IBL
An Inheritance-Based Lexicon Formalism

Sven Hartrumpf

Artificial Intelligence Center
The University of Georgia
Athens, Georgia
June 1994

Abstract

This paper describes a lexicon formalism (IBL for inheritance-based lexicon) that uses multiple default inheritance to state generalizations and exceptions in a natural and efficient way. The formalism is inspired by the lexicon formalism in ELU (Environnement Linguistique d'Unification). However, IBL contains important extensions and modifications: 1. use of typed feature structures for intensive static error-checking and efficient implementation on the computer; 2. finer distinction between defeasible and non-defeasible information; 3. means for defining an interface to applications which use a lexicon; 4. special form of a letter tree to decrease the size of lexicons; 5. more readable syntax. The paper defines the syntax and semantics of IBL lexicons, provides examples of lexicon fragments, discusses the extensions with respect to ELU, compares IBL with other approaches that use inheritance, and gives an overview of the implementation of IBL in Prolog.
Keywords: lexicon, lexicon specification, inheritance, multiple default inheritance, unification, typed feature structures, Prolog.

Contents

1 Overview

This paper describes an inheritance-based lexicon formalism (IBL for short) that is derived from the lexicon formalism used in the ELU (Environnement Linguistique d'Unification). Russell, Ballim, Carroll, and Warwick-Armstrong (1992) present the ELU lexicon formalism in detail. Another paper of interest is written by Russell, Carroll, and Warwick-Armstrong (1990).[1]

[1]I want to thank Michael Covington for helpful comments and suggestions.

I modified and extended the formalism. This is a short summary of the most important changes:

1. IBL uses *typed feature structures* instead of untyped feature structures.
2. The distinction between defeasible and non-defeasible information is extended.
3. Information that is needed during lexicon processing, but isn't useful for the user of the lexicon, is clearly separated from the useful part of the information. This is accomplished by introducing the idea of a global and a local part of a feature structure type.
4. IBL uses a *letter tree* for quick access to the lexicon, while the ELU system uses an indexing scheme.
5. The syntax of the lexicon specification language is modified to make lexicons more readable to humans.

These changes will be discussed in detail in section 3.

Section 2 describes the original ELU lexicon formalism. Section 4 gives a formal definition of the syntax of the IBL lexicon specification language. The examples of class definitions in section 5 illustrate the contents of the preceding sections. Section 6 is intended as a user's guide. It shows how to use the different programs in the IBL system. In section 7, IBL is compared with other lexicon formalisms that use some kind of inheritance scheme. The final section 8 discusses some directions for further development of the IBL system.

2 The ELU Lexicon Formalism

The ELU lexicon formalism combines two different paradigms: unification and inheritance. The type of inheritance used, *multiple default inheritance*, is able to express adequately general statements and exceptions to such statements. Pure unification cannot accomplish this. Touretzky (1986) gives a thorough study of the problems associated with this kind of inheritance and a formalization of inheritance systems.

The most important area for using the combination of unification and inheritance is the lexicon. The ELU lexicon formalism is a part of the ELU system, which ISSCO, Geneva (Switzerland), is developing.

The lexicon is a hierarchy of classes in the form of a directed acyclic graph. The most specific classes represent single lexemes like the verb *love*, more general classes represent a set of words like the set of all transitive verbs or the set of all verbs. These classes could be written in ELU as:

```
#Word love (Transitive Verb)
  ⟨stem⟩ = love
```

```
#Class Transitive ()
  ⟨subcat⟩ = [Subj,Obj]
  ⟨Subj cat⟩ = np
  ⟨Obj cat⟩ = np

#Class Verb ()
  ⟨cat⟩ = v
  ⟨aux⟩ = no
```

Lexical classes begin with #Word, non-lexical classes with #Class. After the class name (love, Transitive, Verb) follows the *superclass declaration*; this is the list of all superclasses of the class. Variables in equations are capitalized, constants are not. Paths are notated in angle brackets.

Ambiguities that can be caused by multiple inheritance, when different superclasses supply contradicting values, are eliminated by introducing the notion of a total ordering on all superclasses of a class. The most specific class provides the information in case of an ambiguity. More specific classes appear to the left of more general classes in the superclass declaration of a class. For instance, Transitive is more specific to love than Verb is. If all indirect superclasses are included in this ordering, one receives the *class precedence list* (CPL) of a class. The CPL is derived by traversing the class hierarchy in a left-right, depth-first manner. The left most class in the CPL of a class C is the class C itself, as it is the most specific of all classes in its CPL.

A *default value* is a value for a feature that is valid, unless a different value is supplied in the definition of a class that is preceding the class containing the default value in the current CPL. This means that default values are defeasible. They are written immediately after the superclass declaration of a class in its *main equation set*.

There are values that cannot be overwritten; these are given in the *variant equation sets* of a class definition. These sets follow after the main equation set. The following example is the above mentioned non-lexical class Verb plus three variant equation sets:

```
#Class Verb ()
  ⟨cat⟩ = v
  ⟨aux⟩ = no
  |
  ⟨tense⟩ = past
  ⟨form⟩ = ⟨stem⟩ && ed
  |
  ⟨agr⟩ = sg3
  ⟨tense⟩ = present
  ⟨form⟩ = ⟨stem⟩ && s
  |
  ⟨agr⟩ = non_sg3
  ⟨tense⟩ = present
  ⟨form⟩ = ⟨stem⟩
```

Every variant equation set starts with a vertical bar. S = Prefix && Suffix unifies S with the result of concatenating Prefix and Suffix. Variants are mutually exclusive alternatives in a class that are on the same level in the hierarchy. For example, the third singular present tense form and the past tense form are both word forms of a verb.

Disjunction of atomic feature structures and negation of atomic feature structures or their disjunctions are allowed as feature values. The disjunction of the atomic values a and b is written as a / b; the negation of the atomic value a is written as ∼a.

The value of an attribute can be a list of feature structures. Lists are written as in Prolog. For example, in the main equation set of Transitive, the attribute subcat has the value [Subj,Obj], the list of two feature structures Subj and Obj.

To define the value of an attribute or to state constraints for several values, one can use a *macro*. For instance,

!umlaut(⟨morph bse stem⟩,⟨morph stem⟩)

calls the macro umlaut, which might be used to derive the stem ⟨morph stem⟩ from ⟨morph bse stem⟩ by changing the stem vowel to an *umlaut*.

Multiple inheritance is useful because the properties of a word can consist of two or more unrelated sets, such as morphological features and syntactic features. So one can distinguish classes for verbs that are morphologically similar, and classes for verbs that are syntactically similar. Then, a verb lexeme will have at least two superclasses: one concerning its morphology, and one concerning its syntax.

Multiple default inheritance helps to express linguistic data in a more compact and readable way, as one can make generalizations, if linguistically possible, and define exceptions, if linguistically necessary.

Unification can be defined for the information given in classes connected by a multiple default inheritance hierarchy, and this unification is tractable by computers; the result of unifying all pieces of information that belong to one lexical class L is the *global extension* of L. This unification starts by unifying the empty feature structure $\perp$ with the left most class in the CPL of L. If information in the main equation set contradicts the current feature structure, it will be ignored; if information in a variant equation set does so, unification with this specific variant will fail. The variant equation sets can introduce more than one feature structure as a result of unification. All of them must be traced when proceeding to the next class in the CPL. The result after processing all classes in the CPL is a non-empty set of feature structures or failure of the unification. For instance, the global extension of the above defined lexical class love consists of three feature structures. The one derived by using the second variant in the class Verb contains information from the lexical class love:

⟨stem⟩ = love

from the non-lexical class Transitive:

⟨subcat⟩ = [Subj,Obj]
⟨Subj cat⟩ = np
⟨Obj cat⟩ = np

from the main equation set of the non-lexical class Verb:

⟨cat⟩ = v
⟨aux⟩ = no

and from the second variant of the non-lexical class Verb:

⟨agr⟩ = sg3
⟨tense⟩ = present
⟨form⟩ = loves

The set of word forms that are admitted by the lexicon is the union of the global extensions of all lexical classes.

To lookup information in the lexicon, it is not necessary to perform an exhaustive search and generate all global extensions at run-time; instead ELU uses indexing for faster access. There is one index file for analysis that contains for every word form in the lexicon a pointer to its lexical class, and one index file for generation that contains for every name of a semantic relation in the lexicon a pointer to the corresponding lexical class. Therefore one can perform a binary search in the index to find the right lexical class; only the global extension of this class must be calculated during run-time.

3 The IBL Lexicon Formalism

In this section the extensions and modifications that lead from the ELU lexicon formalism to the IBL system are described and discussed.

The semantics of the language used to specify IBL lexicons is the same as for the ELU system, which is described in section 2, except for the extensions and modifications that the IBL system provides.

The only changes that influence the semantics of the IBL lexicon specification language greatly are the extension of the distinction between defeasible and non-defeasible information and the introduction of typed feature structures.

3.1 Defeasible and Non-defeasible Information

In an ELU lexicon, information that is given in a main equation set is defeasible, while the information in variant equation sets is not. Writing small lexicons, it becomes obvious that these two distinctions are not very convenient. Sometimes one wants to state a constraint that is not defeasible and not limited to a variant. Therefore IBL splits the main equation

	variant	non-variant
defeasible	—	default equation set
non-defeasible	variant equation set	main equation set

Table 1: The three different types of information in IBL

set of the ELU system into a *default equation set* and a *main equation set*. The default equation set corresponds to the main equation set in ELU, while the main equation set in IBL is something not found in ELU: non-defeasible information not limited to a variant. One can transform an IBL lexicon into an ELU lexicon regarding this extension. If E is the main equation set of a class C in IBL, one moves the equation set E to all variant sets in C. If there is no variant set, one must write a new one containing just the equations in E.

While this finer distinction doesn't extend the complexity of the formalism, it is a valuable tool for two reasons. First, a theoretical reason: it removes the urge upon the lexicon writer to make information defeasible that should not be defeasible or duplicate the information to all variant equation sets in the class definition. Second, a more practical reason: the extended distinction of the type of information encourages to make information non-defeasible and hence the system can detect more cases where a feature value is changed in a way the lexicon writer didn't intend.

The type of information that can be stated in a class definition can be distinguished with regard to the two dichotomies defeasible vs. non-defeasible and variant vs. main/non-variant information as Table 1 shows.

That there is no way to state defeasible, variant information makes sense, because the information given in a variant equation set defines a variant by describing its constituting properties. One would undermine these important distinguishing constraints, if one overrode them later. Therefore information in variants is never defeasible.

The definition of the global extension of the ELU system that Russell, Ballim, Carroll, and Warwick-Armstrong (1992, pp. 323–324) give must be revised, because information stated outside the variant equation sets can be defeasible or non-defeasible in IBL. Let ϕ be a feature structure that is processed during calculating the superclass extension of a lexical class with respect to the class C. If an equation in the main equation set of C cannot be unified with the feature structure ϕ, unification fails; if the same equation occurs in the default equation set of C, this equation is just ignored and unification can still succeed. The global extension of a lexical class is defined formally in the following section 3.2.

3.2 Typed Feature Structures

I introduce typed feature structures, because they have many advantages compared to (untyped) feature structures:

- Types enable the computer to perform a large amount of static error checking. For example, misspelled attributes or values and many kinds of false constraints can be detected. These errors, which are hard to find for humans, cannot be detected by the computer, when untyped feature structures are used, because almost every string of characters can be a valid attribute or value.
- Typed feature structures can be represented in programs more efficiently and more compactly than untyped feature structures. For example, one can use positional encodings that both reduce the amount of memory needed and the access time to specific feature values.
- Using meaningful types makes a lexicon more structured and hence more readable.
- In most case, the person who writes a lexicon doesn't think about all feature structures in the same way, but distinguishes different types. Therefore it is natural to support this way of thinking by the lexicon formalism.
- An application that uses feature structures calculated by another program, e.g. a parser using an IBL lexicon, will know which features will be specified and which won't, if the feature structures are typed. Types for feature structures provide a simple interface definition between different programs. If one adds comments to describe the semantics of the features mentioned in a feature structure type, this might suffice for a complete interface definition.

For more information about typed feature structures, see Carpenter (1992), who points out the first four of these advantages.

IBL uses two different kinds of types: *enumeration types* and *complex types*.

An *enumeration type* is defined by a finite set of values that must be valid Prolog terms. For example,

```
type num_type = { sg, pl }.
```

defines the type num_type to allow exactly the two values sg and pl, and provides a type that might be used as the type of the number feature in natural languages that distinguish only singular and plural forms.

A *complex type* is a set of pairs each consisting of an attribute name and a type name. For example,

```
type v_agr_type = (
  pers : pers_type,
  num : num_type).
```

defines the type v_agr_type which contains a feature called pers of type pers_type and a feature called num of type num_type. Complex types are normal types and can hence appear inside the definition of another complex type. So, one can construct arbitrarily complex types.

IBL has three *predefined types*: boolean_type, string_type, and general_type. The first type is an enumeration type with only two possible values, + and −, which one should use with the normal meaning of the two truth values in binary logic. Possible values for a feature of the type string_type are all Prolog strings, i.e. zero or more characters enclosed in double quotes. Sometimes information must be encoded in the lexicon that is not further processed in IBL, but is just passed to the user of the lexicon. This information can be a list or a set of typed feature structures and the like. IBL provides the type general_type to handle this kind of information. Every Prolog term is a valid value for a feature of this type.

Every class in IBL must indicate with which type of feature structures it should be associated. Therefore a class must either define its type or must have a CPL that contains exactly one class that defines its own type. A class that defines the type of a feature structure is called a *top class*, since this class must not have any superclasses and is hence at the top of the class hierarchy. An example of a top class is given in section 5. Note that the type of feature structures is split into a global and an optional local part, as discussed in section 3.3.

Feature structure types can be arbitrary except for the following constraints. All types must be defined, before they can be used. Feature names at one level of a type must be unique. The set of feature names in a top class definition and the set which contains all possible values of all enumeration types must be disjoint in order to avoid ambiguity. The feature conv must not appear in the type of any top class since it is reserved by IBL. There must be a feature form of the predefined type string_type in the global part of every feature structure type. This constraint is necessary, because the computer will create an index structure on the value of the feature form which is intended to contain the word form that the feature structure describes.

The introduction of types might seem to be too restrictive. Two tasks that will be necessary in a sophisticated lexicon can no longer be performed. First, to describe morphological derivation from one category to another, it is necessary to change the type of the feature structure used in calculating the global extension of a lexical class. In an untyped approach, this is easily accomplished by overriding the (default) information for a feature cat that contains the category of the word form described by the feature structure. Second, a set of information stated in a non-lexical class is sometimes so far generalized that it can be used for words of different categories. For example, if one defines classes for specific combinations of semantic casus (or theta-roles), e.g. *agens* and *theme*, it might be possible to generalize so that the classes are applicable to verbs and nouns, e.g. create and creation can be described as being combined with the same semantic casus, by adding the same class to their superclass list.

To allow these two tasks to be performed in a typed system like IBL, two type-safe methods can be applied: *converters*, i.e. a procedure that converts from one type to another copying only the relevant information, and *metaclasses*, i.e. classes over classes, in IBL a class whose superclass list contains only one element: a parameter or type variable, i.e. a variable whose domain is a set of types.

A converter specifies how to transform a feature structure ϕ_1 of type T_1 into a feature structure ϕ_2 of another type T_2. For instance,

```
converter part1_converter
   from
      v
   to
      a
   conditions
      a^stem_pos = v^form
.
```

states that a feature structure ϕ_2 of type a can be created by taking a feature structure ϕ_1 of type v and fulfilling the condition that the feature stem_pos in ϕ_2 and the feature form in ϕ_1 are to be unified.

To use a converter, a class definition must contain a value for the feature conv, which is implicitly a feature of type *general_type* in every feature structure type. For instance,

```
variant
   conv = part1_converter(a_part1),
   form = morph^prefix & morph^stem & "end",
   vform = pp
```

is a variant equation set that gives the converter part1_converter as the value of the special feature conv. After the name of the converter, the superclass list for the feature structure that the converter will create is given in parentheses. One can think of a converter as creating a new lexical class from the information in another class and specifying the direct superclasses of this new lexical class. This new class will be processed after the normal global extension has been generated.

Converters are much more than just a way to change the category of a feature structure in a typed approach: they are also capable of removing unnecessary and irritating information in generating the global extension of a lexicon, because the conditions in a converter specify which information is copied to or used in another way in the new feature structure. The rest of the old feature structure is discarded. An untyped approach ends up with a large set of unnecessary knowledge after a morphological derivation. In an example lexicon for German, the feature structure from which the adjectivized version of the present participle is calculated contains 23 feature values. All these features will be present in all feature structures for the derived adjective. But only one of them, the feature form, is ever needed! An untyped approach might develop a method to remove unnecessary features that are caused by morphological derivations, etc. However in a typed approach, the handling of this problem is naturally solved with the change of the type of feature structures. This is one of the many reasons suggesting that types are a powerful device in natural language processing.

The introduction of converters leads to a new definition of the global extension of a lexical class L. After calculating the global extension of L as defined in ELU, but including the distinction between the default and the main equation set, each feature structure ϕ in the global extension whose value of the feature conv is not the most general feature structure $\perp$ is removed from the global extension. The converter specified in the feature conv is applied

to ϕ giving a feature structure that one can think of as the result of applying a new lexical class to $\bot$, and the global extension is calculated starting with this new feature structure and using the superclasses given in the value of the feature conv to add information. The result is added to the general extension. This process is iterated as long as there are feature structures with an instantiated value for the feature conv.

To define the global extension formally, some other definitions are needed first. Let $R(\phi)$ be the restriction of the feature structure ϕ to reentrant paths, i.e. only the reentrant paths in ϕ are present in $R(\phi)$. The result of *default unification* ($\sqcup_d$) of a feature structure ϕ and a set of feature structures Ψ is a feature structure defined as follows:

$$\phi \sqcup_d \Psi \;=\; \begin{cases} \phi \sqcup \bigsqcup\{\psi \in \Psi \mid \psi \sqcup \phi \neq \top\} & \text{if } R(\phi) \sqcup \bigsqcup \Psi \neq \top \text{ and } \phi \sqcup R(\bigsqcup \Psi) \neq \top \\ \top & \text{otherwise} \end{cases}$$

The condition about reentrant paths in the feature structures ensures that unification will fail if the default unification is order-sensitive. Note that the arguments of default unification have different types.

Let C be a class with a main equation set that corresponds to the feature structure M, with default equations that correspond to the feature structure set D, and with m variant equation sets that correspond to the feature structures $V_1, \ldots, V_m$, respectively. The *superclass extension* of a set of feature structures Φ with respect to the class C or a class list $\langle C_1, C_2, \ldots, C_n \rangle$ is defined as follows:

$$se(\Phi, C) \;:=\; \begin{cases} \{\psi \mid \phi \in \Phi \wedge \psi = \phi \sqcup M \sqcup_d D \sqcup V_i \wedge \psi \neq \top \wedge 1 \leq i \leq m\} \\ \quad \text{if } m \geq 1 \\ \{\psi \mid \phi \in \Phi \wedge \psi = \phi \sqcup M \sqcup_d D \wedge \psi \neq \top\} \\ \quad \text{otherwise} \end{cases}$$

$$se(\Phi, \langle\rangle) \;:=\; \Phi$$

$$se(\Phi, \langle C_1, C_2, \ldots, C_n \rangle) \;:=\; se(se(\Phi, C_1), \langle C_2, \ldots, C_n \rangle)$$

Using the above definitions, the *global extension* of a lexical class L with a CPL C, $ge(C)$, can be defined as follows:

$$ge_0(C) \;:=\; se(\{\bot\}, C)$$

$$ge_{i+1}(C) \;:=\; ge_i(C) - Conv(ge_i(C)) \bigcup_{C_0 \in New(ge_i(C))} ge(C_0)$$

with $Conv(X) := \{\phi \mid \phi \in X \wedge \phi\hat{}conv \neq \bot\}$

and $New(X) := \{C_0 \mid \phi \in X \wedge \phi\hat{}conv = c(S) \wedge C_0$ is the CPL of a lexical class with a superclass list S and a main equation set that corresponds to the feature structure which results from applying the converter c to $\phi\}$

$$ge(C) \;:=\; ge_{\mu i(Conv(ge_i(C))=\emptyset)}(C)$$

Note that the global extension is undefined if there is no i with $Conv(ge_i(C)) = \emptyset$. The lexicon writer must ensure that the lexicon doesn't specify any cyclic type conversions, e.g.

that one can derive a verb from a noun, and derive a noun from this verb, and derive a verb from this noun, and so on ad infinitum.

The second extension of the type system are metaclasses. A metaclass is used to express information that is applicable to more than one type of feature structures. For example,

```
metaclass c
  main
    a = b
.
```

is the definition of a metaclass. The metaclass c will be instantiated to a class when it appears in the CPL of a class L. The type of the instantiation of c used for L is the type of L. For the semantics of an IBL lexicon, metaclasses can be regarded as syntactic sugar. For example, the metaclass c stands for the following set of classes:

```
class c_T inherit T
  main
    a = b
.
```

where T is the name of a top class and c_T is a legal IBL class, i.e. all paths must exist in T, etc. A reference to the class c in a superclass S is an abbreviation for c_T where T is the type that is associated with the class S.

Metaclasses are not included in IBL since example lexicons showed that metaclasses are rarely needed.

After introducing types one might ask what the properties of IBL's type system are. The type system is *sound*, i.e. there is no need for dynamic type checking. A program can translate an IBL lexicon into a version that doesn't need to check for type errors. Only the translated lexicon is accessed by applications that use IBL lexicons. All type checking that is necessary is performed, in IBL only by the translator. Therefore the implementation of the IBL language is *strongly typed*. This means that the use of a successfully translated lexicon can't cause type errors.

3.3 Global and Local Parts of Feature Structure Types

A typical lexicon specification contains among other things a special kind of information: information that the computer needs, but the user doesn't need. An example of such information are stems that are used for calculating different morphological forms of one lexeme; the computer needs the stem in order to be able to apply the equations in some morphological class, while the application using the lexicon is only interested in the result of this process and not in any intermediate results.

To distinguish these two kinds of information, a class type is split into a *global* and a *local* part. Only the features in the global part are visible to an application that uses a lexicon.

3.4 Fast Lexicon Lookup

The data structure of letter trees is used for lexicon lookup. Letter trees for lexicons are described by Covington (1994, pp. 264–267).

While the ELU system uses an indexing scheme, IBL uses letter trees, because they need less memory than binary trees and have similar access times. They save memory because typically an entry for a word form just contains a one or two letter suffix of the word form, while the rest of the word form is shared by many different words.

The letter tree I use is more compact than an ordinary letter tree. If there exists a sequence of arcs annotated with single letters and there are no alternative arcs leading away, the arcs are collapsed into a single arc annotated with a word. For example, the letter tree

ltree([a,[b,[c,[d, [e, *entry for abcde*]]]], [c, *entry for ac*]])

is replaced by the more efficient structure

ltree([a, [bcde, *entry for abcde*], [c, *entry for ac*]])

An entry in a letter tree for an IBL lexicon contains only a numerical class identifier or a list of numerical class identifiers of all lexical classes whose global extensions contain a feature structure that belongs to the entry. As the class identifier often stays the same at the end of a path in the letter tree, the class identifier is omitted if it is the same as the last one found on a prefix. This is an example of this compact form:

. . . [g, [e, . . . [funden, 13171, [e, [m], [n], [r], [s]]] . . .] . . .], . . .

The savings in space can be quite big, especially for highly inflectional languages that mainly use suffixes in their inflectional paradigms, e.g. French, German, Italian, Russian, Spanish, etc. The above given part of a compact letter tree corresponds to the following index structure:

. . . (gefunden, 13171), (gefundene, 13171), (gefundenem, 13171), (gefundenen, 13171), (gefundener, 13171), (gefundenes, 13171), . . .

For a small German lexicon with 72 lexical classes specifying 407 different word forms, a letter tree needs only 75% of the memory needed by a normal index structure, like the one given above.

As described in section 2, the class identifier found in a letter tree is used to calculate only the part of the global extension of the lexicon that is needed for a given word. For example lexicons, a typical lookup time is 0.1 s - 1 s on a SUN SPARC 1+. Around 0.02 s are needed for searching in a letter tree with 500 entries; for lexical classes with a long CPL, generating the global extension needs the major part of the total lookup time.

It can take much less time to generate the global extension of a lexical class when the relevant word form is known because this information gives a constraint for the feature form. If further constraints are known in the application program using an IBL lexicon, these can

be passed to the lexicon lookup in order to decrease the lookup time even more.

3.5 Basic Aspects of the Syntax of IBL Lexicons

The syntax for ELU lexicons is compact, but not very readable to human readers. The IBL system has hence a more verbose syntax. Each part in a lexicon specification is introduced by a keyword. For instance, variant introduces a variant equation set, while the ELU system uses a vertical bar (|) instead. Some extensions of the syntax reflect extensions of the formalism, e.g. the introduction of types. In addition, the syntax of IBL lexicons differs from that of ELU lexicons in the following aspects.

A path consists of an attribute name or of several attribute names separated by carets, e.g. agr^pers refers to the value of the feature pers in the feature agr.

A disjunction of atomic feature structures a and b is written a \/ b in IBL, and a / b in an ELU lexicon.

A string of characters, which is used to describe a word form or a part of it, must be enclosed in double quotes. The string concatenation operator is the ampersand (&).

IBL provides the possibility to give a predicate in an equation set. This is used for functional dependencies, feature values that can be computed from other feature values, and the like. Predicates must be valid Prolog predicates. ELU uses a similar concept which is called a macro.

IBL allows to include other files. If IBL finds the directive

```
include 'types.ibl'.
```

in a lexicon specification, it will behave as if the contents of the file types.ibl were at this position in the lexicon. If the file has already been loaded, the directive will be ignored.

Comments are written as in Prolog: a percent sign (%) introduces a comment that ends at the end of the current line; a slash immediately followed by a star (/*) starts a comment which ends at the next star that is immediately followed by a slash (*/).

The syntax of an IBL lexicon is formally defined in the following section 4.

4 Formal Definition of the IBL Syntax

The syntax of the language used for specifying an IBL lexicon is defined in extended BNF notation (in the form defined by BS 6154) as shown in Table 2 and Table 3. Note that a lexicon may contain plain Prolog terms in order to define predicates that are mentioned as conditions in an equation set (see section 3.5). Examples of class definitions are given in section 5.

```
lexicon                 =  { lexicon term | term | comment } ;
lexicon term            =  ( directive | class definition | type definition |
                           converter definition )  , '.' ;
directive               =  include directive ;
include directive       =  'include' , atom ;
type definition         =  'type' , type name , '=' , type ;
type name               =  atom ;
type                    =  predefined type | enumeration type | complex type ;
predefined type         =  'boolean_type' | 'string_type' | 'general_type' ;
enumeration type        =  '{' , atomic feature structure ,
                           { ',' , atomic feature structure } , '}' ;
complex type            =  '[' , type name , { ',' , type name } , ']' ;
atomic feature structure =  term ;
class definition        =  top class | non-lexical class | lexical class ;
top class               =  'top' , class name , global type part ,
                           [ local type part ] , equation part ;
class name              =  atom | string ;
global type part        =  'global' , feature structure type ;
local type part         =  'local' , feature structure type ;
feature structure type  =  attribute name , ':' , type name ,
                           { ',' , attribute name , ':' , type name } ;
non-lexical class       =  'class' , class name , 'inherit' , superclass list ,
                           equation part ;
lexical class           =  'word' , class name , 'inherit' , superclass list ,
                           equation part ;
superclass list         =  class name , { ',' , class name } ;
equation part           =  [ main equation set ] , [ default equation set ] ,
                           { variant equation set } ;
main equation set       =  'main' , equation set ;
default equation set    =  'default' , equation set ;
variant equation set    =  'variant' , equation set ;
equation set            =  ( equation | predicate condition )  ,
                           { ',' , ( equation | predicate condition )  } ;
equation                =  path , '=' , right hand side ;
predicate condition     =  atom , [ '(' , term , { ',' , term } , ')' ] ;
right hand side         =  path | attribute value ;
path                    =  attribute name , { '^' , attribute name } ;
attribute name          =  atom ;
attribute value         =  [ '~' ] , atomic feature structure |
                           atomic feature structure ,
                           { '\/' , atomic feature structure } |
                           '~' , '(' , atomic feature structure ,
                           { '\/' , atomic feature structure } , ')' ;
```

Table 2: Syntax of an IBL lexicon (part 1)

converter definition	=	'converter' , atom , 'from' , type name , 'to' , type name , 'conditions' , equation set ;
atom	=	(* as defined in ISO Prolog *) ;
comment	=	(* as defined in ISO Prolog *) ;
string	=	(* as defined in ISO Prolog *) ;
term	=	(* as defined in ISO Prolog *) ;

Table 3: Syntax of an IBL lexicon (part 2)

5 Examples of IBL Class Definitions

This section contains some examples which illustrate the syntax and semantics of an IBL lexicon, especially the three different kinds of classes: *top classes*, *non-lexical classes*, and *lexical classes*. The examples are intended to demonstrate certain aspects of the lexicon specification language; I don't claim that they are linguistically adequate.

The following is an example of a top class that could be used in the description of German nouns:

```
top n
  global
    sem : sem_type,
    case : case_type,
    num : num_type,
    gend : gend_type,
    pers : pers_type,
    form : string_type
  local
    stem : string_type,
    stem_pl : string_type,
    suffix : string_type
  main
    pers = 3
  default
    stem_pl = stem
  variant
    num = sg,
    form = stem & suffix
  variant
    num = pl,
    form = stem_pl & suffix
.
```

After the keywords global and local, the global part and the local part, respectively, of the feature structure type for nouns are defined. The main equation set states that a noun always refers to an object in the third person. The default equation set expresses the observation

that normally the stem for plural forms is the same as for singular forms. The variant equation sets state when and how to use the different stems: a singular word form is derived by concatenating the stem and the morphological suffix; a plural word form uses the plural stem instead of the normal stem.

The class n_singularetantum is a non-lexical class:

```
class n_singularetantum inherit n
  main
    num = sg
.
```

It inherits all the information from the class n and contains only one constraint: all word forms of a singularetantum are singular.

The class "Milch" is an example of a lexical class:

```
word "Milch" inherit n_singularetantum, n_0_en
  main
    stem = "Milch",
    gend = fem
.
```

This class inherits directly from two classes: first from the class n_singularetantum, then from a class called n_0_en, which might contain morphological information about a specific noun paradigm. The lexical class inherits indirectly from the class n, since n is a direct superclass of n_singularetantum, and from all direct and indirect superclasses of the class n_0_en. The only information that is idiosyncratic for the noun *Milch*, at least in this approach, is that its stem is Milch and its gender is feminine.

The following lexical class demonstrates that default information can be overridden:

```
word "Museum" inherit n_s_0
  main
    stem = "Museum",
    stem_pl = "Museen",
    gend = neut
.
```

The plural stem is set to the string Museen and hence overrides the information that the plural stem defaults to the normal stem. This default information comes from the top class n that will be in the superclass list of the morphological class n_s_0.

6 Usage of the IBL System

The software implementing the IBL formalism comprises several programs written in Prolog. The programs comply with the ISO Prolog draft which is edited by Scowen (1993).

Each lexicon is related to a set of files which have the same name prefix, but distinct suffixes. The only file the lexicon designer should modify directly is the .ibl-file that must contain a lexicon in the language of the IBL system.

6.1 Processing IBL Lexicons

A .ibl-file containing an IBL lexicon must be translated by the Prolog program iblp into a .pl-file, which is a Prolog program. All necessary type checking is done by the translator. Therefore the resulting file doesn't need to do any type checking. This is possible because the type system of IBL is sound. In addition, all class names are replaced by numerical identifiers, CPLs are calculated, etc. The .pl-file is more than an intermediate file; it is needed, when applications access the lexicon via a letter tree.

A translated lexicon must be further processed by the Prolog program ibl. This program generates a .ext-file that contains the global parts of all feature structures in the global extensions of all lexical classes in the corresponding .pl-file. The program ibl also generates index files for the lexicon: a .for-file containing pairs of the form (word form, identifier of the lexical class), and a .sem-file containing pairs of the form (semantics of a word, identifier of the lexical class). The generation of .sem-files is not implemented in the current version of the system, because it depends upon how the semantics of a word is described. The task of generating .sem-files is similar to producing .for-files.

The program genltree generates a letter tree from a .for-file and stores the tree as one Prolog term in a .fix-file. A .for-file also contains two commands to include the corresponding translated lexicon and the file iblfix.pl, which provides the access method described in section 6.2. If a .sem-file is produced, one can transform it into a .six-file containing a letter tree for access during language generation. The program genltree is written in C++ for reasons of efficiency.

The complete hierarchy of files for a lexicon named german.ibl is illustrated in Figure 1.

All of the files that are directly or indirectly derived from a .ibl-file can be generated automatically by using the file makefile for the UNIX command make. The file that one wants to be generated must be given as an argument to the make command. To generate a current version of the file german.fix, one can type the command

```
make german.fix
```

6.2 Accessing IBL Lexicons

An IBL lexicon is not very useful, unless it is used by an application program for language analysis or generation or both. There are two ways to access an IBL lexicon.

One can directly refer to the .ext-file that contains the complete extension of the lexicon, i.e. all typed feature structures that are admitted by the lexicon. This way is not recommended, because it will become impossible to pursue this method with a lexicon containing a large number of lexical classes, especially of a highly inflectional language. A short calculation can

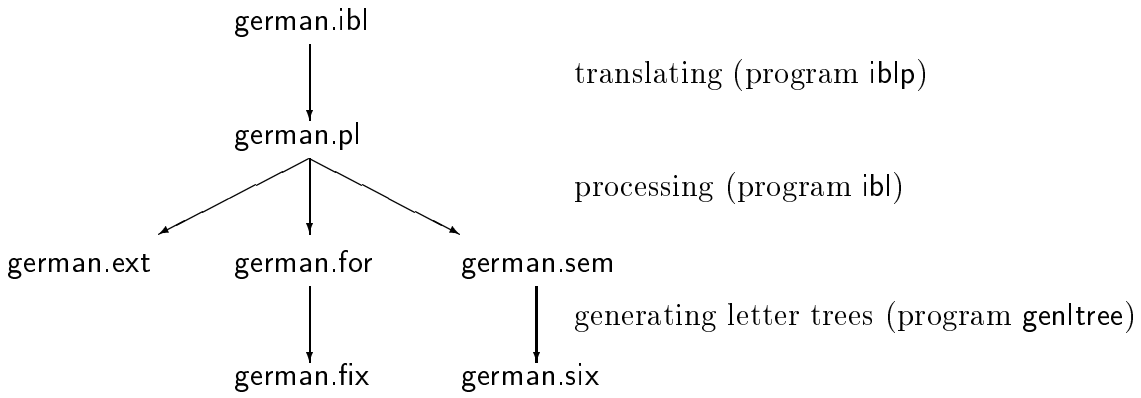

Figure 1: Files belonging to a lexicon named german.ibl

illustrate this point. If a lexicon contains 100000 lexical classes and the global extension of a lexical class has 10 feature structures on the average with an average size of 100 bytes, the global extension of the lexicon written as Prolog terms requires at least 100 Mbyte of main memory. Today, this memory requirement exceeds the capacity of most computer systems. However, this direct access to the extension of the lexicon might be useful for small lexicons and during certain phases of developing systems for natural language processing.

The highly recommended way of accessing the lexicon uses one level of indirection. This level is provided by a letter tree. For example, if one wants to use the lexicon that is specified in the file french.ibl for looking up words in a Prolog program that analyzes French sentences, the program must include the file french.fix with the Prolog directive:

```
:- ensure_loaded('french.fix').
```

To obtain all typed feature structures that have the written form *aimera*, one can use the predicate iBl_fix_lookup/2 to write the following Prolog goal:

```
iBl_fix_lookup( "aimera", Feature_structures)
```

If the word given as the first argument is not admitted by the lexicon, the predicate is false; otherwise the predicate is true and the variable Feature_structures is instantiated to the list of all typed feature structures that are specified by the lexicon french.ibl and satisfy the condition

```
form = "aimera"
```

If one includes a .fix-file, the corresponding .pl-file, the file iblfix.pl, and the file ibl.pl are used and must hence be available.

7 Comparison with Similar Approaches

7.1 DATR

DATR is a lexicon formalism that uses a kind of multiple default inheritance that differs from the one used in IBL. A path can specify another *node* which corresponds to a class in IBL instead of a value and inherits the value from this node. Evans and Gazdar (1989) describe DATR and its kind of inheritance.

The most important advantages of IBL over DATR are the following:

1. In IBL, but not in DATR, there is a way to protect data from being overridden: the definition of non-defeasible information in main and variant equation sets. This distinction between defeasible and non-defeasible information is a powerful error detection device.

2. IBL is typed. As shown in the preceding sections, typedness is an important aid to specify consistent and error free lexicons. DATR doesn't use typed feature structures. In addition, the typedness of feature structures increases the efficiency of implementations of a lexicon formalism. (See also section 3.2.)

There is one point that might seem to be an advantage of DATR: in DATR every path can specify a source of information. This provides a higher degree of flexibility than the concept of a superclass list in IBL. However, the question is whether this flexibility is too high and can lead to badly structured inheritance networks. During the development of IBL, only two inflexibilities, which were due to the introduction of types, became obvious and led to the introduction of converters and metaclasses. The inheritance method in IBL seems to be flexible enough and encourages through its limitations compared to path inheritance a better and clearer structure of lexicon specifications.

Russell, Ballim, Carroll, and Warwick-Armstrong (1992, pp. 325–329) compare the definition of inheritance in ELU with the one in DATR and in comparable formalisms.

7.2 ELU

As IBL is based on ELU, only the modifications and extensions must be considered in a comparison between these two formalisms, which use the same concept of multiple default inheritance. The extensions are discussed at their introduction, mainly in section 3. The main advantage of IBL over ELU seems to be the typedness of feature structures with all its consequences for design and error detection.

8 Perspectives

Although the IBL lexicon formalism is a working system, there are still a lot of things to discuss and to do.

First, a feature of the ELU lexicon formalism that is missing in IBL and is worth including.

- The .sem- and .six-files are not generated by the current system. As mentioned in section 6.1, this task is similar to the one of producing .for- and .fix-files from a translated lexicon file.

Second, features that are not present in the ELU lexicon formalism and are candidates for a further development of the IBL formalism.

- Disjunction of non-atomic feature structures could be allowed. IBL restricts disjunction to atomic feature structures. Therefore one must sometimes use more variant equation sets than linguistically necessary. For instance, one cannot write a variant equation set for the German verb suffix *-t* like this:

```
variant
   morph^suffix = "t",
   agr = (pers = 3, num = sg) \/ (pers = 2, num = pl)
   temp = pres,
   mod = ind,
   vform = fin
```

  If one wants to use these features, one must give two variant equation sets with almost identical equations:

```
variant
   morph^suffix = "t",
   agr^pers = 3,
   agr^num = sg,
   temp = pres,
   mod = ind,
   vform = fin
variant
   morph^suffix = "t",
   agr^pers = 2,
   agr^num = pl,
   temp = pres,
   mod = ind,
   vform = fin
```

  The global extension of every lexical class that describes a German verb has hence (at least) one element more than necessary with general disjunction.

- Negation could be defined for arbitrary feature structures. IBL limits negation to atomic feature structures and disjunctions of them. In some cases, it would be convenient to have negation defined for general feature structures.
- The type system could be further elaborated. Possible extensions are lists and sets of types as new feature structure types. However, it is unlikely that these types are needed for processing in the lexicon. To provide sets or lists of feature structures to users of the lexicon, it might suffice to use the predefined type general_type because the lexicon formalism need not look inside the values and can view them as atomic.
- The way of conflict resolution by CPLs may be too strict. An alternative would be to specify for each feature the relevant superclass or a function that computes the value in case of an ambiguity. But this would increase the complexity of the inheritance system. Therefore one might prefer to keep the restrictions imposed by CPLs. In practice, the restrictions turned out to enforce a clear design and structure of lexicons.

On the one hand, the above mentioned extensions increase the expressiveness of the formalism; on the other hand, they complicate the implementation and the work of the lexicon formalism.

References

Carpenter, B. (1992). *The logic of typed feature structures.* Cambridge Tracts in Theoretical Computer Science. New York: Cambridge University Press.

Covington, M. A. (1994). *Natural language processing for Prolog programmers.* Englewood Cliffs, New Jersey: Prentice Hall.

Evans, R. and G. Gazdar (1989). Inference in DATR. In *Proceedings of the 4th Conference of the European Chapter of the Association for Computational Linguistics*, pp. 66–71.

Russell, G., A. Ballim, J. Carroll, and S. Warwick-Armstrong (1992). A practical approach to multiple default inheritance for unification based lexicons. *Computational Linguistics 18*(3), 311–337.

Russell, G., J. Carroll, and S. Warwick-Armstrong (1990). Multiple default inheritance in a unification-based lexicon. In *Proceedings of the 13th International Conference on Computational Linguistics (COLING 90)*, pp. 215–221.

Scowen, R. (Ed.) (1993). *Prolog: part 1, general core, committee draft (ISO/IEC JTC1 SC22 WG17 N110).* Teddington, England: National Physical Laboratory (for International Organization for Standardization).

Touretzky, D. S. (1986). *The mathematics of inheritance systems.* Research Notes in Artificial Intelligence. London: Pitman Publishing.