AN ENVIRONMENT FOR SUPPORTING EXPERIMENTAL COMPUTATIONAL LINGUISTICS RESEARCH

August 14, 1992

D. Terence Langendoen, Principal Investigator

Department of Linguistics
The University of Arizona
Tucson, AZ 85721
E-mail: langendt@arizona.edu

ATRP-0011-91 Final Report

Abstract

This report describes Twincle, The Windowing Computational Linguistics Environment, a set of computational tools for developing models of grammars for natural languages, using graphical and windowing interfaces. The overall environment consists of a set of more specific environments for specifying the general structure of lexical items, the structure of specific lexical items, the grammatical rules for combining lexical items into phrases and larger linguistic constructs, and for applying the lexical and rule specifications to the analysis of examples. In addition, this report describes the current state of the recommendations being developed by the Principal Investigator for the representation of the linguistic analysis of machine-readable text using SGML, as part of the effort of the Text Encoding Initiative. The grammatical formalisms being used in the Twincle project are a subset of the latter formalisms, and the project will be continued in such a way that the output of the Twincle parser will conform to the final recommendations of the Text Encoding Initiative regarding the representation of the linguistic analyses of text.

Table of Contents

TWINCLE: The WINdowing Computational Linguistics Environment

Introduction

The Twincle project is an effort to develop a friendly computational environment for linguists to write and test grammars for natural languages, by enabling them to specify the components of those grammars using windowing tools and high-resolution graphics.[1] It has long been recognized that the main hindrance to the development of grammatical models for natural languages of any reasonable degree of complexity and detail is the inability of linguists and programmers to understand the complex interrelationships among the various parts of the model. Providing graphical representations of those interrelationships, and the ability to specify grammatical models in terms of those representations will, we believe, enable linguists, even with little or no help from programmers, to build complex grammatical models to their own specifications.

In this phase of the Twincle project, we decided to design an environment that would support the development of ***augmented phrase-structure grammars*** (***attribute grammars*** or ***feature-structure grammars***) [Alle87], because this formalism is widely used in computational linguistics, is general enough to provide models for nearly every currently popular grammatical theory, and is relatively easy to visualize. Such grammars generally provide a set of ***terminal elements***, which for our purposes we can identify with ***lexical entries***, each consisting of a ***label*** (typically a string representing the conventional appearance of the entry in text), a ***lexical class*** and a set of attributes and values. They also generally provide a set of ***nonterminal elements*** consisting of a ***grammatical class*** and a set of attributes and values, and a set of ***productions*** or ***rules***, for combining terminal and nonterminal elements together into representations (often, but not necessarily representable as trees, but always as sets of directed, connected graphs) of linguistic structures of potentially arbitrary length and complexity.

An additional motivation for selecting the formalism of augmented phrase-structure grammars is that a formalism very much like it has been selected by the Text Encoding Initiative (TEI) [Sper90] as the basis for the development of an encoding standard for the interchange of linguistic and other types of analyses of machine-readable text. The actual encoding is to be done using SGML [Gold90], with class and attribute information specified by SGML ***tags*** and ***attributes***. It is a relatively straightforward matter to convert the representations of the parses of textual matter that the Twincle system provides into SGML notation of the appropriate sort. The SGML encoding standard for linguistic representations is being developed jointly by Simons [Simo92] and the Principal Investigator. The relevant aspect of the standard being developed for the representation of textual analysis appears below in "Encoding the Grammatical Structure of Texts" on page 9. We have not yet attempted to provide for the mapping between Twincle representations and the required SGML representations because the latter have not yet been finally settled

[1] This work was carried out at The University of Arizona by the Principal Investigator, with support from the Advanced Telecommunications Research Program of the Department of Electrical and Computing Engineering. He was assisted by Clinton L. Jeffery, a graduate student in the Department of Computer Science, who served as a consultant; and three part-time Research Assistants: Jon Lipp, an undergraduate student in the Department of Computer Science; Young-Gie Min, a graduate student in the Department of Linguistics; and Srikanth Raghavan, a graduate student in the Department of Computer Science. The project will continue in the Fall 1992 semester, with support from the Cognitive Science Program, which will enable Jon Lipp to continue to work with the Principal Investigator as a part-time Research Assistant. A report on the project will also be presented at the ICEBOL6 Conference, South Dakota State University, Madison, SD in October 1992.

upon by the TEI community (the final decision may not be made for another six months to a year yet), and because those representations are more complex than we have been able to provide for yet within Twincle (they represent a goal to strive for in the next phase of this work).

Choice of Programming Language

Programming Language Alternatives

Certain very-high-level languages such as PROLOG and LISP are traditionally advocated for work in natural language processing and computational linguistics; such languages are characterized by their list processing capabilities, automatic memory management, and built-in data structures. These features make them appropriate for exploratory programming and rapid prototyping. Other languages, such as C, C++, Smalltalk, Actor, and Visual BASIC are more appropriate for writing programs with extensive graphical user interface components such as Twincle. This project had strong requirements in both application domains. It also had strong constraints: our limited financial resources forced us to consider only alternatives that would run with acceptable interactive speed on an Intel 486 processor and we had a very limited budget for software.

Neither PROLOG nor LISP was selected because we were aware of no implementation available that would meet our performance, budget, and user-interface requirements. It is possible that such a system exists, but most high-performance implementations of these languages are expensive and run only on workstation hardware.

The Intel platform does offer several affordable languages with which to write user interface software. C and C++ were rejected because they did not offer a high-enough level of interface to support exploratory programming. Actor and Visual BASIC suffer from various limitations and are not portable across window and operating systems. Among commercially available languages SmallTalk comes the closest to meeting our constraints. But although SmallTalk is highly suitable for rapid prototyping in the hands of expert SmallTalk programmers, it has a fairly steep learning curve and implementations of SmallTalk on Intel systems are neither inexpensive nor exceptionally fast.

The Icon programming language has extensive string and list processing capabilities that make it a natural choice for text-oriented applications [Gris90]. With the release of Version 8.6, Icon includes a high-level interface to the X Window System [Jeff92]. Both local expertise with Icon and its public domain status encouraged its selection for use in Twincle.

Selected Features of Icon and Idol

Several features of Icon make it particularly appropriate to the domain of computational linguistics. These features include:

- Sophisticated string scanning makes it easier to do more sophisticated lexical and morphological analyses not possible with stock tools based on regular expressions.
- Direct support for a character-set data type as well as support for more general heterogeneous data structures with polymorphic operations.
- Sophisticated control structures that support goal-directed evaluation and control backtracking simplify the implementation of many algorithms.

In addition to these features, the object-oriented extension to Icon called Idol has features we considered important in a larger-scale effort [Jeff90]

- Classes allow extension of Icon's built-in repertoire of data types in a way that minimizes functional dependencies between different code modules.

- Inheritance allows new types to be specified in terms of existing types, encouraging code re-use. In the computational linguistics domain many components such as lexical items fall into natural hierarchies such as hyponymy that can be expressed directly with classes and inheritance.

- Idol also extends Icon with some basic non object-oriented features that are extremely useful in larger programs that are written in several files, such as constant declarations and source file inclusion.

Taken together, Icon's symbolic processing capabilities, X-Icon's user interface support, and Idol's object-oriented model make this environment very appropriate for computational linguistics applications.

Lessons learned about Icon and Idol in Twincle

Over the course of the project two valuable lessons were learned about Twincle's implementation language, Icon, and the object-oriented extension we used, Idol. The first and most significant result is that Icon's built-in operations need to be more polymorphic.

Polymorphism is the ability to use one notation or set of operations on more than one type of data. Icon has a large number of polymorphic operations, and while these are useful we found that many operations are not as polymorphic as they could be.

For example, during the course of the implementation, we encountered a need to mix records and tables in our parse trees: records are compact and are used to efficiently represent lexical items, while tables are more flexible since arbitrary keys can be inserted into them. Icon's list, table, and record types are all heterogeneous, that is, structure elements may be any type and in particular such structures could contain some elements that are tables and some that are records.

In addition to this minimal requirement that Icon meets well, mixing records and tables requires that the parse engine be able to process nodes polymorphically; it applies the same operations irrespective of whether a given node is represented by a record or a table. In either case, linguistic features of the node are accessed by string name during parsing. The Twincle parse engine processes nodes irrespective of whether they are leafs represented as records or internal nodes represented as tables. But record fields are accessed using the . operator and tables are subscripted using the [] operator.

What was needed in this application was a way to access records by field name using a string instead of the . operator — a subscript operation analogous to table subscripting. The following Icon procedure implements element access by string name independent of whether its first argument is a table or a record.

```
procedure access(x, s)
  if type(x) == "table" then return x[s]
  # else x is a record
  every i := 1 to *x do {
    im := image(x[i])
    if im[*im - *s - 1:0] == "."||s then return x
  }
end
```

The problem with this procedure is that it is a slow way of doing a natural thing. The use of the image() function to obtain the records' field names is clumsy. A better solution is to make Icon's subscript more polymorphic to support record field access by string name. Our familiarity with the Icon implementation allowed us to make this tiny addition. By submitting it back to the Icon Project, the extension was made available to the rest of the Icon community as part of Version 8.7. It is likely that increasing the polymorphic behavior of other operations would result in similar improvements in the expressive power of Icon.

Making additions to a language is not something to be taken lightly however. Icon's subscript operator was already defined for records when the supplied index was an integer, e.g. r[2] refers to the second field of the record independent of its field name. Since Icon automatically converts strings to integers when necessary, the extension had to make sure it did not conflict with existing semantics: it can not blindly assume that a record subscripted by a string is a field name, but must first check to see if the string can be converted to an integer and applied as an integer index.

The second major thing we learned about the implementation language used was regarding the object-oriented preprocessor Idol. Idol object instances are very similar to records--they consist of named fields that may normally be accessed only by operations defined in the instance's class. In order to access an object's fields the object must support field-access methods.

We wished to use objects for the primary lexical structures in our system in order to take advantage of the inheritance mechanism. But lexical elements have an hierarchical feature structure that may be several levels deep. This sort of structure is very naturally expressed using records. Implementing such a structure using objects would entail that many field-access procedures be written, and the end result would be slower than record field accesses.

Since Idol objects are implemented using Icon records, it was a simple matter to lift the field-access restriction and allow object fields to be accessed by normal Icon record field access. This change to Idol compromises its strict object-oriented nature in favor of increased utility. It does not reduce Idol's expressive power or object-oriented functionality, but it reduces objects to more of an organizational tool for Icon programs instead of an all-encompassing execution paradigm.

The Computational Environment

We have completed working prototypes of four environments for the development of attribute grammars. These environments may be used for specifying the following parts of grammars.

- Lexical attributes
- Lexical classes
- Lexical entries
- Syntactic rules

In addition, we have completed two parsing engines for analyzing input text. One of these uses attribute grammars that are developed with the aid of the environments. It is a modification of a standard chart parser [Kay86]. The other uses principles of Head-Driven Phrase-Structure Grammar (HPSG) and lexical items that conform to those principles [Poll87]; the algorithm was developed by Min based on chart-parsing techniques. Since the environment for specifying those lexical items has not yet been fully implemented, we do not report on this parser here.

The Main Menu

When the Twincle program is invoked, a menu appears which allows the user to select an environment to use. Our working prototype displays the menu shown in Figure 1.

Insert Figure 1 about here.

After completing work in any environment, one is returned to the main menu, from which one may also exit the program.

The Lexical Attribute Environment

The lexical attribute environment is the place where one specifies attributes and values for lexical items without regard to their classes. In specifying attributes for lexical classes and for structured attributes (see below), one may restrict their possible values. The updating of possible values for lexical attributes is always carried out in this environment, so that those values are always a superset of the values that those attributes may have in any other environment (e.g., within a particular lexical class or within a structured attribute).

The menu for the lexical attribute environment currently appears as in Figure 2.

Insert Figure 2 about here.

Choosing the option to add a lexical attribute, one is prompted for the name of the attribute. After one types in the name and presses RETURN, one is prompted for its possible values. For example, if one wishes to define an attribute **TAKECOMP** with the possible values **true** and **false**, one enters those names at the appropriate points on the screen, and exits the area where values are specified. One is then prompted for another attribute to define. An example illustrating the specification of the **TAKECOMP** attribute is given in Figure 3.

Insert Figure 3 about here.

One of the options that is available for specifying the values of an attribute is to declare the value as a ***structure***. Selecting this option opens a window in which all of the currently defined attribute names appear, and one simply clicks on the attributes one wishes to specify as contained within this particular structure attribute. At the moment, all of the possible values that are associated with those attributes are declared as possible values for those attributes within a structure attribute. Later, we will enable the user to edit the lists of possible attributes to restrict their values in this environment. For example, if one wishes to define the attribute **SUBJ** as a structure attribute which takes as its value a structure which contains the attribute **NUM** and its associated values, one simply clicks on the ***structure*** option, and then clicks on the **NUM** attribute in the window that opens up. If one wishes to specify that the structure attribute contains an attribute that is not yet defined, one clicks on the ***define new attribute*** option, which returns one to the beginning of the lexical attribute environment for identifying attributes and their possible values. When one has finished specifying this new attribute, one is returned to the structure attribute value environment, which now displays this new attribute. One has a choice now of either accepting or rejecting this attribute as part of the structure attribute one is defining. Upon completing the specification of the structure attribute, one is retured to the beginning of the lexical attribute environment. An illustration showing the definition of the structure attribute **SUBJ** is given in Figure 4.

Insert Figure 4 about here.

The Lexical Class Environment

The lexical class environment is where one specifies lexical classes and their associated attributes. Its menu is shown in Figure 5.

Insert Figure 5 about here.

If one selects the ***add a new lexical class*** option, one is first prompted for the name of the lexical class. After one types it in and presses RETURN, the list of lexical attributes is displayed (as is done also when specifying the values of structure attributes, as described in "The Lexical Attribute Environment" on page 4). One then selects the attributes one wishes to associate with this particular class by clicking on them. Again, if one wishes to define a new attribute, one has the option of doing so, exiting to the lexical attribute environment, and then returning to the lexical class environment in which the new attribute now appears as an option. An illustration showing the specification of the lexical class ***V*** for the attributes **TAKEOBJ** and **SUBJ** is given in Figure 6.

Insert Figure 6 about here.

The Lexical Entry Environment

The lexical entry environment permits one to specify lexical entries as belonging to particular lexical classes with particular lexical attributes and values. The menu for this environment is given in Figure 7.

Insert Figure 7 about here.

If one selects the ***make a new lexical entry*** item, one is prompted for a ***label*** for the lexical entry (e.g. its spelling in a standard orthography). Once one types it in, one is prompted for a lexical class; all the previously defined lexical classes are displayed, and one clicks on the desired class. To the right, a window opens that presents the attributes that are defined for this class, and to the right of the attributes in a pop-down window are the values that are defined for that attribute. One then selects the values, if any, one wishes to associate with that particular lexical entry. In Figures 8 and 9, are given the windows used for specifying that the entry *dog* is a member of the lexical class **N** with the attribute and value specification **NUM: sing** and that the entry *sleep* is a member of the lexical class **V** with the attribute and value specifications **TAKEOBJ: false**, **TAKECOMP: false** and **SUBJ: [NUM: plural]** To define a second set of specifications for a particular label, one must consider it a new entry.

Insert Figures 8 and 9 about here.

At the moment, we enable users to define more than one lexical entry with the same or similar attribute and value structures by editing specific entries and changing the values of the labels or of selected attributes and values. We intend to make further enhancements to the lexical entry environment to enable users to describe related lexical entries in terms of their ***family resemblances***.

One major problem in updating grammars is to redefine entries based on changes made elsewhere in the system. At the moment, we have made no provision for changing lexical entries after other changes have been made to the lexical attribute and lexical class specifications. However, we are aware of the problem and will provide straightforward mechanisms for making such updates.

The Rule Development Environment

Given a set of lexical entries, it is possible to specify rules for combining them into larger expressions and for combining these results further, until the largest linguistically definable entities are created. Twincle provides a rule development environment, whose menu is shown in Figure 10, for defining such rules.

Insert Figure 10 about here.

The ***make a rule*** environment enables users to specify phrase-structure rules which must satisfy certain ***tests*** for particular attribute values and for relations among them, and which carry out certain actions, primarily the assignment of attribute values to the parent. When making a rule, one first specifies the ***arity*** of the rule; i.e., how many children it has. Next, one is prompted for the label of each of the children, starting with the left child, by means of a pop-down window that lists the available labels (initially, just the lexical labels). After the labels of the children have been identified, the attribute structures associated with each child are displayed, and one is asked to indicate what tests, if any one wishes the rule to satisfy. In Figure 11, we give an example of the display for the specification of the rule **VP → V S**.

Insert Figure 11 about here.

A test ***by value*** is one in which the values of particular attributes must be satisfied; i.e., they must not conflict with certain designated values. To specify such a test, one clicks on the attribute one is interested in, whereupon its possible values are displayed. One then clicks on the value that one wishes to designate. For example, in the rule **VP → V S**, one may require that the **TAKECOMP** attribute of **V** not conflict with the value **true**. One specifies this by clicking first on the **TAKECOMP** attribute of the child labeled **V**, and then on the **true** value of that attribute.

A test by ***equality*** is one in which the values of two attributes do not conflict with each other (i.e., those attributes do not differ in specific values). For example, in the rule **NP → Det N**, one may require that the **NUM** attributes of the two children not differ in value (i.e., that it not be the case that one is specified as **sing** and the other as **plural**).

Upon completion of the test specifications, one is then prompted to specify which child if any the parent inherits attributes from; we refer to such a child as the ***head*** of the construction. In addition, one is asked whether the parent should be specified for additional attributes, and whether the parent should have attributes whose values are pointers to the non-head children. Once all of these specifications are made, and one is satisfied with the rule, it is added to the grammar, and the parent class and its associated attributes and values are added to the list of classes that are available for the specification of additional rules.

The Parser

A chart parser has been written which analyzes input strings in accordance with any grammar that has been designed using Twincle. The menu for the parser is shown in Figure 12.

Insert Figure 12 about here.

If one selects the ***Parse data*** option, then one is prompted first to enter the name of a grammatical class, and then to enter a string, which the parser will attempt to determine is analyzable as an instance of that class. If it succeeds, then it displays the attribute structure of the class, and of each of its constituents, as shown in Figure 13 for the string *the dog believes the cat sleeps.* If the string has more than one parse, then each of the parses is displayed in sequence.

Insert Figure 13 about here.

A Sample Grammar

Here is a sample small grammar that has been developed using the environment. The grammar is designed to handle ***number agreement*** between determiners and nouns within noun phrases, and between subject noun phrases and verbs within sentences. It distinguishes between grammatical forms such as *this dog, these dogs, the dog sleeps* and *the dogs sleep* on the one hand, and ungrammatical forms such as **this dogs, *these dog, *the dog sleep* and **the dogs sleeps* on the other. It handles the grammaticality of both *this deer* and *these deer* and of both *the deer sleeps* and *the deer sleep* by ***underspecifying*** the grammatical number of the noun *deer* and the determiner *the.* The grammar also handles a limited amount of recursion, of the sort exemplified by sentences such as *the dog believes the cat sleeps* and *the dog believes the cats believe the geese attack the deer.*

Lexical attributes and values

Attribute	***Values***
NUM	sing, plural
TAKEOBJ	true, false
TAKECOMP	true, false
SUBJ	NUM: sing, plural

Lexical Classes

Class	***Attributes***
Det	NUM
N	NUM
V	TAKEOBJ TAKECOMP SUBJ

Lexical Items

Label	dog, cat, goose
Class	N
NUM	sing

Label	dogs, cats, geese
Class	N
NUM	plural

Label	deer
Class	N

Label.	this
Class	Det
NUM	Sing

Label	these
Class	Det
NUM	plural

Label	the
Class	Det

Label	sleep
Class	V
SUBJ	[NUM: plural]
TAKEOBJ	false
TAKECOMP	false

Label	sleeps
Class	V
SUBJ	[NUM: sing]
TAKEOBJ	false
TAKECOMP	false

Label	chase
Class	V
SUBJ	[NUM: plural]
TAKEOBJ	true
TAKECOMP	false

Label	chases
Class	V
SUBJ	[NUM: sing]
TAKEOBJ	true
TAKECOMP	false

Label	attack
Class	V
SUBJ	[NUM: plural]
TAKECOMP	false

Label	attacks
Class	V
SUBJ	[NUM: sing]
TAKECOMP	false

Label	believe
Class	V
SUBJ	[NUM: plural]

Label	believes
Class	V
SUBJ	[NUM: sing]

Grammatical Classes

Class	*Attributes*
NP	SPEC (pointer to Det) and attributes inherited from N.
VP	OBJ (pointer to NP), COMP (pointer to S) and attributes inherited from V.
S	Attributes inherited from VP.

Rules

1) $S \rightarrow NP\ VP$

Tests	NP and the SUBJ attribute of VP do not conflict.

Actions	S inherits from VP; its SUBJ attribute is the result of unifying the attributes and values of NP and of SUBJ attribute of VP.

2) NP → Det N

Tests	NUM attributes of Det and N do not conflict.
Actions	NP inherits from N; its NUM attribute is result of unifying the NUM attribute and value of Det and N. NP has SPEC attribute whose value is the attribute-value structure of Detrue.

3) VP → V NP

Tests	TAKEOBJ attribute of V does not conflict with true.
Actions	VP inherits from V. VP has OBJ attribute whose value is the attribute-value structure of NP.

4) VP → V S

Tests	TAKECOMP attribute of V does not conflict with true.
Actions	VP inherits from V. VP has COMP attribute whose value is the attribute-value structure of S.

5) VP → V

Tests	TAKEOBJ, TAKECOMP attributes of V do not conflict with false.
Actions	VP inherits from V.

Encoding the Grammatical Structure of Texts

As mentioned above in "Introduction" on page 1, the Twincle system will be extended so that the output of its parser, and also its representations of grammatical specifications, will be encoded in accordance with the guidelines under development by the Text Encoding Initiative for the representation of linguistic analysis. In what follows we present the current state of the guidelines as they have been prepared by the Principal Investigator with an eye toward their incorporation into the output of the Twincle system.

The Twincle construct of a ***structure value*** corresponds to the notion of a ***feature structure***, which is a general purpose data structure for representing many different kinds of information; it is a highly articulated system for encoding complexes of feature-value pairs. Feature structures have their origin in the work of [Knut68], who used the term ***node*** for feature structure and the term ***field*** for feature. More recent treatments can be found in [Shie86] and [Pere87].

Elementary Feature Structures: Features with Binary Values

The fundamental elements of a feature structure system are **<f>** (for ***feature***) and **<fs>** (for ***feature structure***). The simplest **<fs>** elements consist of one or more **<f>** elements. An **<f>** element, in turn, contains a single value; for example, the ***binary values*** represented by the empty tags **<plus>** and **<minus>**.

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]; cf. [Chom68], p. 415.

```
[segment +, consonantal +, vocalic -, nasal -, low -, high -, back -,
round -, anterior +, coronal +, continuant +, delayed release +,
strident +]
```

Here is a possible encoding. Note that **<fs>** elements may have a **type=** attribute which indicates what kind of feature structure it is.

```
<fs type='phonetic segment'>
  <f name=segment><plus>             <f name=consonantal><plus>
  <f name=vocalic><minus>            <f name=nasal><minus>
  <f name=low><minus>                <f name=high><minus>
  <f name=back><minus>               <f name=round><minus>
  <f name=anterior><plus>            <f name=coronal><plus>
  <f name=continuant><plus>          <f name=delayedRelease><plus>
  <f name=strident><plus>
  </fs>
```

Symbolic, Numerical and String Values

By separating out feature values as content of **<f>** elements, we are able to classify those values into ***types***. In section "Elementary Feature Structures: Features with Binary Values" on page 9, the two empty elements which represent binary values are defined. In this section, we define three more feature-value elements: the empty element **<sym>** for expressing ***symbolic values***, the empty element **<nbr>** for expressing ***numerical values***, and the element **<str>** for expressing ***string values***. Unlike **<sym>** and **<nbr>**, **<str>** is not an empty tag.

Symbolic values are values drawn from a closed list of possible string values; these values are represented by the value of the **value=** attribute of the **<sym>** element, as in **<sym value=singular>**. Similarly, numerical values are represented by the value of the **value=** attribute of the **<nbr>** element, as in **<nbr value=10>**. The **<nbr>** element also may have a **type=** attribute specifying whether the value is to be construed as an integer or real number value. String values are values from an open list of possible strings, and are represented by a **<str>** element, whose content is the actual value as in **<str>**Hello, world!**</str>**. An example of a feature structure which contains features with symbolic, numerical and string values is the following representation of the structure of the English word *geese.*

```
<fs type='word structure'>
   <f name=lemma><str>goose</str>
   <f name=category><sym value=noun>
   <f name=barLevel><nbr type=int value=0>
   <f name=number><sym value=plural>
   </fs>
```

Structured Values

Features may have ***structured values*** as well; these values are represented by either the **<fs>** element, or the **fsVal=** attribute on the **<f>** element, which is a pointer to an **<fs>** element. Since an **<fs>** or a pointer to an **<fs>** is permitted to occur as a value of an **<f>**, infinite recursion is possible; i.e., an **<fs>** may contain an **<f>** which may contain or point to an **<fs>**, which may contain an **<f>**, etc. First we give an example of an **<fs>**, which contains six **<f>** tags which have **<fs>** tags as their values. The structure exhibits three degrees of recursion.

```
<fs id=sinksv type='word structure'>
   <f name=form>
      <fs id=sinksvf type='formal structure'>
         <f name=spelling><str>sinks</str>
         <f name=lemma><str>sink</str>
         <f name=affix>
            <fs id=sps type='affix structure'>
               <f name=position><sym value=suffix>
               <f name=spelling><str>s</str>
               </fs>
         </fs>
   <f name=interpretation>
      <fs id=cva0tnmip3ns type='interpretive structure'>
         <f name=stem>
            <fs id=cva0 type='stem structure'>
               <f name=category><sym value=verb>
               <f name=auxiliary><minus>
               </fs>
         <f name=inflection>
            <fs id=tnmip3ns type='inflection structure'>
               <f name=tense><sym value=nonpast>
               <f name=mood><sym value=indicative>
               <f name=agreement>
                  <fs id=p3ns type='agreement structure'>
                     <f name=person><sym value=third>
                     <f name=number><sym value=singular>
                     </fs>
               </fs>
         </fs>
   </fs>
```

Next we provide an encoding of the same structure using the **fsVal=** attribute, on the assumption that the elements **<fs id=sinksvf type='word structure'>** and **<fs id=cva0tnmip3ns type='interpretive structure'>** occur in the document being encoded.

```
<fs id=sinksv type='word structure'>
   <f name=form fsVal=sinksvf>
   <f name=interpretation fsVal=cva0tnmip3ns>
   </fs>
```

The **<fs id=sinksvf type='formal structure'>** element can be similarly encoded as follows, also assuming that the **<fs id=sps type='affix structure'>** element also properly occurs in the document being encoded.

```
<fs id=sinksvf type='formal structure'>
   <f name=spelling><str>sinks</str>
   <f name=lemma><str>sink</str>
   <f name=affix fsVal=sps>
   </fs>
```

Note that when a tag such as **<f name=affix fsVal=sps>** is used, no content is provided for it. If valid nonempty content is provided, application programs should ignore the **fsVal=** attribute.

Grouping Values

Next we introduce the **<valGrp>** element for representing ***grouping values***, which are values that combine any number of other values into one. One use for the **<valGrp>** tag is to represent an ***ambiguous*** value, for example the value of the **<f name=case>** feature associated with the classical Greek word *póleis*, which is the nominative or accusative plural form of *pólis*, 'city', as follows.

```
<!-- ... -->
   <f name=inflection>
     <fs type='inflection structure'>
        <f name=case>
           <valGrp><sym value=nominative><sym value=accusative></valGrp>
        <f name=number><sym value=plural>
        </fs>
<!-- ... -->
```

As it stands however, this representation leaves open the question whether the associated word is to be understood as having both ***nominative*** and ***accusative*** case or as having either ***nominative*** or ***accusative*** case. To make the interpretation specific, a **type=** must be used for **<valGrp>**. (This attribute must also be used for the tag **<fGrp>** that groups features, discussed in section "Feature Groups" on page 17.) The possible values of this attribute are **type=excl** to indicate the ***exclusive disjunction*** of the grouped values, **type=incl** to indicate the ***inclusive disjunction*** of the grouped values, and **type=conj** to indicate the ***conjunction*** of the grouped values. Thus, to indicate that the representation above means the exclusive disjunction of nominative and accusative case values, we may write the following; in section "Echoes" on page 22, we show how a feature structure that contains a **<valGrp type=excl>** element can be disambiguated.

```
<!-- ... -->
   <f name=inflection>
     <fs type='inflection structure'>
        <f name=case>
           <valGrp type=excl><sym value=nominative>
              <sym value=accusative></valGrp>
        <f name=number><sym value=plural>
        </fs>
<!-- ... -->
```

The **<valGrp>** element can also be used to group **<fs>** elements, as in the following example showing the possible interpretations of the English word *sinks*.

```
<f name=interpretation>
   <valGrp id=cnclnp.cva0tnmip3ns type=excl>
      <fs id=cnclnp type='interpretive structure'>
         <f name=stem>
            <fs id=cncl type='stem structure'>
               <f name=category><sym value=noun>
               <f name=common><plus>
               </fs>
         <f name=inflection>
            <fs id=np type='inflection structure'>
               <f name=number><sym value=plural>
               </fs>
      <fs id=cva0tnmip3ns type='interpretive structure'>
         <f name=stem>
            <fs id=cva0 type='stem structure'>
               <f name=category><sym value=verb>
               <f name=auxiliary><minus>
               </fs>
         <f name=inflection>
            <fs id=tnmip3ns type='inflection structure'>
               <f name=tense><sym value=nonpast>
               <f name=mood><sym value=indicative>
               <f name=agreement>
                  <fs id=p3ns type='agreement structure'>
                     <f name=person><sym value=third>
                     <f name=number><sym value=singular>
                     </fs>
               </fs>
         </fs>
```

A **<valGrp>** element which consists entirely of **<fs>** elements can also be pointed at by the **fsVal=** attribute. Thus, assuming the independent existence of the appropriate elements, the preceding structure can also be encoded as follows.

```
<f name=interpretation fsVal=cnc1np.cva0tnmip3ns>
```

This feature, in turn, can appear in an encoding of the analysis of the ambiguous English word *sinks*, as follows.

```
<fs id=sinksws type='word structure'>
   <f name=form fsVal=sinksf>
   <f name=interpretation fsVal=cnc1np.cva0tnmip3ns>
   </fs>
```

Other uses for the **<valGrp>** element are discussed in section "Values as Atoms, Sets and Lists".

Values as Atoms, Sets and Lists

In addition to providing for a number of different types of values for features, we provide for three different ways in which values may be organized, namely as ***atoms***, ***sets*** and ***lists***. To specify how a feature is organized, one may specify the **org=** attribute on the **<f>** element to take on one of the designated values **org=atom**, **org=set** and **org=list**.

In the discussion to this point, we have assumed that values are considered to be atomic elements. However, for many purposes, it is useful to be able to consider the values of certain features to be organized as either sets or lists. For example, suppose one wishes to provide an index-set feature in a feature structure, whose value is a set of numerical indices, say {2, 13, 44}. One method of encoding this feature-value combination is as follows.

```
<f name=indexSet org=set><valGrp type=conj>
   <nbr type=int value=2><nbr type=int value=13><nbr type=int value=44>
   </valGrp>
```

Similarly, suppose one wishes to organize the alternate spellings of a name as a list of strings. Such a feature, with the value ["Kelly" "Kelley"] could be encoded as follows.

```
<f name=altSpell org=list>
   <valGrp type=conj><str>Kelly</str><str>Kelley</str></valGrp>
```

Note that in both these cases, we have grouped the members of the the set and the items of the list within a **<valGrp type=conj>** tag. In the case of sets, this tag is interpreted as forming the set made up of the elements specified within it. Thus repetitions of identical elements is ignored, as is the order in which the elements are specified. The encoding of the set {2, 13, 44} above is equivalent to the following, among infinitely many others.

```
<f name=indexSet org=set><valGrp type=conj>
   <nbr type=int value=2><nbr type=int value=44><nbr type=int value=13>
   </valGrp>

<f name=indexSet org=set><valGrp type=conj>
   <nbr type=int value=2><nbr type=int value=13><nbr type=int value=44>
   <nbr type=int value=2></valGrp>
```

In the case of lists, both order and repetition of the items matter, so the encoding above of the list ["Kelly" "Kelley"] is unique. To encode the list items in the opposite order, one would specify:

```
<f name=altSpell org=list>
   <valGrp type=conj><str>Kelley</str><str>Kelly</str></valGrp>
```

To simplify the encoding of sets and lists which are uniformly of the type **<sym>**, **<nbr>**, **<str>** or **<fs>**, one can omit the **<valGrp type=conj>** tag; thus the initial encodings of the set {2, 23, 44} and of the list ["Kelly" "Kelley"] are to be understood as equivalent to the following.

```
<f name=indexSet org=set>
   <nbr type=int value=2><nbr type=int value=13><nbr type=int value=44>

<f name=altSpell org=list><str>Kelly</str><str>Kelley</str>
```

Any **org=** value on the **<f>** tag can be used with any value on the **<valGrp>** tag, and with any single non-grouping value tag, with the interpretations given in the following table.

ORG VALUE	VALUE NOT VALGRP	VALGRP type=excl	VALGRP type=incl	VALGRP type=conj
atom	specified value	exactly one of specified values	interpretation varies	interpretation varies
set	singleton comprised of value speci- fied	singleton of exactly one of values specified	any subset of set of values specified	set made up of values specified
list	one element list consist- ing of value specified	one element list of exactly one of values specified	any list of values speci- fied	list made up of values specified

When the value of a feature specified as **org=set** is a **<valGrp type=incl>** tag, then the value is understood as any subset of the set made up of the specified values. For example, if one specifies the following as the value of the **<f name=indexSet>**, the value is understood as any subset of the set {2, 13, 44}; that is, any member of the power set of that set.

```
<f name=indexSet org=set><valGrp type=incl>
   <nbr type=int value=2><nbr type=int value=13><nbr type=int value=44>
   </valGrp>
```

Among the subsets of the specified set is the null set, which is the set with no members. To refer to this set directly, we provide the **<null>** element. To specify the null set as the value of <f name=indexSet>, one may write:

```
<f name=indexSet org=set><null>
```

The null set may also be used to specify the null list, which is the list with no items.

When the value of a feature specified as **org=list** is a **<valGrp type=incl>** tag, then the value is understood as any sublist of the list made up of the items in the order given. For example, the following specification means any sublist of the list ["Kelly" "Kelley"]: the null list, the list ["Kelly"], the list ["Kelley"] and the list ["Kelly" Kelley"].

```
<f name=altSpell org=list>
   <valGrp type=incl><str>Kelly</str><str>Kelley</str></valGrp>
```

When the value of a feature specified as **org=set** or **org=list** is a **<valGrp type=excl>** tag, then the value is understood as a set or list of any single element in the group. For example, the value of the **<f name=altSpell org=list>** tag in the following specification is a list whose single item is one of the members of the group, either the list ["Kelly"] or the list ["Kelley"].

```
<f name=altSpell org=list>
   <valGrp type=excl><str>Kelly</str><str>Kelley</str></valGrp>
```

Finally, when the value of a feature specified as **org=set** or **org=list** is a nongrouping element, then the value is understood as a set who single member is the specified element or as a list whose single item is the specified element. For example, the value of the **<f name=indexSet org=set>** tag in the following specification is the set {2}.

```
<f name=indexSet org=set><nbr type=int value=2>
```

When the value of a feature specified as **org=atom** is a **<valGrp>** whose **type=** value is other than *excl*, then the interpretation of the value may vary, and may not even be well defined. For example, both of the following are syntactically well formed specifications, but there may be no interpretation provided for either of them.

```
<f name=spelling org=atom>
   <valGrp type=conj><str>Kelly</str><str>Kelley</str></valGrp>
<f name=index org=atom><valGrp type=incl>
   <nbr type=int value=2><nbr type=int value=13><nbr type=int value=44>
   </valGrp>
```

It may be possible to provide an interpretation for the first of these, say, as the concatenation of the specified strings, and for the second as providing a choice among various index values, including conflated values.

In the discussion so far of the **org=** attribute, we have considered its use on **<f>** elements only. This attribute may also be specified on the **<fDecl>** elements in the feature system declaration discussed in [Simo92]. If the **org=** attribute is specified in the **<fDecl>** tag for a particular feature, then every instance of that feature is understood as having the **org=** specification found in its **<fDecl>**, and the specification may be left off of the **<f>** tags. An **org=** value specification for a particular **<f>** tag that conflicts with the **org=** value specification in its **<fDecl>** tag should be treated as an error.

Boolean, Default and Uncertainty Values

Next we define four special empty value elements: the ***boolean*** elements **<any>** and **<none>**, the **<default>** element, and the **<uncertain>** element. The boolean elements may be used to indicate whether the features they are associated with have values. The element **<any>** corresponds to the boolean value ***true*** (i.e., that the feature it is associated with has a value), and the element **<none>** corresponds to the boolean value ***false*** (i.e., that the feature it is associated with has no value). The **<default>** element may be used to indicate that the feature it is associated with has its default value in the context in which it appears. Finally, the **<uncertain>** element may be used to indicate uncertainty about what value, if any, its associated feature has or even uncertainty about what feature is present.

The elements **<any>**, **<none>** and **<default>** are also designed to be used in conjunction with the **<fDecl>** elements in the feature system declaration discussed in [Simo92]. To illustrate, suppose the **<valRange>** tag in the **<fDecl>** tag for the gender feature is specified as follows.

```
<valRange><valGrp type=excl><sym value=feminine><sym value=masculine>
   <sym value=neuter></valGrp></valRange>
```

Then the representation **<f name=gender><any>** is equivalent to:

```
<f name=gender><valGrp type=excl>
   <sym value=feminine><sym value=masculine><sym value=neuter></valGrp>
```

The interpretation of **<f name=gender><none>** is also restricted to mean that none of the values specified in the **<valRange>** tag for the gender feature are present, but since all other value specifications are also ruled out in virtue of not being in the value range of that feature), there is no practical effect on the interpretation of **<f name=gender><none>** by specifying a value range for that feature.

Next, suppose that the default value for the gender feature is specified in the **<valDefault>** tag of its **<fDecl>** tag as **<sym value=feminine>**. Then the representation **<f name=gender><default>** is equivalent to **<f name=gender><sym value=feminine>**.

An **<f>** tag with no content and no **fsVal=** specified (see section "Structured Values" on page 10) is to be understood as if the **<default>** tag appears as its content. Thus the representation **<f name=gender>** (as part of an **<fs>** tag, with no value specified) is equivalent to **<f name=gender><default>**.

Using such representations as **<f name=gender><any>** and **<f name=gender><default>**, together with an **<fDecl>** tag for the gender feature, can be thought of as ***underspecifying*** the value of the gender feature. In the current illustration, the first means both that the feature has a value and that any legally possible value of the feature may be present; the second means that the one that is normally the value of the feature is present. The ability to underspecify the values of features is based on the notion of ***subsumption*** defined in [Simo92].

The boolean elements **<any>** and **<none>** also have specific uses within **<fsConstraints>** and **<fDecl>** tags in feature system declarations, as described in [Simo92]. For example, the element **<any>** can appear as the value of a feature contained within an **<fs>** of a particular type which appears in the **<cond>** tag of an **<fsConstraints>** tag, to indicate that the feature must appear in feature structures of the designated type (i.e., that it is obligatory) and that when it does appear, it may appear with any of its legal values. Similarly, **<none>** can appear in this way to specify that the feature cannot be present in feature structures of the indicated type (i.e., that it is obligatorily absent from such feature structures).

For example, the following may appear as part of the **<fsConstraints>** of a feature system declaration to indicate that a **<fs type='agreement structure'>** must be specified for a legal value of the number feature but must not be specified for the category feature. All other features that are declared to have values are understood to be optional in such feature structures.

```
<cond><fs type='agreement structure'>
<then><fs>
          <f name=number><any>
          <f name=category><none>
          </fs>
```

We can impose further constraints on feature structures of a particular type in the **<valRange>** tags of other features which take feature structures of that type as values. For example, suppose that **<fs type='agreement structure'>** tags can occur as values of the features **<f name=verbAgreement>** and **<f name=adjAgreement>**, but that when they occur as values of the first feature they must contain the **<f name=person>** feature and not the **<f name=gender>** and **<f name=case>** features. Conversely, when they occur as values of the second feature, they must contain the **<f name=gender>** and **<f name=case>** features but not the **<f name=person>** feature. For the **<f name=verbInflection>**, we can provide the first of the following **<valRange>** tags; for the **<f name=nounInflection>**, we can provide the second of the following **<valRange>** tags. Note that nothing needs to be said about the **<f name=number>** and **<f name=category>** features, since they have already been dealt with in the **<fsConstraints>** tag for feature structures of this type.

```
<valRange>
   <fs type='agreement structure'>
      <f name=person><any>
      <f name=gender><none>
      <f name=case><none>
      </fs>
   </valRange>

<valRange>
   <fs type='agreement structure'>
      <f name=person><none>
      <f name=gender><any>
      <f name=case><any>
      </fs>
   </valRange>
```

As a result of declarations like these, feature structures can also be underspecified in text markup where appropriate. For example, to indicate that the value of a particular instance of the feature **<f**

name=adjInflection> is an **<fs type='inflection structure'>** tag with features specified as plural number and any gender and case, we may write:

```
<f name=adjInflection>
   <fs><f name=number><sym value=plural></fs>
```

The same value when supplied for **<f name=verbInflection>** would be interpreted as an **<fs type='inflection structure'>** with features specified as plural number and any person.

In order not to have to explicitly exclude nonoccurring optional features when marking up instances without encumbering the **<fsConstraints>** and **<valRange>** specifications in the feature system declaration, the default value of the **<valDefault>** tag within the **<fDecl>** tag is specified as **<none>**. Thus, unless a default value other than **<none>** is set for an optional feature, or it is explicitly given a value in a particular instance, optional features within feature structures are assumed not to occur.

It is important to realize that the boolean values **<any>** and **<none>** are very different semantically from the binary values **<plus>** and **<minus>**. The former pair are in fact ***variables*** over possible values and hence cannot be used as specific possible values for features, whereas the latter pair can can be declared as specific possible values for features in the **<valRange>** specifications for those features. For example, the **<valRange>** for the feature **<f name=auxiliary>** may be declared as follows.

```
<valGrp type=excl><plus><minus></valGrp>
```

Then the specification **<f name=auxiliary><plus>** means that the feature has the **<plus>** value; **<f name=auxiliary><minus>** means that the feature has the **<minus>** value; **<f name=auxiliary><any>** means that the feature has either the **<plus>** or the **<minus>** value; and **<f name=auxiliary><none>** means that the feature has no value.

Now suppose that the auxiliary feature is declared to take only the **<plus>** value. Then the specifications **<f name=auxiliary><plus>** and **<f name=auxiliary><any>** are equivalent; **<f name=auxiliary><minus>** is invalid; and **<f name=auxiliary><none>**, as before, means that the feature has no value.

It is even possible to declare that a particular feature can never have values. For example, if the **<valRange>** specification for the feature **<f name=impossible>** is **<valRange></valRange>**, then no specific value can ever be assigned to the impossible feature, and the specifications **<f name=impossible><any>** and **<f name=impossible><none>** are equivalent.

Finally, the specification **<f name=gender><uncertain>** means that it is uncertain what value this particular occurrence of the **<f name=gender>** has, including **<any>**, **<none>** and the possibility that the value is something not specified in the **<fDecl>** for **<f name=gender>**.

To indicate uncertainty about what feature may be present, one may leave off the **name=** attribute of an **<f>** tag whose value is **<uncertain>**, as follows: **<f><uncertain>**.

Feature Groups

We are now in a position to complete the description of the **<fs>** element, which can contain, besides **<f>** elements, also **<fGrp>** elements.

The **<fGrp>** element is to be used whenever **<f>** elements within **<fs>** elements need to be specially grouped. This situation can arise when ambiguity is to be economically represented without the use of embedded **<fs>** elements or **fsVal=** attributes. For example, suppose we choose to represent the structure of the noun and verb forms of the English word *sinks* as follows, using a ***flat*** style of encoding.

```
<fs type='word structure'>
   <f name=spelling><str>sinks</str>
   <f name=lemma><str>sink</str>
   <f name=category><sym value=noun>
   <f name=common><minus>
   <f name=number><sym value=plural>
```

```
<fs type='word structure'>
   <f name=spelling><str>sinks</str>
   <f name=lemma><str>sink</str>
   <f name=category><sym value=verb>
   <f name=auxiliary><minus>
   <f name=tense><sym value=nonpast>
   <f name=mood><sym value=indicative>
   <f name=person><sym value=third>
   <f name=number><sym value=singular>
```

These can be combined within a **<valGrp>** as the value of a **<f name=choice>** element, as follows.

```
<fs type='word structure'>
   <f name=choice>
      <valGrp type=excl>
         <fs type='word structure'>
            <f name=spelling><str>sinks</str>
            <f name=lemma><str>sink</str>
            <f name=category><sym value=noun>
            <f name=common><minus>
            <f name=number><sym value=plural>
            </fs>
         <fs type='word structure'>
            <f name=spelling><str>sinks</str>
            <f name=lemma><str>sink</str>
            <f name=category><sym value=verb>
            <f name=auxiliary><minus>
            <f name=tense><sym value=nonpast>
            <f name=mood><sym value=indicative>
            <f name=person><sym value=third>
            <f name=number><sym value=singular>
            </fs>
         </valGrp>
   </fs>
```

To avoid having to use the **<f name=choice>** element, and having to repeat the lemma and spelling features, the latter can be factored out of the embedded **<fs>**s, and **<fGrp>** tags used instead of the **<valGrp>** and the embedded **<fs>** tags, as in:

```
<fs type='word structure'>
   <f name=lemma><str>sink</str>
   <f name=spelling><str>sinks</str>
   <fGrp type=excl>
      <fGrp type=conj>
         <f name=category><sym value=noun>
         <f name=common><plus>
         <f name=number><sym value=plural>
         </fGrp>
      <fGrp type=conj>
         <f name=category><sym value=verb>
         <f name=auxiliary><minus>
         <f name=mood><sym value=indicative>
         <f name=tense><sym value=nonpast>
         <f name=person><sym value=third>
         <f name=number><sym value=singular>
         </fGrp>
      </fGrp>
   </fs>
```

A more elaborate example showing how the factoring of information into **<fGrp>** elements can be used to avoid repetition is the following, which provides a single structure for some of the various interpretations of the multiply ambiguous word *wash*. The **id**= attributes on various tags have been provided for use in examples in sections "Echoes" on page 22 and "Ambiguity Resolution" on page 23.

```
<fs id=fswash type='word structure'>
   <f name=lemma><str>wash</str>
   <f name=spelling><str>wash</str>
   <fGrp id=fg1 type=excl>
      <fGrp type=conj>
         <f name=category><sym value=noun>
         <f name=common><plus>
         <f name=number><sym value=singular>
         </fGrp>
      <fGrp type=conj>
         <f name=category><sym value=verb>
         <f name=auxiliary><minus>
         <fGrp type=excl>
            <f name=finite><minus>
            <fGrp type=conj>
               <f name=finite><plus>
               <fGrp type=excl>
                  <f name=mood><sym value=subjunctive>
                  <fGrp type=conj>
                     <f name=mood><sym value=imperative>
                     <f name=tense><default>
                     <f name=person><sym value=second>
                     <f name=number><any>
                     </fGrp>
                  <fGrp type=conj>
                     <f name=mood><sym value=indicative>
                     <f name=tense><sym value=nonpast>
                     <fGrp type=excl>
                        <fGrp type=conj>
                           <f name=person><sym rel=ne value=third>
                           <f name=number><any>
                           </fGrp>
                        <fGrp id=fg11 type=conj>
                           <f name=person><sym value=third>
                           <f name=number><sym value=plural>
                           </fGrp>
                        </fGrp>
                     </fGrp>
                  </fGrp>
               </fGrp>
            </fGrp>
         </fGrp>
      </fGrp>
   </fs>
```

The preceding encoding describes the word *wash* as either a singular common noun form or as a non-auxiliary verb form which is either nonfinite or finite. If it is finite then it either is a subjunctive mood form, an imperative mood form (with certain other features defined), or a nonpast indicative mood form. If the latter, then it is either a form which is any person other than third and any number or a form which is third person and plural number.

The REL Attribute

The **rel=** attribute is provided for all value elements except **<valGrp>** and **<uncertain>** to specify whether a value equal to that specified for the feature is to be used, or some other value. This attribute can be specified as **rel=eq** to indicate that the feature value itself is intended, and **rel=ne** to indicate either that any legitimate value not equal to the designated feature value is intended or that no value is intended (if the designated value is in fact not equivalent to **<none>**). A value specified as **rel=ne** can be thought of as the negation of the corresponding value specificied as **rel=eq**. The default specification for the **rel** attribute is **rel=eq**.

In the case of the **<nbr>** and **<str>** elements, the **rel=** attribute may take on other values as well: **rel=lt** indicates any value less than the specified feature value, **rel=le** any value less than or equal to the specified feature value, **rel=gt** any value greater than the specified feature value, and **rel=ge** any value greater than or equal to the specified feature value. The use of these **rel=** values for the **<str>** element requires that a particular character and string ordering (or ***sorting***) convention be specified.

Here are interpretations of feature-value tags with various specifications for the **rel=** attribute.

1) The following equivalences hold among the boolean tags, wherever they occur.

```
<any rel=ne>  == <none>
<none rel=ne> == <any>
```

2) Suppose that the **<valRange>** in the **<fDecl>** specification for the feature **<f name=auxiliary>** is the first one given in "Boolean, Default and Uncertainty Values" on page 15, in which the feature can take on just the specific values **<plus>** and **<minus>**. Then the following equivalences hold.

```
<f name=auxiliary><plus rel=ne>  == <f name=auxiliary><valGrp type=incl>
                                     <minus><none></valGrp>
<f name=auxiliary><minus rel=ne> == <f name=auxiliary><valGrp type=incl>
                                     <plus><none></valGrp>
```

3) Suppose that the **<valRange>** in the **<fDecl>** specification for the feature **<f name=auxiliary>** is the second one specified in "Boolean, Default and Uncertainty Values" on page 15, in which the feature can take on only the specific value **<plus>**. Then the following equivalences hold.

```
<f name=auxiliary><plus rel=ne>  ==  <f name=auxiliary><any rel=ne>  ==
<f name=auxiliary><none>
<f name=auxiliary><none rel=ne>  ==  <f name=auxiliary><plus>        ==
<f name=auxiliary><any>
```

4) Suppose that the feature **<f name=refIndex>** is declared as taking any integer value. The following specifies (i.e., subsumes) any integer value greater than 0 for that feature.

```
<f name=refIndex><nbr rel=gt value=0>
```

5) The following subsumes any integer value greater than 0 and less than 10, i.e. from 1 to 9 inclusive, for that feature.

```
<f name=refIndex><valGrp type=conj><nbr rel=gt value=0>
   <nbr rel=lt value=10></valGrp>
```

6) The following specifications are not equivalent. The first of these subsumes just the numbers that are not equal to 1. The second of these subsumes both those numbers and also no value.

```
<f name=refIndex><valGrp type=excl><nbr rel=gt value=0>
   <nbr rel=lt value=2></valGrp>
<f name=refIndex><nbr rel=ne value=1>
```

7) Suppose that the following feature declaration is made:

```
<fDecl name=indexSet org=set>
   <valRange><valGrp type=conj><nbr type=int rel=gt value=0>
      <nbr type=int rel=lt value=10></valGrp></valRange>
   </fDecl>
```

Then the possible values for **<f name=indexSet>** are the members of the power set of the set {1, 2, 3, 4, 5, 6, 7, 8, 9}.

8) Suppose that the **<valRange>** and **<valDefault>** tags in the **<fDecl>** specification for the feature **<f name=gender>** are as given in section "Boolean, Default and Uncertainty Values" on page 15. Then the following equivalences hold.

```
<f name=gender><default rel=ne>    ==  <f name=gender><valGrp type=excl>
                                          <sym value=neuter>
                                          <sym value=masculine></valGrp>
<f name=gender>                    ==  <f name=gender><valGrp type=excl>
   <sym rel=ne value=masculine>           <sym value=neuter>
                                          <sym value=feminine></valGrp>
```

9) The following specification subsumes any string greater than (presumably longer than) the empty string. Note that this specification is not equivalent to **<str rel=ne></str>**, as the latter also subsumes no value.

```
<str rel=gt></str>
```

10) The following specification subsumes any string less than the string *McQueen* (i.e., any string preceding it in the declared sorting order).

```
<str rel=lt>McQueen</str>
```

11) Suppose that the **<valRange>** tags for the features **<f name=person>** and **<f name=number>** have the following contents.

```
<!-- In <fDecl name=person>  -->
<valGrp type=excl><sym value=first><sym value=second><sym value=third>
   </valGrp>
<!-- In <fDecl name=number>  -->
<valGrp type=excl><sym value=singular><sym value=plural></valGrp>
```

Next, suppose that the feature **<f name=verbAgreement>** has as its possible values all **<fs type='agreement structure'>** tags with the number and person features declared as obligatory and as having any of their possible values, and all other features declared as obligatorily absent.

Finally, suppose that the **<f name=verbAgreement>** is an obligatory feature in whatever feature structures it can occur in. Now consider the following specification.

```
<f name=verbAgreement>
   <fs rel=ne><f name=person><sym value=third>
      <f name=number><sym value=singular></fs>
```

This representation is equivalent under the conditions specified to the following.

```
<f name=verbAgreement>
   <fs><fGrp type=incl>
          <fGrp type=conj>
             <f name=person><valGrp type=excl><sym value=first>
                <sym value=second></valGrp>
             <f name=number><any></fGrp>
          <fGrp type=conj><f name=person><any>
              <f name=number><sym value=plural></fGrp>
          </fGrp></fs>
```

To see how this equivalence is established, first note that since the feature **<f name=verbAgreement>** is declared to be obligatory wherever it occurs, the possibility that the original specification of its value subsumes **<none>** is ruled out.

Second, since a **<fs>** is equivalent to one in which its content is contained in a **<fGrp type=conj>** tag, the content of the example **<f name=verbAgreement>** tag is equivalent to the following.

```
<fs rel=ne><fGrp type=conj><f name=person><sym value=third>
   <f name=number><sym value=singular></fGrp></fs>
```

Third, by the logical equivalence of NEG(CONJ(P, Q)) with DISJ(NEG(P), NEG(Q)), the **rel=ne** specification can be distributed among the values of the features contained within the **<fGrp type=conj>** tag, changing the latter to a **<fGrp type=incl>** tag, as follows.

```
<fs><fGrp type=incl>
   <f name=person><sym rel=ne value=third>
   <f name=number><sym rel=ne value=singular></fGrp></fs>
```

Fourth, the **<fGrp type=incl>** in the preceding representation encompasses the following three possibilities.

```
1.  <f name=person><sym rel=ne value=third>

2.  <f name=number><sym rel=ne value=singular>

3.  <f name=person><sym rel=ne value=third>
    <f name=number><sym rel=ne value=singular>
```

Fifth, the first two of these possibilities are incomplete: the first lacks a value for the obligatory **<f name=number>**, and the second lacks a value for the obligatory **<f name=person>**. Since no specific values can be provided for these missing features, the boolean value **<any>** must be provided, and as a result, the third possibility becomes redundant. Therefore, we can substitute the appropriate **<fGrp type=conj>** tags for the **<f name=person>** and **<f name=number>** specifications in the preceding **<fs>** tag; the value of the originial **<f name=verbAgreement>** tag above is thus equivalent to the following.

```
<fs><fGrp type=incl>
      <fGrp type=conj><f name=person><sym rel=ne value=third>
         <f name=number><any></fGrp>
      <fGrp type=conj><f name=person><any>
         <f name=number><sym rel=ne value=singular></fGrp>
      </fGrp></fs>
```

Finally, we can replace the **<sym rel=ne>** tags in the preceding by their equivalents using **<sym rel=eq>**, which yields the equivalence which was to be shown.

Echoes

The **fsVal=** attribute provides a way of pointing at a value ultimately represented by an **<fs>** element. The **<echo>** element provides a way of pointing at any value, as long as the element representing it has an **id=** value. The **<echo>** tag can be used also to point at **<f>**, **<fGrp>** and root **<fs>** elements, as well as to values of **<f>** elements.

The following encoding is equivalent to that provided in section "Grouping Values" on page 11 for the analysis of the structure of *sinks*.

```
<fs id=sinksws type='word structure'>
   <f name=form><echo target=sinksf>
   <f name=interpretation><echo target=cnclnp.cva0tnmip3ns>
   </fs>
```

Using the **<echo>** element is not recommended when an alternative, such as **fsVal=** is available. Its main utility in the feature structure domain is in making virtual copies of substructures other than **<fs>** elements, particularly of large ones. For example, having created the large **<fGrp id=fg1>** for the representation of *wash* given in "The REL Attribute" on page 19, one can reuse it over and over again in other structures, such as the following, for *wish*.

```
<fs type='word structure'>
   <f name=lemma><str>wish</str>
   <f name=spelling><str>wish</str>
   <fGrp type=excl><echo what=content target=fg1>
<!-- One can also write, instead of the preceding line, the following:
   <echo target=fg1>
  -->
   </fs>
```

On a much smaller scale, one could also provide an **id=** attribute for the value of the **<f name=lemma>** tag, and echo it as the value of following **<f name=spelling>** tag, as in:

```
<fs type='word structure'>
   <f name=lemma><str id=wishstr>wish</str>
   <f name=spelling><echo target=wishstr>
<!-- ... -->
```

Ambiguity Resolution

When encoding an inherently ambiguous segment whose interpretation is contextually resolved, one may have occasion to represent it as a ***disambiguated*** segment. That is, one may wish to represent not just its interpretation in context, nor just its ambiguous interpretation out of context, but the two together. For this purpose, the **select=** attribute is provided. This attribute is to be used either in elements which contain elements with the **exclude=** attribute or in **<fs>** elements which contain grouping elements with the **type=excl** attribute. In this section, we consider its use together with the **type=excl** attribute on grouping elements.

First, note that if the **select=** attribute appears in a **<fs>** which does not contain any element in which the **type=excl** specification appears, nor any element in which the **exclude=** attribute appears, it has no effect. For example, the following two encodings are equivalent.

```
<fs type='agreement structure' select=p3>
   <f name=person><sym id=p3 value=third>
   <f name=number><sym id=ns value=singular>
   </fs>
<fs type='agreement structure'>
   <f name=person><sym id=p3 value=third>
   <f name=number><sym id=ns value=singular>
   </fs>
```

Now consider the following **<fs>** (which may be part of the representation of the analysis of the English word *was*), which does contain a grouping tag with the **type=excl** specification.

```
<fs type='agreement structure' id=p1p3ns>
   <f name=person>
      <valGrp excl>
         <sym id=p1 value=first>
         <sym id=p3 value=third>
         </valGrp>
   <f name=number><sym id=ns value=singular>
   </fs>
```

To disambiguate it, we can create a virtual copy and place a **select=** specification on it, as follows.

```
<fs type='agreement structure' id=s1p3p1p3ns select=p3>
   <echo what=content target=p1p3ns>
   </fs>
```

In this encoding, the effect of the **select=p3** specification is to indicate that the structure has been disambiguated, so as to be equivalent to a virtual copy of the following.

```
<fs type='agreement structure'>
   <f name=person><sym id=p3 value=third>
   <f name=number><sym id=ns value=singular>
   </fs>
```

To see how this works, first note that the exclusive disjunction of a group of elements is equivalent to the inclusive disjunction of those elements in which each one excludes the others. In particular, the following encodings are equivalent.

```
<valGrp type=excl>
   <sym id=p1 value=first>
   <sym id=p3 value=third>
   </valGrp>
<valGrp type=incl>
   <sym id=p1 exclude=p3 value=first>
   <sym id=p3 exclude=p1 value=third>
   </valGrp>
```

Each of these encodings specifies that the element identified as *p1* is present if and only if the element identified as *p3* is not. The effect of the specification **select=p3** in the **<fs>** element is that the element identified as *p3* is present in that structure. Hence the element identified as *p1* is not. Finally, since a **<valGrp>** element which effectively encloses exactly one element is equivalent to its enclosed element no matter what its type, the disambiguation of the structure results.

Note that although the following encoding is equivalent to the one identified as *p1p3ns* (assuming an appropriate feature system declaration), it cannot be disambiguated because there is no appropriate identifier for a **select=** attribute to point to.

```
<fs type='agreement structure' id=nep2ns>
   <f name=person><sym id=nep2 rel=ne value=second>
   <f name=number><sym id=ns value=singular>
   </fs>
```

One can use the **select=** attribute to indicate partial disambiguation, as well as complete disambiguation. For example, the following example indicates that of the three mutually exclusive values in the unselected structure, the two that are pointed at by the **select=** attribute remain available.

```
<fs type='inflection structure' select='cn cd'>
   <f name=case>
      <valGrp type=excl>
         <sym id=cn value=nominative>
         <sym id=cd value=dative>
         <sym id=ca value=accusative>
         </valGrp>
   <f name=number><sym value=singular>
   </fs>
```

Another example of partial disambiguation involves a structure which contains two independent exclusive disjunctions. Consider the following encoding of the German pronoun *sie*.

```
<fs id=fscpsie type='word structure'>
   <f name=lemma><str>sie</str>
   <f name=category><sym value=pronoun>
   <f name=case>
      <valGrp type=excl>
         <sym id=csnm value=nominative>
         <sym id=csac value=accusative>
         </valGrp>
   <fGrp type=excl>
      <fGrp id=gefmnusg type=conj>
         <f id=gefm name=gender><sym value=feminine>
         <f id=nusg name=number><sym value=singular>
         </fGrp>
      <fGrp id=ge01nupl type=conj>
         <f id=ge01 name=gender><any>
         <f id=nupl name=number><sym value=plural>
         </fGrp>
      </fGrp>
   </fs>
```

To indicate that an occurrence of this segment is feminine-gender, singular-number, but leaving unresolved its case, one may associate with it the following structure:

```
<fs id=fscpsiel type='word structure'>
   <echo what=content target=fscpsie select='gefmnusg'>
```

If an exclusive disjunction contains another exclusive disjunction, and all the values of the **select=** attribute point into the second disjunction, then the outer disjunction is implicitly disambiguated (the contained exclusive disjunction just mentioned is selected). Thus the elaborate structure which encodes the ambiguity of the English word *wash* in section "Feature Groups" on page 17 can be disambiguated quite simply. For example, to indicate that an occurrence is understood as a finite third-person plural-number nonpast-tense indicative-mood verb form, one can associate with it a structure like the following.

```
<fs id=fswashdll type='word structure' select=fgll>
   <echo what=content target=fswash>
   </fs>
```

References

[Alle87] Allen, J. *Natural Language Understanding*. Menlo Park, CA: Benjamin/Cummings, 1987.

[Chom68] Chomsky, N. and Halle, M. *The Sound Pattern of English*. New York: Harper and Row, 1968.

[Gold90] Goldfarb, C.F. *The SGML Handbook*. Oxford: Clarendon Press, 1990.

[Gris90] Griswold, R.E. and Griswold, M.T. *The Icon Programming Language*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1990.

[Jeff90] Jeffery, C.L. *Programming in Idol: An Object Primer*. Technical Report 90-10b, Department of Computer Science, University of Arizona, January 1990.

[Jeff92] Jeffery, C.L. *X-Icon: An Icon Window Interface*. Technical Report 91-1c, Department of Computer Science, University of Arizona, February 1992.

[Kay86] Kay, M. "Algorithm schemata and data structures in syntactic processing" In Grosz, B.J., Sparck-Jones, K. and Webber, B.N., eds., *Readings in Natural Language Processing*, pp. 35-70. Los Altos, CA: Morgan-Kaufman, 1986.

[Knut68] Knuth, D.E. *The Art of Computer Programming, Vol. I: Fundamental Algorithms*. Reading, MA: Addison-Wesley, 1968.

[Pere87] Pereira, F. *Grammars and Logics of Partial Information*. SRI International Technical Note 420. Menlo Park, CA: SRI International, 1987.

[Poll87] Pollard, J. and Sag, I.A. *Information-Based Syntax and Semantics: Volume 1: Fundamentals* (CSLI Lecture Notes 13). Stanford, CA: Center for the Study of Language and Information, 1987.

[Shie86] Shieber, S. *An Introduction to Unification-based Approaches to Grammar* (CSLI Lecture Notes 4). Stanford, CA: Center for the Study of Language and Information, 1986.

[Simo92] Simons, G. *Feature System Declarations*. Technical Report of the Analysis and Interpretation Committee, Text Encoding Initiative. Available from Computer Center, University of Illinois at Chicago, 1992.

[Sper90] Sperberg-McQueen, C.M. and Burnard, L., eds. *Guidelines for the Encoding and Interchange of Machine-Readable Texts*, Draft Version 1.1. Chicago and Oxford: Text Encoding Initiative, November 1990.