ISO TC 37/SC 4 N033 Rev. 11
2004-07-10

Language Resource Management

Descriptors and Mechanisms for Language Resources

Title:	Draft - Language Resource Management – Feature Structures Part 1: Feature Structure Representation
Editor(s):	Kiyong Lee
Source:	WG1
Project number:	ISO 24610-1 - This reference will supersede all previous one and remain attached as working ref along the duration of the project.
Status:	pre-DIS
Date:	2004-07-10
Agenda/Action:	For review
References:	WG1 N17; WG1 N23; TEI P5

Mr. Key-Sun Choi - SC4 Secretary
KORTERM, KAIST
373-1 Guseong-dong Yuseong-gu, Daejeon 305-701, Korea
Tel: +82 42 869 35 25 - Fax: +82 42 869 87 90
kschoi@cs.kaist.ac.kr - http://tc37sc4.org

Warning

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard. Recipients of this document are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Warning

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard. Recipients of this document are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Copyright notice

Contents

Harry was asked to draft this section.

Annex C is just a draft which need be revised extensively. Byongrae compiled the drat, and Thierry and Lou are expected to complete it.

Foreword

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. Annexes A and B are normative, while Annexes C and D 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 *P5*. This work should lead to both a thorough revision of the guidelines and the production of an ISO standard on Feature Structure Representation and Declaration.

This standard is organized in two separate main parts. The first part 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 aims at providing 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.

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 components. This international standard provides a format to represent, store or exchange feature structures in natural language applications, both for the purpose of annotation or 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

The reference to XML in ISO 16642 is the following: 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/REC$_{XML}$.
ISO/IEC 639, Information technology – ISO 639: 1988, Code for the representation of names of languages.

ISO 639-2: 1998, Code for the representation of names and languages – part 2: Alpha-3 code.

ISO/IEC 646: 1991, Information technology – ISO 7-bit coded character set for information interchange.

ISO 3166-1: 1997, Code for the representation of names of countries and their subdivisions – Part 1: Country codes

ISO 8601: 1988, Data elements and interchange formats - Information interchange – Representation of dates and times.

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

ISO/IEC 10646-1: 2000, Information technology - Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and basic multilingual plane.

ISO 12620, Computer applications in terminology – Data categories.

3 Terms and Definitions

alternation

Unlike the ordinary notion of disjunction in Boolean logic, alternation in this standard applies to feature values only and is understood as an exclusive disjunction, represented by

a bar | in the AVM notation. Given a feature specification of the form $feature : value0 \mid value1$, FEATURE has a value either $value0$ or $value1$, but not both.

atomic value

In a feature structure, the value of each feature is either atomic or complex. An atomic value is some primitive type of object that has no internal feature specification or value structure. See **complex value**.

attribute

In some feature-based grammar formalisms, the term *attribute* sometimes has been used to refer to a feature in feature structure, as shown by the use of the term *Attribute-Value Matrix* or AVM that represents feature structure.

But in XML, it has a technical use as a qualifier indicating a certain property of a given textual element. It differs from its type which is specified with a generic identifier or from its content which is limited by start and end-tags. Attributes are identified with *name="value"* pairs only on start-tags as in: <ELEMENT *name="value"*>. They usually modify the ELEMENT with the content specified between the start and end tags.

In this standard, the use of the term *attribute* is restricted to its technical sense in XML.

attribute-value matrix

AVM
A very common notation in a matrix form which represents a feature structure consisting of pairs of an attribute, namely feature, and its value.
Note: The acronym AVM stands for "Attribute-Value Matrix" where each row represents a pair of a feature and its value, separated by a colon (:), space ( ) or the equality sign (=).

bag

See the term multiset.

binary value

There are two binary values: *plus* and *minus* or *true* and *false*. In XML, they may be represented as < plus/> and < minus/>.

boxed label

Label in a box like $\boxed{1}$ or $\boxed{A}$ which is used for marking structure sharing in an AVM.
Note: The index is normally marked with an integer, but it can be any alpha-numeric symbol that can be used as a coreferential index.

boolean operator

Negation, conjunction, disjunction and conditional are each called *boolean operator* on truth values or propositions.

boolean value

In bivalent logic, two boolean values, truth and falsity, are admitted. They are often represented as 1 and 0, + and -, or *positive* and *negative* as polarity values.

compatibility

Two feature structures are compatible if and only if none of the features that they have in common has a conflicting value. On the other hand, two incompatible feature structures contain at least one identical feature which has a conflicting value.

complex value

The value of a feature in a feature structure can be complex, not an atomic value. The complex value could be a feature structure, a list, a set, or a bag.

concatenation

Two feature list values may be concatenated into a single list, represented by the concatenation operator $\oplus$.

directed acyclic graph

DAG
graph on which each node, except for the terminal ones, points to other nodes or at least one other node, but it disallows any path that points to itself
Note: A feature structure is often represented by a DAG.

distinctive feature

Feature that distinguishes an object from others. The sound segments /p/ and /b/ in English, for instance, are distinguished from each other in the specification of a feature VOICING: one is *voiceless* and the other *voiced*. This feature is then called *distinctive feature*.

element

An element in XML normally consists of a start-tag, content and end-tag, as shown schematically by: <ELEMENT_TYPE> CONTENT </ELEMENT_TYPE>.

empty element

In XML, an *empty element* is an element that has no content. It is either represented as in the form of <ELEMENT> </ ELEMENT> or simply as <ELEMENT />. Unlike the empty feature structure which is unique, there can be many different empty elements with different names or identifiers or even with attribute specifications.

empty path, the

Path corresponding to the root node of a graph which represents the empty feature structure without any feature specification.
Note: It is represented as a single dot, possibly labelled with the name of a type, on a graph. It may also be represented as [ ] or [$^{\tau}$] labelled with the name of a type τ in the AVM notation.

empty feature structure, the

Feature structure without any feature specification.

eXtensible Markup Language

XML
One of the conventional standards for a set of symbols and rules that are used for marking the structure of a text and the characteristics, properties and attributes of their components or basic elements. This markup language is an official recommendation of the World Wide Web Consortium, W3C, for interchanging data over the Internet.

feature

Property of an object being described. By having an appropriate value for the described object, it constitutes part of a feature specification.
For example, "NUMBER" is a feature, a pair "NUBMER *singular*" is a feature specification, and "*singular*" is a feature value.

feature specification

Assignment of a particular value to a feature in a feature structure.

feature structure

A set of feature specifications which carry (partial) information about some object being described by assigning a value to its features.
Note: The feature structure is defined in set-theoretic terms as a partial function from features to values.

feature structure description

A feature structure is often described in a declarative manner through some description language. This should not be confused with *feature system declaration* FSD which describes the set of all valid feature structures.

generalization

While unification enriches information, *generalization* captures common features from various feature structures. It is a binary *total* operation on two feature structures. Just like *intersection* that operates on sets, it only picks up common feature specifications and puts them into a resulting feature structure.

Here is a formal definition: Given a lattice of feature structures in the subsumption order, the generalization of typed feature structures FS_1 and FS_2 is formally defined as the greatest lower bound of FS_1 and FS_2 in the subsumption lattice. It is then represented as $F_1 \sqcap F_2$.

graph notation

A single-rooted, labelled and directed graph is often used to represent a feature structure. Each graph representing a feature structure starts with a particular single node called *the root*. From the root, more than one arc, each of which is labelled with the name of a feature, may branch out to other nodes. These nodes may each terminate with an atomic value or some of them may again branch out to other nodes. For a typed feature structure, each node including the root is labelled with the name of a type.

identity element

The empty feature structure is an identity element of the operation called *unification* on feature structures, since it yields the identical result when unified with any other feature structure just as the number 0 is an identity element for the algebraic operation called *addition* on natural numbers.

markup

The process of adding formatting or other processing commands to a text in order to provide information about the logical as well as physical structure of its content.

multiple inheritance

Some types may inherit properties from more than one supertypes. For example, a whale is both a mammal and a fish, thus inheriting properties from both of them.

multiset

A multiset, represented as {. .} or $\{\ \}_m$, is a collection of objects that, unlike an ordinary set, allows the occurrence of identical elements in it. Hence, {a, a, b} is not the same as {a, b}. On the other hand, like an ordinary set, its members are not ordered. A multiset is often called *bag*.

multivalued feature

Some features, especially used in linguistics, take sets, multisets or lists as values. These are called *multivalued features*. The feature COMPS, for instance, takes a list of *complements* like object and indirect object as value. Note they do not violate the single value condition of a feature specification in feature structure, which is often defined as a function in mathematical terms.

negation

In this standard, negation applies to feature values only and is not understood as a truth function as in ordinary bivalent logics. It is understood more or less in a set-theoretic sense. If the value of a feature is atomic, then its negation is understood as referring to one of the values in the complement of a unit set which consists of that particular value. If the value of a feature is complex, namely a feature structure, then its negation is any of the feature structures which are incompatible to it.

path

Sequence of feature names which, in the graph notation, label each of the arcs, starting from the root.
The notion of *path* can also be extended in the same manner to other notations.

reentrancy

Structure sharing in a feature structure.
It may be represented in the graph notation as two or more paths pointing to the same node. These paths are then called *equivalent*, having the same value. As a result, these two or more paths leading to that common node share their values. In the AVM notation, reentrancy is conventionally marked by a boxed integer or alphabetic symbol like $\boxed{3}$ by tagging it to the left of the feature structure or the type name of that node and also at the place of the value being shared by the other paths without copying the shared feature structure or feature value. See **shared value**.

root, the

The topmost node on a graph or an (upside-down) tree that has no ancestors.

shared value

Feature value shared by two or more features in a feature structure.
In graph notation, a node to which two or more paths merge represents the value shared by the paths. In matrix notation, the shared value is represented by an identical boxed alpha-numeric index. See **reentrancy** and **structure sharing**.

structure sharing

A feature structure in which one or more feature-values are shared or re-used. See **reentrancy**.

subsumption

A reflexive, anti-symmetric and transitive relation between two feature structures.
A feature structure A is said to subsume a feature structure B, formally represented as $A \sqsubseteq B$, if A is not more informative than B, or A contains a subset of the feature specifications in B.

tag

The feature structure in XML notation is surrounded in <fs> tags, and each of its features with <f> tags.
In some grammar formalisms like HPSG, this term is used to referred to that index, often represented as a boxed integer, which indicates structure sharing in feature structure.

In this standard, the term *tag* is reserved for its use in the sense used in XML only. It is a symbol delimiting a logical element inside a document.

For each element, there are two types of tags: star-tags and end-tags. A start-tag opens with the opening or left angled bracket < followed by the name of an element and other attribute specifications and closes with the closing or right angled bracket >. An end-tag opens with the opening angled bracket with a slash usually followed by the name of the same element and closes with the closing or right angled bracket >.

type, feature structure

Elements of a domain can be sorted into classes in a structured way, based on similarities of properties. These classes are called *types*.
Note: In linguistics, for instance, class names like *phrase*, *word*, *pos* (parts of speech), *noun*, and *verb* are often taken as types. In grammar formalisms based on typed feature structure, each feature structure is assigned a type and each atomic value is treated as a type.

type constraint

The construction of well-formed feature structures are constrained by some type inheritance hierarchy. A feature structure of some subtype must inherit all the constrains laid on its supertypes. For example, a verb carries information on valence and so its subtypes like transitive or intransitive verbs must also be able to provide some information about valence with appropriate feature specifications.

type inheritance hierarchy

Types are ordered in some hierarchical order so that objects of a subtype inherit properties of their supertypes. In linguistics, these hierarchies are often used to organize linguistic descriptions, especially lexical information.
The type inheritance hierarchy is defined by assuming a finite set BF 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.

typed feature structure

Feature structure that is labelled by the name of a type. In the graph notation, each node on a graph is labelled with the name of a type.
A formal definition of typed feature structure can thus be given as follows:[1]

[1]Slightly modified from Carpenter (1992: 36).

Given a finite set of **Features** and a finite set of **Types**, a typed feature structure is a tuple $\mathcal{TFS} = <$ **Nodes**$, r, \theta, \delta >$ such that

i. **Nodes** is a finite set of nodes.

ii. r is a unique member of **Nodes** called *the root*.

iii. θ is a total function that maps **Nodes** to **Types**.

iv. δ is a partial function from **Features**$\times$**Nodes** into **Nodes**.

typing

By *typing*, each feature structure is assigned a particular type. Each feature specification with a particular value is then constrained by this typing. 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*.

unification

A binary operation on feature structures that combine two compatible feature structures into one representing exactly the information contained in the feature structures being unified. Here is a 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.

value, feature

A feature in a feature structure is specified with a particular value. Values can be: atomic, complex, or multivalued. See *atomic value*, *complex value* and *multivalued feature*.

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 of them. 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 to be of specific *types*, and also a set of restrictions to be placed on the values for particular features, by means of *feature system declarations*, which are properly discussed in the second part of this standard. Such restrictions provide the basis for at least partial validation of the feature-structure encodings that are used.[2]

[2]See illustration examples in non-normative Annex A.

4.2 Use of Feature Structures

Feature structures may be understood as providing *partial information* about some object which is described by specifying *values* of some of its features. Suppose we are describing a female employee named Sandy Jones who is 30 years old. We can then talk about at least that person's sex, name and age in a succinct manner by assigning a value to each of these three features of hers. These pieces of information can be put into a simple set notation, as in:

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

The use of feature structures can easily be extended to linguistic descriptions, too. The phoneme /p/ in English, for instance, can be analyzed as a complex of its distinctive features. It can be partially described as a consonantal, anterior, voiceless, non-continuant or stop sound segment. By introducing the boolean values *plus*(+) and *minus*(-), these features can then be listed as a set consisting of pairs of a feature and its value:

(2) Distinctive features of the sound segment /p/
{<CONSONANTAL, + >, <ANTERIOR, + >, <VOICED, - >, <CONTINUANT, - >}

As a result, it can be distinguished from other phonemes in exactly what aspect it differs from them. It differs from the phoneme /b/ in VOICING, while it differs from the phoneme /k/ in their articulatory positions: one is articulated at the anterior, namely lip or alveolar area of the mouth and the other at the non-anterior part, namely back of the oral cavity.

This feature analysis can be extended to the description of other linguistic entities or structures. Consider the 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 building lexicons, especially related to computational work. Feature structures are used to describe and model linguistic entities and phenomena by analyzing them as complexes of their properties. For this purpose, it is considered necessary to specify some of the formal properties of feature structures and ways of representing them in a systematic manner.

4.3 Basic Concepts

Feature structure 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 thus 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) 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 $\mathbf{FeatVal} = \mathbf{AtomVal} \cup \mathbf{FS}$.

In a feature structure, feature values are either atomic or complex. Atomic values are linguistic entities without internal structure, while complex values are themselves feature structures.

Here is a linguistic example. The part of speech feature, abbreviated as POS, takes the name of an atomic category like *verb* as value. The agreement feature AGR in English, on the other hand, takes a complex value in the form of a feature structure where the features PERSON and NUMBER are appropriately specified. The word 'loves', for instance, is here analyzed as having the value of its POS feature specified as *verb* and the value of its AGR feature specified by a feature structure consisting of two feature specifications: one specifies the value of PERSON with *3rd* and the other, the value of NUMBER with *singular*.

4.4 Notations

As a list of feature-value pairs, the general structure of a feature structure is simple. But its internal structure becomes complex when a feature structure contains a complex feature that takes a value which is itself a feature structure or a multivalued feature that takes a value which is a list, set or multiset which consists of atomic values or again of complex values that are themselves feature structures. As a result, a feature structure may recursively be embedded into another feature structure, creating a feature structure with really complex internal structure. Hence, to represent them in a manner easy to understand, we need mathematically precise, well defined notations.

Just as the notion of feature structure may be conceived in different ways, there are a few different ways of representing it. Here are the three notations that are most commonly used: a graph, a matrix and an XML-based notation. Graphs are for mathematical discourses, matrices for linguistic descriptions, and XML notations for computational implementation.

4.4.1 Graph Notation

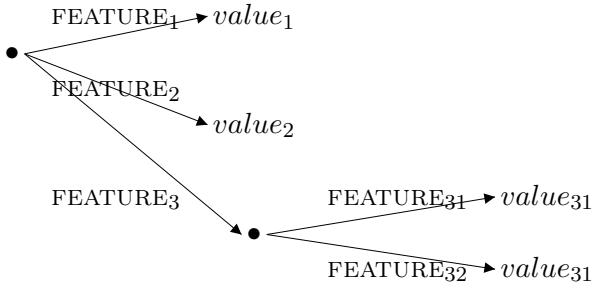
For conceptual coherence and mathematical elegance, feature structures are often represented as labelled directed graphs with a single root.[3]

[3]This graph can be either (1) *acyclic*, thus allowing the acronym DAG for feature structures or (2)*cyclic* for handling cases like the Liar's paradox.

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.

(5) Feature structure in graph notation



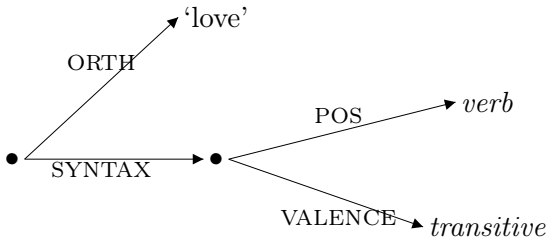
In this graph, the two features FEATURE$_1$ and FEATURE$_2$ are atomic-valued, taking *value*$_1$ and *value*$_2$ on the terminal nodes respectively as their value. The feature FEATURE$_3$ is, however, complex-valued, for it takes as its value the feature structure which is represented by the two arcs 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

DAGs are of course by definition not cyclical. And feature structures are usually represented as DAGs; but there have been some suggestions that cyclical feature structures should be introduced to model some linguistic phenomena.

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.[4]

(7) Matrix notation

$$\begin{bmatrix} \text{FEATURE}_1 & value_1 \\ \text{FEATURE}_2 & value_2 \\ \text{FEATURE}_3 & \begin{bmatrix} \text{FEATURE}_{31} & value_{31} \\ \text{FEATURE}_{32} & value_{32} \\ \multicolumn{2}{c}{\ldots} \\ \text{FEATURE}_{3k} & value_{3k} \end{bmatrix} \\ \multicolumn{2}{c}{\ldots} \\ \text{FEATURE}_n & value_n \end{bmatrix}$$

Each row in the square bracket with a FEATURE name followed by its *value* name represents a feature specification in a feature structure.[5] 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

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

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

[4] The term 'feature' is sometimes called *attribute*.

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

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">ambiguity</word>`

This is the XML way of representing an occurrence of the element *word*, the content of which is the string "ambiguity". 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, 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*. [6]

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:

[6] 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.

(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>
```

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="str">love</value>
     </f>
     <f name="pos">
      <value type="sym">noun</value>
    </f>
    </fs>
```

```
(b) <fs>
    <f name="orth"><str>love</str></f>
    <f name="pos"><sym 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, **str** to enclose a string, **sym** to denote a symbol, **bool** to denote a Boolean 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"><str>love</str></f>
    <f name="syntax">
        <fs>
        <f name="pos"><sym value="verb"/></f>
        <f name="valence"><sym value="transitive"/></f>
        </fs>
    </f>
</fs>
```

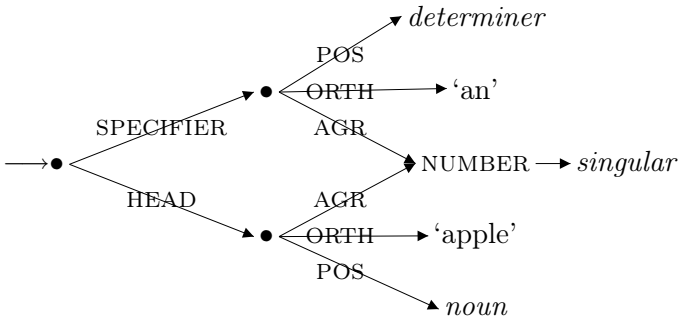
The feature structure in XML notation is surrounded in <fs> tags, and each specification of its features with <f> tags. Note that despite the apparent difference between the representations (6), (8) and (15), the information contained in each structure corresponds quite systematically with that in other structures. For example, arc labels in graph notation like ORTH and SYNTAX are now represented by <f> tags with appropriate names.[7]

4.5 Structure Sharing

The graphic notation can clearly represent *structure sharing* also called *reentrancy*. Consider the following:

(16) Merging paths in graph notation

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



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.

Such structure sharing can also be represented in an AVM.

(17) Structure sharing in AVM notation

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

The two occurrences of the identical boxed integer, namely $\boxed{1}$, in the above representation show that those two occurrences of the feature AGR share the same value, namely *singular*, for the feature NUMBER.

The XML-based notation can also handle such sharing or reentrancy by introducing a new element `var` with attribute `label` to name the sharing point. This element may embed a value or none in the case of a most general value.

(18) Structure sharing in XML notation

```
<fs>
  <f name="specifier">
     <fs>
        <f name="agr">
          <var label="@1">
           <fs>
             <f name="number"><sym value="singular"/></f>
           </fs>
          </var>
        </f>
        <f name="pos"><sym value="determiner"/></f>
     </fs>
  </f>
  <f name="head">
```

```
      <fs>
        <f name="agr"><var label="@1"/></f>
        <f name="pos"><sym value="noun"/></f>
      </fs>
    </f>
  </fs>
```

Reentrancy 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="@1">
            <fs>
              <f name="number"><sym value="singular"/></f>
            </fs>
          </var>
        </f>
        <f name="pos"><sym value="determiner"/></f>
     </fs>
  </f>
  <f name="head">
     <fs>
       <f name="agr">
         <var label="@1">
           <fs>
              <f name="number"><sym value="singular"/></f>
           </fs>
         </var>
       </f>
       <f name="pos"><sym value="noun"/></f>
     </fs>
  </f>
</fs>
```

Related to structure sharing are at least the following two issues: underspecified values and cyclic feature structures.

4.5.1 Sharing of Underspecified Values

Two nodes of a feature structure may be shared, even if their value is underspecified:

(20) Sharing of underspecified values

$$\begin{bmatrix} \text{SPECIFIER} & \begin{bmatrix} \text{AGR} & \boxed{1} \\ \text{POS} & \textit{determiner} \end{bmatrix} \\ \text{HEAD} & \begin{bmatrix} \text{AGR} & \boxed{1} \\ \text{POS} & \textit{noun} \end{bmatrix} \end{bmatrix}$$

(21) Sharing of underspecified values (XML notation)

```
<fs>
  <f name="specifier">
     <fs>
        <f name="agr"><var label="@1"/></f>
        <f name="pos"><sym value="determiner"/></f>
     </fs>
  </f>
  <f name="head">
     <fs>
       <f name="agr"><var label="@1"/></f>
       <f name="pos"><sym value="noun"/></f>
     </fs>
  </f>
</fs>
```

Note that an underspecified value may also be represented in XML by the empty element `<fs/>`. An alternate but equivalent XML representation for the previous examples would therefore be:

(22) Use of empty element `<fs/>` for underspecified values

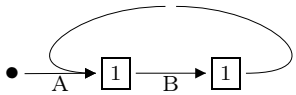
```
<fs>
  <f name="specifier">
     <fs>
        <f name="agr"><var label="@1"><fs/></var></f>
        <f name="pos"><sym value="determiner"/></f>
     </fs>
  </f>
  <f name="head">
     <fs>
       <f name="agr"><var label="@1"></fs></var></f>
       <f name="pos"><sym value="noun"/></f>
     </fs>
  </f>
</fs>
```

4.5.2 Cyclic Feature Structures

The current proposal does not forbid the representation of cyclic feature structures, even if not handled by most FS implementations.

The most straightforward case is provided by direct embedding:

(23) Direct embedding in AVM



(24) Direct embedding in AVM

$$\left[\text{A } \boxed{1} \left[\text{B } \boxed{1}\right]\right]$$

(25) Direct embedding in XML

```
<f name="a">
 <var label="@1">
   <fs>
     <f name="b"><var label="@1"/></f>
   </fs>
  </var>
</f>
```

However, a cycle may be indirect as follows:

(26) Indirect cycle in AVM

$$\left[\begin{array}{l}\text{A } \boxed{1} \left[\text{B } \boxed{2}\right] \\ \text{C } \boxed{2} \left[\text{D } \boxed{1}\right]\end{array}\right]$$

(27) Indirect cycle in XML

```
<fs>
  <f name="a">
    <var label="@1">
     <fs>
       <f name="b"><var label="@2"/></f>
     </fs>
    </var>
  </f>
  <f name="c">
    <var label="@2">
     <fs>
       <f name="d"><var label="@1"/></f>
     </fs>
    </var>
  </f>
</fs>
```

These equivalent examples show that the two structures tagged by $\boxed{1}$ and $\boxed{2}$ share their values with each other in cycle.

4.6 Collections as Complex Feature Values

In feature-based grammar formalisms, such as HPSG and LFG, multivalued features are very common. Some features take list values and others set or multiset values.[8] All the elements in a list are ordered, while there is no such ordering in a set or multiset. In this sense, multisets are still sets. Nevertheless, multisets differ from ordinary sets in that multisets allow the occurrence of identical elements in them as lists do. The multiset {. a, a .}, for instance, is not the same as {. a .}, but the ordinary sets {a, a} and {a} are the same because of the principle of extensionality.

4.6.1 Lists as Feature Values

Perhaps the most famous 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, an accusative noun phrase as its indirect object, and another accusative noun phrase as the direct object, and expects these elements in this order, the feature SUBCAT is given the value:

(28) SUBCAT as a list-valued feature

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

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.

In a more recent version of HPSG, the feature SUBCAT is replaced by a new name, VALENCE, that takes as its value a feature structure which consists of two list-valued features SPECIFIER and COMPS, complements.[9] They are then linked to another list-valued feature ARG-ST, argument structure. Here is an example:

(29) List-valued features

[8] The term 'multiset', with its notation {. .} or $\{\}_m$, is often called 'bag'.

[9] The feature SPEC is treated as a list-valued feature because the feature ARG-ST is treated as the $\oplus$ concatenation of the two list values of SPECIFIER and COMPS.

$$\begin{bmatrix} \text{ORTH } \text{`gives'} \\ \text{SYNAX} \begin{bmatrix} \text{POS } \textit{verb} \\ \text{VALENCE} \begin{bmatrix} \text{SPECIFIER} \left\langle \boxed{1}\ \text{NP}[3\textit{sing}] \right\rangle \\ \text{COMPS} \left\langle \boxed{2}\ \text{NP},\ \boxed{3}\ \text{NP} \right\rangle \end{bmatrix} \end{bmatrix} \\ \text{ARG-ST} \left\langle \boxed{1}, \boxed{2}, \boxed{3} \right\rangle \\ \text{SEMANTICS} \begin{bmatrix} \text{RELATION} \left\langle \text{act, giving} \right\rangle \\ \text{GIVER } \boxed{1} \\ \text{RECIPIENT } \boxed{2} \\ \text{GIVEN } \boxed{3} \end{bmatrix} \end{bmatrix}$$

The order of complements is crucial in English. The indirect object of a ditransitive verb like 'give' or 'buy' precedes the direct object, as in 'John bought Mary a doll'. Hence, the value of the feature COMPS must be presented as an (ordered) list, rather than an (unordered) set.

Note that a list as a feature value may consist of either atomic or complex values, as shown below.

(30) List as a feature value

$$\begin{bmatrix} \text{F } \langle \text{a, b} \rangle \\ \text{G} \left\langle \left[\text{A a}\right], \left[\text{B b}\right] \right\rangle \end{bmatrix}$$

Here, the feature F has a list of two atomic objects as its value, whereas the feature G has a list of two feature structures as its value.

List values can also be represented recursively as shown below:

(31) Recursive representation

$$\begin{bmatrix} \text{F} \begin{bmatrix} \text{FIRST a} \\ \text{REST} \begin{bmatrix} \text{FIRST b} \\ \text{REST null} \end{bmatrix} \end{bmatrix} \\ \text{G} \begin{bmatrix} \text{FIRST} \left[\text{A a}\right] \\ \text{REST} \begin{bmatrix} \text{FIRST} \left[\text{B b}\right] \\ \text{REST null} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

According to this representation, lists as feature values must be viewed as simply serving as a shorthand notation for representing recursively built complex feature structures.[10]

[10]See Shieber (1986: 29-30) for the recursive definition of *list*.

4.6.2 Sets as Feature Values

Some grammatical features may take sets as their values. In free or semi-free order languages like Japanese, Korean or even German, for instance, the subject and the verbal complements in a sentence have no fixed order.[11] Hence, for these languages, the feature COMPS as well as 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:

(32) Set-valued features

$$\begin{bmatrix} \text{ORTH } \text{`gibt'} \\ \text{SYNTAX} \begin{bmatrix} \text{POS } \textit{verb} \\ \text{VALENCE} \begin{bmatrix} \text{SPECIFIER} \left\langle \boxed{1}\ \text{NP}[\textit{nom}] \right\rangle \\ \text{COMPS} \left\langle \boxed{2}\ \text{NP}[\textit{dat}],\ \boxed{3}\ \text{NP}[\textit{acc}] \right\rangle \end{bmatrix} \end{bmatrix} \\ \text{ARG-ST} \left\{ \boxed{1}, \boxed{2}, \boxed{3} \right\} \end{bmatrix}$$

The proper description of relative clauses like 'who runs' may require a multivalued feature RESTRICTION that provides a set of restrictions on some individual PARAMETER.

(33) Set value for the feature RESTRICTION

$$\begin{bmatrix} \text{PARAMETER } \boxed{1} \\ \text{RESTRICTION} \left\{ \begin{bmatrix} \text{RELATION } <\text{property, human}> \\ \text{INSTANCE } \boxed{1} \end{bmatrix}, \begin{bmatrix} \text{RELATION } <\text{activity, running}> \\ \text{RUNNER } \boxed{1} \end{bmatrix} \right\} \end{bmatrix}$$

Here, the set value for the feature RESTRICTION contains two feature structures as its members.[12]

4.6.3 Multisets as Feature Values

Some set-valued features are known to have a particular sort of set called multiset or bag as their value. 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 an example of so-called filler-gap constructions.[13]

(34) Filler-gap construction
{This bus}$_1$, I don't think {Palo Alto}$_2$ is very easy to get to t_2 on t_1?

[11] 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.

[12] In Pollard and Sag (1994), so-called nonlocal features like QUEstion, RELative and SLASH are treated as taking set values for analyzing so-called filler-gap or unbounded dependency constructions. The feature QUE takes as value a set of *non-pronominals*, the feature REL a set of *referential* indices, and the feature SLASH a set of *local* structures.

[13] Taken from Pollard and Moshier (1990: 291).

It has been claimed that the gaps or traces t_1 and t_2 in this example require the feature specification like [SLASH {. NP, NP .}].

Here is a more convincing example:[14]

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

The two occurrences of the pronouns above should be coreferential. So to be able to capture this coreferentiality, a multiset like the following must be set up for some feature specification.

(36) Coreferential multiset

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

4.7 Typed Feature Structure

In many of the recent grammar formalisms, *typed feature structure* has become the main tool for their linguistic descriptions and implementation.

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. Linguistic entities, for instance, like *phrase*, *word*, *pos* (parts of speech), *noun*, and *verb* are treated as features in non-typed feature structure. But in typed feature structure they are rather taken as types.

By *typing*, each feature structure is assigned a particular type. Each feature specification with a particular value is then constrained by this typing. 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*.[15]

4.7.2 Definition

The extension of non-typed feature structure to typed feature structure is very simple in a set-theoretic framework. The main difference between them is the assignment of types to feature structures. A formal definition of typed feature structure can thus be given as follows:[16]

(37) Formal definition of typed feature structure

Given a finite set of **Features** and a finite set of **Types**, a typed feature structure is a tuple $\mathcal{TFS} = < \mathbf{Nodes}, r, \theta, \delta >$ such that

[14] Again, taken from Pollard and Moshier (1990: 294).
[15] Note that atomic feature values are considered *types*, too.
[16] Slightly modified from Carpenter (1992: 36).

i. **Nodes** is a finite set of nodes.

ii. r is a unique member of **Nodes** called *the root*.

iii. θ is a total function that maps **Nodes** to **Types**.

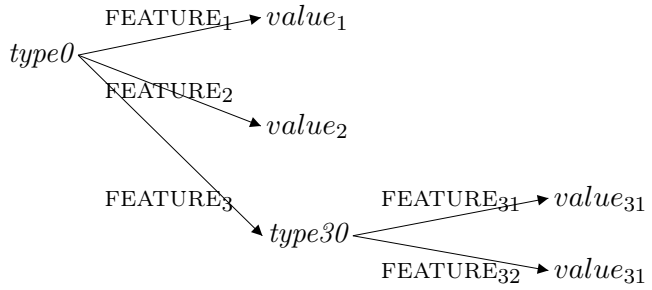
iv. δ is a partial function from **Features**×**Nodes** into **Nodes**.

First, each of the **Nodes** must be rooted at or connected back to the root r. Secondly, there must one and only one root for each feature structure. Thirdly, each of the **Nodes**, including the root r node and terminal nodes, must be assigned a type by the typing function θ. Finally, each of the **Features** labelling each of **Nodes** is assigned a unique value by the feature value function δ.[17]

4.7.3 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:

(38) Typed feature structure in graph



The only difference between the typed graph (38) and the non-typed graph (38) 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:

(39) Typed feature structure in AVM

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

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 an additonal piece of information tells that the features ORTH and SYNTAX are of the type *word* and the feature VALENCE of the type *verb*.

The same type of information can be encoded in an XML notation, as below:

(40) Typed feature structure in XML notation

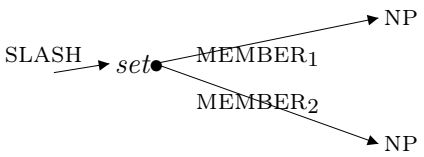
[17] The unique-value restriction on features does not exclude multi-values or alternative values because even in these cases each feature ultimately takes a single value which may be considered complex in structure.

```
<fs type="word">
    <f name="orth"><str>love</str></f>
    <f name="syntax">
        <fs type="verb">
            <f name="valence"><sym 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.

(41) 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 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*.

4.8.1 Definition

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

(42) Properties of type hierarchy

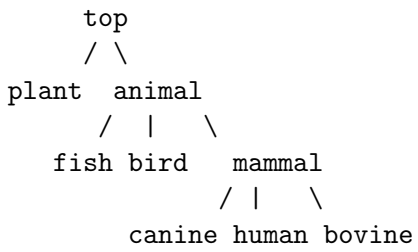
Unique top: It must be a single hierarchy containing all the types with a unique top type.

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*.[18] Incompatible types share no common descendants or subtypes.

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

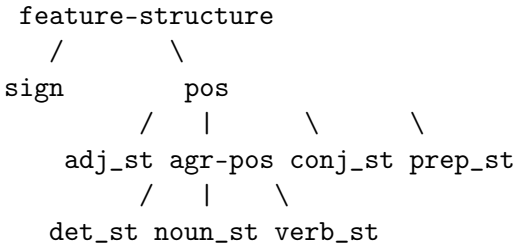
(43) Type hierarchy for some animals



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: [19]

(44) 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*,[20] since they are governed by agreement rules in English.[21]

[18] 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.

[19] Taken modified from Sag, Wasow and Bender (2003: 61).

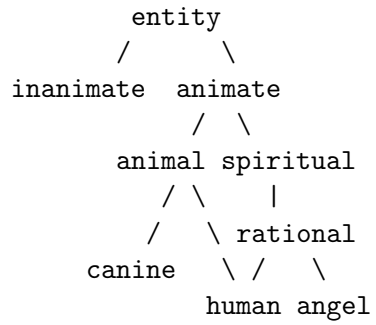
[20] *st* stands for "structure".

[21] In a language like French or Latin, the type *adj_st* should be also be treated as a subtype of *agr-pos*.

4.8.2 Multiple Inheritance

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

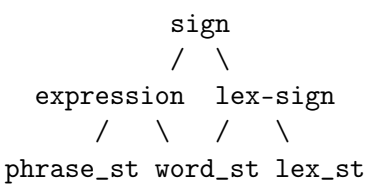
(45) Medieval hierarchy of entities



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.[22].

(46) 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.

4.8.3 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.

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.

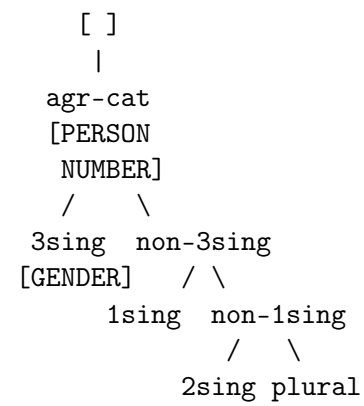
[22]See Sag, Wasow and Bender (2003: 470-475)

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:[23]

(47) 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:

(48) 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:

(49) Ill-formed feature structure

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

[23] 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:

(50) 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.

4.9 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:

(51) 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 alphabets 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).

4.9.1 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:[24]

(52) 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 its subtype.

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:

4.9.2 Condition A on Path Values

(53) 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.[25] 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.

4.9.3 Condition B on Structure Sharing

(54) Case satisfying Condition B

[24] 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.

[25] 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.

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

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:

(55) Another case involving structure sharing

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

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

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

4.9.4 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:

(56) Case involving type ordering

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

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.

4.10 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.

4.10.1 Compatibility

Some feature structures are *compatible* with some others, while there are conflicting cases. Consider the following three AVM's:[26]

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

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

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

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:

(58) 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.

4.10.2 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.[27]

(59) 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 $\bot$.[28]

[27] 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* $\top$.

[28] 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).

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

(60) 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.

(61) 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, $\sqcup$.

4.10.3 Unification of Shared Structures

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

(62) Unification involving reentrancy

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

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

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:

(63) Case for pumping values to each other

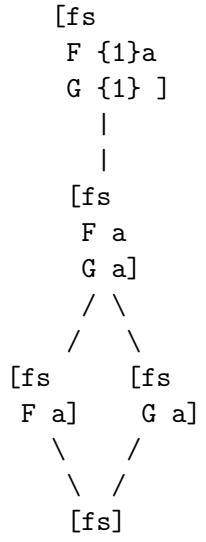
$$G\begin{bmatrix} verb\_st \\ \text{AGR } \boxed{1} \\ \text{SPECIFIER} < \begin{bmatrix} noun\_st \\ \text{AGR}\boxed{1} \end{bmatrix} > \end{bmatrix} \sqcup \underset{J}{\begin{bmatrix} agr\text{-}pos \\ \text{AGR } \begin{bmatrix} \text{PERSON } 3rd \end{bmatrix} \\ \text{SPECIFIER} < \begin{bmatrix}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

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

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.

The following figure represents a subsumption hierarchy, showing how unification works.

(64) Subsumption hierarchy



4.10.4 Generalization

While unification enriches information, *generalization* captures common features from various feature structures. It is a binary *total* operation on two feature structures. Just like *intersection* that operates on sets, it only picks up common feature specifications and puts them into a resulting feature structure.

(65) Definition of generalization

> Given a lattice of feature structures in the subsumption order, the generalization of typed feature structures FS_1 and FS_2 is formally defined as the greatest lower bound of FS_1 and FS_2 in the subsumption lattice. It is then represented as $F_1 \sqcap F_2$.

This definition means that the generalization of two feature structures FS_1 and FS_2 is the smallest, least general, most specific and most informative feature structure that subsumes both of them.

Here are some major properties of generalization:

The empty feature structure: The generalization of any feature structure with the empty feature structure results in the empty feature structure.
Given any feature structure FS_i, $FS_i \sqcap [\,] = []$.

subsumption: The resultant feature structure always subsumes either of the feature structures being generalized. Given two feature structures FS_1 and FS_2, we thus have:

- $(FS_1 \sqcap FS_2) \sqsubseteq FS_1$
- $(FS_1 \sqcap FS_2) \sqsubseteq FS_2$
- For any feature structure FS that subsumes FS_1,
  $FS \sqcap FS_1 = FS$ and $FS_1 \sqcap FS = FS$.

While the unification of two feature structures may fail or be equated with inconsistency $\perp$ when it fails, generalization always succeeds. If there is no common feature specification between any two given feature structures, then their generalization results in the empty feature structure.

4.11 Operations on Feature Values and Types

4.11.1 Concatenation

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.

(66) Concatenated list

$$\begin{bmatrix} \textit{verb\_st} & \\ \text{VALENCE} & \begin{bmatrix} \text{SPECIFIER } \boxed{A} \\ \text{COMPS } \boxed{B} \end{bmatrix} \\ \text{ARG-ST } \boxed{A} \oplus \boxed{C} & \end{bmatrix}$$

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.

4.11.2 Alternation

Some features may take an alternative value.[29] For example, the name '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.

(67) Alternative value

$$\begin{bmatrix} \textit{name} \\ \textsc{orth} \text{ 'Sandy'} \\ \textsc{gender} \ \textit{feminine} \mid \textit{masculine} \end{bmatrix}$$

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.

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:

(68) 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 it might be the case that we should avoid using the curly brackets for alternation.

4.11.3 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.

[29] Strictly speaking, this alternation differs from the boolean disjunction, or the inclusive disjunction, often represented by a wedge $\wedge$, which results in truth even when both of its disjuncts are 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.

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:

(69) 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:

(70) 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).

4.12 Informal Semantics of Feature Structure

[Review:Nancy-7] (4.12) The group felt there was little need to add a full formal semantics for feature structures - specific aspects of the meaning of a given FS representation would be conveyed by a feature system declaration (or equivalent), while general concepts are well described elsewhere in the literature. An informal discussion should give some pointers to the relevant literature.

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 introduction, 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 and Uncertain 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.

Annex A (normative) Formal Definition and Implementation provides formal definitions for all the elements introduced in section 5.

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:

- <**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 for a feature-value specification which can take either of a pair of values.

  value supplies a Boolean 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].[30]

[30] Adapted from Chomsky and Halle (1968, 415).

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

This representation may be encoded in XML as follows:

```
<fs type="phonological segment">
   <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">     <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.

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 (normative) 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 section 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.

- <**symbol**> provides a symbolic feature value.
  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 a numeric value or range of values for a feature.

  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**> provides a string value for a feature.

  – 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' has accusative case, feminine gender and plural number:

```
<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:

```
<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:

```
<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:

```
<fs>
 <f name="houseNumber">
  <numeric value="3418"></numeric>
 </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:

```
<fs>
 <f name="houseNumber">
  <numeric value="3418" max="3440"></numeric>
 </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:

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

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

```
<fs>
 <f name="dailyRainFall">
  <numeric value="0.0" max="1.3" trunc="yes"></numeric>
 </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 x 3 x 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 to be discussed in Part 2.

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:

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

  type indicates type of feature library (i.e., what kind of features it contains).

- <**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.

```
<fLib type="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.

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

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:

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

Once defined, these feature structure values can also be reused. Other <fs> elements may invoke them by reference, using the fVal attribute. one could thus define a feature structure such as :

```
<fs type="dental-fricative" fval="T.DF"></fs>
```

rather than expanding the hierarchy of 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 <var> element, as discussed in 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:

```
<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):

```
<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:

```
<!-- ... -->

<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:

```
<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:

```
<fs type="word">
   <f name="surf"><stringVal>love</stringVal></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 recommended only for cases where the feature value required is a reference to some predefined feature structure.

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 `<var>` element is provided as a means of labelling each such re-entrancy point:

- <**var**> indicates that the feature value is shared with other values bearing the same label within the current scope.

  label supplies a label for the variable.

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:

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

  <f name="verbal">
    <fs>
      <f name="vb-num">
      <var 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 `var` elements, are understood to be references to the same point in the feature structure, which is labelled by their *label* attribute.

The scope of the labels used for 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) any labels 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:

```
<fs fVal="NVA"></fs>
```

then the labels in the example would be interpreted as if they had the value 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:

- <**coll**> contains (or points to) several feature structures which together provide a value for their parent feature element, organized as indicated by the value of the org attribute.

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

  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.

  fVal pointer to features.

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, unless the *feats* attribute is used to specify its contents. If values are supplied within a <coll> element which also specifies values on its *feats* attribute, the implication is that the two sets of values are to be unified.

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:

```
<fs id="p027" type="person">
   <f name="forenames">
     <coll>
       <f name="name"><stringVal>Daniel</stringVal></f>
```

```
      <f name="name"><stringVal>Edouard</stringVal></f>
    </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" feats="pnb005 pfb010 prb001"/>
  </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. 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.

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.

```
<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:

```
<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:

```
<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.

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

- <**vAlt**> represents a feature value which is the disjunction of the values it contains.

– 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)
- <**vColl**> represents a feature value which is the result of collecting together the feature values it combines, using the organization specified by the ORG attribute.

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

  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.

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 'number.of.bathrooms'. 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:

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

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

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

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

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

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

```
<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.

```
<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:

```
<f name="number.of.bathrooms">
 <vNot>
  <numeric value="2"></numeric>
 </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 FSD* or by some other means. Then the following two specifications are equivalent:

```
(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 of the same type 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 <vColl> 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.

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:

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

```
    <sym value="feminine"></sym>
   </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 <vColl> element can be used to achieve this:

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

5.9 Default and Uncertain 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:

- <**default**> provides default value for a feature.
  - 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.

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

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

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

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

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

<f name="gender">
 <vAlt>
  <symbol value="feminine"></symbol>
  <symbol value="masculine"></symbol>
 </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 *TEI 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:

```
<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):

```
<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):

```
 <fLib>
   <!-- ... -->
   <f id="bv" name="verbbase"> <sym value="main"/> </f>
   <f id="bp" name="prepbase"> <sym value="lexical"/> </f>
   <f id="ds" name="degree">   <sym value="superlative"/> </f>
   <f id="fd" name="verbform"> <sym value="ed"/> </f>
   <f id="ns" name="number">   <sym value="singular"/> </f>
   <f id="rp" name="prontype"> <sym value="personal"/> </f>
   <f id="tc" name="nountype"> <sym value="common"/> </f>
   <f id="wj" name="class">    <sym value="adjective"/> </f>
   <f id="wl" name="class">    <sym value="article"/> </f>
   <f id="wn" name="class">    <sym value="noun"/> </f>
   <f id="wp" name="class">    <sym value="preposition"/> </f>
   <f id="wr" name="class">    <sym value="pronoun"/> </f>
   <f id="wv" name="class">    <sym 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.

```
<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:

```
<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>
<!-- ... -->
```

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

```
<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 (normative): Formal Definition and Implementation for the XML Representation of Feature Structures

This elements discussed in this Annex constitute a module of the TEI scheme which is formally defined as follows: Module iso-fs *Element classes for this module.*

A.1 Basic Elements

```
<rng:define name="fs">
 <rng:element name="fs">
  <rng:ref name="fs.content"/>
   </rng:element>
  </rng:define>
<rng:define name="fs.content">
 <rng:zeroOrMore>
  <rng:ref name="f"/>
   </rng:zeroOrMore>
 <rng:ref name="tei.global.attributes"/>
```

Complex values group elements which express complex feature values in feature structures. They also group empty elements which describe the status of other elements, for example by holding groups of links or of abstract interpretations, or by providing indications of certainty etc., and which may appear at any point in a document. Encoders may find it convenient to localize all metadata elements, for example to contain them within the same division as the elements that they relate to; or to locate them all to a division of their own. They may however appear at any point in a text of a certain, say TEI, form.

```
  <rng:ref name="fs.attributes.type"/>
  <rng:ref name="fs.attributes.feats"/>
  <rng:ref name="fs.newattributes"/>
  <rng:optional>
   <rng:attribute name="TEIform" a:defaultValue="fs">
    <rng:text/>
     </rng:attribute>
    </rng:optional>
   </rng:define>
<rng:define name="fs.newattributes" combine="choice">
 <rng:empty/>
  </rng:define>
<rng:define name="tei.complexVal" combine="choice">
 <rng:ref name="fs"/>
  </rng:define>
<rng:define name="tei.metadata" combine="choice">
 <rng:ref name="fs"/>
  </rng:define>
<rng:define name="fs.attributes.type">
 <rng:optional>
```

```
  <rng:attribute name="type">
   <rng:ref name="fs.attributes.type.content"/>
    </rng:attribute>
   </rng:optional>
  </rng:define>
<rng:define name="fs.attributes.type.content">
 <rng:ref name="datatype.Key"/>
  </rng:define>
<rng:define name="fs.attributes.feats">
 <rng:optional>
  <rng:attribute name="feats">
   <rng:ref name="fs.attributes.feats.content"/>
    </rng:attribute>
   </rng:optional>
  </rng:define>
<rng:define name="fs.attributes.feats.content">
 <rng:data type="IDREFS"/>
  </rng:define>


<rng:define name="f">
 <rng:element name="f">
  <rng:ref name="f.content"/>
   </rng:element>
  </rng:define>
<rng:define name="f.content">
 <rng:zeroOrMore>
  <rng:ref name="tei.featureVal"/>
   </rng:zeroOrMore>
 <rng:ref name="tei.global.attributes"/>
 <rng:ref name="f.attributes.name"/>
 <rng:ref name="f.attributes.fVal"/>
 <rng:ref name="f.newattributes"/>
 <rng:optional>
  <rng:attribute name="TEIform" a:defaultValue="f">
   <rng:text/>
    </rng:attribute>
   </rng:optional>
  </rng:define>
<rng:define name="f.newattributes" combine="choice">
 <rng:empty/>
  </rng:define>
<rng:define name="f.attributes.name">
 <rng:attribute name="name">
  <rng:ref name="f.attributes.name.content"/>
   </rng:attribute>
  </rng:define>
<rng:define name="f.attributes.name.content">
```

```
 <rng:data type="NMTOKEN"/>
  </rng:define>
<rng:define name="f.attributes.fVal">
 <rng:optional>
  <rng:attribute name="fVal">
   <rng:ref name="f.attributes.fVal.content"/>
    </rng:attribute>
   </rng:optional>
  </rng:define>
<rng:define name="f.attributes.fVal.content">
 <rng:data type="IDREF"/>
  </rng:define>
```

A.2 Elementary Feature Values

```
<rng:define name="binary">
 <rng:element name="binary">
  <rng:ref name="binary.content"/>
   </rng:element>
  </rng:define>
<rng:define name="binary.content">
 <rng:empty/>
 <rng:ref name="tei.global.attributes"/>
```

Atomic values group elements used to represent atomic feature values in feature structures.

```
  <rng:ref name="binary.attributes.value"/>
  <rng:ref name="binary.newattributes"/>
  <rng:optional>
   <rng:attribute name="TEIform" a:defaultValue="binary">
    <rng:text/>
     </rng:attribute>
    </rng:optional>
  </rng:define>
<rng:define name="binary.newattributes" combine="choice">
 <rng:empty/>
  </rng:define>
<rng:define name="tei.singleVal" combine="choice">
 <rng:ref name="binary"/>
  </rng:define>
<rng:define name="binary.attributes.value">
 <rng:attribute name="value">
  <rng:ref name="binary.attributes.value.content"/>
   </rng:attribute>
  </rng:define>
<rng:define name="binary.attributes.value.content">
```

```
   <rng:data type="boolean"/>
    </rng:define>

  <rng:define name="symbol">
   <rng:element name="symbol">
    <rng:ref name="symbol.content"/>
     </rng:element>
    </rng:define>
  <rng:define name="symbol.content">
   <rng:empty/>
   <rng:ref name="tei.global.attributes"/>
```

Atomic values group elements used to represent atomic feature values in feature structures.

```
   <rng:ref name="symbol.attributes.value"/>
   <rng:ref name="symbol.newattributes"/>
   <rng:optional>
    <rng:attribute name="TEIform" a:defaultValue="symbol">
     <rng:text/>
      </rng:attribute>
     </rng:optional>
    </rng:define>
  <rng:define name="symbol.newattributes" combine="choice">
   <rng:empty/>
    </rng:define>
  <rng:define name="tei.singleVal" combine="choice">
   <rng:ref name="symbol"/>
    </rng:define>
  <rng:define name="symbol.attributes.value">
   <rng:attribute name="value">
    <rng:ref name="symbol.attributes.value.content"/>
     </rng:attribute>
    </rng:define>
  <rng:define name="symbol.attributes.value.content">
   <rng:ref name="datatype.Key"/>
    </rng:define>

  <rng:define name="numeric">
   <rng:element name="numeric">
    <rng:ref name="numeric.content"/>
     </rng:element>
    </rng:define>
  <rng:define name="numeric.content">
   <rng:empty/>
   <rng:ref name="tei.global.attributes"/>
```

Atomic values group elements used to represent atomic feature values in feature structures.

```
 <rng:ref name="numeric.attributes.value"/>
 <rng:ref name="numeric.attributes.max"/>
 <rng:ref name="numeric.attributes.trunc"/>
 <rng:ref name="numeric.newattributes"/>
 <rng:optional>
  <rng:attribute name="TEIform" a:defaultValue="numeric">
   <rng:text/>
    </rng:attribute>
   </rng:optional>
  </rng:define>
<rng:define name="numeric.newattributes" combine="choice">
 <rng:empty/>
  </rng:define>
<rng:define name="tei.singleVal" combine="choice">
 <rng:ref name="numeric"/>
  </rng:define>
<rng:define name="numeric.attributes.value">
 <rng:attribute name="value">
  <rng:ref name="numeric.attributes.value.content"/>
   </rng:attribute>
  </rng:define>
<rng:define name="numeric.attributes.value.content">
 <rng:data type="float"/>
  </rng:define>
<rng:define name="numeric.attributes.max">
 <rng:optional>
  <rng:attribute name="max">
   <rng:ref name="numeric.attributes.max.content"/>
    </rng:attribute>
   </rng:optional>
  </rng:define>
<rng:define name="numeric.attributes.max.content">
 <rng:data type="float"/>
  </rng:define>
<rng:define name="numeric.attributes.trunc">
 <rng:optional>
  <rng:attribute name="trunc">
   <rng:ref name="numeric.attributes.trunc.content"/>
    </rng:attribute>
   </rng:optional>
  </rng:define>
<rng:define name="numeric.attributes.trunc.content">
 <rng:data type="boolean"/>
  </rng:define>


<rng:define name="string">
 <rng:element name="string">
```

```
    <rng:ref name="string.content"/>
     </rng:element>
    </rng:define>
  <rng:define name="string.content">
   <rng:text/>
   <rng:ref name="tei.global.attributes"/>
```

Atomic values group elements used to represent atomic feature values in feature structures.

```
   <rng:ref name="string.newattributes"/>
   <rng:optional>
    <rng:attribute name="TEIform" a:defaultValue="string">
     <rng:text/>
      </rng:attribute>
     </rng:optional>
    </rng:define>
  <rng:define name="string.newattributes" combine="choice">
   <rng:empty/>
    </rng:define>
  <rng:define name="tei.singleVal" combine="choice">
   <rng:ref name="string"/>
    </rng:define>

  <rng:define name="var">
   <rng:element name="var">
    <rng:ref name="var.content"/>
     </rng:element>
    </rng:define>
  <rng:define name="var.content">
   <rng:optional>
    <rng:ref name="tei.featureVal"/>
     </rng:optional>
   <rng:ref name="tei.global.attributes"/>
```

Atomic values group elements used to represent atomic feature values in feature structures.

```
   <rng:ref name="var.attributes.label"/>
   <rng:ref name="var.newattributes"/>
   <rng:optional>
    <rng:attribute name="TEIform" a:defaultValue="var">
     <rng:text/>
      </rng:attribute>
     </rng:optional>
    </rng:define>
  <rng:define name="var.newattributes" combine="choice">
   <rng:empty/>
```

```
  </rng:define>
 <rng:define name="tei.singleVal" combine="choice">
  <rng:ref name="var"/>
   </rng:define>
 <rng:define name="var.attributes.label">
  <rng:attribute name="label">
   <rng:ref name="var.attributes.label.content"/>
    </rng:attribute>
   </rng:define>
 <rng:define name="var.attributes.label.content">
  <rng:ref name="datatype.Key"/>
   </rng:define>
```

A.3 Multiple and Default Values

```
 <rng:define name="coll">
  <rng:element name="coll">
   <rng:ref name="coll.content"/>
    </rng:element>
   </rng:define>
 <rng:define name="coll.content">
  <rng:group>
   <rng:zeroOrMore>
    <rng:choice>
     <rng:ref name="fs"/>
     <rng:ref name="tei.singleVal"/>
      </rng:choice>
     </rng:zeroOrMore>
    </rng:group>
  <rng:ref name="tei.global.attributes"/>
```

Complex values group elements which express complex feature values in feature structures.

```
   <rng:ref name="coll.attributes.org"/>
   <rng:ref name="coll.attributes.fVal"/>
   <rng:ref name="coll.newattributes"/>
   <rng:optional>
    <rng:attribute name="TEIform" a:defaultValue="coll">
     <rng:text/>
      </rng:attribute>
     </rng:optional>
   </rng:define>
 <rng:define name="coll.newattributes" combine="choice">
  <rng:empty/>
   </rng:define>
 <rng:define name="tei.complexVal" combine="choice">
```

```
 <rng:ref name="coll"/>
  </rng:define>
<rng:define name="coll.attributes.org">
 <rng:optional>
  <rng:attribute name="org">
   <rng:ref name="coll.attributes.org.content"/>
    </rng:attribute>
   </rng:optional>
  </rng:define>
<rng:define name="coll.attributes.org.content">
 <rng:choice>
  <rng:value>set  </rng:value>
  <rng:value>bag  </rng:value>
  <rng:value>list  </rng:value>
   </rng:choice>
  </rng:define>
<rng:define name="coll.attributes.fVal">
 <rng:optional>
  <rng:attribute name="fVal">
   <rng:ref name="coll.attributes.fVal.content"/>
    </rng:attribute>
   </rng:optional>
  </rng:define>
<rng:define name="coll.attributes.fVal.content">
 <rng:data type="IDREFS"/>
  </rng:define>

<rng:define name="default">
 <rng:element name="default">
  <rng:ref name="default.content"/>
   </rng:element>
  </rng:define>
<rng:define name="default.content">
 <rng:empty/>
 <rng:ref name="tei.global.attributes"/>
```

Atomic values group elements used to represent atomic feature values in feature structures.

```
 <rng:ref name="default.newattributes"/>
 <rng:optional>
  <rng:attribute name="TEIform" a:defaultValue="default">
   <rng:text/>
    </rng:attribute>
   </rng:optional>
  </rng:define>
<rng:define name="default.newattributes" combine="choice">
 <rng:empty/>
```

```
  </rng:define>
<rng:define name="tei.singleVal" combine="choice">
 <rng:ref name="default"/>
  </rng:define>
```

A.4 Alternation and Negation

```
<rng:define name="vAlt">
 <rng:element name="vAlt">
  <rng:ref name="vAlt.content"/>
   </rng:element>
  </rng:define>
<rng:define name="vAlt.content">
 <rng:group>
  <rng:choice>
   <rng:ref name="tei.featureVal"/>
    </rng:choice>
  <rng:oneOrMore>
   <rng:choice>
    <rng:ref name="tei.featureVal"/>
     </rng:choice>
    </rng:oneOrMore>
   </rng:group>
 <rng:ref name="tei.global.attributes"/>
```

Complex values group elements which express complex feature values in feature structures.

```
 <rng:ref name="vAlt.newattributes"/>
 <rng:optional>
  <rng:attribute name="TEIform" a:defaultValue="vAlt">
   <rng:text/>
    </rng:attribute>
   </rng:optional>
  </rng:define>
<rng:define name="vAlt.newattributes" combine="choice">
 <rng:empty/>
  </rng:define>
<rng:define name="tei.complexVal" combine="choice">
 <rng:ref name="vAlt"/>
  </rng:define>

<rng:define name="vNot">
 <rng:element name="vNot">
  <rng:ref name="vNot.content"/>
   </rng:element>
  </rng:define>
```

```
<rng:define name="vNot.content">
 <rng:group>
  <rng:ref name="tei.featureVal"/>
   </rng:group>
 <rng:ref name="tei.global.attributes"/>
```

Complex values group elements which express complex feature values in feature structures.

```
  <rng:ref name="vNot.newattributes"/>
  <rng:optional>
   <rng:attribute name="TEIform" a:defaultValue="vNot">
    <rng:text/>
     </rng:attribute>
    </rng:optional>
   </rng:define>
<rng:define name="vNot.newattributes" combine="choice">
 <rng:empty/>
  </rng:define>
<rng:define name="tei.complexVal" combine="choice">
 <rng:ref name="vNot"/>
  </rng:define>

<rng:define name="vColl">
 <rng:element name="vColl">
  <rng:ref name="vColl.content"/>
   </rng:element>
  </rng:define>
<rng:define name="vColl.content">
 <rng:oneOrMore>
  <rng:ref name="tei.featureVal"/>
   </rng:oneOrMore>
 <rng:ref name="tei.global.attributes"/>
```

Complex values group elements which express complex feature values in feature structures.

```
  <rng:ref name="vColl.attributes.org"/>
  <rng:ref name="vColl.newattributes"/>
  <rng:optional>
   <rng:attribute name="TEIform" a:defaultValue="vColl">
    <rng:text/>
     </rng:attribute>
    </rng:optional>
   </rng:define>
<rng:define name="vColl.newattributes" combine="choice">
 <rng:empty/>
  </rng:define>
```

```
<rng:define name="tei.complexVal" combine="choice">
 <rng:ref name="vColl"/>
  </rng:define>
<rng:define name="vColl.attributes.org">
 <rng:optional>
  <rng:attribute name="org">
   <rng:ref name="vColl.attributes.org.content"/>
    </rng:attribute>
   </rng:optional>
  </rng:define>
<rng:define name="vColl.attributes.org.content">
 <rng:choice>
  <rng:value>set  </rng:value>
  <rng:value>bag  </rng:value>
  <rng:value>list  </rng:value>
   </rng:choice>
  </rng:define>
```

A.5 Feature Libraries

```
<rng:define name="fLib">
 <rng:element name="fLib">
  <rng:ref name="fLib.content"/>
   </rng:element>
  </rng:define>
<rng:define name="fLib.content">
 <rng:oneOrMore>
  <rng:ref name="f"/>
   </rng:oneOrMore>
 <rng:ref name="tei.global.attributes"/>
```

These values group empty elements which describe the status of other elements, for example by holding groups of links or of abstract interpretations, or by providing indications of certainty etc., and which may appear at any point in a document.Encoders may find it convenient to localize all metadata elements, for example to contain them within the same division as the elements that they relate to; or to locate them all to a division of their own. They may however appear at any point in a text of a certain, say TEI, form.

```
 <rng:ref name="fLib.attributes.type"/>
 <rng:ref name="fLib.newattributes"/>
 <rng:optional>
  <rng:attribute name="TEIform" a:defaultValue="fLib">
   <rng:text/>
    </rng:attribute>
   </rng:optional>
  </rng:define>
<rng:define name="fLib.newattributes" combine="choice">
```

```
 <rng:empty/>
  </rng:define>
<rng:define name="tei.metadata" combine="choice">
 <rng:ref name="fLib"/>
  </rng:define>
<rng:define name="fLib.attributes.type">
 <rng:optional>
  <rng:attribute name="type">
   <rng:ref name="fLib.attributes.type.content"/>
    </rng:attribute>
   </rng:optional>
  </rng:define>
<rng:define name="fLib.attributes.type.content">
 <rng:text/>
  </rng:define>

<rng:define name="fvLib">
 <rng:element name="fvLib">
  <rng:ref name="fvLib.content"/>
   </rng:element>
  </rng:define>
<rng:define name="fvLib.content">
 <rng:zeroOrMore>
  <rng:choice>
   <rng:ref name="tei.featureVal"/>
    </rng:choice>
   </rng:zeroOrMore>
 <rng:ref name="tei.global.attributes"/>
```

These values group empty elements which describe the status of other elements, for example by holding groups of links or of abstract interpretations, or by providing indications of certainty etc., and which may appear at any point in a document.Encoders may find it convenient to localize all metadata elements, for example to contain them within the same division as the elements that they relate to; or to locate them all to a division of their own. They may however appear at any point in a text of a certain, say *tei*, form.

```
 <rng:ref name="fvLib.attributes.type"/>
 <rng:ref name="fvLib.newattributes"/>
 <rng:optional>
  <rng:attribute name="TEIform" a:defaultValue="fvLib">
   <rng:text/>
    </rng:attribute>
   </rng:optional>
  </rng:define>
<rng:define name="fvLib.newattributes" combine="choice">
 <rng:empty/>
  </rng:define>
<rng:define name="tei.metadata" combine="choice">
```

```
  <rng:ref name="fvLib"/>
   </rng:define>
 <rng:define name="fvLib.attributes.type">
  <rng:optional>
   <rng:attribute name="type">
    <rng:ref name="fvLib.attributes.type.content"/>
     </rng:attribute>
    </rng:optional>
   </rng:define>
 <rng:define name="fvLib.attributes.type.content">
  <rng:ref name="datatype.Key"/>
   </rng:define>
```

In a TEI conformant document, this module is selected as described in *TEI P5, chapter ST*

Annex B (non-normative): 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 noun form $\theta\varepsilon\alpha\iota$ goddesses has accusative case, feminine gender and plural number.

```
<fs type="word structure">
   <f name="case">   <sym value="accusative"/> </f>
   <f name="gender"> <sym value="feminine"/> </f>
   <f name="number"> <sym 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">   <sym value="feminine"/> </f>
   <f name="gender"> <sym 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>
    <sym value='nominative'/>
    <sym value='genitive'/>
```

```
    <sym value='dative'/>
    <sym value='accusative'/>
    <sym 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.

[Nancy01d: Laurent Romary] (Survey) Responding to the request from the German delegation for a survey of existing implementations, it was agreed that such a survey should be undertaken for eventual inclusion as an informative annex to the Standard, or as a free-standing TEI document. This work would be undertaken by Thierry Declerck in collaboration with Lou Burnard, since the TEI was also keen to document implementations of all aspects of the Guidelines.

Annex C (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.

C.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` +>}

C.2 Grammar Formalisms or Theories

Almost all current computational grammar formalisms or theories incorporate feature structures (attribute value structures). They use the operation of unification to merge the information encoded in feature structures, hence sometimes the term unification-based grammars. 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), Pollard and Sag (1994) and Sag, Wasow and Bender (2003)

C. Lexical Functional Grammar (LFG): Bresnan (1982),

D. Functional Unification Grammar (FUG): Kay (1983) and Kay (1985)

E. Definite Clause Grammar (DCG): Pereira and Warren (1980)

F. Tree Adjoining Grammar (TAG)

G. Construction Grammar (CG)

C.3 Implementations of (Typed) Feature Structures

A. ALE (Carpenter and Penn, 1995)
The Attribute Logic Engine (ALE, Carpenter and Penn, 1995), created at the Carnegie Mellon University, 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. 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).

C. 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.

D. ALEP (Groenendijk and Simpkins, 1994)
The Advanced Linguistic Engineering Platform (ALEP, Simpkins, 1994b), 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.

E. 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.

F. ProFIT (Erbach, 1994b)
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.

G. CL-ONE (Manandhar, 1994b)
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.

H. 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.

I. 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) and Krieger and Schäfer (1994b).

J. TFS (Emele, 1993)
The Typed Feature Structure representation formalism, cf. Emele and Zajac (1990b) and Zajac (1992), was developed by Martin Emele and Rémi Zajac within the German Polygloss project at IMS, University of Stuttgart.

K. 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.

L. Grammar Writer's Workbench for Lexical Functional Grammar (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...) 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).

M. LFG-PC and LFGW system
Avery Andrews developed a small LFG system that runs on PC's (XT's, in fact), that is basically orientated 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. DATA (Roger Evans and Gerald Gazdar)
DATR, developed at the University of Sussex by Roger Evans and Gerald Gazdar, 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.

O. QDATR (James Kilbury, Petra Barg, Ingrid Renz, Christof Rumpf, Sebastian Varges)
QDATR is an implementation of the DATR formalism. QDATR supports syntactic analysis, text generation, machine translation and can be used for testing linguistic theories making use of non-monotonic inheritances.

P. FGW (Functional Grammar Workbench)
This 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.

Q. Malaga (Björn Beutel)
Malaga is a development environment for Left-Associative Grammars (LAGs). The Left-Associative Grammar 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 special- The language has a Pascal-ish syntax, but it uses a powerful dynamic typing system with attribute-value structures and lists that can be nested. A collection of 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 is on its way.

R. GULP – Graph Unification Logic Programming
A preprocessor for handling feature structures, such as case:nom..number:plural, 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

S. VERBMOBIL ???

C.4 Artificial Intelligence

A. Knowledge Representation: Aït-Kaci (1985)

B. Date Types: Cardelli (1984)

Bibliography

[1] British National Corpus, http://www.hcu.ox.ac.uk/BNC/.

[2] Candito, Marie-H??e, (1999), Structuration d'une grammaire LTAG : application au francais et l'italien, Paris 7 Thèse d'université.

Check missing characters.

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

[4] homsky, Noam and Morris Halle (1968), *The Sound Pattern of English*, Harper & Row, New York.

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

[6] Crabbé, Benoit and Gaiffe, Bertrand and Roussanaly, Azim (2003), "Représentation et gestion de grammaires TAG lexicalisées", *Traitement Automatique des Langues*.

Give Volume No, page nos.

[7] Fenstad, Jens Erik, Tore Langholm, and Espen Vestre (1992), "Representations and interpretations", in Michael Rosner and Roderick Johnson (Eds.), *Computational Linguistics and Formal Semantics*, 31-96, Cambridge University Press, Cambridge.

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

[9] Herwijnen, Eric van (1990), *Practical SGML*, Kluwer Academic Publishers, Dordrecht.

[10] Ide, Nancy, Jacques Le Maitre, and Jean Veronis (1993), "Outline of a model for lexical databases", *Information Processing and Management*, 29(2): 159-186. Reprinted in Antonio Zampolli et al. (eds.) (2001), *Current Issues in Computational Linguistics: In Honour of Don Walker*, Kluwer Academic Publishers, Dordrecht.

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

[12] Kay, Martin (1992), "Unification", in Michael Rosner and Roderick Johnson (Eds.), *Computational Linguistics and Formal Semantics*, 1-30, Cambridge University Press, Cambridge.

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

[14] Lee, Kiyong (2004), Lou Burnard, Laurent Romary, Eric de la Clergerie, Ulrich Schaefer, Thierry Declerck, Syd Bauman, Harry Bunt, Lionel Clément, Tomaz Erjavec, Azim Roussanaly, and Claude Roux, "Towards an international standard on feature structure representation (2)", *Proceedings of LREC2004 Fourth International Conference on Language Resources and Evaluation Satellite Workshop on A Registry of Linguistic Data Categories within an Integreated Language Resources Repository Area (INTERA)*, (no page numbering), Lisbon.

[15] Lopez, Patrice (1999), Analyse d'énoncés oraux pour le dialogue homme-machine à l'aide de grammaires lexicalisées d'arbres, Thèse d'université.

[16] Pereira, Fernando C. N. (1987), *Grammars and Logics of Partial Information*, SRI International Technical Note 420, SRI International, Menlo Park, CA.

[17] Pollard, Carl J. and Ivan A. Sag (1987), *Information-based Syntax and Semantics*, Vol. 1 Fundamentals, CSLI Lecture Notes 13, Stanford.

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

[19] Pollard, Carl J. and M. Andrew Moshier (1990), "Unifying partial descriptions of sets", in Philip P. Hanson (ed.), *Information, Language, and Cognition*, 285-322, The University of British Columbia Press, Vancouver.

[20] Pustejovsky, James (1995, 1996), *The Generative Lexicon*, The MIT Press, Cambridge, MA.

[21] Sag, Ivan A. (2003), "Coordination and underspecification", in Jong-Bok Kim and Stephen Wechsler (eds.), *Proceedings of the 9th International Conference on Head-driven Phrase Structure Grammar*, 267-291, CSLI Publications, Stanford.

[22] Sag, Ivan A. and Thomas Wasow (1999, 2003), *Syntactic Theory: A Formal Introduction*, CSLI Publications, Stanford.

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

[24] Vijay-Shanker, K. (1987), *A Study of Tree Adjoining Grammar*, University of Pennsylvania Ph.D. thesis.

[25] Vijay-Shanker, K. and Joshi, A.K. (1988), "Feature-structure based tree adjoining grammar", *Proceedings of COLING'88*.

NB: page numbers.

[26] Young, Michael J. (2002), *XML Step by Step*, 2nd edition, Microsoft Press, Redmond, Washington.