The Compleat LKB

Ann Copestake

Computer Laboratory
University of Cambridge

August 1993

Abstract

This report is a full description of the lexical knowledge base system (LKB) and the representation language (LRL) developed on the Esprit ACQUILEX project. The LKB system is designed to allow the representation of multilingual lexical information in a way which integrates lexical semantics with syntax and formal semantics. The LRL is a typed feature structure language which makes it possible to represent the lexicon as a highly structured object and to capture relationships between individual word senses by (default) inheritance and by lexical rules. The extension to multilingual representation allows a concise and natural description of translation mismatches. Most of this report consists of a detailed formal description of the LRL — this is augmented with appendices containing the user manual, an implementation outline and a discussion of some of the algorithms used, and a bibliography of papers which describe the LKB and its use within ACQUILEX. (Some of this material has been published previously, but is included here to make this report a convenient reference source.)

Contents

Chapter 1

Introduction

This report is a full description of the ACQUILEX lexical knowledge base (LKB) system and the lexical representation language (LRL).[1] To a large extent it is an amalgamation of material from other sources. In particular, Chapters 2 and 3 are largely taken from Copestake (1992) and Chapters 4 and 5 come from Copestake (1993). This report does not describe our use of the LKB system in any more detail than is necessary to explain and motivate the formalism. An annotated bibliography of papers describing the LKB and its utilisation within the ACQUILEX projects is given in Appendix E.

In general terms, our goal in developing the LRL was to produce a language which allowed us to represent multilingual lexical information and in particular to integrate lexical semantics with syntax and formal semantics. This language had to be formally defined and possible to implement in a computationally tractable fashion. Since we are concerned with the ultimate development of very large lexicons, the representation has to capture generalisations, not just because this makes the lexicon smaller for processing but, more importantly, because it makes it much easier to maintain and develop. The more novel features in the ACQUILEX LRL (which we will sometimes refer to as LAUREL) were mostly designed in order to allow the lexicon to be structured for maximum economy of representation. Although we are primarily concerned with lexical representation, a lexicon is not an end in itself, but has to be developed and evaluated in the context of some assumptions about the use of the lexical entries, in particular in parsing, generation and machine translation (MT). The LKB supports this, in that grammar rules, lexical rules and translation equivalence statements may be described in the LRL and the LKB system allows such descriptions and the lexical entries to be tested, since the system incorporates a simple chart parser and a general constraint resolution mechanism.

The first ACQUILEX project in particular was largely concerned with the (semi-) automatic extraction of lexical information from MRDs. However, our philiosophy was that the formalism used for the LKB should not be based on this, since our objective was to build lexical knowledge base fragments which were suitable for NLP and thus should, as far as possible, be motivated by their adequacy as (computational) linguistic representations, independent of the source of the information. The use of MRDs did influence the language in some respects (although we would claim that there is nothing there which is specific

[1]This work was carried out as part of the ACQUILEX projects BRA-3030 and 7315 supported by Esprit. Although the LKB and LRL were mainly developed by the current author, a substantial contribution was made by other researchers on the projects, in particular, Ted Briscoe, Bernie Jones, Valeria de Paiva and Antonio Sanfilippo. I am also very grateful to Bob Carpenter, John Carroll, Roger Evans, Ewan Klein and especially Gerald Gazdar for their comments and advice.

to lexical entries extracted from MRDs). In particular the default inheritance mechanism was designed to make it convenient to represent lexical semantic inheritance hierarchies which we have found can be extracted from MRDs.

1.1 Unification based representation languages

The adoption of a unification-based approach seems natural, given the goal of integrating the representation of syntax and semantics. Unification based approaches also allow the paradigmatic and syntagmatic aspects of representation to be tightly interrelated, which is particularly important because there are close similarities between some lexical processes and others which would traditionally be classed as morphological or grammatical.

LAUREL belongs in the tradition of unification-based language such as PATR-II (Shieber *et al.*, 1983; Shieber, 1986) and FUG (Kay, 1979, 1984) which are not tied to any particular linguistic theory, but are sufficiently flexible that they can be used to implement a range of possible approaches. This makes experimentation much easier, but has the disadvantage that the language itself provides few constraints on the theory: a lexical entry could contain almost anything, for example. Languages such as PATR have no mechanism for stating restrictions that apply globally, but some more recent unification-based languages (see e.g. Calder, 1987; Carpenter, 1990, 1992; Emele and Zajac, 1990; Zajac, in press) allow the specification of a *type system* which gets round this problem to a considerable extent. Using a type system it is possible to specify, for example, that lexical entries have precisely the three components phonology, syntax and semantics, and to limit the possible values of these components. In principle, at least, it is possible to make a clear distinction between the global constraints imposed by the linguistic theory and the particular instantiation embodied by the lexical items which have been encoded. Typing can be used as a way of capturing generalisations. Template mechanisms in untyped languages, such as PATR, allow generalisations in the lexicon to be captured by inheritance, although templates, unlike types, do not restrict the class of linguistic entities. Furthermore, the effect of templates is restricted to the language for description of structures, whereas types are an integral part of the structure, and thus can affect the results of combination of entities in a way which templates cannot.

We suggested above that unification-based approaches allowed tight relationships between the paradigmatic and syntagmatic aspects of representation. This is based on the assumption that the same underlying operation of unification should be used both to capture the paradigmatic structure of the lexicon and to account for syntagmatic combination. The language for description of lexical entries allows structures which correspond to parts of entries, or underspecified entries, to be combined using exactly the same unification operation as that for syntagmatic combination. But this assumption is certainly not necessary. DATR (Evans and Gazdar, 1990) is a language for the description of feature structures which is not itself based on the unification of feature structures. One reason why such a differentiation might be desirable is that it seems that the most succinct and elegant descriptions of many aspects of lexical structure involve the use of default inheritance (see, for example Flickinger, 1987), while it is frequently assumed that combination in the syntagmatic plane should not involve any notion of defeasibility. If this is accepted, the lexical description language necessarily incorporates operations which are not allowed in the combination of lexical entries. We will discuss the use of defaults in lexical representation in more detail in Chapter 3.

However, there are advantages in retaining a close correspondence between paradigmatic and syntagmatic representations. The interface between DATR and a parser is non-trivial (see Kilbury *et al.*, 1991) because DATR does not incorporate a notion of reentrancy. If the benefits of typing are accepted, it is desirable that the type system operates over both paradigmatic and syntagmatic components. Finally, the theoretical division between the lexicon and the rest of the system is not completely clearcut, particularly when extended senses and derivational morphology are considered (this point is discussed in more detail in Copestake, 1992:Ch 4). Minimising the distinction means that the representation language is too general, in that the stipulation that defaults are not used in syntagmatic combination, for example, is somewhat arbitrary, but it does allow such issues to be investigated more easily, and is thus appropriate in a relatively 'theory-neutral' language.

1.2 LAUREL

LAUREL can be viewed as an augmentation of a typed graph-based unification formalism with default inheritance; default inheritance is formalised in terms of default unification of feature structures (e.g. Carpenter, 1991). The typed feature structure language is based on Carpenter's (1990, 1992) work, although there are some significant differences. The type system can be regarded as a way of providing (non-default) inheritance, combined with error-checking. The notion of types, and features appropriate for a given type, gives some of the functionality of frame representation languages, such as KL-ONE (Brachman and Schmolze 1985); in particular, classification of a feature structure is possible. The most recent version of LAUREL incorporates a type driven constraint resolution mechanism which can be used for parsing, generation and machine translation.

The typed feature structure language is augmented with a default inheritance mechanism. This can be used to organise the lexicon in a completely user-defined way, to allow morphological or syntactic information to be concisely specified, for example, as has been done with DATR and other systems (for example, Russell *et al.*, 1991; Krieger and Nerbonne, in press). However, much of the motivation behind the formalisation of default inheritance in LAUREL comes from the concept of using inheritance between lexical entries to structure the lexicon along a semantic dimension.

Thus the operations that LAUREL supports are (default) inheritance, (default) unification, rule application (which can be formalised in terms of inheritance and unification) and constraint resolution. The type system provides the non-default inheritance mechanism and constrains default inheritance. Lexical rules can be used as a further component of lexical structure, but are also constrained by the type system.

Compared with many recent unification based formalisms, LAUREL is in some ways an impoverished language. It does not support disjunction of feature structures, negation or implication. Although it is based on Carpenter's work, it does not incorporate inequalities nor the extensional/intensional distinction which he describes. Some of these extensions may well turn out to be desirable but the aim here was to start with a relatively straightforward language and to extend it in response to the particular problems which arose.

1.3 An example of representation in LAUREL

As an introductory example consider the following lexical entry for rabbit:

$$\left[\begin{array}{l} \textbf{lex-count-noun} \\ \text{ORTH} = \textbf{string} \\ \text{CAT} = \boxed{\textbf{noun-cat}} \\ \text{SEM} = \left[\begin{array}{l} \textbf{obj-noun-formula} \\ \text{IND} = \boxed{0}\left[\begin{array}{l}\textbf{obj}\\ \text{AGR} = \left[\begin{array}{l}\textbf{agr}\\ \text{NUM} = \textbf{sg}\end{array}\right]\end{array}\right] \\ \text{PRED} = \textbf{pred} \\ \text{ARG1} = \boxed{0} \\ \text{PLMOD} = \textbf{false} \\ \text{QUANT} = \textbf{true}\end{array}\right] \\ \text{QUALIA} = \left[\begin{array}{l} \textbf{nomqualia} \\ \text{AGENTIVE} = \textbf{nomagent} \\ \text{TELIC} = \boxed{\textbf{verb-sem}} \\ \text{FORM} = \left[\begin{array}{l}\textbf{nomform}\\ \text{ABSOLUTE} = \textbf{real-form} \\ \text{RELATIVE} = \textbf{individual}\end{array}\right] \\ \text{CONSTITUENCY} = \left[\begin{array}{l}\textbf{nomconst}\\ \text{PARTICLES} = \textbf{string}\end{array}\right] \\ \text{OBJECT-INDEX} = \boxed{0}\end{array}\right]\end{array}\right]$$

Figure 1.1: The psort `lex-individual`

```
rabbit 1
   <> < lex-individual <>
   < QUALIA > < animal_1 <>.
```

The notation will be explained fully in the next chapter: `<` and `>` are used to delimit paths of zero or more features, and `<` is used to stand for default inheritance. The orthography "rabbit" is implicitly connected with the predicate **rabbit_1**. The inheritance from `lex-individual` means that the sign as a whole is stated to have a combination of syntactic, formal semantic and lexical semantic properties associated with individual denoting nouns. We refer to named feature structures such as `lex-individual` from which another structure inherits as *psorts* — the structure for `lex-individual` is shown in Figure 1.1.[2]

The lexical semantic properties of noun signs are associated with the feature QUALIA and thus further information is inherited from the lexical semantic portion of the entry for *animal 1* (thus `animal_1` is also a psort, shown in Figure 1.2). From this it follows that the overall lexical semantic type of the entry for *rabbit* is **animal**. (The similarity in the names of the psort and the type is coincidental.)

From this lexical entry the typed feature structure shown in Figure 1.3 is produced automatically in LAUREL. The way that this expansion happens will be discussed in detail in Chapter 2, but in this case no defaults are overridden.

1.4 Outline

The rest of this report consists of four main chapters. The first describes the non-default aspects of the LRL in detail, the second discusses defaults. The third chapter is a short description of the use of the system for constraint resolution, and the fourth details how this

[2] From now on we will use following conventions: types are represented in boldface font, features in capitals, identifiers for lexical entries and other feature structures in lowercase in typewriter font (e.g. `animal_1`). A box round a type in displayed feature structure diagrams indicates that that portion of the feature structure is not shown. Boxed numerals in feature structure diagrams are used to indicate reentrancy.

$$
\left[\begin{array}{l}
\textbf{lex-count-noun} \\
\text{ORTH} = \textbf{animal} \\
\text{CAT} = \boxed{\textbf{noun-cat}} \\
\text{SEM} = \left[\begin{array}{l}
\textbf{obj-noun-formula} \\
\text{IND} = \boxed{0} \left[\begin{array}{l} \textbf{obj} \\ \text{AGR} = \left[\begin{array}{l} \textbf{agr} \\ \text{NUM} = \textbf{sg} \end{array}\right] \end{array}\right] \\
\text{PRED} = \textbf{animal\_1} \\
\text{ARG1} = \boxed{0} \\
\text{PLMOD} = \textbf{false} \\
\text{QUANT} = \textbf{true}
\end{array}\right] \\
\text{QUALIA} = \left[\begin{array}{l}
\textbf{animal} \\
\text{AGENTIVE} = \left[\begin{array}{l} \textbf{agentive} \\ \text{ORIGIN} = \textbf{basic} \end{array}\right] \\
\text{TELIC} = \boxed{\textbf{verb-sem}} \\
\text{FORM} = \left[\begin{array}{l} \textbf{nomform} \\ \text{ABSOLUTE} = \textbf{real-form} \\ \text{RELATIVE} = \textbf{individual} \end{array}\right] \\
\text{CONSTITUENCY} = \left[\begin{array}{l} \textbf{nomconst} \\ \text{PARTICLES} = \textbf{string} \end{array}\right] \\
\text{OBJECT-INDEX} = \boxed{0} \\
\text{PROPERTIES} = \left[\begin{array}{l} \textbf{creature-properties} \\ \text{STATE} = \textbf{solid\_a} \\ \text{SEX} = \textbf{gender} \end{array}\right]
\end{array}\right]
\end{array}\right]
$$

Figure 1.2: Expanded lexical entry for *animal* 1

$$
\left[\begin{array}{l}
\textbf{lex-count-noun} \\
\text{ORTH} = \textbf{rabbit} \\
\text{CAT} = \boxed{\textbf{noun-cat}} \\
\text{SEM} = \left[\begin{array}{l}
\textbf{obj-noun-formula} \\
\text{IND} = \boxed{0} \left[\begin{array}{l} \textbf{obj} \\ \text{AGR} = \left[\begin{array}{l} \textbf{agr} \\ \text{NUM} = \textbf{sg} \end{array}\right] \end{array}\right] \\
\text{PRED} = \textbf{rabbit\_1} \\
\text{ARG1} = \boxed{0} \\
\text{PLMOD} = \textbf{false} \\
\text{QUANT} = \textbf{true}
\end{array}\right] \\
\text{QUALIA} = \left[\begin{array}{l}
\textbf{animal} \\
\text{AGENTIVE} = \left[\begin{array}{l} \textbf{agentive} \\ \text{ORIGIN} = \textbf{basic} \end{array}\right] \\
\text{TELIC} = \boxed{\textbf{verb-sem}} \\
\text{FORM} = \left[\begin{array}{l} \textbf{nomform} \\ \text{ABSOLUTE} = \textbf{real-form} \\ \text{RELATIVE} = \textbf{individual} \end{array}\right] \\
\text{CONSTITUENCY} = \left[\begin{array}{l} \textbf{nomconst} \\ \text{PARTICLES} = \textbf{string} \end{array}\right] \\
\text{OBJECT-INDEX} = \boxed{0} \\
\text{PROPERTIES} = \left[\begin{array}{l} \textbf{creature-properties} \\ \text{STATE} = \textbf{solid\_a} \\ \text{SEX} = \textbf{gender} \end{array}\right]
\end{array}\right]
\end{array}\right]
$$

Figure 1.3: Expanded lexical entry for *rabbit* 1

may be used in machine translation. The conclusion briefly describes some areas of research leading on from work on the ACQUILEX LKB which are under current investigation.

Chapter 2

The lexical representation language

This chapter covers the definitions of typed feature structures used in LAUREL, the syntax of the description language and the non-default aspects of its semantics. In order to make the definitions more comprehensible, we start by considering untyped feature structures. This chapter is almost entirely taken from Copestake (1992:Ch 2).

2.1 Untyped feature structures

In this section we briefly recapitulate some of the basic definitions of untyped feature structures, including subsumption, unification and generalisation. This description closely follows Carpenter (1991), however we will use a notation in which more specific entities (feature structures and, later on, types) are uniformly described as being lower in some hierarchy.

A feature structure is either basic, in which case it must be an *atomic value*, or it is complex, in which case it provides values for one or more *features*. These values in turn are either atomic or complex feature structures. The set Feat of features and set Atom of atomic values are assumed to be finite. Feature structures can be defined in terms of labelled finite-state automata following Kasper and Rounds (1986, 1990).

Definition 1 (Feature Structure) *A feature structure is a tuple* $F = \langle Q, q_0, \delta, \alpha \rangle$ *where:*

- Q*: a finite set of nodes*
- $q_0 \in Q$*: the root node*
- $\delta : \mathsf{Feat} \times Q \to Q$ *: the partial feature value function (transition function)*
- $\alpha : Q \to \mathsf{Atom}$ *: the partial atomic value function*

with the following constraints:

Connectedness *There must be some* path *from the root to every node.*

Paths are defined as sequences of zero or more features. Let $\mathsf{Path} = \mathsf{Feat}^*$ *and* ϵ *be the empty path. The definition of the transition function can be extended to paths by taking* $\delta(\epsilon, q) = q$ *and* $\delta(f \cdot \pi, q) = \delta(\pi, \delta(f, q))$*. Then the connectedness condition can be expressed as stating that there must be a path* π *such that* $q = \delta(\pi, q_0)$ *for all* $q \in Q$

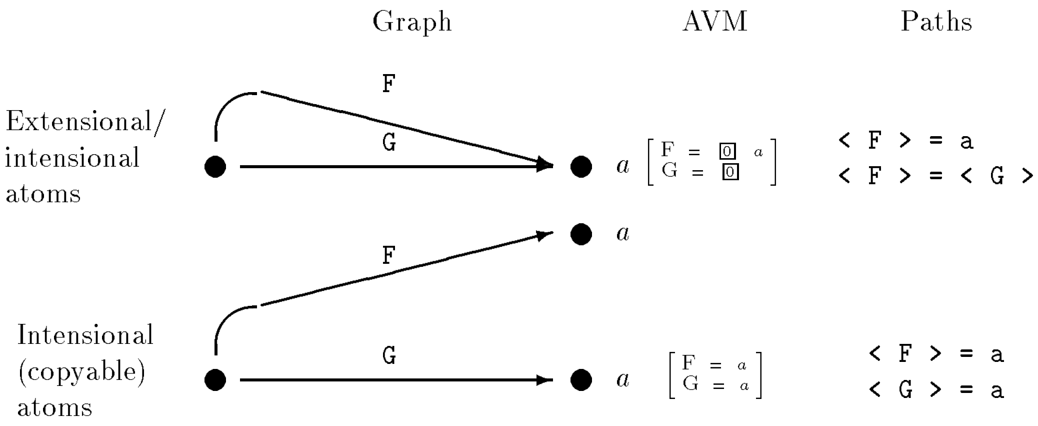


Figure 2.1: Extensional vs. intensional behaviour of atoms

Atomic values *Only nodes without features can be atomic values so that if $\alpha(q)$ is defined then $\delta(f, q)$ is undefined for every $f \in$* Feat (But note that not necessarily all of the nodes without features are labelled with atomic values.)

Acyclicity *The resulting graph is acyclic in that there is no path π and non-empty path π' such that $\delta(\pi, q_0) = \delta(\pi \cdot \pi', q_0)$.*
(Although it is not necessary to rule out cycles in untyped or typed feature structures doing so makes some definitions simpler and the implementation more straightforward and so the acyclicity condition was adopted in LAUREL.)

In Pereira and Shieber (1984) the atomic value function α is required to be one-to-one; that is atoms are treated as *extensional* or non-copyable (see Carpenter, 1992:35–36). This requirement is not adopted here and so all feature structures are treated as intensional. Thus the feature structures shown in Figure 2.1 are both possible and distinct, whereas under an extensional treatment only the first is allowed (although, because there is no ambiguity, it would be usual to write it without making the reentrancy explicit in the AVM notation, as in the matrix for the intensional example). Although this distinction is not particularly important in practice when using untyped feature structures and non-default operations, since the structures are equivalent with respect to unification, there is a difference in behaviour with typed feature structures, since atomic values may be further specialised, and, arguably, there should also be a difference with default unification on untyped feature structures (see Section 3.1). Carpenter (1992:124f) discusses extensionality and intensionality and generalises the concept so that all types, not just atomic values, may be either intensional or extensional. However, for simplicity, the treatment adopted in LAUREL is purely intensional.

2.1.1 Subsumption

Feature structures can be regarded as being ordered by information content — a feature structure is said to *subsume* another if the latter carries extra information. Subsumption can be formalised by assuming that a pair of sets is associated with each feature structure which determine the path equivalences that hold and the atomic values assigned to paths. Let $\equiv_F$ be the equivalence relation induced between paths by the structure sharing in F and $\mathcal{P}_F$ be the partial function induced by F which maps paths in F to atomic values.

Definition 2 (Abstract Feature Structure) *If $F = \langle Q, q_0, \delta, \alpha \rangle$ is a feature structure, we let $\equiv_F \subseteq \mathsf{Path} \times \mathsf{Path}$ and $\mathcal{P}_F : \mathsf{Path} \to \mathsf{Atom}$ be such that:*

- *(Path Equivalence)*
  $\pi \equiv_F \pi'$ *if and only if* $\delta(\pi, q_0) = \delta(\pi', q_0)$

- *(Path Value)*
  $\mathcal{P}_F(\pi) = \sigma$ *if and only if* $\alpha(\delta(\pi, q_0)) = \sigma$.

The pair $\langle \mathcal{P}, \equiv_F \rangle$ is called the abstract feature structure corresponding to F.

A feature structure F subsumes another feature structure F' if and only if the information in F is contained in the information in F'; that is, if F' provides at least as much information about path values and structure sharing as F. The abstract feature structure corresponding to F is sufficient to determine its information content and thus subsumption.

Definition 3 (Subsumption) *F subsumes F', written $F' \sqsubseteq F$, if and only if:*

- $\pi \equiv_F \pi'$ *implies* $\pi \equiv_{F'} \pi'$

- $\mathcal{P}_F(\pi) = \sigma$ *implies* $\mathcal{P}_{F'}(\pi) = \sigma$

Thus F subsumes F' if and only if every piece of information in F is contained in F'. For untyped feature structures, $\top$ is defined to be the single node feature structure with no atomic value assigned. Thus $\pi \equiv_\top \pi'$ if and only if $\pi = \pi' = \epsilon$ and $\mathcal{P}(F)$ is undefined everywhere. Note that $F \sqsubseteq \top$ for every feature structure F. $\top$ is used in attribute-value matrices to denote the lack of any known value.

Feature structures which only vary in the identity of their nodes are said to be *alphabetic variants*. Thus if $F \sqsubseteq F'$ and $F' \sqsubseteq F$, then F and F' are alphabetic variants and we write $F \sim F'$. The definitions of operations on feature structures such as unification are not sensitive to any distinction between alphabetic variants.

2.1.2 Unification

Unification corresponds to conjunction of information, and thus can be defined in terms of subsumption, which is a relation of information containment. The unification of two feature structures is defined to be the most general feature structure which contains all the information in both of the feature structures.

Definition 4 (Unification) *The unification $F \sqcap F'$ of two feature structures F and F' is taken to be the greatest lower bound of F and F' in the collection of feature structures ordered by subsumption.*

Thus $F \sqcap F' = F''$ if and only if $F'' \sqsubseteq F$, $F'' \sqsubseteq F'$ and for every F''' such that $F''' \sqsubseteq F$ and $F''' \sqsubseteq F'$ it is also the case that $F''' \sqsubseteq F''$.

2.1.3 Generalisation

Generalisation is the opposite of unification in which the lowest upper bound of two feature structures is taken. The generalisation of two feature structures is defined to be the most specific feature structure which contains only information found in both feature structures.

Definition 5 (Generalisation) *The generalisation $F \sqcup F'$ of two feature structures is defined to be their greatest lower bound in the subsumption ordering.*

Thus $F \sqcup F' = F''$ if and only if $F \sqsubseteq F''$, $F' \sqsubseteq F''$ and for every F''' such that $F \sqsubseteq F'''$ and $F' \sqsubseteq F'''$ then $F'' \sqsubseteq F'''$. Although unification corresponds to conjunction, generalisation does not correspond to disjunction but is more like information intersection. In languages which support it, disjunction of feature structures will give a result which is either more specific than that produced by generalisation (defined as an operation which produces a non-disjunctive feature structure) or equal to it.

2.2 Typed feature structures

As stated earlier, the approach to typed feature structures adopted here is based on that of Carpenter (1992), but whereas Carpenter describes both appropriateness conditions and constraints on types, here the system is simplified and a notion of constraint alone is used, which is, however, less expressive than that employed by Carpenter in that disjunction is not allowed. Constraints in LAUREL do support type checking and classification in much the same way as Carpenter's appropriateness conditions do and can alternatively be regarded as a generalisation of the notion of appropriateness conditions.[1] A type system defined in LAUREL can be described as having three components; a fixed finite set of features Feat, a fixed type hierarchy $\langle \mathsf{Type}, \sqsubseteq \rangle$ with a finite set of types, and a constraint function C which associates a constraint feature structure with every type which determines which feature structures are *well-formed.*

2.2.1 The type hierarchy

The definition of a type hierarchy is almost identical to that used by Carpenter but as before we follow the reverse convention on ordering, so that more specific types are described as being lower in the hierarchy. The type hierarchy $\langle \mathsf{Type}, \sqsubseteq \rangle$ defines a partial order (notated $\sqsubseteq$, "is more specific than") on the types and specifies which types are *consistent.* A set of types is said to be consistent if the members of the set share a common subtype. That is the subset $S \subseteq \mathsf{TYPE}$ is consistent iff there is some t_0 in TYPE such that $t_0 \sqsubseteq t$ for any t in S. The significance of this is that only feature structures with mutually consistent types can be unified. Two types which are unordered in the hierarchy are assumed to be inconsistent unless a common subtype has been explicitly specified. It is a condition on the type hierarchy that every *consistent* set of types $S \subseteq \mathsf{TYPE}$ must have a unique greatest lower bound or meet (notation $\sqcap S$). The unique greatest lower bound condition on consistent types allows feature structures to be typed deterministically; if two feature structures of types **a** and **b** are unified the type of the result will be $\mathbf{a} \sqcap \mathbf{b}$, which must be unique if it exists. If $\mathbf{a} \sqcap \mathbf{b}$ does not exist unification fails. Thus in the fragment of a type hierarchy shown in Figure 2.2 **artifact** and **physical** are consistent; $\mathbf{artifact} \sqcap \mathbf{physical} = \mathbf{art\_phys}$. We will use a very simple type system at this point for ease of exposition.

Because the type hierarchy is a partial order it has properties of reflexivity, transitivity and anti-symmetry (from which it follows that the type hierarchy cannot contain cycles). Note that the empty set is (vacuously) consistent, as for any t_0 in TYPE it satisfies the

[1] The systems are compared in more detail in section 2.2.5. As we will see in Chapter 4, LAUREL constraints can also be used for constraint resolution in the same way as Carpenter's.

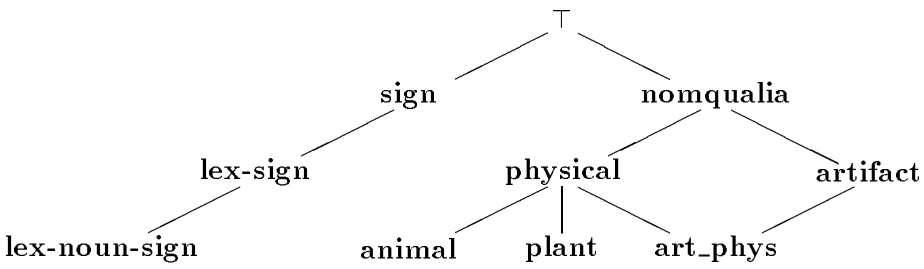


Figure 2.2: A fragment of a type hierarchy

condition that $t_0 \sqsubseteq t$ for all t's in the empty set. The maximal element $\top$ of $\langle \mathsf{TYPE}, \sqsubseteq \rangle$ is defined as the meet of the empty set, $\top = \sqcap\emptyset$. This element $\top$ is such that $t \sqsubseteq \top$ for any t in TYPE. Thus the join operation on the type hierarchy is total.

The meet $\sqcap$ operation can be made total by adding the join of the empty-set $\bot = \sqcup\emptyset$ to $\langle \mathsf{TYPE}, \sqsubseteq \rangle$. But even if $\bot$ is added to make $\langle \mathsf{TYPE}, \sqsubseteq \rangle$ a lattice, this lattice need not be distributive. For example, given the type hierarchy in Figure 2.2 $(\textbf{animal} \sqcup \textbf{plant}) \sqcap \textbf{artifact} = \textbf{art\_phys}$ but $(\textbf{animal} \sqcap \textbf{artifact}) \sqcup (\textbf{plant} \sqcap \textbf{artifact}) = \bot$. Although meet will be taken as corresponding to conjunction, join does not correspond to disjunction. (The lattice could be completed so that join was equivalent to disjunction, see section 2.3.) There is thus a symmetry with the behaviour of feature structures, where generalisation and disjunction are not equivalent. Nothing in what follows depends on whether the existence of a bottom element in the type hierarchy, $\bot$, is assumed or not. We use $\bot$ as a notational convenience to indicate inconsistency, just as $\bot$ is used to indicate unification failure for feature structures.

2.2.2 Typed feature structures

Initially we will define the set $\mathcal{F}$ of typed feature structures to include both those which are, and those which are not, well-formed with respect to a particular type system. A typed feature structure is defined as a tuple $F = \langle Q, q_0, \delta, \alpha \rangle$ where the only significant difference from the definition in the untyped case is that instead of α being a partial atomic value function it is a total node typing function, $\alpha\colon Q \rightarrow \mathsf{TYPE}$. Thus every node has a type.

Definition 6 (Typed Feature Structure) *A typed feature structure is a tuple* $F = \langle Q, q_0, \delta, \alpha \rangle$ *where:*

- Q*: a finite set of (connected, acyclic) nodes*
- $q_0 \in Q$*: the root node*
- $\delta : \mathsf{Feat} \times Q \rightarrow Q$ *: the partial feature value function*
- $\alpha : Q \rightarrow \mathsf{Type}$ *: the total node typing function*

with connectedness and acyclicity being defined as in the untyped case.

Consider the following example of a feature structure:

$$F_1 = \left[\begin{array}{ll} \textbf{artifact} & \\ \text{TELIC} = & \left[\begin{array}{l} \textbf{verb-sem} \\ \text{IND} = \boxed{0}\ \textbf{eve} \\ \text{PRED} = \textbf{drink\_1} \\ \text{ARG1} = \boxed{0} \\ \text{ARG2} = \textbf{obj} \\ \text{ARG3} = \textbf{obj} \end{array}\right] \end{array}\right]$$

As every feature structure has a unique initial node, q_0, the type of a feature structure can be said to be the type of its initial node, that is:

Definition 7 (Type of a feature structure) *If* $F = \langle Q, q_0, \delta, \alpha\rangle$ *then* $Typeof(F) = \alpha(q_0)$.

Thus the type of feature structure F_1 is **artifact**.

The definition of subsumption of typed feature structures is very similar to that for untyped feature structures, with the additional proviso that the ordering must be consistent with the ordering on their types. The symbol $\sqsubseteq$ ("is-more-specific-than", "is-subsumed-by") is thus overloaded to express subsumption of feature structures as well as the ordering on the type hierarchy. Thus if F_1 and F_2 are feature structures of types t_1 and t_2 respectively, then $F_1 \sqsubseteq F_2$ only if $t_1 \sqsubseteq t_2$. If two feature structures are identical except for their types then the subsumption ordering on the feature structures will be equivalent to the ordering on their types.

Definition 8 (Abstract Typed Feature Structure) *If* $F = \langle Q, q_0, \delta, \alpha\rangle$ *is a feature structure, we let* $\equiv_F \subseteq \mathsf{Path} \times \mathsf{Path}$ *and* $\mathcal{P}_F : \mathsf{Path} \rightarrow \mathsf{Type}$ *be such that:*

- *(Path Equivalence)*
  $\pi \equiv_F \pi'$ *if and only if* $\delta(\pi, q_0) = \delta(\pi', q_0)$
- *(Path Value)*
  $\mathcal{P}_F(\pi) = t$ *if and only if* $\alpha(\delta(\pi, q_0)) = t$.

Definition 9 (Subsumption of typed feature structures) F *subsumes* F'*, written* $F' \sqsubseteq F$*, if and only if:*

- $\pi \equiv_F \pi'$ *implies* $\pi \equiv_{F'} \pi'$
- $\mathcal{P}_F(\pi) = t$ *implies* $\mathcal{P}_{F'}(\pi) = t'$ *and* $t' \sqsubseteq t$

Unification of typed feature structures is defined in the same way as for untyped feature structures, that is the unification of two typed feature structures will be their greatest lower bound in the subsumption ordering. Since if F and F' are feature structures of types t and t' respectively, their unification $F \sqcap F'$ has to have type $t \sqcap t'$, unification will fail if $t \sqcap t'$ does not exist. Generalisation can also be defined in the same way as for untyped feature structures, that is as the lowest upper bound of two feature structures in the subsumption ordering.

Thus the type hierarchy makes a finer grained differentiation of feature structures possible, because the ordering is dependent on the type hierarchy as well as the order of untyped feature structure subsumption. For example, if F_1 and F_2 are the two feature structures given below, then $F_2 \sqsubset F_1$ assuming that **e-sentient** $\sqsubset$ **obj**:

$$F_1 = \left[\begin{array}{ll} \textbf{artifact} & \\ \text{TELIC} = & \left[\begin{array}{l} \textbf{verb-sem} \\ \text{IND} = \boxed{1}\ \textbf{eve} \\ \text{PRED} = \textbf{drink\_1} \\ \text{ARG1} = \boxed{1} \\ \text{ARG2} = \textbf{obj} \\ \text{ARG3} = \textbf{obj} \end{array}\right] \end{array}\right] \quad F_2 = \left[\begin{array}{ll} \textbf{artifact} & \\ \text{TELIC} = & \left[\begin{array}{l} \textbf{verb-sem} \\ \text{IND} = \boxed{1}\ \textbf{eve} \\ \text{PRED} = \textbf{drink\_1} \\ \text{ARG1} = \boxed{1} \\ \text{ARG2} = \textbf{e-sentient} \\ \text{ARG3} = \textbf{obj} \end{array}\right] \end{array}\right]$$

This is in itself useful; for example it allows selectional restrictions to be encoded as sorts in a way which is extremely cumbersome with untyped feature structures (cf. Moens *et al.*, 1989). However the type system also specifies a set of *well-formed* feature structures, which satisfy a constraint function, and this gives the system a functionality which includes the properties of inheritance, error-checking and classification. It is at this point that LAUREL begins to differ significantly from Carpenter's system.

2.2.3 Constraints

The constraint function C defines the set of feature structures $\mathcal{WF}$ which are well-formed with respect to the type system. In LAUREL every type must have exactly one associated feature structure which acts as a constraint on all feature structures of that type; by subsuming all well-formed feature structures of that type. The constraint also defines which features are *appropriate* for a particular type; in a well-formed feature structure each node must have all the features appropriate to its type and no others. Constraints are inherited by all subtypes of a type, but a subtype may introduce new features (which will be inherited as appropriate features by all its subtypes). A constraint on a type is a well-formed feature structure of that type; all constraints must therefore be mutually consistent.

For example, considering some of the types shown in Figure 2.2, the constraint associated with the type **artifact** might be:

$$\begin{bmatrix} \textbf{artifact} & \\ \text{TELIC} = & \begin{bmatrix} \textbf{verb-sem} \\ \text{IND} = \boxed{0}\ \textbf{eve} \\ \text{PRED} = \textbf{string} \\ \text{ARG1} = \boxed{0} \\ \text{ARG2} = \textbf{obj} \\ \text{ARG3} = \textbf{obj} \end{bmatrix} \end{bmatrix}$$

This constraint states that any feature structure of type **artifact** must have a feature structure of type **verb-sem** as the value for its TELIC feature. This constraint must be consistent with the constraints on all of its subparts, i.e. it must itself be a well-formed feature structure. In this case, for example, the constraint on **verb-sem** is:

$$\begin{bmatrix} \textbf{verb-sem} \\ \text{IND} = \boxed{0}\ \textbf{eve} \\ \text{PRED} = \textbf{string} \\ \text{ARG1} = \boxed{0} \\ \text{ARG2} = \textbf{obj} \\ \text{ARG3} = \textbf{sem} \end{bmatrix}$$

and thus the constraint on **artifact** is well-formed, given that $\textbf{obj} \sqsubseteq \textbf{sem}$.

The type **physical** might have constraint:

$$\begin{bmatrix} \textbf{physical} \\ \text{FORM} = \textbf{form} \end{bmatrix}$$

Here **form** is an atomic type, and has no appropriate features, the constraint on **form** is simply the atomic feature structure [**form**]. The constraint on **art_phys** will contain information inherited from both parents, thus:

$$\begin{bmatrix} \textbf{art\_phys} & \\ \text{FORM} = \textbf{form} & \\ \text{TELIC} = & \begin{bmatrix} \textbf{verb-sem} \\ \text{IND} = \boxed{0}\ \textbf{eve} \\ \text{PRED} = \textbf{string} \\ \text{ARG1} = \boxed{0} \\ \text{ARG2} = \textbf{obj} \\ \text{ARG3} = \textbf{obj} \end{bmatrix} \end{bmatrix}$$

Given that **mass** is an atomic subtype of **form**, and that **e-sentient** is an atomic subtype of **obj**, the feature structure below is well-formed since it contains all the appropriate features and no inappropriate ones, it is subsumed by the constraints on its type and all its substructures are well-formed.

$$\begin{bmatrix} \textbf{art\_phys} & \\ \text{FORM} = \textbf{mass} & \\ \text{TELIC} = \begin{bmatrix} \textbf{verb-sem} \\ \text{IND} = \boxed{0}\ \textbf{eve} \\ \text{PRED} = \textbf{string} \\ \text{ARG1} = \boxed{0} \\ \text{ARG2} = \textbf{e-sentient} \\ \text{ARG3} = \textbf{obj} \end{bmatrix} \end{bmatrix}$$

Formally, the notion of appropriate features is defined as follows:

Definition 10 (Appropriate features) *If $C(t) = \langle Q, q_0, \delta, \alpha \rangle$ then the appropriate features of t are defined as $Appfeat(t) = Feat(\langle F, q_0 \rangle)$ where $Feat(\langle F, q \rangle)$ is defined to be the set of features labelling transitions from the node q in some feature structure F i.e. $f \in Feat(\langle F, q \rangle)$ such that $\delta(f, q)$ is defined.*

We can then define the constraint function:

Definition 11 (Constraint function) *The constraint function which associates constraint feature structures with types is given by $C\colon \langle \mathsf{TYPE}, \sqsubseteq \rangle \rightarrow \mathcal{F}$.*
This must satisfy the following conditions:

Monotonicity *Given types t_1 and t_2 if $t_1 \sqsubseteq t_2$ then $C(t_1) \sqsubseteq C(t_2)$*

Type *For a given type t, if $C(t)$ is the feature structure $\langle Q, q_0, \delta, \alpha \rangle$ then $\alpha(q_0) = t$.*

Compatability of constraints *For all $q \in Q$ the feature structure $F' = \langle Q', q, \delta, \alpha \rangle \sqsubseteq C(\alpha(q))$ and $Feat(q) = Appfeat(\alpha(q))$.*

Maximal introduction of features *For every feature $f \in \mathsf{FEAT}$ there is a unique type $t = Maxtype(f)$ such that $f \in Appfeat(t)$ and there is no type s such that $t \sqsubset s$ and $f \in Appfeat(s)$. The maximal appropriate value of a feature $Maxappval(f)$ is the type t such that if $C(Maxtype(f)) = \langle Q, q_0, \delta, \alpha \rangle$ then $t = \alpha(\delta(f, q_0))$*

The compatability condition implies that no constraint feature structure $C(t) = F$ can strictly contain a feature structure of type t or any subtype of t. That is, if F is given by $\langle Q, q_0, \delta, \alpha \rangle$, then for all non-initial nodes $q \in Q$ such that $q \neq q_0$ the type of the node $\alpha(q) \not\sqsubseteq t$. If such a node existed it would have to be the initial node of a feature structure $F|_q$ which was more specific than F, i.e. $F|_q \sqsubseteq F$, and would therefore itself have to contain such a node, and so on. Thus such a constraint could only be satisfied by a cyclic or infinite structure, and we disallow both of these possibilities. This condition does not, however, rule out recursive structures such as lists, because the type **list** can be defined to have two subtypes **empty-list** and **nonempty-list**, where the former has no appropriate features and the latter has two, HEAD which can take any value, and TAIL, which will take a value of type **list**. The type **nonempty-list** does not violate the compatability condition, since the structure can be terminated.

Definition 12 (Well-formed feature structures) *We say that a given feature structure $F = \langle Q, q_0, \delta, \alpha \rangle$ is a well-formed feature structure iff for all $q \in Q$, we have that $F' = \langle Q', q, \delta, \alpha \rangle \sqsubseteq C(\alpha(q))$ and $Feat(q) = Appfeat(\alpha(q))$.*

From these definitions it can be seen that all constraint feature structures are themselves well-formed feature structures.[2]

Since the type system gives a concept of a well-formed feature structure, it follows that non-well-formed feature structures can be detected, allowing error checking. Typing also allows for a form of classification; a feature may only be introduced as appropriate at one point in the type hierarchy (and will be inherited as an appropriate feature by all subtypes of that type); it follows from this that there is a unique maximal type for any set of features, and therefore an untyped feature structure can always be typed deterministically. For example, assuming the type system introduced above, the attribute value specification:

< FORM > = **mass**
< TELIC : ARG2 > = **e-sentient**

would be expanded out into the following feature structure:

$$\begin{bmatrix} \textbf{art\_phys} & \\ \text{FORM} = \textbf{mass} & \\ \text{TELIC} = & \begin{bmatrix} \textbf{verb-sem} \\ \text{IND} = \boxed{0}\ \textbf{eve} \\ \text{PRED} = \textbf{string} \\ \text{ARG1} = \boxed{0} \\ \text{ARG2} = \textbf{e-sentient} \\ \text{ARG3} = \textbf{obj} \end{bmatrix} \end{bmatrix}$$

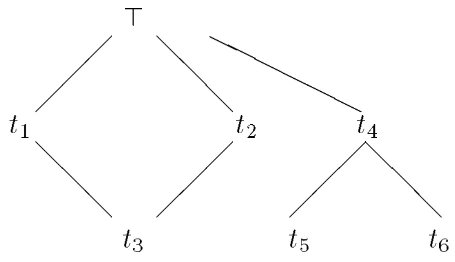
(Full details of the feature structure description language are given in section 2.4.1, below.) The type of the feature structure is determined automatically; since the features FORM and TELIC are specified, its type has to be **art_phys** (or some subtype of that type). The procedure for making a feature structure well-formed, WF, involves recursively unifying the constraints of the maximal types of the subparts of the feature structure; since we disallow cyclic feature structures, this operation terminates straightforwardly at the atomic feature structures.

2.2.4 Unification and well-formedness

The definition of subsumption for well-formed feature structures is exactly the same as that for typed feature structures. The result of well-formed unification of well-formed feature structures, $\langle \mathcal{WF}, \sqsubseteq \rangle$, is defined to be the greatest lower bound in the subsumption ordering of well-formed feature structures and is thus itself well-formed. This does however mean that the algorithm for unification involves an additional step, because it is potentially necessary to unify in the constraint feature structure associated with the meet of the types of the feature structures being unified.

If F_1 and F_2 are well-formed feature structures of types t_1 and t_2 respectively, then $F_1 \sqcap F_2$, if it exists, has type $t_1 \sqcap t_2$. Since F_1 and F_2 are well-formed, in particular we know that $F_1 \sqsubseteq C(t_1)$ and $F_2 \sqsubseteq C(t_2)$. Thus if F_1 and F_2 are consistent, $F_1 \sqcap F_2 \sqsubseteq C(t_1) \sqcap C(t_2)$. But to be well-formed $F_1 \sqcap F_2$ has to satisfy $F_1 \sqcap F_2 \sqsubseteq C(t_1 \sqcap t_2)$ and $C(t_1 \sqcap t_2)$ might be more specific than $C(t_1) \sqcap C(t_2)$. Consider the following example of a type hierarchy:

[2] As we will discuss in Chapter 4, stronger notions of well-formedness are possible and desirable in some circumstances.



Assume that the types t_4, t_5 and t_6 are atomic (i.e. they have constraints $[t_4]$, $[t_5]$ and $[t_6]$, respectively) and the constraints on types t_1, t_2 and t_3 are:

$$C(t_1) = \begin{bmatrix} t_1 \\ \mathrm{F} = t_4 \end{bmatrix} \quad C(t_2) = \begin{bmatrix} t_2 \\ \mathrm{G} = \top \end{bmatrix} \quad C(t_3) = \begin{bmatrix} t_3 \\ \mathrm{F} = t_5 \\ \mathrm{G} = \top \\ \mathrm{H} = \top \end{bmatrix}$$

We then have: $C(t_1) \sqcap C(t_2) = \begin{bmatrix} t_3 \\ \mathrm{F} = t_4 \\ \mathrm{G} = \top \end{bmatrix}$

Then $t_3 = t_1 \sqcap t_2$ but $C(t_3) \sqsubset C(t_1) \sqcap C(t_2)$

Consider the following well-formed feature structures, F_1, F_2 and F_3:

$$F_1 = \begin{bmatrix} t_1 \\ \mathrm{F} = t_4 \end{bmatrix} \qquad F_2 = \begin{bmatrix} t_2 \\ \mathrm{G} = \top \end{bmatrix} \quad F_3 = \begin{bmatrix} t_1 \\ \mathrm{F} = t_6 \end{bmatrix}$$

$$F_1 \sqcap F_2 = \begin{bmatrix} t_3 \\ \mathrm{F} = t_4 \\ \mathrm{G} = \top \end{bmatrix}$$

$$F_2 \sqcap F_3 = \begin{bmatrix} t_3 \\ \mathrm{F} = t_6 \\ \mathrm{G} = \top \end{bmatrix}$$

$F_1 \sqcap F_2$ is not a well-formed feature structure of type t_3 as $F_1 \sqcap F_2 \not\sqsubseteq C(t_3)$. To extend it to a well-formed feature structure involves unifying in $C(t_3)$. $F_2 \sqcap F_3$ is also not a well-formed feature structure but it cannot be extended to a well-formed feature structure, because its value for F is inconsistent with the constraint for t_3.

The same situation can arise with Carpenter's appropriateness specifications (see below) as discussed in Carpenter (1992:104f). We could impose a condition of meet preservation on the constraints, analagous to Carpenter's join preservation on appropriateness conditions, but this is over-restrictive. Instead we use the following definition for the operation of well-formed unification, $\sqcap_w$:

Definition 13 (Well-formed unification) *The well-formed unification of F_1 and F_2, $F_1 \sqcap_w F_2$ is given by $F_1 \sqcap F_2 \sqcap C(t_1 \sqcap t_2)$. This will be well-formed if it exists.*

No such complication arises with generalisation, since $F = F' \sqcup F''$ will always be well-formed if F' and F'' are well-formed. From now on, when talking about typed feature structures, we will use the term feature structure to mean well-formed typed feature structure and $\sqcap$ will be used to refer to well-formed unification, $\sqcap_w$, dropping the subscript.

2.2.5 A comparison with Carpenter's system

Rather than specifying typing by associating a feature structure with every type, Carpenter uses appropriateness specifications, which are defined as follows: [3]

[3] This definition is taken from Carpenter (1992:86) but we use the reverse directionality on the type hierarchy for consistency with our definitions.

Definition 14 (Appropriateness Specification) *An appropriateness specification over the inheritance hierarchy* $\langle \mathsf{TYPE}, \sqsubseteq \rangle$ *and features* $Feat$ *is a partial function* $Approp$: $Feat \times Type \rightarrow Type$ *that meets the following conditions:*

Feature Introduction *For every feature* $f \in \mathsf{FEAT}$ *there is a most general type* $Intro(f) \in Type$ *such that* $Approp(f, Intro(f))$ *is defined.*

Upward Closure / Right Monotonicity *If* $Approp(f,t)$ *is defined and* $t' \sqsubseteq t$ *then* $Approp(f,t')$ *is also defined and* $Approp(f,t') \sqsubseteq Approp(f,t)$

As mentioned above, our constraints can be seen as a generalisation of Carpenter's appropriateness specification in that the two would be equivalent if we added the condition to our constraint function that all constraints be one-level, non-reentrant feature structures (i.e. for all features, f, in $Feat(F, q_0)$, the feature structures $\delta(f, q_0)$ are distinct and atomic). Our monotonicity condition is derived from Carpenter's upward closure (downward closure in our terms) and similarly our maximal introduction of features is derived from Carpenter's feature introduction condition. Our other two conditions are necessary because constraints have themselves to be well-formed feature structures in order to have a consistent system.

Carpenter defines two notions of typing; *well-typedness*, where feature structures may only contain appropriate features with appropriate values, and *total well-typedness* where feature structures must meet the well-typedness conditions and must also contain all the appropriate features. Our notion of well-formedness is thus comparable to Carpenter's *total well-typedness*. As Carpenter shows finite total typing, that is the extension of a well-typed feature structure to a most general finite totally typed feature structure, is only possible under conditions where *Approp* contains no loops; as shown above, an equivalent condition arises from our consistency condition.

2.3 Extensions to the language

A type system has essentially to be fully defined before lexical entries can be built. This causes obvious problems with respect to atomic types representing orthography and predicate names for example, where it is unrealistic to assume that the complete set can be known in advance. To get round this any string is allowed as a valid type; all strings are assumed to be subtypes of the predefined atomic type **string**, but to be unordered with respect to one another. Particular features such as ORTH, which are specified as having value **string**, thus in effect take arbitrary string values.[4]

Although many feature structure based languages allow disjunctive feature structures, this is avoided in LAUREL. Arbitrary disjunction can result in a computationally intractable system and it is not clear whether it is in fact necessary, given that the type system can be set up in a way which allows degrees of underspecification. However the lowest common supertype (join) of the types might be more general than the disjunction, in which case a new type might have to be created to get the required level of specification. For example if the type **person** was defined to have the subtypes **first**, **second**, **third**, a new type could be inserted in the hierarchy in order to express the equivalent of the disjunction **second** or **third** (see Figure 2.3).

[4] It would be necessary to extend LAUREL for morphophonemic representation; there are several possible techniques, but the most interesting approach might be based on Bird's (1992) suggestions for integration of non-linear phonology and HPSG.

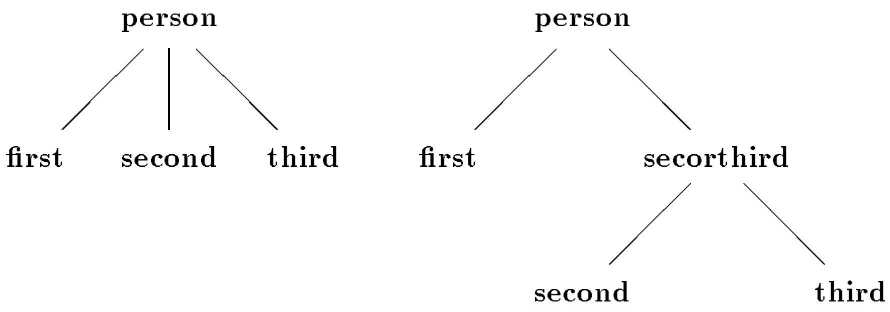


Figure 2.3: Adding types to the hierarchy

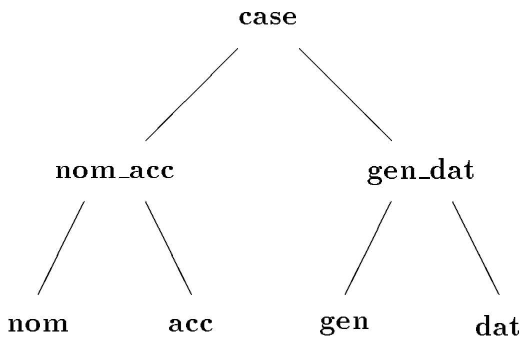


Figure 2.4: Restricting the possibilities for case specification

The effect of the non-equivalence of disjunction and join may in fact be exploited to allow a more precise specification of a language. For example, Pollard and Sag (in press) give a description of the inflection of German adjectives such as *klein*, in which the following values for case are given as possibilities: **nom**, **acc**, **gen**, **dat**, **nom** $\vee$ **acc**, **gen** $\vee$ **dat**, *unspecified*. Encoding this in the type system, as shown in Figure 2.4, rather than making use of disjunction, would directly express the restriction that only these values were available and that the following were not: **nom** $\vee$ **gen**, **acc** $\vee$ **dat**, **nom** $\vee$ **acc** $\vee$ **gen**, **nom** $\vee$ **acc** $\vee$ **dat**, **nom** $\vee$ **gen** $\vee$ **dat**, **acc** $\vee$ **gen** $\vee$ **dat**.

However, in LAUREL, a limited form of disjunction is supported; particular atomic types may be specified as allowing disjunction with an effect which is formally similar to creating all possible additional join types; this is expressed as a list of values, for example, (**second third**). For atomic types, completing some fragment of the type hierarchy under join gives the equivalent of disjunction (cf. Carpenter 1992;17f). The approach of completing the type hierarchy could be extended to complex feature structures but would, however, give a finer grain of generalisation rather than disjunction. In order to have a well-defined notion of constraint under this operation the following extra condition on constraints could be added:

Generalisation of constraints The constraint on a type t should be equal to the generalisation of the set of constraints of all its subtypes.
Let $t_1, t_2 \dots t_n$ be the set of all types which are subtypes of t. Then $C(t) = \sqcap C(t_1), C(t_2) \dots C(t_n)$.

From the usual monotonicity condition it follows that $\sqcap C(t_1), C(t_2) \dots C(t_n) \sqsubseteq C(t)$ but it is desirable to avoid the situation where $\sqcap C(t_1), C(t_2) \dots C(t_n) \sqsubset C(t)$, given that the type hierarchy is complete, because it would correspond to the situation where information which should be true of all feature structures of type t was being specified redundantly, at multiple points in the type hierarchy.

This condition would probably be over-restrictive during development of a type system. However, if we did introduce it, we could automatically complete the type lattice under the operation $\sqcap$ and incorporate new types with complex constraints which were the generalisation of the constraints on their subtypes. But this only removes the effect that $t_1 \vee t_2$ might be more specific than $t_1 \sqcap t_2$; the non-equivalence of disjunction and generalisation still holds, as it does for untyped feature structures. This has not been implemented, because it does not seem likely that a compromise between generalisation and disjunction would give particularly perspicuous representations.

The motivation for avoiding disjunction is not just computational efficiency, but also that the use of disjunction in a description can be seen as an indication that there is something rather arbitrary about the representation. We should perhaps distinguish between two uses of disjunction. The first could be characterised as disjunction for structure sharing, where one disjunctive feature structure is specified rather than two individual structures because a considerable amount of information is equivalent in both. For example, rather than specifying different lexical entries for *worry* in the senses most obviously involved in the following sentences:

(1) The greenhouse effect worries Mary.
That dog worries sheep.

there might be a temptation, given an approach to the lexicon which minimised semantic information, to use a single lexical entry with a disjunctive specification of the predicate $\mathbf{worry1'} \vee \mathbf{worry2'}$. This would be an inappropriate use of disjunction at the descriptive level, given a theory of the lexicon where different senses were regarded as distinct linguistic entities, because implementational considerations would be determining the representation. Practical problems would arise in trying to extend the lexicon to contain semantic information, for example. In LAUREL a succinct description of feature structures which have structure in common is available through the inheritance mechanisms, which would potentially allow an implementation to use structure sharing in a transparent way. In this particular case, both entries would be specified as having the type appropriate for transitive verbs, which would allow a parser to efficiently determine the possibility of packing the lexical entries (or to avoid expansion of the lexical entries until required).

Alternatively disjunction can be used to capture the behaviour of some single linguistic entity. Whether general disjunction is really required depends on the predictability of the alternation; if there is little regularity and the disjuncts are not predictable then general disjunction is needed because of the essentially static nature of the type system. However, if only certain sets of values are possible in principle, then the type system can be used to capture this, as illustrated above. Furthermore, if we are dealing with information in two distinct feature structures, and attempting to form a coherent description from that information, generalisation may sometimes be more appropriate than disjunction, because generalisation of feature structures will give results which reflect in specificity the compatability of the feature structures. Thus it seems plausible that disjunction might be avoided in the representation and potentially worthwhile to try and do so. We will thus only make use of limited atomic disjunction in the representations which follow.

2.4 Description language

We will split the discussion of the LAUREL description language into two sections. The first covers the *local* language which is quite similar to the PATR-II path descriptions. The formalisation of such a language in the typed case is discussed by Carpenter (1992). The second section describes the *non-local* description language, which allows the definition of feature structures in terms of other feature structures. This is more distinctive to LAUREL, although it has parallels in many other unification-based languages, such as the template mechanism in PATR-II. The default mechanism operates purely at the level of non-local descriptions — we will introduce default inheritance in this chapter, but the detailed discussion and definition is to be found in Chapter 3.

2.4.1 Local descriptions

The syntax of the description language which is used to define feature structures in LAUREL is based on the path notation used in PATR, with the modification that : is used to separate features. However, in a notation for a typed system, it must be possible to specify the types of non-terminal nodes in the feature structure; in LAUREL this may either be done by making the type the value of the relevant path or, for compactness, by specifying the type in the path (as an optional value, before the feature).[5] For example:

< **lex-count-noun** `QUALIA` : **physical** `TELIC : ARG2` > = **e-sentient**

is equivalent to:

<> = **lex-count-noun**
< `QUALIA` > = **physical**
< `QUALIA : TELIC : ARG2` > = **e-sentient**

Instead of specifying the empty path in a path value specification, the type name alone can be used:

lex-count-noun
< `QUALIA` > = **physical**
< `QUALIA : TELIC : ARG2` > = **e-sentient**

Type specifications consist of the name of the type followed by a bracketed list indicating the immediate parents. This is followed, if appropriate, by a local description of a feature structure which forms the constraint on the type when unified with the constraints on the parent types. Figure 2.5 shows a description of a tiny type system which we will use in subsequent examples in this chapter and the next.

The following BNF description gives the syntax of the type specification language:

```
Type specification ->
         typename Parents [Comment][Constraint].
Comment -> " string "
Parents -> ( typename* )
Constraint -> Path_spec_list | Enumeration
```

[5]To make it easier to read the descriptions we will keep the same typographic conventions as in the rest of the text, that is types in bold font, features in capitals and feature structure identifiers in lower case typewriter font. Case and font distinctions are not, however, significant in the language.

```
top ().

boolean (top)
    (OR true false).

string (top).

sign (top)
    < ORTH > = string.

lex-sign (sign).

lex-noun-sign (lex-sign)
    < COUNT > = boolean
    < QUALIA > = nomqualia.

rule (top)
    < 0 > = sign
    < 1 > = sign.

grammar-rule (rule)
    < 2 > = sign.

lexical-rule (rule)
    < 0 > = lex-sign
    < 1 > = lex-sign
    < 0 : ORTH > = < 1 : ORTH >.

nomqualia (top).

physical (nomqualia)
    < FORM > = form.

form (top)
    (OR mass individual plural).

animal (physical)
     < SEX > = gender.

gender (top)
    (OR male female).

plant (physical).

artifact (nomqualia)
    < TELIC > = verb-sem.

art_phys (physical artifact).

verb-sem (top)
    < IND > = eve
    < PRED > = string
    < ARG1 > = < IND >
    < ARG2 > = obj
    < ARG3 > = sem.

sem (top).

eve (sem).

obj (sem).

e-sentient (obj).
```

Figure 2.5: Illustrative type system

```
Path_spec_list -> Path_spec*
Path_spec -> EPath = Typevalue | Path = Path
Typevalue -> typename | ( typename+ )
Epath -> Path | <>
Path -> < Type_F_pair_list >
Type_F_pair_list -> Type_Feature_pair
         | Type_Feature_pair :  Type_F_pair_list
Type_Feature_pair -> [typename] feature
Enumeration -> ( OR typename+ )
```

Disjunction is indicated by bracketed lists of atomic types as values for a path. Enumeration is syntactic sugar for enumerated atomic types. For example:

> **boolean (top)** (OR **true false**).

expands out into the equivalent of:

> **boolean (top).**
> **true (boolean).**
> **false (boolean).**

The descriptions can be taken as formulas which must be satisfied by the feature structures they describe, as was done in Pereira and Shieber (1984). The logic of the description language used here is identical to that described by Carpenter (1992:52f) with the exception that we do not allow disjunctive descriptions (and thus the algorithm for deciding the satisfiability of a formula has the same order of complexity as unification). The following definition is a slightly modified version of the one given by Carpenter:

Definition 15 (Descriptions) *The set of* descriptions *over the poset* $\langle \mathsf{TYPE}, \sqsubseteq \rangle$ *of types and the collection* FEAT *of features is the least set* DESC *such that:*

- $t \in \mathsf{DESC}$ *if* $t \in \mathsf{TYPE}$
- $\pi : \phi \in \mathsf{DESC}$ *if* $\pi \in \mathsf{FEAT}^*$ *and* $\phi \in \mathsf{DESC}$
- $\pi_1 \doteq \pi_2 \in \mathsf{DESC}$ *if* $\pi_1, \pi_2 \in \mathsf{FEAT}^*$
- $\phi \wedge \psi \in \mathsf{DESC}$ *if* ϕ *and* $\psi \in \mathsf{DESC}$

A description of the form $\pi : \phi$ applies to objects whose value for the path π satisfies ϕ. A description of the form $\pi \doteq \pi'$ means that the value at the end of path π is token identical to that at the end of path π'.

The LAUREL path notation for local descriptions can be mapped directly into the syntax used by Carpenter.

LAUREL	Carpenter
t	t
`<> =` **t**	t
`< f1 : f2 ... fn > =` **t**	$\pi : t$
`< f1 : f2 ... fn > = < g1 : g2 ... gn >`	$\pi \doteq \pi'$

where π is the path f1, f2 ...fn and π' is the path g1, g2 ...gn. A list of LAUREL path specifications in a description are to be taken as conjoined.

Thus the results given by Carpenter concerning the satisfiability of a description also apply to LAUREL:

Definition 16 (Satisfaction) *The* satisfaction relation *relates the collection of feature structures $\mathcal{F}$ and the set of descriptions* DESC*. It is the least relation '$\models$' such that, if F is the feature structure $\langle Q, q_0, \delta, \alpha\rangle$ and $\phi \in \mathsf{DESC}$*

- $F \models t$ *if* $t \in \mathsf{TYPE}$ *and* $\alpha(q_0) \sqsubseteq t$
- $F \models \pi{:}\,\phi$ *if* $F@\pi \models \phi$
- $F \models \pi_1 \doteq \pi_2$ *if* $\delta(q_0, \pi_1) = \delta(q_0, \pi_2)$
- $F \models \phi \wedge \psi$ *if* $F \models \phi$ *and* $F \models \psi$

where if $F = \langle Q, q_0, \delta, \alpha\rangle$, $F@\pi$ *is* $\delta(\pi, q_0)$.

Definition 17 *Consider the set of all feature structures that satisfy a certain description ϕ, that is $Sat(\phi) = \{F \in \mathcal{F} | F \models \phi\}$. If ϕ is a formula in* DESC*, ϕ is said to be* satisfiable *if there exists a feature structure F that satisfies it, that is the set $Sat(\phi)$ is not empty.*

As Carpenter shows we have the usual results for the satisfaction relation, $\models$:

Monotonicity If ϕ is a formula then if $F_1 \models \phi$ and $F_2 \sqsubseteq F_1$ then $F_2 \models \phi$.

Most general satisfier For every satisfiable formula ϕ, there is a unique most general feature structure $MGSat(\phi)$ that satisfies it.

Description For any feature structure in $\mathcal{F}$ there is a description $\mathsf{Desc}(F)$ such that

$$F \sim MGSat(\mathsf{Desc}(F))$$

2.4.2 Non-local descriptions

The flexibility of LAUREL is enhanced by allowing feature structures to be described in terms of other feature structures. Particular feature structures may have identifiers associated with them; feature structures representing complete lexical entries are identified by a combination of orthography plus sense information, while lexical and grammar rules also have associated names, and in general any feature structure may be defined with an associated identifier. The term *psort* will be used to refer to any such named structure; the significance of this is the use of psorts in the description of other feature structures.[6] The function FS returns the feature structure associated with a psort identifier. Although formally this scheme is very similar to the use of templates, at least when the non-default operations are considered, its particular significance is that it allows relationships between word senses (and rules) to be encoded.

Feature structure descriptions are thus not just local, and relationships between feature structures may be set up in a variety of ways. In general terms a path can be specified as having a value which has a particular relationship with a feature structure, which is either some specified part of a psort, or of some combination of psorts. Ordinary path/value

[6] The term psort is arbitrary here and should not be taken as having any connection with any other use of the term. It originally stood for "pseudo-sort", but it is probably not very helpful to think of psorts as being like sorts in any strict sense.

equations can be seen as a special case of this, where the feature structure involved is the constraint on the value type. LAUREL supports the following relationships: [7]

`=` Local description (as above)
`=*` Equality (with a type constraint)
`==` Equality (with a psort)
`<=` Non default inheritance from a psort (subsumption)
`<` Default inheritance from a psort

Operators available to combine feature structures are:

`/\` Unification
`\/` Generalisation
`+` Lexical rule application (see section 2.4.4, below)

The complete syntactic description for lexical entries is as follows:

```
Lexentry -> LexID PPath_spec+ .
LexID -> orth sense-id
PPath_spec -> EPath < FS | EPath <= FS | EPath == FS
          | EPath =* Typevalue | Path_spec | typename
FS -> psortname Epath | CFS Epath
CFS -> ( FS \/ FS ) | ( FS /\ FS )
          | ( CFS + rulename ) | ( psortname + rulename )
Path_spec -> EPath = Typevalue | Path = Path
Typevalue -> typename | ( typename+ )
Epath -> Path | <>
Path -> < Type_F_pair_list >
Type_F_pair_list -> Type_Feature_pair
          | Type_Feature_pair :  Type_F_pair_list
Type_Feature_pair -> [typename] feature
```

`orth`, `sense-id`, `typename`, `psortname`, `rulename` and `feature` are terminal symbols.

The description language for rules and general psorts differs from that of lexical entries only in that the identifier is not split.

The system allows the global specification of path values in terms of the identifier component which apply to all lexical entries. This avoids the need to specify the values for orthography, predicate identity and so on individually for each lexical entry. In the following discussion we will use a mutually consistent set of 'toy' lexical entries to illustrate the language. These make use of the types shown in the left column of Figure 2.5 and of the global path specification:

```
< ORTH > = [orth]
```

where `[orth]` will be instantiated by a string type corresponding to the orthography for the particular lexical entry.

The following lexical entry uses no non-local descriptions, so the expansion involved in its evaluation to produce the psort `animal_1` is due to the type system.

[7] In the first version of the language only default inheritance from psorts was supported. Non-default inheritance and equality constraints were added later, in response to particular representation issues.

animal 1
```
< COUNT > = true
< QUALIA > = animal
< QUALIA : FORM > = individual.
```

$$\mathtt{animal\_1} = \begin{bmatrix} \textbf{lex-noun-sign} \\ \text{ORTH} = \textbf{"animal"} \\ \text{COUNT} = \textbf{true} \\ \text{QUALIA} = \begin{bmatrix} \textbf{animal} \\ \text{FORM} = \textbf{individual} \\ \text{SEX} = \textbf{gender} \end{bmatrix} \end{bmatrix}$$

From now on we will drop the use of double quotes to mark string types in the feature structure diagrams, since it will be clear from the context whether a type is a string or not.

The simplest of the non-local relationships conceptually is non-default inheritance, (notated `<=`), which allows a feature structure to be described as inheriting information from a psort. For example, we could define the feature structure corresponding to the lexical entry for *sheep* as inheriting its qualia structure from `animal_1`.

sheep 1
```
< COUNT > = true
< QUALIA > <= animal_1 < QUALIA >.
```

$$\mathtt{sheep\_1} = \begin{bmatrix} \textbf{lex-noun-sign} \\ \text{ORTH} = \textbf{sheep} \\ \text{COUNT} = \textbf{true} \\ \text{QUALIA} = \begin{bmatrix} \textbf{animal} \\ \text{FORM} = \textbf{individual} \\ \text{SEX} = \textbf{gender} \end{bmatrix} \end{bmatrix}$$

Values may be further instantiated, thus the following entry for `ewe_1` inherits its qualia structure from `sheep_1`, but specifies the SEX to be **female**:

ewe 1
```
< QUALIA : SEX > = female
< QUALIA > <= sheep_1 < QUALIA >.
```

$$\mathtt{ewe\_1} = \begin{bmatrix} \textbf{lex-noun-sign} \\ \text{ORTH} = \textbf{ewe} \\ \text{COUNT} = \textbf{true} \\ \text{QUALIA} = \begin{bmatrix} \textbf{animal} \\ \text{FORM} = \textbf{individual} \\ \text{SEX} = \textbf{female} \end{bmatrix} \end{bmatrix}$$

Non-default inheritance is simply implemented by unification of the feature structure with a copy of the relevant part of the psort.

The default inheritance relationship (notated as `<`) allows values to be overridden, thus we could (albeit somewhat perversely) specify `ram_1` as inheriting information by default from `ewe_1`, but override the value for SEX:

ram 1
```
< QUALIA : SEX > = male
< QUALIA > < ewe_1 < QUALIA >.
```

$$\texttt{ram\_1} = \begin{bmatrix} \textbf{lex-noun-sign} \\ \text{ORTH} = \textbf{ram} \\ \text{COUNT} = \textbf{true} \\ \text{QUALIA} = \begin{bmatrix} \textbf{animal} \\ \text{FORM} = \textbf{individual} \\ \text{SEX} = \textbf{male} \end{bmatrix} \end{bmatrix}$$

Non-default inheritance can be seen as producing a feature structure which is the minimal satisfier of the union of the set of local descriptions given in the attribute value language with the set of descriptions of which the psort is the minimal satisfier. Default inheritance, in contrast, produces a feature structure which is the minimal satisfier of the union of the set of local descriptions with non-conflicting descriptions from the psort. Deciding how to define the non-conflicting descriptions is a complex problem, which we leave to the next chapter.

We also allow a feature structure to be specified as being identical (modulo alphabetic variance) to a psort (notated `==`) or to a constraint on a type (notated `=*`).[8] We will discuss these together, since they behave very similarly, although the `=*` descriptions are local. The main significance of these *equality constraints* is that they allow default inheritance of information to be blocked by stating that the value for some feature is known to be underspecified (the rationale behind this is discussed in the next chapter in Section 3.4). For example, both the descriptions given below expand out into the feature structure shown, with an underspecified value of `< QUALIA : SEX >`, despite inheriting information from a psort where this value is specified:

lamb 1
```
< QUALIA : SEX > =* gender
< QUALIA > < ewe_1 < QUALIA >.
```

lamb 2
```
< QUALIA : SEX > == sheep_1 < QUALIA : SEX >.
< QUALIA > < ewe_1 < QUALIA >.
```

$$\texttt{lamb\_1}, \texttt{lamb\_2} = \begin{bmatrix} \textbf{lex-noun-sign} \\ \text{ORTH} = \textbf{lamb} \\ \text{COUNT} = \textbf{boolean} \\ \text{QUALIA} = \begin{bmatrix} \textbf{animal} \\ \text{FORM} = \textbf{individual} \\ \text{SEX} = \textbf{gender} \end{bmatrix} \end{bmatrix}$$

The non-default inheritance relationship can be seen as a constraint that the daughter feature structure is subsumed by the psort feature structure; the equality relationship corresponds to a mutual subsumption constraint. We can formalise the notion of satisfaction of an equality description as follows:

Definition 18 (Feature structure equality constraint) *A feature structure $F = \langle Q, q_0, \delta, \alpha \rangle$ satisfies a feature structure equality constraint $\pi == F'$ iff $F@\pi \sim F'$*

Definition 19 (Type equality constraint) *A feature structure $F = \langle Q, q_0, \delta, \alpha \rangle$ satisfies a type equality constraint $\pi =* t$ iff $F@\pi \sim C(t)$*

Psort feature structures may also be combined by unification and generalisation. Suppose that the lexical entry for `plant_1` is the following:

[8] Earlier versions of the language used `==` for non-default inheritance.

```
plant 1
    < COUNT > = true
    < QUALIA > = plant
    < QUALIA : FORM > = individual.
```

$$\texttt{plant\_1} = \begin{bmatrix} \textbf{lex-noun-sign} & \\ \text{ORTH} = \textbf{plant} & \\ \text{COUNT} = \textbf{true} & \\ \text{QUALIA} = \begin{bmatrix} \textbf{plant} \\ \text{FORM} = \textbf{individual} \end{bmatrix} & \end{bmatrix}$$

We can then build a lexical entry for `albino_1` which inherits information from the generalisation of `animal_1` and `plant_1`:

```
albino 1
    < > <= ( animal_1 < >  \/  plant_1 < > ) < >.
```

$$\texttt{albino\_1} = \begin{bmatrix} \textbf{lex-noun-sign} & \\ \text{ORTH} = \textbf{albino} & \\ \text{COUNT} = \textbf{true} & \\ \text{QUALIA} = \begin{bmatrix} \textbf{physical} \\ \text{FORM} = \textbf{individual} \end{bmatrix} & \end{bmatrix}$$

Finally, psorts may be transformed by lexical rule application. Lexical rules are feature structures of type **lexical-rule**, which has the constraint:

$$\begin{bmatrix} \textbf{lexical-rule} \\ 0 = \textbf{lex\_sign} \\ 1 = \textbf{lex\_sign} \end{bmatrix}$$

Application of a particular lexical rule simply involves unification of the input of the psort with the input part of the lexical rule, indicated by the path `<1>`, and returns the instantiated output of the rule, given by the path `<0>`. Lexical rules are discussed in more detail in Section 2.4.4, below. The rule `plural` is as follows:

```
plural
    lexical-rule
    < 0 > = lex-noun-sign
    < 1 > = lex-noun-sign
    < 0 : COUNT > = true
    < 1 : COUNT > = true
    < 0 : QUALIA : FORM > = plural
    < 1 : QUALIA : FORM > = individual.
```

$$\texttt{plural} = \begin{bmatrix} \textbf{lexical-rule} & \\ 0 = \begin{bmatrix} \textbf{lex-noun-sign} \\ \text{ORTH} = \boxed{0}\ \textbf{string} \\ \text{COUNT} = \textbf{true} \\ \text{QUALIA} = \begin{bmatrix} \textbf{physical} \\ \text{FORM} = \textbf{plural} \end{bmatrix} \end{bmatrix} & \\ 1 = \begin{bmatrix} \textbf{lex-noun-sign} \\ \text{ORTH} = \boxed{0} \\ \text{COUNT} = \textbf{true} \\ \text{QUALIA} = \begin{bmatrix} \textbf{physical} \\ \text{FORM} = \textbf{individual} \end{bmatrix} \end{bmatrix} & \end{bmatrix}$$

Applying the rule to the psort `sheep_1` we get the following feature structure as output:

$$\begin{bmatrix} \textbf{lex-noun-sign} \\ \text{ORTH} = \textbf{sheep} \\ \text{COUNT} = \textbf{true} \\ \text{QUALIA} = \begin{bmatrix} \textbf{physical} \\ \text{FORM} = \textbf{plural} \end{bmatrix} \end{bmatrix}$$

Thus the lexical entry below is equivalent to specifying that the qualia structure is inherited from this feature structure:

flock 1
```
<  > < ( sheep_1 + plural ) < >.
```

$$\texttt{flock\_1} = \begin{bmatrix} \textbf{lex-noun-sign} \\ \text{ORTH} = \textbf{flock} \\ \text{COUNT} = \textbf{true} \\ \text{QUALIA} = \begin{bmatrix} \textbf{physical} \\ \text{FORM} = \textbf{plural} \end{bmatrix} \end{bmatrix}$$

2.4.3 Evaluation of descriptions

The process of evaluation can be described as having two stages:

1. Construction of three components from the descriptions: a non-default feature structure, a default feature structure and a series of equality constraints.

2. Combination of these components to form the final entry.

A number of constraints on non-local descriptions are needed, in order that evaluation can be carried out. The first is that there are no cycles — that is that the induced hierarchy of psorts is a partial order. Because of the non-cyclicity condition, expansion of psorts will always terminate, at psorts which are defined using local descriptions alone. Thus the right hand side of the descriptions can be evaluated by expanding the feature structures for the psorts, and combining the specified structures by unification and generalisation.

So we have the following possibilities for local and non-local descriptions where F is the expanded right hand side feature structure.

- t
- $\pi = t$ (i.e. path value specification)
- $\pi_1 = \pi_2$ (i.e. path equivalence specification)
- $\pi =* t$
- $\pi <= F$
- $\pi == F$
- $\pi < F$

Since the first three cases were covered in the previous section, we will define the evaluation in terms of a feature structure F which corresponds to the most general satisfier of the set of these local descriptions, and a set of non-local descriptions, Φ.

Definition 20 (Non default feature structure evaluation) *The non-default feature structure is* $AddN(F, \Phi)$ *where:*
$AddN(F, \pi <= F') = F[\pi \sqcap F']$
$AddN(F, \pi == F') = F[\pi \sqcap F']$
$AddN(F, \pi =* t) = F[\pi \sqcap C(t)]$
$AddN(F, \pi < F') = F[\pi \sqcap C(TypeOf(F'))]$
where the notation $F[\pi \sqcap F']$ *is used to indicate the result of unifying* F*'s* π *value with* F'.

Definition 21 (Default feature structure evaluation) *The default feature structure given by a description* Φ *is* $AddD(\top, \Phi)$ *where:*
$AddD(F, \pi < F') = F[\pi \sqcap F']$
$= F$ *for all other types of description.*

The incorporation of information into the non-default feature structure from the default description arises from the constraint that defaults must be consistent with the type hierarchy.

Definition 22 (Consistency of defaults with type hierarchy) *If the description of a psort, p, contains a default description,* $\pi < F$*, then* $TypeOf(FS(p)@\pi) \sqsubseteq TypeOf(F)$

The default descriptions are further constrained to be mutually consistent, which is why they can be unified together to form a single feature structure:

Definition 23 (Mutual consistency of defaults) *If the description of a psort, p, contains the default descriptions,* $\pi < F$ *and* $\pi' < F'$ *then these are said to be mutually consistent if* $\top[\pi \sqcap F] \sqcap \top[\pi' \sqcap F'] \neq \bot$

Thus we define the final feature structure in terms of a well-formed non-defeasible feature structure, a series of equality constraints and a single defeasible feature structure.

Definition 24 (Evaluation of descriptions) *The feature structure* F *associated with the identifier* p *where there is a description* Φ *is given by* $FS(p) = \langle WF(F'), C\rangle \overset{<}{\sqcap}_e F''$, *where* F' *is the non-defeasible feature structure given by* $AddN(\top, \Phi)$, C *is the set of equality constraints and* F'' *is the default feature structure, given by* $AddD(\top, \Phi)$.

The definition of the operator $\overset{<}{\sqcap}_e$ (default unification in the presence of equality constraints) is given in the next chapter. Intuitively it corresponds to incorporating the maximal information from the default feature structure which is consistent with the constraints imposed both by the non-defeasible information and the equality constraints. The final feature structure must be consistent with the equality constraints.

Various applications of these descriptions will be found in subsequent chapters. Although introducing psorts as well as types may seem unnecessarily complex, given that the type mechanism allows inheritance, there are compelling reasons for doing so. The first point is that a notion of defaults is essential to allow lexical generalisations to be captured, but typing and defaults are incompatible notions. Types are used to enforce an organisation on the lexicon; if the typing scheme can be overridden then none of the advantages of a typed system over a straight inheritance system hold. Error checking and classification both require that information associated with types is non-defeasible (as does constraint resolution). Non-default inheritance from psorts also turns out to be desirable because of its flexibility. The type system is taken to be complete, and various conditions are imposed on it, such as the greatest lower bound condition, which ensure that deterministic classification is possible. However this condition is quite difficult to comply with, and where no introduction of features is involved it is much easier to define inheritance in terms of psorts. In implementational terms a type scheme should be relatively static; any alterations may affect a large amount of data and checking that the scheme as a whole is still consistent is a non-trivial process. In contrast, adding psorts need not involve significant recomputation. The design of the description language was essentially motivated by the desire to use it to capture semantic relationships between lexical items, in particular the hyponymy relationship, which allows a fine-grain of inheritance that it is inappropriate to describe in the type system.

2.4.4 Rules

The rule mechanism involves no further extensions to LAUREL, but indicates how typing and inheritance may be applied to feature structures other than lexical entries. Grammar and lexical rules are typed feature structures, which represent relationships between two or more signs.

The expanded constraint for the type **rule** is:

$$\begin{bmatrix} \textbf{rule} \\ 0 = \textbf{sign} \\ 1 = \textbf{sign} \end{bmatrix}$$

All rules are feature structures of **rule** or some subtype of **rule**. Thus all rules have to have the features 0 and 1 which must both have values which are of type **sign**. Further features may be introduced as normal on subtypes of **rule**, but in the grammar treatment considered here all rules will be unary or binary:

$$\begin{bmatrix} \textbf{grammar-rule} \\ 0 = \textbf{sign} \\ 1 = \textbf{sign} \\ 2 = \textbf{sign} \end{bmatrix}$$

Rules can be regarded as a means of generating new signs; if a series of signs, $F_1, F_2 \ldots F_n$ can be unified with the feature structures at the end of the paths `<1>`, `<2>` ... `<n>` in the rule, then the feature structure at the end of the path `<0>` is a new sign. Linear order of constituents can be specified independently, but here we will always assume that linear order matches numerical order. A parser can operate by applying rules, either bottom-up according to the description above, or top-down, where an input sign can generate a series of output signs. Alternatively rules can be regarded statically, as expressing the relationship between existing signs.

As mentioned in section 2.4.2, lexical rules can be defined as feature structures of type **lexical-rule** which is a subtype of **rule**:

$$\begin{bmatrix} \textbf{lexical-rule} \\ 0 = \textbf{lex\_sign} \\ 1 = \textbf{lex\_sign} \end{bmatrix}$$

The notation for lexical rule application in the description language given above is not essential but simplifies the description which would otherwise involve an intermediate psort. For example:

```
( feather_1 + plural )
```

would be equivalent to

```
( plfeather < 0 >)
```

where `plfeather` was set up as follows:

```
plfeather
   < > <= plural < >
   < 1 > <= feather_1 < >.
```

This treatment is general enough to encode a variety of approaches to the notion of rule. Theoretical constraints on rules are assumed to be encoded in the type system. The ACQUILEX type system makes use of a UCG style categorial framework which combines three grammar rules (forward application, backward application and wrapping) with a series of unary and lexical rules for type shifting operations, but other treatments could be encoded. The parser makes no distinction between lexical and non-lexical rules, although lexical rules, as defined above, will not unify with any phrasal signs if these have a type which is incompatible with **lex_sign**.

The use of a rule-based approach may seem somewhat incongruous in a typed feature structure system, since most recent work has assumed the use of constraint-resolution. The basic LAUREL language can also be used in a constraint based system, along the same lines as those described by Carpenter (1992) and by Zajac (in press) as described in Chapter 4. However there are advantages to the use of rules. In practical terms it is much simpler to implement a rule-based parser than a general constraint solving mechanism and, currently, explicitly rule-based systems can be made reasonably efficient much more easily. To a large extent, a constraint based approach can be mapped into a rule-based one, given that rules are themselves typed feature structures, since principles such as the HPSG Head Feature Principle can be defined over rules, rather than phrasal signs. However there are some operations which are possible in constraint based systems which are not easy to map into a rule based approach. The two approaches are compared in Section 6.1.

Chapter 3

Default inheritance and default unification

The formalisation of defaults in LAUREL is based on the assumption that the use of defaults is confined to the paradigmatic level of description. Lexical entries and rules can be described in a language which makes use of defaults, but no notion of defeasibility is carried to the syntagmatic plane where individual instances of signs are combined. The motivation for the use of default inheritance in the lexicon has been discussed by Flickinger (1987) and Gazdar (1987, 1990). Essentially it seems as though it is necessary to have a notation which admits defaults in some way, in order to capture regularity of structure in the lexicon, since there are exceptions to many of the generalisations which one would like to state. The most straightforward examples are morphological, for example past participle formation by suffixation by *ed* can be described as a regular process which is blocked for verbs such as *hold* by the presence of the irregular form *held*, but syntactic and semantic exceptions can also be treated by defaults. For some recent examples of the use of defaults within unification based representations see Russell *et al.* (1991), Krieger and Nerbonne (in press), Cahill and Evans (1990).

However, the need for defaults is also acute in lexical semantics, where many generalisations have exceptions. Consider, for example, the connection between number agreement and semantic plurality of nominals in English. There are lexical exceptions to the generalisation that these correlate. One class of exceptions are pair nouns such as *scissors*, *trousers*, *binoculars*[1]. There are also individual examples of nouns which take either singular or plural agreement, although semantically they behave as singulars, such as *gallows*. Without going into the details of an analysis here, it seems intuitively that to account for these patterns a treatment of defaults will be needed which allows for exceptions both at the level of classes and of individual items.

We shall also make extensive use of default inheritance between lexical entries (rather than specially defined classes), in particular lexical entries which stand in a hyponymy relationship. As will be seen, this significantly influences the design of the default language. For example, *book* has hyponyms such as *autobiography*, *novel* and *dictionary*. We assume that the lexicon is structured in a way which reflects the hyponymy relationship, so that the lexical entries for *autobiography*, *novel*, *dictionary* will inherit lexical semantic information, encoded in the qualia structure, from *book*. But such inheritance must be

[1]Although there have been proposals that these be treated as semantically plural, Copestake (1992) argues that this is not reasonable.

defeasible. Although *book*, *autobiography* and *novel* all have their telic roles instantiated to *read*, this is inappropriate for *dictionary* which should have a telic role corresponding to *refer_to*. Furthermore, hyponyms of *dictionary*, such as *lexicon* (in the more generally used sense), will inherit the modified telic role of *refer_to* rather than read. Thus, as is usual, the inheritance of lexical information should be taken as obeying the 'Penguin Principle' familiar from inheritance networks in AI.[2] However, in this case, the nodes on the network do not correspond to classes which are set up by the linguist in order that the appropriate information is inherited, but instead correspond to full lexical entries which stand in a particular semantic relationship. Furthermore, these hyponymy relationships may be acquired semi-automatically from MRDs. This essentially precludes the use of complex hierarchical inheritance schemes which allow the linguist to determine the order in which defaults are overridden in case of conflicts, because the organisation of the hierarchy is not determined by the linguist.

So for example, assume that the following is part of the lexical entry for `book_L_1_1` (i.e. `book_L_1_1` is the name of the psort, where the code L_1_1 refers to the sense number in LDOCE).

$$\texttt{book\_L\_1\_1} = \begin{bmatrix} \textbf{lex-noun-sign} & \\ \text{QUALIA} = & \begin{bmatrix} \textbf{art\_phys} & \\ \text{TELIC} = & \begin{bmatrix} \textbf{verb-sem} \\ \text{PRED} = \textbf{read\_L\_1\_1} \end{bmatrix} \\ \text{FORM} = & \textbf{individual} \end{bmatrix} \end{bmatrix}$$

We have assumed the type system shown in the previous chapter, but we have omitted some features such as IND and COUNT from the diagram. The following path specifications make the lexical entries defined inherit their qualia structure from `book_L_1_1`:

autobiography L_0_0

`< QUALIA > < book_L_1_1 < QUALIA >.`

dictionary L_0_1

`< QUALIA > < book_L_1_1 < QUALIA >`

`< QUALIA : TELIC : PRED > = "`**refer_to_L_0_2**`" .`

lexicon L_0_0

`< QUALIA > < dictionary_L_0_1 < QUALIA > .`

`autobiography_L_0_1` thus has the same values as `book_L_1_1` for both TELIC and FORM. `dictionary_L_0_1` will inherit the value **individual** for the feature FORM but the value of the TELIC role, which is specified as **refer_to_L_0_2**, overrides that which would be inherited from `book_L_1_1` (**read_L_1_1**). `lexicon_L_0_0` inherits its value for the telic role from `dictionary_L_0_1` rather than from `book_L_1_1`.

$$\texttt{lexicon\_L\_0\_0} = \begin{bmatrix} \textbf{lex-noun-sign} & \\ \text{QUALIA} = & \begin{bmatrix} \textbf{art\_phys} & \\ \text{TELIC} = & \begin{bmatrix} \textbf{verb-sem} \\ \text{PRED} = \textbf{refer\_to\_L\_0\_2} \end{bmatrix} \\ \text{FORM} = & \textbf{individual} \end{bmatrix} \end{bmatrix}$$

[2] The Penguin Principle states that when there is a conflict between defaults, the default which applies to a more specific class should apply in preference to one which applies to a more general class. The eponymous example is:
Birds can normally fly.
All penguins are birds.
Penguins cannot normally fly.
Tweety is a penguin.
Therefore conclude (non-monotonically) that Tweety cannot fly.

The representation of hyponymy is discussed in detail in Copestake (1992). The main purpose of this chapter is to consider the formal definition of default inheritance. Default inheritance in LAUREL involves default unification and a large part of this chapter will involve a discussion of various definitions of default unification. It will become apparent that default inheritance in feature structure languages does relate to more general notions of default inheritance in AI.

Default inheritance in LAUREL is defined in terms of the non-local description language, discussed in the previous chapter. As we mentioned there non-default inheritance is implemented by unification of the feature structure with a copy of the relevant part of the psort. The default inheritance mechanism is similar except that default unification, rather than ordinary unification is involved. We will use $A \overset{<}{\sqcap} B$ to indicate default unification where A is non-default. The effect of default unification is that incompatible values for attributes are ignored, rather than causing unification failure. For example:

$$F = \begin{bmatrix} \textbf{lex-noun-sign} \\ \text{ORTH} = \textbf{string} \\ \text{COUNT} = \textbf{boolean} \\ \text{QUALIA} = \begin{bmatrix} \textbf{animal} \\ \text{FORM} = \textbf{individual} \\ \text{SEX} = \textbf{male} \end{bmatrix} \end{bmatrix} \quad F' = \begin{bmatrix} \textbf{lex-noun-sign} \\ \text{ORTH} = \textbf{ewe} \\ \text{COUNT} = \textbf{true} \\ \text{QUALIA} = \begin{bmatrix} \textbf{animal} \\ \text{FORM} = \textbf{form} \\ \text{SEX} = \textbf{female} \end{bmatrix} \end{bmatrix}$$

$$F \sqcap F' = \bot$$

$$F \overset{<}{\sqcap} F' = \begin{bmatrix} \textbf{lex-noun-sign} \\ \text{ORTH} = \textbf{ewe} \\ \text{COUNT} = \textbf{true} \\ \text{QUALIA} = \begin{bmatrix} \textbf{animal} \\ \text{FORM} = \textbf{individual} \\ \text{SEX} = \textbf{male} \end{bmatrix} \end{bmatrix}$$

Since psorts may themselves inherit information, default inheritance in effect operates over a hierarchy of psorts and the ordering on the psort hierarchy gives us an ordering on defaults (since cycles in the inheritance ordering are prohibited). However once information is incorporated into a feature structure no distinction between inherited and non-inherited information is maintained and a psort will be fully expanded before it is used: inheritance thus operates top-down.

Our variant of default unification is restricted by the type system so that a feature structure will not inherit any information from a feature structure of incompatible type. However, we will begin by discussing default unification of untyped feature structures.

3.1 Default unification of untyped feature structures

There are now a considerable number of descriptions of non-monotonic operations on feature structures. Shieber (1986) described overwriting in PATR-II: this allows particular atomic values of feature structures to be non-monotonically altered. Kaplan's (1987) outline of priority union is more similar to default unification. It does apply to non-atomic feature structures but does not allow for cases where the feature structures are reentrant, which can give rise to conflicts between default information. Recent definitions which do consider general feature structures have been put forward by Bouma (1990a,b), Calder (1991), Carpenter (1991, in press), Russell *et al.* (1991) and van den Berg and Prüst (1991). The issues are discussed in detail by Carpenter; here we wish to review his proposal and to extend the discussion slightly in order to illustrate the varieties of default unification which have been proposed.

Default unification is defined so that when a non-default feature structure is unified with a default feature structure only values in the default structure which do not conflict with values in the non-default structure are incorporated. It should ideally have the following properties:

1. $A \overset{<}{\sqcap} B \sqsubseteq A$
   Default unification adds information monotonically to the non-default.
   Clearly it should not be possible to remove non-default information, and all definitions of default unification of which we are aware do meet this criterion.

2. if $A \sqcap B \neq \bot$ then $A \sqcap B = A \overset{<}{\sqcap} B$
   Default unification behaves like ordinary unification in the cases where ordinary unification would succeed.
   Intuitively ordinary unification should correspond to the case where the default feature structure is totally compatible with the non-default structure.

3. $A \overset{<}{\sqcap} B \neq \bot$
   Default unification never fails.
   This is a property which could be taken as definitional for default unification. However, as shown later, some versions of default unification drop this condition (or put preconditions on default unification such that it cannot be applied to some pairs of feature structures) in order to meet the other requirements.

4. Default unification returns a single result, deterministically.
   It seems desirable not to introduce non-determinism into the system. Multiple results are awkward both from the point of view of implementation and usability.

It is, in general, necessary that default unification be implementable with reasonable efficiency and it is highly desirable that it give results which are intuitively plausible to users.

The examples of default unification given earlier are unproblematic. However there are cases where there are conflicts between parts of the default information, because of reentrancy in the default or in the non-default feature structure, as in the following examples:

(1) a $\begin{bmatrix} \mathrm{F} = \boxed{1} \\ \mathrm{G} = \boxed{1} \\ \mathrm{H} = \top \end{bmatrix} \overset{<}{\sqcap} \begin{bmatrix} \mathrm{F} = a \\ \mathrm{G} = b \\ \mathrm{H} = c \end{bmatrix}$

b $\begin{bmatrix} \mathrm{F} = a \\ \mathrm{G} = \top \\ \mathrm{H} = \top \end{bmatrix} \overset{<}{\sqcap} \begin{bmatrix} \mathrm{F} = \boxed{1}\ b \\ \mathrm{G} = \boxed{1} \\ \mathrm{H} = c \end{bmatrix}$

As shown in section 3.1.2 the various definitions of default unification have been proposed are not in agreement about the result of default unification for such examples. The difficulty in defining default unification is to exclude the possibility of the result depending on the order in which individual parts of the default feature structure are unified with the non-default feature structure. All the definitions which we will discuss here exclude such order dependence, but they do so in different ways.

3.1.1 Definitions of default unification

In many ways we regard Carpenter's (1991) definition of skeptical default unification as the ideal; it meets all the conditions enumerated above and has a definition which can be simply paraphrased as "incorporate the maximal consistent information from the default". We repeat Carpenter's definitions here (as usual with order reversed)[3].

Definition 25 (Credulous Default Unification) *The result of credulously adding the default information in G to the strict information in F is given by:*

> $F \stackrel{<}{\sqcap}_c G = \{F \sqcap G' \mid G \sqsubseteq G'$ *and* G' *is maximally specific such that* $F \sqcap G'$ *is defined*$\}$

Credulous default unification gives multiple results, for example:

(2) a $$\begin{bmatrix} \mathrm{F} = \boxed{1} \\ \mathrm{G} = \boxed{1} \\ \mathrm{H} = \top \end{bmatrix} \stackrel{<}{\sqcap}_c \begin{bmatrix} \mathrm{F} = a \\ \mathrm{G} = b \\ \mathrm{H} = c \end{bmatrix} = \left\{ \begin{bmatrix} \mathrm{F} = \boxed{1}\ a \\ \mathrm{G} = \boxed{1} \\ \mathrm{H} = c \end{bmatrix}, \begin{bmatrix} \mathrm{F} = \boxed{1}\ b \\ \mathrm{G} = \boxed{1} \\ \mathrm{H} = c \end{bmatrix} \right\}$$

b $$\begin{bmatrix} \mathrm{F} = a \\ \mathrm{G} = \top \\ \mathrm{H} = \top \end{bmatrix} \stackrel{<}{\sqcap}_c \begin{bmatrix} \mathrm{F} = \boxed{1}\ b \\ \mathrm{G} = \boxed{1} \\ \mathrm{H} = c \end{bmatrix} = \left\{ \begin{bmatrix} \mathrm{F} = \boxed{1}\ a \\ \mathrm{G} = \boxed{1} \\ \mathrm{H} = c \end{bmatrix}, \begin{bmatrix} \mathrm{F} = a \\ \mathrm{G} = b \\ \mathrm{H} = c \end{bmatrix} \right\}$$

Skeptical default unification results in only the information which is common to all the credulous results being retained.

Definition 26 (Skeptical Default Unification) *The result of skeptical default unification is the generalisation of credulous default unification:* $F \stackrel{<}{\sqcap}_s G = \bigsqcup(F \stackrel{<}{\sqcap}_c G)$

(3) a $$\begin{bmatrix} \mathrm{F} = \boxed{1} \\ \mathrm{G} = \boxed{1} \\ \mathrm{H} = \top \end{bmatrix} \stackrel{<}{\sqcap}_s \begin{bmatrix} \mathrm{F} = a \\ \mathrm{G} = b \\ \mathrm{H} = c \end{bmatrix} = \begin{bmatrix} \mathrm{F} = \boxed{1} \\ \mathrm{G} = \boxed{1} \\ \mathrm{H} = c \end{bmatrix}$$

b $$\begin{bmatrix} \mathrm{F} = a \\ \mathrm{G} = \top \\ \mathrm{H} = \top \end{bmatrix} \stackrel{<}{\sqcap}_s \begin{bmatrix} \mathrm{F} = \boxed{1}\ b \\ \mathrm{G} = \boxed{1} \\ \mathrm{H} = c \end{bmatrix} = \begin{bmatrix} \mathrm{F} = a \\ \mathrm{G} = \top \\ \mathrm{H} = c \end{bmatrix}$$

In order to compare Carpenter's definition with some of the other varieties of default unification which have been proposed, We will formalise them all in terms of successively unifying pieces of information carried by the default into the non-default feature structure while taking account of possible conflicts (cf. Russell *et al.*, 1991). A critical notion here is that of "pieces of information"; we will make use of a general notion of decomposition of a feature structure $Decomp(F)$ into component pieces of information. This must meet the following criterion (if default unification is to have the property of being equivalent to ordinary unification in the cases where that would succeed):

> $\sqcap(Decomp(F)) = F$
> F is equal to the unification of all the information in its decomposition.

The first case to consider is where decomposition is into the minimal atomic units of information. Carpenter's definition of atomic feature structures is as follows:

Definition 27 (Atomic Feature Structure) *A feature structure is* atomic *if it is of one of the following two forms:*

---

[3]Calder (1991) independently defined priority union in a way which is equivalent to Carpenter's definition of credulous default unification. A notion of maximal incorporation of information which is very similar to Carpenter's has also been proposed by van den Berg and Prüst (1991). However in their paper the specific application to feature structures uses a similar process of normalisation to that of Bouma (see below), which results in reentrant structures with specified values being treated as equivalent to the non-reentrant case, where the paths terminate in distinct identical values.

- (Path Value)
the feature structure contains a single path assigned to an atomic value.
- (Path Sharing)
the feature structure contains only a pair of (possibly identical) paths which are shared.

The function At when applied to a feature structure F gives the set of atomic units:

$$At(F) = \{F \sqsubseteq F' \mid F' \text{ atomic}\}$$

Carpenter shows that At meets the criterion for the decomposition function (i.e. $\sqcap(At(F)) = F$).

A definition of default unification in terms of incorporation of atomic feature structures can thus be given which is equivalent to Carpenter's skeptical default unification:

Definition 28 (Skeptical Default Unification)
$F_1 \overset{<}{\sqcap} F_2 = F_1 \sqcap \bigsqcap\{F \in At(F_2) \mid F_1 \sqcap F \neq\perp$ *and there is no* F' *such that* $F_2 \sqsubseteq F'$ *and* $F' \sqcap F_1 \neq\perp$ *and* $F' \sqcap F_1 \sqcap F =\perp\}$

The intuitive basis for this definition is to consider successively adding the minimal (atomic) units of information from the default into the non-default. In the cases where there are conflicts, such as the examples just given, this would give different results depending on the order in which the atomic feature structures were added in. To produce the equivalent of credulous default unification we would do the addition once for each possible ordering of default atomic feature structures (and remove duplicates). The definition above is equivalent to skeptical default unification because only information which is consistent with all possible orderings is added. It is thus obvious that the complexity of the algorithm as described is exponential since checking for all possible F' would involve creating the unification of each member of the power-set of $At(F_2)$. An illustration of the type of feature structure which exhibits this worst case behaviour is shown in example 5, below.

The option taken by Russell *et al.* (1991) is to keep the tractable (near-linear) behaviour of ordinary unification by defining default unification in such a way that it fails under the circumstances where there are conflicts in the default information. In terms of the definition above $At(F)$ is split into $PE(F)$, the set of path equivalence specifications, and $PV(F)$ the set of path value specifications. If the reentrant part of the default unifies with the non-default, and the reentrant part of the non-default unifies with the default, no conflicts can arise in the default information. Thus Russell *et al.* have:

Definition 29 (Russell et al. default unification)

$$\begin{aligned} F_1 \overset{<}{\sqcap} F_2 &= \perp \ \textit{if} \sqcap\mathrm{PE}(F_2) \sqcap F_1 = \perp \\ &\qquad \textit{or} \sqcap\mathrm{PE}(F_1) \sqcap F_2 = \perp \\ &= F_1 \sqcap \bigsqcap\{F \in At(F_2) \mid F_1 \sqcap F \neq\perp\} \ \textit{otherwise} \end{aligned}$$

The ACQUILEX LKB's original default unification algorithm also made use of a distinction between reentrant and non-reentrant atomic feature structures. The definition used was:

Definition 30 (ACQUILEX LKB original)
$F_1 \overset{<}{\sqcap} F_2 = F_1 \sqcap \bigsqcap\{F \in At(F_2) \mid F_1 \sqcap F \neq \perp$ and for all conflicting F' such that $F_2 \sqsubseteq F'$ and $F' \sqcap F_1 \neq \perp$ and $F' \sqcap F_1 \sqcap F = \perp$, F "takes precedence over" $F'\}$
Where F takes precedence over F' iff F is a specification of path equivalence ($F \in \mathrm{PE}(F_1)$) and F' contains at least one path value specification.

Thus a precedence order was introduced between path equivalence and path value specifications. This was actually done because the linguists involved in designing the LRL expressed a preference for a behaviour where reentrancy took precedence over values; for the application to automatic acquisition of information from MRDs in particular this was desirable because reentrancy is usually set up manually, see e.g. Sanfilippo (in press), whereas values are more likely to be acquired automatically. However it was found in practice that reentrancy tended to be set up in the type system, and was thus, in effect, non-default. This is considered further below.

This approach to default unification can, in practice, be implemented considerably more efficiently than Carpenter's, although its worst case behaviour is still exponential. Initially the re-entrant parts of the default feature structure can be unified individually with the non-default, and it is only necessary to consider conflicts that arise in the reentrant set. Thus the exponential term involves only the path equivalence specifications, and since typically there are many fewer path equivalence specifications than path value specifications the implementation is not unreasonably slow. Furthermore this is the worst case behaviour; it is usually possible to split $\mathrm{PE}(F)$ into sets which are guaranteed not to interact. The procedure then reduces to one of default unifying a tree-structured feature structure with a non-default reentrant structure. There are still possible conflicts, of the type in example 3 below (which would cause unification failure by Russell *et al.*'s definition). However it is possible to allow for these with a near-linear algorithm by storing the original non-default value in the feature structure representation at reentrant points as unification proceeds, and reverting to it if a conflict arises. In effect, what this relies on is that if the non-default feature structure is the only reentrant one, all conflicts are localised.

However this definition still seems unsatisfactory, even though it meets all the criteria we enumerated at the beginning of this section. The worst case complexity is exponential, the implementation is awkward, and the behaviour can be obscure. A better compromise for the ACQUILEX LKB seemed to be to specify that inheritance of information about reentrancy is non-defeasible, and that default unification fails in the case where the non-default feature structure and the reentrant part of the default feature structure do not unify.

Definition 31 (ACQUILEX LKB)
$F_1 \overset{<}{\sqcap} F_2 = F_3 \sqcap \bigsqcap\{F \mid F \in PV(F_2)$ and $F_1 \sqcap F \neq \perp$ and there is no $F' \in PV(F_2)$ such that $F' \sqcap F_3 \neq \perp$ and $F' \sqcap F_3 \sqcap F = \perp\}$
where $F_3 = F_1 \sqcap \bigsqcap PE(F_2)$

Such a definition, where default unification involves filling in values, and expanding, rather than modifying, existing feature structure skeletons, is relatively simple to understand.[4] It

[4]It also avoids the rather complex behaviour of Carpenter's definition with respect to the difference between specified and unspecified paths, see Carpenter (1991). However we will not discuss that further here, since default unification of totally typed features structures, as used in LAUREL, avoids the problem in any case.

does, however, violate the third criterion that we gave earlier, that default unification never fail. In practice this definition will be adequate if reentrancy is not seen as defeasible, which turned out to be the case for the ACQUILEX LKB system, where the path equivalence specifications were always set up in the type system.

Bouma's (1990a,b) treatment of default unification can also be described in terms of addition of pieces of information from the default structure to the non-default structure. However we will not attempt to reformalise it in these terms here, since the mechanism by which conflicts are excluded is complex and Bouma's definition is lengthy. Instead we will give examples of the behaviour of Bouma's approach below, which illustrate that it does not conform to all of the original criteria.

3.1.2 Examples of default unification

The following examples illustrate the differences in behaviour between the definitions that have been proposed, by Bouma (1990a,b), Carpenter (1991), Calder (1991), Russell *et al.* (1991). We have limited these examples to the cases where the default and non-default structures have the same features, since this discussion is essentially a precursor to the description of default unification in a totally typed system.

(1) $\begin{bmatrix} \mathrm{F} = a \\ \mathrm{G} = \top \end{bmatrix} \overset{<}{\sqcap} \begin{bmatrix} \mathrm{F} = b \\ \mathrm{G} = c \end{bmatrix} = \begin{bmatrix} \mathrm{F} = a \\ \mathrm{G} = c \end{bmatrix}$

This is the simplest case of default unification; the conflicting information in the default is ignored, but the non-conflicting information is incorporated.

(2) $\begin{bmatrix} \mathrm{F} = \boxed{1} \\ \mathrm{G} = \boxed{1} \end{bmatrix} \overset{<}{\sqcap} \begin{bmatrix} \mathrm{F} = a \\ \mathrm{G} = \top \end{bmatrix} = \begin{bmatrix} \mathrm{F} = \boxed{1} \\ \mathrm{G} = \boxed{1} \end{bmatrix}$ Bouma

$= \begin{bmatrix} \mathrm{F} = \boxed{1}\ a \\ \mathrm{G} = \boxed{1} \end{bmatrix}$ other definitions

We include this example to illustrate that Bouma's definition of default unification does not meet the second criterion, since it gives a different result from ordinary unification. This arises because in Bouma's approach any pieces of information in the default which potentially conflict with each other are excluded, and Bouma's algorithm treats the values of F and G as being in potential conflict. In practice this behaviour is presumably not apparent since Bouma proposes that ordinary unification is attempted before default unification.

(3) $\begin{bmatrix} \mathrm{F} = \boxed{1} \\ \mathrm{G} = \boxed{1} \end{bmatrix} \overset{<}{\sqcap} \begin{bmatrix} \mathrm{F} = a \\ \mathrm{G} = b \end{bmatrix} = \left(\begin{bmatrix} \mathrm{F} = \boxed{1}\ a \\ \mathrm{G} = \boxed{1} \end{bmatrix}, \begin{bmatrix} \mathrm{F} = \boxed{1}\ b \\ \mathrm{G} = \boxed{1} \end{bmatrix} \right)$ Calder Carpenter (credulous)

$= \begin{bmatrix} \mathrm{F} = \boxed{1} \\ \mathrm{G} = \boxed{1} \end{bmatrix}$ Bouma Carpenter (skeptical) ALKB

$= \perp$ Russell *et al.*

Here the presence of reentrancy in the non-default means that the two default values are in conflict. A credulous definition will return multiple values; skeptical definitions return only that information which is common to all the credulous results.

$$
(4) \quad
\begin{array}{lll}
\begin{bmatrix} \mathrm{F} = a \\ \mathrm{G} = \top \end{bmatrix} \overset{<}{\sqcap} \begin{bmatrix} \mathrm{F} = \boxed{1}\ b \\ \mathrm{G} = \boxed{1} \end{bmatrix} & = \left(\begin{bmatrix} \mathrm{F} = \boxed{1}\ a \\ \mathrm{G} = \boxed{1} \end{bmatrix}, \begin{bmatrix} \mathrm{F} = a \\ \mathrm{G} = b \end{bmatrix} \right) & \text{Calder Carpenter (credulous)} \\
 & = \begin{bmatrix} \mathrm{F} = a \\ \mathrm{G} = \top \end{bmatrix} & \text{Carpenter (skeptical)} \\
 & = \begin{bmatrix} \mathrm{F} = \boxed{1}\ a \\ \mathrm{G} = \boxed{1} \end{bmatrix} & \text{ALKB} \\
 & = \begin{bmatrix} \mathrm{F} = a \\ \mathrm{G} = b \end{bmatrix} & \text{Bouma} \\
 & = \bot & \text{Russell } \textit{et al.}
\end{array}
$$

In example 3 there is no basis for deciding which of the conflicting information in the default structure should be incorporated. However in example 4 we would claim that there is a basis for distinguishing between the two pieces of default information which could be potentially incorporated but which are in conflict with each other, since one involves a path equivalence specification and the other a path value specification;

```
< F > = < G >
< G > = b
```

As described above, in the ACQUILEX LKB a decision was made to prefer specifications of equivalence to specifications of values. Bouma's result arises because he normalises default structures, in a way which gives:

```
< F > = b
< G > = b
```

for the default structure in example 4, whereas Carpenter's decomposition function *At* would give:

```
< F > = b
< G > = b
< F > = < G >
```

Bouma's normalisation function thus does not meet my criterion for the decomposition function, since the result of reunifying the pieces would be:

$$\begin{bmatrix} \mathrm{F} = b \\ \mathrm{G} = b \end{bmatrix}$$

Bouma justifies this on the basis that this structure has equivalent behaviour with respect to unification to the original default structure. This is a reasonable position to take if feature structure values are defined to be extensional (see section 2.2.5), but it does not seem natural for an intensional treatment, such as adopted in LAUREL, nor does it extend naturally to the default unification of typed feature structures, where atomic values are not necessarily maximally specific.

(5)

$$\begin{bmatrix} \mathrm{F} = \top \\ \mathrm{G} = a \\ \mathrm{H} = b \\ \mathrm{J} = \top \end{bmatrix} \overset{<}{\sqcap} \begin{bmatrix} \mathrm{F} = \boxed{1} \\ \mathrm{G} = \boxed{1} \\ \mathrm{H} = \boxed{1} \\ \mathrm{J} = \boxed{1} \end{bmatrix} = \text{(4 possibilities)} \quad \text{Calder, Carpenter (credulous)}$$

$$= \begin{bmatrix} \mathrm{F} = \top \\ \mathrm{G} = a \\ \mathrm{H} = b \\ \mathrm{J} = \top \end{bmatrix} \quad \text{Carpenter (skeptical), ALKB (old)}$$

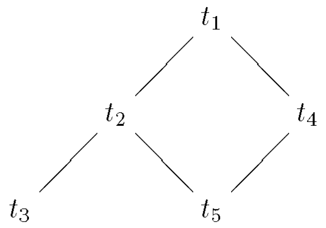
$$= \begin{bmatrix} \mathrm{F} = \boxed{1} \\ \mathrm{G} = a \\ \mathrm{H} = b \\ \mathrm{J} = \boxed{1} \end{bmatrix} \quad \text{Bouma}$$

$$= \bot \quad \text{Russell \textit{et al.}, ALKB (current)}$$

Here the conflict in the default information is entirely between specifications of reentrancy. This example illustrates why default unification involving maximal incorporation of information has exponential complexity; the atomic feature structure `< F > = < J >` conflicts only with a particular combination of the other atomic feature structures from the default.

3.2 Default unification of typed feature structures

Consideration of typed feature structures further increases the possible definitions of default unification. We have not attempted to even approximate to a definition which corresponds to incorporating the maximal amount of information carried by the types in a feature structure, although Carpenter's definition could be extended in this way. There are two main reasons for this; first, as mentioned earlier, in LAUREL default inheritance is constrained by using the type system. Furthermore, maximal incorporation of typed information can give results which are quite unintuitive. Consider the following type system:



We assume the atomic types **true** and **false** to be subtypes of **bool** and that the constraints on the other types are:

$$C(t_1) = \begin{bmatrix} t_1 \\ \mathrm{F} = \mathbf{bool} \end{bmatrix} \; C(t_2) = \begin{bmatrix} t_2 \\ \mathrm{F} = \mathbf{bool} \\ \mathrm{G} = \mathbf{bool} \end{bmatrix} \; C(t_3) = \begin{bmatrix} t_3 \\ \mathrm{F} = \mathbf{bool} \\ \mathrm{G} = \mathbf{true} \end{bmatrix}$$

$$C(t_4) = \begin{bmatrix} t_4 \\ \mathrm{F} = \mathbf{bool} \\ \mathrm{H} = \mathbf{bool} \end{bmatrix} \; C(t_5) = \begin{bmatrix} t_5 \\ \mathrm{F} = \mathbf{true} \\ \mathrm{G} = \mathbf{bool} \\ \mathrm{H} = \mathbf{true} \end{bmatrix}$$

This gives the following result for a definition of default unification based on unification of the non-default structure with the maximally informative well-formed feature structures which subsume the default, followed by generalisation of the results:

(6) $$\begin{bmatrix} t_4 \\ \mathrm{F} = \mathbf{bool} \\ \mathrm{H} = \mathbf{bool} \end{bmatrix} \overset{<}{\sqcap} \begin{bmatrix} t_3 \\ \mathrm{F} = \mathbf{bool} \\ \mathrm{G} = \mathbf{true} \end{bmatrix} = \begin{bmatrix} t_5 \\ \mathrm{F} = \mathbf{true} \\ \mathrm{G} = \mathbf{true} \\ \mathrm{H} = \mathbf{true} \end{bmatrix}$$

The result seems wrong, because information which is not carried by either feature structure is appearing in the result. The effect arises because t_3 and t_4 are incompatible types and the maximally informative well-formed feature structure which subsumes the default and unifies with the non-default is:

$$(7)\quad \begin{bmatrix} t_2 & & \\ \mathrm{F} & = & \mathbf{bool} \\ \mathrm{G} & = & \mathbf{true} \end{bmatrix}$$

This is identical to the default structure, except that its type is t_2, the immediate parent of the type of the default, t_3. Types t_2 and t_4 are compatible, but their greatest lowest bound is type t_5, which introduces extra constraints, and thus we obtain the additional information in the result of default unification.

The situation becomes even more complex if we add the further information to the default about the value of F even though this does not conflict with the non-default:

$$(8)\quad \begin{bmatrix} t_4 & & \\ \mathrm{F} & = & \mathbf{bool} \\ \mathrm{H} & = & \mathbf{bool} \end{bmatrix} \overset{<}{\sqcap} \begin{bmatrix} t_3 & & \\ \mathrm{F} & = & \mathbf{false} \\ \mathrm{G} & = & \mathbf{true} \end{bmatrix} = \begin{bmatrix} t_4 & & \\ \mathrm{F} & = & \mathbf{bool} \\ \mathrm{H} & = & \mathbf{bool} \end{bmatrix}$$

This result arises because there are two maximally specific feature structures which subsume the default, and well-formed unify with the non-default:

$$(9)\quad \begin{bmatrix} t_2 & & \\ \mathrm{F} & = & \mathbf{bool} \\ \mathrm{G} & = & \mathbf{true} \end{bmatrix}, \begin{bmatrix} t_1 & & \\ \mathrm{F} & = & \mathbf{false} \end{bmatrix}$$

To work out the result we generalise over the result of unifying the non-default structure with each of these:

$$(10)\quad \begin{bmatrix} t_5 & & \\ \mathrm{F} & = & \mathbf{true} \\ \mathrm{G} & = & \mathbf{true} \\ \mathrm{H} & = & \mathbf{true} \end{bmatrix} \sqcup \begin{bmatrix} t_4 & & \\ \mathrm{F} & = & \mathbf{false} \\ \mathrm{H} & = & \mathbf{bool} \end{bmatrix}$$

These results seem unintuitive; adopting this approach leads to a situation where working out the results of default unification involves considering the interaction of the information ordering given by the type hierarchy with that of the feature structure system.

Thus in LAUREL a definition was used in which the type system constrains default inheritance. In both the examples above, since t_3 and t_4 are incompatible types, default unification would simply return the non-default feature structure. In general information which is carried by any feature structure which is part of the default is incorporated if it is fully compatible with the non-default (i.e. unifies with the relevant part), but is only ever partially incorporated (i.e. default unified) if its type is the same as, or more general than, the non-default.

This can be formalised in terms of a decomposition function (TypeDecomp) which differs from the atomic decomposition function in that it does not split up the feature structures completely. Only parts of the feature structure which are fully type compatible with the non-default structure are split; TypeDecomp is thus defined relative to the non-default structure. In what follows reentrancy is regarded as non-defeasible.

Definition 32 (Decomposition) *If $F_1 = \langle Q_1, q_0, \delta_1, \alpha_1 \rangle$ and $F_2 = \langle Q_2, q_0', \delta_2, \alpha_2 \rangle$ where F_1 contains no path equivalence specifications then we define the decomposition of F_1 with respect to F_2,* TypeDecomp(F_1, F_2), *as follows:*

If $\alpha_2(q'_0) \sqsubseteq \alpha_1(q_0)$ *then* $\mathrm{TypeDecomp}(F_1, F_2)$ *is the set of all feature structures,* $F_n = \langle Q_n, q''_0, \delta_n, \alpha_n \rangle$ *such that:*

- $\alpha_n(q''_0) = \alpha_1(q_0)$
- *If* F_1 *is non-atomic (i.e.* $\mathrm{Feat}_0(F_1)$ *is non-empty), then* $\delta_n(q''_0, f)$ *for some* $f \in \mathrm{Feat}_0(F_1)$ *is a member of* $\mathrm{TypeDecomp}(\delta_1(q_0, f), \delta_2(q'_0, f))$ *(where because of the type compatability condition* $\delta_2(q'_0, f)$ *must be defined) and* $\delta_n(q''_0, f')$ *for all* $f' \neq f$ *is undefined*

If $\alpha_2(q'_0) \not\sqsubseteq \alpha_1(q_0)$ *then* $\mathrm{TypeDecomp}(F_1, F_2) = \{F_1\}$

Note that the members of $\mathrm{TypeDecomp}(F_1, F_2)$ are not necessarily well-formed feature structures by this definition.

Typed default unification is then defined as follows:

Definition 33 (Typed default unification) $F_1 \overset{<}{\sqcap} F_2 = F_3 \sqcap \bigsqcap\{F \mid F \in D(F_2)$ *and* $F_1 \sqcap F \neq \perp$ *and there is no* $F' \in D(F_2)$ *such that* $F' \sqcap F_3 \neq \perp$ *and* $F' \sqcap F_3 \sqcap F = \perp\}$
where $F_3 = F_1 \sqcap \bigsqcap PE(F_2)$ *and* $D(F_2)$ *is* $\mathrm{TypeDecomp}(\sqcap PV(F_2), F_1)$

This is the definition used in LAUREL when no equality constraints are involved. For example, given:

$$F_1 = \begin{bmatrix} \mathbf{t1} & \\ \mathrm{F} = & \begin{bmatrix} \mathbf{t2} \\ \mathrm{g} = \mathbf{t5} \\ \mathrm{h} = \top \end{bmatrix} \\ \mathrm{J} = & \begin{bmatrix} \mathbf{t3} \\ \mathrm{g} = \mathbf{t5} \\ \mathrm{h} = \top \end{bmatrix} \end{bmatrix} ; F_2 = \begin{bmatrix} \mathbf{t1} & \\ \mathrm{F} = & \begin{bmatrix} \mathbf{t2} \\ \mathrm{g} = \mathbf{t6} \\ \mathrm{h} = \mathbf{t7} \end{bmatrix} \\ \mathrm{J} = & \begin{bmatrix} \mathbf{t4} \\ \mathrm{g} = \mathbf{t6} \\ \mathrm{h} = \mathbf{t7} \end{bmatrix} \end{bmatrix} ; \mathbf{t5} \sqcap \mathbf{t6} = \perp \text{ and } \mathbf{t3} \not\sqsubseteq \mathbf{t4}$$

then:

$$F_1 \overset{<}{\sqcap} F_2 = \begin{bmatrix} \mathbf{t1} & \\ \mathrm{F} = & \begin{bmatrix} \mathbf{t2} \\ \mathrm{g} = \mathbf{t5} \\ \mathrm{h} = \mathbf{t7} \end{bmatrix} \\ \mathrm{J} = & \begin{bmatrix} \mathbf{t3} \\ \mathrm{g} = \mathbf{t5} \\ \mathrm{h} = \top \end{bmatrix} \end{bmatrix}$$

since $\mathrm{TypeDecomp}(\sqcap PV(F_2), F_1) =$

$$\left\{ \begin{bmatrix} \mathbf{t1} & \\ \mathrm{F} = & \begin{bmatrix} \mathbf{t2} \\ \mathrm{g} = \mathbf{t6} \end{bmatrix} \end{bmatrix}, \begin{bmatrix} \mathbf{t1} & \\ \mathrm{F} = & \begin{bmatrix} \mathbf{t2} \\ \mathrm{h} = \mathbf{t7} \end{bmatrix} \end{bmatrix}, \begin{bmatrix} \mathbf{t1} & \\ \mathrm{J} = & \begin{bmatrix} \mathbf{t4} \\ \mathrm{g} = \mathbf{t6} \\ \mathrm{h} = \mathbf{t7} \end{bmatrix} \end{bmatrix} \right\}$$

The following definition is for the combination of a non-default and a default feature structure in the presence of equality constraints:

Definition 34 (Typed default unification with equality constraints) *The default unification of a feature structure* F_1 *with a default feature structure* F_2 *given a set of equality constraints,* C*, is given by* $\langle F_1, C \rangle \overset{<}{\sqcap}_e F_2$
$= F_3 \sqcap \bigsqcap\{F \mid F \in D(F_2)$ *and* $F_1 \sqcap F \neq \perp$ *and for all* $\phi \in C$*,* $F_1 \sqcap F$ *satisfies* ϕ*, and there is no* $F' \in D(F_2)$ *such that* $F' \sqcap F_3 \neq \perp$ *and* $F' \sqcap F_3 \sqcap F = \perp\}$
where $F_3 = F_1 \sqcap \bigsqcap PE(F_2)$ *and* $D(F_2)$ *is* $\mathrm{TypeDecomp}(\sqcap PV(F_2), F_1)$

where the definition of satisfaction of an equality constraint repeated from Section 2.4.2 is:

Definition 35 (Feature structure equality constraint) *A feature structure* $F = \langle Q, q_0, \delta, \alpha \rangle$ *satisfies a feature structure equality constraint* $\pi == F'$ *iff* $F@\pi \sim F'$

Definition 36 (Type equality constraint) *A feature structure* $F = \langle Q, q_0, \delta, \alpha \rangle$ *satisfies a type equality constraint* $\pi =\ast\ t$ *iff* $F@\pi \sim C(t)$

The motivation for equality constraints is considered in detail in Section 3.4.

3.3 Inheritance hierarchies and multiple inheritance

Clearly, since default unification is non-monotonic, the order in which a sequence of default unification operations are carried out will be significant. Carpenter (1991) discusses inheritance based on templates and inheritance over an explicitly defined hierarchy. Inheritance in LAUREL is similar to Carpenter's template scheme in that the inheritance hierarchy is implicit, but unlike it in that complete feature structures rather than descriptions are related.

One way of allowing inheritance to operate over a hierarchy would be to modify the definition of default unification, to order the information units in a way which corresponded to the inheritance hierarchy (compare the way in which we defined preference of path equivalence specifications to path value specifications). This would give a definition which, in effect, ignored the level of granularity given by the complete feature structures. Clearly this is not any more computationally feasible than the formulation of default unification which we gave originally. Conflicts can arise, not just from reentrancy, but also from multiple inheritance conflicts of the 'Nixon diamond' type, where there is no ordering between defaults to allow resolution. Again we could produce variant definitions; if reentrancy is regarded as non-defeasible for example, all the reentrant atomic feature structures could be unified first and if that succeeded the non-reentrant structures could be considered in groups according to their priority. Essentially definitions along these lines give a skeptical, 'bottom-up', inheritance scheme, but there are a whole range of possibilities, which give more or less intuitive results.

The definition used in LAUREL was intended to give behaviour which did respect the integrity of the feature structures, which was relatively simple to understand and which could be used with data automatically acquired from MRDs (at the cost of some flexibility in ordering defaults). The semantics of the non-local description language leads to inheritance operating top-down over whole feature structures; that is a psort will be fully expanded with inherited information before it is used for default inheritance. Once a psort is expanded, no distinction is retained between inherited and locally specified information. As Carpenter explains top-down and bottom-up definitions give differing behaviour since default unification is non-associative. The particular example that Carpenter uses does not have non-associative behaviour under our definition of default unification, but there are other cases which have non-associative behaviour under both definitions. Consider an example lexical entry file:

```
A 1
   < F > = < G > .

B 1
   < F > = true
   < G > = true
   < > < A_1 < >.

C 1
```

< F > = **false**
< > < B_1 < >.

We have the following results for the value of C_1 (assuming a typing scheme such that F and G are appropriate features for all lexical entries with value **bool**):

$\begin{bmatrix} \text{F} = \boxed{1}\ \textbf{false} \\ \text{G} = \boxed{1} \end{bmatrix}$	LKB default unification, top-down	(both versions)
$\begin{bmatrix} \text{F} = \textbf{false} \\ \text{G} = \textbf{true} \end{bmatrix}$	LKB default unification, bottom-up	(old version)
$\perp$		(new version)
$\begin{bmatrix} \text{F} = \textbf{false} \\ \text{G} = \textbf{boolean} \end{bmatrix}$	Carpenter's skeptical default unification, top-down	
$\begin{bmatrix} \text{F} = \textbf{false} \\ \text{G} = \textbf{true} \end{bmatrix}$	Carpenter's skeptical default unification, bottom-up	

If we were to view default inheritance in terms of individual units of information being asserted at various points in an inheritance hierarchy, top-down inheritance can thus result in information which is asserted at a higher level being preferred over information asserted at the lower level.

Thus the main use of default inheritance in LAUREL is as a relationship between coherent parts of fully expanded lexical entries and from this the top-down behaviour follows. This is far more efficient than bottom-up inheritance would be, for this application, since the expanded psort can be cached. Again, in practice, top-down, bottom-up distinctions in behaviour arise with very low frequency.

As shown in section 2.4.2, all psorts from which information is to be inherited are unified before default unification takes place, thus removing any order dependency in the description language. The decision to restrict multiple default inheritance to the cases where the information inherited is consistent was determined by the use of semi-automatically acquired data. A fundamental point is that we cannot decide on an appropriate way of resolving conflicts in multiple inheritance without knowing what type of conflicts actually arise. Given that automatic extraction of information from MRDs is inevitably error prone, and that lexicographers' definitions are frequently not mutually consistent, we expected that most conflicts would be due to errors in the extraction process, or to inadequate definitions. Thus disallowing multiple inheritance conflicts is reasonable as an initial position. This at least gives the user the option of manually editing the lexical entries in order to get the desired behaviour, whereas any approach which did not signal the presence of conflicts would not.

3.4 Defaults and equality constraints

We may wish to override inherited information not by giving a conflicting value for the attribute, but by giving a value which is less specified or only partially conflicts. This is the motivation for the introduction of equality constraints.

As an illustrative example assume that substance_1 is a psort of type **physical**:

substance 1
< QUALIA > = **physical**
< QUALIA : FORM > = **mass**.

We can override the value for FORM for a more specific psort by specifying < QUALIA : FORM > = **individuated**, for example. However suppose we want to create the lexical entry for `cake_1` and that this should inherit information from `substance_1`. In this case we might want to override the information that `< QUALIA : FORM > =` **mass** without specifying a new value, in order to allow both for mass and count usages of *cake*. But if we specify

```
cake 1
   < QUALIA > < substance_1 < QUALIA >
   < QUALIA : FORM > = form.
```

default unification by any of the definitions suggested gives:

$$\texttt{cake\_1} = \begin{bmatrix} \textbf{lex-noun-sign} \\ \text{QUALIA} = \begin{bmatrix} \textbf{physical} \\ \text{FORM} = \textbf{mass} \end{bmatrix} \end{bmatrix}$$

Clearly we cannot simply claim that the lexical entry:

```
cake 1
   < QUALIA > < substance_1 < QUALIA >.
```

should give

$$\texttt{cake\_1} = \begin{bmatrix} \textbf{lex-noun-sign} \\ \text{QUALIA} = \begin{bmatrix} \textbf{physical} \\ \text{FORM} = \textbf{mass} \end{bmatrix} \end{bmatrix}$$

whereas

```
cake 1
   < QUALIA > < substance_1 < QUALIA >
   < QUALIA : FORM > = form.
```

should give

$$\texttt{cake\_1} = \begin{bmatrix} \textbf{lex-noun-sign} \\ \text{QUALIA} = \begin{bmatrix} \textbf{physical} \\ \text{FORM} = \textbf{form} \end{bmatrix} \end{bmatrix}$$

given that the type entry was

```
physical (nomqualia)
   < FORM > = form.
```

since the expansion of the non-default feature structure would result in the same thing in both cases.

A related problem arose when constructing entries automatically as part of the ACQUILEX project. For example consider the following definitions from the Italian Garzanti Dictionary:[5]

cacio latte di vacca, pecora o capra cagliato, salto e seccato in forma; formaggio

marzolino cacio fatto con latte di pecora o di bufala

Schematically the LKB entries can be represented as:

[5]My thanks to Elisabetta Marinai and Simonetta Montemagni for bringing these examples to my attention.

cacio

< SOURCE > = (**cow sheep goat**).

marzolino

< > < cacio < >

< SOURCE > = (**sheep buffalo**).

where disjunction is indicated by bracketing a list of types. But (default) unification results in SOURCE having a value **sheep** in `marzolino` whereas the desired result seems obviously to be (**sheep buffalo**).

Equality up to alphabetic variance of a path with the constraint on a type is represented by `=*` in the path notation. Thus we have:

cake 1

< QUALIA > < substance_1 < QUALIA >

< QUALIA : FORM > =* **form**.

which expands into the desired result:

$$\texttt{cake\_1} = \left[\begin{array}{l}\textbf{lex-noun-sign} \\ \text{QUALIA} = \left[\begin{array}{l}\textbf{physical} \\ \text{FORM} = \textbf{form}\end{array}\right]\end{array}\right]$$

Some other applications of equality constraints will be seen in subsequent chapters.

Chapter 4

Types and constraints

4.1 Well-formedness

The notion of well-formedness which was described in Chapter 2 is relatively weak. It is possible to have well-formed feature structures which cannot be extended to a structure which contains only most specific types. One disadvantage of this is that it is not possible, in general, to enforce cooccurrence restrictions, even of a quite limited sort. For example, suppose we have a type **t** with two features, F and G which both take boolean values, but we wish to state that when F has value **true**, G must have value **false** and vice versa. We might try to specify this by defining two subtypes of **t**, **t1** and **t2** with the following constraints:

$$\begin{bmatrix} \mathbf{t1} \\ \mathrm{F} = \mathbf{true} \\ \mathrm{G} = \mathbf{false} \end{bmatrix} \qquad \begin{bmatrix} \mathbf{t2} \\ \mathrm{F} = \mathbf{false} \\ \mathrm{G} = \mathbf{true} \end{bmatrix}$$

A type system in the LKB is assumed to be completely specified, in that, for example, two types can only be unified if they have a specified meet (greatest lower bound). Therefore we might assume that this fragment of a type system specifies the restriction that if F is **true** then G is **false**, and if F is **false** then G is **true**, since **t** has only the two subtypes **t1** and **t2**. However this is not the case. Consider the following structure:

$$\begin{bmatrix} \mathbf{t} \\ \mathrm{F} = \mathbf{true} \\ \mathrm{G} = \mathbf{true} \end{bmatrix}$$

This is still a well-formed feature structure, by the definition given in Chapter 2 despite the fact that it cannot be extended to be a well-formed structure with a type corresponding to that of any leaf node in the type hierarchy. Thus we have no good way of enforcing cooccurrence restrictions. Note that an equivalent result applies to Carpenter's appropriateness specifications (see Carpenter, 1992:144f).

To implement cooccurrence restrictions in this way we need a notion of typing which enforces the idea that the type system is complete. We can do this by checking that a feature structure can be extended to a well-formed structure where every type is a leaf type (cf the HPSG concept of a sort-resolved feature structure (Pollard and Sag, forthcoming)). We refer to such a feature structure as 'ultimately well-formed' and we can enforce such cooccurrence restrictions by checking for ultimate well-formedness.

Definition 37 *(Ultimate well-formedness) A feature structure F is said to be ultimately well-formed iff it subsumes some well-formed type-resolved feature structure F', where a* *feature structure $\langle Q, q_0, \delta, \alpha \rangle$ is type-resolved if the types of all its nodes are leaf types, that is for all nodes $q \in Q$ there is no type t such that $t \sqsubset \alpha(q)$*

However, in order to check for ultimate well-formedness, we have to consider not only the immediate subtypes of a type, but their subtypes, and so on, and recursively apply this process to every node in the feature structure.

The notion of a type-resolved feature structure has similarities with Carpenter's description of ground feature structures (Carpenter, 1992:144f). A ground feature structure according to Carpenter is one which may not be further specialised, that is there is no possible (well-formed) feature structure which is strictly subsumed by a ground feature structure. However, a type-resolved feature structure by the definition above is only fully specified with respect to path values; a more specific feature structure may be possible if reentrancy is introduced. Our definition is equivalent to Zajac's (1991, in press) definition of ground feature structure.

4.2 Constraint solving

If instead of simply checking whether a feature structure is ultimately well-formed by ensuring that it can be extended to at least one type-resolved feature structure, we enumerate all such type resolved structures, we have a notion of type inference which is, in effect, a a constraint resolution system. Constraint resolution is very powerful — it can be used, for example, to drive parsing and generation. Carpenter (1992:227f) and Zajac (1991, in press) have discussed how typed feature structures may be viewed as encoding constraint systems. The basic concept used in the LKB constraint system is similar, thus an LKB FS can be regarded as a description of a set of type-resolved or ground feature structures as defined above. Solving a constraint FS involves enumerating all the ground feature structures of which it is a description. Unlike Zajac and Carpenter, we assume, as before, that the constraint on a type is itself simply a feature structure.[1]

For example, assume we have the following type system:

top ().

list (**top**).

e-list (**list**).

ne-list (**list**)
`< HEAD >` = **thing**
`< TAIL >` = **list**.

thing (**top**).

a (**thing**).

b (**thing**).

[1]The LKB constraint resolution is broadly equivalent to Carpenter's description, on the assumption of "coverage", that is we assume that the type system is complete in that the subtypes of a type completely cover all the possibilities. Carpenter allows disjunctive constraints, but the effect of these can always be mimicked in our system by adding types, so there is no difference in power. Thus in the example in the previous section, type **t** is in effect associated with a disjunctive constraint on the valid feature structures such that either F is **true** and G is **false** or G is **true** and F is **false**. Zajac allows conditional constraints, which again make the system more expressive but do not (as far as I can tell) add power since their effect can be obtained by the use of "junk slots" (see below and also Carpenter, 1992:238f).

The FS specified as:

< TAIL : TAIL > = **e-list**.

can be solved in this type system to give the following ground FSs:

$$\begin{bmatrix} \textbf{ne-list} \\ \text{HEAD} = \textbf{a} \\ \text{TAIL} = \begin{bmatrix} \textbf{ne-list} \\ \text{HEAD} = \textbf{a} \\ \text{TAIL} = \textbf{e-list} \end{bmatrix} \end{bmatrix}$$
$$\begin{bmatrix} \textbf{ne-list} \\ \text{HEAD} = \textbf{a} \\ \text{TAIL} = \begin{bmatrix} \textbf{ne-list} \\ \text{HEAD} = \textbf{b} \\ \text{TAIL} = \textbf{e-list} \end{bmatrix} \end{bmatrix}$$
$$\begin{bmatrix} \textbf{ne-list} \\ \text{HEAD} = \textbf{b} \\ \text{TAIL} = \begin{bmatrix} \textbf{ne-list} \\ \text{HEAD} = \textbf{a} \\ \text{TAIL} = \textbf{e-list} \end{bmatrix} \end{bmatrix}$$
$$\begin{bmatrix} \textbf{ne-list} \\ \text{HEAD} = \textbf{b} \\ \text{TAIL} = \begin{bmatrix} \textbf{ne-list} \\ \text{HEAD} = \textbf{b} \\ \text{TAIL} = \textbf{e-list} \end{bmatrix} \end{bmatrix}$$

In contrast the FS with the description:

< HEAD > = **a**
< TAIL : HEAD > = **b**

has an infinite number of solutions, because the list can be extended indefinitely. Because of this possibility, constraint resolution is, in general, undecidable.

The algorithm used for type resolution in the LKB is quite straightforward (and not at all efficient!). We start at the initial node of the feature structure and attempt to "push down" its type to a leaf type. That is, for each subtype of the type, we check to see whether its constraint unifies with the given feature structure. If it does, we attempt to push down the unified structure. If we arrive at a leaf type, we add the new structure to the list of possibilities. If unification fails, we stop investigating that possibility. For example, when resolving:

$$\begin{bmatrix} \textbf{list} \\ \text{TAIL} = \begin{bmatrix} \textbf{top} \\ \text{TAIL} = \textbf{e-list} \end{bmatrix} \end{bmatrix}$$

the structure can only be a **ne-list**. [2] At the end of this phase, we will have a set of possibilities, with the initial node typed as a leaf type. In this case there is only one member of the set:

$$\begin{bmatrix} \textbf{ne-list} \\ \text{HEAD} = \textbf{thing} \\ \text{TAIL} = \begin{bmatrix} \textbf{top} \\ \text{TAIL} = \textbf{e-list} \end{bmatrix} \end{bmatrix}$$

[2] This is slightly misleading since the actual algorithm the LKB uses to do type inference involves making the feature structure well-formed according to the original definition first, since this is a relatively efficient process. However the end effect is the same.

For each of the possibilities, we attempt to push down all the nodes at the end of paths of length 1, effectively in parallel. In this case, we try to resolve **thing** and **top**. This results in the set of feature structures:

$$\left[\begin{array}{l}\textbf{ne-list}\\ \text{HEAD} = \textbf{a}\\ \text{TAIL} = \left[\begin{array}{l}\textbf{ne-list}\\ \text{HEAD} = \textbf{thing}\\ \text{TAIL} = \textbf{e-list}\end{array}\right]\end{array}\right]$$
$$\left[\begin{array}{l}\textbf{ne-list}\\ \text{HEAD} = \textbf{b}\\ \text{TAIL} = \left[\begin{array}{l}\textbf{ne-list}\\ \text{HEAD} = \textbf{thing}\\ \text{TAIL} = \textbf{e-list}\end{array}\right]\end{array}\right]$$

This process is then repeated with paths of length 2, and so on, until all the nodes in the feature structure are type resolved.

For a more complex example, assume that the following types are added to the system:

```
append (top)
   < LEFT > = list
   < RIGHT > = list
   < RESULT > = list.

append1 (append)
   < LEFT > = e-list
   < RIGHT > = < RESULT >.

append2 (append)
   < JUNK > = append
   < LEFT : HEAD > = < RESULT : HEAD >
   < LEFT : TAIL > = < JUNK : LEFT >
   < RIGHT > = < JUNK : RIGHT >
   < RESULT : TAIL > = < JUNK : RESULT >.
```

The idea is that the solutions to a goal feature structure of type **append** will be structures where the RESULT feature is the concatenation of the lists in the LEFT and RIGHT slots. This illustrates the use of "junk slots" which in effect allow for recursion. The type system encoding of append is equivalent to the Prolog code:

```
append([],L,L)
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3)
```

where `[X|L1]` is the Prolog version of the head/tail notation — that is X is the head of the list and L1 the tail. In both cases there are two possibilities for append — the first case is the end condition where the left list is empty, in which case the right hand list is equal to the result. This identity is implemented by reentrancy in typed feature structures and by the shared variable, L, in Prolog. The second case is where the left hand list is not empty, in which case append is satisfied if the head of the result is equal to the head of the left hand list given that the tail of the left hand list, the whole of the right hand list and the tail of the result themselves satisfy append. This recursion is implemented in Prolog by the conditional statement and in the LKB type system by the use of the extra "junk" slot, which itself takes a feature structure of type **append**. Just as in Prolog, **append** can be used not just in the 'conventional' direction to concatenate two lists, but also to enumerate the possible solutions that will give a particular result or to give the results for partially instantiated structures.

For example, the following structure specifies part of each of the lists:

$$\begin{bmatrix} \textbf{append} & \\ \text{LEFT} = \begin{bmatrix} \textbf{ne-list} \\ \text{HEAD} = \textbf{a} \\ \text{TAIL} = \textbf{list} \end{bmatrix} \\ \text{RIGHT} = \begin{bmatrix} \textbf{ne-list} \\ \text{HEAD} = \textbf{thing} \\ \text{TAIL} = \begin{bmatrix} \textbf{ne-list} \\ \text{HEAD} = \textbf{c} \\ \text{TAIL} = \textbf{list} \end{bmatrix} \end{bmatrix} \\ \text{RESULT} = \begin{bmatrix} \textbf{ne-list} \\ \text{HEAD} = \textbf{thing} \\ \text{TAIL} = \begin{bmatrix} \textbf{ne-list} \\ \text{HEAD} = \textbf{b} \\ \text{TAIL} = \begin{bmatrix} \textbf{ne-list} \\ \text{HEAD} = \textbf{thing} \\ \text{TAIL} = \textbf{e-list} \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

The result of giving this as a goal to the constraint system is the following:

$$\begin{bmatrix} \textbf{append2} \\ \text{LEFT} = \begin{bmatrix} \textbf{ne-list} \\ \text{HEAD} = \boxed{2}\ \textbf{a} \\ \text{TAIL} = \boxed{0}\ \textbf{e-list} \end{bmatrix} \\ \text{RIGHT} = \boxed{1} \begin{bmatrix} \textbf{ne-list} \\ \text{HEAD} = \textbf{b} \\ \text{TAIL} = \begin{bmatrix} \textbf{ne-list} \\ \text{HEAD} = \textbf{c} \\ \text{TAIL} = \textbf{e-list} \end{bmatrix} \end{bmatrix} \\ \text{RESULT} = \begin{bmatrix} \textbf{ne-list} \\ \text{HEAD} = \boxed{2} \\ \text{TAIL} = \boxed{1} \end{bmatrix} \\ \text{JUNK} = \begin{bmatrix} \textbf{append1} \\ \text{LEFT} = \boxed{0} \\ \text{RIGHT} = \boxed{1} \\ \text{RESULT} = \boxed{1} \end{bmatrix} \end{bmatrix}$$

Constraint resolution is sufficiently powerful to allow grammars encoded in the type system to be exploited for conventional parsing, generation from logical form or generation from any partially specified structures. In such a type system, instead of conventional grammar rules, there are constraints on phrases which partially specify their daughters. For a trivial example, instead of having a grammar rule

```
S -> NP VP
```

the type **s** could be defined as having an attribute DTRS which takes a value which is a list of feature structures of type **np** and **vp**.

$$\begin{bmatrix} \textbf{s} \\ \text{DTRS} = \begin{bmatrix} \textbf{ne-list} \\ \text{HEAD} = \textbf{np} \\ \text{TAIL} = \begin{bmatrix} \textbf{ne-list} \\ \text{HEAD} = \textbf{vp} \\ \text{TAIL} = \textbf{e-list} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

We will sometimes use angle bracket notation to indicate a list with elements separated by commas, for example:

$$\begin{bmatrix} \text{s} \\ \text{DTRS} = \langle \textbf{np}, \textbf{vp} \rangle \end{bmatrix}$$

Instead of the two rules:

```
VP -> Vtrans NP
```

and

```
VP -> Vintrans
```

we can define the type **vp** to have two subtypes **vp1** and **vp2** such that,

$$\begin{bmatrix} \textbf{vp1} & \\ \text{DTRS} = & \begin{bmatrix} \textbf{ne-list} & \\ \text{HEAD} = \textbf{vtrans} & \\ \text{TAIL} = & \begin{bmatrix} \textbf{ne-list} \\ \text{HEAD} = \textbf{np} \\ \text{TAIL} = \textbf{e-list} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

$$\begin{bmatrix} \textbf{vp2} & \\ \text{DTRS} = & \begin{bmatrix} \textbf{ne-list} \\ \text{HEAD} = \textbf{vintrans} \\ \text{TAIL} = \textbf{e-list} \end{bmatrix} \end{bmatrix}$$

If we now set **john** and **mary** to be subtypes of **np**, **sleeps** to be a subtype of **vintrans** and **likes** to be a subtype of **vtrans**, the feature structure:

$$\begin{bmatrix} \text{s} \end{bmatrix}$$

will be resolved to generate structures (almost) isomorphic to the tree structures produced by the grammar rules:

$$\begin{bmatrix} \text{s} \\ \text{DTRS} = \langle \textbf{john}, \begin{bmatrix} \textbf{vp} \\ \text{DTRS} = \langle \textbf{sleeps} \rangle \end{bmatrix} \rangle \end{bmatrix}$$

$$\begin{bmatrix} \text{s} \\ \text{DTRS} = \langle \textbf{mary}, \begin{bmatrix} \textbf{vp} \\ \text{DTRS} = \langle \textbf{sleeps} \rangle \end{bmatrix} \rangle \end{bmatrix}$$

$$\begin{bmatrix} \text{s} \\ \text{DTRS} = \langle \textbf{john}, \begin{bmatrix} \textbf{vp} \\ \text{DTRS} = \langle \textbf{likes}, \textbf{mary} \rangle \end{bmatrix} \rangle \end{bmatrix} \quad \ldots$$

Naturally a much more complex series of constraints must be used to encode a realistic grammar as is done in HSPG (Pollard and Sag, 1987, forthcoming). In the following examples we will use a type system which is similar to that found in Pollard and Sag (forthcoming), although considerably simplified.[3] The structure shown in Figure 4.1 is that assigned to the sentence *john likes mary* by resolution using this type system. Note that the arguments of the lexical signs have been co-indexed. This structure can be produced by constraint resolving either of the structures shown in Figure 4.2, the first of which has only the orthographies of the sentence sign instantiated (parsing) and the second only the logical form (generation).

In Figure 4.1, **s-sign** is a specialisation of the type **idp1** which implements the HPSG immediate dominance principle schema 1. **idp2** corresponds to schema 2. We have omitted

[3]The existing ACQUILEX type system could be converted into a constraint based system in a way which is quite straightforward in principle: instead of the existing three types of grammar rules there would be three types of phrasal sign, which explicitly incorporated the daughters of the rule. In practice, some less straightforward modifications would have to be made, since not all signs allowed by the grammar in the ACQUILEX system can be extended to ground feature structures.

$$
\left[\begin{array}{lll}
\textbf{s-sign} & & \\
\text{ORTH} & = & < \textbf{john, likes, mary} > \\
\text{LANG} & = & \textbf{english} \\
\text{SYNSEM} & = & \left[\begin{array}{lll}
\textbf{s-synsem} & & \\
\text{CAT} & = & \boxed{\textbf{scat}} \\
\text{SEM} & = & \boxed{8} \left[\begin{array}{lll}
\textbf{tvsem} & & \\
\text{PRED} & = & \textbf{and} \\
\text{ARG1} & = & \left[\begin{array}{lll}
\textbf{tvformula} & & \\
\text{PRED} & = & \textbf{like\_1} \\
\text{ARG1} & = & \boxed{9}\ \textbf{obj} \\
\text{ARG2} & = & \boxed{10}\ \textbf{obj}
\end{array}\right] \\
\text{ARG2} & = & \left[\begin{array}{lll}
\textbf{compl-formula} & & \\
\text{PRED} & = & \textbf{and} \\
\text{ARG1} & = & \boxed{11} \left[\begin{array}{lll}
\textbf{npsem} & & \\
\text{PRED} & = & \textbf{john\_1} \\
\text{ARG1} & = & \boxed{9}
\end{array}\right] \\
\text{ARG2} & = & \boxed{12} \left[\begin{array}{lll}
\textbf{npsem} & & \\
\text{PRED} & = & \textbf{mary\_1} \\
\text{ARG1} & = & \boxed{10}
\end{array}\right]
\end{array}\right]
\end{array}\right]
\end{array}\right] \\
\text{DTRS} & = & \left[\begin{array}{lll}
\textbf{head-comp-struc} & & \\
\text{HEAD-DTR} & = & \left[\begin{array}{lll}
\textbf{idp2} & & \\
\text{SYNSEM} & = & \boxed{\textbf{vp-synsem}} \\
\text{DTRS} & = & \left[\begin{array}{lll}
\textbf{head-comp-struc} & & \\
\text{HEAD-DTR} & = & \left[\begin{array}{lll}
\textbf{tverb-sign} & & \\
\text{ORTH} & = & < \textbf{likes} > \\
\text{SYNSEM} & = & \left[\begin{array}{lll}
\textbf{tv-synsem} & & \\
\text{CAT} & = & \boxed{\textbf{tvcat}} \\
\text{SEM} & = & \boxed{8}
\end{array}\right]
\end{array}\right] \\
\text{C/DTRS} & = & \left[\begin{array}{lll}
\textbf{ne-dtrs-list} & & \\
\text{HD-DTR} & = & \left[\begin{array}{lll}
\textbf{lex-np-sign} & & \\
\text{ORTH} & = & < \textbf{mary} > \\
\text{SYNSEM} & = & \left[\begin{array}{lll}
\textbf{np-synsem} & & \\
\text{CAT} & = & \boxed{\textbf{npcat}} \\
\text{SEM} & = & \boxed{12}
\end{array}\right]
\end{array}\right] \\
\text{TL-DTR} & = & \boxed{\textbf{e-dtrs-list}}
\end{array}\right]
\end{array}\right]
\end{array}\right] \\
\text{C/DTRS} & = & \left[\begin{array}{lll}
\textbf{ne-dtrs-list} & & \\
\text{HD-DTR} & = & \left[\begin{array}{lll}
\textbf{lex-np-sign} & & \\
\text{ORTH} & = & < \textbf{john} > \\
\text{SYNSEM} & = & \left[\begin{array}{lll}
\textbf{np-synsem} & & \\
\text{CAT} & = & \boxed{\textbf{npcat}} \\
\text{SEM} & = & \boxed{11}
\end{array}\right]
\end{array}\right] \\
\text{TL-DTR} & = & \boxed{\textbf{e-dtrs-list}}
\end{array}\right]
\end{array}\right]
\end{array}\right]
$$

Figure 4.1: Structure for parse of “John likes Mary”. (C/DTRS is used as an abbreviation for COMP-DTRS).

$$\left[\begin{array}{l}\textbf{s-sign}\\ \text{ORTH} = \langle \textbf{john, likes, mary} \rangle\\ \text{LANG} = \textbf{english}\end{array}\right]$$

$$\left[\begin{array}{l}\textbf{s-sign}\\ \text{LANG} = \textbf{english}\\ \text{SYNSEM} = \left[\begin{array}{l}\textbf{s-synsem}\\ \text{SEM} = \left[\begin{array}{l}\textbf{sem}\\ \text{PRED} = \textbf{and}\\ \text{ARG1} = \left[\begin{array}{l}\textbf{formula}\\ \text{PRED} = \textbf{like\_1}\\ \text{ARG1} = \boxed{9}\ \textbf{obj}\\ \text{ARG2} = \boxed{10}\ \textbf{obj}\end{array}\right]\\ \text{ARG2} = \left[\begin{array}{l}\textbf{formula}\\ \text{PRED} = \textbf{and}\\ \text{ARG1} = \left[\begin{array}{l}\textbf{sem}\\ \text{PRED} = \textbf{john\_1}\\ \text{ARG1} = \boxed{9}\end{array}\right]\\ \text{ARG2} = \left[\begin{array}{l}\textbf{sem}\\ \text{PRED} = \textbf{mary\_1}\\ \text{ARG1} = \boxed{10}\end{array}\right]\end{array}\right]\end{array}\right]\end{array}\right]\end{array}\right]$$

Figure 4.2: Goal structures which can be resolved to parse and generate from logical form

several features from the figure. In particular the type **head-comp-struc** actually has features which take as values a list of all the orthographies and synsems of the daughter structures, and the same is true of the types **ne-dtrs-list**. All structures also have a feature which specifies the language, in this case English, which is necessary when translating using a single type system, to ensure that only types and lexical entries which are valid for that language are used. In all cases the language of all the daughters of a phrase must be identical and equal to that of the phrase itself, but we have left out the feature here for simplicity. We have omitted the orthographies of the phrasal signs except for that of the sentence sign itself. We have not shown the syntactic part of the signs in any detail, but both types of phrasal sign obey the subcategorisation principle which states that the SUBCAT value of the HEAD-DTR is the concatenation (append) of the phrases SUBCAT list with the list of the SYNSEM values of the COMP-DTRS. They also obey the head feature principle, that the HEAD of the phrase is the HEAD of the HEAD-DTR. For further details, see Pollard and Sag (forthcoming).

The simplest treatment of lexical entries is to make them all distinct leaf types and to specify all their orthographies as distinct leaf types. This is cumbersome, so we have assumed that the type **string** covers arbitrary (Lisp) strings as usual in the LKB and that lexical signs are "pseudo-types". The system makes no attempt to resolve a node of type **string**. Pseudo-types can be regarded as (unnameable) types which cannot introduce new features and cannot have multiple parents. Thus lexical signs behave with respect to constraint resolution as though they are subtypes of types such as **tverb-sign**, but the set of lexical signs can be varied without redefining the type system.

Note that in the LKB system there is no separate linear precedence component. The phrasal types are defined so that their orthography is the result of appending the orthographies of their daughters in an order specified by the sign. Note also that lexical rules in this system have to be regarded in the conventional way as expanding the lexicon in a preliminary processing phase. This is discussed further in Section 6.1. The main reason for investigating the use of a constraint system in the LKB is its naturalness as a way of implementing a testbed MT system. This is the subject of the next chapter.

Chapter 5

Multilingual lexical representation

It should be emphasised that the intention of this work is to concentrate on the linguistic aspects of multilingual representation and the problems of the lexicon in particular, rather than MT as such. MT is not the only possible application for a multilingual LKB — ultimately such an object should be able to support the needs of linguists, lexicographers and others who need access to substantial quantities of cross-linguistic information. Therefore here we concentrate on whether the lexicon contains adequate syntactic and semantic information for MT, leaving open issues such as how to support disambiguation, pragmatically appropriate lexical choice and so on which are crucial to real MT but depend heavily on general and domain-specific real world knowledge.[1] For example, it would clearly be problematic for real applications if when translating (1) into German, *Ufer* rather than *Bank* was used, but giving both options would be the correct behaviour for the LKB system.

(1) John went to the bank and paid in the cheque for £36.49.

We take it as a goal that our encoding of multilingual lexical information should support approaches which are declarative and bidirectional and that, as far as possible, we should avoid encoding information which is specific to particular techniques for MT. Thus we have no interest in producing a practical MT system within the ACQUILEX project, but we clearly have to support construction of the multilingual lexicon and show that it contains sufficient information to allow MT, and the best way of achieving this is to build a test-bed MT system.

5.1 MT in unification based formalisms

Like parsing and generation, MT can be regarded very generally as a process of constraint solving. The target language (TL) sentence generated is constrained by the monolingual grammar and lexicon and also by constraints derived from the parse of the source language (SL) sentence. Different approaches to MT can be characterised according to what sort of

[1] Nothing we shall say is incompatible with the assumption that some knowledge needed for such processing is associated with the lexicon. There is clearly an issue of how to integrate the mechanism with the further processing needed and this is briefly addressed in Section 5.7.

cross-linguistic constraints are imposed, in addition to the monolingual constraints. Here we will briefly discuss interlingua and then consider some transfer based approaches.

The archetypal use of an interlingua corresponds to the situation where the semantic structure produced by the parse of the SL sentence is identical to that which would be produced by the parse of the TL sentence (see Emele *et al* 1992, for an MT system which uses constraint resolution to produce an interlingual representation). The difficulty with designing interlingual representations which can be used in this way is to ensure that they can be generated by the grammars of the individual languages, without the grammars becoming too unnatural or the job of the grammar writer unreasonably hard. One problem is that structures which may be intended to be interpreted as having identical meanings may not be identical with respect to the grammar in that they will not generate identical strings (see Shieber, 1993, for the problem of logical form equivalence in generation). To take a simple example, the structures

$$\begin{bmatrix} \textbf{formula} \\ \text{PRED} = \textbf{and} \\ \text{ARG1} = \begin{bmatrix} \textbf{formula} \\ \text{PRED} = \textbf{young} \\ \text{ARG1} = \boxed{1}\ \textbf{x} \end{bmatrix} \\ \text{ARG2} = \begin{bmatrix} \textbf{formula} \\ \text{PRED} = \textbf{man} \\ \text{ARG1} = \textbf{x} \end{bmatrix} \end{bmatrix} \quad \begin{bmatrix} \textbf{formula} \\ \text{PRED} = \textbf{and} \\ \text{ARG1} = \begin{bmatrix} \textbf{formula} \\ \text{PRED} = \textbf{man} \\ \text{ARG1} = \boxed{1}\ \textbf{x} \end{bmatrix} \\ \text{ARG2} = \begin{bmatrix} \textbf{formula} \\ \text{PRED} = \textbf{young} \\ \text{ARG1} = \textbf{x} \end{bmatrix} \end{bmatrix}$$

might be interpreted as $man(x) \wedge young(x)$ and $young(x) \wedge man(x)$ and as therefore being equivalent but this does not mean that they will generate identical strings in a grammar or indeed that a grammar will accept both. Emele *et al*'s use of a constraint-based formalism reduces the problem to some extent, because it makes it possible to write grammars where the semantic structure does not have to mirror the constituent structure, but it still leaves open the problem of ensuring that each grammar will generate compatible interlingua. It is difficult to see how this can be done in the absence of a solution to the AI-complete meaning representation problem unless the grammar writer has detailed knowledge of at least two languages and writes or tunes the grammars so that the representations fit each other.

Transfer based approaches, in contrast, do not assume identity between any part of the SL structure and TL structure and this can allow the grammars for the different languages to be developed independently. However transfer systems differ with respect to the sort of correspondences which can be described. Just as grammar rules expressed in a unification-based formalism can be regarded non-procedurally as constraints on well-formed monolingual structures, transfer rules or translation links can be regarded as constraints on the mappings between source and target structures which are translation equivalent. These mappings may be stated in terms of orthography, syntax, semantics or a combination of these components, and may be described at a lexical, phrasal or sentential level. To some extent, the choice depends on the monolingual grammar framework used — for example, within sign-based lexicalist approaches such as HPSG, it is natural to think of transfer in terms of signs, and to concentrate on lexical transfer.

Thus we can classify MT systems in terms of such a model according to the sort of translation constraints they assume, and how they control the process of translation. For example in SRI's BCI system (Alshawi *et al.*, 1991) the source language string is parsed and transfer is carried out on the (quasi-)logical form representation. This produces a quasi-logical form appropriate for the TL, which can be used to drive a head-driven generator. In contrast, the Shake-and-Bake approach (Whitelock, 1992) relies on lexical transfer operating on lexical signs which have had their variables instantiated as a result of parsing. As in the BCI system, transfer operates before generation, but in Shake-and-Bake generation is constrained by 'bags' of instantiated lexical entries rather than an ordered representation of the sentence as a whole. This has the advantage that there is no problem of potential mismatch between the monolingual grammars. However it also means that normal generation techniques are not applicable; the algorithm actually used roughly involves producing all possible orderings of transferred target language signs and using standard parsing techniques to determine which of these orderings meet the constraints of the target language grammar (which means that the number of sentences which must be parsed is factorial in the length of the SL sentence). The ELU system (Estival *et al.*, 1990) demonstrates a third approach; transfer operates at both a lexical and a phrasal level, and transfer and generation are interwoven by making use of transfer variables which indicate explicitly that parts of the structure remain untranslated.

Transfer based approaches may put a heavy burden on the writers of the monolingual grammars, if the structures which are stated to be translation equivalent have to have some degree of isomorphism. It is also desirable to avoid the situation where detailed knowledge of the monolingual structures is required in order to write transfer rules. The particular approach to translation which we are exploring attempts to avoid these pitfalls. It is heavily lexicalist in orientation, in that constraints are described between signs in the SL and TL structures, but it is also possible to incorporate non-lexical cross-linguistic constraints.

We define lexical translation equivalence in terms of cross-linguistic links, *tlinks*, stated in terms of lexical entries in the monolingual lexicons. Because tlinks are defined in terms of inheritance from lexical entries and rules, translation equivalence can be stated through direct reference to properties of word syntax and semantics as they appear in the monolingual lexical descriptions. Tlinks can be viewed as information structures describing ways in which lexical entries from the source and target languages are mapped into translation-equivalent pairs. The simplest cases of tlinks are where the lexical entries themselves can be considered translation equivalent but the mechanism allows translation equivalence to be expressed between lexical entries which have been in some way modified or grouped together in a phrase. A fully instantiated tlink can be viewed as stating that a particular set of ground FSs in the SL are translation equivalent to a set of ground FSs in the TL. Thus they can be seen as acting as constraints on valid SL/TL translation equivalent pairings. As in Shake-and-Bake, the translation process is driven by stating that the variables in the semantic structures in the source and target are identical.

The LKB testbed system applies tlinks in a way which is reminiscent of Shake-and-Bake. The source language sentence is parsed, which results in instantiation of the variables in the lexical signs which make up part of the parsed structure. Tlinks are applied to appropriate subparts of the structure. This results in a "bag" of partially instantiated feature structures in the target language. Generation then proceeds in a similar way as before, except that the lexical entries available to act as pseudo-types are restricted to those in the generation bag and (unlike in Shake-and-Bake) phrasal constraints may also be expressed. Furthermore, we allow correspondences to be stated in terms of feature structures which can be resolved to one or more lexical entries, rather than insisting that the actual lexical entry is known in all cases. This allows us to exploit a notion of a limited interlingua, as will become apparent below. In the rest of this chapter, we give a detailed description of tlinks and of the process of translation in the LKB.

$$
\begin{tikzcd}
\mathrm{SFS1} \arrow[r, "\text{source mapping}"] & \mathrm{SFS0} \arrow[r, leftrightarrow, "\textbf{tlink}"] & \mathrm{TFS0} & \mathrm{TFS1} \arrow[l, "\text{target mapping}"'] \\
\text{source LE} & \text{source TE} & \text{target TE} & \text{target LE}
\end{tikzcd}
$$

Figure 5.1: Figurative representation of a word-word translation link.

$$
\begin{tikzcd}
\mathrm{SFS1} \arrow[r, "\text{identity}"] & \mathrm{SFS0} \arrow[r, leftrightarrow, "\textbf{tlink}"] & \mathrm{TFS0} & \mathrm{TFS1} \arrow[l, "\text{identity}"'] \\
\text{fly\_1} & \text{fly\_1} & \text{volare\_1} & \text{volare\_1}
\end{tikzcd}
$$

Figure 5.2: Tlink between *fly* (intransitive) and *volare*.

5.2 Tlinks

Tlinks can be viewed as information structures describing ways in which input lexical entries (LE) from the source and target languages are mapped into output translation-equivalent (TE) pairs. Most tlinks are of the word to word type, which can be represented diagrammatically as in Figure 5.1. In these cases, one lexical item in the source language is related to one lexical item in the target language, although the FSs which are actually stated to be translation equivalent may be specialisations of the lexical entry, or be derived from the entry by rule application.[2] For example, the English lexical entry for (intransitive) *fly* is translation equivalent to the Italian *volare* (see Figure 5.2). In this case the source and target mappings are identity, since the base lexical entries themselves can be stated to be translation equivalent.

Somewhat rarer and more complex cases of linking arise when a translation mismatch involves transformations operating on FSs. For example, the equivalence class resulting from translation pairs such as 'English:*furniture* ≈ Spanish:*muebles*' can be represented by establishing a link between a word and its translation in the plural form. In this case the equivalence holds between a basic lexical entry and a lexical entry after rule application. Here the mapping on the English side is identity again, but the Spanish mapping is the morphological rule for plural formation, as shown in Figure 5.3.

In more complicated cases, more than one lexical item may be involved on one or both sides of the tlink. A word-to-phrase tlink is shown diagrammatically in Figure 5.4 (phrase-to-phrase or phrase-to-word tlinks are also possible). Here the constraint on translation equivalence is stated partially at the phrasal level and partly in terms of the lexical structures that the phrase contains. For example, the Spanish-English VOX dictionary gives *novillo* the translation *young bull*. Diagrammatically we can represent this as Figure 5.5.

[2] Tlinks are both symmetrical and reversible; we use the terminology source and target for ease of exposition.

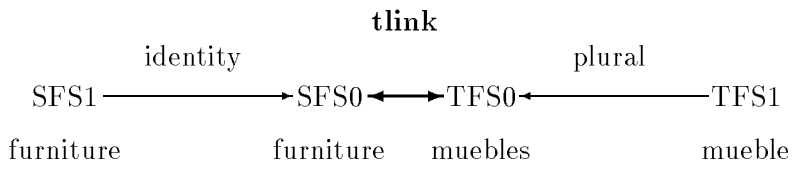


Figure 5.3: Tlink relating *furniture* and *muebles*

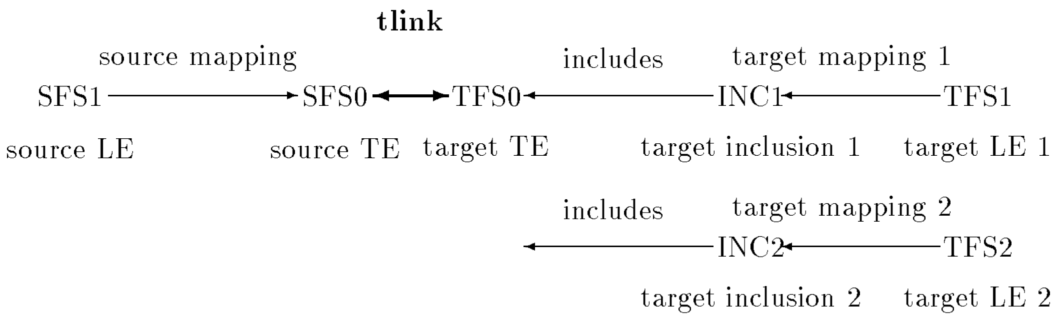


Figure 5.4: Figurative representation of a word-phrase translation link.

Note that this is less restrictive than stating the constraint as a simple FS, where the components were specified precisely. This tlink allows *novillo ingles* to be translated as *young English bull*, for example, where the inserted adjective is allowed because the specification is simply that the phrase includes *young* and *bull*, not that it consists of exactly those two lexical items.

More formally, tlinks are defined as FSs of type **tlink** which relates the source FS (SFS) and the target FS (TFS) which are both of type **rule**:

tlink (**top**)
    `< SFS >` = **rule**
    `< TFS >` = **rule**.

In the simplest case, a rule establishes a correspondence between an input sign (1) and an output sign (0) in the same language:

rule (**top**)
    `< 0 >` = **sign**
    `< 1 >` = **sign**
    `< 0 : LANG > = < 1 : LANG > .`

unary-rule (**rule**) .

The rule-inputs of tlinks are meant to be instantiated by FS representations of word senses in the source and target languages; rule-outputs provide the TE pairs. Thus, when the FSs at the end of the paths `< SFS : 1 >` and `< TFS : 1 >` are instantiated by lexical

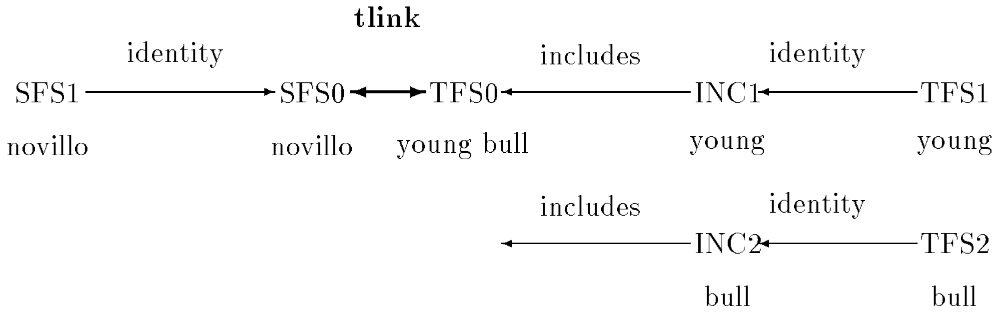


Figure 5.5: Tlink relating *novillo* and *young bull*

entries, the FSs at the end of the paths `< SFS : 0 >` and `< TFS : 0 >` are defined to be translation equivalent.

Tlinks can be regarded statically as defining constraints on translation equivalent pairings or dynamically as generating new FSs; given a FS in one language, and an appropriate tlink, unification with the FS at the end of the appropriate path, the *SL output*, (e.g. `< SFS : 0 >`) in the tlink, will result in the FS at the end of the other output path (the *TL output*) being returned (e.g. `< TFS : 0 >`).

The concept of translation equivalence is constrained by defining an inheritance network of tlink types encoding generalisations relative to classes of crosslinguistic equivalences. The commonest and simplest cases of noun translation equivalence can be represented as **simple-tlink**s.

simple-tlink (**tlink**)
```
< SFS : 0 > = < SFS : 1 >
< TFS : 0 > = < TFS : 1 >
< SFS : 0 : SYNSEM : SEM : ARG1 > =
    < TFS : 0 : SYNSEM : SEM : ARG1 >.
```

A **simple-tlink** is applicable in the case where two nouns are straightforwardly translation equivalent, without any transformation being necessary. The semantic variables relative to the semantic entity described by two output structures are specified to be identical. This variable equivalence is the basis for the use of tlinks within unification based approaches to MT and is crucial in expressing translation mismatches such as thematic divergence and head switching (Sanfilippo *et al.*, 1992). The particular paths equated will depend on the way that the type system is used to encode semantic structure — in the examples which follow a simplified encoding has been used for ease of exposition.

For example, assuming that the English word sense *chocolate* 4, is translation equivalent to the Dutch *chocolade* 2, we would have the tlink:

simple-tlink
```
< SFS : 1 > <= chocolate_4 <>
< TFS : 1 > <= chocolade_2 <>.
```

where `<=`, indicates non-default inheritance from a named FS (the particular lexical entries). This tlink will yield the English entry as output when the Dutch entry is supplied as input and vice versa. In order to make the description more readable we use a notation of the following form:

```
psort / psort :
   description .
```

where the psort on the LHS of the slash is taken as the value for the path `< SFS : 1 >` and the psort on the RHS of the slash is taken as the value for the path `< TFS : 1 >` and `description` is a normal FS description. Thus the tlink above is equivalent to:

```
chocolate_4 / chocolade_2 :
   simple-tlink.
```

For intransitive verbs we make use of a slightly different tlink type:

iverb-tlink (**tlink**)
```
   < SFS : 0 > = < SFS : 1 >
   < TFS : 0 > = < TFS : 1 >
   < SFS : 1 > = iverb-sign
   < TFS : 1 > = iverb-sign
   < SFS : 0 : SEM : ARG1 : ARG1 > =
       < TFS : 0 : SEM : ARG1 : ARG1 >.
```

For example the tlink for 'English:*fly* ≈ Italian:*volare*' would be:

```
fly_1 / volare_1 :
   iverb-tlink.
```

Some restrictions on translation can be expressed by making the target or source FSs of tlinks more specific. For example, we can define a type **human-tlink** and state as a constraint that the values for the SEX feature must be the same in the translation equivalent feature structures:

human-tlink (**simple-tlink**)
```
   < SFS : 0 : QUALIA : SEX > =
       < TFS : 0 : QUALIA : SEX >.
```

A tlink of this type would be suitable for establishing equivalences such as that between the word *teacher* in English and its translation in Spanish as either *maestro* or *maestra*. The restriction that *maestro* denotes a male teacher and *maestra* a female one — i.e. the values for `< QUALIA : SEX >` in the FS descriptions for the two words are **male** and **female** respectively — follows from the use of tlinks of type **human-tlink** to relate *teacher* with *maestro* and *maestra*:

```
teacher / maestro :
human-tlink.
```

```
teacher / maestra :
human-tlink.
```

The path equation induced by the **human-tlink** will ensure that translation equivalent words denote individuals of the same (natural) gender.[3] The use of a common type system

[3] As we will see later, the morphological generalisation involved in this example can be captured, making it unnecessary to manually specify both tlinks.

makes it easier to express such relationships, although it is not essential, provided that comparable information is encoded in both monolingual representations.

In the case of examples such as 'English:*furniture* ≈ Spanish:*muebles*' we instantiate one half of the tlink with the appropriate lexical rule. This is shown in the tlink below where the target-language side (`< TFS >`) inherits from the FS representation for the plural rule whose input is the tlink's target input (`mueble`) and output is the tlink's target output.

```
furniture / mueble :
tlink
< SFS : 0 > = < SFS : 1 >
< TFS > <= plural <>.
```

Note that the lexical/morphological rule for plural formation used in the tlink is needed anyway for use in parsing/generation, so that its use in tlinks does not involve introduction of elements other than those needed in the monolingual grammars. Since the singular form *mueble* would not unify with the feature structure at the end of the output path `< TFS : 0 >`, a translation of *mueble* as *furniture* would not be generated and another tlink would be required to relate the two words when translating a phrase such as *a piece of furniture*.

As indicated by Figure 5.4 above, phrasal tlinks in general have a more complex structure. The translation equivalent structure is not a simple FS, but consists of a single FS corresponding to the phrase as a whole, plus one or more FSs describing lexical items which are components of the phrase. In order to allow the component lexical items to be non-basic forms, we allow for a mapping between the input signs and the included signs. To describe such tlinks we use a more complex type of **rule** on the phrasal side(s) of the tlink. The type constraint appropriate for the case where two lexical signs are described as being included in a phrase is shown below:

$$
\begin{bmatrix}
\textbf{binary-inclusion} & \\
0 = \begin{bmatrix} \textbf{sign} \\ \text{LANG} = \boxed{1} \end{bmatrix} & \\
1 = \boxed{0}\begin{bmatrix} \textbf{sign} \\ \text{LANG} = \boxed{1}\ \textbf{lang} \end{bmatrix} & \\
2 = \boxed{2}\ \textbf{sign} & \\
\text{INC1} = \begin{bmatrix} \textbf{unary-rule} \\ 1 = \boxed{0} \\ 0 = \begin{bmatrix} \textbf{sign} \\ \text{LANG} = \boxed{1} \end{bmatrix} \end{bmatrix} & \\
\text{INC2} = \begin{bmatrix} \textbf{unary-rule} \\ 1 = \boxed{2} \\ 0 = \begin{bmatrix} \textbf{sign} \\ \text{LANG} = \boxed{1} \end{bmatrix} \end{bmatrix} &
\end{bmatrix}
$$

The description of the phrase as a whole is given by `< 0 >`, the input lexical signs are the values of `< 1 >` and `< 2 >` (which are the same as the values of `< INC1 : 1 >` and `< INC2 : 1 >`) and the signs included in the phrase are found as the values of `< INC1 : 0 >` and `< INC2 : 0 >`. As before, the input lexical signs may be transformed by lexical rule application, but in the simple cases the mapping is identity and in such cases `< INC1 : 1 >` will be identical to `< INC1 : 0 >` and `< INC2 : 1 >` will be identical to `< INC2 : 0 >`. For inclusion of a single item 2 and INC2 will not be present, for three items we add 3 and INC3 and so on. There will normally be further constraints on how the included items interrelate.

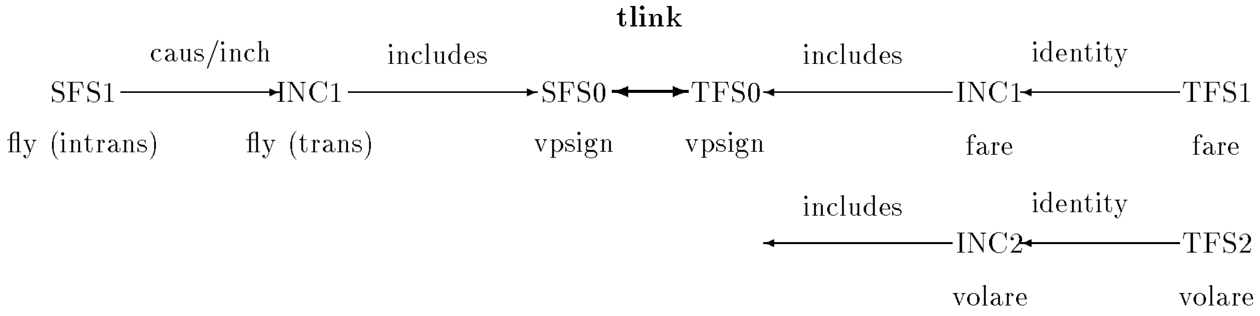


Figure 5.6: Tlink relating transitive *fly* to *fare volare*.

For example, consider the case of the transitive use of motion verbs such as *fly* in English, which has to be translated in Italian with an explicit causative using *fare* (see Copestake and Sanfilippo, 1993):

(2) Jamie *flew* his kite
Jamie *fece volare* l'acquilone

Diagrammatically, a tlink which could be used to express this is shown in Figure 5.6 (there are other possibilities). The path description is partially shown below:

```
< SFS : 1 > <= fly_1 <>
< TFS : 1 > <= fare_1 <>
< TFS : 2 > <= volare_1 <>
< SFS : 0 > = idp2
< SFS : INC1 > <= causative-inchoative-alternation <>
< SFS : 0 : SYNSEM : SEM > =
       < SFS : INC1 : 0 : SYNSEM : SEM >
< TFS : 0 > = idp2
< TFS : INC1 > = identity
< TFS : INC2 > = identity
< TFS : 0 : SYNSEM : SEM > =
       < TFS : INC1 : 0 : SYNSEM : SEM >.
```

Again, the slashed notation is simpler:

```
fly_1 / fare_1 volare_1 :
   < SFS : 0 > = idp2
   < SFS : INC1 > <= causative-inchoative-alternation <>
   < SFS : 0 : SYNSEM : SEM > =
          < SFS : INC1 : 0 : SYNSEM : SEM >
   < TFS : 0 > = idp2
   < TFS : INC1 > = identity
   < TFS : INC2 > = identity
   < TFS : 0 : SYNSEM : SEM > =
          < TFS : INC1 : 0 : SYNSEM : SEM >.
```

In this example, the English phrase is stated to be an **idp2**, with a semantic structure equal to that of the included lexical sign, which is the transitive version of *fly*. The Italian phrase is also a **idp2** including *fare* and *volare*, suitably constrained. Here we have not shown the path equality statements for the semantics, since this depends on the encoding and a full discussion of the possibilities is out of place here. One simple possibility would be for the English semantics to be equivalent to

$$[CAUSE(e, x, e') \wedge fly(e', y)]$$

(where the intransitive is $fly(e', y)$) and the Italian to be

$$[fare(e, x, e') \wedge volare(e', y)]$$

but more complex encodings are possible. The two semantic structures need not be isomorphic as they are here, but it has to be possible to relate the variables. In particular, the event variables have to be directly related to deal correctly with adverbs and with cases where the structure is embedded.

In fact this translation pattern generalises over a number of lexical items, such as *march*, *run* and so on, so it makes sense to define a type of tlink which captures the generalisations, which would leave us with a much simpler bilingual entry.[4] In Copestake and Sanfilippo (1993) we discussed the use of a "partial interlingua" using thematic roles which allows us to state the general translation pattern in a more linguistically motivated fashion.

In general, the set of ground FSs described by a phrasal tlink is infinite. In this example, the phrases *fly the kite*, *fly the big red kite*, *fly the aeroplane* and so on, will all match the description of the English phrase. However, these structures will not be generated unless licenced by other parts of the input. Thus *kite* will not be valid unless *acquilone* appears in the Italian sentence and there is a tlink that links the two. The conditions on translation are discussed in detail in section 5.5.

5.3 Tlink rules

In some cases the existence of a tlink between two lexical items implies a further translation relationship. For example, in English, there is a regular sense extension such that a word used primarily to denote an animal can also be used to denote the meat of that animal (e.g. *lamb*, *chicken*, see Copestake and Briscoe (1992)). A similar sense extension rule applies to Italian (Östling 1991) but in Dutch a compound is generally used (*lam*, *lamsvlees*). The correspondence between the two processes (e.g. sense extension in English and compounding in Dutch) and their import on translation equivalence can be expressed by means of a *tlink-rule* which allows transformations to be defined for tlinks. Tlink rules are analogous to lexical rules, that is they can be regarded as generating new tlinks by operating on existing ones. They are treated as applying in a phase which is preliminary to translation proper in order to expand the set of defined tlinks.

For example, we can automatically generate the relationship between the FS representing the animal sense of *lamb* (`lamb_1`) and `lamsvlees` by transforming the simple tlink between the FS representing the meat sense of *lamb* (`lamb_2`) and `lam` into a new tlink

[4] As we will see below, this tlink could actually be derived automatically from the simpler tlink between intransitive *fly* and *volare*.

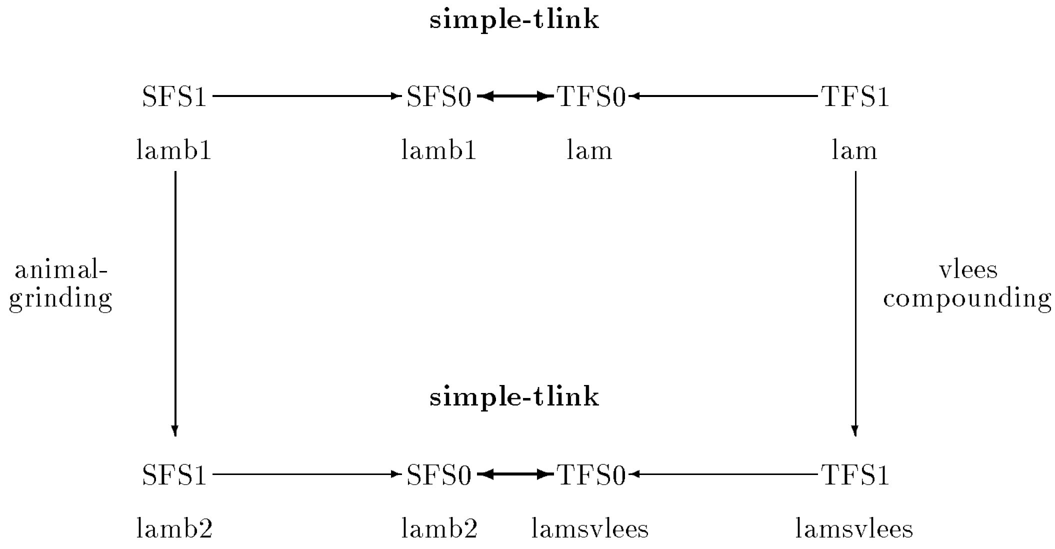


Figure 5.7: Tlink rule relating animal and meat senses of *lamb* to *lam* and *lamsvlees*

where the source input FS results from applying the sense extension rule 'animal-grinding' to `lamb_1` and the source input FS results from compounding *lam* with *vlees*, as shown diagrammatically in Figure 5.7.

Again tlink-rules are themselves FSs. The type tlink-rule is defined as follows:

tlink-rule (**top**)
`< t0 >` = **tlink**
`< t1 >` = **tlink**
`< SRULE >` = **rule**
`< SRULE : 1 > = < t0 : SFS : 1 >`
`< SRULE : 0 > = < t1 : SFS : 1 >`
`< TRULE >` = **rule**
`< TRULE : 1 > = < t0 : TFS : 1 >`
`< TRULE : 0 > = < t1 : TFS : 1 >`.

In the case of relationships involving unary rules and simple-tlinks the type **simple-tlink-rule** is appropriate:

simple-tlink-rule (**top**)
`< t0 >` = **simple-tlink**
`< t1 >` = **simple-tlink**
`< SRULE >` = **unary-rule**
`< TRULE >` = **unary-rule**.

The tlink rule involved for the lamb example is then simply defined as:

simple-tlink-rule
`< SRULE > <= animal-grinding <>`
`< TRULE > <= vleescompounding <>`.

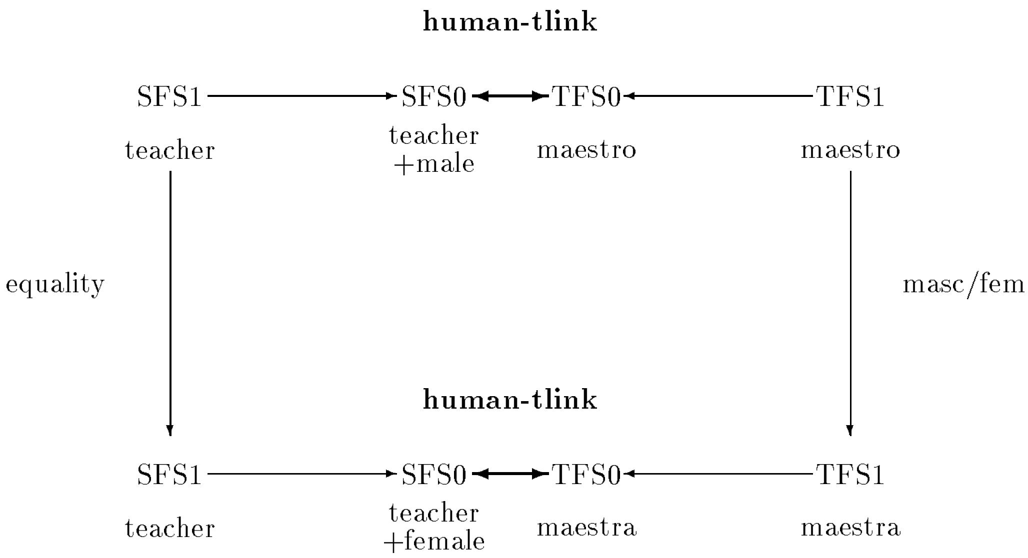


Figure 5.8: Tlink rule for *teacher* ≈ *maestro/a*

and again we use an abbreviated notation:

```
psort / psort :
   description .
```

where the LHS psort is the value of `< srule >` and the RHS is the value of `< trule >`. The example above then becomes:

`animal-grinding / vleescompounding :`
simple-tlink-rule.

Since we are just making use of the monolingual sense extension mechanism here we can rely on that to handle cases where the sense extension is blocked (e.g. *pig*). It does not necessarily matter for translation purposes whether the lexical rule can fully predict the effects of the sense extension; even if it is used to encode the regular aspects of the relationship between two existing lexicalised items, an appropriate translation link will be generated if the monolingual processes are sufficiently similar.

There are many examples of such correspondences; for example the English sense extension between trees and their fruits (*pear* etc) is mirrored in Italian with a gender distinction; the trees are masculine but the fruits feminine (*pero*, *pera*). In fact, our earlier examples of *muebles* ≈ *furniture* and *maestro/a* ≈ *teacher* can also be described more generally. A tlink-rule can be defined for human-denoting nouns so that given the equivalence between two entries of one gender the tlink the other gender can be produced. Since gender in English is normally unmarked, one instantiation of this is with a 'null lexical rule', which states equality between two FSs. Thus given the tlink between *teacher* and *maestra* we can automatically generate that between *teacher* and *maestro* (see Figure 5.8).

The relationships at the 'sides' of the tlink-rule need not be lexical rules as they are in this case, indeed in many cases the motivation for a tlink rule is that a process which can be regarded as a lexical rule in one language has to be modelled by phrase formation in another language. Thus in the example of causative formation, the lexical rule in English which relates the causative and inchoative uses of movement verbs such as *fly* can be related to the process of forming a phrase with the explicit causative verb *fare* in Italian.

5.4 A full example

In this section we will discuss a relatively trivial example in some detail in order to give a better idea of the use of tlinks and to introduce the translation system. Consider the translation of the structure given for *John likes Mary* in Figure 4.1 into Spanish *Maria gusta a Juan* the structure for which is given in Figure 5.9. We will assume the following, relatively straightforward tlinks.

```
john_1 / juan_1 :
   simple-tlink .

mary_1 / maria_1 :
   simple-tlink .

likes_1 / gusta_1 :
   arg-switch-tlink .
```

The arg-switch-tlink type allows the subject of *like* to be the object of *gustar* and vice versa. Thus the normal **tverb-tlink** type is:

```
tverb-tlink (tlink)
   < SFS : 0 > = < TFS : 0 >
   < SFS : 1 > = < TFS : 1 >
   < SFS : 0 : SYNSEM : SEM : ARG1 : ARG1 > =
       < TFS : 0 : SYNSEM : SEM : ARG1 : ARG1 >
   < SFS : 0 : SYNSEM : SEM : ARG1 : ARG2 > =
       < TFS : 0 : SYNSEM : SEM : ARG1 : ARG2 > .
```

and **arg-switch-tlink** is:

```
arg-switch-tlink (tlink)
   < SFS : 0 > = < TFS : 0 >
   < SFS : 1 > = < TFS : 1 >
   < SFS : 0 : SYNSEM : SEM : ARG1 : ARG1 > =
       < TFS : 0 : SYNSEM : SEM : ARG1 : ARG2 >
   < SFS : 0 : SYNSEM : SEM : ARG1 : ARG2 > =
       < TFS : 0 : SYNSEM : SEM : ARG1 : ARG1 > .
```

The difficulty with this translation is accounting for the Spanish particle *a* which precedes the (animate) direct object. We will use this example to illustrate some of the tlink mechanisms.[5] We will assume that NPs in Spanish have the boolean valued feature

[5] The analysis here is for illustrative purposes only and should not be taken as a serious proposal for the treatment of Spanish.

$$
\left[\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{s\text{-}sign}} \\
\mathrm{ORTH} = & \langle\, \mathbf{maria,\ gusta,\ a,\ juan} \,\rangle \\
\mathrm{LANG} = & \mathbf{spanish} \\
\mathrm{SYNSEM} = & \left[\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{s\text{-}synsem}} \\
\mathrm{CAT} = & \boxed{\mathbf{scat}} \\
\mathrm{SEM} = & \boxed{\scriptstyle 12} \left[\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{tvsem}} \\
\mathrm{PRED} = & \mathbf{and} \\
\mathrm{ARG1} = & \left[\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{tvformula}} \\
\mathrm{PRED} = & \mathbf{gustar\_1} \\
\mathrm{ARG1} = & \boxed{\scriptstyle 13}\ \mathbf{obj1} \\
\mathrm{ARG2} = & \boxed{\scriptstyle 14}\ \mathbf{obj2}
\end{array}\right] \\
\mathrm{ARG2} = & \left[\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{compl\text{-}formula}} \\
\mathrm{PRED} = & \mathbf{and} \\
\mathrm{ARG1} = & \boxed{\scriptstyle 15} \left[\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{npsem}} \\
\mathrm{PRED} = & \mathbf{maria\_1} \\
\mathrm{ARG1} = & \boxed{\scriptstyle 13}
\end{array}\right] \\
\mathrm{ARG2} = & \boxed{\scriptstyle 16} \left[\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{npsem}} \\
\mathrm{PRED} = & \mathbf{juan\_1} \\
\mathrm{ARG1} = & \boxed{\scriptstyle 14}
\end{array}\right]
\end{array}\right]
\end{array}\right]
\end{array}\right] \\
\mathrm{D{:}HEAD} = & \left[\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{idp2}} \\
\mathrm{SYNSEM} = & \boxed{\mathbf{vp\text{-}synsem}} \\
\mathrm{D{:}HEAD} = & \left[\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{tverb\text{-}sign}} \\
\mathrm{ORTH} = & \mathbf{gusta} \\
\mathrm{SYNSEM} = & \left[\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{tv\text{-}synsem}} \\
\mathrm{CAT} = & \boxed{\mathbf{tvcat}} \\
\mathrm{SEM} = & \boxed{\scriptstyle 12}
\end{array}\right]
\end{array}\right] \\
\mathrm{D{:}COMP} = & \Big\langle \left[\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{non\text{-}lex\text{-}np\text{-}sign}} \\
\mathrm{SYNSEM} = & \left[\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{np\text{-}synsem}} \\
\mathrm{CAT} = & \boxed{\mathbf{npcat}} \\
\mathrm{SEM} = & \boxed{\scriptstyle 16}
\end{array}\right] \\
\mathrm{D{:}HEAD} = & \left[\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{idp2}} \\
\mathrm{SYNSEM} = & \boxed{\mathbf{objn\text{-}synsem}} \\
\mathrm{D{:}HEAD} = & \left[\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{objn\text{-}sign}} \\
\mathrm{ORTH} = & \langle\, \mathbf{juan} \,\rangle \\
\mathrm{SYNSEM} = & \left[\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{objn\text{-}synsem}} \\
\mathrm{CAT} = & \boxed{\mathbf{objncat}} \\
\mathrm{SEM} = & \boxed{\scriptstyle 16}
\end{array}\right]
\end{array}\right] \\
\mathrm{D{:}COMP} = & \langle\ \rangle
\end{array}\right] \\
\mathrm{D{:}COMP} = & \Big\langle \left[\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{a\text{-}sign}} \\
\mathrm{ORTH} = & \langle\, \mathbf{a} \,\rangle \\
\mathrm{SYNSEM} = & \left[\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{a\text{-}synsem}} \\
\mathrm{CAT} = & \boxed{\mathbf{acat}} \\
\mathrm{SEM} = & \mathbf{dummy}
\end{array}\right]
\end{array}\right] \Big\rangle
\end{array}\right] \Big\rangle
\end{array}\right] \\
\mathrm{D{:}COMP} = & \Big\langle \left[\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{lex\text{-}np\text{-}sign}} \\
\mathrm{ORTH} = & \langle\, \mathbf{maria} \,\rangle \\
\mathrm{SYNSEM} = & \left[\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{np\text{-}synsem}} \\
\mathrm{CAT} = & \boxed{\mathbf{npcat}} \\
\mathrm{SEM} = & \boxed{\scriptstyle 15}
\end{array}\right]
\end{array}\right] \Big\rangle
\end{array}\right]
$$

Figure 5.9: The translated structure. I have abbreviated the DTRS : HEAD-DTR and DTRS : COMP-DTRS paths to D:HEAD and D:COMP respectively and shown COMP-DTRS as a list. All other omissions are as in the figure for *John likes Mary*.

OBJECT in their CAT which is set to **true** if they can be a direct object, and false otherwise, and that the entries for *juan* and *maria* have OBJECT set to **false**. A lexical rule, spanish-objn, can apply to animate **lex-np-signs** producing a sign which can combine with *a* (using **idp2**) to give a **non-lex-np-sign** which has OBJECT set to **true**. Given this, and the psorts above, the following treatments are possible for the bilingual component:

1. Treat *a* as an item which makes no semantic contribution and can therefore be introduced at will. In Figure 5.9 it can be seen that the SEM value of *a* is **dummy**. The LKB can be parametrised to accept this (and/or other possibilities) as a specification of a semantically vacuous lexical entry. Thus when translating from English into Spanish, the generation bag will contain *a* as well as the FSs resulting from tlink application. The sentence *Maria gusta Juan* will not be generated, because of the monolingual condition that animate direct objects must be marked. The tlink specified above for *john* will only translate it into the non direct object form of *juan*. In order to get the form with the lexical rule applied into the bag, another tlink has to be specified:

   ```
   john_1 / juan_1 :
      < SFS : 0 > = < SFS : 1 >
      < SFS > <= spanish-objn <>.
   ```

   Rather than doing this explicitly for each proper noun, we set up the following tlink-rule:

   ```
   equality / spanish-objn :
      simple-tlink-rule.
   ```

   This is applied to expand the tlinks. Note that only the name of the lexical rule need be known by whoever constructs the tlinks, and that there is no reference to the feature OBJECT in the bilingual component. [6]

2. Link the phrase *a juan* to *john*. The tlink involved is:

   ```
   john_1 / a_1 juan_1 :
      <> = tlink
      < SFS > = unary-rule
      < SFS : 0 > = < SFS : 1 >
      < TFS > = binary-inclusion
      < TFS : 0 > = non-lex-np-sign
      < TFS : INC1 : 0 > = < TFS : INC1 : 1 >
      < TFS : INC2 > <= spanish-objn <> .
   ```

   This is an example of a word to phrase tlink. The lexical entry for *john* is unchanged. The lexical entry for *a* and the result of applying the lexical rule spanish-objn to *juan* are included in the phrase, which is simply stated to be a **non-lex-np-sign**. Again this could be generated automatically, given a suitable tlink-rule, from the simple tlink between *john* and *juan*.

[6] The system should perhaps be modified so that the tlink-rule does not have to be explicitly specified, in that a semantically vacuous transformation such as spanish-objn should always be allowed. The difficulty with this however is that it is not entirely clear how to define semantic vacuity.

3. Link the phrase *john* to a phrase containing *juan* plus a psort. The tlink involved is:

```
john_1 / apsort juan_1 :
    <> = tlink
    < SFS > = unary-rule
    < SFS : 0 > = < SFS : 1 >
    < TFS > = binary-inclusion
    < TFS : 0 > = non-lex-np-sign
    < TFS : INC1 : 0 > = < TFS : INC1 : 1 >
    < TFS : INC2 > <= spanish-objn <> .
```

   This is very similar to the previous case except that apsort is an under-defined place holder for a lexical sign rather than the sign a_1 itself. It is necessary to specify that there is some lexical sign licensed by the tlink, even if nothing about it is specified. This is because one of the conditions on translation is that all lexical signs which do not have vacuous semantics have to be licensed by a translation link (see section 5.5).

The first of these possibilities is probably the most natural one to use in this case, but in examples where the lexical item cannot be treated as semantically vacuous the choice between specifying it explicitly and using an underspecified representation depends essentially on whether the grammar naturally allows a way of specifying the lexical item or items involved, without using language specific terms. If the phrase can be sufficiently constrained in this way, then this option is to be preferred because it avoids duplicating information in the monolingual grammar. For example, in translating *mueble* into English possible phrases include *piece of furniture*, *item of furniture*, *article of furniture* and (possibly) *stick of furniture*. It is clear that such individuating words depend on the mass noun involved, for example *blade of grass* is acceptable but **item of grass* is not, nor is **blade of furniture*. If it is possible to determine from the monolingual information which the valid individuating phrases for the mass noun are, then it is clearly inappropriate to encode this in the bilingual lexicon and all that should be necessary is to specify that an individuating phrase is present.[7]

In all these cases, the constraints given by the tlinks and the monolingual structure shown in Figure 4.1 are such that only the single structure shown in Figure 5.9 can be a solution. In the next section, we give a detailed description of the conditions on translation equivalence which determine the valid pairings. In the section following that, the algorithm actually used for translation is described.

5.5 Conditions on translation

A source structure (SS)/target structure (TS) pair is valid under a given set of tlinks iff

1. Both the SS and the TS are type resolved.

2. For all lexical signs in the SS:

[7] Clearly it is necessary in practice to avoid generating a ridiculous number of alternative translations by giving all these possibilities for a particular sentence containing *mueble*. But this should not be done by restricting the bilingual lexicon to one possibility as this would mean that the system were not reversible, since the other possibilities would not be translated. Of course this does not preclude weighting alternatives for acceptability in generation etc.

(a) there is a tlink such that the tlink source unifies with the lexical sign and the tlink target matches some part of the TS, or

(b) there is a tlink such that the tlink source unifies with a phrase which includes the lexical sign and a specification of an included sign from that tlink source unifies with the lexical sign and the tlink target matches some part of the TS, or

(c) the lexical sign is specified to make no contribution to the semantics of the SS.

3. For all lexical signs in the TS:

   (a) there is a tlink such that the tlink target unifies with the lexical sign and the tlink source matches some part of the SS, or

   (b) there is a tlink such that the tlink target unifies with a phrase which includes the lexical sign and a specification of an included sign from that tlink target unifies with the lexical sign and the tlink source matches some part of the SS, or

   (c) the lexical sign is specified to make no contribution to the semantics of the TS.

Normally we are interested in translating sentences, but SS/TS pairs of other types of sign may also be valid under this definition. Clearly these conditions are symmetrical with respect to the SS and TS. The identification of the tlink source and target sides depends on the language pair — if the tlinks have been defined as L1/L2 tlinks then the SFS side of the tlink is the source and the TFS side the target when translating from L1 to L2 and the TFS side of the tlink is the source and the SFS side the target when translating from L2 to L1.

A tlink source or target is said to match part of an SS or TS if:

1. the tlink source or target is lexical (that is, its type is a subtype of **lexical sign**) and unifies with some part of the SS or TS, or

2. the tlink source or target is phrasal (that is, its type is a subtype of **phrasal sign**), it unifies with some phrase which is a part of the SS or TS and all included structures in the tlink unify with some structure which is a subpart of that phrase.

In some circumstances a further condition on translation is necessary, that the target structure produced be the minimal one that satisfies these conditions. For example, this condition might be necessary to avoid translations which iterated adjectives, though this depends on the semantics.

The inclusion condition requires an extension to the LRL. It can be formalised most simply as follows:

Definition 38 (Inclusion) *A feature structure* $F' = \langle Q', q_0', \delta', \alpha' \rangle$ *includes another feature structure* $F = \langle Q, q_0, \delta, \alpha \rangle$ *if* $q_0 \in Q'$.

The definition is the same as that used in LFG. However it is important to note that we do not have cope with general inclusion as defined above in order to produce the effects we are seeking with tlinks. Firstly, we only define inclusion relationships between phrasal and lexical signs. Secondly, inclusion relationships need only be defined to hold between type-resolved feature structures, since both the SS and the TS will be type-resolved. Rather than implement inclusion in its full generality, the LKB uses task specific code to check for inclusion as described in the next section.

There is one more condition which is necessary for the Shake-and-Bake style approach, which is that variables which are (potentially) distinct in the source are also distinct in the target. This condition is necessary to avoid the possibility of conflating two variables and producing incorrect translations. For example, assume that we are translating the English sentence "John sees Mary and John sleeps" into English with a series of tlinks which specify identity. If we do not distinguish the variables pertaining to John and Mary, we will get not only the original sentence, but also "Mary sees Mary and John sleeps" etc. This is obviously a silly example, but the same problem can occur when doing more realistic translations. Again, formalising this requires an extension to the LRL, such as inequalities (Carpenter, 1992), but the implementation actually involves special purpose code.

5.6 The translation algorithm

The definition given above simply specifies what valid SS/TS pairings are, it does not describe how we actually translate a sentence in L1 into L2. It would in theory be possible to treat translation purely as a process of constraint resolution. For example, suppose we had only lexical tlinks and treated individual tlinks as pseudo-types like lexical entries and had no items with dummy semantics, then if the LRL was extended to allow set valued-features we could describe the translation relationship as follows:

```
[ translation
  SS : [ s-sign
         SET-OF-SIGNS = sign-set [ 1 ]]
  TS : [ s-sign
         SET-OF-SIGNS = sign-set [ 2 ]]
  TLINK-LIST : [ tlink-list
                 INPUT-SET = [ 1 ]
                 OUTPUT-SET = [ 2 ] ]]

tlink-list (top).

e-tlink-list (tlink-list)
< INPUT-SET > = {}
< OUTPUT-SET > = {}.

ne-tlink-list (tlink-list)
< HEAD > = tlink
< TAIL > = tlink-list
< INPUT-SET > = { < HEAD : SFS : 0 > } union < TAIL : INPUT-SET >
< OUTPUT-SET > = { < HEAD : TFS : 0 > } union < TAIL : OUTPUT-SET > .
```

Here the feature SIGN-SET on a phrase is assumed to be equal to the union of the SIGN-SET of each of its daughters, and the SIGN-SET value of a lexical entry is the singleton set containing that lexical entry.

Resolving the feature structure of type **translation** would give us the valid SS/TS pairings. However this would be totally impractical as an implementation technique. It is quite unnecessarily general — it would allow us to perform such operations as deciding whether a partial sentence in L1 was a possible translation of a partial sentence in L2, or translating given pieces of logical form, rather than orthographic strings for example, but all we are actually interested in doing is generating a sentence in L2 given an input sentence in L1 and vice versa. Thus we want the system to be reversible, but we do not need it to be able to resolve an arbitrary combination of information from the SS and TS.

The algorithm currently used in the LKB for translation is the following:

1. The source language sentence is parsed using general constraint resolution, that is the constraint program is given a goal of a feature structure of type **s-sign** (or other sign type) with the language specified as the source language and the orthography instantiated according to the input sentence.

2. All potentially distinct variables in the semantics of this structure are set to have distinct values. For example, if there are two nodes with type **obj** in the structure and **obj** is declared to be a type which applies to variables, then these values are further instantiated with the specially generated types **obj1** and **obj2** (these can be regarded as similar to the subtypes of **string**).

3. The available tlinks are applied to the source language structure to yield a *generation bag* of potential target language structures. This is done by traversing the target structure, checking for applicable tlinks at each node which is of a type applicable to phrasal or lexical signs.

   (a) For phrasal sign nodes, only phrasal input tlinks may apply. For all that do, the output of the tlink is added to the generation bag. Checking application of a phrasal tlink involves checking that the SS phrase to which the tlink is being applied contains all the appropriate lexical entries. As a side effect of application, these included nodes are marked.

   (b) For lexical sign nodes, all lexical sign input tlinks are tried and the output added to the generation bag. If no tlinks apply, the sign is checked to see whether it has "dummy semantics", that is whether it is specified that it will not make any contribution to the semantics of the sentence. If not, the node is checked for marks resulting from the previous application of phrasal constraints. If these exist it is noted that one of these phrasal constraints must apply in any TS. If none exist, translation fails, since there is no possibility of any output corresponding to the sign appearing in the TS. If any (phrasal or lexical) tlink does apply it is noted that this tlink translated this particular part of the SS.

4. The constraint resolution program is given a goal which is simply the feature structure of type **s-sign** (or other sign type). Normally resolving this would generate all the possible sentences of the target language, but in this case the resolution is constrained so that instead of a free choice of lexical items only items in the generation bag may be used. Since these are mutually constraining, because of their instantiated semantics, the structures generated are further restricted. The two modifications to the general constraint resolution code are:

   (a) Any phrasal constraints in the bag are treated as alternatives to the existing subtypes of a type in the push-down phase of resolution.

(b) Instead of all items in the lexicon being available as pseudo-types the only structures available are those with dummy semantics, those which unify with one of the set of lexical tlink outputs in the generation bag (which may be used anywhere in generating the TS) plus those which unify with one of the set of included lexical signs. These may only be used as subparts of phrases to which their including constraint has been applied.

5. All the generated structures are then checked to ensure that all the conditions are met, that is that they incorporate results from tlinks translating all semantically contributing lexical signs in the SS, that any phrasal input tlinks that were required were actually used, and that for all phrasal output tlinks used in the structure, all the included signs are actually included in the phrase as generated.

5.7 Limitations

As stated above, this approach is intended to allow expression of translation divergences where languages differ systematically in the ways in which a concept is expressed, such as the differing lexicalisation patterns mentioned. We do not attempt to address all the problems of translation mismatches. One standard example of a translation mismatch is that *wear* in English should be translated differently into Japanese according to the part of the body on which the garment is worn — *hateiru* for shoes, *kiteiru* for coat and so on (Wilks, 1991). The tlink mechanism is not particularly suitable for representing such mismatches. We do not want a phrasal tlink that links *hateiru* with *wear on the feet* or any similar phrase, since *on the feet* would usually be redundant.[8]

Linking *hateiru* and *wear* and attempting to represent the distinction in meaning within the feature structure for *wear* by means of lexical semantic features is an even less promising solution, since it would lead to a proliferation of features which could not be motivated monolingually, and could not be instantiated in general by the sort of processing that we are assuming. If we introduced a feature BODY-LOC for *wear* we might envisage instantiating it with "foot" in the cases where the sentence was (4a) or even (4b) but it is hardly reasonable to assume that this could be done without some inference for (4c). [9]

[8] In fact it is possible to envisage situations for which some of the material is redundant for most examples of MT where a phrase has to be used to translate a lexical item, including the example which we gave earlier of *novillo* being translated as *young bull*. Ultimately, any good MT system will have to be able to reason about the content of the sentence to avoid redundancy. In the absence of such a mechanism, it seems reasonable to use phrasal tlinks in cases where a phrasal translation will normally be correct.

[9] To some extent similar arguments could also be raised against the tlinks described above where we made *teacher* with a SEX instantiated to **male** or **female** translation equivalent to *maestro* or *maestra* respectively, given that the sex feature could not be instantiated without inference in sentences such as (3)

(3) The teacher had a beard.

The justification for the tlink's reference to sex is the assumption that it is sensible to encode SEX as a lexical semantic feature monolingually even in English, because of the personal pronouns and affixes such as *er* and *ess*.

(4) a John wore red shoes.
   b John wore boots/slippers/Doc Martens.
   c The company insisted that employees had regulation footwear.
     John wore these symbols of his enslavement to the capitalist system
     to work every day.

It is clear that in some cases some degree of inference on real world knowledge is required for correct translation and the tlink mechanism is not designed to supply this. In this example tlinks should be described for each possible translation of *wear* and augmented with the discriminating information. The expression of this would depend on how the distinction was to be made, whether by some further component of the MT system or by the user (as in the BCI system). It would also be possible to accept the limited degree of discrimination possible within a system such as the LKB, on the grounds that sentences such as (4c) are very rare compared with the cases where a simple mechanism would be adequate, but if the signs are augmented to allow this, the additional information should still be distinguished from that needed for monolingual processing.

Chapter 6

Conclusion

As we described in the previous chapters, the basic data structure used in LAUREL are typed feature structures very similar to those proposed by Carpenter. We have simplified Carpenter's scheme slightly, to remove the distinction between appropriateness specifications and constraints, so that constraints on types are themselves just well-formed feature structures. What is more novel is the notion of a psort, because this allows inheritance to operate from objects such as lexical entries and lexical rules. The significance of this is that it allows the lexicon to be structured not only in terms of predefined types or templates, but also by encoding relationships between (parts of) lexical entries. The built-in relationships are non-default and default inheritance and equality, but by defining lexical rules it is also possible to describe lexical entries in terms of more complex transformations to other entries. Psorts are themselves well-formed feature structures, rather then descriptions of feature structures. This distinction is important when we consider default inheritance, as we discussed in section 3.3. Psorts are not explicitly organised into a hierarchy but it is necessary to impose a non-cyclicity condition on the induced hierarchy of psorts in order to allow evaluation.

The use of defaults in lexical description languages raises a number of issues. We have discussed the problems involved at arriving at a definition of default unification and given an overview of some of the possible alternatives. We proposed a definition which is computationally efficient and meets most of the other desiderata and extended this to a typed framework, in a way which results in the types constraining defaults. The conditions on default inheritance from psorts mean that multiple inheritance is only allowed when the inherited information is mutually consistent. This is restrictive, but avoids the complication of having to introduce an explicit mechanism for the resolution of conflicts. Practical experience with LAUREL so far suggests that the consistent multiple inheritance condition is not overrestrictive and that the non-associative behaviour of default unification and the potential complications involving reentrant structures are not significant problems when using the language. But we are aware that the solution to defaults adopted is not completely adequate. See, for example, Briscoe *et al* (in press) which considers the use of defaults and lexical rules to give an adequate treatment of blocking.

6.1 Constraints vs grammar rules

There are advantages and disadvantages to a uniform constraint based approach. The concept of ultimate well-formedness with respect to a type system appears natural and is very expressive, allowing complex constraints to be captured. As described in Section 4.1, the original definition of well-formedness with respect to a type system seems intuitively too weak, if the type system is viewed as a complete description of the set of possible feature structures. We have a uniform notion of constraint resolution which allows structures to be generated from any sufficient amount of information, whether it be the orthography of the sentence, the logical form or the description of the component parts of the sentence and their co-dependencies which arises from the lexicalist model of transfer adopted.

However, this process of constraint resolution can be very slow. While this is not too problematic for an application such as the ACQUILEX LKB, where the primary concern is to ensure sufficiency of information in lexical entries, it is a serious concern more generally. A second related disadvantage is that there is no distinction between categories of types which are intuitively distinct — types such as **phrasal-sign**, **agr** and **synsem-list** are treated identically. Yet these encode quite different concepts and while the first relates to an entity which intuitively has some clear linguistic significance, the third is really an artefact of the formalisation. The current account of types in the LKB does not allow a distinction to be made, but it is clear that a distinction is warranted both from the viewpoint of the grammar/lexicon writer and in terms of processing considerations, since efficiency in constraint resolution depends on applying the constraints in the most effective order.

In particular, it is unclear that the HPSG account of phrasal signs as feature structures which incorporate their daughters is the best one to adopt. Constraint resolution can be used to perform operations which cannot be straightforwardly mimicked by more conventional grammar rules. For example, as shown in Section 4.2, we can model the operation of appending two lists of arbitrary length in a constraint framework and we cannot do this by simple feature structure unification. In general terms, recursive structures built up by simple unification using grammar rules have to mirror the structure of the parse tree. Constraint grammars allow the possibility that different parts of the sign are constructed differently. Building up lists by append is a simple example of this, where the structure formed is always a flat list despite the fact that the parse tree has quite a different structure.

However, it is not clear to me whether HPSG currently takes advantage of this possibility in any very significant way. There have to be good reasons to adopt an approach which makes most known parsing technology inapplicable. Furthermore the description seems too general, one would like the value of DTRS to be regarded as different in kind from the value of features such as AGR — in particular a phrase which has a phrasal sign as a daughter can have a head feature which is dependent on the value of the enclosed sign's AGR feature, but should not depend directly on any values encoded inside its DTRS.

Lexical and morphological rules are also problematic within a uniform constraint based account. The conventional option of treating lexical rules as expanding the lexicon is unsatisfactory for several reasons. It implies an enormous expansion of the size of the lexicon which is computationally problematic, given that lexical rules can be applied in sequence. It is not even clear whether this process would necessarily achieve closure without an arbitrary limit being placed on the number of applications of rules. There is no clearly principled restriction on affixation and if sense extensions such as those relating count and mass usages of nouns are encoded as lexical rules (see Copestake and Briscoe, 1992) it is not clear at what point the application of these rules should terminate. Furthermore, such processes can apply to phrases, to a limited extent at least. But the alternative of treating signs derived by morphological or lexical rules as complex with some sort of DTRS feature (by analogy with phrasal rules) also seems unattractive, because there is no linguistic motivation for regarding this information as part of the derived sign. A third possibility is to encode lexical rule like effects directly in the type hierarchy, by making the input form underspecified with respect to the output form. For example, we could have a type representing a noun which was unspecified for number which would have two subtypes, one corresponding to the singular and another to the plural form. Constraint resolution would then result in all nouns being specified as one of these subtypes. While this is a plausible way of treating inflectional morphology, it does not allow for the effects found in derivational morphology and sense extension where the usual lexical rules involve a shift in type from one part of the hierarchy to another, not simply a specialisation. Although it might be possible to develop a concept of the lexicon where the lexical entries were underspecified with respect to type and derivational and sense extension rules acted to specialise types, this would be a big move to make.

It therefore seems that there are advantages in keeping the concept of the parse tree and constituent structure distinct from the description of signs. Blackburn *et al.* (1993) discusses how both trees and feature structures may be modelled in a *layered modal language*, one layer describing the trees and the other the feature structures which decorate the nodes. This allows the formalisation of trees and signs using different modal languages with the interrelationship between them also being formally described. At least in principle, this seems preferable to attempting to encode everything in terms of feature structures. It is still possible to view this as a constraint language and such an approach is largely compatible with the description of machine translation given here. In fact it is preferable to the monostratal account, since the interrelationship between phrasal signs and their constituents is treated as a special case within the current framework. In particular, in a layered account, it would not be necessary to describe inclusion relationships within feature structures, but only within trees.

Further work on using tlinks will be reported in Copestake and Sanfilippo (in preparation). There are many problems in determining how a multilingual lexical knowledge base can be used within a real MT system, some of which apply regardless of the architecture or approach to MT assumed. One major issue which I have not addressed is what we actually mean by the concept of translation equivalence. It is clear that a bilingual transfer lexicon will always be restrictive, in that it will not support all possible ways of translating a phrase into a (roughly) denotation equivalent phrase in another language. For example, if a word sense in English is close enough in meaning to one in Spanish that they are adequate translations of each other with respect to denotation in all contexts, the bilingual lexicon will presumably only give the word to word translation equivalence and not an alternative word to phrase relationship, even if the phrase was also a good translation (unless the word was rare or restricted in register). In such a case, the bilingual lexicon would be restricting the options for translation correctly but in doing so would be usurping part of the function of the monolingual system since avoiding periphrasis is a general principle. (In an idealised interlingual system, on the other hand, the generation of the phrase would have to be blocked by the existence of the word.) Our concepts of a multilingual lexicon are clearly quite closely related to experience of bilingual dictionaries.[1] It is not clear whether printed dictionaries give us a good model for the multilingual lexicon, even if we add the ability to generalise over classes of translation equivalence as in the

[1]The particular classes of mismatch which were discussed on the first description of tlinks were partly based on an analysis by Piek Vossen of the types of definition found in the Van Dale dictionaries, but turned out to be very similar to categories discussed in the MT literature.

tlink model. Further work will address this issue in more depth.

Appendix A

Algorithms etc

This appendix contains algorithms and other details of the LKB that might prove useful (although much of it is rather obvious). Much of it is derived from comments in the code and therefore the prose isn't particularly polished. At least some of the algorithms are almost certainly non-optimal, but they do all appear to work! Some of the data structures used are descibed in detail in the programmer's guide, which is included as Appendix B. The algorithms for constraint resolution and machine translation using tlinks were described in sections 4.2 and 5.6 respectively.

A.1 The LKB interface

The LKB interface is entirely menu-driven — the feature structures, type hierarchies, parse trees and so on are displayed in windows which allow the user to do various useful things, such as print structures out to a file as TeX macros,[1] store structures as psorts and so on. These features are described in the user manual which is included as Appendix C. One of the most important feature of this interface is the ability to shrink parts of feature structures so that they are not displayed (shown as a box round the type). Without this the interface would be almost unusable for large feature structures. Shrunkenness is coded for on feature structure nodes, and unification is defined so that the result of unifying a shrunk node is also shrunk. This allows the user to set up constraints on types so that any parts which are not currently of interest are shrunk and have the shrunkenness inherited by all structures of that type. (These settings may also be stored in a file associated with a type system description file.)

A.2 Checking the type system

First, check that the type hierarchy itself is valid viewed as a graph without considering the constraints.[2] There are several relevant conditions:

1. connectedness

2. existance of top

[1] Hence the gratuitous use of incomprehensible and indigestible feature structure diagrams thoughout this report.

[2] I am grateful to John Bowler for describing most of the techniques in this bit, even though I suspect he did this to avoid having to make his own supper while I tried to work it all out.

3. no cycles
4. unique greatest lower bound

We also check for redundant links, that is, where a node is both an immediate and a non-immediate daughter of another node, since allowing these complicates some of the other algorithms.

The first two conditions are trivial. Checking for cyclicity uses a well-known algorithm, which involves using two colours or marks here referred to as *active* and *seen*:

1. Start from top — mark node as active and seen.
2. Go depth first, marking all nodes as active and seen until a leaf node is reached. Mark this as seen (but not active) and go back up. When all leaf nodes of a parent have been marked remove its active mark, and so back up the hierarchy.
3. If you find a node marked as seen but not active then treat it as a leaf node.
4. If at any point you find that you are about to mark an already active node then you have a cycle.

This has a simple variant to check for redundant links:

1. Mark all daughters of the node as active as well as the node itself.
2. Marking an already active node either indicates a cycle or a redundant link.
3. Unmark as active only when all daughters have been checked.

Checking for the unique greatest lower bound condition could be expensive, since the obvious algorithm involves doing a pairwise check on each possible pairing of types. However this can be done much more efficiently by checking for multiple lowest upper bounds on the assumption that there are a relatively small number of types with multiple parents, since if two types have multiple greatest lower bounds, they themselves will both be lowest upper bounds for that pair of types. The actual algorithm is:

1. Store a list of ancestors for each node (this is useful for other things too).
2. Find each node with multiple parents.
3. For each such pair x and y
   (a) Let x1 be the set including x plus all its ancestors.
   (b) Let y1 be the set including y plus all its ancestors.
   (c) Let z be intersection of x1 and y1
   (d) If x or y are in z then they are ordered — stop.
   (e) Otherwise for each element in z remove its ancestors from z.
   (f) If there is $>$ 1 element in z at the end of this process then there is a problem — x and y are both glbs for these elements.

Once the type hierarchy has been checked as a graph, the constraints on types can be expanded and checked. To recapitulate from Chapter 2 the conditions on constraints are:

1. Type of constraint FS is equal to the type it is a constraint of.
2. Monotonicity — a constraint FS is subsumed by the constraint of its parent(s).
3. Compatability of constraints (that is well-formedness of all substructures of a constraint FS).
4. Maximal introduction of features (a feature may only be introduced at one point in the type hierarchy).

The specifications of constraints given by the user are not taken to be complete, but rather as specifications to be expanded so that they do meet the first three conditions given. First of all we determine which types are atomic, that is have no constraint specification themselves and have no ancestors or descendants with a constraint specification. (Thus they have no appropriate features, and cannot be specialised into types with appropriate features.) Once this is done, for each non-atomic type:

1. If any constraint specification is given expand it into an un(der)typed FS without reference to any types.
2. Set the type of the constraint FS.
3. Add the features specified to the maximal introduction table (see below).
4. Unify this FS with the constraints of the parents of the type, recursively expanding these with inherited information if they are not already marked as expanded.
5. Mark the type as expanded.

The type system is also checked to see is the maximal introduction condition is satisfied. The procedure is as follows:

1. Set up a hash table to associate maximal types with features.
2. When a feature is first found in a constraint specification construct an entry for it with its value temporarily set to that type.
3. Subsequently if the feature is found on another type there are three possibilities:
   (a) If the new type is a subtype of the existing type (or of one of the list of existing types) then do nothing.
   (b) If one or more of the existing types is a subtype of the new type, replace them with the new one.
   (c) If there is no subtype relationship this may be because there is an as yet unencountered type with the feature specified which is a supertype of both or it may be an error. Add the new type to the old.
4. Once all types have been processed, the table is checked for any features which have more than one associated type. These are instances of multiple maximal introduction.

Finally, all constraints are made well-formed. This is very similar to the process of making a feature structure well-formed when the constraints on types are already well-formed:

1. Start from the initial node.
2. Set the node type to be the maximal one for which all the nodes existing features are appropriate (this may introduce new features).
3. Recurse from step 2 on the nodes found by following each feature.
4. Unify the node with the constraint on its type.

Thus the initial step of determining the type according to the features takes place top-down from the initial node, but the process of unifying with the constraint on each type happens bottom-up with the terminal nodes being made well formed first. (This would be problematic if we allowed cycles, but since we don't, it isn't.) The only differences when making constraints well-formed are:

1. The initial node is not checked (it is well-formed by definition).
2. If a type is encountered which has not already had its constraint made well-formed, then this is done at this point.
3. Any occurences inside the constraint FS of the type with which the constraint is associated or any of its subtypes are signalled as an error.

A.3 Unification

Typed unification normally differs little from ordinary unification except that the type of the FS has to be set to the greatest common subtype of the types of the two FS being unified. In most cases where unification succeeds, one type will be a subtype of the other and in this case, no further complication is needed to ensure that the type of the result will be well formed if the initial structures were well-formed. However in the case where the greatest common subtype is a strict subtype of both the original types, a further unification with the constraint of that type is necessary. Thus the efficiency of unification is reduced, by a factor which depends on how frequent this situation is and how large the constraints are. In practice this appears to be an unimportant factor in the efficiency of the system as a whole.

The speed of a straightforward implementation is largely dependent on the amount of copying of FS that has to be done before attempting unification, given that the algorithm is desructive. Most unsuccessful attempts at unification can be avoided by performing a non-destructive check first. I have found that doing such preunification checks improves the speed of the parser by a factor of about three. All this involves is checking the specified values for compatability, without actually performing the unification.

A.4 Default unification

The version of default unification currently used in the LKB is defined to take two dags — one default and one non-default and to always return a result (which may just be the original non-default dag) unless the reentrant part of the default dag doesn't unify with the non default

The first step is to split the default dag into reentrant and non-reentrant parts (which is done by the trick of writing the dag out in path notation, checking for unifications with paths on lhs and rhs, and then recreating the dags). If ordinary unification of the non-default with the reentrant structures succeeds then use the result — otherwise fail. Doing the actual default unification involves marking the dag in order to determine the reentrant nodes. This allows the original value of reentrant nodes to be stored and thus the results of the default unification can be removed if necessary when the node is revisited.

If real unification succeeds then the type of a fs will be set to the glb, otherwise the type will be unchanged If the default type is not the same as, or higher up the hierarchy than the non-default, then the non-default fs is completely unchanged.

A.5 Lexical entries

When expanding a lexical entry we want to produce three things:

1. a non-default fs (well-typed)
2. a list of equality constraints of the form

   ```
   path = fs
   ```

   (the non-default fs will already have these incorporated but we need to check for non-violation, and mark the equality constraint on the fs so that default unification does not overrride them.
3. a default fs

The list of unification specification on a lexical entry can contain things of the following types:

1. a type (atom) — unify in constraint of dag to non-default fs
2. a unification of the old form — unify in to non-default fs
3. a c-identity — treat as ordinary unification, but also add to equality constraints
4. a non-local inheritance specification, equality, inheritance or default-inheritance. Here we first expand out the psort fs involved and then treat the non-local specification according to what sort of thing it is:

   inheritance unify in to non-default fs

   default inheritance unify in to default fs and unify type in to non-default fs

   equality unify in to non-default fs and add to equality constraints
5. We also have the implicit unifications involving the orthography etc to unify into the non-default FS.

Appendix B

LKB implementation outline

The following description is intended for users who wish to interface programs to the LKB. The functions which are described here will not be changed (without advance warning, at least).

B.1 Files

Some files were not included in the latest version of the LKB since they are unchanged from earlier releases, peripheral to the LKB as a whole and in most cases will have been superceded by work on other sites.

loadlkb.lsp load all the other files, start up system etc. Should contain the only pathname references. Small patches to the code will also be here until fixed in the main files.

types.lsp data structures for types and all operations on types which occur at runtime (rather than load time for which see typeinput.lsp and checktypes.lsp). Externally accessed functions are described in more detail below.

dag.lsp all operations on (typed) feature structures. Externally accessed functions are described in more detail below.

gen.lsp functions for subsumption, generalisation and equality of FSs

structs.lsp manipulating feature structures using the path notation (see below)

globals.lsp described in the main documentation — this is intended to contain information about the conventional meaning of certain types and features (eg **string**, **sign**)

checktypes.lsp Checking defined types for the conditions described in the documentation

marks.lsp Ancillary functions to implement certain checking operations efficiently

typeinput.lsp Reading in type files and checking for syntactic correctness.

outputfs.lsp Printing and displaying feature structures.

activefs.lsp Managing interactive display of feature structures.

lexinput.lsp Reading in lexicon files and checking for syntactic correctness.

lex.lsp Storage and handling of the lexicon

lexbatch.lsp Indexing and checking lexical entries

defaults.lsp Default unification etc

toplevel.lsp Menus and commands

rules.lsp Reading in and storing grammar and lexical rules

parse.lsp Chart parser

morph.lsp Morphological analyser/generator

dialog.lsp Graphical utility functions

tree.lsp Displaying the type hierarchy tree

graph.lsp General functions for displaying graphs used for the type hierarchy and for parse trees. Written by John Carroll.

check-unif.lsp Functions for doing quick precheck before destructive unification operations to avoid unnecessary copying.

constraint.lsp General constraint resolution.

append.lsp Optional special purpose code to do appends, increasing the efficiency of constraint resolution.

mt.lsp Machine translation using constraint resolution.

tlink.lsp (was in tlinking:tlink.lsp) Functions concerned with tlink application, storage, reading and outputting.

tlink-rule.lsp Tlink-rule application etc.

convert.lsp Skeleton function for converting LDB entries into LKB entries plus example applications. Not included in latest release.

tlinking:mapping.lsp The main control file for the mapping functions; contains the definitions of the functions called from the **Map** menu. Not included in latest release.

tlinking:biling.lsp Interfacing bilinguals stored in the LDB to the mapping functions. Not included in latest release.

tlinking:user-interface.lsp User interface functions for the feature structure matching system. Not included in latest release.

tlinking:weighted-match-fs.lsp Feature structure matching. Not included in latest release.

tlinking:spanglish.lsp Functions for accessing the "fake" bilingual. Not included in latest release.

B.2 Toplevel commands

The LKB menu calls the following commands (none take arguments, functions marked with an asterisk are described further below):

Menu	**Option**	**Function**	**File**
LOAD	Type file	read-type-file	typeinput.lsp
	Type files	read-type-files	typeinput.lsp
	Lexicon file	read-lex-file	lexinput.lsp
	Compiled lexicon file	read-q-lex-file	lexinput.lsp
	Psort file	read-psort-file	lexinput.lsp
	Grammar file	read-grammar-file	rules.lsp
	Lexical rule file	read-lex-rule-file	rules.lsp
	Morphological data file	morph-file-compile	morph.lsp
	Tlink file	read-tlink-file	tlink.lsp
	Tlink rule file	read-tlink-rule-file	tlink.lsp
CONVERT		make-LKB-lexicon-from-LDB	convert.lsp
VIEW	Type definition	show-type-spec*	toplevel.lsp
	Expanded type	show-type*	toplevel.lsp
	Lex or psort definition	show-lex-def*	toplevel.lsp
	Lex or psort entry	show-lex*	toplevel.lsp
	Word definitions	show-word-defs*	toplevel.lsp
	Word entries	show-words*	toplevel.lsp
	Grammar rule	show-grammar-rule*	toplevel.lsp
	Lexical rule	show-lex-rule*	toplevel.lsp
	Tlinks	show-tlinks	toplevel.lsp
	Tlink rules	show-tlink-rules	toplevel.lsp
PARSE	Parse input	do-parse*	toplevel.lsp
	Show parse	show-parse*	parse.lsp
	Show edge	show-edge	parse.lsp
	Show chart	show-chart	parse.lsp
LINKS	Apply lexical rule	apply-lex*	toplevel.lsp
	Apply all lex rules	apply-lex-rules*	toplevel.lsp
	Apply ordinary tlink	apply-tlink	toplevel.lsp
	Expand tlinks	create-new-tlinks	tlink-rule.lsp
CONSTRAINTS	Resolve psort	do-resolve-psort	toplevel.lsp
	Parse	do-constraint-parse	toplevel.lsp
	Translate	do-translation	toplevel.lsp
	Translate psort	do-translate-psort	toplevel.lsp
MAP	Define languages	map-define-language	mapping.lsp
	Select source entries	select-psorts-to-link	mapping.lsp
	Select Bilingual	map-select-bilingual	mapping.lsp
	Perform mapping	map-do-map	mapping.lsp
INDEX	Index & Check	index-do-index	lexbatch.lsp
	Load indices	read-indices	lexbatch.lsp
	Save indices	save-indices	lexbatch.lsp
TIDY UP		clear-non-parents	lex.lsp
EDIT	Inherit display settings	inherit-display-settings	toplevel.lsp
OUTPUT	Dump system	dump-lkb	toplevel.lsp
	Display settings	output-display-settings	toplevel.lsp
	Lexicon file	output-lexicon-file	toplevel.lsp

B.2.1 Display

All the functions above which display feature structures call display-fs, or display-fs-and-parents, or display-fs-and-paths (defined in file activefs.lsp).

display-fs *feature-structure string*
The feature structure is displayed in a window with title *string*. The types in

the feature structure have associated menus, and the identifier of the feature structure as displayed at the top of the window also has an associated menu.

display-fs-and-parents *feature-structure string list-of-typenames*
The feature structure is displayed as above, with the *list-of-typenames* displayed as parents (these will have associated menus).

display-fs-and-paths *feature-structure string list-of-dpaths*
The feature structure is displayed as for display-fs, with the *list-of-dpaths* displayed as default unification paths. The psorts in such paths will have associated menus.

There are two generic functions in outputfs.lsp, display-dag and display-dag1 (called by display-dag) which are used to display feature structures in a variety of formats:

display-dag *feature-structure device* &optional *file-name.*
display-dag1 *feature-structure device stream*

Device must be one of the following:

simple displays to the ordinary output

window displays to a bitmap window — used to produce printed output. Called by print-fs etc.

swindow as window but displays shrunk output in the same way as edit.

edit displays to a bitmap window with menus associated with parts of the feature structure. Called by display-fs and associated functions (as listed above) and also by redisplay-fs. Parts of the feature structure may not be displayed if they have been defined to be "shrunk". (edit is a misnomer).

path produces the path notation

shrunk is used just to return a value (ie no output occurs) The value is a list of paths which terminate in a dag whose representation will not be fully displayed by display-fs etc.

B.2.2 Parsing

do-parse calls parse which is in parse.lsp:

parse *list of strings*
Applies the chart parser to the sentence which is represented as a list of strings. Displays the feature structure(s) associated with any valid parse(s) and also updates the chart etc. The edges which are full parses are stored in `*parse-record*` which is a list of edge structures.

```
(defstruct (edge)
id
rule-number ; either the word or the number of the rule
            ; that formed the edge
```

```
dag ; the feature structure
category ; the type of the feature structure associated
         ; with the edge
leaves
children)
```

show-parse is called by parse to display the results.

B.3 Data structures and basic operations on types

The data structures and the basic operations on types are defined in types.lsp.

> get-type-entry *type-name*
> returns a type structure if defined or NIL

```
(defstruct (type)

;; basic structure

name
parents ; the set of immediate supertypes of a type.
constraint ; the feature structure C(t).

;; to allow information to be associated with a type and
;; retrieved by the user

comment ; a string which should provide some helpful information

;; For implementational reasons we also need

daughters ; the set of immediate subtypes of a type.
ancestors ; the set of all the supertypes of a type.
constraint-spec ; the constraint as specified by the user
                ; (in path structures)
local-constraint ; the feature structure formed from
                 ; the constraint-spec
appfeats ; the top level features of the constraint.
atomic-p ; t if the type is atomic, nil otherwise
enumerated-p ; true if the type specification made use of a
             ; disjunctive list of subtypes.
             ; This information is retained in order
             ; to block attempts to define further subtypes
             ; (which would probably be mistakes).
marks        ; used to perform some operations efficiently
             ; (see marks.lsp)
shrunk-p     ;
visible-p    ; for displaying the type hierarchy
             ; portions of which may be shrunk
)
```

The following functions are defined in types.lsp. Apart from is-valid-type and greatest-common-subtype, all assume that they are being passed a (potentially) genuine type name rather than a string.

1. is-valid-type (name) $\rightarrow$ boolean
   checks for a genuine type entry or for a string.
2. string-type-p (type-name) $\rightarrow$ boolean
   true if this type can take string values
3. constraint-of (type) $\rightarrow$ fs
   $C(type)$
4. constraint-spec-of (type) $\rightarrow$ fs
5. appropriate-features-of (type) $\rightarrow$ set of features.
6. subtype-p (type1, type2) $\rightarrow$ bool
   $t_1 \sqsubset t_2$
7. greatest-common-subtype (type1, type2) $\rightarrow$ type3 (or nil)
   type1 $\sqcap$ type2

The following take a type-entry:

1. get-descendants (type-entry) $\rightarrow$ list of type names
   returns the set of all types which are strict subtypes of the type specified by type-entry

B.4 Accessing parts of feature structures

The definition of a path structure is in structs.lsp:

```
(defstruct (unification) lhs rhs)
;; the path notation in a file gets converted into unification
;; structures
;; eg < t1 f1 : t2 f2 > = t3
;;    < f1 > = < f4 >.
;; will be converted to
;; ( #S(UNIFICATION
;;        LHS #S(PATH TYPED-FEATURE-LIST
;;               ( #S(TYPE-FEATURE-PAIR TYPE T1 FEATURE F1)
;;                 #S(TYPE-FEATURE-PAIR TYPE T2 FEATURE F2)))
;;        RHS #S(U-VALUE TYPES (T3)))
;;    #S(UNIFICATION
;;        LHS #S(PATH TYPED-FEATURE-LIST
;;               ( #S(TYPE-FEATURE-PAIR TYPE TOP FEATURE F1)))
;;        RHS S(PATH TYPED-FEATURE-LIST
;;               ( #S(TYPE-FEATURE-PAIR TYPE TOP FEATURE F4)))))
;; assuming *toptype* is top
(defstruct (path) typed-feature-list)
```

```
;; typed-feature-list is a list of type-feature-pair structures
(defstruct (type-feature-pair) type feature)
; type may be nil if ignored
(defstruct (u-value) types)
; representation of atomic values
; types is a list of type names
; (a list to allow disjunction of atomic values)
```

The functions called by most of the modules which access feature structures are:

process-unifications *list of unifications* &optional *boolean* → fs or nil
Returns a new feature structure which results from the unification of the unification structures specified. If unification fails a warning message is output. If the optional boolean value is set a well-formed feature structure is returned. (process-default-unifications is similar but carries out default unification. It is defined in lex.lsp)
unify-paths *path-or-value1 fs1 path-or-value2 fs2* &optional *boolean*
Either takes two paths and two feature structures and attempts to destructively modify fs1 so that the feature structure at the end of path-or-value1 is the unification of the feature structure at the end of path-or-value2 in fs2, or takes one feature structure and a path and an atomic value (u-value) and attempts to destructively modify the feature structure so that the value at the end of the path is the atomic value. Returns nil if unification fails. (If the optional boolean value is true then a well-formed feature structure is returned.)

The function to return the feature structure at the end of a given path is:

dag-at-end-of *path-or-value fs1* → fs2
path-or-value must either be a path or a u-value structure

To create a path from a list of feature names:

create-path-from-feature-list *list-of-features* → path
returns a path structure with types specified to be the most general type, `*toptype*`.

The following are defined on feature structures in dag.lsp:

1. type-of-fs (fs) → type
   returns $\theta(q_0)$

2. top-level-features-of (fs) → set of features

3. create-wffs (fs) → fs or nil
   returns a new strictly well-formed feature structure if possible or nil if fs cannot be made into a well-formed feature structure. Non-destructive.

4. unify-dags (fs1, fs2) → boolean
   Destructively modifies fs1 to produce the unified feature structure. Returns nil on failure (fs1 may still have been modified).

5. unify-wffs (fs1, fs2) → boolean
   As above but feature structure returned will be well-formed if the input feature structures were well-formed. (This may involve an extra unification with the constraint of the type of the feature structure formed, hence it will be less efficient than unify-dags.)

6. copy-dag-completely (fs) → fs
   Returns a copy of the feature structure. This should normally be done before any destructive operations are performed.

B.5 The lexicon and psorts file

The storage of lexical entries and psorts is managed by functions in lex.lsp. Lexical entries are feature structures indexed by a string (orthographic form) and an identifier; psorts are similar but only indexed by identifier. The structure for both is:

```
(defstruct (lex-or-psort)
  orth ; orthographic form (a string)
  sense-id ; a number or sequence of letters and numbers
  id ; a unique identifying atom which is formed by combining
     ; orth and sense-id for lexical entries
  unifs ; the unifications specified
  language ; defaults to *current-language*
  local-fs ; feature structure formed by expanding
           ; the unifications
  mother-p ; set if the psort is specified as a parent
           ; clear-non-parents is sensitive to this value
  full-fs ; final fully expanded feature structure)
```

Psorts and lexical entries are stored on a file in unexpanded form and only expanded (and checked for well-formedness) when accessed. Once expanded they are cached.

The following access functions return the full structures:

1. get-psort-entry *id* → lex-or-psort (or nil)

2. get-lex-entry *string* → list of lex-or-psort (or nil)

B.6 Tlinks

Tlink functions are in tlink.lsp. Tlinks are stored as a list of tlink structures:

```
(defvar *tlinks* nil) ; a list of tlink structures

(defvar *constructed-tlinks* nil) ; a list of tlink structures
                                  ; created when tlink-rules are applied

(defstruct tlink
   psort1 ; psort id
   lang1 ; language of psort1
   psort2
```

```
   lang2 ; language of psort2
   unexpanded-fs
   fs)
```

Two basic access functions are defined, and a third function which translates feature structures according to the transfer component of the tlink:

1. find-tlinks *psort-id* → list of tlink-structures (or nil)
2. find-tlinks-by-language *psort-id, target-language* → list of tlink-structures (or nil)
   Returns all tlinks transforming the given psort into a feature structure in the target language.
3. translate-feature-structure *fs, psort-id, target-language* → list of fs (or nil)
   psort-id ought to be appropriate to the feature structure given, typically this will correspond to a predicate name. If the fs unifies with the input half of the transfer component of the tlink (determined according to the target-language specified), the feature structure returned will be the output half of the tlink.

B.7 Tlink rules

Tlink functions are in tlink-rule.lsp. Tlink-rules are stored as a list of tlink-rule structures:

```
(defvar *tlink-rules* nil) ; a list of tlink rule structures


(defstruct (tlink-rule (:include lex-or-psort))
   rule1
   rule2)
```

One basic access function is defined, and another function which generates new tlinks according to the tlink-rules:

1. find-tlink-rules *rule-id* → list of tlink-rule structures (or nil)
2. create-new-tlinks → nil (constructs new tlinks and adds them to `*constructed-tlinks*` as a side effect). If the transfer component of the tlink, unifies with the input half of the tlink rule (with sides determined according to the target-language specified) then the output half of the tlink rule is used as a potential new tlink.

B.8 Indexing

Indexing and checking functions are in lexbatch.lsp. The indices calculated may be changed by editing globals.lsp. Access to indices, once created, is via two functions:

1. find-entries-of-type *path-list, type, language* → list of psort-ids (or nil)
2. find-entries-of-type *path-list, psort, language* → list of psort-ids (or nil)

B.9 Scripts

In order to load type systems, lexicons etc in a particular group it can be useful to write a file containing a script which loads the entire set of files needed. The commands which should be used are:

read-type-file-aux takes a pathname for the type file and an optional display settings pathname.

read-type-files-aux as above, but takes a list of pathnames.

read-lex-file-aux takes a pathname and a second argument which should be set to t to overwrite the existing lexicon and nil otherwise.

read-psort-file-aux takes a pathname for a psort file

read-grammar-file-aux takes a pathname for a grammar file

read-lex-rule-file-aux takes a pathname for a lexical rule file

read-tlink-file-aux takes a pathname for a tlink file

read-tlink-rule-file-aux takes a pathname for a tlink rule file

All these commands are those called by the interactive functions so any comments which apply to those, which don't pertain to the interaction itself, also apply to the auxiliary functions above. For the lexicon files, `*current-language*` should be set appropriately.

Appendix C

User reference manual

C.1 Introduction

The LKB is run from a menu driven interface. Most commands should be reasonably self-explanatory and prompting for files etc should occur in the obvious way. (However many error and warning messages appear in the toploop window; if something doesn't seem to be working please look there.) These notes are intended to be read in conjunction with a system.

C.2 Starting up the LKB

These notes assume that you have a copy of the directory `lkb4` on your machine. This should contain the following:

lkb.fasl The compiled lkb code

globals.lsp A lisp file which can be edited by the user to customise certain parameters (see Section C.4.12).

lkbload.lsp File to load the LKB. All pathnames are specified in this file; this is the only file which it should be necessary to edit before it is possible to run the LKB on your system.

ldbint.lsp Only loaded when inside the LDB, this file allows the LKB to interface to the LDB's querying mechanism etc.

data A folder containing all the data files.

texmacros LaTeX macros for typed feature structures.

The LKB may be run either directly from Procyon Common Lisp or from the LDB image. In either case loading the file `lkbload.lsp` will load the code and set up a system menu for interaction. (Note — you will have to edit `lkbload.lsp` to give the correct pathname for your directory. With the current version of the LDB, the LKB must be loaded before any dictionaries are loaded.) A menu called **Lkb** should appear. The only highlighted option in the menu at this stage is **Load** because a type system is needed before the other options are applicable.

C.3 Data files

Some data files are being distributed with this release because they illustrate various aspects of the LRL and the system. They are not necessarily compatible with the ACQUILEX type system, nor are they intended as replacements for it.

The folder `data` contains the following:

test A folder with three files, intended to demonstrate the extended LRL language.

> **test.types**
>
> **test.lex**
>
> **test.rules**

demo This is a folder containing more substantial type system files, sample lexicons, lexical rules and translation links which have been used in various demos.

> **globals.lsp** This should be loaded in, as a Lisp file, before attempting to expand any lexical entries.
>
> **common.types** Types in common to the verb and qualia type files.
>
> **link.types** Types for defining rules and tlinks.
>
> **qualia.types** Types for the noun qualia structure.
>
> **nouns.lex**
>
> **nounlex.rules**
>
> **dutch.lex** Trivial lexicon for demonstrating tlinks.
>
> **e-dutch.tlink** Depends on `nouns.lex` and `dutch.lex`.
>
> **demo.trule** To be used in conjunction with `e-dutch.tlink` and `nounlex.rules`.

other A further collection of lexicons and so on.

> **verb.types** Detailed syntactic and semantic description of verbs (see Sanfilippo, in press). Requires *common.types*, can be used with `qualia.types` and link.types.
>
> **misc.lex** English lexicon, mostly hand coded, for demonstration of LKB. (Includes entries from `nouns.lex`, some entries depend on `verb.lex`)
>
> **verb.lex** English lexicon, samples of automatically derived data - see Sanfilippo and Poznanski (1992). Some entries depend on `verb.psorts`.
>
> **verb.psorts** Psort file for use with `verb.lex`.
>
> **grammar.rules** Categorial style grammar rules for demonstrating verb types and tlinks. Depends on `links.types`.
>
> **lex.rules** Lexical rules for verbs and all the noun lexical rules from `nounlex.rules`. Depends on `links.types`.
>
> **morph.rules** Sample morphological rules to demonstrate the new system.
>
> **demo.tlink** Examples of tlinks to be demonstrated in conjunction with `misc.lex` and `spanish.lex`.

The files in the data folder were split in order to make it possible to work on verbs without loading the qualia types, and vice versa. `common.types` contains the types necessary for both, `links.types` contains the types needed for parsing, lexical rules and translation links. `common.types` should be read in before the other type files, because one type is deliberately redefined in `qualia.types`.

constraint data These files are intended to illustrate the constraint and toy MT system.

constraint.types

test.psort

engtest.lex

spanish.lex

test.tlink

C.4 Commands

C.4.1 Load

These commands allow the user to load various types of file using the menu interface. See section C.4.14 below, for details of how to load files from a Lisp function.

Type file A type system is loaded from one or more type files. If this option is selected, the user is prompted for the name of a single type file (e.g. `test.types`). A file should be selected and **Open** chosen. Choosing **Cancel** will abort the process. Once a type file is loaded a further prompt, for a display settings file, will appear. If a file which defines how types may be abbreviated for display has been created (see below), then this may be specified at this stage. If such a file does not exist, or if all feature structures are to be displayed in fully expanded form, then choose **Cancel**.

The following checks are performed:

1. connectedness
2. top specified uniquely
3. no cycles (this also checks for redundant links, where a type is specified to be both an immediate and a non-immediate descendant of a parent). The assumption is that this would only happen because of an error.
4. uniqueness of greatest lower bound
5. maximal introduction of features
6. well formed constraint expansion possible

If any of them fail, further checks are not performed. Warning messages are printed. If everything succeeds the message "Type file checked successfully" will appear. [1]

Once a type file is loaded successfully, a window showing the type hierarchy is displayed. The top of the hierarchy, that is the most general type, is displayed at the left of the

[1]Warnings of unary branches in the type hierarchy will be displayed if the appropriate flag is set, see section C.4.12.

window. This is scrollable by the user and is automatically scrolled by various **View** options. Nodes in the window are active; clicking on a type node will give a menu with the following options:

Hierarchy Scroll the type hierarchy window so that the type is centred, and highlight it.

Help This menu option is only available if a comment has been specified for a type. The type name and comment are printed in a separate window.

Shrink/Expand Shrinking a type node results in the type hierarchy being redisplayed without the part of the hierarchy which appears under that type being shown. The shrunk type is indicated by an outline box. Any subtypes of a shrunk type which are also subtypes of an unshrunk type will still be displayed. Selecting this option on a shrunk type reverses the process.[2]

Type definition Display the definition for that type (see Section C.4.3, below).

Expanded type Display the expanded definition for that type (see Section C.4.3, below).

As an illustration:

```
main-verb (verb) "Covers all verbs apart from auxiliaries" < syn : cat
: m-feats : inverted > = false.
```

This states that the parent type of **main-verb** is **verb** and its constraint will be found by unifying the feature structure specified, with the constraint of **verb**. It is unnecessary in general to specify the types in the path specifications — these are inferred from the features specified. A comment or example may be specified as a string which must occur after the parent list; such a comment will be displayed when **Help** is chosen from a menu associated with the type.

(When a type system is reloaded, any lexical rules etc which have been loaded will have also to be reloaded — they are cleared because they may have become invalid. Only the expanded forms of lexical entries are cleared however, since the unexpanded forms will not in general be affected by the change in the type system. Thus it is not usually necessary to reload a lexicon after modifying the type system.)

Type files This option should be chosen to load a series of type files. A prompt will appear for the first file name; clicking on **Cancel** will abort the load at this stage. If a file is chosen, the second and subsequent files are prompted for in the same way. When all files have been chosen, click **Cancel**. The display settings option now appears, as above.

Lexicon file Once a type system is set up, a lexicon which has been defined using those types can be loaded. When a new lexicon file is loaded, existing lexicons (and psort files) may either be cleared, by responding **Yes** to the query which appears **Overwrite existing lexicon?**, or added to incrementally, by responding **No**. A prompt will appear for the language of the lexicon file; this is used by the translation link system. The default language selection and the possible choice of languages can be customised, see Section C.4.12, below.

[2]This command redisplays the type hierarchy window. It does not work properly in places where the hierarchy is reentrant. Both of these features will be fixed at some point, perhaps.

Lexicon files are only checked for syntactic correctness when they are read in — entries are expanded as they are accessed. The lexicon is stored in a temporary file (called by default `templex`) when it is read in, not in main memory. For checking and indexing lexicon files see Section C.4.8.

A lexical specification contains the following elements:

1. orthographic form (eg "sleep")
2. a sense identifier (eg "1")
   a unique identifier should be formed by concatenating this with the orthographic form with an underscore (eg "sleep_1"). This id will be used when treating the lexical entry as a psort. (Both the orthographic form and the id have to be Lisp atoms. Accents may be introduced using a backslash.)
3. a specification of the associated typed feature structure.
   The feature structure specification may contain any of the following:
   (a) a type name
   (b) one or more ordinary unification specifications
   (c) one or more non-local unification specifications

   The only requirement is that a type is specified or can be inferred.

For example:

```
teacher 1 < qualia > = human.
```

The lexical entry as a whole will be given the type **lex-noun-sign**, given the type system found in `common.types` and `qualia.types`, because this is the most general type for which the feature QUALIA is appropriate.

It is possible to specify paths which will have their values set to the orthographic form, sense-id and language of a lexical entry (see Section C.4.12). By default these paths are `<orth>`, `<sense-id : fs-id >`, `<sense-id : language >`. This is exactly equivalent to specifying that the paths take particular atomic values in the lexical entry, for example:

```
teacher 1 < qualia > = human < orth > = "teacher" < sense-id : fs-id >
= "teacher_1" < sense-id : language > = "english".
```

These values are all strings; **string** is a special type which, in effect has all (Lisp) strings as disjoint subtypes. See Section C.4.12, below.

The syntax of the psort inheritance specifications is fully described in the appendix. The syntax of default inheritance is basically:

```
path < feature-structure path
```

where, in the simplest case, `feature-structure` is specified as a name of another lexical entry or other psort. Non-default inheritance is indicated by `<=`, equality by `==`.

The feature list may thus be empty but the angle brackets must be present. For example, to specify that the sense "teachers 1" is exactly the same as "teacher_1" except for orthography and number:

```
teachers 1 < > < teacher1 < > < cat : m-feats : agr : num> = pl.
```

Compiled lexicon file Selecting the option **Lexicon File** under **Output** results in the current lexicon (which may consist of a combination of lexicon files) being output in a more machine readable format[3]. Reading in such files is considerably quicker than reading in the path notation files from which they are derived.

Psort file The psorts file is intended to allow psorts which are not also lexical entries to be defined. (For example `verb.psorts` which is needed for some entries in `verb.lex`.) The format for such psorts is identical to that for lexical entries except that there is only a single identifier. Psorts specified in this file differ from lexical entries only in that there is no orthographic form specified to index them.
NOTE: Psorts are removed just like lexical entries if you answer **Yes** to 'Override existing lexicon' when loading in a new lexicon file. Reading in a new psorts file will override the old one.

Grammar rule file The format for all rules is very similar to that for lexical entries except that there is only a single identifier instead of an orthographic form plus identifier. See the file `grammar-rules` for examples. The types **rule**, **unary-rule**, **lexical-rule** and so on, set up in `link.types`, define the feature structure skeletons for rules so that the daughter feature structures unify with the feature structures at the ends of the paths `<1>`, `<2>` ... and the mother at the end of the path `<0>`. Linear ordering is in general established by the function `establish-linear-precedence`; the default value of this function is thus currently rather simple, see Section C.4.12.

When a new grammar rule file is loaded, existed grammar rules may either be cleared, by responding **Yes** to the query which appears **Overwrite existing grammar?**, or added to incrementally, by responding **No**.

Lexical rule file Lexical rules are defined very similarly to grammar rules, but can be applied without invoking the parser. See the file `lex.rules` for examples. The same procedure applies for replacing or adding to existing lexical rules as described previously for grammar rules.

Morphological rule file Morphological rules are hybrids (see Appendix D). The portions which are concerned with feature-based unification operations are to be treated as standard lexical rules, and hence must be read into the system via the previously described procedure for entering lexical rules. That portion of the morphological rules, however, which describes the morphographemic alternation, must be compiled into a suitable data structure for the morphological analyser/generator. This is done through the currently described menu option. Note that this means that each morphological rule must be processed twice by the system — once as morphological data, and once as a lexical rule.

Tlink file Examples of tlink files are `e-dutch.tlink` and `demo.tlink`. the user is prompted for the source and target languages. The syntax is defined in the Appendix.

Tlink rule file Examples of tlink rule files are `demo.trule`. The syntax is defined in the Appendix.

[3] Lisp structures in fact, rather than genuine compiled code.

C.4.2 Convert

This menu option allows LDB dictionaries to be converted to LKB entries using user defined functions. It will only be available when running the LKB under the LDB. Since this is unchanged since the last release, it is not described further here.

C.4.3 View

Most of these options display feature structures plus other information in a window. Our usual orthographic conventions for drawing feature structures are followed; types are lowercased bold, features are uppercased. Psorts are uppercased bold. Such windows are active - currently the following operations are supported:

1. Clicking on the window identifier (ie the first item in the window) will display a menu of options which apply to the whole window.

   Print shrunk The feature structure will be printed as currently shown in the window. If a printer is selected in the Mac Chooser then the appropriate dialogue box will appear and the feature structure can be printed directly. It can be saved as a file if "Print2Pict" rather than an actual printer has been selected with the Chooser. Such a file can then be edited with MacDraw etc, and incorporated into MacWrite or TeXtures documents.

   Print expanded Print the feature structure etc. The feature structure will be printed in fully expanded form.

   Output TeX Outputs the FS as TeX macros to a file selected by the user. The macros are defined in `slidewrap.tex` and `docwrap.tex` to give appropriate sized output for overhead slides and documents respectively.

   Store fs The displayed feature structure will be stored as a temporary psort with a user given name. Lexical rules etc may then be applied to it. It is not written to a file or permanently saved in any way.

   LDB entry Only available if the LKB is being run from inside the LDB. If the appropriate dictionary is already loaded in the LDB and selected as the current dictionary in the toploop and the FS displayed has an LDB entry number in the predefined place (currently `< sense-id : ldb-entry-no >`) then the LDB entry will be displayed, when this option is chosen.

2. Clicking on a type (either a parent, or a type in the feature structure itself) will give a sub-menu with the following options:

   Hierarchy Scroll the type hierarchy window so that the type is centred. If the type hierarchy window has been closed (this is possible from the top menu), it will be redisplayed when this option is chosen.

   Help This menu option is only available if a comment has been specified for a type. The type name and comment are printed in a separate window.

   Shrink/Expand Shrinking means that the feature structure will be redisplayed without the feature structure which follows the type being shown. The existence of further undisplayed structure is indicated by a box round the type. Atomic feature structures may not be shrunk. Shrinking modifies the actual feature structure being displayed, so that if the window is closed, and subsequently a new window opened onto that feature structure, the shrunken status will be retained. Furthermore, any feature structures which are formed by unification with the shrunken feature structure, will also be displayed with equivalent parts hidden. Thus if the *expanded* constraint on a type has parts shrunk, any lexical entry which involves that type, which is expanded subsequent to any shrinking on the type, will also be displayed with parts hidden. Since types are evaluated when they are read in, shrinking parts of the constraint on a type will not affect the daughters of that type, unless the option **Inherit display settings** (under the **Edit** submenu) is chosen (see below). The **Display settings** option under **Output** allows settings to be saved — the suggested procedure for defining a display settings file is described in Section C.4.11.

If this option is chosen on an already shrunken feature structure then the feature structure will be expanded. Again this modifies the actual feature structure.

Type definition Display the definition for that type (see Section C.4.3, below).

Expanded type Display the expanded definition for that type (see Section C.4.3, below).

Clicking on a type which is in fact a string, and thus has no definition etc, will result in the warning beep, and no display.

3. Clicking on a psort will display a menu with the following options:

   Psort definition the definition for that psort (see Section C.4.3, below).

   Expanded psort Display the expanded definition for that psort (see Section C.4.3, below).

The order in which features are displayed is determined according to their order when introduced in the type specification file. For example if we have the following fragment of a type file:

```
sign (top) <syn> = syntax <sem> = semantics.

lexsign (sign) <orth> = string.
```

then when a feature structure of type **sign** is displayed, the features will be displayed in the order syntax, semantics; when a **lexsign** is displayed the order will be syntax, semantics, orth.

The various suboptions to View are described below; if the example files are loaded then accepting the default selection should illustrate how the commands operate.

Type definition Scrolls the type hierarchy window so that the type is centred and highlighted (redisplaying it if necessary). Displays the type's parents and the constraint specification.

Expanded type Scrolls the type hierarchy window so that the type is centred and highlighted (redisplaying it if necessary). Displays the type's parents and the full constraint on the type.

Lex or psort definition The local feature structure associated with a lexical entry plus any equality constraints and inheritance specifications.

Lex or psort entry The expanded feature structure associated with a lexical entry.

Word definitions All the local feature structures associated with a particular orthographic form.

Word entries All the expanded feature structures associated with a particular orthographic form.

Grammar rule

Lexical rule

Tlinks All tlinks defined using a particular lexical entry are displayed. The user may restrict this to tlinks appropriate for a particular target language. (Tlinks formed by **Expand tlinks** will not be displayed.)

Tlink rules All are displayed!

C.4.4 Parse

The parser is included in the LKB as an aid to developing the lexicon, rather than as a viable analyser. Parsing is very simple at the moment. Grammar rules are indexed by the type of the rightmost daughter. Lexical rules and grammar rules are treated identically by the parser, which means that lexical rules may actually be applied to non-lexical signs if the type system has not been set up to prevent this. Whilst there was no morphological component to previous versions of the system, the current version includes such a component — its functionality is descibed further in Appendix D.

Parse input This takes a list of strings as input and displays the parse trees associated with any spanning edges, which are of a type compatible with `*start-symbol*`. Clicking on a node in the parse tree will give the following options:

Feature structure Displays the feature structure associated with a node.

Rule Displays the grammar or lexical rule used to form the node.

Show parse Shows the results of the last parse (again). If the parse failed, prompts for the number of an edge to use as the top node in the parse tree (the default is the last node created).

Show edge Shows the parse tree from a given edge number.

Show chart Displays the chart (crudely). This can be useful if the parse has failed, in order to select an appropriate edge to display with **Show edge**.

C.4.5 Links

Apply lexical rule This prompts for a lexical entry and a lexical or morphological rule and applies the rule to the entry. The result is displayed if application succeeds. The user is then asked if they wish to check the result against an existing entry to see if it is compatible. This is useful, for example, to determine whether entries which have been automatically extracted from a dictionary could potentially be described as the result of a rule application.

Apply all lex rules This prompts for a lexical entry and tries to apply all the defined lexical and morphological rules to the entry, and to any results of the application and so on. (To prevent infinite recursion on inappropriately specified rules the number of applications is limited, see C.4.12.)

Apply ordinary tlink This prompts for the name of a psort and a language to translate it to. The results of application of any appropriate tlinks are displayed. Warning — this may be very slow if the result of the tlink application is not specified for orthography if no previous such tlink has been applied, because the system will attempt to expand out the entire lexicon in order to match up the tlink.

Expand tlinks New translation links are created by applying all possible tlink rules to all possible tlinks.

C.4.6 Constraints

Warning — don't try this with anything except a very small lexicon etc. All these functions are slow.

Resolve psort Prompts for the name of a psort and displays the resolved FS or FSs which result from treating the psort as a query to the constraint system.

Parse Takes a list of words, prompts for the language, builds a FS instantiated with their orthography (as determined by the function construct-orth-fs in globals.lsp) and attempts to resolve it.

Translate Prompts for a list of words — the SL (source language) string — prompts for the SL and the TL and asks whether the user wishes to set input and output types. If the answer is no, they are left as before (defaulting to the values in the globals file). If yes, prompts for SL and TL types (e.g. **s-sign**, **np-sign**) which will be used to give the overall type of the FS generated from the orthography and for the type of the TL FS respectively. Thus it is possible to use the system to translate for example, noun phrases as well as sentences.

Translate psort Similar to above, but takes a psort id rather than a SL string and only prompts for the TL output type.

C.4.7 Map

This option allows lexical entries in different languages to be compared, guided by a bilingual, in order to create tlinks. Since this is unchanged since the last release, it is not described further here.

C.4.8 Index

Lexicon files may be checked and indexed. Once this is done, retrieval of appropriate lexical entries is reasonably efficient. Indices may be saved and loaded. Indexing information may be customised by the user (see C.4.12).

Index & check Expands every entry in the current lexicon, checking its validity and indexing it if it is valid. This will take a considerable time on lexicons containing several hundred words. If expansion fails a warning message will be printed to the top-level window, rather than the usual continuable error message. After indexing is complete, this prompts for a file to save the indices to.

Load indices Prompts for a saved index file. You may choose whether to remove or add to existing indices.

Save indices Saves the current indices to a file.

C.4.9 Tidy up

This command clears expanded lexical entries which are cached, and which are not inherited from. If accessed again they will be read from file and expanded again.

Expansion of a large number of word senses will tend to fill up memory with a large number of feature structures. Most commands which are likely to do this actually clear the cached feature structures themselves, but if memory is becoming restricted this option can be used. However much more memory will be reclaimed by closing any open feature structure windows, and especially the Type Hierarchy window (from the top menu since it has no close box).

C.4.10 Edit

Inherit display settings The display settings are inherited down the type hierarchy, starting from the given type. Thus if the display of a type constraint has been interactively altered so that a particular path is shrunk, **Inherit display settings** will make this affect all subtypes of that type (see below).

C.4.11 Output

Dump system This creates an image file, which will essentially save the state of the current system, finishing the Lisp session. The session is then restarted. (This is more convenient way of saving start-up time than the earlier technique of saving a compiled type file.) The drawback is that the lexicon will have to be reloaded, because there is no way of saving the open stream to the temporary lexicon file. If you do not reload the lexicon it will still be possible to manipulate lexical entries which are cached, i.e. had been expanded before the image was made, but an attempt to access an uncached entry will give an error message, which complains that a stream is not open.

Display settings This saves the current status of the display settings for the type system in a file which can then (optionally) be read in when the type system is reloaded. The file format is not intended to be particularly human readable. The suggested approach to creating persistent display settings is as follows:

1. Set up the display settings for the highest type which you wish to display in reduced format, by using the shrink option on the feature structure representing the constraint of that type.
2. Use the **Inherit display settings** option under **Edit** to set up the displays of all subtypes of that type.
3. Modify the displays of any subtypes as necessary, and repeat the inheritance if required.
4. When the types are displayed in the desired way, do **Output Display settings** to a file. It is sensible to use some conventional name like types.disp to indicate the type file it is applicable to.
5. When loading the type file subsequently, specify that display file when prompted for a display file.

Lexicon file Now outputs a "compiled" lexicon file, containing all currently loaded lexical entries.

C.4.12 Global values

Various flags control the operation of the LKB and some feature and type names are regarded as special in various ways. These are all kept in the files called `globals.lsp`. Different files may be used for different type systems etc, so it may be necessary to load the correct globals file before something will work. Changing the values and reevaluating the forms in the file will have a permanent effect (if you save the file). This should not be necessary for the novice user working with the type system provided.

The current values (suitable for the systems in **constraint data**) are:

***toptype* = top** This gives *toptype* a value before a type file is read in, which is necessary if the definition of $\top$ does not appear before any path specifications.

***string-type* = string** The name of the type which is special, in that all Lisp strings are recognised as valid subtypes of it.

make-sense-unifications A function which should be defined to take three arguments so that the orthographic form of a lexical entry, its id and the language are recorded in an appropriate place in the feature structure. Currently equivalent to specifying:

```
< ORTH : HD > = sense
< LANG > = language
```

(The structures used etc are described in the document "LKB implementation outline").

***sign-type* = sign** To improve efficiency in compiling the grammar it is assumed that only types which are subtypes of *sign-type* may correspond to chart edges. If *sign-type* is set to nil then any type is a candidate.

establish-linear-precedence A function which takes a rule feature structure and returns the top level features in a list structures so that the ordering corresponds to mother, daughter 1, daughter 2 ... daughter n.

***start-symbol* = sign** a type which any complete parse must be compatible with.

***morph-rule-type* = morph-rule** a type which should apply to all morphological rules which affect orthography so that the parser knows how to apply them.

***maximal-lex-rule-applications* = 7** The number of lexical rule applications which may be made before the system assumes that they are being applied circularly.

feature-abbreviations This is a list of pairs — the rhs is a string which defines how a feature name will be displayed if the lhs string matches (the end of) it. This only affects active feature structure windows, not the printed versions. It is designed to allow list representations to be displayed more compactly. For example

```
(defparameter *feature-abbreviations* '(("-first" . "H") ("-last"
    . "H")))
```

would result in the feature names ARG-FIRST, NS-FIRST being displayed as H and ARG-LAST, NS-LAST being displayed as T.

***active-fs-page-width* = 700** If feature structures are too large for the default page width of 700 they will be "wrapped around" and become almost unreadable. If this happens this parameter may be increased but doing so decreases the number of feature structures which can be displayed before the system runs out of space.

***batch-mode* = nil** If set, errors in expanding lexical entries will be shown on the top-level window rather than signalled as continuable errors. This is done automatically by the indexing code.

type-indices

parent-indices Currently, type indexes are created on the type of the whole lexical entry (eg **lex-noun-sign**) and the type of the QUALIA (eg **c_subst**). Taxonomy indices are also created on the psort parent directly specified for the QUALIA, such as **drink_L_2_1**.

current-language Default language selected in user interaction windows.

possible-languages Possible languages displayed in user interaction windows.

***warn-of-unary-branches* = nil** Unary branches were discussed in the originalformal descriptions of the LKB (e.g. de Paiva, in press). In principle it is undesirable for the type hierarchy to contain unary branches, but in practise they are unavoidable when developing a type system (and it seems unlikely that anyone will claim to have a complete type system for any interesting linguistic fragment ...). If this variable is set warning of unary branches are given when the type system is loaded.

***do-really-well-formed-check* = nil** Checks during indexing for 'ultimate well-formedness' of lexical entries, i.e. that their feature structures can be extended so that all types are fully-specified leaf types. This is slow, so the default setting is nil.

***tlink-syntactic-sugar*= t** The slashed tlink syntax may be too restrictive for some purposes. If this flag is not set, the system assumes that tlink files have the ordinary psort file syntax.

***bag-types* = '(word)** For constraint solving, this sets the type(s) of the structures which are assumed to form the "bag" of (usually lexical) items which are available for generation.

***orth-path* = '(orth hd-orth)** For constraint solving and ordinary tlink application this gives the path which will provide the orthography for a sign, if any is specified.

***lang-path* = '(lang)** For constraint solving and ordinary tlink application this gives the path which will provide the language of a sign, if any is specified.

***semantics-path* = '(synsem sem)** For constraint solving, this gives the path which will provide the semantics for a sign.

***dummy-type* = 'dummy** For constraint solving, if the values of *semantics-path* is of this type (or a subset of this type) then the item is considered to have dummy semantics and can be inserted in a translation structure without being explicitly justified by a tlink.

type-of-isign

type-of-osign These two variables specify the type of the target FS for the parsed source structure and the generated target structure, respectively.

construct-generate-all For constraint solving, a function which generates a feature structure which covers all valid signs of type *type-of-osign*.

construct-orth-fs For constraint solving, a function which takes a list of input words and generates a FS of type *type-of-isign* with the orthography instantiated to that list.

***top-variable-type* = 'entity** For constraint solving, the most general type of all variables.

***top-lex-sign-type* = 'word** For constraint solving, the most general type of all lexical signs.

feature-ordering a list of features in what might be an appropriate order to attempt to resolve them. In practice this does not seem to make much difference.

C.4.13 Debugging constraint resolution and translation

The debugging facilities provided are very crude. If *debugging-constraints* is set then *problem-path* will be set to the path at which constraint resolution fails. In subsequent attempts to resolve a FS, the FS at the end of the problem path will be displayed. If *interactive* is set, the user will be asked when there is a (local) choice of paths for constraint resolution. Using *interactive* may make an otherwise intractable query possible to check for correctness.

Calling (display-generation-bag) will display the FSs in the generation bag and their included structures if any. These structures are the bits out of which the translation has to be constructed. Problems which cause a failure in translation may show up here. For example if a tlink incorrectly links a language specific predicate between the SL and TL this will not be easilt apparent from the tlink, since it won't be instantiated but may be clear from the generation bag.

C.4.14 Scripts

Sometimes the same sequence of items has to be loaded time after time. Sometimes the best thing to do is to dump an image but in other cases it may be better to load the items from a file. The following is a relatively complex example of such a script.

```
(let ((data-dir1 (pathname "internal:local-lkb:data:demo:"))
      (data-dir2 (pathname "internal:local-lkb:data:other:")))
     (load
      (merge-pathnames (make-pathname :name "globals.lsp") data-dir1))
     (read-type-files-aux
        (list
          (merge-pathnames (make-pathname :name "common.types") data-dir1)
          (merge-pathnames (make-pathname :name "link.types") data-dir1)
          (merge-pathnames (make-pathname :name "qualia.types") data-dir1)
          (merge-pathnames (make-pathname :name "verb.types") data-dir2)))
      (setf *current-language* 'english)
      (read-lex-file-aux
         (merge-pathnames (make-pathname :name "misc.lex") data-dir2) t)
      (read-lex-file-aux
         (merge-pathnames (make-pathname :name "verb.lex") data-dir2) nil)
      (setf *current-language* 'dutch)
      (read-lex-file-aux
         (merge-pathnames (make-pathname :name "dutch.lex") data-dir1) nil)
      (read-psort-file-aux
         (merge-pathnames (make-pathname :name "verb.p-sorts") data-dir2))
      (read-grammar-file-aux
         (merge-pathnames (make-pathname :name "grammar.rules") data-dir2))
      (read-lex-rule-file-aux
         (merge-pathnames (make-pathname :name "lex.rules") data-dir2))
      (setf *source-language* 'english)
      (setf *target-language* 'dutch)
      (read-tlink-file-aux
         (merge-pathnames (make-pathname :name "e-dutch.tlink") data-dir1))
```

```
(setf *source-language* 'english)
(setf *target-language* 'dutch)
(read-tlink-rule-file-aux
   (merge-pathnames (make-pathname :name "demo.trule") data-dir1))
(collect-all-lexical-items)
nil)
```

One point to note is that the globals *current-language*, *source-language* and *target-language* have to be set in the script (normally these would be set interactively). The other point is the call to (collect-all-lexical-items) which has the effect of cacheing all the lexical entries and the results of applying lexical rules to lexical entries. This process is normally carried out when attempting to match a result of tlink application which does not have fully specified orthography with a lexical item. Expanding the lexicon in this way will be very slow for big lexicons and large numbers of lexical rules — it should not be necessary in most cases of tlink application but is included here because when it is needed it is preferable to do the expansion at the preliminary loading stage rather than for everything to stop for some minutes ...

Appendix D

The LKB Morphology System

The origin of the morphological analysis and generation system used here is described in more detail in Jones (1993). Previously, any morphographemic or morphophonemic alternations had to be described in the following somewhat crude manner, where the new orthography is given as a complex or stem and affix:

$$\begin{bmatrix} \textbf{lex-noun-sign} & \\ \text{ORTH} = \begin{bmatrix} \textbf{complex-orth} \\ \text{ORTH1} = \textbf{teach} \\ \text{ORTH2} = \textbf{er} \end{bmatrix} \\ \text{CAT} = \boxed{\textbf{noun-cat}} \\ \text{SEM} = \boxed{\textbf{obj-noun-formula}} \\ \text{QUALIA} = \boxed{\textbf{human}} \end{bmatrix}$$

$$\begin{bmatrix} \textbf{lex-noun-sign} & \\ \text{ORTH} = \begin{bmatrix} \textbf{complex-orth} \\ \text{ORTH1} = \textbf{rabbit} \\ \text{ORTH2} = \textbf{+s} \end{bmatrix} \\ \text{CAT} = \boxed{\textbf{noun-cat}} \\ \text{SEM} = \boxed{\textbf{obj-noun-formula}} \\ \text{QUALIA} = \boxed{\textbf{nomqualia}} \end{bmatrix}$$

The current system bypasses the restrictions of the unification process to allow orthographic modifications to take place outside the unification system. The morphological rules used will be hybrids, to a certain extent, since they must contain some degree of feature-based unification, — in order to modify syntactic and semantic features associated with the morphological change, — and they must contain some data encoding how the orthographic modification is to proceed. Thus as well as such components as name, unification-related information and position of affix attachment (suffix, prefix etc.), the rule will contain a set of subrules in the form of matched simple partial regular expressions. The first of each pair of expressions will match a set of letters at the crucial site of affix attachment in the stem (word-end, word-beginning, word-middle) and the second expression describes the set of letters to replace them with to form the morphologically complex form. Thus the following might form a part of the past-tense rule in English:

```
past-tense  suffix  (* ed) (e ed) (y ied) ...
```

In this rule, the asterisk represents a null character, so that the first sub-rule is the default (*ed* is added at the word-final position). The rest of the sub-rules represent exceptions, so that if e.g. the root form ends with *e* then that *e* is replaced by *ed*. Since the final rules (see section D.3) are structured in such a manner that the exception sub-rules override the default, this prevents the formation of morphological complexes such as *hateed*, or (since the ultimate example of an exception sub-rule is one where a particular stem has a totally non-regular complex form) *gived*.

D.1 LKB Considerations

Bearing in mind the similarity between the LKB's lexical rules and the proposed morphological rules, the morphological variant is best included in all the places that the lexical rules are used. The key functional locations are the following:

D.1.1 Parsing

It should be possible for words that are outside the LKB lexicon due to morphological complexity to be included in a phrase for parsing, and have feature structures built for these words from the lexical entries of their stem morphemes. Thus the flexibility of the parser is much enhanced, and the lexicon can be permitted to stay smaller. The effect on the final parse should be much as the effect of the application of lexical rules. That is that there should be an edge in the chart for the result and argument of each rule that is applied (morphological rules included). Thus if the phrase "John sleeps" is to be parsed, both *sleep* and *sleeps* should appear in the chart.

One problem that is raised by this is the interaction between morphological and lexical rules. When analysing a morphologically complex word, to first apply all the morphological operations, and then apply all the lexical ones, or vice versa will result in certain interpretations of the word being lost[1].

Therefore lexical and morphological operations must have the potential to be totally interleaved, so that they can be applied in whichever order suits the unification constraints. As a result of this, it may be the case that lexical rules must be formulated in a rather more strict fashion to avoid over-generation.

D.1.2 Generation

When the menu options to apply lexical rules to psorts are chosen in the LKB menu, they should also have the ability to apply morphological rules. Thus the option to apply a particular rule should check whether that rule is morphological, and apply the appropriate orthographic transformation on the resultant feature-structure given by the unification procedure, and the function of applying all lexical rules to a psort should exhibit similar behaviour. Obviously with the latter possibility it should be possible to interleave application of morphological and lexical rules when there is the possibility of applying more than one rule to a psort.

D.1.3 Translation

In the same way that lexical rules are translated through the tlink rule mechanism, morphological rules should have translation equivalents in other languages so that correct translations can be obtained if a morphologically complex form is to be translated — that

[1]See Jones (1993) for a more complete explanation.

is, the appropriate morphological rule is applied to the translation of the stem-form of the complex word in the source language.

Therefore it seems that the best way to implement morphology in the LKB is as a form of lexical rule. Indeed, the lexical rule mechanism is admirably suited to that facet of the morphological manipulation requiring default unification — namely the specification of various features in the syntax and semantics of the stem and final forms[2]. A morphological rule must how ever be distinctively different to a lexical rule, so that it can be clearly distinguished as one. This is obviously necessary so that the extra process of orthographic modification can be implemented correctly.

D.2 Implementation of LKB morphology

Morphological rules are implemented in this system as a subtype of lexical rule. Thus they are similar enough to lexical rules for standard LKB unification operations to work, and yet distinctive enough for the orthographic modification to be able to operate. The morphological rules (whose structure is described in a later section) are read in by the LKB twice: once by the lexical rules mechanism, which deals with the information suitable for unification processing, and once by a mechanism that uses a different section of the rules to build the double rooted tree, described earlier, for use in the generation/analysis stages.

Operations involving generation of morphologically complex forms are accomplished in a fairly simple manner. The code is modified to include a check for a morphological rule. If such a one is found, then the unification function is called with an additional value for the new orthography; if unification fails nothing happens, but if it is successful the new orthography is inserted into the new feature structure before it is returned to the calling function.

Parsing is a little more difficult since the interleaving of lexical and morphological rules and possibility of many different stems must be taken into consideration. Morphological analysis is attempted on every word in the parse and the root morphemes and intermediate forms (through morphological or lexical modification) are added to the chart. Then if the word is also a valid lexical entry it is processed normally; if not, then normal processing is inhibited, so that the 'word unknown' report is not generated. If, however, there are no morphological possibilities for the word, then normal processing is forced, whether the word is lexical or not. This is so that the appropriate error message can be generated if the word does not occur in the lexicon and is not the morphological complex of a lexical stem.

The morphological analysis is processed one step at a time, and between each morpheme addition all possible combinations of lexical rules are tried. All the resulting edges are stored in a temporary list, the whole of which is added to the chart if and only if all rule applications unify properly and there is a lexical entry for the stem form. This does of course cause problems, since the subordinate edges to the morphologically complex word in the parse only exist in a theoretical sense within the structure of the parse. We have to make sure that these edges cannot be picked up and spanned by rule applications higher up in the parse. To ensure that this happens, the subordinate edges are all made zero length, which means that they are not considered by the rest of the parsing mechanism,

[2] In the case of *plural* for instance, the argument must be a singular count noun, and the result a plural one.

Figure D.1: Parse tree for *Mary eats the nuts*

but still appear in the final output of the parse. So, for example, parsing "Mary eats the nuts" results in the parse tree shown in figure D.1 — note the fact that in the parse tree the orthography of the stem lexeme is given at the leaf nodes. This reinforces the fact that they are a notional component of the real parse.

D.3 Rule structure

Since, by the description above, the same rule must be read twice by two different input mechanisms (lexical rule and morphological data), the two different sets of information associated with each rule must each be hidden from each other. This is easily accomplished through the use of a special character, which is treated by the lexical rule input mechanism as a comment, yet indicates to the morphological data compiler that the current line contains information relevant to it. The following rule, for noun pluralisation, illustrates this.

```
plural morph-rule
 %suffix (!s !ss) (!ss !ssses) (ss sses) (!ty !ties) (f ves) (tch tches) (sh shes)
 % (person people) (man men) (formula formulae) (goose geese) (fish fish)
 % (child children) (sheep sheep) (chicken chicken) (mouse mice) (louse lice)
   < 1 > = lex-count-noun
   < 1 : cat : m-feats : agr : num > = sg
   < 0 : cat : m-feats : agr : num > = pl
   < 0 > = lex-count-noun
   < 1 : sem > = < 0  : sem >
   < 1 : qualia > =  < 0 : qualia  >.
```

The portions following the % sign at the start of the line form the sub-rules relevant to the morphological compiler. The name of the rule — *plural* in this case — is extracted from the header of each rule. The first word after the header — *suffix* in this case — indicates which data structure to compile the current rule into. The *!* structure is used to denote special macro symbols internal to the morphological system. Use and justification for these is provided more fully in Jones (1993), but briefly they each expand to a set of letters that could possibly replace them. These sets are also defined in the file with the morphological rules, as follows:

```
%(letter-set (!s abcdefghijklmnopqrtuvwxyz))
```

They must, obviously, be defined prior to their first use in the rule file, and only one particular macro symbol may be used in each sub-rule.

Of course, this rule structure is slightly untidy, in that all the exceptions to the productive morphological processes must be listed explicitly. This has only been implemented in such a fashion for the sake of simplicity, and since the testing lexicon in the LKB is rather small. In a larger system one could envisage features for irregular morphology being included in the lexical entries for stems. Hence the presence of an irregular morphology for a particular rule would block the application of that particular rule to the stem, and instead the morphologically complex form would be taken from a feature in the lexical entry.

The sub-rules af a morphological rule should be constructed in such a fashion that there is as little ambiguity as possible. Naturally, in the process of analysis, ambiguity is not really an issue since multiple possibilities are supported, and invalid stems are dropped when checked with the lexicon. However, it is the process of generation which requires care in sub-rule formulation. Since we only really want one possible morphological complex of every stem (blocking considerations aside) we must ensure that only one sub-rule is applied to the stem. Since the software implementing the morphological system enforces this by matching as long a sub-rule to the stem as possible, and only using the longest one that matches, sub-rules must be written so that only the most precise applies to a given stem. What this means in practice is that the subrule `(mouse mice)` will apply to the word *dormouse* rather than one such as `(e es)`.

D.4 Functionality

This morphological system integrates well with the LKB, and enhances its functionality. Examples with the parsing function have already been shown in the previous section, and examples of use in the other two functions amount to little more than the display of some feature structures.

In the current system, there is no facility to tackle compounding phenomena, unless a specific rule is written for each compound — one could, for example, envisage the rule in figure D.2 being written to cope with the animal grinding phenomenon in Dutch, and the figure also shows a feature structure of the translation of *chicken* (meat) into Dutch using this rule, via a translation rule linking the English ANIMAL-GRINDING rule to the Dutch VLEESCOMPOUNDING one. This is very important since it illustrates just how closely morphological and lexical rules are related, since the translation link rule in this case is between a lexical and morphological rule, rather than between two rules of the same type.

$$
\begin{bmatrix}
\textbf{lex-noun-sign} & \\
\text{ORTH} = \textbf{kippevlees} & \\
\text{CAT} = \boxed{\textbf{noun-cat}} & \\
\text{SEM} = \boxed{\textbf{unary-formula-entity-arg1}} & \\
\text{SENSE-ID} = \boxed{\textbf{sense-id}} & \\
\text{QUALIA} = & \begin{bmatrix}
\textbf{c\_natural} \\
\text{AGENTIVE} = \boxed{\textbf{agentivestuff}} \\
\text{TELIC} = \boxed{\textbf{verb-sem}} \\
\text{FORM} = \begin{bmatrix} \textbf{nomform} \\ \text{RELATIVE} = \textbf{mass} \end{bmatrix} \\
\text{CONSTITUENCY} = \boxed{\textbf{nomconst}} \\
\text{OBJECT-INDEX} = \boxed{0}\ \textbf{entity} \\
\text{PROPERTIES} = \boxed{\textbf{phys-properties}}
\end{bmatrix}
\end{bmatrix}
$$

```
vleescompounding morph-rule
%suffix (p ppevlees) (m mvlees)
     < 1 : sense-id : language > = Dutch
     < 1 : qualia > = animal
     < 1 : cat : m-feats : agr : num > = sg
     < 0 : qualia : form : relative > = mass
     < 0 : qualia > = c_natural .
```

Figure D.2: An illustration of the Dutch version of *animal-grinding*

Appendix E

An annotated Bibliography of ACQUILEX Papers Connected with the LKB

Ageno, A., S. Cardoze, I. Castellon, M.A. Marti, G. Rigau, H. Rodriguez, M. Taule and M.F. Verdejo (1991) *An Environment for the Management and Extraction of Taxonomies from On-line Dictionaries.*
ESPRIT BRA-3030 ACQUILEX WP NO.20

Ageno, A., I. Castellon, M.A. Marti, G. Ribas, G. Rigau, H. Rodriguez, M. Taule and M.F. Verdejo (1991) *The Extraction of Semantic Information from MRDs.*
ESPRIT BRA-3030 ACQUILEX WP NO.28

Ageno, A., I. Castellon, M.A. Marti, G. Ribas, G. Rigau, H. Rodriguez, M. Taule and M.F. Verdejo (1991) *SEISD: User Manual. Guide to the Extraction and Conversion of Taxonomies.*
ESPRIT BRA-3030 ACQUILEX WP NO.30

Ageno, A., I. Castellon, M.A. Marti, G. Ribas, G. Rigau, H. Rodriguez, M. Taule and M.F. Verdejo (1992) *A semi-automatic process to create LKB entries.*
ESPRIT BRA-3030 ACQUILEX WP NO.38

Ageno, A., I. Castellon, M.A. Marti, G. Ribas, G. Rigau, H. Rodriguez, M. Taule and M.F. Verdejo (1992) *From LDB to LKB.*
ESPRIT BRA-3030 ACQUILEX WP NO.39

Ageno, A., I. Castellon, G. Rigau, H. Rodriguez, M.F. Verdejo, M.A. Marti and M. Taule (1992) 'SEISD: An Environment for Extraction of Semantic Information from On-Line Dictionaries', *Proceedings of the 3rd Conference on Applied Natural Language Processing (ANLP-92),* Trento, Italy, pp. 253–255.

This group of papers discuss in detail the environment developed in Barcelona for extracting information from MRDs and converting them to LKB entries.

Briscoe, E. (1991) 'Lexical Issues in Natural Language Processing' in E. Klein and F. Veltman (eds.), *Natural Language and Speech,* Springer-Verlag, pp. 39–68.
ESPRIT BRA-3030 ACQUILEX WP NO.41

This paper contains an outline of the ACQUILEX project as a whole, and discusses some issues such as the distinction between a lexical database (LDB) and a LKB and what it means for a lexicon to be reusable which were important considerations in designing the LKB.

Briscoe, E., A. Copestake and B. Boguraev (1990) 'Enjoy the Paper: Lexical Semantics via Lexicology', *Proceedings of the 13th Coling,* Helsinki, pp. 42–47.
ESPRIT BRA-3030 ACQUILEX WP NO.11

Discusses the representation of logical metonymy in a system which was essentially a first prototype for the LKB.

Briscoe, E. and A. Copestake (1991) 'Sense Extensions as Lexical Rules', *Proceedings of the IJCAI Workshop on Computational Approaches to Non-Literal Language,* Sydney, Australia, pp. 12–20.
ESPRIT BRA-3030 ACQUILEX WP NO.22

This paper is very similar to Copestake and Briscoe (1992) but also discusses how MRDs represent some sense extensions.

Briscoe, E. J., A. Copestake and A. Lascarides (in press) 'Blocking' in P. St Dizier and E. Viegas (eds.), *Computational lexical semantics,* Cambridge University Press.
ESPRIT 7315 ACQUILEX II WP NO.2

This discusses why the LKB's approach to the representation of defaults and lexical rules is (at least in some respects) inadequate, as are other existing representation languages.

Calzolari, N. (1991) 'Acquiring and Representing Semantic Information in a Lexical Knowledge Base', *Proceedings of the Proceedings of ACL SIGLEX Workshop on Lexical Semantics and Knowledge Representation,* Berkeley, California, pp. 188–198.
ESPRIT BRA-3030 ACQUILEX WP NO.16

A very general paper which discusses how MRDs can be used to determine what should go in to a lexical knowledge base, based on determining the attributes which lexicographers find important enough to mention in the definitions of a particular class of words.

Calzolari, N., J. Hagman, E. Marinai, S. Montemagni, A. Spanu and A. Zampolli (1993) 'Encoding Lexicographic Definitions as Typed Feature Structures' in F. Beckmann and G. Heyer (ed.), *Theorie und Praxis des Lexikons,* Walter de Gruyler, Berlin, New York, pp. 274–315.

This paper surveys the overall process of extracting lexical and conceptual knowledge contained in machine-readable dictionaries and of representing it in a formal and consistent way within a lexical knowledge base. In particular, it gives an overview of the main lines of research undertaken in Pisa within the Acquilex-I project. As far as representation is concerned, it reports on Pisa's experience with the encoding of dictionary entries using the Typed Feature Structure formalism, pointing out and evaluating the advantages and disadvantages that have emerged.

Copestake, A. (1990) 'An approach to building the hierarchical element of a lexical knowledge base from a machine readable dictionary', *Proceedings of the Workshop on Inheritance in Natural Language Processing,* Tilburg, The Netherlands, pp. 19–29.
ESPRIT BRA-3030 ACQUILEX WP NO.8

Discusses the construction of taxonomies and some ideas about their representation. Largely superceded by Copestake (1992).

Copestake, A. (1992) 'The ACQUILEX LKB: Representation Issues in Semi-automatic Acquisition of Large Lexicons', *Proceedings of the 3rd Conference on Applied Natural Language Processing (ANLP-92),* Trento, Italy, pp. 88–96.
ESPRIT BRA-3030 ACQUILEX WP NO.36

This paper gives an overview of the LKB and its use in ACQUILEX. To a large extent the current work supercedes it.

Copestake, A. (1992) 'The Representation of Group Denoting Nouns in a Lexical Knowledge Base', *Proceedings of the Second Seminar on Computational Lexical Semantics,* Toulouse, France, pp. 106–118.
ESPRIT BRA-3030 ACQUILEX WP NO.37

Describes the use of the LKB in representing group denoting nouns and the extent to which the information can be extracted from MRDs.

Copestake, A. (1993) *Constraints, tlinks and MT.* University of Cambridge Computer Laboratory.
ESPRIT 7315 ACQUILEX II WP NO.??

Describes the translation link mechanism. Mostly incorporated in the current document.

Copestake, A. and E. J. Briscoe (1992) 'Lexical operations in a unification based framework' in J. Pustejovsky and S. Bergler (eds.), *Lexical Semantics and Knowledge Representation. Proceedings of the first SIGLEX Workshop, Berkeley, CA, 1991,* Springer-Verlag, Berlin, pp. 101–119.
ESPRIT BRA-3030 ACQUILEX WP NO.21

Describes the LKB lexical rule mechanism and its use in the representation of sense extensions or regular polysemy.

Copestake, A., B. Jones, A. Sanfilippo, H. Rodriguez, P. Vossen, S. Montemagni and E. Marinai (1992) *Multilingual Lexical Representation.*
ESPRIT BRA-3030 ACQUILEX WP NO.43

Describes tlinks and how they might be extracted semi-automatically from bilingual dictionaries.

Copestake, A. and A. Sanfilippo (1993) 'Multilingual lexical representation', *Proceedings of the AAAI Spring Symposium on Building lexicons for machine translation,* Stanford, CA.
ESPRIT 7315 ACQUILEX II WP NO.3

Discusses the use of the tlinks to represent generalisations about translation mismatches etc.

Östling, A. (1991) *Sense Extensions in the Italian Food Subset.*
ESPRIT BRA-3030 ACQUILEX WP NO.24

In this paper, the results of a study of the metaphoric and metonymic sense extensions for the Italian Food Subset are presented, along with a brief discussion of the examples of derivation and conversion encountered. The study is based on "Il Nuovo Dizionario Italiano Garzanti". The model for lexical rules proposed by Briscoe and Copestake has been applied to the Italian data.

de Paiva, V. (in press) 'Types and constraints in the LKB' in E. J. Briscoe, A. Copestake and V. de Paiva (ed.), *Inheritance, defaults and the lexicon,* Cambridge University Press, Cambridge, England.

A formal/mathematical paper about feature structures and typing.

Sanfilippo, A. (in press) 'LKB encoding of lexical knowledge from machine-readable dictionaries' in E. J. Briscoe, A. Copestake and V. de Paiva (ed.), *Inheritance, defaults and the lexicon,* Cambridge University Press, Cambridge, England.

Describes the verb type system.

Sanfilippo, A. (1993) *Verbal Diatheses: Knowledge Acquisition, Lexicon Construction and Dictionary Compilation.* University of Cambridge Computer Laboratory.
ESPRIT 7315 ACQUILEX II WP NO.1

Discusses the acquisition of information about verbal diathesis alternation from MRDs and the representation in the LKB.

Sanfilippo, A. and V. Poznanski (1992) 'The Acquisition of Lexical Knowledge from Combined Machine-Readable Sources', *Proceedings of the 3rd Conference on Applied Natural Language Processing (ANLP-92),* Trento, Italy, pp. 80–88.
ESPRIT BRA-3030 ACQUILEX WP NO.35

Describes how detailed lexical entries may be constructed by combining information from two MRDs.

Sanfilippo, A., E. Briscoe, A. Copestake, M.A. Marti and A. Alonge (1992) 'Translation Equivalence and Lexicalization in the ACQUILEX LKB', *Proceedings of the 4th International Conference on Theoretical and Methodological Issues in Machine Translation (TMI-92),* Montreal, Canada.
ESPRIT BRA-3030 ACQUILEX WP NO.42

Gives a detailed analysis of some difficult MT examples using the LKB tlink mechanism.

Spanu, A. (1992) *Extending the Type System from the Food-Subset to the Place-Subset.*
ESPRIT BRA-3030 ACQUILEX WP NO.51

A possible way to evaluate a type system designed to represent the information contained in a given semantic subset is to test its validity for another subset. In this way it is possible to see which system components (types, attributes, attribute values) can be reused as they are, being valid for both the subsets, which can be reused but need some modifications (small or big deplacements), which are superfluous for the new subset and especially what needs to be added. According to this requirement, the type system designed for representing the food-subset has been tested on the place-subset. In the latter we analysed particularly the definitions of lexical entries which refer to places associated to food i.e. "agrumeto" (citrus plantation), "pasticceria" (confectioner's), "pastificio" (pasta factory). We present here the different steps of the type system extension process, from taxonomy building to a final evaluation.

Vossen, P. (1991) *Converting Data from a Lexical Database to a Knowledge Base.*
ESPRIT BRA-3030 ACQUILEX WP NO.27

Vossen, P. (1992) 'An Empirical Approach to Automatically Constructing a Knowledge Base from Dictionaries', *Proceedings of the 5th EURALEX,* Tampere, Finland.

These two papers give the most detailed published analysis of the issues involved in trying to construct a detailed lexical semantic representation on the basis of MRDs.

Vossen, P. (1993) *Extracting equivalence relations for a multilingual Lexical knowledge base.* University of Amsterdam.
ESPRIT 7315 ACQUILEX II WP NO.??

Extraction of tlinks for food and drinks using bilinguals via software developed in Amsterdam based on principles of TGE.

Vossen, P. (1993) *Grammatical and conceptual individuation of nouns in the lexicon..* University of Amsterdam.
ESPRIT 7315 ACQUILEX II WP NO.??

Development of a TYPE system for classes of nouns in which grammatical and conceptual properties are correlated. Extraction of grammatical properties via this correlations for nouns in VanDale.

Vossen, P. and A. Copestake (in press) 'Untangling definition structure into knowledge representation' in E. J. Briscoe, A. Copestake and V. de Paiva (ed.), *Inheritance, defaults and the lexicon,* Cambridge University Press, Cambridge, England.

Describes how various categories of noun definition in LDOCE give rise to different types of relationships between word senses and how this may be represented in the LKB.

References

Alshawi, H., D. Carter, M. Rayner and B. Gambäck (1991) 'Translation by quasi logical form transfer', *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics (ACL-91),* Berkeley, California, pp. 161–168.

van den Berg, M. and H. Prüst (1991) 'Common denominators and default unification', *Proceedings of the Computational Linguistics in the Netherlands (CLIN), First Meeting,* Utrecht, pp. 1–16.

Bird, S. (1992) 'Finite-state phonology in HPSG', *Proceedings of the 15th International Conference on Computational Linguistics (COLING-92),* Nantes, France, pp. 74–80.

Blackburn, P., C. Gardent and W. Meyer-Viol (1993) 'Talking about trees', *Proceedings of the 6th Conference of the European Chapter of the Association for Computational Linguistics (EACL-93),* Utrecht, the Netherlands, pp. 21–29.

Bouma, G. (1990a) 'Defaults in unification grammar', *Proceedings of the 28th Annual Meeting of the Association for Computational Linguistics (ACL-90),* Pittsburgh, USA, pp. 165–173.

Bouma, G. (1990b) 'Non-monotonic inheritance and unification', *Proceedings of the First International Workshop on Inheritance in Natural Language Processing,* Tilburg, The Netherlands, pp. 1–8.

Brachman, R. J. and J. G. Schmolze (1985) 'An overview of the KL-ONE knowledge representation language', *Cognitive Science,* **9**, 171–216.

Briscoe, E. J., A. Copestake and V. de Paiva (eds) (1991) 'Proceedings of the ACQUILEX workshop on default inheritance in the lexicon', Technical Report No 234, University of Cambridge, Computer Laboratory.

Briscoe, E. J., A. Copestake and V. de Paiva (eds) (in press) *Inheritance, defaults and the lexicon,* Cambridge University Press, Cambridge, England.

Briscoe, E. J., A. Copestake and A. Lascarides (in press) 'Blocking' in P. St Dizier and E. Viegas (ed.), *Computational lexical semantics,* Cambridge University Press.

Cahill, L. J. and R. Evans (1990) 'An Application of DATR: the TIC Lexicon', *Proceedings of the 9th European Conference on Artificial Intelligence (ECAI-90),* Stockholm, pp. 120–125.

Calder, J. (1987) 'Typed unification for natural language processing' in E. Klein and J. van Benthem (ed.), *Categories, polymorphism and unification,* Centre for Cognitive Science, University of Edinburgh, Edinburgh, pp. 65–72.

Calder, J. (1991) 'Some notes on priority union', Paper presented at the ACQUILEX Workshop on Default Inheritance in the Lexicon, Cambridge.

Carpenter, R. (1990) 'Typed feature structures: inheritance, (in)equality and extensionality', *Proceedings of the First International Workshop on Inheritance in Natural Language Processing,* Tilburg, The Netherlands, pp. 9–18.

Carpenter, R. (1991) 'Skeptical and credulous default unification with applications to templates and inheritance' in E. J. Briscoe, A. Copestake and V. de Paiva (ed.), *ACQUILEX Workshop on Default Inheritance in the Lexicon,* Technical Report No 234, University of Cambridge, Computer Laboratory.

Carpenter, R. (1992) *The logic of typed feature structures,* (Tracts in Theoretical Computer Science), Cambridge University Press, Cambridge, England.

Carpenter, R. (in press) 'Skeptical and credulous default unification with applications to templates and inheritance' in E. J. Briscoe, A. Copestake and V. de Paiva (ed.), *Defaults, Inheritance and the Lexicon,* Cambridge University Press, Cambridge, England.

Copestake, A. (1992) 'The representation of lexical semantic information', D.Phil thesis, University of Sussex, Cognitive Science Research Paper, CSRP 280.

Copestake, A. (1993) 'Constraints, tlinks and MT', ACQUILEX Working Paper.

Copestake, A. and E. J. Briscoe (1992) 'Lexical operations in a unification based framework' in J. Pustejovsky and S. Bergler (ed.), *Lexical Semantics and Knowledge Representation. Proceedings of the first SIGLEX Workshop, Berkeley, CA, 1991,* Springer-Verlag, Berlin, pp. 101–119.

Copestake, A. and A. Sanfilippo (1993) 'Multilingual lexical representation', *Proceedings of the AAAI Spring Symposium on Building lexicons for machine translation,* Stanford, CA.

Copestake, A. and A. Sanfilippo (in preparation) 'Representing generalisations about translation mismatches', ACQUILEX II working paper.

Emele, M. and R. Zajac (1990) 'Typed unification grammars', *Proceedings of the 13th International Conference on Computational Linguistics (COLING-90),* Helsinki, pp. 293–298.

Emele, M., U. Heid, S. Momma and R. Zajac (1992) 'Interactions between linguistic constraints: procedural vs. declarative approaches', *Machine Translation,* **7 (1–2)**, 61–98.

Estival, D., A. Ballim, G. Russell and S. Warwick (1990) 'A syntax and semantics for feature structure transfer', *Proceedings of the 3rd International Conference on theoretical and methodological issues in MT of NLs (TMI-90),* Austin,Texas, pp. 131–143.

Evans, R. and G. Gazdar (1990) 'The DATR Papers. Cognitive Science Research Paper CSRP 139', School of Cognitive and Computing Sciences, University of Sussex.

Flickinger, D. P. (1987) 'Lexical rules in the hierarchical lexicon', PhD thesis, Stanford University.

Gazdar, G. (1987) 'Linguistic applications of default inheritance mechanisms' in P. Whitelock, H. Somers, P. Bennett, R. Johnson, and M. McGee Wood (ed.), *Linguistic theory and computer applications,* Academic Press, London, pp. 37–68.

Gazdar, G. (1990) 'An introduction to DATR' in R. Evans and G. Gazdar (ed.), *The DATR Papers. Cognitive Science Research Paper CSRP 139,* School of Cognitive and Computing Sciences, University of Sussex, pp. 1–14.

Jones, B. (1993) 'Integrating Morphology into the LKB', ACQUILEX Working Paper.

Kaplan, R. (1987) 'Three seductions of computational psycholinguistics' in P. Whitelock, H. Somers, P. Bennett, R. Johnson, and M. McGee Wood (ed.), *Linguistic Theory and Computer Applications,* Academic Press, London, pp. 149–188.

Kasper, R. T. and W. C. Rounds (1986) 'A logical semantics for feature structures', *Proceedings of the 24th Annual Conference of the Association for Computational Linguistics (ACL-86),* Columbia University, pp. 235–242.

Kasper, R. T. and W. C. Rounds (1990) 'The logic of unification in grammar', *Linguistics and Philosophy,* **13 (1)**, 35–58.

Kay, M. (1979) 'Functional Grammar', *Proceedings of the Fifth Annual Meeting of the Berkeley Linguistic Society,* Berkeley, pp. 142–158.

Kay, M. (1984) 'Functional unification grammar: a formalism for machine translation', *Proceedings of the 10th International Conference on Computational Linguistics (COLING-84),* Stanford, California, pp. 75–78.

Kilbury, J., P. Naerger, and I. Renz (1991) 'DATR as a lexical component for PATR', *Proceedings of the 6th Conference of the European Chapter of the Association for Computational Linguistics (EACL-91),* Berlin.

Krieger, H-U. and J. Nerbonne (in press) 'Feature-based inheritance networks for computational lexicons' in E. J. Briscoe, A. Copestake and V. de Paiva (ed.), *Inheritance, defaults and the lexicon,* Cambridge University Press, Cambridge, England.

Moens, M., J. Calder, E. Klein, M. Reape, and H. Zeevat (1989) 'Expressing generalizations in unification-based grammar formalisms', *Proceedings of the 4th Conference of the European Chapter of the Association for Computational Linguistics (EACL-89),* Manchester, England, pp. 66–71.

Östling, A. (1991) *Sense Extensions in the Italian Food Subset.*
ESPRIT BRA-3030 ACQUILEX WP NO.24

de Paiva, V. (in press) 'Types and constraints in the LKB' in E. J. Briscoe, A. Copestake and V. de Paiva (ed.), *Inheritance, defaults and the lexicon,* Cambridge University Press, Cambridge, England.

Pereira, F. C. N. and S. M. Shieber (1984) 'The semantics of grammar formalisms seen as computer languages', *Proceedings of the 10th International Conference on Computational Linguistics (COLING-84),* Stanford, California, pp. 123–129.

Pollard, C. and I. Sag (1987) *An information-based approach to syntax and semantics: Volume 1 fundamentals,* CSLI Lecture Notes 13, Stanford CA.

Pollard, C. and I. Sag (in press) *Head-driven phrase structure grammar,* Chicago University Press, Chicago.

Russell, G., J. Carroll and S. Warwick (1991) 'Multiple default inheritance in a unification-based lexicon', *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics (ACL-91),* Berkeley, California, pp. 215–221.

Saint-Dizier, P. and E. Viegas (1992) *Proceedings of the second seminar on computational lexical semantics,* IRIT, Toulouse, France, forthcoming with Cambridge University Press, England.

Sanfilippo, A. (in press) 'LKB encoding of lexical knowledge from machine-readable dictionaries' in E. J. Briscoe, A. Copestake and V. de Paiva (ed.), *Inheritance, defaults and the lexicon,* Cambridge University Press, Cambridge, England.

Sanfilippo, A. and V. Poznanski (1992) 'The acquisition of lexical knowledge from combined machine-readable sources.', *Proceedings of the 3rd Conference on Applied Natural Language Processing (ANLP-92),* Trento, Italy, pp. 80–88.

Sanfilippo, A., E. J. Briscoe, A. Copestake, M. A. Marti and A. Alonge (1992) 'Translation equivalence and lexicalization in the ACQUILEX LKB', *Proceedings of the 4th International Conference on Theoretical and Methodological Issues in Machine Translation (TMI-92),* Montreal, Canada.

Shieber, S. M. (1986) *An introduction to unification-based approaches to grammar,* CSLI Lecture Notes 4, Stanford CA.

Shieber, S.M. (1993) 'The problem of logical form equivalence', *Computational Linguistics,* **19 (1)**, 179–190.

Shieber, S. M., H. Uszkoreit, F. C. N. Pereira, J. Robinson and M. Tyson (1983) 'The formalism and implementation of PATR-II', Research on Interactive Acquisition and Use of Knowledge, volume 1894 of SRI Final Report. SRI International, Menlo Park, California.

Whitelock, P. (1992) 'Shake-and-bake translation', *Proceedings of the 14th International Conference on Computational Linguistics (COLING-92),* Nantes, France.

Wilks, Y. (1975) 'A preferential pattern-seeking semantics for natural language inference', *Artificial Intelligence,* **6**, 53–74.

Zajac, R. (1991) 'Issues in the design of a language for representing linguistic information based on inheritance and feature structures' in E. J. Briscoe, A. Copestake and V. de Paiva (ed.), *Proceedings of the ACQUILEX Workshop on Default Inheritance in the Lexicon,* University of Cambridge Computer Laboratory, Technical Report No. 238.

Zajac, R. (in press) 'Issues in the design of a language for representing linguistic information based on inheritance and feature structures' in E. J. Briscoe, A. Copestake and V. de Paiva (ed.), *Inheritance, defaults and the lexicon,* Cambridge University Press, Cambridge, England.