([CT],0TT),
         reqs_ackd(CG,CT,FG,FT)),
        FTT2),
  min_pages(FTT2,FTT3),
  min_place(L,FTT3,FTT).
```

that guarantees an ordering under these criteria by means of other Prolog defined predicates that we shall describe presently.

`parse([CG],[CT|CTT],L,[])` parses the input list of content portions L (bound to `["TEXT1", "TEXT2","TEXT3"]`) according to the logical structure grammar CG (bound to `simple`), binding CT to the resulting parse tree. Similarly, `parse([FG],[FT|FTT1],L,[])` parses the input list of content portions L according to the layout structure grammar FG (bound to `simple_layout`), binding FT to the resulting parse tree. The success of `FT^^out_trees([CT],0TT)` guarantees that the appropriate stylistic correspondences obtain between elements of the specific logical structure represented by CT and the specific layout structure represented by FT (*i.e.* they are coordinated). Recall that in the discussion above we mentioned the `apart` "request" and "response". The `reqs_ackd` predicate assures that for each request in the logical structure there is indeed a respondent in the layout structure. Successful acknowledgement demands (among other things) the rejection of any context-free parse that does not have the content associated with the requesting `text` object appearing in a `plate` distinct from that receiving the content associated with the previous `text` object. Among the parse trees of figures 8-11 then, only those of figures 10 and 11 are acceptable. Figure 8 is rejected by the guard of the `plate` rule and figure 9 is rejected by `reqs_ackd`. FTT2 is now bound to a list of parse trees (bindings of FT) for which all the foregoing conditions obtained, that is, those of figures 10 and 11. `min_pages(FTT2,FTT3)` succeeds just in case FTT3 is bound to those parse trees FT (on the list FTT2) having the least number of pages, (pages being those subtrees having an attribute `layout_type` with value `page`). In this instance, that means exactly the parse tree of figure 10. Finally, `min_place(L,FTT3,FTT)` guarantees that FTT is bound to those FT's on the list FTT3 such that the content items appear as early as possible in the layout order (when compared with other members of FTT3) among the subtrees having an attribute `layout_type` with value `page`. Again, that means exactly the parse tree of figure 10.

6 Partial parsing

We have shown how the logic grammar representations of documents together with attributed parsing can give a declarative account of document layout. Our approach as described thus far would be hopelessly inefficient as a *practical* basis for document layout. The main problems (in order of increasing gravity) are three:

1. The parsers that we generated from the grammars are the obvious sorts of top-down, recursive descent parsers that naturally arise from context-free grammars. These parsers exhibit exponential worst-case performance. This problem is straightforwardly

remedied by adopting any of the well-known $O(n^3)$ parsers [5] for context-free grammars.

2. Our evaluation of attributes is on an "as-needed" basis. This engenders both the *re*evaluation of attributes and recurring visits to individual nodes of a particular parse tree. Again, the adoption and adaptation of one of the efficient batch-oriented attribute evaluation strategies described in [3] or incremental strategies described in [9] would address this problem.
3. We have posed the layout task as a certain optimization over competing attributed parses of a document's content. The optimization scheme we described is an instance of "generate and test". To recapitulate, we generate all of the candidate context-free parses, evaluate their attributes, and compare the various parses to find the optimal ones. Most of the ultimately rejected candidates *could* have been rejected before carrying their parses or attribute evaluations to completion.

We shall devote the remainder of this section to describing our solution, *partial parsing*, to the last problem above.

Partial parsing depends on the *partial parse tree* abstraction. Before characterizing this abstraction, we need to examine some of the details of *total* parse trees (with respect to a given grammar): A *node* is either a content object (restricted still to text strings), or a 3-ary term with principal (Prolog) functor `node`, whose first subterm is a nonterminal of the given grammar (the *label* of the node), whose second subterm is a list of nodes, and whose third subterm is a list of valued attributes. A node that is simply a content object is said to be *terminal* and self-labeling. A *valued attribute* is a variable-free n-ary term whose principal functor is an attribute. A *parse tree* is a nonempty finite set of nodes such that

1. there is a unique node, designated the *root*;
2. the remaining nodes are partitioned into m disjoint sets of nodes each of which is a (sub)tree (of the original) rooted in one of the m nodes forming the list that comprises the second subterm of the node originally designated as the root; and
3. for each parse tree rooted at a node with label M and having subtrees rooted at nodes with labels $N_1, \ldots, N_m$, there is a production of the grammar whose left-hand-side is M and whose right-hand-side is the concatenation of the symbols $N_1, \ldots, N_m$ (in that order).

A total parse tree is a ground (variable-free) term from the Prolog point of view. A partial parse tree will be a generalization of a total parse tree permitting the occurrence of variables at certain locations. We amend the definitions of node and valued attribute thus: A *node* is either a variable, a content object (restricted still to text strings), or a 3-ary term with principal functor `node`, whose first subterm is a nonterminal of the given grammar, whose second subterm is a *node list*, and whose third subterm is a list of valued attributes. A node list is either an empty list, a variable or a pairing of a node and a node list. A *valued attribute* is an n-ary variable-free term whose principal functor is an attribute. Every total parse tree is a substitution instance (a uniform replacement of variables by other terms) of some partial parse tree. Indeed, a total parse tree *is* a partial parse tree. We can define a preference ordering on the partial parse trees analogous to the one we defined on the total parse trees in such a way that any maximally preferred parse tree among the *total* parse trees also happens to be maximally preferred among the partial parse trees.

We can greatly restrict the parse trees that we need to consider in our optimization problem. This follows from the fact that *layout* is the assignment of content items to pages in a manner consistent with the layout order of the content items. Thus, we are interested in the partial parses that correspond to having filled the first n pages with some initial segment of the input list of content portions. For a particular layout structure grammar and input list of content portions we define the n-page partial parse trees to be those partial parse trees generated by the grammar, each of which has n disjoint subtrees that are variable-free and whose root nodes all number among their valued attributes `layout_type(page)`, and each of which spans an initial segment of the input list of content portions. For *these* partial parse trees we define the following preference ordering: An n-page partial parse tree P1 is preferred to an n-page partial parse tree P2 if

1. P1 satisfies the guards on all the productions used in its construction, but P2 does not;
2. P1 and P2 are unordered by the previous criterion, but P1 is stylistically consistent (coordinated) with the portion of the logical grammar (total) parse tree spanning the same initial segment of the input list as P1 while P2 is not; and
3. P1 and P2 are unordered by the previous criteria, but P1 spans a longer initial segment of the input list than does P2.

These partial parse trees can be generated in a top-down or bottom-up fashion, and in the order of increasing n. Straightforward modification of virtually any context-free grammar parsing algorithm and associated attribute valuation algorithms will provide a framework that will facilitate the early rejection of partial parses of which the eventually rejected total parses are substitution instances.

7 Future directions

We have constructed parsers for coordinated grammars that produce layouts in essentially the fashion described in sections 5 and 6. In order to address the first two efficiency-related problems that we described in section 6, we shall redesign and reimplement our parsers to exploit the classes of efficient context-free parsers and incremental attribute evaluators described in [5] and [9]. Our current parsers are of an interpretive nature. That is, they take as input a coordinated pair of grammars and an input string of content items and produce a coordinated pair of parse trees. It is possible to compile the coordinated pair of grammars so as to generate a Prolog program specific to parsing according to those grammars. We intend to implement such a compilation stategy and thereby make additional gains in efficiency.

We have alluded to the fact that coordinated grammars (and hence document descriptions) have a declarative formal semantics. Such a formal semantics could be had simply by considering the usual minimal model semantics [8] of the definite clauses into which coordinated grammars can be compiled. We have instead chosen to take a more illuminating path: We are exploring a mathematical abstraction called a *markup scheme* [2], a formal framework modeled after program schemes [6]. Markup schemes admit a least fixed point semantics analogous to that of logic programs. Moreover, within this framework it is possible to abstract away the details of various electronic markup languages and compare their expressive power according to the presence or absence of various features. We intend to carry out such a comparison among selected markups, examining their expressive power with respect to both structural (*e.g.* dynamic changes in page style) and functional (*e.g.* the degree of forward reference permitted) features of those representations.

In addition to making analytic use of markup schemes, we shall employ them in a synthetic fashion. The logic grammar formulation of document representation that we have presented is not particularly specialized to the representation of documents. We should like to conceive such a specialization, both for reasons of efficiency of document processing and to enhance the usability of the description language. To that end we hope to formulate a constraint logic programming language whose domain of discourse includes certain tree structures that describe particular documents, and whose constraints govern the admissibility of these tree structures according to structural or functional considerations.

While the first phase of our research has demonstrated that grammars and parsing can give declarative accounts of traditional document representation and document processing, it remains to be demonstrated that this point of view is a practical basis for operational document processing systems. A main thrust of our second phase of research will be to enhance the descriptive power (beyond the bounds we have imposed to simulate strictly ODA-like expressibility) and explore efficient parsers that can make a syntax-directed document layout a practical reality. In the third phase we shall articulate and explore a constraint logic programming language for markup.

8 Conclusions

We have offered here a powerful logic grammar-based paradigm for electronic document markup. This markup is of a declarative character, being, in effect, a collection of constraints on the logical and physical structure of documents. Moreover, this logic grammar representation admits a formal semantics that can be used directly to compare and contrast a variety of extant (and possible) electronic markup languages. To the best of our knowledge, coordinated grammars and their parsers can accommodate all of the descriptive and layout processing functionality enjoyed by extant electronic markup languages. We have demonstrated the possibility of syntax-directed basis for formalizing and realizing document layout. We recognize that substantially more work is needed to make a syntax-directed document layout a practical reality within the coordinated grammar framework. We have embarked upon an effort to achieve that reality.

References

[1] Harvey Abramson and Veronica Dahl. *Logic Grammars.* Springer-Verlag, New York, 1989.

[2] Allen L. Brown, Jr. and Toshiro Wakayama. Assigning meaning to markup. Manuscript to be submitted for publication, 1990.

[3] 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.

[4] Paul Franchi-Zannettacci and Dennis S. Arnon. Context-sensitive semantics as a basis for processing structured documents. In Jacques André and Jean Bézivin, editors, *Proceedings of the Workshop on Object-Oriented Document Manipulation*, pages 135–146, 1989.

[5] Susan L. Graham, Michael A. Harrison, and Walter L. Ruzzo. An improved context-free recognizer. *ACM Trans. on Programming Languages and Systems*, 2(3):415–462, July 1980.

[6] Sheila A. Greibach. *Theory of Program Structures: Schemes, Semantics, Verification.* Springer-Verlag, Berlin, 1975.

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

[8] John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, second edition, 1988.

[9] Thomas W. Reps. *Generating Language-Based Environments.* MIT Press, Cambridge, Massachusetts, 1984.

[10] Stuart M. Shieber. *An Introduction to Unification-Based Approaches to Grammar*, volume 4 of *CSLI Lecture Notes.* Center for the Study of Language and Information, Stanford, California, 1986.

[11] 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), March 1988.