# Implementing the TEI's Feature-Structure Markup by Direct Mapping to the Objects and Attributes of an Object-Oriented Database System

GARY F. SIMONS

One of the more intriguing parts of the Text Encoding Initiative's proposals for text encoding (Sperberg-McQueen and Burnard 1994) has been the system for feature-structure markup. This is because it has the potential, although originally developed for the purpose of encoding linguistic analysis, for encoding virtually any kind of information. The real power of the system lies in an auxiliary file called the Feature System Declaration (FSD) which encodes the conceptual model (Borgida 1985) that underlies a particular system of feature-structure markup. SGML can only validate the syntactic integrity of feature-structure markup and of the FSD. The TEI community has been waiting for an application that can validate the conceptual integrity of feature structures and FSDs in order to evaluate their true significance.

This paper describes one such application, but it is not a special-purpose application. The implementation strategy capitalizes on the correspondence between feature structures (with features) and objects (with attributes) to map feature structures onto the objects of a generalized object-oriented database system named CELLAR. Conceptual integrity is achieved by mapping the FSD onto the conceptual modeling component of the database system. The paper first gives an introduction to feature-structure markup and to the CELLAR system, and then explains and demonstrates how the implementation of TEI feature-structure markup has been achieved. This leads to the conclusion that not only is it possible to use feature-structure markup to interchange data between database systems, but also that the FSD formalism and its markup provides a means for these systems to interchange the specification of a data model.

## 1. An Overview of TEI Feature-Structure Markup

A feature structure is a set of features paired with values. The conventional notation uses a single pair of square brackets to enclose all the feature-value pairs that comprise a feature structure. For example, consider the German

word *kind* 'child', occurring in the sentence *Das Kind war einsam*, 'The child was lonely'. The conventional notation for a possible feature structure analysis of this word is given in Fig. 1. In this example the feature *category* has the value 'noun', the feature *wordForm* has the value 'Kind', the feature *proper* (for, 'Is it a proper noun?') has the value '–' (or minus, for, 'No'), and the feature *agreement* holds an embedded feature structure which groups the features of *gender*, *number*, and *case*. The values for these represent neuter, singular, and nominative, respectively.

Following the TEI guidelines, this feature structure would be encoded in SGML as follows:

```
⟨fs⟩
   ⟨f name=category⟩⟨sym value=noun⟩⟨/f⟩
   ⟨f name=wordForm⟩⟨str⟩Kind⟨/str⟩⟨/f⟩
   ⟨f name=proper⟩⟨minus⟩⟨/f⟩
   ⟨f name=agreement⟩
      ⟨fs⟩⟨f name=gender⟩⟨sym value=neut⟩⟨/f⟩
         ⟨f name=number⟩⟨sym value=sg⟩⟨/f⟩
         ⟨f name=case⟩⟨sym value=nom⟩⟨/f⟩
      ⟨/fs⟩⟨/f⟩
⟨/fs⟩
```

This example illustrates the following characteristics of feature-structure markup:

(1) Feature structures are marked by the ⟨fs⟩ tag.
(2) Individual features are marked by the ⟨f⟩ tag, with the feature name given as the value of the *name* attribute.
(3) The value of a feature is encoded as the content of the ⟨f⟩ tag.
(4) A feature value from a closed set is encoded as the *value* attribute of the empty tag ⟨sym⟩.

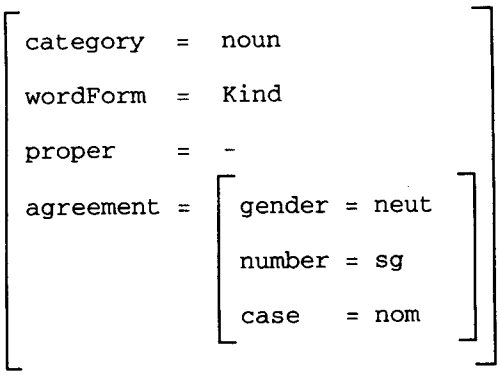


**Fig. 1.** Conventional Feature Structure Notation for a Word Analysis

(5) A feature value which is an arbitrary string is encoded as the content of a ⟨str⟩ tag.
(6) The value of a binary (that is, Boolean) feature is encoded as one of the empty tags ⟨plus⟩ or ⟨minus⟩.

Further details of the TEI system for feature-structure markup are described in Langendoen (1994). The rationale that lies behind the design of this system of markup is given in Langendoen and Simons (forthcoming).

## 2. From Linguistic Markup to General Data Markup

Though feature-structure markup was first developed within the TEI as a means of encoding linguistic analysis of text, in fact, it has a much wider applicability than just linguistics. Feature structures can provide for the encoding of information of nearly any sort. This is because they are just instances of the more general data structure referred to by Donald Knuth as 'nodes' (here called feature structures) and 'fields' (here called features). About such structures, Knuth (1968) writes:

> . . . the ideas we have encountered are not limited to computer programming alone; they apply more generally to everyday life. A collection of nodes containing fields, some of which point to other nodes, appears to be a very good abstract model for structural relations of all kinds; it shows how we can build up complicated structures from simple ones, and we have seen that corresponding algorithms for manipulating the structure can be designed in a natural manner. (p. 462)

A similar sentiment is echoed by Shieber (1986) who is writing specifically about the use of feature structures in representing grammatical formalisms. After explaining how six different formal approaches to grammar can be handled by the single computational model of unification of feature structures, he says:

> In fact, viewed from a computational perspective, it is not surprising that so many paradigms of linguistic description can be encoded directly with generalized feature/value structures of this sort. Similar structures have been put forward by various computer scientists as general mechanisms for knowledge representation (Ait-Kaci 1985) and data types (Cardelli 1984). Thus we have hardly constrained ourselves at all even though limited to this methodology. (p. 10)

For instance, Ide, Le Maitre, and Veronis (1993) have shown that a lexical database can be modeled as a construct of feature structures, which can in turn be implemented as objects in an object-oriented database.

Feature structures with features are essentially equivalent to many familiar schemes of data organization like records with fields, objects with attributes, frames with slots, property lists with properties, and even abstract data types with access functions. Thus, ⟨fs⟩ and ⟨f⟩ could just

as well be used to markup instances of records, objects, frames, property lists, or abstract data types. In fact, the tag names ⟨fs⟩ and ⟨f⟩ were deliberately chosen to allow the alternate reading of 'field structure' and 'field'. For instance, the following might be the encoding of a bibliographic record:

```
⟨fs type=book⟩
   ⟨f name=author⟩⟨str⟩Goldfarb, Charles⟨/str⟩⟨/f⟩
   ⟨f name=year⟩⟨nbr value=1990⟩⟨/f⟩
   ⟨f name=title⟩⟨str⟩The SGML Handbook⟨/str⟩⟨/f⟩
   ⟨f name=publisher⟩⟨str⟩Clarendon Press⟨/str⟩⟨/f⟩
   ⟨f name=pubPlace⟩⟨str⟩Oxford⟨/str⟩⟨/f⟩
⟨/fs⟩
```

This example illustrates two further characteristics of feature-structure markup:

(7) The ⟨fs⟩ element has a *type* attribute which make it possible to distinguish different types of records or objects.
(8) A feature value which is a number is encoded as the *value* attribute of the empty tag ⟨nbr⟩.

Another important component of TEI feature-structure markup is the Feature System Declaration, or FSD (Simons 1994). It is a file that is external to the file containing ⟨fs⟩ markup. It uses SGML markup to express the well-formedness constraints on feature structures. This includes declarations of the types of feature structures that are allowed, the features that are allowed with each type, the types of values that are allowed for each feature, default values for unspecified features, and co-occurrence constraints on feature values. For instance, the above bibliographic record would be sanctioned by a declaration like the following:

```
⟨fsDecl type=book⟩
   ⟨fDecl name=author⟩
      ⟨vRange⟩⟨str rel=ne⟩⟨/str⟩⟨/vRange⟩⟨/fDecl⟩
   ⟨fDecl name=year⟩⟨vRange⟩⟨nbr⟩⟨/vRange⟩⟨/fDecl⟩
   ⟨fDecl name=title⟩
      ⟨vRange⟩⟨str rel=ne⟩⟨/str⟩⟨/vRange⟩⟨/fDecl⟩
   ⟨fDecl name=publisher⟩
      ⟨vRange⟩⟨str rel=ne⟩⟨/str⟩⟨/vRange⟩⟨/fDecl⟩
   ⟨fDecl name=pubPlace⟩
      ⟨vRange⟩⟨str rel=ne⟩⟨/str⟩⟨/vRange⟩⟨/fDecl⟩
⟨/fsDecl⟩
```

⟨fsDecl⟩ gives a 'feature-structure declaration' for the given type of feature structure, while ⟨fDecl⟩ gives the 'feature declaration' for a named feature. ⟨vRange⟩ gives the range of allowed values. This declaration says

that the value of an occurrence of the *year* feature must be an instance of ⟨nbr⟩, while the other features must store nonempty strings. It also says that a feature structure of type *book* is allowed only to have these five features. For any other feature to occur in an instance of ⟨fs type=book⟩ would be a conceptual error.

The remainder of this paper describes how a generalized implementation of the TEI system of feature-structure markup has been achieved by extending an object-oriented database system to use TEI-style ⟨fs⟩ tagging as a possible format for the representation of its objects.

## 3. An Overview of the CELLAR Object-Oriented Database System

The database system used for this implementation is called CELLAR—for Computing Environment for Linguistic, Literary, and Anthropological Research. Developed by the Summer Institute of Linguistics, it is an object-oriented database system for storing multilingual textual information. A full discussion of the user requirements that motivated the development of the system is given in Simons (forthcoming, b). Rettig, Simons, and Thomson (1993) discuss some of the significant ways in which CELLAR extends the traditional object model. More information about CELLAR can be found in the following Worldwide Web page: *http://www.sil.org/cellar/cellar.html.*

To build an application in CELLAR one does not write a program in the conventional sense of a structure of imperative commands. Rather, one builds a declarative model of the problem domain. A complete *domain model* contains the following four components:

- *Conceptual model.* Declares all the object classes in the problem domain and their attributes, including integrity constraints on attributes that store values and built-in queries on those that compute their values on-the-fly.
- *Visual model.* Declares one or more ways in which objects of each class can be formatted for display to the user.
- *Encoding model.* Declares one or more ways in which objects of each class can be encoded in plain text files so that users can import data from external sources or export them.
- *Manipulation model.* Declares one or more tools which translate the interactive gestures of the user into direct manipulation of objects in the knowledge base.

Before looking at the details of how feature-structure markup has been implemented in this system, we first look at the basic strategy for mapping concepts of the CELLAR system onto concepts of the TEI feature-structure system.

## 4. The Basic Strategy for Implementing Feature-Structure Markup

One possible strategy for implementing feature-structure markup would be to define a CELLAR class named FeatureStructure and another named Feature. Occurrences of ⟨fs⟩ and ⟨f⟩ would be mapped onto instances of these two classes when a TEI-encoded file was loaded. This would be trivially easy; that is, it would be easy until we got to the point of wanting to use the FSD to validate the feature structures that had been loaded. At that point we would also have to implement CELLAR classes like FeatureSystemDeclaration which would contain instances of FeatureStructureDeclaration which would contain instances of FeatureDeclaration. (Note that the names of CELLAR classes are capitalized and may not contain spaces.) Then we would have to implement the logic for comparing instances of feature structures and features with their corresponding declarations to ensure that they were valid, and for computing default values for unexpressed features. Before long we would realize that we were reinventing the class definition component that was already built into our database system!

Our strategy is therefore to build on the analogy that feature structure is to feature as object is to attribute. The above strategy does not do this; it models both feature structures and features as objects. To model a feature as an attribute, it is necessary for the object that corresponds to the feature structure to have an attribute with the same name as the feature. The only way an object can get an attribute is that the definition of its class declares the attribute. Thus, feature-structure type must correspond to object class, so that every type of feature structure can map onto a different class of object which appropriately defines the attributes needed for its features.

This strategy means that before a file with ⟨fs⟩ and ⟨f⟩ markup can be loaded into the database as objects, the database system must already know the definitions for the classes that correspond to the feature-structure types. This is where the Feature System Declaration (FSD) comes in. It already provides an SGML encoding of the conceptual model for the different types of feature structure that are used in the markup. Our strategy is to first load this knowledge into the CELLAR database. The FSD as a whole maps onto a CELLAR DomainModel object, while each feature structure declaration maps onto a ClassDefn object and each feature declaration maps onto an AttributeDefn object. These mappings are summarized in Fig. 2.

Once the FSD has been loaded to initialize CELLAR with the needed class definitions, the feature structures themselves may be loaded. Fig. 3 summarizes the mappings from the elements of TEI feature-structure markup to the elements of CELLAR. All but the last three entries in that table should be self explanatory. ⟨sym⟩ is used in feature-structure markup

| TEI Markup Element | Corresponding CELLAR Element |
|---|---|
| <teiFsd> | DomainModel |
| <fsDecl type=X> | ClassDefn (for class X) |
| <fDecl name=Y> | AttributeDefn (for attribute Y) |

**Fig. 2.** Mappings from Elements of TEI FSD to Elements of CELLAR

| TEI Markup Element | Corresponding CELLAR Element |
|---|---|
| <fs type=X> | Object of class X |
| <f name=Y> | Attribute named Y |
| <plus> | Boolean true |
| <minus> | Boolean false |
| <nbr> | Integer |
| <str> | String |
| <sym> | Pointer to AuthorityListItem |
| <uncertain> | Missing |
| <none> | None |

**Fig. 3.** Mappings from Elements of TEI ⟨fs⟩ Markup to Elements of CELLAR

to encode a symbolic value that comes from a closed set; the possible values are enumerated in the FSD. In CELLAR, a closed set of symbolic values is modeled as an AuthorityList stored in the class definition. ⟨sym⟩ thus maps onto a pointer to one of the items stored in such a list. ⟨uncertain⟩ and ⟨none⟩ map onto two special objects which CELLAR uses when there is no attribute value: *missing* signifies that the value is unknown or uncertain, while *none* signifies that there is known to be no value at all.

The next two sections explain the implementation further by giving samples of the CELLAR source code and by demonstrating it with two extended examples. In order to illustrate the point that feature-structure markup can be used for general markup, not just linguistic analysis, the examples are not from linguistics. Section 5 uses a critically annotated text to demonstrate that any objects in the database may be viewed in terms of feature-structure markup. Section 6 uses a 'canine medical history' to demonstrate that arbitrary objects in feature-structure markup can be loaded into the CELLAR database.

## 5. Using Views to Map from Objects in a Database to Feature-Structure Markup

Section 3 explained that one component of a CELLAR application is the visual model which declares one or more ways in which objects of each class can be formatted for display to the user. In an earlier paper (Simons, forthcoming a), I demonstrated how CELLAR may display a single object in many different ways using multiple views defined on its class. One of the examples used in that paper was a CELLAR implementation of a critical edition of the Second Epistle of Clement. Fig. 4 reproduces one possible view of the critical text. This example is used in the following subsections to demonstrate that any object can be displayed in conventional feature-structure notation or in TEI feature-structure markup.

### 5.1 *Mapping Objects onto Conventional Feature-Structure Notation*

Given that an object with attributes is analogous to a feature structure with features, it should be possible to display the contents of any object in feature-structure notation. Fig. 5 gives an example. It is a screen shot of a CELLAR window displaying the first verse of the critical text in Fig. 4 through a view named *fs* for 'feature structure'. The screen is not large enough to show the entire feature structure for the verse; it goes as far as the comma in the second line of Fig. 4. The scroll bar can be used to examine the rest of the feature structure.

In the feature-structure notation of Fig. 5, square brackets enclose a

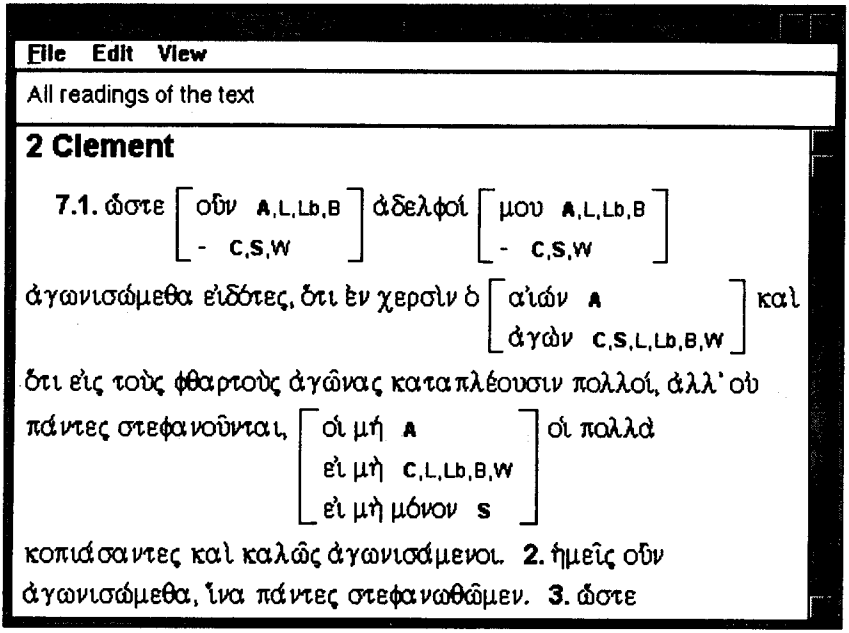


**Fig. 4.** A View Showing All Readings of a Critical Text

single object. The class of the object is given in italics at the top of the left bracket. The attributes of an object are named down the left side followed by an equals sign and the attribute value. When an attribute has a set or a sequence of values, the values are separated on the same line by commas or listed in a vertical pile. In this example, the attribute values are either simple strings, embedded complex objects (displayed as feature structures), or pointers to complex objects. The latter are displayed as an up arrow followed by the name of the pointed-to object.

The following is the source code for the view that produced the feature structure shown in Fig. 5:

```
enrich Object with
  view fs : frame showing (
    pile showing (
      name of my class using default
        with text.italic=true,
      lead with lead.height=6,
      for all attr in allOwningAttrNames of my class
        show (if storesValue(^attr) of self then
          row showing (
            ^attr using default, '=',
            pile showing (
```

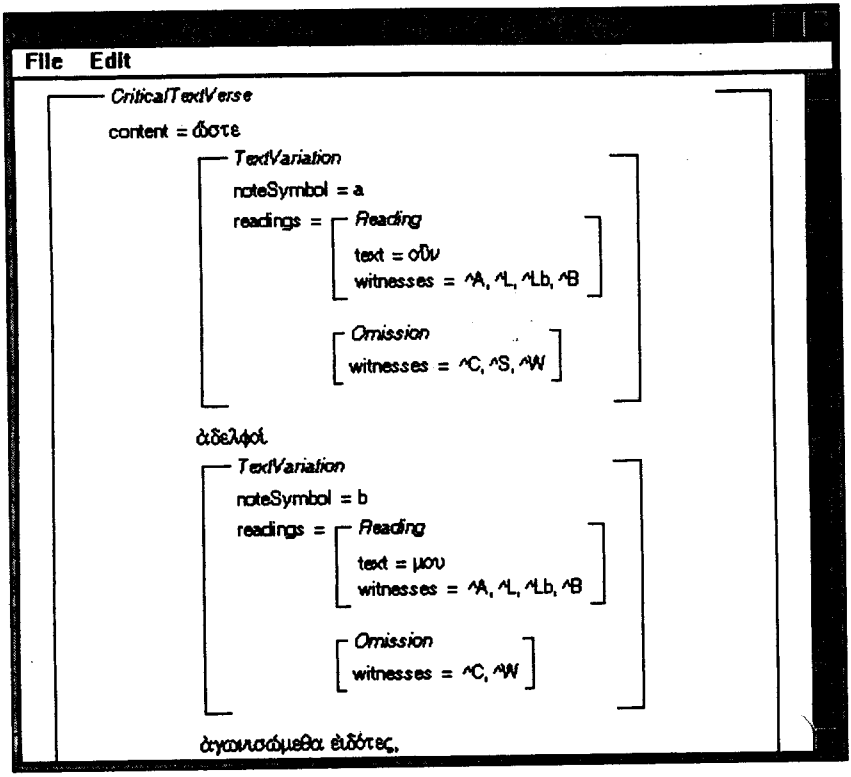


**Fig. 5.** A Verse from a Critical Text Displayed as a Feature Structure

```
        evaluate ^attr of self using fs)
    )),
  for all attr in allReferenceAttrNames of my class
    show (if storesValue(^attr) of self then
      row showing (
        ^attr using default, '=',
        paragraph showing (
          evaluate ^attr of self using fsPointer)
          with target.between=run showing
            (glue, ',')
        )),
  lead with lead.height = 12
  ))
with frame[borderStyle=bracket, above=false,
           below=false, extraLength=-6]
```

The view definition begins by declaring that the new view is an enrichment to the Object class. Every class in the CELLAR system is ultimately

a subclass of Object. Thus, by adding this view to the visual model for Object, we have added it (by means of object-oriented inheritance) to every class of object. This means that any object stored in the database can now be displayed as a feature structure.

Views are defined by specifying how the multiple data elements within them are to be laid out. CELLAR, following the lead from Donald Knuth's (1986) TEX system, builds a display as a structure of boxes within boxes. there are three basic kinds of grouping boxes: a *row* places its component boxes side by side, a *pile* places its component boxes one over the other, and a *paragraph* places its component boxes side by side until reaching the limit of available space at which point it continues making another line of boxes below the first and so on. Another kind of box is a *frame*; it embeds a single box and displays some kind of frame around it. Still another kind of box is *lead*; it generates a blank vertical space of a specified height.

The *fs* view is essentially a pile (third line) showing the class name and the attributes of the objects. This is enclosed in a frame (second line) which selects 'bracket' as its border style and turns off the display of border segments above and below the enclosed box (last two lines). The result is square brackets on the left and right; *extraLength=–6* reduces the length of the brackets by six points with the result that the top of the bracket aligns with the middle of the class name.

The heart of the *fs* view is the two *for all* constructs which iterate through all the attributes, first the owning attributes (which store component objects in place) and then the reference attributes (which point to related objects stored elsewhere). Each *for all* construct consults the definition for the class to obtain a list of all the attributes of the given type that are possible for this object. Each possible attribute name is tested in turn. If the object stores a value for that attribute, then a row is displayed containing the attribute name, an equals sign, and a view of the attribute value.

For an owning attribute, the values are shown recursively using this *fs* view (see the twelfth line). If there are multiple values, they are displayed in a pile. The recursion of feature structure within feature structure needs to stop when the value is simply a string, a number, or a Boolean. These are all subclasses of BasicObject which is in turn a subclass of Object. The definition of BasicObject is therefore enriched with a definition of the *fs* view that overrides the definition on Object; it simply shows the basic object in its default view. Thus:

```
enrich BasicObject with
   view fs : self using default
```

But we want Booleans to be displayed as plus and minus rather than in their default view (which is *true* or *false*). Thus, we enrich a Boolean with a yet deeper overriding definition as follows:

```
enrich Boolean with
   view fs : if self then '+' else '-'
```

For a reference attribute, the values are shown in a special view named *fsPointer*. This, too, is a device for stopping the recursion of feature structure within feature structure. Using the *fs* view to display the pointed-to object as a feature structure would not only repeat the same information every time it was pointed to, it would also result in an infinite recursion when attribute values formed a cycle of pointers. Thus, we define the following view which simply displays the name of the object preceded by an up arrow:

```
enrich Object with
   view fsPointer :
      row showing ('^', glue, my name using default)
```

The *glue* causes the boxes on both sides of it to be displayed without an intervening space.

### 5.2 *Mapping Objects onto TEI Feature-Structure Markup*

The *fs* view provides the graphic notation that is conventionally used for displaying feature structures. As discussed above in section 1, the TEI guidelines define another notation: an SGML-based notation for information interchange. In CELLAR terms, this interchange format is just one more possible view of an object. When we extend the definition of class Object by adding such a view, any object stored in a CELLAR database (no matter what its class) can be displayed in TEI feature-structure markup. The information can be exported in this format by executing a general menu command that writes the current formatted display to a plain text file.

In Simons (forthcoming, b) I demonstrated how views defined for the classes of the critical text domain could output the data in the TEI-conformant markup for critical texts. Here, by using the view called *teiFs* (which is defined on Object), the critical text objects can also be output in TEI feature-structure markup. Fig. 6 is a screen shot showing the same information as Fig. 5 but in ⟨fs⟩ notation.

The view which generated Fig. 6 is defined as follows:

```
enrich Object with
  view teiFs : pile showing (
    row showing ('⟨fs type=', glue,
                 name of my class using default,
                 glue, '⟩'),
    for all attr in allOwningAttrNames of my class
      show (if storesValue(^attr) of self then
        pretty-print showing (
          row showing ('⟨f name=', glue,
```

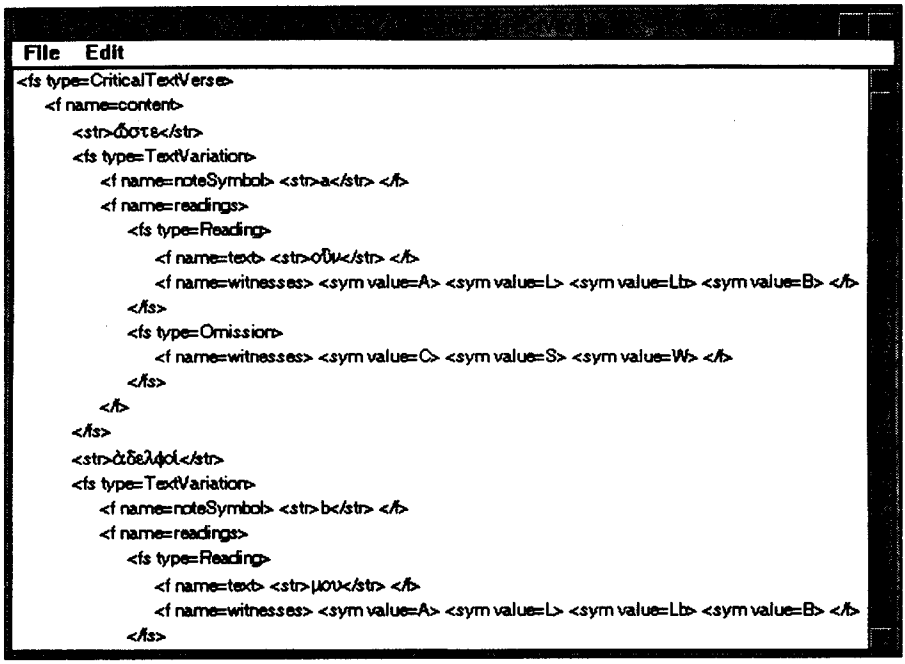


**Fig. 6.** Same Information as Fig. 5 Displayed in TEI Feature-Structure Markup

```
                    ^attr using default, glue, '⟩'),
        evaluate ^attr of self using teiFs,
        '⟨/f⟩')),
    for all attr in allReferenceAttrNames of my class
      show (if storesValue(^attr) of self then
        pretty-print showing (
          row showing ('⟨f name=', glue,
                      ^attr using default, glue, '⟩'),
          evaluate ^attr of self using teiSym,
          '⟨/f⟩')),
    '⟨/fs⟩')
    with pile[leftIndent=18, firstLineIndent=-18,
              lastLineIndent=-18]
```

This view definition is similar to the one for *fs*. It introduces just one new element of the programming language, namely, the *pretty-print* construct. A pretty-print box formats itself like a row if all of its contents will fit on one line; otherwise, it formats like a pile. For instance, in the first ⟨*fs type=TextVariation*⟩ of Fig. 6, the *noteSymbol* feature is formatted as a row while the *readings* feature is formatted as a pile. The indentation properties declared in the last two lines are what cause the indentation to

track the level of nesting. These declarations say that each pile (including those generated by pretty-prints) is to indent its content by 18 points, with the exception of the first and last lines (for the start and end tags, respectively) which are back at the original left edge of the pile.

The views for the BasicObject subclasses are straightforward, simply gluing the correct markup before and after the default view of the object.

```
enrich String with
   view teiFs : row showing (
      '⟨str⟩', glue, self using default, glue, '⟨/str⟩')
enrich Integer with
   view teiFs : row showing (
      '⟨nbr value=', glue, self using default, glue, '⟩' )
enrich Boolean with
   view teiFs : if self then '⟨plus⟩' else '⟨minus⟩'
```

The values of reference attributes are displayed with a view named *teiSym* which treats the name of the pointed-to object as a symbol from a closed list:

```
enrich Object with
   view teiSym : row showing (
      '⟨sym value=', glue, my name using default,glue, '⟩')
```

### 5.3 *Mapping Class Definitions onto a TEI Feature System Declaration*

A file encoded in TEI feature-structure markup is not complete for interchange unless it has an accompanying Feature System Declaration (FSD) to document what the feature structure types and their features represent and to specify constraints on well-formedness. Such information is already inside the class definitions of a CELLAR database. In the same way that we can use view definitions to map arbitrary objects onto TEI ⟨fs⟩ markup, we can use view definitions to map the class definitions of those objects onto a TEI FSD. Fig. 7 is an example; it shows the first screen of the automatically generated FSD for the objects displayed as TEI feature structures in Fig. 6.

A view named *teiFsd* was used to generate the display in Fig. 7. The correspondence between CELLAR elements and FSD elements is given above in Fig. 2. Space does not permit inclusion of the source code for these view definitions. Suffice it to say that the *teiFsd* view of class DomainModel generates a ⟨teiFsd⟩ tag, the ⟨teiHeader⟩, all its ClassDefns in their *teiFsd* view, and the ⟨/teiFsd⟩ end tag. The *teiFsd* view of class ClassDefn generates a comment header line, an ⟨fsDecl⟩ tag, the ⟨fsDescr⟩, all its AttributeDefns in their *teiFsd* view, and the ⟨/fsDecl⟩ end tag. The *teiFsd* view of class AttributeDefn generates an ⟨fDecl⟩ tag, the ⟨fDescr⟩, the ⟨vRange⟩, and the ⟨/fDecl⟩ end tag.

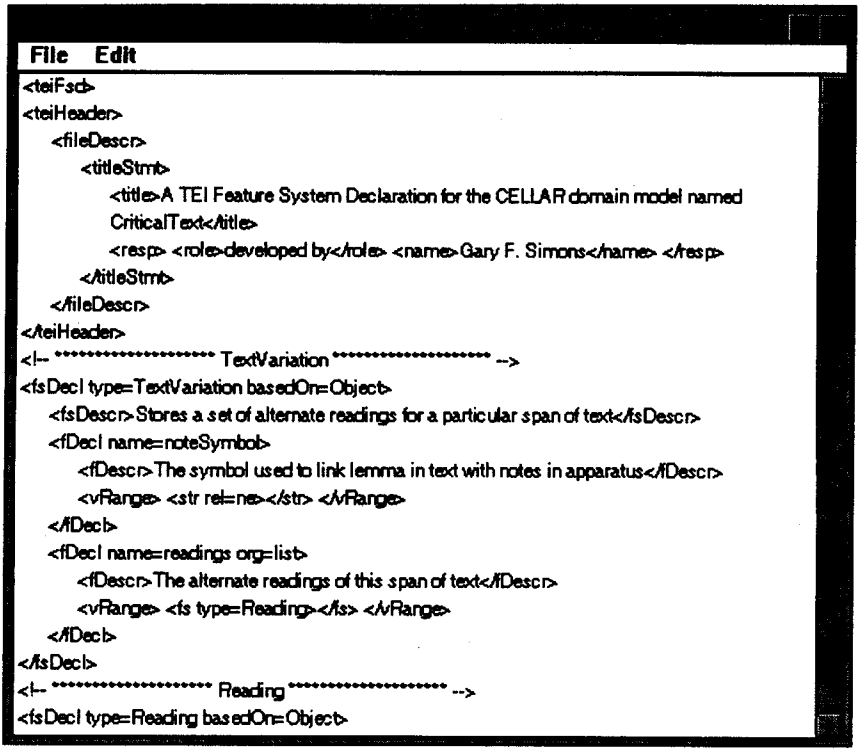


**Fig. 7.** The Domain Model for CriticalText Displayed as a TEI Feature System Declaration

## 6. Using Parsers to Load Objects into a Database from Feature-Structure Markup

Section 3 explained that another component of a CELLAR application is the encoding model which declares one or more ways in which objects of each class can be encoded in plain text files so that users can import data from external sources. The aim of this section is to demonstrate how a file of data in TEI feature-structure markup can be imported into a CELLAR database. To demonstrate this I use the 'canine medical history' developed by Langendoen (1994, 512–13) as an illustrative example in the TEI guidelines. Fig. 8 gives a listing of the input file; it was extracted without change from a downloaded version of the TEI guidelines. It is an encoding of the key information in a text from the British National Corpus named 'Memoirs of a Dog Shrink' which is about a collie who had a phobia for lights.

**Fig. 8.** Feature-Structure Markup of a Canine Medical History (*follows*)

```
<fs type='canine medical history'id=j37>
  <f name=name id=j37pn><str>Jessie</str></f>
  <f name=called.by org=set id=j37pc>
    <str>Jessie</str>
    <str>Jess</str>
    </f>
  <f name=breed id=j37b><sym value=collie>
  <f name=owner id=j37o>
    <fs type='owner description'>
      <f name=name><uncertain></f>
      <f name=address id=j37or><str>Surrey</str></f>
      </fs>
  <f name=illness org=list id=j37i>
    <fs type='case history' id=j37i1>
      <f name=name.of.specialist id=j37i1sn>
        <fs type='name structure'>
          <f name=last.name><str>Neville</str></f>
          <f name=first.name><str>Peter</str></f>
          </fs>
      <f name=title.of.specialist><uncertain>
      <f name=case.number id=j37i1n><nbr value=72></f>
      <f name=age.at.incidence><uncertain></f>
      <f name=date.of.incidence><uncertain>
      <f name=baseline.condition org=set id=j37i1b>
        <sym value=lazy>
        <sym value=friendly>
        <sym value=indoor>
        </f>
      <f name=symptoms id=j37i1s>
        <fs type='symptom structure'>
          <f name=behaviors org=set id=j37i1sb>
            <sym value=agitated>
            <sym value=destructive>
            <sym value=unfriendly>
            </f>
          <f name=particulars id=j37i1sp>
            <str>ran off, then returned and
             destroyed every lamp in the house</str></f>
          </fs>
        </f>
      <f name=diagnosis id=j37i1d>
        <fs type='diagnosis structure'>
          <f name=date.of.diagnosis><uncertain>
          <f name=disease id=j37i1dd>
            <str>light bulb phobia</str></f>
          <f name=presumed.cause id=j37i1dc>
            <str>explosion of light bulb over patient's head</str>
            </f>
          </fs>
        </f>
      <f name=treatment id=j37i1t>
        <fs type='treatment history'>
          <f name=medicine><none></f>
          <f name=regime id=j37i1tr><str>positive reinforcement</str></f>
          <f name=particulars id=j37i1tp>
            <str>systematically decreased distance between
             feeding bowl and table lamp</str></f>
          <f name=duration.of.treatment id=j37i1td>
            <msr unit=week value=2>
            </f>
          </fs>
        </f>
      <f name=result id=j37i1r>
        <str>return to baseline condition</str>
        </f>
      </fs>
    </f>
  </fs>
```

## 6.1 *Loading Objects from TEI Feature-Structure Markup*

An encoding model consists of a collection of parser definitions. Each parser is defined on a particular class and specifies how input text is converted to an object of that class. In order to convert feature-structure markup into objects in the database, we add a set of parsers named *teiFs* to the system. Again, this is done by extending the definition of class Object so that all classes in the system inherit the new behavior. As a result, any kind of object (no matter what its class) can be loaded into a CELLAR database by reading in a text file that uses TEI feature-structure markup.

The *teiFs* parser for class Object is defined as follows:

```
enrich Object with
  parser teiFs →
    { (peek '⟨fs'       Object.teiFs2)
      (peek '⟨minus⟩'   Boolean.teiFs)
      (peek '⟨plus⟩'    Boolean.teiFs)
      (peek '⟨nbr'      Integer.teiFs)
      (peek '⟨str⟩'     String.teiFs)
      (peek '⟨sym'      Object.teiSym)
      (peek '⟨msr'      String.teiMsr)
      ('⟨none⟩'         do( lit. none ))
      ('⟨uncertain⟩'    do( lit. missing ))
    } ?blank
```

Curly braces enclose a set of alternative patterns, one of which should succeed. In this case, the parser is peeking at the next tag in the input file to determine which parser for which class of objects to invoke. In the case of ⟨none⟩ and ⟨uncertain⟩, no further parser is needed; the response is to return a literal *none* value or *missing* value, respectively. The final *?blank* optionally matches a span of blank (space, tab, or newline) characters.

The parsers for the BasicObjects are straightforward. The Boolean parser returns a copy of the Boolean values *true* or *false* depending on whether it finds ⟨plus⟩ or ⟨minus⟩ in the file.

```
enrich Boolean with
  parser teiFs →
    { ('⟨plus⟩'    do(copy of true))
      ('⟨minus⟩'   do(copy of false))}
```

The Integer parser sets the basic value of a new integer to the value returned by calling the built-in parser of integers named *default*. The construct ⟨. . . =. . .⟩ is the syntax for setting an attribute of the object being constructed.

```
enrich Integer with
  parser teiFs →
    build Integer matching (
```

```
    '⟨nbr value='
    ⟨basicValue = Integer.default⟩ '⟩' )
```

The String parser works in two stages. It first matches all the content between the start and end tags and assigns it to a variable. It then submits that value to a String parser called *reduceBlanks* and uses the result of that for the value of the returned String. The latter parser converts spans of blanks to a single space, thus dealing with the extra spaces put in the input for pretty-printing purposes (see Fig. 8).

```
enrich String with
  parser teiFs →
    var string
    build String matching (
      '⟨str⟩'
      string := String.upTo('⟨/') '⟨/' ?'str' '⟩'
      ⟨basicValue= do(
        parse ^string using String.reduceBlanks )⟩
      )
```

The parser that does most of the work is *Object.teiFs2*. This is the parser that converts an ⟨fs⟩ construct into an object:

```
enrich Object with
  parser teiFs2 →
    var className, attr, ignore
    build Action(!Action (^className)) matching (
      '⟨fs type=' className:=
          { String.cellarName String.teiFsName} ?blank
      ?( 'id=' ignore:=String.upToAny(' ⟩') ?blank)
      '⟩' ?blank
      *('⟨f name=' attr:=String.teiFName ?blank
        *({ 'org=' 'id=' } ignore:=String.upToAny( '⟩')
          ?blank)
        '⟩' ?blank
        ⟨Action(^attr)=+Object.teiFs⟩
        {('⟨/f⟩' ?blank) (peek '⟨f ')})
      '⟨/fs⟩' )
```

The fourth line is performed after the entire pattern has been matched; it builds an object of the class named in the *className* variable. The pair of alternatives in the sixth line means that the name of the class (that is, the value of the SGML *type* attribute) may already be a valid CELLAR name or should be converted to one by invoking the parser *String.teiFsName*. The latter removes spaces from within a name and capitalizes the first letters of the words. The parser *String.teiFName* is similarly used for the feature names to remove internal periods and capitalize the first letters of the

words. The asterisk at the beginning of the line which matches '*⟨f name=*' means that the pattern within that set of parentheses is matched as many times as it can, in other words, once for each feature that is encoded. The *org* and *id* SGML attributes are simply ignored; *org* can be ignored because the CELLAR attribute definition already knows whether or not it is valid for the attribute to store more than one value. The third-to-last line is where the feature values are actually set; the attribute with the most recently matched *name* is set to the values returned by matching the *Object.teiFs* parser (the plus sign means that the parser must match one or more times). The alternation in the second-to-last line allows for the possibility that the end tag for the feature has been omitted.

Given that the CELLAR database has a valid class definition for each of the feature-structure types in the input file, the *teiFs* parser converts the TEI-encoded file into objects in the database. Fig. 9 shows the result of parsing the input file of Fig. 8. It is the *fs* view (as defined in section 5.1) of the resulting CanineMedicalHistory object. Once the *teiFs* parser has been used to import the encoded data into the CELLAR database, all the mechanisms of CELLAR's programming language are available to build views and interactive tools that are specific to manipulating that kind of data.

### 6.2 *Loading Class Definitions from a TEI Feature System Declaration*

As just stated, feature-structure markup can be converted into objects only if the CELLAR system already has definitions for the classes that correspond to the feature-structure types. The Feature System Declaration (FSD) is the component of TEI markup that documents what the feature structure types are and specifies well-formedness constraints in terms of allowed features and feature values. Fig. 10 is a listing of the FSD which was written to document the markup of the canine medical history shown in Fig. 8. The ⟨fsDecl⟩s for CanineMedicalHistory and OwnerDescription are given in full; those for the five other feature-structure types have been omitted to save space.

A parser named *teiFsd* was used to convert the file listed in Fig. 10 into a CELLAR domain model containing the class definitions corresponding to all the feature-structure types. Space does not permit inclusion of the source code for the parser definitions. In short, the *teiFsd* parser converts the ⟨teiFsd⟩ to a DomainModel, each ⟨fsDecl⟩ into a ClassDefn, and each ⟨fDecl⟩ into an AttributeDefn. The only change I made to the canine medical history FSD (that was originally encoded in 1993) was to place the feature declarations that use enumerated lists last and precede them with the SGML comment ⟨*!—Reference attributes* –⟩. In this way the *teiFsd* parser for ClassDefn easily knows when to create an OwningAttrDefn versus when to create a ReferenceAttrDefn. In the latter

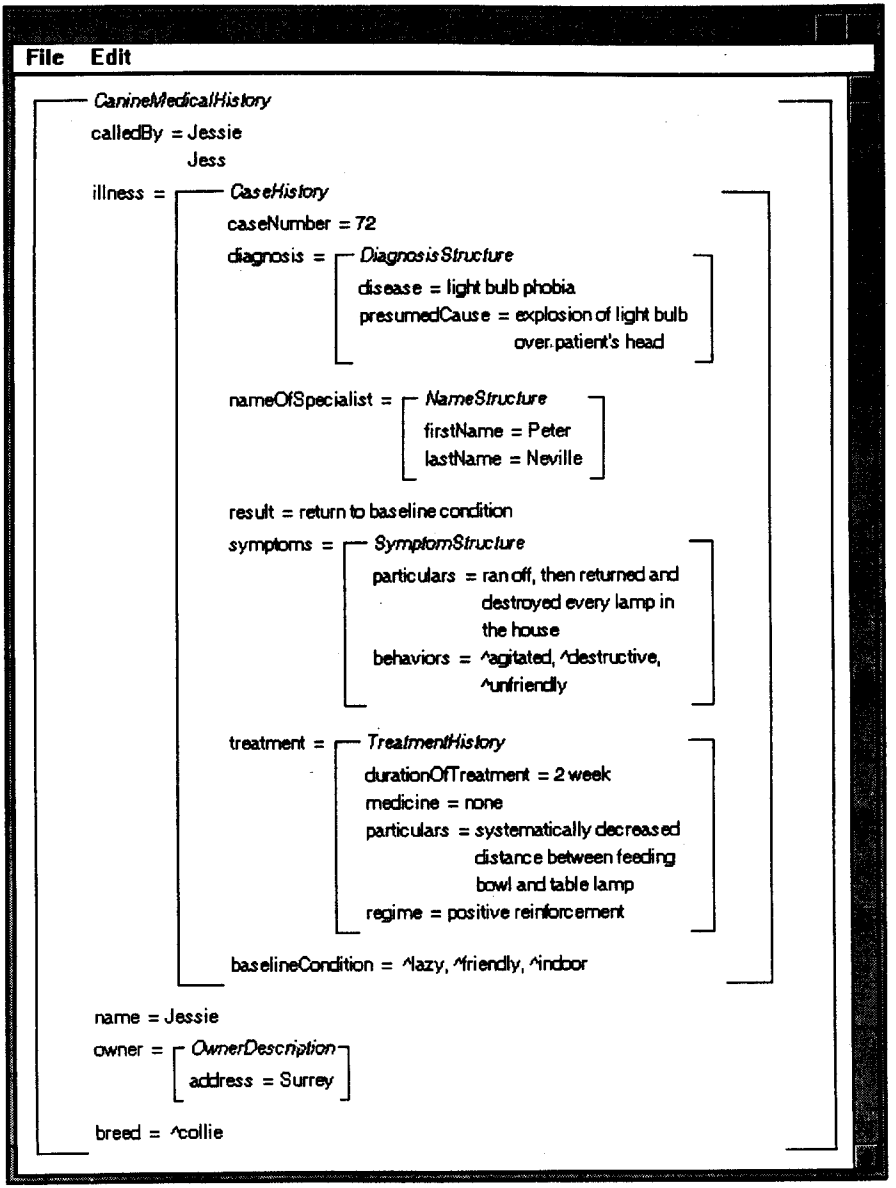


**Fig. 9.** Canine Medical History Displayed in Conventional Feature-Structure Notation

```
<teiFsd>
<teiHeader>
<fileDesc>
   <titleStmt>
      <title>An FSD for the "Canine medical history" example in the
         Feature Structures chapter of TEI P3, pp. 512-513</title>
      <resp><role>encoded by</role><name>Gary F. Simons</name></resp>
   </titleStmt>
   <publicationStmt>This FSD was encoded August 14, 1993 for the
purpose of serving as a nonlinguistic example of the use of FSDs.
It is a hypothetical example based on the sample case history encoded
in pages 512-513 of the TEI Guidelines.</publicationStmt>
</fileDesc>
</teiHeader>
<!-- ************ Canine Medical History ************** -->
<fsDecl type='Canine medical history'>
<fsDescr>Used to encode the medical history of a dog; see
example on pages 512-513 of TEI P3.</fsDescr>
<fDecl name=name>
   <fDescr>The given name of the dog</fDescr>
   <vRange><str rel=ne></str></vRange>
   </fDecl>
<fDecl name=called.by org=set>
   <fDescr>Names by which the dog is called</fDescr>
   <vRange><str rel=ne></str></vRange>
   </fDecl>
<fDecl name=owner>
   <fDescr>Identification of the dog's owner</fDescr>
   <vRange><fs type='owner description'></fs></vRange>
   </fDecl>
<fDecl name=illness org=list>
   <fDescr>The history of illnesses for which this dog has
been diagnosed and treated</fDescr>
   <vRange><fs type='case history'></fs></vRange>
   </fDecl>
<!-- Reference attributes -->
<fDecl name=breed>
   <fDescr>Breed of dog (selected from authority list)</fDescr>
   <vRange>
      <vAlt><sym value='cocker spaniel'><sym value=collie>
            <sym value=dachsund><sym value='english setter'>
            <sym value='german shepherd'><sym value='great dane'>
            <!--Add symbols to extend the authority list-->
      </vAlt></vRange>
   </fDecl>
</fsDecl>
<!-- *************** Owner Description *************** -->
<fsDecl type='owner description'>
<fsDescr>Encodes information about the person who owns adog.</fsDescr>
<fDecl name=name>
   <fDescr>Name of the dog's owner</fDescr>
   <vRange><str rel=ne></str></vRange></fDecl>
<fDecl name=address>
   <fDescr>Address of the dog's owner</fDescr>
   <vRange><str rel=ne></str></vRange></fDecl>
</fsDecl>
<!-- The <fsDecl>s for the following are omitted to save space:  -->
<!-- *************** Case History ******************** -->
<!-- *************** Name Structure ****************** -->
<!-- *************** Symptom Structure *************** -->
<!-- *************** Diagnosis Structure ************* -->
<!-- *************** Treatment History *************** -->
</teiFsd>
```

**Fig. 10.** Feature System Declaration for a Canine Medical History: Tweedie, Singh & Holmes

case, the parser reads the enumerated list of symbols in the ⟨vAlt⟩ and creates an AuthorityList out of them.

## 7. Conclusion

The TEI guidelines propose feature-structure markup with the suggestion that it can be used to encode virtually any text-oriented information or analysis of such information. An obstacle to the use of this notation has been the absence of software for processing it.

This paper has demonstrated that one approach to solving this problem is to map the feature structures and features of a TEI-encoded file onto the objects and attributes of an object-oriented database. The CELLAR system, with its user-definable views for export formatting and parsers for import processing, has proven able to do this task. By adding a *teiFs* view to the definition of class Object, any object already stored in the database can be exported in feature-structure markup; by adding a *teiFs* parser, an object of any class can be created by importing it from a marked up file.

Perhaps even more significant than this is the way the elements of a TEI Feature System Declaration have been mapped onto CELLAR class definitions. The implementation is achieved by adding a *teiFsd* view and parser to the definition of class ClassDefn (which defines the behavior of all class definitions in the system). As a result, the definition of any class that has been implemented in CELLAR can be displayed and exported as a TEI FSD. (Note, however, that any information for which there is no FSD equivalent must necessarily be lost.) Conversely, a TEI FSD can be loaded by CELLAR to create the class definition that is needed for loading TEI feature structures of a particular type. The net result is that the TEI FSD formalism becomes an alternate programming language syntax for expressing conceptual models in CELLAR. More generally this demonstrates the potential of the TEI's FSD formalism for serving as a *lingua franca* among database systems for the interchange of basic data models.

## References

Ait-Kaci, H. (1985), 'A New Model of Computation Based on a Calculus of Type Subsumption', Doctoral dissertation (University of Pennsylvania, Philadelphia, PA).

Borgida, A. (1985), 'Features of Languages for the Development of Information Systems at the Conceptual Level', *IEEE Software* 2(1): 63–72.

Cardelli, L. (1984), *A Semantics of Multiple Inheritance*, Technical Report, Bell Laboratories (Murray Hill, N.J.).

Ide, N., Le Maitre, J., and Veronis, J. (1993), 'Outline of a Model for Lexical Databases', *Information Processing and Management* 29(2): 159–86.

ISO (1986), *Information Processing—Text and Office Systems—Standard Generalized Markup Language (SGML)*, ISO 8879-1986 (Geneva, International Organization for Standardization).

Knuth, D. E. (1968), *The Art of Computer Programming, vol. 1: Fundamental Algorithms* (Reading, MA).

Knuth, D. E. (1986), *The TEXbook*, Volume A of *Computers and Typesetting* (Reading, MA).

Langendoen, D. T. (1994), 'Feature Structures', in C. M. Sperberg-McQueen and L. Burnard, 1: 475–519.

Langendoen, D. T., and Simons, G. F. (forthcoming), 'A Rationale for the TEI Recommendations for Feature-Structure Markup'. To appear in *Computers and the Humanities* special issue on the Text Encoding Initiative.

Rettig, M., Simons, G., and Thomson, J. (1993), 'Extended Objects', *Communications of the ACM* 36(8): 19–24.

Shieber, S. (1986), *An Introduction to Unification-Based Approaches to Grammar*, Center for the Study of Language and Information, Lecture Notes 4 (Stanford, CA).

Simons, G. F. (1994), 'Feature System Declaration', in C. M. Sperberg-McQueen and L. Burnard, 2: 701–13.

Simons, G. F. (forthcoming), 'The Nature of Linguistic Data and the Requirements of a Computing Environment for Linguistic Research'. To appear in J. Lawler and H. Dry (eds.), *Computing and the Ordinary Working Linguist* (New York).

Sperberg-McQueen, C. M., and Burnard, L. (1994). *Guidelines for the Encoding and Interchange of Machine-readable Texts* (Chicago and Oxford).