Reference number of working document: **ISO/TC 37/SC 4 N 188**

Date: 2004-10-01

Reference number of document: **ISO/proposed DIS 24610-1**

Committee identification: ISO/TC 37/SC 4/WG 1

Secretariat: KATS

Language Resource Management — Feature Structures — Part 1: Feature Structure Representation

Gestion des ressources linguistiques — Structures de traits — Partie 1: Representation de structures de traits

Document type: International standard
Document subtype: if applicable
Document stage: (20) Preparation
Document language: E

Contents

Page

Forword

ISO (The International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization. International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3.

Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75% *of the member bodies casting a vote*.

International Standard 24610, Part 1 was prepared by Technical Committee ISO/TC 37, Terminology and Other Language Resources (principles and coordination), Subcommittee SC 4, Language Resource Management. ISO 24610, Part 1 is designed to coordinate closely with Part 2 Feature System Declaration. All the Annexes are are for information only.

Introduction

This standard proposal results from the agreement between the Text Encoding Initiative Consortium and the ISO committee TC 37/SC 4 that a joint activity should take place to revise the two existing chapters on Feature Structures and Feature System Declaration in *The TEI Guidelines* called *P4*. This work should lead to both a thorough revision of the guidelines and the production of an ISO standard on Feature Structure Representation and Feature System Declaration.

This standard is organized in two separate main parts. The present document, constituting Part One of ISO 24610, is dedicated to the description of what feature structures are, providing an informal and yet explicit outline of their basic characteristics, as well as an XML-based structured way of representing feature structures. This preliminary task is designed to lay a basis for constructing an XML-based reference format for exchanging feature structures between applications. The second part of ISO 24610, will provide an implementation standard for XML-based feature structures, first by formulating constraints on a set of features and a set of their appropriate values and then by introducing a set of wellformedness conditions on feature structures for particular applications, especially related to the goal of language resource management.

Language Resource Management — Feature Structures — Part 1: Feature Structure Representation

1 Scope

Feature structures are an essential part of many linguistic formalisms as well as an underlying mechanism for representing the information consumed or produced by and for language engineering applications. This international standard provides a format to represent, store or exchange feature structures in natural language applications, for both annotation and production of linguistic data. It is ultimately designed to provide a computer format to describe the constraints that bear on a set of features, feature values, feature specifications and operations on feature structures, thus offering means to check the conformance of each feature structure with regards to a reference specification.

2 Normative References

There are four main normative references for the current International Standard:

— The TE I Guidelines, Text-Encoding Initiative Consortium.

— Bray, Tim, Jean Paoli, C.M. Sperberg-McQueen, and Eve Maler (eds.), *extensible Markup Language (XML)*, 1.0 (Second edition), World Wide Web Consortium (W3C) Recommendation, October 6, 2000. (Referenced in ISO 16642 and available at http://www.w3.org/TR/1998/REC-xml-19980210).

— ISO 8879: 1986 (SGML) as extended by TC2 (ISO/IEC JTC 1/SC 34 N029: 1998-12-06) to allow for XML.

— ISO 19757-2, Document Schema Definition Language, part 2.

3 Terms and Definitions

3.1
alternation
operation on feature values that returns one and only one value
NOTE Given a feature specification $F : a \mid b$, where $a \mid b$ denotes the alternation of a and b, F has either the value a or the value b, but not both.

3.2
atomic value
value that has no internal **feature specifications**

3.3
boxed label
label used in a **matrix notation** to denote **structure sharing**
NOTE The index is generally marked with an integer, but it can be any alpha-numeric symbol that can be used as a coreferential index.

3.4
collection
list, set or multiset of **values**
NOTE A list is an ordered collection of entities some of which may be identical, whereas a set or a multiset is an unordered collection of entities. These two differ from each other in that identical entities may not occur in a set, but may occur in a multiset. A multiset may some times be referred to as a bag.

3.5
complex value
value represented either as a **feature structure** or as **collection**

3.6
concatenation
operation that combines two lists of values into a single list
NOTE The equivalent operation for sets and multisets is **union**.

3.7
empty feature structure
feature structure containing no **feature specifications**

3.8
feature
property of an entity
NOTE When provided with a value, it constitutes a feature specification, for example “number" is a feature, a pair “number singular" is a feature specification, “singular" is a value.

3.9
feature specification
assignment of a **value** to a **feature**
NOTE Formally, it is treated as a pair of a feature and its value.

3.10
feature structure
set of **feature specifications**
NOTE The minimum feature structure is the **empty feature structure**

3.11
graph notation
notation that uses a single rooted graph with a finite sequence of labelled and directed arcs to represent **feature structures**
NOTE The name of each node including the root represents a type, the label of each arc a feature, and the name of the terminating node of each arc a value.

3.12
incompatibility
relation between two **feature structures** which have at least one common feature with a conflicting value
NOTE Two **feature structures** that are not incompatible can be unified. The **empty feature structure** is compatible with any other **feature structure**.

3.13
matrix notation
attribute-value matrix
AVM
notation that uses square brackets to represent **feature structures**
NOTE In a matrix notation, each row represents a **feature specification**, separated by a colon (:), space ( ) or the equality sign (=).

3.14

negation
(unary) operation on a **value** denoting any other **value** incompatible with it
NOTE In this standard, negation applies to **values** only and is not understood as a truth function as in ordinary bivalent logics.

3.15
path
sequence of arc labels in the **graph notation** originating from the root node
NOTE The notion of path can be extended in the same manner to the **matrix notation**. If the graph consists only of the root node, then the sequence is null, thus forming a null path. A path can also be partial, by forming a sequence from the root to a given node.

3.16
structure sharing
re-entrancy
relation between two or more **features** within a **feature structure** that share a value
NOTE Sharing can be reciprocal, but fails to apply if it would result in incompatibility.

3.17
subsumption
relationship between two feature structures in which one is more specific than the other
NOTE A feature structure A is said to subsume a feature structure B if A is not more informative than B. Subsumption is a reflexive, anti-symmetric and transitive relation between two feature structures.

3.18
type
class of entities
NOTE Primitive entities that **features** take as their **value** are considered types, too, namely unit classes consisting of themselves. For example, words can be sorted into types such as ‘verbs’, ‘nouns’, ‘adjectives’, and so on.

3.19
typed feature structure
feature structure sorted into a **type** which is labelled by the name of the **type**
NOTE In the graph notation, each node is labeled with the name of a type. In the matrix notation, a type name is ordinarily placed at the upper left corner of the inside of the pair of square brackets that represents a typed feature structure.

3.20
unification
operation that combines two compatible feature structures into one
NOTE Feature structures can be unified only if they do not have an **incompatibility**. Feature structures carry partial information and, by unification, their information is incremented.

3.21
union
operation that combines two sets, or multisets, into one
NOTE The equivalent operation for lists is **concatenation**.

3.22
value
information about an entity
NOTE There are two **types** of feature values: **atomic value** and **complex value**.

4 General Characteristics of Feature Structure

4.1 Overview

A *feature structure* is a general-purpose data structure that identifies and groups together individual *features* by assigning a particular value to each. Because of the generality of feature structures, they can be used to represent many different kinds of information. Interrelations among various pieces of information and their instantiation in markup provide a *metalanguage* for representing linguistic content analysis and interpretation. Moreover, this instantiation allows a specification of a set of features with values of specific *types* and restrictions, by means of *feature system declarations*, or other XML mechanisms discussed in the second part of ISO 24610.

4.2 Use of Feature Structures

Feature structures provide *partial information* about some objects by specifying *values* for some or all of its features. For example, if we are describing a female employee named Sandy Jones who is 30 years old, we can talk about that person's sex, name and age in a succinct manner by assigning a value to each of these three features. These pieces of information can be put into a simple set notation, as in:

(1) Employee
{<SEX, *female*>, <NAME, *Sandy Jones*>, <AGE, *30*>}

The use of feature structures can easily be extended to linguistic descriptions. The phoneme /p/ in English, for instance, can be analyzed in terms of its distinctive features: consonantal, anterior, voiceless, non-continuant or stop sound segment, etc. By using the Boolean values *plus*(+) and *minus*(-), these features can be listed as a set consisting of feature/value pairs, where the value of a feature specifies the presence or absence of that feature:

(2) Sound segment /p/
{<CONSONANTAL, + >, <ANTERIOR, + >, <VOICED, - >, <CONTINUANT, - >}

As a result, the sound segment /p/ can be distinguished from other phonemes in terms of presence or absence of specific features. For example, /p/ differs from the phoneme /b/ in VOICING, and from /k/ in articulatory position: one is articulated at the anterior, namely lip or alveolar area of the mouth, and the other at the non-anterior, namely the back of the oral cavity.

This feature analysis can be extended to the description of other linguistic entities or structures. Consider a verb like 'love'. Its features can be divided into syntactic and semantic properties: as a transitive verb, it takes an object as well as a subject as its arguments, expressing the semantic relation of loving between two persons or animate beings. The exact representation of these feature specifications requires a detailed elaboration of what feature structures are. For now, we can roughly represent these grammatical features in a set format like the following:

(3) Grammatical features of the verb 'love'
{<POS, *verb*>, <VALENCE, *transitive*>, <SEMANTIC_RELATION, *loving*>}

Since its first extensive use in generative phonology in mid-60's, a feature structure has become an essential tool not only for phonology, but also for doing syntax and semantics as well as lexicon building, especially in computational work. Feature structures are used to describe and model linguistic entities and phenomena by specifying their properties. In the next sections, we outline some of the formal properties of feature structures together with means to represent them in a systematic manner.

4.3 Basic Concepts

Feature structures may be viewed in a variety of ways. The most common and perhaps the most intuitive views are the following:

— (i) a set of *feature specifications* that consists of pairs of *features* and their *values*

— (ii) *labelled directed graphs with a single root* where each arc is labelled with the name of a feature and directed to its value.

In set-theoretic terms, a feature structure $\mathcal{FS}$ can be defined as a *partial function* from a set **Feat** of features to a set **FeatVal** of values, where **FeatVal** consists of a set **AtomVal** of atomic values and a set **FS** of feature structures.

(4) A feature structure as a set or partial function

$\mathcal{FS} \subseteq \{< F_i, v_i > | F_i \in \mathbf{Feat}, v_i \in \mathbf{FeatVal}\}$
or
$\mathcal{FS} : \mathbf{Feat} \longrightarrow \mathbf{FeatVal}$,
where **FeatVal** = **AtomVal** $\cup$ **FS**.

where featval stands for all possible values. Values may be regarded as either atomic or complex. Atomic values are entities without internal structure, while complex values may be feature structures or collections of values (either complex or atomic).

For example, the part of speech feature (POS) can take the name of an atomic morpho-syntactic category such as *verb* as its value. Conversely, the agreement feature AGR in English takes a complex value in the form of a feature structure with features PERSON and NUMBER. The word 'loves', for instance, has a POS feature with the value *verb*, while the value of its AGR feature consists of a feature structure comprising two feature specifications: PERSON with value *3rd*, and NUMBER with value *singular*.

4.4 Notations

As a list of feature-value pairs, the overall form of a feature structure is simple. However, the internal structure of a feature structure may be complex when a feature structure contains either (1) a feature whose value is itself a feature structure or (2) a multivalued feature whose value is a list, set or multi-set. Lists, sets, and multi-sets may be made up of atomic values, or they may include complex values that are themselves feature structures. That is, feature structures allow limitless *recursive embedding*. It is therefore necessary to represent them in an understandable, mathematically precise notation.

Three notations are commonly used to represent features structures: (1) graphs, (2) matrixes, and (3) an XML-based notation. In the following subsections, we discuss how the same feature structure may be represented using each of these equivalent notations. Graphs are suitable for mathematical discourses, matrices for linguistic descriptions, and XML notations for computational implementation.

4.4.1 Graph Notation

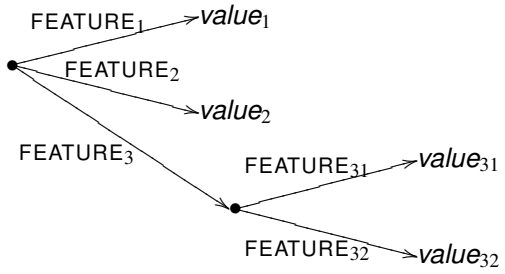
For conceptual coherence and mathematical elegance, feature structures are often represented as labelled directed graphs with a single root.[1)]

Each graph starts with a single particular node called *the root*. From this root, any number of *arcs* may branch out to other nodes and then some of them may terminate or extend to other nodes. The extension of directed arcs must, however, stop at some terminal nodes. On such a graph which is representing a feature structure, each arc is labelled with a *feature* name and its directed node, labelled with its *value*.

Here is a very simple example for a directed graph representing a feature structure.

1) This graph can be either (1) *acyclic*, in which case they are referred to as *directed acyclic graphs* (DAG) or (2)*cyclic* for handling cases like the Liar's paradox.
Feature structures are usually represented as DAGs; but it has been suggested that cyclical feature structures should be introduced to model some linguistic phenomena.

(5) Feature structure in graph notation



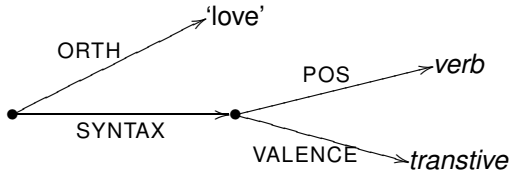
In this graph, the two features FEATURE$_1$ and FEATURE$_2$ are atomic-valued, taking $value_1$ and $value_2$ on the terminal nodes respectively as their value. The feature FEATURE$_3$ is, however, complex-valued, for it takes as its value the feature structure which is represented by the two arcs FEATURE$_{31}$ and FEATURE$_{32}$ with their respective values, $value_{31}$ and $value_{32}$.

A graph may just consist of the root node only, that is, without any branching arcs. Such a graph represents the *empty feature structure*.

>From the root, more than one arcs may branch out and each of them forms a sequence of feature names of length 1. Here, each sequence consists of a single feature name. Some of these labelled arcs originating from the root may again stretch out to another node and then from this node to another, forming an indefinitely long sequence of feature names. Such a sequence of feature names, labelling the arcs from the unique root node to each of the terminal nodes on a graph, is called *path*. For example, there are four paths in (5): FEATURE$_1$, FEATURE$_2$, FEATURE$_3$.FEATURE$_{31}$ and FEATURE$_3$.FEATURE$_{32}$.

Here is a linguistically more relevant example.

(6) Linguistic example in graph notation



This graph consists of three paths: ORTH, SYNTAX.POS, and SYNTAX.VALENCE. The path consisting of a single feature name ORTH is directed to the terminal node 'love', which is an atomic value. The path SYNTAX.POS terminates with the atomic value *verb* and the path SYNTAX.VALENCE with the atomic value *transitive*. The non-terminal feature SYNTAX takes a complex value, namely a feature structure consisting of the two feature specifications, <POS, *verb*> and <VALENCE, *transitive*>.

4.4.2 Matrix Notation

Despite its mathematical elegance, graphs cause problems of typesetting and readability when they get complex. To remedy some of these problems, feature structures are more often depicted in a matrix notation called *attribute-value matrix*, or simply AVM.[2)]

(7) Matrix notation

2) The term 'feature' is sometimes called *attribute*.

$$\begin{bmatrix} \text{FEATURE}_1\ value_1 \\ \text{FEATURE}_2\ value_2 \\ \text{FEATURE}_3 \begin{bmatrix} \text{FEATURE}_{31}\ value_{31} \\ \text{FEATURE}_{32}\ value_{32} \\ \ldots \\ \text{FEATURE}_{3k}\ value_{3k} \end{bmatrix} \\ \ldots \\ \text{FEATURE}_n\ value_n \end{bmatrix}$$

Each row in the square bracket with a FEATURE name followed by its *value* name represents a feature specification in a feature structure.[3] Feature values can be either atomic or complex. Each row with an atomic value terminates at that value. But if the value is complex, then that row leads to another feature structure, as in the case of FEATURE_3 above.

The notion of *path* is also important in the AVM notation for its applications that will be discussed presently. A *path* in an AVM is a sequence of feature names, as is the case with feature structure graphs. If an AVM has no row and thus no occurrences of features, being represented as [ ], such an AVM represents the empty feature structure and only has the empty path of length 0. But if an AVM has at least one row consisting of a feature name and its value, then there is a path of length 1 corresponding to each occurrence of a feature name in each row. Given a path of length i, if the last member of that path takes a nonempty feature structure as value, then that path forms a new path of length $i+1$ by taking each one of the features in that feature structure as its member.

For illustration, consider the following AVM which represents the same feature structure as is represented by the graph notation (6).

(8) Example of an AVM notation

$$\begin{bmatrix} \text{ORTH}\ \text{‘love’} \\ \text{SYNTAX} \begin{bmatrix} \text{POS}\ \textit{verb} \\ \text{VALENCE}\ \textit{transitive} \end{bmatrix} \end{bmatrix}$$

This AVM has three paths: ORTH, SYNTAX.POS and SYNTAX.VALENCE.

4.4.3 XML-based Notation

Like any other formalism, XML has its own terminology, which we have used in the remainder of this section. An XML document consists of typed *element*s, occurrences of which are marked by *start-* and *end-* tags which enclose the *content* of the element. Element occurrences can also carry named *attributes*, (or, more exactly, attribute-value pairs). For example:

(9) `<word class="noun">love</word>`

This is the XML way of representing an occurrence of the element *word*, the content of which is the string “love". The *word* element is defined as having an attribute called *class*, whose value in this case is the string “noun".

The term *attribute* is thus used in a way which potentially clashes with its traditional usage in discussions of feature structures. The acronym AVM, used in the preceding section, stands for *Attribute-Value Matrix*: in this method, feature structures are represented in a square bracket form, with each row representing an *attribute-value* pair. In this usage, an *attribute* does not correspond necessarily to an XML attribute as we show below,

3) A colon, an equality sign or a space separates a feature from its value on each row of an AVM.

the same information may be conveyed in many different ways. To avoid this potential source of confusion, the reader should be aware that in the remainder of this section we will use the term *attribute* only in its technical XML sense.

To illustrate further the distinction, we now consider how a feature structure in AVM may be represented in XML. Consider the following AVM that represents a non-typed feature structure:

(10) Representation of a feature structure in AVM

$$\begin{bmatrix} \text{ORTH } \textit{love} \\ \text{POS } \textit{noun} \end{bmatrix}$$

This feature structure consists of two feature specifications: one is a pair of a feature ORTH and its value *love* and the other, a pair of a feature POS and its value *noun*. [4)]

The basic structure to be represented in XML is that we have an element called a *feature structure*, consisting of two *feature* specifications. We could use the generic names **fs** and **f** for these elements:

(11)
```
<fs>
 <f>...</f>
 <f>...</f>
</fs>
```

This is a more generic approach than another, equally plausible, representation, in which we might represent each feature specification by a specifically-named element:

(12)
```
<fs>
 <orth>...</orth>
 <pos>...</pos>
</fs>
```

Using this more generic approach means that systems can be developed which are independent of the particular feature set, at the expense of slightly complicating the representation in any particular case. By representing the specification of a *feature* as an **f** element, rather than regarding each specific feature as a different element type simplifies the overall processing model considerably.

Feature specification, as we have seen, has two components: a name, and a value. Again, there are several equally valid ways of representing this pair in XML. We could, for example, choose any of the following three:

(13) (a)
```
<f name="pos" value="noun"/>
```

(b)
```
<f name="pos">
  <value>noun</value>
</f>
```

(c)
```
<f>
 <name>pos</name>
 <value>noun</value>
</f>
```

4) Note incidentally that this representation says nothing about the range of possible values for the two features: in particular, it does not indicate that the range of possible values for the ORTH feature is not constrained, whereas (at least in principle) the range of possible values for the POS feature is likely to be constrained to one of a small set of valid POS codes. In a constraint-based grammar formalism, possible values for the feature ORTH must be strings consisting of a finite number of characters drawn from a well-defined character set, say the Roman Alphabet plus some special characters, for each particular language.

The fact that all three of these are possible arises from a redundancy introduced in the design of the XML language (largely for historical reasons which need not concern us: see further [ref to be supplied]) by its support for attributes (in the XML sense!). As currently formulated, the present recommendation is to use a formulation like (b) above, though (c) may be preferable on the grounds of its greater simplicity.

Finally, we consider the representation of the *value* part of a feature specification. A name is simply a name, but a value in the system discussed here may be of many different types: it might for example be an arbitrary string, one of a predefined set of codes, a Boolean value, or a reference to another (nested) feature structure. In the interests of greater expressivity, the present system proposes to distinguish amongst these kinds of value in the XML representation itself.

Once more, there are a number of more or less equivalent ways of doing this. For example:

(14) (a)
```
<fs>
  <f name="orth">
    <value type="string">love</value>
  </f>
  <f name="pos">
   <value type="symbol">noun</value>
  </f>
</fs>
```

(b)
```
<fs>
  <f name="orth"><string>love</string></f>
  <f name="pos"><symbol value="noun"/></f>
</fs>
```

The number of possible different value types is comparatively small, and the advantages of handling values generically rather than specifically seem less persuasive. The approach currently taken is therefore to define different element types for the different kinds of value (for example, **string** to enclose a string, **symbol** to denote a symbol, **binary** to denote a binary value, **fs** to denote a nested feature structure).

Representing feature structures in XML notation is thus possible and rather straightforward, although a fuller implementation might involve various problems and possibilities. For example, the feature structure in the following shows a way to represent the same feature structure given in (6) and (8) in the above.

(15) Feature structure in XML notation

```
<fs>
  <f name="orth"><string>love</string></f>
  <f name="syntax">
    <fs>
      <f name="pos"><symbol value="verb"/></f>
      <f name="valence"><symbol value="transitive"/></f>
    </fs>
  </f>
</fs>
```

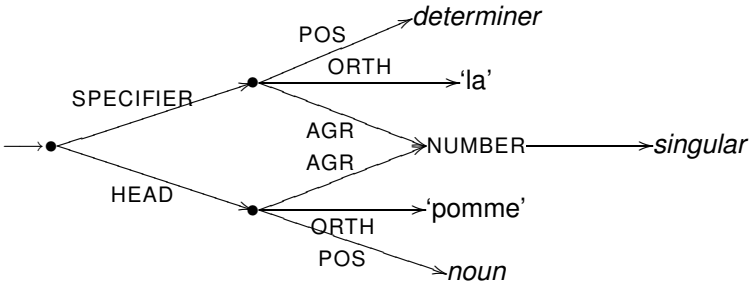
The feature structure in XML notation is surrounded in <fs> tags, and each specification of its features with <f> tags. Note that despite the apparent difference between the representations (6), (8) and (15), the information contained in each structure corresponds quite systematically with that in other structures. For example, arc labels in graph notation like ORTH and SYNTAX are now represented by <f> tags with appropriate names.[5)]

5) A detailed discussion including ways to simplify the representation in (15) will be provided in Section 5.

4.5 Structure Sharing

A feature structure representation may need to represent shared components. For example, in representing the structure of a noun phrase which includes an article (“la pomme") it may be desired to show that the feature “number" is shared between the article and the noun. This sharing can be directly represented in the graphic notation as follows:

(16) Merging paths in graph notation



Here, the two SPECIFIER.AGR and HEAD.AGR paths merge on the node NUMBER, indicating that they share one and the same feature structure as their value.

In AVM notation, structure sharing is represented by providing a boxed integer label at the point where structures are shared, as in the following example:

(17) Structure sharing in AVM notation

$$\begin{bmatrix} \text{SPECIFIER} & \begin{bmatrix} \text{POS} & \textit{determiner} \\ \text{ORTH} & \text{'la'} \\ \text{AGR} & \boxed{1}\begin{bmatrix}\text{NUMBER} & \textit{singular}\end{bmatrix} \end{bmatrix} \\ \text{HEAD} & \begin{bmatrix} \text{POS} & \textit{noun} \\ \text{ORTH} & \text{'pomme'} \\ \text{AGR} & \boxed{1} \end{bmatrix} \end{bmatrix}$$

In a similar way, in the XML notation a special element *var* is used to name the sharing point. It supplies a label for the point which can then be referenced elsewhere in the structure, as in the following example:

(18) Structure sharing in XML notation

```
<fs>
  <f name="specifier">
     <fs>
        <f name="agr">
          <var label="n1">
           <fs>
             <f name="number"><symbol value="singular"/></f>
           </fs>
          </var>
        </f>
        <f name="pos"><symbol value="determiner"/></f>
     </fs>
  </f>
  <f name="head">
```

```
      <fs>
        <f name="agr"><var label="n1"/></f>
        <f name="pos"><symbol value="noun"/></f>
      </fs>
    </f>
  </fs>
```

Re-entrancy is symmetric and there is no distinguished representative among the different occurrences of a shared node. In particular, this implies that a value may be attached to any or all occurrences of a shared label:

(19) Specifying shared values

```
<fs>
  <f name="specifier">
     <fs>
        <f name="agr">
          <var label="n1">
            <fs>
              <f name="number"><symbol value="singular"/></f>
            </fs>
          </var>
        </f>
        <f name="pos"><symbol value="determiner"/></f>
     </fs>
  </f>
  <f name="head">
     <fs>
       <f name="agr">
         <var label="n1">
           <fs>
              <f name="number"><symbol value="singular"/></f>
           </fs>
         </var>
       </f>
       <f name="pos"><symbol value="noun"/></f>
     </fs>
  </f>
</fs>
```

A particularly significant case of structure sharing arises when the aim is to express that two features have the same value, even thought that value is not known or is *underspecified*.

4.6 Collections as Complex Feature Values

In feature-based grammar formalisms, such as HPSG and LFG, multivalued features, taking *collections* as their values, are very common. Collections may be organized as lists, sets, or multisets (also known as bags).The items in a list are ordered, and need not be distinct. The items in a set are not ordered, and must be distinct. The items in a bag are neither ordered nor distinct. Sets and bags are thus distinguished from lists in that the order in which the values are specified does not matter for the former, but does matter for the latter, while sets are distinguished from bags and lists in that repetitions of values do not count for the former but do count for the latter.

Collections of any kind can be represented directly in both AVM and XML notations. There is however no theoretical consensus on how they should be represented using graph notation.

4.6.1 Lists as Feature Values

A well-known example of a list-valued feature is the SUBCAT feature in HPSG. It is used to describe the kind of a grammatical subject and objects that a verb expects ("subcategorizes for"). For example, to represent that the English verb form 'gives' as used in structures like 'John gives Mary a kiss' expects a nominative noun phrase as the subject, a dative noun phrase as its indirect object, and an accusative noun phrase as the direct object, and expects these elements in this order, the feature SUBCAT is given the value:

(20) SUBCAT as a list-valued feature

$$\left[\text{SUBCAT}\left\langle\begin{bmatrix}\text{POS } \textit{noun}\\ \text{CASE } \textit{nominative}\end{bmatrix},\begin{bmatrix}\text{POS } \textit{noun}\\ \text{CASE } \textit{dative}\end{bmatrix},\begin{bmatrix}\text{POS } \textit{noun}\\ \text{CASE } \textit{accusative}\end{bmatrix}\right\rangle\right]$$

Here, each of the nouns may carry a label in the form of a boxed integer to indicate structure sharing with the arguments of a predicate that expresses the semantics of a verb.

The AVM representation 20 can be represented in XML by introducing a special element **vcoll** and an attribute **org** whose value is either a list, a set or a multiset, as below:

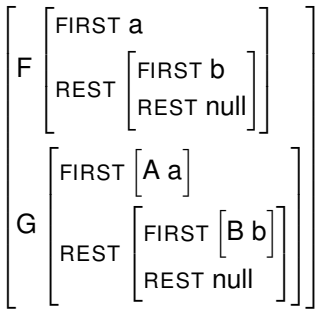
(21) Representing the value of SUBCAT as a list

```
<fs>
  <f name="subcat">
    <vcoll org="list">
      <fs>
        <f name="pos"><symbol value="noun"></f>
        <f name="case"><symbol value="nominative"></f>
      </fs>
      <fs>
        <f name="pos"><symbol value="noun"></f>
        <f name="case"><symbol value="dative"></f>
      </fs>
      <fs>
        <f name="pos"><symbol value="noun"></f>
        <f name="case"><symbol value="accusative"></f>
      </fs>
    </vcoll>
  </f>
</fs>
```

Note that a list as a feature value may contain either atomic or complex values, as shown above.

Note that list values can also be represented recursively as in this example:

(22) Recursive representation



The recursive interpretation of a list given in 22 can also be represented in XML.

(23) Recursive representation of a list in XML

```
<fs>
  <f name="F">
    <fs>
      <f name="first"><symbol value="a"></f>
      <f name="rest">
        <fs>
          <f name="first"><symbol value="b"></f>
          <f name="rest"><symbol value="null"></f>
        </fs>
      </f>
    </fs>
  </f>
  <f name="G">
    <fs>
      <f name="first">
        <fs>
          <f name="A"><symbol value="a"></f>
        </fs>
      <f name="rest">
        <fs>
          <f name="first">
            <fs>
              <f name="B"><symbol value="b"></f>
            </fs>
          <f name="rest"><symbol value="null"></f>
        </fs>
      </f>
    </fs>
  </f>
</fs>
```

According to this recursive representation, lists as feature values may be viewed as a shorthand notation for representing recursively built complex feature structures.

4.6.2 Sets as Feature Values

Features taking sets as their values are frequently used to represent grammatical features of languages like Japanese, Korean or German, in which word order is free or semi-free. For instance, the subject and the verbal complements in a sentence have no fixed order.[6] Hence, for these languages, the feature COMPS as well as

6) There have been some controversies among practicing linguists concerning the validity of presenting semi-free or free word order as supporting evidence for the introduction of sets as feature values.

the feature ARG-ST may be treated as taking a set, not a list of complements or arguments as its value. For the German equivalent of 'gives', the verb form 'gibt', we would have the following:

(24) Set-valued features

$$\begin{bmatrix} \text{ORTH } \text{'gibt'} \\ \text{POS } \textit{verb} \\ \text{SUBCAT } \left\{ \begin{bmatrix} \text{POS } \textit{noun} \\ \text{CASE } \textit{nominative} \end{bmatrix}, \begin{bmatrix} \text{POS } \textit{noun} \\ \text{CASE } \textit{dative} \end{bmatrix}, \begin{bmatrix} \text{POS } \textit{noun} \\ \text{CASE } \textit{accusative} \end{bmatrix}, \right\} \end{bmatrix}$$

The AVM representation in 24 can be represented in XML in an equivalent manner.

(25) Set-valued features in XML

```
<fs>
  <f name="orth"><string>gibt</string></f>
  <f name="pos"><symbol value="verb></f>
  <f name="subcat">
    <vcoll org="set">
      <fs>
        <f name="pos"><symbol value="noun"></f>
        <f name="case"><symbol value="nominative"></f>
      </fs>
      <fs>
        <f name="pos"><symbol value="noun"></f>
        <f name="case"><symbol value="dative"></f>
      </fs>
      <fs>
        <f name="pos"><symbol value="noun"></f>
        <f name="case"><symbol value="accusative"></f>
      </fs>
    </vcoll>
  </f>
</fs>
```

Unlike lists, sets may not be defined recursively.

4.6.3 Multisets as Feature Values

For completeness, we include the possibility of multiset-valued features, although there are few applications for them. The set-valued feature SLASH in HPSG, for instance, is used for dealing with *wh*-movement and other extraction-like phenomena, where the values of SLASH contains the (properties of the) extracted gaps and these gaps can at times be identical.

For illustration, consider the following example:

(26) Coreferential pronouns
John$_{1=2}$ is an idiot. But he$_1$ thinks he$_2$ is smart.

The two occurrences of the pronouns above are coreferential. A multiset value might be used to represent this coreference as follows:

(27) Coreferential multiset

$$\{. \begin{bmatrix} \text{POS } \textit{pronoun} \\ \text{PERSON } \textit{3rd} \\ \text{NUMBER } \textit{sing} \\ \text{GENDER } \textit{masc} \end{bmatrix}, \begin{bmatrix} \text{POS } \textit{pronoun} \\ \text{PERSON } \textit{3rd} \\ \text{NUMBER } \textit{sing} \\ \text{GENDER } \textit{masc} \end{bmatrix} ...\}$$

An XML representation equivalent to 27 can be given as below:

(28) Representing coreferential multiset in XML

```
<vcoll org="multiset">
  <fs>
    <f name="pos"><symbol value="pronoun"></f>
    <f name="person"><symbol value="3rd"></f>
    <f name="number"><symbol value="singular"></f>
    <f name="gender"><symbol value="masc"></f>
  </fs>
  <fs>
    <f name="pos"><symbol value="pronoun"></f>
    <f name="person"><symbol value="3rd"></f>
    <f name="number"><symbol value="singular"></f>
    <f name="gender"><symbol value="masc"></f>
  </fs>
</vcoll>
```

Unlike lists, multisets cannot be defined recursively.

4.7 Typed Feature Structure

In many recent grammar formalisms, the *typed feature structure* has become a major component of linguistic description.

4.7.1 Types

Elements of a domain can be sorted into classes called *types* in a structured way, based on commonalities of their properties. Usually, one or other of the features of a non-typed feature structure can be identified which groups commonly co-occurring features. Linguistic entities such as *phrase*, *word*, *pos* (part of speech), *noun*, and *verb* which might be treated as features of a non-typed feature structure, could be taken as a type in a typed feature structure.

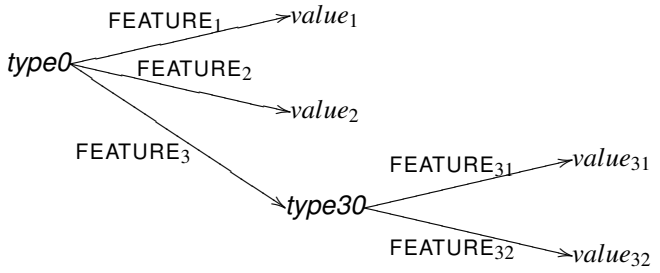
By *typing*, a feature structure it becomes possible to constrain its content. A feature structure of the type *noun*, for instance, would not allow a feature like TENSE in it or a specification of its feature CASE with a value of the type *feminine*.[7)]

4.7.2 Notations

The typing of feature structures can easily be treated in our notations. A graph for a typed feature structure will have the following form:

7) Note that atomic feature values are considered *types*, too.

(29) Typed feature structure in graph notation



The only difference between the typed graph (29) and the non-typed graph (5) is that each of the two nodes has been assigned a type: one is of *type0* and the other of *type30*.

Corresponding to the non-typed AVM (8), here is a typed AVM:

(30) Typed feature structure in AVM notation

$$\begin{bmatrix} \textit{word} \\ \text{ORTH } \text{`love'} \\ \text{SYNTAX} \begin{bmatrix} \textit{verb} \\ \text{VALENCE } \textit{transitive} \end{bmatrix} \end{bmatrix}$$

Here, the entire AVM is assigned the type *word*, whereas the inner AVM is assigned the type *verb*. Unlike the non-typed (8), this typed AVM carries the additional piece of information that features ORTH and SYNTAX are of type *word* and the feature VALENCE is of type *verb*.

The same information can be represented in an XML notation, as below:

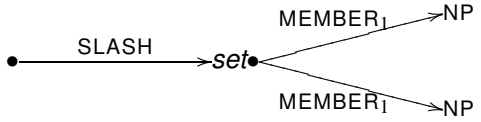
(31) Typed feature structure in XML notation

```
<fs type="word">
    <f name="orth"><string>love</string></f>
    <f name="syntax">
        <fs type="verb">
            <f name="valence"><symbol value="transitive"/></f>
        </fs>
    </f>
</fs>
```

Note here that the line <f name="pos"><sym value="verb"/></f> in the embedded feature structure <fs> has been replaced by typing that <fs> as in <fs type="verb">.

The use of *type* may also increase the expressive power of a graph notation. On the typed graph notation, for instance, multi- values can be represented as terminating nodes branching out of the node labelled with the type *set*, *multiset* or *list*. This node in turn is a terminating node of the arc labelled with a multivalued feature, say SLASH. Each arc branching out of the multivalued node, say *set*, is then labelled with a feature appropriate to the type.

(32) Set-valued feature SLASH in graph notation



The features like MEMBER_1 and MEMBER_2 here should be considered appropriate for the type *set*.

4.8 Subsumption: Relation on Feature Structures

The primary goal in constructing feature structures is thus to capture and represent partial information. No feature structure is expected to represent the total information describing all possible worlds or states of affairs. It may be of greater interest and value to focus on particular aspects of interesting situations and capture various sorts of related information. The relation of *subsumption* on feature structures is thus introduced to be able to tell which feature structure carries more information than the others.

Some feature structures carry less information than others. The extreme case, perhaps the most uninteresting case, is the *empty* feature structure [ ] sometimes called *variable* that carries no information at all. For more interesting cases, consider the following two feature structures:

(33) a. $\begin{bmatrix} \textit{word\_st} \\ \text{ORTH } \text{‘loves’} \end{bmatrix}$

b. $\begin{bmatrix} \textit{word\_st} \\ \text{ORTH } \text{‘loves’} \\ \text{SYN } \begin{bmatrix} \textit{verb\_st} \\ \text{FORM } \textit{finite} \end{bmatrix} \end{bmatrix}$

The feature structure (a) says that the word is a string consisting of 5 letters spelled as ‘l-o-v-e-s’ and that’s all. But the feature structure (b) says more than that by providing the additional information that it is a finite verb. Hence, (a) is said to be less informative than (b).

To describe such a relation among some feature structures, a technical term is introduced that is called *subsumption*. In the above case, (a) is said to subsume (b).

4.8.1 Definition

Intuitively speaking, a feature structure A subsumes a feature structure B if A is not more informative than B, thus subsuming all feature structures that are at least equally informative as itself. Since it carries no information, the empty feature structure [ ] *subsumes* not only the feature structures (a) and (b), but also any other feature structures including itself. More strictly speaking, the subsumption relation is a partial ordering over feature structures and is defined as follows:[8)]

(34) Definition of Subsumption
Given two typed feature structures, FS_1 and FS_2, FS_1 is said to subsume FS_2, written as $A \sqsubseteq B$ if and only if the following conditions hold:

A. Path values Every path **P** which exists in FS_1 with a value of type **t** also exists in FS_2 with a value which is either **t** or one of its subtypes.

B. Path equivalences Every two paths which are shared in FS_1 are also shared in FS_2.

C. Type ordering Every type assigned by FS_1 to a path subsumes the type assigned to the same path in FS_2 in the type ordering.

Each of the three conditions A, B, and C can be illustrated as below:

8) Carpenter (1992: 43) claims that the subsumption relation is a pre-ordering on the collection of feature structures. It is transitive and reflexive, but not anti-symmetric because it is possible to have two distinct feature structures that mutually subsume each other. But these are alphabetic variants.

4.8.2 Condition A on Path Values

(35) Example satisfying Condition A

$$A\begin{bmatrix}\textit{verb\_st}\\ \text{AGR}\begin{bmatrix}\textit{agr-cat}\\ \text{NUMBER } \textit{singular}\end{bmatrix}\end{bmatrix} \sqsubseteq B\begin{bmatrix}\textit{verb\_st}\\ \text{AGR}\begin{bmatrix}\textit{agr-cat}\\ \text{PERSON } \textit{3rd}\\ \text{NUMBER } \textit{singular}\end{bmatrix}\\ \text{TENSE present}\end{bmatrix}$$

There is only one path in A: <AGR.NUMBER>. This path exists in B and their values are the same.[9] Hence, Condition A is satisfied. Condition B is inapplicable here, since there is no structure sharing in either of the two feature structures. Condition C is satisfied because every type assigned by A to a path is identical with the type assigned to the same path in B. Hence, A subsumes B.

4.8.3 Condition B on Structure Sharing

(36) Case satisfying Condition B

$$A\begin{bmatrix}\textit{syn-cat}\\ \text{VALENCE}\begin{bmatrix}\textit{val-cat}\\ \text{SPECIFIER} <\boxed{1}\ \text{NP}>\\ \text{COMPS} <>\end{bmatrix}\\ \text{ARG-ST} <\boxed{1}>\end{bmatrix} \sqsubseteq B\begin{bmatrix}\textit{syn-cat}\\ \text{ORTH `walks'}\\ \text{HEAD}\begin{bmatrix}\textit{verb\_st}\\ \text{FORM finite}\end{bmatrix}\\ \text{VALENCE}\begin{bmatrix}\textit{val-cat}\\ \text{SPECIFIER} <\boxed{1}\ \text{NP}>\\ \text{COMPS} <>\end{bmatrix}\\ \text{ARG-ST} <\boxed{1}>\end{bmatrix}$$

This example looks a bit complicated. But one can easily check that the structure sharing tagged by $\boxed{1}$ in A also exists in B and the other two conditions are also satisfied. Hence, this subsumption relation holds here.

Consider the following pair of examples related to the structure sharing condition:

(37) Another case involving structure sharing

$$C\begin{bmatrix}\textit{word}\\ \text{SYN}\begin{bmatrix}\textit{syn-cat}\\ \text{HEAD}\begin{bmatrix}\textit{noun\_st}\\ \text{AGR}\begin{bmatrix}\textit{agr-cat}\\ \text{PERSON } \textit{3rd}\end{bmatrix}\end{bmatrix}\\ \text{VALENCE}\begin{bmatrix}\textit{val-cat}\\ \text{SPECIFIER} <\begin{bmatrix}\textit{noun}\\ \text{AGR}\begin{bmatrix}\textit{agr-cat}\\ \text{PERSON } \textit{3rd}\end{bmatrix}\end{bmatrix}>\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

$\sqsubseteq$

9) The labels like A and B in the lower left corner of each AVM here is not part of the feature structure being represented itself. They are there merely to refer to each AVM for the convenience of our present discussions only.

$$D\begin{bmatrix}\textit{word\_st}\\ \text{SYN}\begin{bmatrix}\textit{syn-cat}\\ \text{HEAD}\begin{bmatrix}\textit{noun\_st}\\ \text{AGR}\ \boxed{1}\begin{bmatrix}\textit{agr-cat}\\ \text{PERSON}\ \textit{3rd}\end{bmatrix}\end{bmatrix}\\ \text{VALENCE}\begin{bmatrix}\textit{val-cat}\\ \text{SPECIFIER}<\begin{bmatrix}\textit{noun}\\ \text{AGR}\ \boxed{1}\end{bmatrix}>\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

Here, C subsumes D because it satisfies Condition A and C, while Condition B is not relevant. On the other hand, D does not subsume C because Condition B applies here and is violated.

4.8.4 Condition C on Type Ordering

This condition applies only to typed feature structures under the assumption of some kind of type inheritance hierarchy assumed. Pronouns, proper nouns, and common nouns are subtypes of the supertype noun. Hence, all these subtypes share some properties of each being a noun. Thus, the following is a simple example of subsumption:

(38) Case involving type ordering

$$A\begin{bmatrix}\textit{noun\_st}\\ \text{ORTH}\ \text{'Mary'}\\ \text{AGR}\begin{bmatrix}\text{PERSON}\ \textit{3rd}\\ \text{NUMBER}\ \textit{singular}\end{bmatrix}\end{bmatrix} \sqsubseteq B\begin{bmatrix}\textit{name\_st}\\ \text{ORTH:}\ \text{'Mary'}\\ \text{AGR:}\begin{bmatrix}\text{PERSON}\ \textit{3rd}\\ \text{NUMBER}\ \textit{singular}\\ \text{GENDER}\ \textit{feminine}\end{bmatrix}\end{bmatrix}$$

where $\textit{noun} \sqsubseteq \textit{name}$

Since the type *noun_st* of A is a supertype of the type *name* of B, A subsumes B. Furthermore, B has an extra piece of information about the gender. Hence, A properly subsumes B.

4.9 Operations on Feature Structures and Feature Values

As a corollary to subsumption, unification is the main topic of this section. Feature structures are generally unified to augment the amount of information content. Generalization is its dual: two feature structures are generalized to select identical feature specifications and put them into one general feature structure. In addition to these operations on feature structures, some operations on feature values or types like the concatenation $\oplus$ of list values, alternative feature values and conjunctive types will also be topics of this section.

4.9.1 Compatibility

Some feature structures are *compatible* with some others, while there are conflicting cases. Consider the following three AVM's:[10)]

(39) a. $$A\begin{bmatrix}\textit{noun\_st}\\ \text{AGR}\begin{bmatrix}\text{PERSON}\ \textit{3rd}\end{bmatrix}\end{bmatrix}$$

10) Type labels like *syn-cat*, *val-cat* and *agr-cat* are not very informative, so they will be omitted from now on.

b. B
$$\begin{bmatrix} \textit{noun\_st} \\ \text{AGR} \begin{bmatrix} \text{NUM } \textit{singular} \\ \text{GENDER } \textit{feminine} \end{bmatrix} \end{bmatrix}$$

c. C
$$\begin{bmatrix} \textit{noun\_st} \\ \text{AGR} \begin{bmatrix} \text{PERSON } \textit{3rd} \\ \text{GENDER } \textit{masculine} \end{bmatrix} \end{bmatrix}$$

The feature structure A is compatible with B and also with C. But the feature structures B and C are incompatible because their specification of the feature *gender* is conflicting: one has the value *feminine* and the other the value *masculine*.

Incompatibility may also arise when there is a type difference, as shown below:

(40) Incompatible feature structures

a. E
$$\begin{bmatrix} \textit{noun\_st} \\ \text{AGR} \begin{bmatrix} \text{PERSON } \textit{3rd} \\ \text{NUMBER singular} \end{bmatrix} \end{bmatrix}$$

b. F
$$\begin{bmatrix} \textit{verb\_st} \\ \text{AGR} \begin{bmatrix} \text{PERSON } \textit{3rd} \\ \text{NUMBER } \textit{singular} \end{bmatrix} \end{bmatrix}$$

The feature structures E and F both have the same agreement feature specifications, but these two feature structures are incompatible because they belong to two different types which are not in a subsumption relation.

4.9.2 Unification

Compatible feature structures often represent different aspects of information from different sources. Merged together, they may convey a more coherent picture of information. This process of information merge is captured by the operational process of unifying two compatible feature structures, FS_1 and FS_2, represented $FS_1 \sqcup FS_2$. Compatible feature structures can be unified together to form a more (or at least equally) informative feature structure. The unification of two typed feature structures FS_1 and FS_2 is the most informative typed feature structure which is subsumed by both FS_1 and FS_2, if it exists.[11)]

(41) Formal definition of unification
The unification of $F_1 \sqcup F_2$ of two typed feature structures F_1 and F_2 is the least upper bound of F_1 and F_2 in the collection of typed feature structures ordered by subsumption.

It should be noted here that the type hierarchy is constructed with the most non-specific or the least informative type at the bottom, which is represented as $\bot$.[12)]

The feature structure A, for instance, can be *unified* with C, yielding a little bit more enriched feature structure D.

11) Formally speaking, the unification of two incompatible feature structures results in inconsistency and may just be represented with some inconsistency symbol like $\top$ without calling into the procedural aspect of its failure. With this symbol, unification may be treated as a *total* operation that always yields a result even with the *inconsistency* $\top$.

12) If the type hierarchy is depicted with the most general or non-specific type at the top $\top$, then the unification of two typed feature structures FS_1 and FS_2 must be defined as the greatest lower bound of FS_1 and FS_2 in the type hierarchy and should also be represented as $FS_1 \sqcap FS_2$ as in Copestake (2002).

(42) Unified feature structure

$$D\begin{bmatrix} \textit{noun\_st} \\ \text{AGR} \begin{bmatrix} \text{NUMER } \textit{singular} \\ \text{PERSON } \textit{3rd} \\ \text{GENDER } \textit{masculine} \end{bmatrix} \end{bmatrix}$$

Unification normally adds information as illustrated just now. But identical features unify without adding any further information. The empty feature structure may be unified with every feature structure without changing the content of the latter, thus formally treated as the *identity element* of unification.

(43) Basic properties of unification

- — Unification adds information.
- — Unification is idempotent: the unification of identical feature structures remains the same without anything added.
- — The empty structure is the identity element for unification: it adds nothing to the resulting feature structure.

Unification may thus be considered an analogue of the set-theoretic *union*. Hence, they are usually represented in the same symbol, $\sqcup$.

4.9.3 Unification of Shared Structures

The operation of unification gets complicated when it involves shared structures. Consider the following example:

(44) Unification involving re-entrancy

$$G\begin{bmatrix} \textit{verb\_st} \\ \text{AGR } \boxed{1} \\ \text{SPECIFIER} < \begin{bmatrix} \textit{noun\_st} \\ \text{AGR} \boxed{1} \end{bmatrix} > \end{bmatrix} \sqcup H\begin{bmatrix} \textit{verb\_st} \\ \text{AGR } \begin{bmatrix} \text{PERSON } \textit{3rd} \end{bmatrix} \end{bmatrix}$$

$$= I\begin{bmatrix} \textit{verb\_st} \\ \text{AGR } \boxed{1}\begin{bmatrix} \text{PERSON } \textit{3rd} \end{bmatrix} \\ \text{SPECIFIER} < \begin{bmatrix} \textit{noun\_st} \\ \text{AGR} \boxed{1} \end{bmatrix} > \end{bmatrix}$$

The unification of feature structures G and H results in a feature structure I. This unification involves structure sharing. Here, the AGREE value of H in the first path is simply unified with the AGREE value of G occurring in the first path which is tagged with $\boxed{1}$. Furthermore, on the assumption that the type *verb* is a subtype of the type *agr-pos*, the resulting feature structure I becomes of the type *verb*.

Consider a bit more complicated case:

(45) Case for pumping values to each other

$$G\begin{bmatrix} \textit{verb\_st} \\ \text{AGR } \boxed{1} \\ \text{SPECIFIER} < \begin{bmatrix} \textit{noun\_st} \\ \text{AGR} \boxed{1} \end{bmatrix} > \end{bmatrix} \sqcup J\begin{bmatrix} \textit{agr-pos} \\ \text{AGR} \begin{bmatrix} \text{PERSON } \textit{3rd} \end{bmatrix} \\ \text{SPECIFIER} < \begin{bmatrix} \textit{noun\_st} \\ \text{AGR} \begin{bmatrix} \text{NUMBER } \textit{singular} \end{bmatrix} \end{bmatrix} > \end{bmatrix}$$

$$= K\begin{bmatrix} \textit{verb\_st} \\ \text{AGR } \boxed{1} \begin{bmatrix} \text{PERSON } \textit{3rd} \\ \text{NUMBER } \textit{singular} \end{bmatrix} \\ \text{SPECIFIER} < \begin{bmatrix} \textit{noun\_st} \\ \text{AGR} \boxed{1} \end{bmatrix} > \end{bmatrix}$$

As before, the two occurrences of the path AGR in G and J are unified, yielding the first path as in K. Another pair of the two occurrences of the path SPECIFIER.AGR are also unified. As a result, this path in K shares the value with the path AGR as is marked with $\boxed{1}$ and then its original value can be transferred to the place where the index $\boxed{1}$ first has occurred, namely in the path AGR.

4.10 Operations on Feature Values and Types

4.10.1 Concatenation and Union operations

A list as a feature value may be appended to anther list by *concatenation* $\oplus$. In HPSG, the value of argument structure ARG-ST is treated as the concatenation of the SPECIFIER and COMPS value lists.

(46) Concatenated list

$$\begin{bmatrix} \textit{verb\_st} \\ \text{VALENCE} \begin{bmatrix} \text{SPECIFIER } \boxed{A} \\ \text{COMPS } \boxed{B} \end{bmatrix} \\ \text{ARG\_ST } \boxed{A} \oplus \boxed{B} \end{bmatrix}$$

Here, variable lists are tagged with boxed uppercase Latin characters. Hence, the value of ARG_ST should be understood as the concatenation of two lists.

The similar operation of combining the members of two sets or multisets is known as union. These operations are theoretically possible, but it is hard to find some examples in linguistic descriptions.

In the XML notation, we use the same element to represent both concatenation and union:

The example 46 can be represented in XML with a special purpose element **vcoll**.

(47) Representing a list of values in XML

```
<fs type="verb_st>
   <f name="valence>
      <fs>
        <f name="specifier">
          <var label="nA"></var>
```

```
        </f>
        <f name="comps">
          <var label="nB"></var>
        </f>
      </fs>
   </f>
   <f name="arg_st">
     <vcoll org="list">
       <var label="nA"></var>
       <var label="nB"></var>
     </vcoll>
   </f>
</fs>
```

4.10.2 Alternation

Some features may take an alternative value.[13] For example, the word 'Sandy' can be either feminine or masculine, but not neuter. Hence, the value of its gender can be specified with the vertical bar '|' as follows.

(48) Alternative value

$$\begin{bmatrix} \textit{word} \\ \text{ORTH } \text{'Sandy'} \\ \text{GENDER } \textit{feminine} \mid \textit{masculine} \end{bmatrix}$$

Many cases that require an alternative feature value can be handled just by underspecifying or omitting feature specifications. But in the above example, it is not possible. If there were only two possible values for the gender in English, one could have just skipped specifying the value of the gender. Since there is a third value, namely *neuter*, for the English gender, one cannot but specify the value, which is still an alternative value.

Alternation can also be systematically represented by XML by introducing a special purpose element <**vAlt**>.

(49) Representing Alternative value in XML

```
<fs type="word">
   <f name="orth">
   <string>Sandy</string>
   </f>
   <f name="gender">
      <vAlt>
      <symbol value="feminine"></symbol>
      <symbol value="masculine"></symbol>
      </vAlt>
   </f>
</fs>
```

The curly brackets '{ }' are often used to represent the alternation of feature values, especially when it is in the form of a full-blown feature structure. The following is an abstract and yet simple example:

13) Strictly speaking, this alternation differs from the Boolean disjunction, or the inclusive disjunction, often represented by a wedge $\vee$, which results in truth if and only if one of its disjuncts is true. Alternation here, however, allows the choice of only one value. Hence, the mathematical characterization of feature structure as a function still remains valid. The same is true with those cases in which features have collections like lists, sets or multisets as values, for they are not allowed to have more than one multi-values as their values.

(50) Alternative feature structures as feature value

$$\begin{bmatrix} \text{A a} \\ \text{B} \left\{ \begin{bmatrix} \text{C c} \end{bmatrix}, \begin{bmatrix} \text{D d} \\ \text{E e} \end{bmatrix} \right\} \end{bmatrix}$$

Here the value of the feature B is specified with an alternative value, which consists of two feature structures. Set feature values are also represented by the pair of curly brackets. Hence, the use of curly brackets for representing alternation may be confusing. So the use of curly brackets for alternation is better avoided.

4.10.3 Negation

The notion of negation is either truth-functional or set-theoretic. In Boolean bivalent logic, negation is a truth function that returns the opposite value to a value: it assigns truth to falsity and vice versa. According to its set-theoretic notion, on the other hand, the negation of a value which is a member of some set *A* is understood as being some value in the complement $_{\overline{A}}$ of that value set.

In the current standard, negation applies only to the value of a feature in a feature structure and is understood more or less in a set-theoretic sense. If the value of a feature is atomic, then its negation is a member in the complement of the admissible value set to which that value belongs. suppose the value set has only two members. Then, negation picks up the value opposite to a given value. For example, if the feature *number* is *singular* or *plural*, then the negation of *singular* is *plural* and the negation of *plural singular*. But suppose the admissible value set of the feature *Gender* has more than two values: *masculine, feminine* and *neuter*. Then, the negation of the value *masculine* is one of the other two values in the complement set {feminine, neuter}, namely either *feminine* or *neuter*.

The value of a feature can, however, be complex. It can be a feature structure. In this case, the negation of a feature structure, say *F*, which is the value of some feature, is any of the feature structures that are incompatible with that feature structure *F*. Given a feature structure that has a single feature specification '$FEATURE0 : value0$, then its negation is a set of feature structures each of which contains at least one feature specification such that '$FEATURE0 : \neg value0)$'.

Here is an illustration:

(51) Negation of a Complex Feature Value

$$\begin{bmatrix} \text{noun} \\ \text{AGR: } \neg \begin{bmatrix} \text{Person: third} \\ \text{Number: singular} \end{bmatrix} \end{bmatrix}$$

This feature structure carries the information about some noun that has an agreement feature specification other than that of a third person singular noun. Such a noun in English, for instance, does not have to undergo the Subject-Verb agreement as in 'Mia snores' opposed to the ill-formed sentence 'Mia snore'. Suppose we have:

(52) Incompatible Feature Structures

a. $$\begin{bmatrix} \text{noun} \\ \text{AGR: } \begin{bmatrix} \text{Person: third} \\ \text{Number: singular} \end{bmatrix} \end{bmatrix}$$

b. $\begin{bmatrix} \text{noun} \\ \text{AGR: } \begin{bmatrix} \text{Person: first} \\ \text{Number: singular} \end{bmatrix} \end{bmatrix}$

These two are incompatible and (a) can be considered to be a particular instance of the negation of the AGR value of (b).

Negation can also be represented by XML with a special element <**vNot**>. Here is an example:

(53) Representing negation in XML

```
<fs type="noun">
  <f name="ARG">
    <vNot>
       <fs>
         <f name="Person">
         <symbol value="third"></symbol>
         </f>
         <f name="Number">
         <symbol value="singular"></symbol>
         </f>
       </fs>
    </vNot>
  </f>
</fs>
```

This representation is equivalent to the AVM representation (51) just given above.

4.11 Informal Semantics of Feature Structure

Semantically, a feature structure is a partial characterization of an object. It is a specification of a number of properties, where a property is described by specifying the value of a feature. For instance, an object is (partly) characterized as a plural noun by specifying that the feature POS (part-of-speech) has the value *noun* and the value NUM (number) has the value *plural*.

A common way to express this mathematically is by using the so-called *lambda operator*, which is a device for defining functions. For example, the meaning of the feature structure

(54) $\begin{bmatrix} \text{POS noun} \\ \text{NUM plural} \end{bmatrix}$

can be expressed using the lambda-operator as:

(55) $\lambda x : pos(x) = noun \wedge num(x) = plural$

The part of the expression that follows the initial '$\lambda x :$' part, is called the *body* of the expression. Formally, such an expression defines a function that is applicable to the kind of object that x stands for, and that delivers the value that is obtained by substituting a specific argument for the variable x everywhere in the body and evaluating the body. For instance, applied to the argument *house*, the expression $\lambda x : pos(x) = noun \wedge num(x) = sing$ delivers the value false, since $pos(house) = noun$ but $num(x) \neq plural$, hence the conjunction is false.

A functions which has true and false as its values, like the function defined by (55), is in fact a one-place *predicate*: if its application to an object yields true, the object in question has the property expressed by the predicate; if it yields false, it doesn't. The lambda-expression (55) represents a complex predicate formed by conjoining the predicate of having part-of-speech *noun* with the predicate of having number *plural*. Lambda-expressions are commonly used in mathematics, in formal logic, and in natural language semantics, and have a well-established model-theoretic interpretation; hence a translation of feature structures into lambda-expressions can be used as a semantics of feature structures. For a formal account of the semantics of feature structures see Annex C.

Generally speaking, the meaning of a single feature specification [A: v] is the simple predicate (or 'property') of having the value v for the feature A, and the meaning of a (more complex) feature structure is the complex predicate formed by conjoining the simple predicates corresponding to the feature specifications in the FS, taking into account feature nestings and the negations, alternations, and collections of values that may occur in the feature values. Example (56), involving alternation, negation and a set-valued feature, illustrates this. This feature structure partly describes a phrase of which the head daughter is a noun of which the number feature is either *singular* or *mass*, which is not *inanimate*, and which takes prepositional objects using the preposition *of* or *by*. This is formally expressed by the lambda-expression (57), where 'hdtr' stands for 'head daughter'. Note that, since alternation in a feature value specification expresses a choice between mutually exclusive alternatives, the corresponding predicate contains an 'exclusive or', denoted by the symbol $\underline{\vee}$. (An exclusive or construction, like $p \underline{\vee} q$, can be seen as an abbreviation of $(p \vee q) \wedge \neg(p \wedge q)$.)

(56)
$$\begin{bmatrix} \text{POS noun} \\ \text{HDTR} \begin{bmatrix} \text{POS noun} \\ \text{NUMBER (singular}|\text{mass)} \\ \text{GENDER } \textit{not}\text{(inanimate)} \\ \text{PREPOBS} \left\{ \textit{of, by} \right\} \end{bmatrix} \end{bmatrix}$$

(57) $\lambda x : pos(x) = noun \wedge pos(hdtr(x)) = noun$
$\wedge (number(hdtr(x)) = singular \underline{\vee} number(hdtr(x)) = plural)$
$\wedge \neg(gender(hdtr(x)) = neuter) \wedge prepobs(hdtr(x)) = \{\text{“}of\text{”}, \text{“}by\text{”}\}$

A complication in describing the meaning of feature structures is formed by the phenomenon of re-entrancy. Re-entrancy can take three different forms, differing in the amount of information in the values of the features that share their values. Let a FS contain two features A and B that share their values. (The case of more than two features sharing their values is an easy generalization of this.) The following cases may arise:

a) neither A nor B has a value specified;

b) A has no value specified; B has a value specified;

c) A and B both have a value specified.

The first case is illustrated in its simplest possible form by (58):

(58)
$$\begin{bmatrix} \text{A } \boxed{1} \\ \text{B } \boxed{1} \end{bmatrix}$$

This says that the features A and B have the same value, whatever that value may be (or may become later in the course of a process in which the feature structure is involved). Hence it corresponds to the predicate (59):

(59) $\lambda x : A(x) = B(x)$

The second case corresponds in its simplest possible form to a feature structure like (60):

(60) $\begin{bmatrix} \text{A } \boxed{1} \\ \text{B } \boxed{1}\ \text{v} \end{bmatrix}$

Again, the structure sharing means that the features involved should have the same value, so the predicate corresponding to the FS again contains the stipulation (59). Moreover, the FS also says that B has the value v; hence altogether this FS corresponds to the predicate (61):

(61) $\lambda x : A(x) = B(x) \wedge B(x) = v$

Note that, since it is a logical consequence of $A(x) = B(x)$ and $B(x) = v$ that $A(x) = v$, the latter condition does not need to be included in (61).

The case of one of the shared feature values being specified and the other unspecified, as illustrated by (60), is in fact a special case of the more general third case, which takes the simplest possible form (62):

(62) $\begin{bmatrix} \text{A } \boxed{1}\ \text{v1} \\ \text{B } \boxed{1}\ \text{v2} \end{bmatrix}$

If $v1$ and $v2$ in this FS are different atomic values, then the FS expresses an impossible situation and is meaningless. But if $v1$ and $v2$ are complex values, than something meaningful may be expressed if these values are compatible. This is in particular the case if $v1$ and $v2$ are feature structures that can be unified. Other possible cases arise through the use of negation or alternation, since the use of these operators corresponds to not really specifying a value of a feature, but specifying *contraints* on the values of the feature. For instance, suppose $v1 = a$ and $v2 = a|b$, where a and b are atomic values; in that case the shared value is in fact a. Similarly if $v1 = a$ and $v2 = \neg b$.

Structure sharing can of course be combined with the use of complex values. This is illustrated in example (63), where sharing occurs with a complex value that includes an alternation.

(63) $\begin{bmatrix} \text{SPEC} & \begin{bmatrix} \text{POS det} \\ \text{AGR } \boxed{1} \left[\text{GENDER (feminine|masculine)}\right] \end{bmatrix} \\ \text{HEAD} & \begin{bmatrix} \text{POS noun} \\ \text{AGR } \boxed{1} \left[\text{NUMBER plural}\right] \end{bmatrix} \end{bmatrix}$

Semantically, this feature structure characterises those phrases consisting of a determiner which has feminine or masculine gender and a plural noun, as expressed by the predicate (64):

(64) $\lambda x : pos(spec(x)) = det$
$\wedge\ agr(spec(x)) = agr(head(x))$
$\wedge\ (gender(agr(spec(x))) = feminine \veebar$
$(gender(agr(spec(x))) = masculine)$
$\wedge\ pos(head(x)) = noun$
$\wedge\ num(agr(spec(x))) = plural$

5 XML-Representation of Feature Structures

This section describes a standard for the representation of feature structures using XML, the eXtensible Markup Language, which has been officially recommended by the World Wide Web Consortium W3C as a document-processing standard for interchanging data especially over the Internet. XML provides a rich, well-defined and platform-independent markup language for all varieties of electronic document.

5.1 Overview

This section is organized as follows. Following this overview, section 5.2 Elementary Feature Structures and the Binary Feature Value introduces the elements <fs> and <f>, used to represent feature structures and features respectively, together with the elementary binary feature value.

Section 5.3 Other Atomic Feature Values introduces elements for representing other kinds of atomic feature values such as symbolic, numeric, and string values.

Section 5.4 Feature and Feature-Value Libraries introduces the notion of predefined libraries or groups of features or feature values along with methods for referencing their components.

Section 5.5 Feature Structures as Complex Feature Values introduces complex values, in particular feature-structures as values, thus enabling feature structures to be recursively defined. Section 5.6 Re-entrant Feature Structures treats structure sharing in feature structures.

Section 5.7 Collections as Complex Feature Values discusses other complex values, in particular values which are collections, organized as sets, bags, and lists.

Section 5.8 Feature Value Expressions discusses how the operations of alternation, negation, and collection of feature values may be represented.

Section 5.9 Default Values discusses ways of representing underspecified, default, or uncertain values.

Section 5.10 Linking Text and Analysis discusses how analyses may be linked to other parts of an encoded text.

Formal definitions for all the elements introduced in this section are provided in Annex A Formal Definition and Implementation of Feature Structures in XLM.

5.2 Elementary Feature Structures and the Binary Feature Value

The fundamental elements used to represent a feature structure analysis are <f> (for feature), which represents a feature-value pair, and <fs> (for feature structure), which represents a structure made up of such feature-value pairs. The <fs> element has an optional *type* attribute which may be used to represent typed feature structures, and may contain any number of <f> elements. An <f> element has a required *name* attribute and an associated value. The value may be simple: that is, a single binary, numeric, symbolic (i.e. taken from a restricted set of legal values), or string value, or a collection of such values, organized in various ways, for example, as a list; or it may be complex, that is, it may itself be a feature structure, thus providing a degree of recursion. Values may be under-specified or defaulted in various ways. These possibilities are all described in more detail in this and the following sections.

Feature and feature-value representations (including feature structure representations) may be embedded directly at any point in an XML document, or they may be collected together in special-purpose feature or feature-value libraries. The components of such libraries may then be referenced from other feature or feature-value representations, using the feats or fVal attribute as appropriate.

We begin by considering the simple case of a feature structure which contains binary-valued features only. The following three XML elements are needed to represent such a feature structure:

(65)

— **<fs>** represents a feature structure, that is, a collection of feature-value pairs organized as a structural unit. Selected attributes:

  type specifies the type of the feature structure.

  feats references the feature-value specifications making up this feature structure.

— **<f>** represents a feature value specification, that is, the association of a name with a value of any of several different types. Selected attributes:

  name provides a name for the feature.

  fVal references any element which can be used to represent the value of a feature.

— **<binary>** represents the value part of a feature-value specification which can contain either of exactly two possible values.

  value supplies a binary value (true or false, plus or minus).

The attributes *feats* and the *fVal* are not discussed in this section: they provide an alternative way of indicating the content of an element, as further discussed in section 5.4 Feature and Feature-Value Libraries.

An `<fs>` element containing `<f>` elements with binary values can be straightforwardly used to encode the matrices of feature-value specifications for phonetic segments, such as the following for the English segment [s].[14)]

(66)

$$\begin{bmatrix} \text{consonantal} + \\ \text{vocalic} - \\ \text{voiced} - \\ \text{anterior} + \\ \text{coronal} + \\ \text{continuant} + \\ \text{strident} + \end{bmatrix}$$

This representation may be encoded in XML as follows:

(67)

```
<fs type="phonological segments">
 <f name="consonantal">
  <binary value="true"/>
 </f>
 <f name="vocalic">
  <binary value="false"/>
 </f>
 <f name="voiced">
  <binary value="false"/>
 </f>
 <f name="anterior">
  <binary value="true"/>
 </f>
 <f name="coronal">
```

14) Adapted from Noam Chomsky and Morris Halle (1968: 415).

```
  <binary value="true"/>
 </f>
 <f name="continuant">
  <binary value="true"/>
 </f>
 <f name="strident">
  <binary value="true"/>
 </f>
</fs>
```

Note that <fs> elements may have an optional *type* attribute to indicate the kind of feature structure in question, whereas <f> elements must have a *name* attribute to indicate the name of the feature. Feature structures need not be typed, but features must be named. Similarly, the <fs> element may be empty, but the <f> element must have (or reference) some content.

The restriction of specific features to specific types of values (e.g. the restriction of the feature “strident"to a binary value) requires additional validation, as does any restriction on the features available within a feature structure of a particular type (e.g. whether a feature structure of type “phonological segment" necessarily contains a feature “voiced". Such validation may be carried out at the document level, using special purpose processing, at the schema level using additional validation rules, or at the declarative level, using an additional mechanism such as the feature-system declaration discussed in chapter FD of the TEI Guidelines.

Although we have used the term binary for this kind of value, and its representation in XML uses values such as `true` and `false` (or, equivalently, `1` and `0`), it should be noted that such values are not restricted to propositional assertions. As this example shows, this kind of value is intended for use with any binary-valued feature.

Formal declarations for the <fs>, <f> and <binary> elements are provided below in Annex A Formal Definition and Implementation.

5.3 Other Atomic Feature Values

Features may take other kinds of atomic value. In this section, we define elements which may be used to represent: symbolic values, numeric values, and string values. The module defined by this chapter allows for the specification of additional datatypes if necessary, by extending the underlying class class.singleValue. If this is done, it is recommended that only the basic W3C datatypes should be used; more complex datatyping should be represented as feature structures.

(68)

— **<symbol>** represents the value part of a feature-value specification which contains one of a finite list of symbols. Selected attributes:

  value supplies the symbolic value for the feature, one of a finite list that may be specified in a feature declaration.

— **<numeric>** represents the value part of a feature-value specification which contains a numeric value or range.

  value supplies a lower bound for the numeric value represented, and also (if max is not supplied) its upper bound.

  max supplies an upper bound for the numeric value represented.

  trunc specifies whether the value represented should be truncated to give an integer value.

— **<string>** represents the value part of a feature-value specification which contains a string.

  * No attributes other than those globally available (see definition for tei.global.attributes)

The <symbol> element is used for the value of a feature when that feature can have any of a small, finite set of possible values, representable as character strings. For example, the following might be used to represent the claim that the Latin noun form “mensas"(tables) has accusative case, feminine gender and plural number:

(69)

```
<fs>
 <f name="case">
  <symbol value="accusative"/>
 </f>
 <f name="gender">
  <symbol value="feminine"/>
 </f>
 <f name="number">
  <symbol value="plural"/>
 </f>
</fs>
```

More formally, this representation shows a structure in which three features (case, gender and number) are used to define morpho-syntactic properties of a word. Each of these features can take one of a small number of values (for example, case can be nominative, genitive, dative, accusative etc.) and it is therefore appropriate to represent the values taken in this instance as <symbol> elements. Note that, instead of using a symbolic value for grammatical number, one could have named the feature singular or plural and given it an appropriate binary value, as in the following example:

(70)

```
<fs>
 <f name="case">
  <symbol value="accusative"/>
 </f>
 <f name="gender">
  <symbol value="feminine"/>
 </f>
 <f name="singular">
  <binary value="false"/>
 </f>
</fs>
```

Whether one uses a binary or symbolic value in situations like this is largely a matter of taste.

The <string> element is used for the value of a feature when that value is a string drawn from a very large or potentially unbounded set of possible strings of characters, so that it would be impractical or impossible to use the <symbol> element. The string value is expressed as the content of the <string> element, rather than as an attribute value. For example, one might encode a street address as follows:

(71)

```
<fs>
 <f name="address">
  <string>3418 East Third Street</string>
 </f>
</fs>
```

The <numeric> element is used when the value of a feature is a numeric value, or a range of such values. For example, one might wish to regard the house number and the street name as different features, using an encoding like the following:

(72)

```
<fs>
 <f name="houseNumber">
  <numeric value="3418"/>
 </f>
 <f name="streetName">
  <string>East Third Street</string>
 </f>
</fs>
```

If the numeric value to be represented falls within a specific range (for example an address that spans several numbers), the *max* attribute may be used to supply an upper limit:

(73)

```
<fs>
 <f name="houseNumber">
  <numeric value="3418" max="3440"/>
 </f>
 <f name="streetName">
  <string>East Third Street</string>
 </f>
</fs>
```

It is also possible to specify that the numeric value (or values) represented should (or should not) be truncated. For example, assuming that the daily rainfall in mm is a feature of interest for some address, one might represent this by an encoding like the following:

(74)

```
<fs>
 <f name="dailyRainFall">
  <numeric value="0.0" max="1.3" trunc="no"/>
 </f>
</fs>
```

This represents any of the infinite number of numeric values falling between 0 and 1.3; by contrast

(75)

```
<fs>
 <f name="dailyRainFall">
  <numeric value="0.0" max="1.3" trunc="yes"/>
 </f>
</fs>
```

represents only two possible values: 0 and 1.

As noted above, additional processing is necessary to ensure that appropriate values are supplied for particular features, for example to ensure that the feature `singular` is not given a value such as `<symbol value="feminine"/>`. There are two ways of attempting to ensure that only certain combinations of feature names and values are used. First, if the total number of legal combinations is relatively small, one can predefine all of them in a construct known as a feature library, and then reference the combination required using the *feats* attribute in the enclosing `<fs>` element, rather then give it explicitly. This method is suitable in the situation described above, since it requires specifying a total of only ten (5 + 3 + 2) combinations of features and values. Similarly, to ensure that only feature structures containing valid combinations of feature values are used, one can put definitions for all valid feature structures inside a feature value library (so called, since a feature structure may be the value of a feature). A total of 30 feature structures ($5 \times 3 \times 2$) is required to enumerate all the possible combinations of individual case, gender and number values in the preceding illustration. We discuss the use of such libraries and their representation in XML further in section 5.4 Feature and Feature-Value Libraries below.

However, the most general method of attempting to ensure that only legal combinations of feature names and values are used is to provide a feature-system declaration discussed in chapter FD of the TEI Guidelines.

5.4 Feature and Feature-Value Libraries

As the examples in the preceding section suggest, the direct encoding of feature structures can be verbose. Moreover, it is often the case that particular feature-value combinations, or feature structures composed of them, are re-used in different analyses. To reduce the size and complexity of the task of encoding feature structures, one may use the *feats* attribute of the `<fs>` element to point to one or more of the feature-value specifications for that element. This indirect method of encoding feature structures presumes that the `<f>` elements are assigned unique *id* values, and are collected together in `<fLib>` elements (feature libraries). In the same way, feature values of whatever type can be collected together in `<fvLib>` elements (feature-value libraries). If a feature has as its value a feature structure or other value which is predefined in this way, the *fVal* attribute may be used to point to it, as discussed in the next section. The following elements are used for representing feature, and feature-value libraries:

(76)

— **<fLib>** assembles library of feature elements.

  * No attributes other than those globally available (see definition for tei.global.attributes)

— **<fvLib>** assembles a library of reusable feature value elements (including complete feature structures).

  type indicates type of feature-value library (i.e., what type of feature values it contains).

For example, suppose a feature library for phonological feature specifications is set up as follows.

(77)

```
<fLib n="phonological features">
 <f id="CNS1" name="consonantal">
  <binary value="true"/>
 </f>
 <f id="CNS0" name="consonantal">
  <binary value="false"/>
 </f>
 <f id="VOC1" name="vocalic">
  <binary value="true"/>
```

```
 </f>
 <f id="VOC0" name="vocalic">
  <binary value="false"/>
 </f>
 <f id="VOI1" name="voiced">
  <binary value="true"/>
 </f>
 <f id="VOI0" name="voiced">
  <binary value="false"/>
 </f>
 <f id="ANT1" name="anterior">
  <binary value="true"/>
 </f>
 <f id="ANT0" name="anterior">
  <binary value="false"/>
 </f>
 <f id="COR1" name="coronal">
  <binary value="true"/>
 </f>
 <f id="COR0" name="coronal">
  <binary value="false"/>
 </f>
 <f id="CNT1" name="continuant">
  <binary value="true"/>
 </f>
 <f id="CNT0" name="continuant">
  <binary value="false"/>
 </f>
 <f id="STR1" name="strident">
  <binary value="true"/>
 </f>
 <f id="STR0" name="strident">
  <binary value="false"/>
 </f><!-- ... -->
</fLib>
```

Then the feature structures that represent the analysis of the phonological segments (phonemes) /t/, /d/, /s/, and /z/ may be defined as follows.

(78)

```
<fs feats="CNS1 VOC0 VOI0 ANT1 COR1 CNT0 STR0"/>
<fs feats="CNS1 VOC0 VOI1 ANT1 COR1 CNT0 STR0"/>
<fs feats="CNS1 VOC0 VOI0 ANT1 COR1 CNT1 STR1"/>
<fs feats="CNS1 VOC0 VOI1 ANT1 COR1 CNT1 STR1"/>
```

The preceding are but four of the 128 logically possible fully specified phonological segments using the seven binary features listed in the feature library. Presumably not all combinations of features correspond to phonological segments (there are no strident vowels, for example). The legal combinations, however, can be collected together, each one represented as an identifiable <fs> element within a feature-value library, as in the following example:

(79)

```
<fvLib id="fsl1" n="phonological segment definitions"><!-- ... -->
 <fs id="T.DF" feats="CNS1 VOC0 VOI0 ANT1 COR1 CNT0 STR0"/>
 <fs id="D.DF" feats="CNS1 VOC0 VOI1 ANT1 COR1 CNT0 STR0"/>
 <fs id="S.DF" feats="CNS1 VOC0 VOI0 ANT1 COR1 CNT1 STR1"/>
 <fs id="Z.DF" feats="CNS1 VOC0 VOI1 ANT1 COR1 CNT1 STR1"/><!-- ... -->
</fvLib>
```

Once defined, these feature structure values can also be reused. Other <f> elements may invoke them by reference, using the fVal attribute; for example, one might use them in a feature value pair such as:

(80)

```
<f n="dental-fricative" fval="T.DF"/>
```

rather than expanding the hierarchy of the component phonological features explicitly.

Feature structures stored in this way may also be associated with the text which they are intended to annotate, either by a link from the text (for example, using the TEI global ana attribute), or by means of standoff annotation techniques (for example, using the TEI <link> element): see further section 5.10 Linking Text and Analysis below.

Note that when features or feature structures are linked to in this way, the result is effectively a copy of the item linked to into the place from which it is linked. This form of linking should be distinguished from the phenomenon of structure-sharing, where it is desired to indicate that some part of an annotation structure appears simultaneously in two or more places within the structure. This kind of annotation should be represented using the <vLabel> element, as discussed in section 5.6 Re-entrant feature structures below.

5.5 Feature Structures as Complex Feature Values

Features may have complex values as well as atomic ones; the simplest such complex value is represented by supplying a <fs> element as the content of an <f> element, or (equivalently) by supplying the identifier of an <fs> element as the value for the fVal attribute on the <f> element. Structures may be nested as deeply as appropriate, using this mechanism. For example, an <fs> element may contain or point to an <f> element, which may contain or point to an <fs> element, which may contain or point to an <f> element, and so on.

To illustrate the use of complex values, consider the following simple model of a word, as a structure combining surface form information, a syntactic category, and semantic information. Each word analysis is represented as a <fs type='word'> element, containing three features named surface, syntax, and semantics. The first of these has an atomic string value, but the other two have complex values, represented as nested feature structures of types category and act respectively:

(81)

```
<fs type="word">
 <f name="surface">
  <string>love</string>
 </f>
 <f name="syntax">
  <fs type="category">
   <f name="pos">
    <symbol value="verb"/>
   </f>
   <f name="val">
```

```
    <symbol value="transitive"/>
   </f>
  </fs>
 </f>
 <f name="semantics">
  <fs type="act">
   <f name="rel">
    <symbol value="LOVE"/>
   </f>
  </fs>
 </f>
</fs>
```

This analysis does not tell us much about the meaning of the symbols `verb` or `transitive`. It might be preferable to replace these atomic feature values by feature structures. Suppose therefore that we maintain a feature-value library for each of the major syntactic categories (N, V, ADJ, PREP):

(82)

```
<fvLib n="Major category definitions"><!-- ... -->
 <fs id="N" type="noun"><!--  noun features defined here -->
 </fs>
 <fs id="V" type="verb"><!-- verb features defined here -->
 </fs>
</fvLib>
```

This library allows us to use shortcut codes (`N`, `V` etc.) to reference a complete definition for the corresponding feature structure. Each definition may be explicitly contained within the `<fs>` element, as a number of `<f>` elements. Alternatively, the relevant features may be referenced by their identifiers, supplied as the value of the *feats* attribute, as in these examples:

(83)

```
<!-- ... --><fs id="ADJ" type="adjective" feats="N1 V1"/>
 <fs id="PREP" type="preposition" feats="N0 V0"/><!-- ... -->
```

This ability to re-use feature definitions within multiple feature structure definitions is an essential simplification in any realistic example. In this case, we assume the existence of a feature library containing specifications for the basic feature categories like the following:

(84)

```
<fLib type="categorial features">
 <f id="N1" name="nominal">
  <binary value="true"/>
 </f>
 <f id="N0" name="nominal">
  <binary value="false"/>
 </f>
 <f id="V1" name="verbal">
  <binary value="true"/>
```

```
 </f>
 <f id="V0" name="verbal">
  <binary value="false"/>
 </f><!-- ... -->
</fLib>
```

With these libraries in place, and assuming the availability of similarly predefined feature structures for transitivity and semantics, the preceding example could be considerably simplified:

(85)

```
<fs type="word">
 <f name="surf">
  <string>love</string>
 </f>
 <f name="syntax">
  <fs type="category">
   <f name="pos" fVal="V"/>
   <f name="val" fVal="TRNS"/>
  </fs>
 </f>
 <f name="semantics">
  <fs type="act">
   <f name="rel" fVal="LOVE"/>
  </fs>
 </f>
</fs>
```

Although in principle the *fVal* attribute could point to any kind of feature value, its use is not recommended for simple atomic values.

5.6 Re-entrant feature structures

Sometimes the same feature value is required at multiple places within a feature structure, in particular where the value is only partially specified at one or more places. The `<vLabel>` element is provided as a means of labelling each such re-entrancy point:

(86)

— **<vLabel>** represents the value part of a feature-value specification which appears at more than point in a feature structure

name supplies a name for the sharing point.

For example, suppose one wishes to represent noun-verb agreement as a single feature structure. Within the representation, the feature indicating (say) number appears more than once. To represent the fact that each occurrence is another appearance of the same feature (rather than a copy) one could use an encoding like the following:

(87)

```
<fs id="NVA">
 <f name="nominal">
  <fs>
   <f name="nm-num">
    <vLabel label="L1">
     <symbol value="singular"/>
    </vLabel>
   </f><!-- other nominal features -->
  </fs>
 </f>
 <f name="verbal">
  <fs>
   <f name="vb-num">
    <vLabel label="L1"/>
   </f>
  </fs><!-- other verbal features -->
 </f>
</fs>
```

In the above encoding, the features named `vb-num` and `nm-num` exhibit structure sharing. Their values, given as `vLabel` elements, are understood to be references to the same point in the feature structure, which is labelled by their *name* attribute.

The scope of the names used to label re-entrancy points is that of the outermost `<fs>` element in which they appear. When a feature structure is imported from a feature value library, or referenced from elsewhere (for example by using the *fVal* attribute) the names of any sharing points it may contain are implicitly prefixed by the identifier used for the imported feature structure, to avoid name clashes. Thus, if some other feature structure were to reference the `<fs>` element given in the example above, for example in this way:

(88)

```
<f name="class" fVal="NVA"/>
```

then the labelled points in the example would be interpreted as if they had the name `NVAL1`.

5.7 Collections as Complex Feature Values

Complex feature values need not always be represented as feature structures. Multiple values may also be organized as as sets, bags (or multisets), or lists of atomic values of any type. The `<coll>` element is provided to represent such cases:

(89)

— **<coll>** represents the value part of a feature-value specification which contains multiple values organized as a set, bag, or list.

org indicates organization of given value or values as set, bag or list. Legal values are:

set indicates that the given values are organized as a set.

bag indicates that the given values are organized as a bag (multiset).

list indicates that the given values are organized as a list.

A feature whose value is regarded as a set, bag or list may have any positive number of values as its content, or none at all, (thus allowing for representation of the empty set, bag or list). The items in a list are ordered, and

need not be distinct. The items in a set are not ordered, and must be distinct. The items in a bag are neither ordered nor distinct. Sets and bags are thus distinguished from lists in that the order in which the values are specified does not matter for the former, but does matter for the latter, while sets are distinguished from bags and lists in that repetitions of values do not count for the former but do count for the latter.

If no value is specified for the *org* attribute, the assumption is that the <coll> defines a list of values. If the <coll> element is empty, the assumption is that it represents the null list, set, or bag.

To illustrate the use of the *org* attribute, suppose that a feature structure analysis is used to represent a genealogical tree, with the information about each individual treated as a single feature structure, like this:

(90)

```
<fs id="p027" type="person">
 <f name="forenames">
  <coll>
   <string>Daniel</string>
   <string>Edouard</string>
  </coll>
 </f>
 <f name="mother" fVal="p002"/>
 <f name="father" fVal="p009"/>
 <f name="birthDate">
  <fs type="date" feats="y1988 m04 d17"/>
 </f>
 <f name="birthPlace" fVal="austintx"/>
 <f name="siblings">
  <coll org="set">
   <fs copyOf="pnb005"/>
   <fs copyOf="prb001"/>
  </coll>
 </f>
</fs>
```

In this example, the <coll> element is first used to supply a list of 'name' feature values, which together constitute the 'forenames' feature. Other features are defined by reference to values which we assume are held in some external feature value library (not shown here). For example, the <coll> element is used a second time to indicate that the persons's siblings should be regarded as constituting a set rather than a list. Each sibling is represented by a feature structure: in this example, each feature structure is a copy of one specified in the feature value library.

If a specific feature contains only a single feature structure as its value, the component features of which are organized as a set, bag or list, it may be more convenient to represent the value as a <coll> rather than as a <fs>. For example, consider the following encoding of the English verb form "sinks" which contains an **agreement** feature whose value is a feature structure which contains **person** and **number** features with symbolic values.

(91)

```
<fs type="word">
 <f name="category">
  <symbol value="verb"/>
 </f>
 <f name="tense">
```

```
  <symbol value="present"/>
 </f>
 <f name="agreement">
  <fs>
   <f name="person">
    <symbol value="third"/>
   </f>
   <f name="number">
    <symbol value="singular"/>
   </f>
  </fs>
 </f>
</fs>
```

If the names of the features contained within the **agreement** feature structure are of no particular significance, the following simpler representation may be used:

(92)

```
<fs type="word">
 <f name="word.oddss">
  <symbol value="verb"/>
 </f>
 <f name="tense">
  <symbol value="present"/>
 </f>
 <f name="agreement">
  <coll org="set">
   <symbol value="third"/>
   <symbol value="singular"/>
  </coll>
 </f>
</fs>
```

The <coll> element is also useful in cases where an analysis has several components. In the following example, the French word “auxquels" has a two-part analysis, represented as a list of two values. The first specifies that the word contains a preposition; the second that it contains a masculine plural relative pronoun:

(93)

```
<fs>
 <f name="lex">
  <symbol value="auxquels"/>
 </f>
 <f name="maf">
  <coll org="list">
   <fs>
    <f name="cat">
     <symbol value="prep"/>
    </f>
   </fs>
   <fs>
    <f name="cat">
```

```
     <symbol value="pronoun"/>
    </f>
    <f name="kind">
     <symbol value="rel"/>
    </f>
    <f name="num">
     <symbol value="pl"/>
    </f>
    <f name="gender">
     <symbol value="masc"/>
    </f>
   </fs>
  </coll>
 </f>
</fs>
```

The set, bag or list which has no members is known as the null (or empty) set, bag or list. A <coll> element with no content and with no value for its *feats* attribute is interpreted as referring to the null set, bag, or list, depending on the value of its *org* attribute.

If, for example, the individual described by the feature structure with identifier p027 (above) had no siblings, we might specify the “siblings" feature as follows.

(94)

```
<f name="siblings">
 <coll org="set"/>
</f>
```

A <coll> element may also collect together one or more other <coll> elements, if, for example one of the members of a set is itself a set, or if two lists are concatenated together. Note that such collections pay no attention to the contents of the nested <coll> elements: if it is desired to produce the union of two sets, the <vMerge> element discussed below should be used to make a new collection from the two sets.

5.8 Feature Value Expressions

It is sometimes desirable to express the value of a feature as the result of an operation over some other value (for example, as ‘not green’, or as ‘male or female’, or as the concatenation of two collections). Three special purpose elements are provided to represent disjunctive alternation, negation, and collection of values:

(95)

— **<vAlt>** represents the value part of a feature-value specification which contains a set of values, only one of which can be valid.

  * No attributes other than those globally available (see definition for tei.global.attributes)

— **<vNot>** represents a feature value which is the negation of its content.

  * No attributes other than those globally available (see definition for tei.global.attributes)

— **<vMerge>** represents a feature value which is the result of merging together the feature values contained by its children, using the organization specified by the ORG attribute.

  org indicates the organization of the resulting merged values as set, bag or list. Legal values are:

set indicates that the resulting values are organized as a set.
bag indicates that the resulting values are organized as a bag (multiset).
list indicates that the resulting values are organized as a list.

5.8.1 Alternation

The <vAlt> element can be used wherever a feature value can appear. It contains two or more feature values, any one of which is to be understood as the value required. Suppose, for example, that we are using a feature system to describe residential property, using such features as âĂŸnumber.of.bathroomsâĂŹ. In a particular case, we might wish to represent uncertainty as to whether a house has two or three bathrooms. As we have already shown, one simple way to represent this would be with a numeric maximum:

(96)

```
<f name="number.of.bathrooms">
 <numeric value="2" max="3"/>
</f>
```

A better, and more general, way would be to represent the alternation explicitly, in this way:

(97)

```
<f name="number.of.bathrooms">
 <vAlt>
  <numeric value="2"/>
  <numeric value="3"/>
 </vAlt>
</f>
```

The <vAlt> element represents alternation over feature values, not feature-value pairs. If therefore the uncertainty relates to two or more feature value specifications, each must be represented as a feature structure, since a feature structure can always appear where a value is required. For example, suppose that it is uncertain as to whether the house being described has two bathrooms or two bedrooms, a structure like the following may be used:

(98)

```
<f name="rooms">
 <vAlt>
  <fs>
   <f name="number.of.bathrooms">
    <numeric value="2"/>
   </f>
  </fs>
  <fs>
   <f name="number.of.bedrooms">
    <numeric value="2"/>
   </f>
  </fs>
 </vAlt>
</f>
```

Note that alternation is always regarded as exclusive: in the case above, the implication is that having two bathrooms excludes the possibility of having two bedrooms and vice versa. If inclusive alternation is required, a `<coll>` element may be included in the alternation as follows:

(99)

```
<f name="rooms">
 <vAlt>
  <fs>
   <f name="number.of.bathrooms">
    <numeric value="2"/>
   </f>
  </fs>
  <fs>
   <f name="number.of.bedrooms">
    <numeric value="2"/>
   </f>
  </fs>
  <coll>
   <fs>
    <f name="number.of.bathrooms">
     <numeric value="2"/>
    </f>
   </fs>
   <fs>
    <f name="number.of.bedrooms">
     <numeric value="2"/>
    </f>
   </fs>
  </coll>
 </vAlt>
</f>
```

This analysis indicates that the property may have two bathrooms, two bedrooms, or both two bathrooms and two bedrooms.

As the previous example shows, the `<vAlt>` element can also be used to indicate alternations among values of features organized as sets, bags or lists. Suppose we use a feature `selling.points` to describe items that are mentioned to enhance a property's sales value, such as whether it has a pool or a good view. Now suppose for a particular listing, the selling points include an alarm system and a good view, and either a pool or a jacuzzi (but not both). This situation could be represented, using the `<vAlt>` element, as follows.

(100)

```
<fs type="real estate listing">
 <f name="selling.points">
  <coll org="set">
   <string>alarm system</string>
   <string>good view</string>
   <vAlt>
    <string>pool</string>
    <string>jacuzzi</string>
   </vAlt>
  </coll>
```

```
  </f>
 </fs>
```

Now suppose the situation is like the preceding except that one is also uncertain whether the property has an alarm system or a good view. This can be represented as follows.

(101)

```
<fs type="real estate listing">
 <f name="selling.points">
  <coll org="set">
   <vAlt>
    <string>alarm system</string>
    <string>good view</string>
   </vAlt>
   <vAlt>
    <string>pool</string>
    <string>jacuzzi</string>
   </vAlt>
  </coll>
 </f>
</fs>
```

If a large number of ambiguities or uncertainties need to be represented, involving a relatively small number of features and values, it is recommended that a stand-off technique, for example using the general-purpose `<alt>` element discussed in TEI P5, be used, rather than the special-purpose `<vAlt>` element.

5.8.2 Negation

The `<vNot>` element can be used wherever a feature value can appear. It contains any feature value and returns the complement of its contents. For example, the feature “number.of.bathrooms" in the following example has any whole numeric value other than 2:

(102)

```
<f name="number.of.bathrooms">
 <vNot>
  <numeric value="2"/>
 </vNot>
</f>
```

Strictly speaking, the effect of the `<vNot>` element is to provide the complement of the feature values it contains, rather than their negation. If a feature system declaration is available which defines the possible values for the associated feature, then it is possible to say more about the negated value. For example, suppose that the available values for the feature `case` are declared to be nominative, genitive, dative, or accusative, whether in a TEI feature system declaration or by some other means. Then the following two specifications are equivalent:

(103)

```
(i) <f name="case">
 <vNot>
```

```
  <symbol value="genitive"/>
 </vNot>
</f>

(ii) <f name="case">
 <vAlt>
  <symbol value="nominative"/>
  <symbol value="dative"/>
  <symbol value="accusative"/>
 </vAlt>
</f>
```

If however no such system declaration is available, all that one can say about a feature specified via negation is that its value is something other than the negated value.

Negation is always applied to a feature value, rather than to a feature-value pair. The negation of an atomic value is the set of all other values which are possible for the feature.

Any kind of value can be negated, including collections (represented by a `<coll>` elements) or feature structures (represented by `<fs>` elements). The negation of any complex value is understood to be the set of values which cannot be unified by it. Thus, for example, the negation of the feature structure F is understood to be the set of feature structures which are not unifiable with F. In the absence of a constraint mechanism such as the Feature System Declaration, the negation of a collection is anything that is not unifiable with it, including collections of different types and atomic values. It will generally be more useful to require that the organization of the negated value be the same as that of the original value, for example that a negated set is understood to mean the set which is a complement of the set, but such a requirement cannot be enforced in the absence of a constraint mechanism.

5.8.3 Collection of values

The `<vMerge>` element can be used wherever a feature value can appear. It contains two or more feature values, all of which are to be collected together. The organization of the resulting collection is specified by the value of the org attribute, which need not necessarily be the same as that of its constituent values if these are collections. For example, one can change a list to a set, or vice versa.

As an example, suppose that we wish to represent the range of possible values for a feature 'genders' used to describe some language. It would be natural to represent the possible values as a set, using the `<coll>` element as in the following example:

(104)

```
<fs>
 <f name="genders">
  <coll org="set">
   <sym value="masculine"/>
   <sym value="feminine"/>
  </coll>
 </f>
</fs>
```

Suppose however that we discover for some language it is necessary to add a new possible value, and to treat the value of the feature as a list rather than as a set. The `<vMerge>` element can be used to achieve this:

(105)

```
<fs>
 <f name="genders">
  <vMerge org="list">
   <coll org="set">
    <sym value="masculine"/>
    <sym value="feminine"/>
   </coll>
   <sym value="neuter"/>
  </vMerge>
 </f>
</fs>
```

5.9 Default Values

The value of a feature may be underspecified in a number of different ways. It may be null, unknown, or uncertain with respect to a range of known possibilities, as well as being defined as a negation or an alternation. As previously noted, the specification of the range of known possibilities for a given feature is not part of the current specification: in the TEI scheme, this information is conveyed by the feature system declaration. Using this, or some other system, we might specify (for example) that the range of values for an element includes symbols for masculine, feminine, and neuter, and that the default value is neuter. With such definitions available to us, it becomes possible to say that some feature takes the default value, or some unspecified value from the list. The following special element is provided for this purpose:

(106)

— **<default>** represents the value part of a feature-value specification which contains a defaulted value.

  * No attributes other than those globally available (see definition for tei.global.attributes)

The value of an empty `<f>` element which also lacks a fVal attribute is understood to be the most general case, i.e. any of the available values. Thus, assuming the feature system defined above, the following two representations are equivalent.

(107)

```
<f name="gender"/>
<f name="gender">
 <vAlt>
  <symbol value="feminine"/>
  <symbol value="masculine"/>
  <symbol value="neuter"/>
 </vAlt>
</f>
```

If, however, the value is explicitly stated to be the default one, using the `<default>` element, then the following two representations are equivalent:

(108)

```
<f name="gender">
 <default/>
</f>
```

(109)

```
<f name="gender">
 <symbol value="neuter"/>
</f>
```

Similarly, if the value is stated to be the negation of the default, then the following two representations are equivalent:

(110)

```
<f name="gender">
 <vNot>
  <default/>
 </vNot>
</f>
```

(111)

```
<f name="gender">
 <vAlt>
  <symbol value="feminine"/>
  <symbol value="masculine"/>
 </vAlt>
</f>
```

5.10 Linking Text and Analysis

Text elements can be linked with feature structures using any of the linking methods discussed elsewhere in the Guidelines. In the simplest case, the `ana` attribute may be used to point from any element to an annotation of it, as in the following example:

(112)

```
<s n="00741">
 <w ana="at0">The</w>
 <w ana="ajs">closest</w>
 <w ana="pnp">he</w>
 <w ana="vvd">came</w>
 <w ana="prp">to</w>
 <w ana="nn1">exercise</w>
 <w ana="vbd">was</w>
 <w ana="to0">to</w>
 <w ana="vvi">open</w>
 <w ana="crd">one</w>
 <w ana="nn1">eye</w>
 <phr ana="av0">
  <w>every</w>
  <w>so</w>
  <w>often</w>
 </phr>
```

```
 <c ana="pun">,</c>
 <w ana="cjs">if</w>
 <w ana="pni">someone</w>
 <w ana="vvd">entered</w>
 <w ana="at0">the</w>
 <w ana="nn1">room</w><!-- ... -->
</s>
```

The values specified for the *ana* attribute reference components of a feature-structure library, which represents all of the grammatical structures used by this encoding scheme. (For illustrative purposes, we cite here only the structures needed for the first six words of the sample sentence):

(113)

```
<fsLib id="C6" type="Claws 6 POS Codes"><!-- ... -->
 <fs id="ajs" type="grammatical structure" feats="wj ds"/>
 <fs id="at0" type="grammatical structure" feats="wl"/>
 <fs id="pnp" type="grammatical structure" feats="wr rp"/>
 <fs id="vvd" type="grammatical structure" feats="wv bv fd"/>
 <fs id="prp" type="grammatical structure" feats="wp bp"/>
 <fs id="nn1" type="grammatical structure" feats="wn tc ns"/><!-- ... -->
</fsLib>
```

The components of each feature structure in the library are referenced in much the same way, using the *feats* attribute to identify one or more `<f>` elements in the following feature library (again, only a few of the available features are quoted here):

(114)

```
<fLib><!-- ... --><f id="bv" name="verbbase">
  <symbol value="main"/>
 </f>
 <f id="bp" name="prepbase">
  <symbol value="lexical"/>
 </f>
 <f id="ds" name="degree">
  <symbol value="superlative"/>
 </f>
 <f id="fd" name="verbform">
  <symbol value="ed"/>
 </f>
 <f id="ns" name="number">
  <symbol value="singular"/>
 </f>
 <f id="rp" name="prontype">
  <symbol value="personal"/>
 </f>
 <f id="tc" name="nountype">
  <symbol value="common"/>
 </f>
 <f id="wj" name="class">
  <symbol value="adjective"/>
 </f>
```

```
 <f id="wl" name="class">
  <symbol value="article"/>
 </f>
 <f id="wn" name="class">
  <symbol value="noun"/>
 </f>
 <f id="wp" name="class">
  <symbol value="preposition"/>
 </f>
 <f id="wr" name="class">
  <symbol value="pronoun"/>
 </f>
 <f id="wv" name="class">
  <symbol value="verb"/>
 </f><!-- ... -->
</fLib>
```

Alternatively, a stand-off technique may be used, as in the following example, where a <linkGrp> element is used to link selected characters in the text “Caesar seized control" with their phonological representations.

(115)

```
<text><!-- ... --><body><!-- ... --><s id="S1">
   <w id="S1W1">
    <c id="S1W1C1">C</c>ae<c id="S1W1C2">s</c>ar</w>
   <w id="S1W2">
    <c id="S1W2C1">s</c>ei<c id="S1W2C2">z</c>e<c id="S1W2C3">d</c>
   </w>
   <w id="S1W3">con<c id="S1W3C1">t</c>rol</w>.
   </s><!-- ... -->
 </body>
 <fvLib id="FSL1" n="phonological segment definitions"><!-- as in previous example -->
 </fvLib>
 <linkGrp type="phonology"><!-- ... --><link targets="S.DF S1W3C1"/>
  <link targets="Z.DF S1W2C3"/>
  <link targets="S.DF S1W2C1"/>
  <link targets="Z.DF S1W2C2"/><!-- ... -->
 </linkGrp>
</text>
```

As this example shows, a stand-off solution requires that every component to be linked to must bear an identifier. To handle the POS tagging example above, therefore, each annotated element would need an identifier of some sort, as follows:

(116)

```
<s id="mds09" n="00741">
 <w id="mds0901">The</w>
 <w id="mds0902">closest</w>
 <w id="mds0903">he</w>
 <w id="mds0904">came</w>
 <w id="mds0905">to</w>
 <w id="mds0906">exercise</w><!-- ... -->
</s>
```

It would then be possible to link each word to its intended annotation in the feature library quoted above, as follows:

(117)

```
<linkGrp type="POS-codes"><!-- ... --><link targets="mds0901 at0"/>
 <link targets="mds0902 ajs"/>
 <link targets="mds0903 pnp"/>
 <link targets="mds0904 vvd"/>
 <link targets="mds0905 prp"/>
 <link targets="mds0906 nn1"/>
 <link targets="mds0907 vbd"/>
 <link targets="mds0908 to0"/>
 <link targets="mds0909 vvi"/>
 <link targets="mds0910 crd"/><!-- ... -->
</linkGrp>
```

Annex A
(informative)
Formal Definitions and Implementation of the XML Representation of Feature Structures

The elements discussed in this chapter constitute a module of the TEI scheme which, like other modules, may be expressed using a variety of schema languages. This Annex provides a formal definition for the whole module, using the compact syntax defined for ISO 19757-2 (Document Schema Definition Language, part 2). This is followed by formal documentation for each element defined by the module, identical to that provided by the TEI Guidelines.

A.1 RelaxNG specification for the module

```
\relax \@setckpt{iso-fs.rnc}
{\setcounter{page}{47}
\setcounter{equation}{0}
\setcounter{enumi}{3}
\setcounter{enumii}{0}
\setcounter{enumiii}{0}
\setcounter{enumiv}{0}
\setcounter{footnote}{14}
\setcounter{mpfootnote}{0}
\setcounter{part}{0}
\setcounter{section}{1}
\setcounter{subsection}{1}
\setcounter{subsubsection}{0}
\setcounter{paragraph}{0}
\setcounter{subparagraph}{0}
\setcounter{figure}{0}
\setcounter{table}{0}
\setcounter{subsubparagraph}{0}
\setcounter{term}{0}
\setcounter{enums}{57}
\setcounter{tempcnt}{0}
\setcounter{enumsi}{2}
\setcounter{lstlisting}{0}
\setcounter{lstnumber}{1} }
```

A.2 Element documentation

A.2.1 binary [element]

Description: (binary value) represents the value part of a feature-value specification which can contain either of exactly two possible values.

Classes: tei.singleVal

Declaration:

```
element binary { tei.global.attributes, binary.attributes.value,
empty }
```

Attributes: (In addition to global attributes and those inherited from tei.singleVal)

value supplies a binary value (true or false, plus or minus).

Example:

```
<f name="strident">
 <binary value="true"/>
</f>
<f name="exclusive">
 <binary value="no"/>
</f>
```

The value attribute may take any value permitted for attributes of the XML datatype Boolean: this includes for example the strings `true`, `yes`, or `1` which are all equivalent.

Module: iso-fs

A.2.2 coll [element]

Description: (collection of values) represents the value part of a feature-value specification which contains multiple values organized as a set, bag, or list.

Classes: tei.complexVal

Declaration:

```
element coll {
   tei.global.attributes,
   coll.attributes.org,
   ( ( fs | tei.singleVal )* )
}
```

Attributes: (In addition to global attributes and those inherited from tei.complexVal)

org indicates organization of given value or values as set, bag or list. Legal values are:

- **set** indicates that the given values are organized as a set.
- **bag** indicates that the given values are organized as a bag (multiset).
- **list** indicates that the given values are organized as a list.

Example:

```
<f name="name">
 <coll>
  <string>Jean</string>
  <string>Luc</string>
  <string>Godard</string>
 </coll>
</f>
```

Example:

```
<fs>
 <f name="lex">
  <symbol value="auxquels"/>
 </f>
 <f name="maf">
  <coll org="list">
   <fs>
    <f name="cat">
     <symbol value="prep"/>
    </f>
   </fs>
   <fs>
    <f name="cat">
     <symbol value="pronoun"/>
    </f>
    <f name="kind">
     <symbol value="rel"/>
    </f>
    <f name="num">
     <symbol value="pl"/>
    </f>
    <f name="gender">
     <symbol value="masc"/>
    </f>
   </fs>
  </coll>
 </f>
</fs>
```

Module: iso-fs

A.2.3 default [element]

Description: (Default feature value) represents the value part of a feature-value specification which contains a defaulted value.

Attributes: Global attributes only

Classes: tei.singleVal

Declaration:

```
element default { tei.global.attributes, empty }
```

Example:

```
<f name="gender">
 <default/>
</f>
```

Module: iso-fs

A.2.4 f [element]

Description: (Feature) represents a feature value specification, that is, the association of a name with a value of any of several different types.

Declaration:

```
element f {
   tei.global.attributes,
   f.attributes.name,
   f.attributes.fVal,
   tei.featureVal*
}
```

Attributes: (In addition to global attributes)

name provides a name for the feature.

fVal references any element which can be used to represent the value of a feature.

Example:

```
 <f name="gender">
  <sym value="feminine"/>
 </f>
```

If the element is empty then a value must be supplied for the *fVal* attribute.

Module: iso-fs

A.2.5 fLib [element]

Description: (Feature library) assembles library of feature elements.

Classes: tei.metadata

Declaration:

```
element fLib { tei.global.attributes, f+ }
```

Attributes: Global attributes and those inherited from tei.metadata

Example:

```
 <fLib n="agreement features">
  <f id="p1" name="person">
   <sym value="first"/>
  </f>
  <f id="p2" name="person">
   <sym value="second"/>
  </f><!-- ... --><f id="ns" name="number">
   <sym value="singular"/>
  </f>
  <f id="np" name="number">
   <sym value="plural"/>
  </f><!-- ... -->
 </fLib>
```

The global *n* attribute may be used to supply an informal name to categorise the library's contents.

Module: iso-fs

A.2.6 fs [element]

Description: (Feature structure) represents a feature structure, that is, a collection of feature-value pairs organized as a structural unit.

Classes: tei.complexVal tei.metadata

Declaration:

```
element fs {
   tei.global.attributes,
   fs.attributes.type,
   fs.attributes.feats,
   f*
}
```

Attributes: (In addition to global attributes and those inherited from tei.complexVal, tei.metadata)

type specifies the type of the feature structure.

feats references the feature-value specifications making up this feature structure.

Example:

```
 <fs type="agreement structure" rel="ns">
  <f name="person">
   <sym value="third"/>
  </f>
  <f name="number">
   <sym value="singular"/>
  </f>
 </fs>
```

Module: iso-fs

A.2.7 fvLib [element]

Description: (Feature-value library) assembles a library of reusable feature value elements (including complete feature structures).

Classes: tei.metadata

Declaration:

```
element fvLib { tei.global.attributes, fvLib.attributes.type,
tei.featureVal* }
```

Attributes: (In addition to global attributes and those inherited from tei.metadata)

type indicates type of feature-value library (i.e., what type of feature values it contains).

Example:

```
<fvLib type="symbolic values">
 <symbol id="sfirst" value="first"/>
 <symbol id="ssecond" value="second"/><!-- ... --><symbol id="ssing" value="singular"/>
 <symbol id="splur" value="plural"/><!-- ... -->
</fvLib>
```

A feature value library may include any number of values of any kind, including multiple occurrences of identical values such as `<binary value="true"/>` or `default`. The only thing guaranteed unique in a feature value library is the set of labels used to identify the values.

Module: iso-fs

A.2.8 numeric [element]

Description: (Numeric value) represents the value part of a feature-value specification which contains a numeric value or range.

Classes: tei.singleVal

Declaration:

```
element numeric {
   tei.global.attributes,
   numeric.attributes.value,
   numeric.attributes.max,
   numeric.attributes.trunc,
   empty
}
```

Attributes: (In addition to global attributes and those inherited from tei.singleVal)

value supplies a lower bound for the numeric value represented, and also (if `max` is not supplied) its upper bound.

max supplies an upper bound for the numeric value represented.

trunc specifies whether the value represented should be truncated to give an integer value.

Example:

```
<numeric value="42"/>
```

This represents the numeric value 42.

Example:

```
<numeric value="42.45" max="50" trunc="yes"/>
```

This represents any of the nine possible integer values between 42 and 50 inclusive. If the `trunc` attribute had the value NO, this example would represent any of the infinite number of numeric values between 42.45 and 50.0

It is an error to supply the max attribute in the absence of a value for the value attribute.

Module: iso-fs

A.2.9 string [element]

Description: (String value) represents the value part of a feature-value specification which contains a string.

Classes: tei.singleVal

Declaration:

```
element string { tei.global.attributes, text }
```

Attributes: Global attributes and those inherited from tei.singleVal

Example:

```
 <f name="greeting">
  <string>Hello, world!</string>
 </f>
```

Module: iso-fs

A.2.10 symbol [element]

Description: (Symbolic value) represents the value part of a feature-value specification which contains one of a finite list of symbols.

Classes: tei.singleVal

Declaration:

```
element symbol { tei.global.attributes, symbol.attributes.value,
empty }
```

Attributes: (In addition to global attributes and those inherited from tei.singleVal)

value supplies the symbolic value for the feature, one of a finite list that may be specified in a feature declaration.

Example:

```
 <f name="gender">
  <symbol value="feminine"/>
 </f>
```

Module: iso-fs

A.2.11 vAlt [element]

Description: (Value alternation) represents the value part of a feature-value specification which contains a set of values, only one of which can be valid.

Classes: tei.complexVal

Declaration:

```
element vAlt { tei.global.attributes, ( ( tei.featureVal ),
tei.featureVal+ ) }
```

Attributes: Global attributes and those inherited from tei.complexVal

Example:

```
 <f name="gender">
  <vAlt>
   <sym value="masculine"/>
   <sym value="neuter"/>
   <sym value="feminine"/>
  </vAlt>
 </f>
```

Module: iso-fs

A.2.12 vLabel [element]

Description:(value label) represents the value part of a feature-value specification which appears at more than point in a feature structure

Classes: tei.singleVal

Declaration:

```
element vLabel {
   tei.global.attributes,
   vLabel.attributes.name,
   tei.featureVal?
}
```

Attributes: (In addition to global attributes and those inherited from tei.singleVal)

name supplies a name for the sharing point.

Example:

```
 <fs id="NVA">
  <f name="nominal">
   <fs>
    <f name="nm-num">
     <vLabel name="L1">
      <symbol value="singular"/>
     </vLabel>
    </f><!-- other nominal features -->
   </fs>
  </f>
```

```
  <f name="verbal">
   <fs>
    <f name="vb-num">
     <vLabel name="L1"/>
    </f>
   </fs><!-- other verbal features -->
  </f>
 </fs>
```

Module: iso-fs

A.2.13 vMerge [element]

Description: (Merged collection of values) represents a feature value which is the result of merging together the feature values contained by its children, using the organization specified by the ORG attribute.

Classes: tei.complexVal

Declaration:

```
element vMerge {
   tei.global.attributes,
   vMerge.attributes.org,
   tei.featureVal+
}
```

Attributes: (In addition to global attributes and those inherited from tei.complexVal)

org indicates the organization of the resulting merged values as set, bag or list. Legal values are:

- **set** indicates that the resulting values are organized as a set.
- **bag** indicates that the resulting values are organized as a bag (multiset).
- **list** indicates that the resulting values are organized as a list.

Example:

```
 <vMerge org="list">
  <coll org="set">
   <symbol value="masculine"/>
   <symbol value="neuter"/>
   <symbol value="feminine"/>
  </coll>
  <symbol value="indeterminate"/>
 </vMerge>
```

This example returns a list, concatenating the indeterminate value with the set of values masculine, neuter and feminine.

Module: iso-fs

A.2.14 vNot [element]

Description: (Value negation) represents a feature value which is the negation of its content.

Classes: tei.complexVal

Declaration:

```
element vNot { tei.global.attributes, ( tei.featureVal ) }
```

Attributes: Global attributes and those inherited from tei.complexVal

Example:

```
<vNot>
 <sym value="masculine"/>
</vNot>
```

Example:

```
<f name="mode">
 <vNot>
  <vAlt>
   <symbol value="infinitive"/>
   <symbol value="participle"/>
  </vAlt>
 </vNot>
</f>
```

Module: iso-fs

Annex B
(informative)
Examples for Illustration

Consider the problem of specifying the grammatical *case*, *gender* and *number* features of classical Greek noun forms. Assuming that the case feature can take on any of the five values `nominative`, `genitive`, `dative`, `accusative` and `vocative`; that the gender feature can take on any of the three values `feminine`, `masculine`, and `neuter`; and that the number feature can take on either of the values `singular` and `plural`, then the following may be used to represent the claim that the Latin word *rosas* meaning 'roses' has accusative case, feminine gender and plural number.

```
<fs type="word structure">
   <f name="case">   <symbol value="accusative"/> </f>
   <f name="gender"> <symbol value="feminine"/> </f>
   <f name="number"> <symbol value="plural"/> </f>
</fs>
```

An XML parser by itself cannot determine that particular values do or do not go with particular features; in particular, it cannot distinguish between the presumably legal encodings in the preceding two examples and the presumably illegal encoding in the following example.

```
<!-- *PRESUMABLY ILLEGAL* ... --> <fs type="word structure">
   <f name="case">   <symbol value="feminine"/> </f>
   <f name="gender"> <symbol value="accusative"/> </f>
   <f name="number"> <minus/> </f>
</fs>
```

There are two ways of attempting to ensure that only legal combinations of feature names and values are used. First, if the total number of legal combinations is relatively small, one can simply list all of those combinations in `<fLib>` elements (together possibly with `<fvLib>` elements), and point to them using the `feats` attribute in the enclosing `<fs>` element. This method is suitable in the situation described above, since it requires specifying a total of only ten (5 + 3 + 2) combinations of features and values. Further, to ensure that the features are themselves combined legally into feature structures, one can put the legal feature structures inside `<fsLib>` elements. A total of 30 feature structures (5 x 3 x 2) is required to enumerate all the legal combinations of individual case, gender and number values in the preceding illustration. Of course, the legality of the markup requires that the `feat` attributes actually point at legally defined features, which an XML parser, by itself, cannot guarantee. A more general method of attempting to ensure that only legal combinations of feature names and values are used is to provide a feature system declaration that includes a `<valRange>` element for each feature one uses. Here is a sample `<valRange>` element for the 'case' feature described above. For further discussion of the `<valRange>` element, see Annex A; the `<vAlt>` element is discussed in 5.8.1 Alternation.

```
<!-- VALRANGE specification for CASE feature --> <valRange>
  <vAlt>
    <symbol value='nominative'/>
    <symbol value='genitive'/>
    <symbol value='dative'/>
    <symbol value='accusative'/>
    <symbol value='vocative'/>
  </vAlt>
</valRange>
```

Similarly, to ensure that only legal combinations of features are used as the content of feature structures, one should provide `<fsConstraint>` elements for each of the types of feature structure one employs. Validation of the feature structures used in a document based on the feature-system declaration, however, requires that there be an application program that can use the information contained in the feature-system declaration.

Annex C
(informative)
Type Inheritance Hierarchies

Types organize feature structures into natural classes and perform the same role as concepts in terminological knowledge representation systems or abstract data-types in object oriented programming languages. It is therefore natural to think of types as being organized into an inheritance hierarchy based on their generality.

The type inheritance hierarchy is defined by assuming a finite set **type** of types, ordered according to their specificity, where type τ is more specific than type σ if τ inherits all properties and characteristics from σ. In this case σ *subsumes* or is more *general* than τ: for $\sigma, \tau \in \mathbf{type}, \sigma \sqsubseteq \tau$. If $\sigma \sqsubseteq \tau$, then σ is also said to be a *supertype* of τ, or, inversely, τ is a *subtype* of σ.

The standard approach in knowledge representation systems, which is adopted in the definition of type hierarchies, has been to define a finite number of ISA arcs which link subtypes to supertypes. The full subsumption relation is then defined as the *transitive* and *reflexive* closure of the relation determined by the ISA links. A standard restriction on the ISA links is that they must not be cyclic, i.e. it should not be possible to follow the links from a type back to itself. This restriction makes the subsumption relation a *partial order*.

C.1 Definition

A type hierarchy forms a tree-like finite structure. It must have the following properties:

(118) Properties of type hierarchy

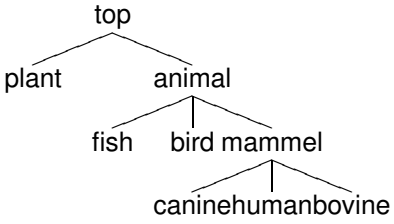
Unique top: It must be a single hierarchy containing all the types with a unique top type.

No cycle: It must have no cycles.

Unique greatest lower bounds: Any two compatible types must have a unique highest common descendant or subtype called *greatest lower bound*.[15] Incompatible types share no common descendants or subtypes.

Here is an example of a type hierarchy depicting a part of the natural world:

(119) Type hierarchy for some animals



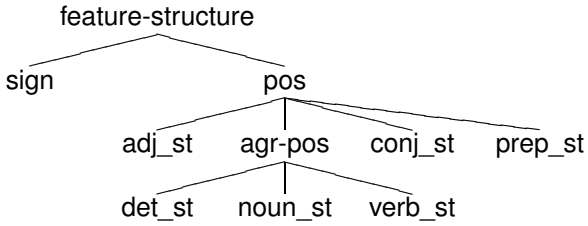
Here, while the types *human* and *canine* are not compatible, the types *animal* and *human* are compatible and thus must have a unique greatest lower bound. Being in the hierarchical relation, the type *human* becomes that lower bound in a trivial manner.

A linguistically more relevant example can be given as below: [16]

15) If the most general type is depicted not as the top, but as the bottom such that the hierarchical tree branches out upward like a real tree with the root at the bottom, then this property must be restated as: Any two compatible types must have a unique least upper bound.

16) Taken modified from Sag, Wasow and Bender (2003: 61).

(120) Linguistic example for type hierarchy

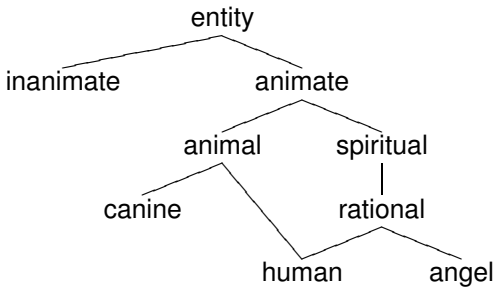


Here the type *feature-structure* is treated as the unique top type. The types *det_st*, *noun_st* and *verb_st* are treated as subtypes of the type *agr-pos*,[17] since they are governed by agreement rules in English.[18]

C.2 Multiple Inheritance

Unlike trees, type hierarchies allow common parents or supertypes. Consider a naive medieval picture of entities as depicted as below:

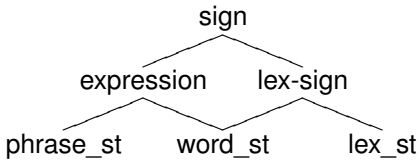
(121) Medieval hierarchy of entities



Subtypes inherit all the properties from their supertypes. The type *human*, for instance, inherits all the properties of its supertypes, both *animal* and *rational*, *spiritual*, *animate* and the top type *entity*. Note that it has two immediate supertypes or *parents*, thus being entitled for so-called *multiple inheritance*. A *human* thus is a *spiritual* and *rational animal* animate being.

Linguistic signs may also allow multiple inheritance like the following.[19].

(122) Multiple inheritance



Here, the type *word_st* inherits all the properties from both of its immediate supertypes *expression* and *lex-sign*. Hence, a word is a lexical expression.

C.3 Type Constraints

In the feature structures discussed so far there is no notion of *type constraints* or simply *typing*: although the nodes in a feature structure graph were labelled with types, arbitrary labellings with type symbols and features were permissible. What is missing is *appropriateness conditions* which model the distinction between features which are not appropriate for a given type and those whose values are simply unknown.

17) *st* stands for “structure".
18) In a language like French or Latin, the type *adj_st* should be also be treated as a subtype of *agr-pos*.
19) See Sag, Wasow and Bender (2003: 470-475)

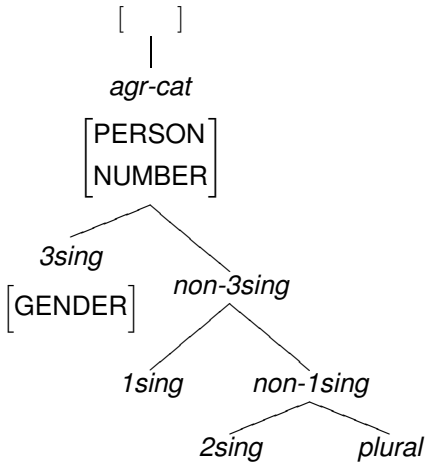
The extension to feature typing is bound to the type hierarchy: for each feature there must be a least type where the feature is introduced and the type of the value for the feature is specified. Furthermore, if a feature is appropriate for a type, then it is appropriate for all of its subtypes.

Consider features like GENDER and AUX for English. The feature GENDER is appropriate for the type *noun_st*, while it is not so for the type *verb_st*. Likewise, the feature AUX is appropriate for the type *verb_st*, but not for the type *noun_st*. Hence, each type is closely associated with a set of appropriate features.

The values of each feature are again restricted as types. The appropriate or permissible values of the feature GENDER are *feminine*, *masculine*, *neuter* etc., but cannot be boolean or binary values, namely + and -. On the other hand, the appropriate values of the feature AUX are only Boolean values.

Consider the following type hierarchy for agreement:[20)]

(123) Agreement type hierarchy



Here, the type *agr-cat* and its left daughter *3sing* are annotated with a set of *appropriate features*, {PERSON,NUMBER} and {GENDER} for their respective type. By such an annotated hierarchy, the construction of well-formed feature structures is strictly constrained. In constructing feature structures, the type *agr-cat* licenses the specification of the features PERSON and NUMBER only, while the type *3sing* allows the specification of the feature GENDER as well as those two inherited features *person* and *number* from its supertype *agr-cat*.

Furthermore, when each feature is introduced, it is also associated with a particular set of its admissible values. The following would be an example:

(124) Admissible Values

features	admissible values
PERSON	{1st, 2nd, 3rd}
NUMBER	{singular, plural}
GENDER	{feminine, masculine, neuter}

These two working together lay a basis for deciding on *well-formed feature structures*. For example, the following would not be a well-formed feature structure:

(125) Ill-formed feature structure

$$\begin{bmatrix} \textit{agr-cat} \\ \text{PERSON } \textit{3rd} \\ \text{TENSE } \textit{singular} \end{bmatrix}$$

20) Copied from the type hierarchy presented in Sag, Wasow and Bender (2003: 492).

The feature TENSE is not appropriate for the type *agr-cat* nor the value *singular* can be admissible for the feature TENSE. Thus, the feature structure above is declared to be ill-formed.

On the other hand, the following is a well-formed feature structure:

(126) Well-formed feature structure

$$\begin{bmatrix} \textit{1sing} \\ \text{PERSON } \textit{1st} \\ \text{NUMBER } \textit{singular} \end{bmatrix}$$

Being a subtype of *agr-cat*, the type *1sing* inherits two of its appropriate features. These two features are then assigned admissible values.

Annex D
(informative)
Formal Semantics of Feature Structure

Annex E
(informative)
Use of Feature Structures in Applications

Feature structures are used in increasingly many current grammatical theories, grammar development platforms and natural-language processing systems. This Annex is intended to help the reader overview the diversity and the usefulness of feature structures in scientific practice.

E.1 Phonological Representation

The earliest use of feature structures may be found in phonology. In phonological representation, for example in Chomsky & Halle (1968), the segment of the consonant /s/ is represented in feature structures as follows: {<consonantal +>, <vocalic −>, <voiced −>, <anterior +>, <coronal +>, <continuant +>, <strident +>}

E.2 Grammar Formalisms or Theories

Almost all the current computational grammar formalisms or theories incorporate feature structures. They use the operation of unification to merge the information encoded in feature structures, hence sometimes the term unification-based grammars. LAG is an exception, adopting a strictly time-linear derivation procedure. The following grammar formalisms or theories share a property that they all use feature structures to represent some important aspects of the grammatical information. (Shieber, 1986).

a) Generalized Phrase Structure Grammar (GPSG): Gazdar, Klein, Pullum and Sag (1985)

b) Head-driven Phrase Structure Grammar (HPSG): Pollard and Sag (1987; 1994) and Sag, Wasow and Bender (2003)

c) Lexical Functional Grammar (LFG): Bresnan (1982)

d) Functional Unification Grammar (FUG): Kay (1983; 1985)

e) Definite Clause Grammar (DCG): Pereira and Warren (1980)

f) Tree Adjoining Grammar (TAG): Vijay-Shanker and Joshi (1988)

g) Left-Associative Grammar (LAG): Hausser (1999)

h) Construction Grammar (CG): Kay (2002)

E.3 Computational Implementations

In parallel to these grammar formalisms and theories, many computational implementations of feature structure have emerged, some of which are listed below.

a) ALE (Carpenter and Penn, 1995)

   Created at the Carnegie Mellon University, the Attribute Logic Engine is one of the oldest systems still being used for the implementation of HPSG grammars. This system is an integrated phrase structure parsing and definite clause logic programming system in which terms are typed feature structures. Originally ALE, based on Kasper-Rounds logic (Kasper and Rounds, 1986) and Carpenter (1992), was created in collaboration between Bob Carpenter and Gerald Penn, but then became the sole work of Gerald Penn and is still being continually upgraded.

b) ALEP (Groenendijk and Simpkins, 1994)

The Advanced Linguistic Engineering Platform, created at the Advanced Information Processing Group at BIM, Belgium, is a platform for developers of linguistic software and provides a distributed, multitasking (Motif-based) environment containing tools for developing grammars and lexicons. The system comes with a unification-based formalism and parsing, generation and transfer components. A facility to implement two-level morphology is also provided. The ALEP formalism has been developed on the basis of the ET 6.1 project (Alshawi et al., 1991). The core of the ET 6.1 formalism follows a rather conservative design and ALEP is very much a traditional rule based grammar development environment. The rules are based on a context free phrase structure backbone with associated types.

c) AMALIA (Wintner, Gabrilovich, and Francez, 1997)

AMALIA is a grammar development system that includes a compiler of grammars (for parsing and generation) to abstract-machine instructions, and an interpreter for the abstract-machine language. The generation compiler inverts input grammars (designed for parsing) to a form more suitable for generation. The compiled grammars are then executed by the interpreter using one control strategy, regardless of whether the grammar is the original or the inverted version. They thus obtain a unified, efficient platform for developing reversible grammars.

d) BABEL (Müller, 1996)

BABEL is a natural-language processing system for German in Prolog. The linguistic theory behind the system is a version of HPSG by Pollard and Sag. It focuses mainly on syntactic issues; word order phenomena in particular. As German is a language with relatively free constituent order, there are problems quite different from those known for the processing of languages like English. BABEL follows the program of linearization grammar begun by Pollard, Levine, and Kasper (1993), Reape (1994), and Kathol (1995).

e) ConTroll (Götz, 1995)

ConTroll is a grammar development system (or a pure logic program) which supports the implementation of current constraint-based theories. It uses strongly typed feature structures as its principal data structure and offers definite relations, universal constraints, and lexical rules to express grammar constraints. ConTroll comes with the graphical interface Xtroll which allows displaying of AVMs and trees, as well as a graphical debugger. The ConTroll-System was based on the logical foundations of HPSG grammars created by Paul King with his Speciate Re-entrant Logic (SRL).

f) CL-ONE (Manandhar, 1994)

CL-ONE extends PROFIT's typed feature structure description language. The system comprises constraint solvers for set descriptions, linear precedence and guarded constraints. Set and linear precedence constraints are defined over constraint terms (cterm), which are internal CL-ONE data structures. Unification of sets differs from standard Prolog terms unification. The system has been designed in the project The Reusability of Grammatical Resources, at the University of Edinburgh and at Universität des Saarlandes, Saarbrücken.

g) CUF (Dörre and Dorna, 1993)

The Comprehensive Unification Formalism (CUF) has been developed at IMS, University of Stuttgart, within the DYANA research project as an implementational tool for grammar development. CUF has been designed to provide mechanisms broadly used in current unification based grammar formalisms such as HPSG or LFG. The main features of CUF are a very general type system and relational dependencies on feature structures. It is a constraint based system, with no specialized components, e.g., for morphology or transfer. It can be seen as a general constraint solver and it does not include a parser or a generator.

h) DATR (Evans and Gazdar, 1996)

Developed at the University of Sussex, DATR is a formal language for representing a restricted class of inheritance networks, permitting both multiple and default inheritance with path/value equations. DATR has been designed specifically for lexical knowledge representation. QDATR is an implementation of the DATR formalism. It supports syntactic analysis, text generation, machine translation and can be used for testing linguistic theories making use of non-monotonic inheritances.

i) DBS (Hausser, 1999, 2001)

Developed by Roland R. Hausser at the University of Erlangen, DBS has been implemented with JAVA to model natural language communication in the form of an artificial agent (talking robot). In DBS, content stored in the context and coded in language has the form of concatenated propositions. Propositional content is defined as a set of features structures, called *proplets*. Dealing with various English constructions, the system demonstrates how natural language communication is successfully carried out at both the hearer and speaker modes by various processes of navigation through a semantic database. Based on LAG, navigation proceeds in a strictly time-linear order, while operating on feature structures without unification.

j) FGW

The Functional Grammar Workbench system uses Functional Grammar (FG) as a model for linguistic generation. The user must provide a grammar and lexicon for the target language and FGW is able to produce surface strings from predicate-argument formulas.

k) Grammar Writer's Workbench for LFG (Kaplan and Maxwell, 1996)

The Xerox LFG Grammar Writer's Workbench is a complete parsing implementation of the LFG syntactic formalism, including various features introduced since the original Kaplan and Bresnan (1982) paper (functional uncertainty, functional precedence, generalization for coordination, multiple projections, etc.) It includes a very rich c-structure rule notation, plus various kinds of abbreviatory devices (parameterized templates, macros, etc.). It does not directly implement recent proposals for lexical mapping theory, although templates can be used to simulate some of its effects. The system has an elaborate mouse-driven interface for displaying various grammatical structures and substructures – the idea is to help a linguist understand and debug a grammar without having to comprehend the details of specific processing algorithms.

The workbench runs on most Unix systems (Sun, DEC, HP, $\cdots$) and under DOS on PC's, although most of our experience is on Sun's. It does not have a teletype interface–it only runs as a graphical program. It requires at least 16MB of ram on UNIX and 8MB under DOS, plus 40 or more MB of disk (for program storage and swapping).

l) GULP (Covington, 1989)

The Graph Unification Logic Programming is a preprocessor for handling feature structures in Prolog programs. It solves a pesky problem with Prolog, i.e., the lack of a good way to represent feature structures in which features are identified by name rather than position.

m) LFG-PC and LFGW

Avery Andrews developed a small LFG system that runs on PC's (XT's, in fact), that is basically oriented towards producing small fragments to illustrate aspects of grammatical analysis in basic LFG. It uses some nonstandard notations (mostly in the interests of brevity), and is missing some things that really ought to be there (like a properly working treatment of long-distance dependencies), but it does have a primitive morphological component, something that the author has found essential for pedagogical use (even tiny fragments tend to have so many inflected word forms that typing in the lexicon is a major deterrent to serious use). This system is currently available in a (non-Windows) PC-version (LFG-PC) and a Windows version (LFGW).

n) LIFE (Aït-Kaci and Podelski, 1993)

LIFE is an experimental programming language proposing to integrate three orthogonal programming paradigms proven useful for symbolic computation. From the programmer's standpoint, it may be perceived as a language supporting logic programming, functional programming, and object-oriented programming. From a formal perspective, it may be seen as an instance (or rather a composition of three instances) of a Constraint Logic Programming scheme due to Hohfeld and Smolka (1988) refining that of Jaffar and Lassez (1986). LIFE's object unification is seen as constraint-solving over specific domains. LIFE is built on work by Smolka and Rounds to develop type-theoretic, logical, and algebraic renditions of a calculus of order-sorted feature approximations.

o) LiLFeS (Makino, et al., 2000)

The LiLFeS system provides a programming environment with efficient processing of typed feature structures. It is a Prolog-like logic-programming language integrating feature-structure descriptions and definite-clause programs by expressing predicate arguments of Prolog in typed feature structures instead of first-order terms. A user who already knows Prolog syntax can easily understand the LiLFeS syntax. The core

engine of the LiLFeS system is developed in C++ as an implementation of Warren's Abstract Machine (Aït-Kaci, 1991) and Abstract Machine for Attribute-Value Logics (Carpenter and Qu, 1995) so as to maximize efficiency.

Large-scale systems such as grammars, parsers and grammar-development tools have been developed in LiLFeS. For example, a wide-coverage treebank grammar of HPSG for analyzing both syntactic structures and predicate-argument relations is developed by a grammar-development tool implemented in LiLFeS (Miyao, et al., 2004). The LiLFeS system also provides a C++ class library for manipulating typed feature structures and their databases. Enju (Tsuruoka, et al., 2004), an efficient probabilistic HPSG parser for the English treebank grammar, is implemented in C++ using this library.

p) LKB (Copestake, 2002)

The Linguistic Knowledge Building (LKB) system is a grammar and lexicon development environment for use with constraint-based linguistic formalisms. The LKB software is distributed by the LinGO initiative, a loosely-organized consortium of research groups working on unification-based grammars and processing schemes.

q) Malaga (Beutel, 1997)

Malaga is a development environment for Left-Associative Grammars (LAGs). The LAG is a formalism that describes the analysis and generation of words and sentences strictly from the left to the right. It has been introduced by Roland Hausser. The Grammars for morphological and/or syntactical analysis are written in a Pascal-ish language, but it uses a powerful dynamic typing system with attribute-value structures and lists that can be nested with a collection of newly-defined operators and standard functions. It contains constructs to branch the program execution into parallel paths when a grammatical ambiguity is encountered. The grammars are compiled into virtual code that is executed by an interpreter. The toolkit comprises parser generators for the development of morphological and syntactic parsers (including sample grammars), source-code lexicon/grammar debuggers (with Emacs modes), and an optional visualization tool to display derivation paths and categorial values using GTK+. The toolkit is written in ANSI/ISO-compliant C and its source code is freely available. It can be used and distributed under the terms of the GNU General Public License. It should work out-of-the-box on most POSIX systems. Porting to other platforms should be easy. A Windows port has already been made publicly available.

r) PAGE (Krieger and Schäfer, 1994a; 1994b)

Platform for Advanced Grammar Engineering (PAGE) is a grammar development environment and run-time system which evolved from the DISCO project (Uszkoreit et al., 1994) at the German Research Center for Artificial Intelligence (DFKI). The system has been designed to facilitate development of grammatical resources based on typed feature logics. It consists of several specialized modules which provide mechanisms that can be used to directly implement notions of HPSG, LFG and other grammar formalisms.

PAGE provides a general type system (TDL), a feature constraint solver (UDiNe), a chart parser, a feature structure editor (Fegramed), a chart display to visualize type hierarchies and an ASCII-based command shell. In the remainder, we will concentrate mostly on the TDL module which is best documented and constitutes the fundamental part of the environment. The description of TDL is based on Krieger and Schäfer (1994a; 1994b).

s) PET (Callmeier, 2000)

The PET system for efficient processing of unification-based grammars is being developed at the Department of Computational Linguistics at Saarland University by Ulrich Callmeier and others. The system is developed in C so as to maximize efficiency, and employs a quick check technique for unifiability. It is the most efficient parser in the benchmarks of parsers for the LinGO grammar in 1999 (Oepen, et al., 2000).

t) ProFIT (Erbach, 1994)

Prolog with Features, Inheritance and Templates (ProFIT), developed at Universität des Saarlandes, Saarbrücken, is an extension of Prolog. ProFIT programs consist of data type declarations and Prolog definite clauses. ProFIT provides mechanisms to declare an inheritance hierarchy and define feature structures. Templates are widely used to simplify descriptions, encode constraint-like conditions and abstract over complex data structures.

u) TDL (Krieger and Schäfer, 1994a; 1994b)

Type Description Language (TDL) is a typed feature-based language, which is specifically designed to support highly lexicalized grammar theories, such as HPSG,FUG, or CUG. TDL offers the possibility to define (possibly recursive) types, consisting of type and feature constraints over the Boolean connectives AND, OR, and NOT, where the types are arranged in a subsumption hierarchy. TDL distinguishes between avm types (open-world reasoning) and sort types (closed-world reasoning) and allows the declaration of partitions and incompatible types. Working with partially and fully expanded types as well as with undefined types is possible, both at definition and at run time. TDL is incremental in that it allows the redefinition of types. Efficient reasoning is accomplished through specialized modules. Large grammars and lexicons for English, German, and Japanese are available.

v) TFS (Emele and Zajac, 1990; Emele, 1993)

The Typed Feature Structure (TFS) representation formalism was developed in the German Polygloss project at IMS, University of Stuttgart. The inheritance-based constraint architecture embodied in the TFS system integrates two computational paradigms: the object-oriented approach offers complex, recursive, possibly nested, record objects represented as typed feature structures with attribute-value restrictions and (in)equality constraints, and multiple inheritance; the relational programming approach offers declarativity, logical variables, non-determinism with backtracking, and existential query evaluation. The constraint-based properties of the TFS formalism are fully exploited in the Delis lexicon model and pertaining tools. These include corpus search and support for the major steps and types of activity in lexicon building: creation, population and modification of lexical models; exportation towards formats for both NLP and human use.

w) TRALE

The TRALE system is a combination of the experience from the development and application of ALE and ConTroll. The idea is to combine the advantage of efficiency which ALE offers with the original concept of ConTroll, to submit a depiction, as true to form as possible, of theoretical HPSG grammars into computational implementation. The continuing work on TRALE, which focuses on the requirements of current linguistic research of HPSG, is motivated by this depiction. Our objective is to provide task-specific solutions for typical, computationally expensive mathematical constructs of HPSG grammars. It should then become possible to implement even theoretically well grounded grammars efficiently, whose computational treatment goes beyond the capabilities of a pure, general constraint solving system such as ConTroll, without forcing linguists to renounce their specific assumptions about the structure of language.

Bibliography

[1] Hassan Aït-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction*. The MIT Press, 1991.

[2] Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16:195–234, 1993.

[3] Hiyan Alshawi, David Carter, Bjorn Gamback, and Manny Rayner. Translation by quasi logical form transfer. In *Proc. 29th Annual Mtg. Assoc. Computational Linguistics*, Berkeley, CA, 1991.

[4] Björn Beutel. MALAGA 4.0. Online-documentation, Abteilung für linguistische Informatik, Friedrich Alexander Universität, Erlangen Nürnberg, Germany, 1997.

[5] Joan W. Bresnan, editor. *The Mental Representation of Grammatical Relations*. MIT Press, Cambridge, Massachusetts, 1982.

[6] Bob Carpenter. *The Logic of Typed Feature Structures*. Cambridge University Press, Cambridge, 1992.

[7] Bob Carpenter and Gerard Penn. Compiling typed attribute-value logic grammars. In H. Bunt and M. Tomita, editors, *Current Issues in Parsing Technologies*, volume 2. Kluwer, 1995.

[8] Bob Carpenter and Yan Qu. An abstract machine for attribute-value logics. In *Proc. of IWPT-95*, pages 59–70, 1995.

[9] Noam Chomsky and Morris Halle. *The Sound Pattern of English*. Harper & Row, New York, 1968.

[10] Ann Copestake. *Implementing Typed Feature Structure Grammars*. CSLI Publications, Stanford, 2002.

[11] Michael A. Covington. GULP 2.0: An extension of prolog for unification-based grammar. Research report ai-1989-01, artificial intelligence programs, The University of Georgia, 1989.

[12] Jochen Dörre and Michael Dorna. CUF: A formalism for linguistic knowledge representation. In Jochen Dörre, editor, *Computational aspects of constraint based linguistic descriptions I, DYANA-2 Deliverable R1.2.A*, pages 1–22. Universität Stuttgart, 1993.

[13] Martin Emele. TFS — the typed feature structure representation formalism. In Hans Uszkoreit, editor, *Proc. of the EAGLES Workshop on Implemented formalisms*. DFKI, 1993.

[14] Martin Emele and Rémi Zajac. Typed unification grammars. In *Proc. of the 13th International Conference on Computational Linguistics*, 1990.

[15] Gregor Erbach. ProFit — prolog with features, inheritance, and templates. CLAUS-report no. 42, Saarbrücken: Universität des Saarlandes, 1994.

[16] Roger Evans and Gerald Gazdar. DATR: A language for lexical knowledge representation. *Computational Linguistics*, 22:167–216, 1996.

[17] Gerald Gazdar, Ewan Klein, Geoffrey Pullum, and Ivan Sag. *Generalized Phrase Structure Grammar*. Harvard University Press, Cambridge, MA, 1985.

[18] Thilo Götz and Walt Detmar Meurers. Compiling HPSG type constraints into definite clause programs. In *Proc. of the 33rd Annual Meeting of the Association for Computational Linguistics*, 1995.

[19] Roland H. Hausser. *Foundations of Computational Linguistics: Human-Computer Communication in Natural Language*. Springer-Verlag, Berlin, 2nd edition, 1999.

[20] Roland R. Hausser. Database semantics for natural language. *Artificial Intelligence*, 130(1):27–74, 2001.

[21] Markus Hohfeld and Gert Smolka. Definite relations over constraint languages. LILOG report 53, IWBS, IBM Deutschland, 1988.

[22] Joxan Jaffer and Jean-Lous Lassez. Constraint logic programming. In *Proc. of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of programming languages*, pages 111–119, Munich, 1987.

[23] Mark Johnson. *Attribute-Value Logic and the Theory of Grammar*. CSLI, Stanford, 1988. CSLI Lecture Notes 16.

[24] Ronald M. Kaplan and Joan W. Bresnan. Lexical-functional grammar: A formal system for grammatical representation. In Joan W. Bresnan, editor, *The Mental Representation of Grammatical Relations*, pages 173–281. MIT Press, 1982.

[25] Ronald M. Kaplan and John T. Maxwell III. *Grammar Writer's Workbench*. Xerox Corporation, 1996. ftp://ftp.parc.xerox.com/pub/lfg/lfgmanual.ps.

[26] Robert T. Kasper and William C. Rounds. A logical semantics for feature structures. In *Proc. of the 24th Annual Meeting of the Association for Computational Linguistics*, pages 235–242, 1986.

[27] Andreas Kathol. *Linearization-Based German Syntax*. PhD thesis, Ohio State University, 1995.

[28] Martin Kay. Unification. In Michael Rosner and Roderick Johnson, editors, *Computational Linguistics and Formal Semantics*, pages 1–30. Cambridge University Press, Cambridge, 1992.

[29] Paul Kay. An informal sketch of a formal architecture for construction grammar. *Grammars*, 5:1–19, 2002.

[30] Hans-Ulrich Krieger and Ulrich Schäfer. TDL — a type description language for HPSG, part 1: Overview. Technical report RR-94-37, DFKI, 11 1994.

[31] Hans-Ulrich Krieger and Ulrich Schäfer. TDL — a type description language for HPSG, part 2: User guide. Technical report d-94-14, DFKI, 11 1994.

[32] Terence D. Langendoen and Gary F. Simons. A rationale for the TEI recommendations for feature-structure markup. *Computers and the Humanities*, 29:191–209, 1995.

[33] Takaki Makino, Yusuke Miyao, Kentaro Torisawa, and Jun ichi Tsujii. Native-code compilation of feature structures. In Stephan Oepen, Dan Flickinger, Jun ichi Tsujii, and Hans Uszkoreit, editors, *Collaborative Language Engineering: A Case Study in Efficient Grammar-based Processing*. CSLI Publications, 2000.

[34] Suresh Manandhar. *User's Guide for CL-ONE*. Centre for Cognitive Science, University of Edinburgh, Scotland, 1994.

[35] Yusuke Miyao, Takashi Ninomiya, and Jun'ichi Tsujii. Corpus-oriented grammar development for acquiring a head-driven phrase structure grammar from the penn treebank. In *Proc. of IJCNLP-04*, 2004.

[36] Stefan Müller. The babel-system – an HPSG prolog implementation. In *Proceedings of the Fourth International Conference on the Practical Application of Prolog*, pages 263–277, London, 1996. http://www.cl.uni-bremen.de/~stefan/Pub/babel.html.

[37] Stephan Oepen, Dan Flickinger, Jun ichi Tsujii, and Hans Uszkoreit. *Collaborative Language Engineering: A Case Study in Efficient Grammar-based Processing*. CSLI Publications, 2000.

[38] Fernando C. N. Pereira. Grammars and logics of partial information. SRI International Technical Note 420, SRI International, Menlo Park, CA, 1987.

[39] Fernando C. N. Pereira and David H. D. Warren. Definite clause grammars for language analysis — a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.

[40] Carl Pollard, Robert Levine, and Robert Kasper. Studies in constituent ordering: Toward a theory of linearization in head-driven phrase structure grammar, 1994. Grant Proposal to the National Science Foundation.

[41] Carl J. Pollard and M. Andrew Moshier. Unifying partial descriptions of sets. In Philip P. Hanson, editor, *Information, Language, and Cognition*, pages 285–322. The University of British Columbia Press, Vancouver, 1990.

[42] Carl J. Pollard and Ivan A. Sag. *Fundamentals*, volume 1 of *Information-based Syntax and Semantics*. CSLI, Stanford, 1987. CSLI Lecture Notes 13.

[43] Carl J. Pollard and Ivan A. Sag. *Head-driven Phrase Structure Grammar*. The University of Chicago Press, Chicago, 1994.

[44] Mike Reape. Domain union and word order variation in german. In et al. J. Nerbonne, editor, *German in Head-Driven Phrase Structure Grammar*, pages 151–198. CSLI Publications, 1994.

[45] Ivan A. Sag and Thomas Wasow. *Syntactic Theory: A Formal Introduction*. CSLI Publications, Stanford, 1999.

[46] Ivan A. Sag, Thomas Wasow, and Emily M. Bender. *Syntactic Theory: A Formal Introduction*. CSLI Publications, Stanford, 2nd edition, 2003.

[47] Stuart M. Shieber. *An Introduction to Unification-Based Approaches to Grammar*. CSLI, Stanford, 1986. CSLI Lecture Notes 4.

[48] Neil Simpkins and Marius Groenendijk. The ALEP project. Technical report, Cray Systems/CEC, Luxembourg, 1994.

[49] Yoshimasa Tsuruoka, Yusuke Miyao, and Jun'ichi Tsujii. Towards efficient probabilistic HPSG parsing: integrating semantic and syntactic preference to guide the parsing. In *Proc. of IJCNLP-04 Workshop: Beyond Shallow Analyses - Formalisms and Statistical Modeling for Deep Analyses*, 2004.

[50] Callmeier Ulrich. Pre-processing and encoding techniques in PET. In Stephan Oepen, Dan Flickinger, Jun ichi Tsujii, and Hans Uszkoreit, editors, *Collaborative Language Engineering: A Case Study in Efficient Grammar-based Processing*, pages 127–143. CSLI Publications, 2000.

[51] K. Vijay-Shanker and Aravind K. Joshi. Feature-structure based tree adjoining grammar. In *Proceedings of 12nd International Conference on Computational Linguistics(COLING'88)*, 1988.

[52] Shuly Wintner, Evgeniy Gabrilovich, and Nissim Francez. Amalia – a unified platform for parsing and generation. In *Proceedings of Recent Advances in Natural Language Programming (RANLP97)*, pages 135–142, 1997.