Committee Draft ISO/IEC CD	
Date: **2005-06-23**	Reference number: ISO/JTC 1/SC 32N**1330**
Supersedes document SC 32N1238	

THIS DOCUMENT IS STILL UNDER STUDY AND SUBJECT TO CHANGE. IT SHOULD NOT BE USED FOR REFERENCE PURPOSES.

ISO/IEC JTC 1/SC 32 Data Management and Interchange Secretariat: USA (ANSI)	Circulated to P- and O-members, and to technical committees and organizations in liaison for voting (P-members only) by: **2005-09-23** Please return all votes and comments in electronic form directly to the SC 32 Secretariat by the due date indicated.

ISO/IEC CD 24707:200x(E)

Title: Information technology -- Common Logic (CL) – A Framework for a Family of Logic-Based Languages

Project: 1.32.25.01.00.00

Introductory note: The attached document is hereby submitted for a three-month letter ballot to the National Bodies of ISO/IEC JTC 1/SC 32. The ballot starts 2005-06-23.

Medium: E

No. of pages: 54

Address Reply to: Douglas Mann, Secretariat, ISO/IEC JTC 1/SC 32, Farance Inc, 360 Pelissier Lake Road, Marquette, MI 49855, United States of America

Telephone: +1 906-249-9275; E-mail: MannD@battelle.org

ISO/IEC JTC 1/SC 32 N 1330

2005-06-23

ISO/WD 24707

ISO/JTC 1/SC 32/WG2

ANSI

Information technology — Common Logic (CL) – A Framework for a Family of Logic-based Languages

Logique commune (logique commune) - Cadre pour une famille des langages logique-basés

Warning

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

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

Copyright notice

Contents

Page

Foreword

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

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

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

Working Draft ISO/IEC 24707 was prepared by Joint Technical Committee ISO/IEC JTC JTC 1, *Information technology*, Subcommittee SC 32, *Data management and interchange*.

Introduction

Common Logic is a logic framework intended for information exchange and transmission. The framework allows for a variety of different syntactic forms, called dialects, all expressible within a common XML-based syntax and all sharing a single semantics.

Common Logic has some novel features, chief among them being a syntax which is signature-free and permits 'higher-order' constructions such as quantification over Common Logicasses or relations while preserving a first-order model theory, and a semantics which allows theories to describe intensional entities such as Common Logicasses or properties. It also fixes the meanings of a few conventions in widespread use, such as numerals to denote integers and quotation marks to denote character strings, and has provision for the use of datatypes and for naming, importing and transmitting content on the World Wide Web using XML.

Information technology — Common Logic (Common Logic) – Framework for a family of logic-based languages

1 Scope

This standard specifies a family of languages designed for use in the representation and interchange of knowledge among disparate computer systems.

The following features are essential to the design of this standard:

- Languages in the family have declarative semantics. It is possible to understand the meaning of expressions in these languages without appeal to an interpreter for manipulating those expressions.
- Languages in the family are logically comprehensive—at its most general, they provide for the expression of arbitrary logical sentences.

The following are within the scope of this standard:

- interchange of knowledge among heterogeneous computer systems;
- representation of knowledge in ontologies and knowledge bases;
- specification of expressions that are the input or output of inference engines.

The following are outside the scope of this standard:

- the specification of proof theory or inference rules;
- specification of translators between the notations of heterogeneous computer systems.
- free logics
- conditional logics
- methods of providing relationships between symbols in the logical “universe” and individuals in the “real world”.

This document describes Common Logic's syntax and semantics.

The standard defines an abstract syntax and an associated model-theoretic semantics for a specific extension of first-order logic. The intent is that the content of any system using first-order logic can be represented in the standard. The purpose is to facilitate interchange of first-order logic-based knowledge and information between systems.

Issues relating to computability using the standard (including efficiency, optimization, etc.) are not addressed.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

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

ISO/IEC 2382-15:1999, Information technology -- Vocabulary -- Part 15: Programming languages

ISO/IEC 10646:2003, Information technology -- Universal Multiple-Octet Coded Character Set (UCS)

ISO/IEC 14977, Information technology -- Syntactic metalanguage -- Extended BNF

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

3.1 Definitions

3.1.1
conceptual graph

A graphical or textual display of symbols arranged according to the style of conceptual graph theory as introduced by John Sowa [1].

3.1.2
Common Logic Interchange Format

The KIF-based syntax that is used for illustration purposes in the standard. It is one of the concrete syntaxes as described in Annex B. NOTE: The name "KIF" is not used for this syntax in order to distinguish it from the commonly used (but non-standard) KIF dialects.

3.1.3
Conceptual Graph Interchange Format

Rules for the formation of a textual string that conforms to Annex B of this document. Sometimes may refer to an example of a character string that conforms to Annex B. Intended to convey exactly the same structure and semantics as an equivalent conceptual graph.

3.1.4
dialect

A concrete syntax that shares (at least some of) the uniform semantics of Common Logic. A dialect may be textual or graphical or possibly some other form. A dialect by definition is also a conforming language (see clause 7.1 for further details).

3.1.5
interpretation

A relationship between individuals in a universe of discourse and the symbols and relations in a model such that the model expresses truths about the individuals. See section 6.2 for a more precise description of how an interpretation is defined.

3.1.6
Knowledge Interchange Format

A text-based first order formalism, using a LISP-like list notation, originating with the Knowledge Sharing Effort sponsored by the U.S. DARPA [2]. It forms the basis for one of the three Common Logic dialects included in this standard.

3.1.7
segregated dialect

A dialect in which some names are non-denoting names. That is, where some names do not map to individuals in the universe of discourse.

3.1.8 sort

For the purposes of this standard, **sort** means a type which is an attribute of a symbol. In practice, a sort represents a class of individuals.

3.1.9
sorted logic

A logic system (whether first-order or not) that requires all nonlogical symbols to be assigned a sort (i.e., a type), and in which a failure to conform to the sort structure is considered to be a semantic error.

3.1.10
traditional first order logic

The traditional algebraic (or mathematical) formulations of logic as introduced by Russell, Peano, Frege, and Peirce dealing with quantification, negation, logical relations as expressed in propositions which are strictly true or false. This specifically excludes reasoning over relations and excludes using the same name as both an individual and a relation name.

3.1.11
universe of discourse
universe

A nonempty set over which the quantifiers of a logical language are understood to range. Sometimes called a “domain of discourse”.

3.1.12
Uniform Resource Identifier

A Uniform Resource Identifier (URI) is a compact sequence of characters that identifies an abstract or physical resource. A generic version of a Web-based URL [3]. A URI is a locator string, generally designed for use over a network, that uniquely identifies a resource (e.g., a document) that exists somewhere on the network. A means of identifying any individual in a universe of discourse, used mainly in XCL.

3.1.13
XCL

The XML-based syntax for Common Logic, designed for use by mechanical systems to interchange logical content. XCL also serves as a syntactic common target for other Common Logic dialects; a specification of a formal meaning-preserving translation into XCL is sufficient to qualify a formal language as a Common Logic dialect.

4 Symbols and Abbreviations

These symbols and abbreviations are generally for the main clauses of the standard. Each annex may introduce its own symbols and abbreviations which will be grouped together within that annex.

4.1 Symbols

D_I	A domain of discourse; a non-empty set of individuals that an interpretation is “about”.
fun_I	A mapping from U_I to functions D_I*->D_I
I	An interpretation
int_I	A set of interpretations from names in V to U_I, informally, a means of associating names in V to individuals in U_I.
int_I(x)	A relationship between a name x in V and one or more individuals in U_I. Informally, it represents the individual(s) denoted by x. See clause 6.2.
rel_I	A mapping from U_I to subsets of D_I*
S*	For any set S, the set of all possible subsets of S, including S itself.
seq_I	A mapping from sequence variables in V to D_I*.
V	a vocabulary of a set of names and sequence variables
U_I	the universe of individuals represented by some interpretation I

4.2 Abbreviations

These abbreviations are used in this document. See sec. 3.1 for definitions or further elaboration on these terms.

CG	Conceptual graph
CGIF	Conceptual Graph Interchange Format
CL	Common Logic
CLIF	Common Logic Interchange Format
DF	Display form (used in Annex B)
KIF	Knowledge Interchange Format
OWL	Ontology Markup Language for the Web
RDF	Resource Definition Framework
RDFS	Resource Definition Framework Schema

TFOL	traditional first order logic
URI	Uniform Resource Identifier
XCL	XML-based Common Logic markup language
XML	eXtendable Markup Language

5 Introduction and rationale

This section is informative.

5.1 Requirements

Common Logic has been designed and developed with several requirements in mind, all arising from its intended role as a medium for transmitting logical content on an open communication network.

1. Common Logic is full first-order logic with equality. Common Logic syntax and semantics should provide for the full range of first-order syntactic forms, with their usual meanings. Any conventional first-order syntax should be directly translateable into Common Logic without loss of information or alteration of meaning.

2. Common Logic provides a general-purpose syntax for communicating logical expressions.

- a. There should be a single XML syntax for communicating Common Logic content.
- b. The language should be able to express various commonly used 'syntactic sugarings' for logical forms or commonly used patterns of logical sentences
- c. The syntax should relate to existing conventions; in particular, it should be capable of rendering any content expressible in RDF, RDFS or OWL.
- d. The syntax should provide for future syntactic extensions to the language, such as modalities, extended quantifier forms, non-monotonic constructions, etc… If these do not have a currently defined semantics, then the language should be able to state any necessary requirements on their semantics.
- e. There should be at least one compact, human-readable syntax defined which can be used to express the entire language.

3. Common Logic should be easy and natural for use on the Web

- a. The XML syntax must be compatible with the published specifications for XML, URI syntax, XML Schema, Unicode and other conventions relevant to transmission of information on the Web.
- b. URI references should be usable as logical names in the language
- c. URI references should be usable to give names to expressions and sets of expressions, in order to facilitate Web operations such as retrieval, importation and cross-reference.

4. Common Logic should be an open-network Common Logic

- a. Transmission of content between Common Logic-aware agents should not require negotiation about syntactic roles of symbols, or translations between syntactic roles.

- b. Any piece of Common Logic text should have the same meaning, and support the same inferences, everywhere on the network. Every name should have the same logical meaning at every node of the network.

- c. No agent should be able to limit the ability of another agent to refer to any entity or to make assertions about any entity.

- d. The language should support ways to refer to a local universe of discourse and be able to relate it to other universes.

- e. Users of Common Logic should be free to invent new names and use them in published Common Logic content.

5. The Common Logic semantics should not make gratuitous or arbitrary assumptions about logical relationships between different expressions, particularly if these assumptions can be expressed in Common Logic directly.

5.2 Design Overview

5.2.1 A family of notations

First, if we follow the convention whereby any language has a grammar, then Common Logic is a family of languages rather than a single language. Different Common Logic languages, referred to here as *dialects*, may differ sharply in their surface syntax, but they have a single uniform semantics and can all be transcribed into the common syntax. Membership in the family is defined by being inter-translatable with the other dialects while preserving meaning, rather than by having any particular syntactic form. Several existing logical notations and languages, therefore, can be considered to be Common Logic dialects.

One Common Logic dialect plays a special role. The XML syntax for Common Logic, here called XCL (see Annex C), is designed to serve three functions. It is a standard intercommunication notation for machine use on a network; it is the single 'reference syntax' into which all other dialects can be translated or embedded (requirement 5.1(2)a); and it provides forms which can be used to express various extensions and relations to existing standards (requirements 5.1(2)b, 5.1(2)c, 5.1(2)d) and to support existing conventions regarding URI usage (requirements 5.1(2)c, 5.1(3)c, 5.1(3)d and 5.1(4)e). XCL syntax is intended to be unambiguous, extendable, capable of expressing a wide variety of logical forms, and to provide ways to access logically meaningful subexpressions using conventional XML technology; it is not intended to be compact or human-readable. This document also independently defines a more conventional dialect, CLIF, which is intended to be compact, human-readable (requirement 5.1(2)e) and more similar to conventional machine-oriented logics. A Common Logic syntax called CLIF based on KIF (see Annex A) is used in giving examples throughout this document. CLIF can be considered an updated and simplified form of KIF 3.0 [2], and hence a separate language in its own right, and so a complete self-contained description is given which can be understood without reference to the rest of the specification. Conceptual graphs [1] are also a well-known first-order logic for machine processing; the CGIF language is specified in Annex B.

5.2.2 Background discussion and motivation

Most of the design aspects of Common Logic can be understood by considering the following scenario. Two people, or more generally two agents, A and B, each have a logical formalization of some knowledge and access to a communication network which allows them to send such formalized information to one another and to other agents. They now wish to communicate their knowledge to a third agent C which will make use of the combined information so as to draw some conclusions. All three agents are using first-order logic, so they should be able to communicate this information fully, so that any inferences which C draws from A's input should also be derivable by A using basic logical principles, and vice versa; and similarly for C and B. The goal of Common Logic is to provide a logical framework which can support this kind of communication and use without requiring complex negotations between the agents (requirement 5.1(4)a). This ought to be simple, but in practice there are many barriers to such communication.

First, A and B may have used different surface syntactic forms to express their knowledge. This is a well-known problem and various proposals have been made to solve it, usually by defining a standard syntax into which others can be translated, such as KIF [2]. Common Logic approaches this by requiring all Common Logic dialects to be translateable into XCL. The advent and widespread adoption of XML technology has made this problem easier to

solve, by making it possible to divorce the 'structured' specification of the language from any particular surface form. XCL allows the same Common Logic logical structure to be assigned a variety of surface forms, and for various surface forms - dialects - to be transmitted unchanged within XCL markup if required.

Second, A and B may have made divergent assumptions about the logical signatures of their formalizations. Conventionally, a first-order language is defined relative to a particular signature, which is a lexicon sorted into disjoint sublexica of individual names, relation names and perhaps function names, the latter two each associated with a particular arity, i.e. a fixed number of arguments with which it must be provided. This provides a simple context-free method of checking well-formedness of terms and atomic sentences, and ensures that any expression obtained by substitution of well-formed terms for variables will be assigned a meaningful interpretation. However, it is not practical to assume that all users of a logic on an open network will adopt the same signature, particularly when new names can be invented and published at any time (requirement 5.1(4)b). It is common for one agent to use a relation name for a concept described by another as a function, for example, or for two agents to use the same relation with different argument orderings or even different numbers of arguments. More radically, a particular concept, such as marriage, might be represented by A as an individual, but by B as a relation. Often, one can give mappings between the logical forms of such divergent choices, many of which are widely familiar; but in a conventional first-order framework, these have to be considered *metasyntactic* mappings which translate *between* distinct first-order formalizations; and very few general frameworks exist to define such meta-syntactic mappings on a principled basis, in a way that allows reasoning agents to draw appropriate conclusions. The notion of signature is appropriate for textbook uses of logic and for stand-alone formalizations: but applied to an interchange situation, where sentences are distributed and transmitted over a network (in particular, on the Web) and may be utilized in ways that are remote from their source, this would require either that a single global convention for signatures be adopted, which is impractical and violates requirement 5.1(4)b, or else that communication between agents involve a negotiation about how to translate one signature into another, in violation of requirement 5.1(4)a.

Consider the situation of two agents A and B communicating information to C, and suppose that there is some concept used by both A and B but with different signatures. For example, A might use a relation *parent* with two arguments – the parent and child – while B might use it with three – two people and a birth date – or as a function from people to birth-dates. Cases like this can be handled very simply merely by allowing a symbol to play more than one role. Common Logic simply allows relation and function symbols to be variadic, i.e. to take any number of arguments, and it allows a relation name to be used as a function name and vice versa. Thinking in traditional terms, this amounts to allowing a kind of 'punning', often allowed in mechanized inference systems, where a single name is allowed to play several syntactic, and hence semantic, roles at once. Common Logic treats this more simply by conflating the various relational roles into a single notion of a variadic relation, and similarly for functions.

This is purely a syntactic 'permission' to use a name in multiple roles: Common Logic does not assume any particular relationship between the truth of sentences in which the name plays these various roles, so for example `(R a)` neither entails nor is entailed by `(R a b)` or `(exists (x) (R a x))` or any other sentence constructed using `R` with two arguments; and similarly Common Logic sanctions no logical entailments between sentences using `R` as a relation and other sentences using `R` as a function. (These and other examples are written using the CLIF dialect defined in sec. 6.) This is in conformity with requirement 5.1(5). Such conventions can be stated as axioms in Common Logic itself, but the Common Logic semantics does not impose them as a matter of logical necessity. Since there are no relevant logical principles which would mandate any such connection if distinct names were used for the various signature-distinct cases of the name, this punning does not violate the 'common logic' principle; for example, the various uses of a single name could be rendered lexically distinct by subscripting names with tags such as `-rel-3` or `-fun-2`, and the meaning of the Common Logic would be unchanged.

It is also possible that A uses a name such as *parent* as an individual name while B uses it as a relation name. For example, A might have an ontology of interpersonal states, perhaps categorized in some way. Identifying an individual with a relation may seem like a direct violation of first-order conventions, but it is important to understand that this is not necessarily an irreconcilable difference of opinion between A and B. The traditional Tarskian semantics for first-order logic requires that the universe of discourse be a nonempty set of things called 'individuals', but it does not mandate what *kind* of things must be in this set. In particular, the individuals can themselves be relations; in fact, it is quite common to introduce relations into a first-order theory by treating them as arguments to a 'dummy' relation, writing expressions such as `(Holds R a b)` rather than `(R a b)`. Thus, A's use of a name to denote an individual, and B's use of the same name to denote a relation, need not be considered a difference of

opinion: they can both be right, and can both correctly use first-order reasoning. In this case, however, the agent C who wishes to make use of both their sentences is faced with a dilemma, since traditional first-order signature-based syntax is unable to accommodate both points of view at once. Common Logic relaxes this rigidity and allows 'punning' between relation and individual names, as between relation and function names.

In this case, the requirement that C must be able to understand both A and B and still be able to reason appropriately has some more significant consequences. Since A can state equations between individuals and use terms to denote them, as for example in an equation such as

```
(= parent (childRelationBetween Joe Jane))
```

our 'common logic' requirement implies that C must be able to reason similarly, even when faced with an expression from B in which the name appears in a relation position.

```
(parent Jack Jill)
```

The resulting expression has a term in a relation position and would be considered syntactically illegal in most traditional syntaxes for first-order logic:

```
((childRelationBetween Joe Jane) Jack Jill)
```

However, it has a perfectly clear meaning which can be taken directly from the conventional first-order readings of the sentences used by A and B, viz.: the relational entity which is the value of the term **`(childRelationBetween Joe Jane)`**, being identical to *parent*, holds true between Jack and Jill.

Similarly, since A may use existential generalization on this individual name, for example to infer

```
(exists (x) (= x (childRelationBetween Joe Jane)))
```

then (by virtue of requirement 5.1(4)a) C must be able to use similar logical inferences on the name, even when it occurs in sentences from B as a relation name, producing sentences even less conventional when viewed as first-order syntax:

```
(exists (x) (x Jack Jill))
```

Common Logic also allows this as legal syntax, and treats it semantically in a first-order way: it asserts that there is something in the universe which can play a relational role and, in that role, is true between Jack and Jill; and of course, in this situation, there is such an entity in the universe. However, Common Logic semantics does not require the universe to contain any particular relations; it does not impose comprehension principles or require the set of relations in the universe to be closed under any relational-construction operations.

The resulting syntactic freedom allows a wide variety of alternative first-order axiomatic styles to co-exist within a common syntactic framework, with their meanings related by axioms, all expressed in a single uniform language. For example, the idea of parenthood can be rendered in a formal ontology as a binary relation between persons, or a more complex relation between persons, legal or ecclesiastical authorities and times; or as a legal state which 'obtains', or as a class or category of 'eventualities', or as a kind of event (e.g., “birth”). Much the same factual information can be expressed with any of these ontological options, but resulting in highly divergent signatures in the resulting formal sentences. Rather than requiring that one of these be considered to be correct and the others translated into it by some external syntactic transformation, Common Logic allows them all to co-exist, with the connections between the various ontological frameworks expressible in the logic itself. For example, consider the following sentences (all written in CLIF):

```
(parent Jack Jill)
(exists (x) (and (parent x) (= Jack (father x)) (= Jill (child x)) ))
(= (when (parent Jack Jill)) (hour 3 (pm (thursday (week 12 (year 1997))))) )
(= (child (parent 32456)) Jill)
(ParentalStatus parent Jack)
((ParentalStatus Jack) Jill)
```

These all express similar propositions about a Jack being Jill’s parent, but in widely different ways. The logical name *parent* appears here as a binary relation between two people; a binary relation between a parent-child pair and a

time; a function from numbers to individuals; as an individual 'status', and finally, although not named explicitly, as the value of a function from persons to predicates on persons. In a fully expressive Common Logic dialect, such as CLIF, all these forms can be used at the same time.

Common Logic also allows quantification over relations and functions. Common Logic syntax accepts the use of relation-valued functions, relations applied to other relations, and so on. The resulting syntax is reminiscent of a higher-order logic: but unlike traditional higher-order logic, Common Logic syntax is completely type-free. It requires no allocation of higher types to functionals such as **`ParentalStatus`**, and imposes no type discipline. Higher-order languages traditionally require more elaborate signatures than first-order languages, in order to prohibit 'circular' constructions involving self-application, such as

```
(rdfs:class rdfs:class)
```

(This example expresses a basic semantic assumption of RDFS [4].) Common Logic syntax, in contrast, is obtained from conventional first-order syntax by *removing* signature restrictions, so that both Common Logic atomic sentences and terms can be described uniformly as a term followed by a sequence of argument terms.

5.2.3 On being first-order

Readers familiar with conventional first-order syntax will recognize that this syntactic freedom is unusual, and may suspect that Common Logic is a higher-order logic in disguise. However, as has been previously noted on several occasions [5], a superficially 'higher-order' syntax can be given a completely first-order semantics, and Common Logic indeed has a purely first-order semantics, and (without sequence variables: see below) satisfies all the usual semantic criteria for a first-order language, such as compactness and the Skolem-Lowenheim property. What makes a language semantically first-order is the way that its names and quantifiers are interpreted. A first-order interpretation has a single homogenous universe of individuals over which all quantifiers range; any expression that can be substituted for a quantified variable is required to denote something in this universe. The only logical requirement on this universe is that is a non-empty set. Relations hold between finite sequences of individuals, and all that can be said about an individual is the relations it takes part in: there is no other structure. Individuals are 'logically atomic' in a first-order language: they have no logically relevant structure other than the relationships in which they participate. Expressed mathematically, a first-order interpretation is a purely relational structure. It is exactly this essential simplicity which gives first-order logic much of its power: it achieves great generality by refusing to countenance any restrictions on what kinds of 'thing' it talks about. It requires only that they stand in relationships to one another. Contrast this with for example higher-order type theory, which requires an interpretation to be sorted into an infinite hierarchy of types, satisfying very strong conditions. The goal of higher-order logic is to express logical truths about the domain of *all* relations over some basic universe, so a higher-order logic supports the use of comprehension principles which guarantee that relations exist. In HOL, a quantification over a higher type is required to be understood as ranging over all possible entities of that type; often a set of a very high cardinality. In contrast, Common Logic allows the universe to contain relations, but only requires that the entities in the first-order domain of discourse have relational extensions.

The semantic technique used by Common Logic is an extension of the 'punning' by which it identifies relation and function symbols. In a conventional interpretation, a relation symbol denotes a relational extension (a set of n-tuples over the domain), a function symbol denotes a functional extension (a set of pairs of n-tuples and elements, one for each possible sequence), and an individual name denotes an element in the domain. In order to satisfy requirement 5.1(4)b, occurrences of the same name in different contexts must not have different meanings; in order to satisfy requirement 5.1(4)c, it must be possible to interpret any name as playing any logical role. The obvious construction, then, is to require that every name denote an individual and to associate the relational and functional extensions with that individual. In order to accommodate the situation in traditional logic where a relation or function name is not required to denote something in the universe of discourse, Common Logic interpretations distinguish a universe and a domain. The universe may be larger than the domain (over which the quantifiers range) and names that denote elements outside the domain play exactly the role of a non-individual name in a conventional syntax. The Common Logic syntax provides for excluding items from the domain, allowing users to exert precise control over their intended domains of discourse where required; but communication with another agent requires that the *universes* of interpretations can be identified. Distinguishing the universe from the domain in this way therefore allows global coherence to be achieved without sacrificing local expressiveness.

6 Common Logic abstract syntax and semantics

This section is normative.

6.1 Common Logic abstract syntax.

We describe the syntax of Common Logic 'abstractly' here in order to not be committed to any particular dialect's syntactic conventions.

6.1.1 Abstract syntax categories

Each of the following entries is called an *abstract syntax category*.

6.1.1.1 A text is a set, list or bag of phrases. A piece of text may be *identified* by a name. *A Common Logic text may be a sequence, a set or a bag of phrases; dialects may specify which is intended or leave this undefined. Re-orderings and repetitions of phrases in a text are semantically irrelevant. However, applications which transmit or re-publish Common Logic text should preserve the structure of texts, since other applications may utilise the structure for other purposes, such as indexing. If a dialect imposes conditions on texts then these conditions must be preserved by conforming applications.*

6.1.1.2 A phrase is either a comment, or a module, or a sentence, or an importation, or a phrase with an attached comment.

6.1.1.3 A comment is a piece of data. *Comments may be attached to other comments and to commented phrases. No particular restrictions are placed on the nature of Common Logic comments; in particular, a comment may be Common Logic text. Particular dialects may impose conditions on the form of comments.*

6.1.1.4 A module consists of a name, an optional set of names called the *exclusion set*, and a text called the *body text*. *The module name indicates the 'local' domain of discourse in which the text is understood; the exclusion list indicates any names in the text which are excluded from the local domain. A module name may also be used to identify the module.*

6.1.1.5 An importation contains a name. The intention is that the name *identifies* a piece of Common Logic content represented externally to the text, and the importation re-asserts that content in the text. *The notion of identification is discussed more fully below.*

6.1.1.6 A sentence is either a quantified sentence or a Boolean sentence or an atom, or a sentence with an attached comment, or an irregular sentence.

6.1.1.7 A quantified sentence has a type, called a *quantifier*, and a set of names called the *bound names*, and a sentence called the *body* of the quantified sentence. Every Common Logic dialect must distinguish the *universal* and the *existential* types of quantified sentence. Any occurrence of a bound name in the body is said to be *bound in* the body; any name which occurs in the body and is not bound in the body is *free in* the body.

6.1.1.8 A Boolean sentence has a type, called a *connective*, and a number of sentences called the *components* of the Boolean sentence. The number depends on the particular type. Every Common Logic dialect must distinguish the *conjunction* , *disjunction*, *negation*, *implication* and *biconditional* types with respectively any number, any number, one, two and two components.

The current specification does not recognize any irregular sentence forms. They are included in the abstract syntax to accommodate future extensions to Common Logic, such as modalities.

6.1.1.9 An atom is either an *equation* containing two *arguments* which are terms, or consists of a term, called the *predicate*, and a term sequence called the *argument sequence*, containing terms called *arguments* of the atom. *Dialects which use a name to identify equality may consider it to be a predicate.*

6.1.1.10 A term is either a name or a functional term, or a term with an attached comment.

6.1.1.11 A functional term consists of a term, called the *operator*, and a term sequence called the *argument sequence*, containing terms called *arguments* of the functional term.

6.1.1.12 A term sequence is a finite sequence of terms and an optional sequence variable. A sequence may be empty or may consist of a single sequence variable. Only one sequence variable may occur in a term sequence.

6.1.2 Metamodel of the Common Logic Abstract Syntax

In order to better describe the structure of the abstract syntax, this section provides a metamodel showing relationships among the syntactic categories, and describes some of the rationale for decisions, The abstract syntax categories and their allowable structure is depicted diagrammatically using UML notation [6].

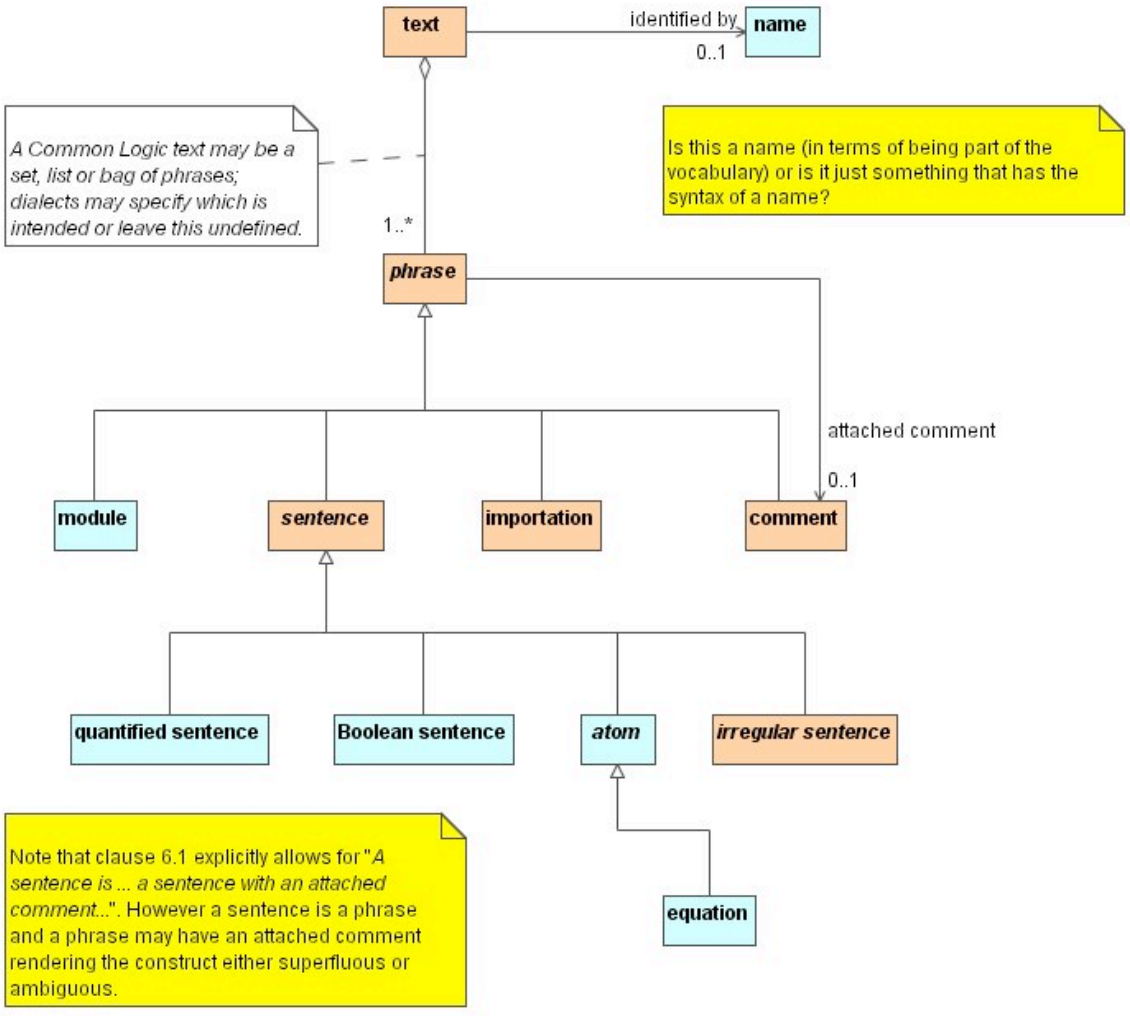

Figure 1 – Structure of a text and the taxonomy of the phrase category *text*

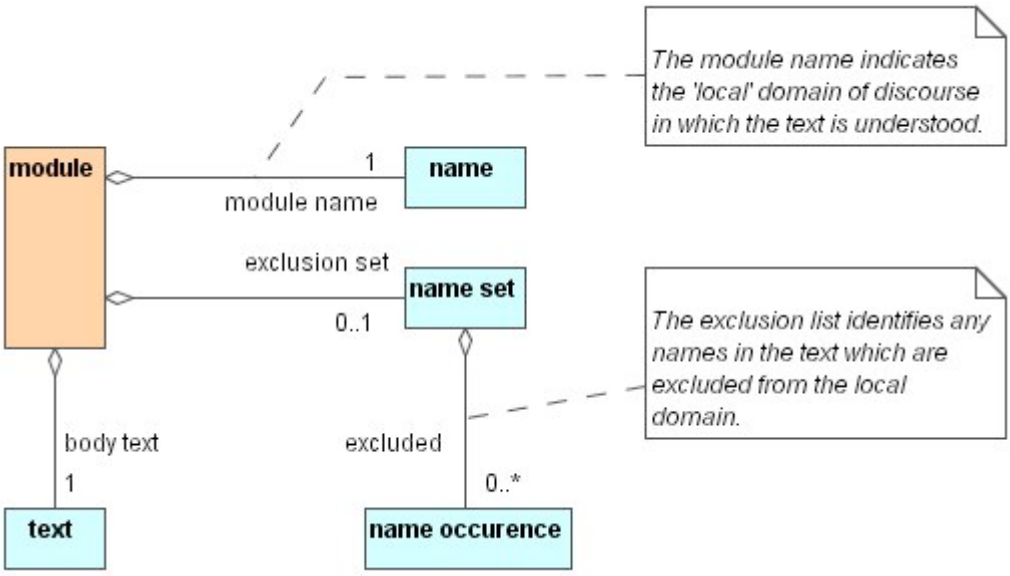

Figure 2 – Abstract syntax of a *module*

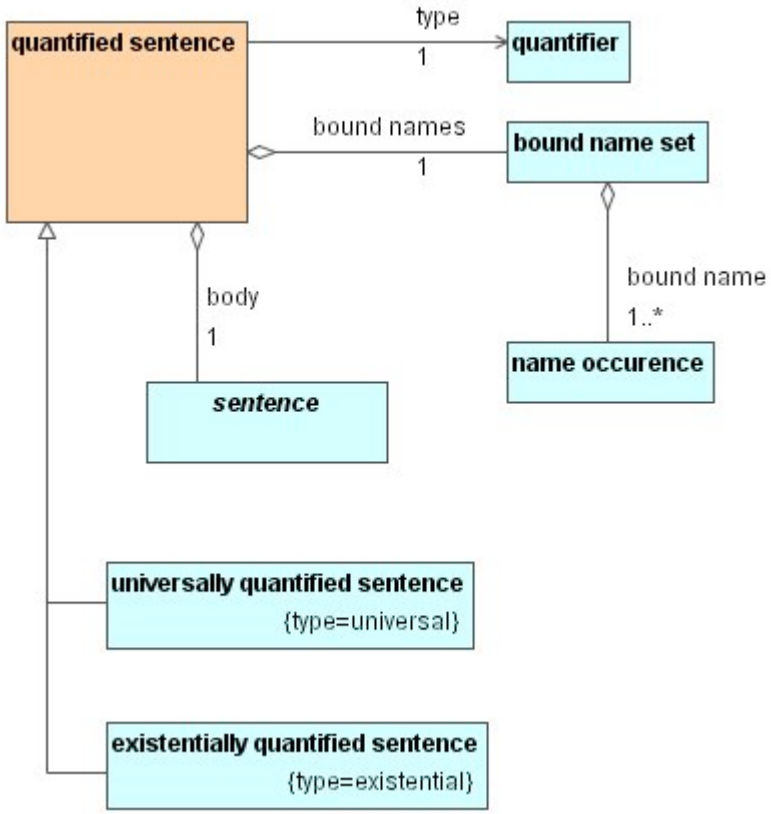

Figure 3 – Abstract syntax of a *quantified sentence*

Figure 3 depicts the abstract syntax of a quantified sentence. A *universally quantified sentence* is a quantified sentence whose quantifier is *universal*. An *existentially quantified sentence* is a quantified sentence whose quantifier is *existential*.

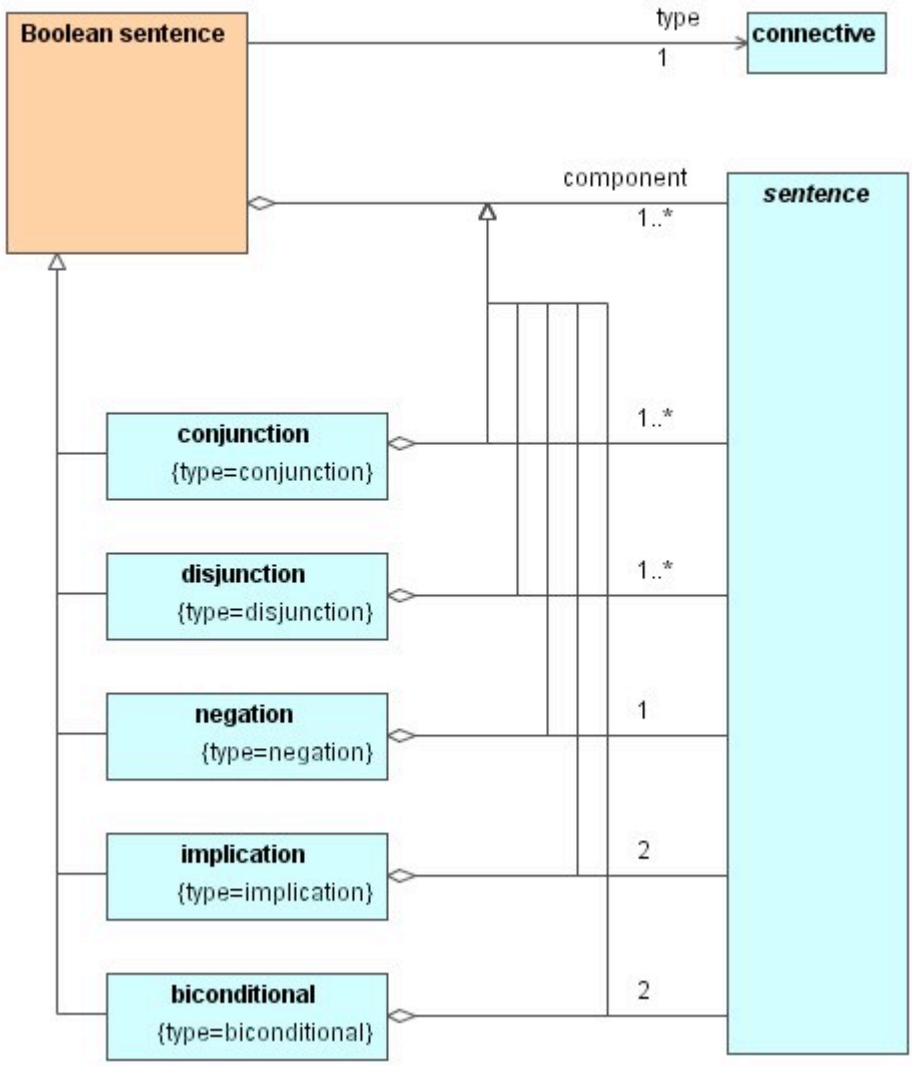

Figure 4 – Abstract syntax of a boolean sentence

Figure 4 depicts the abstract syntax of a Boolean sentence. Two possible readings of the diagram are possible. A sentence representing the conjunction of a number of sentences may be considered as a Boolean sentence with an explicit connective of *conjunction* and whose number of components is restricted by the component relation as specialized to the case where the Boolean sentence is a conjunction or it may be considered as a conjunction where conjunction is equivalent to a Boolean sentence whose type is the *conjunction* connective. Both are considered valid readings.

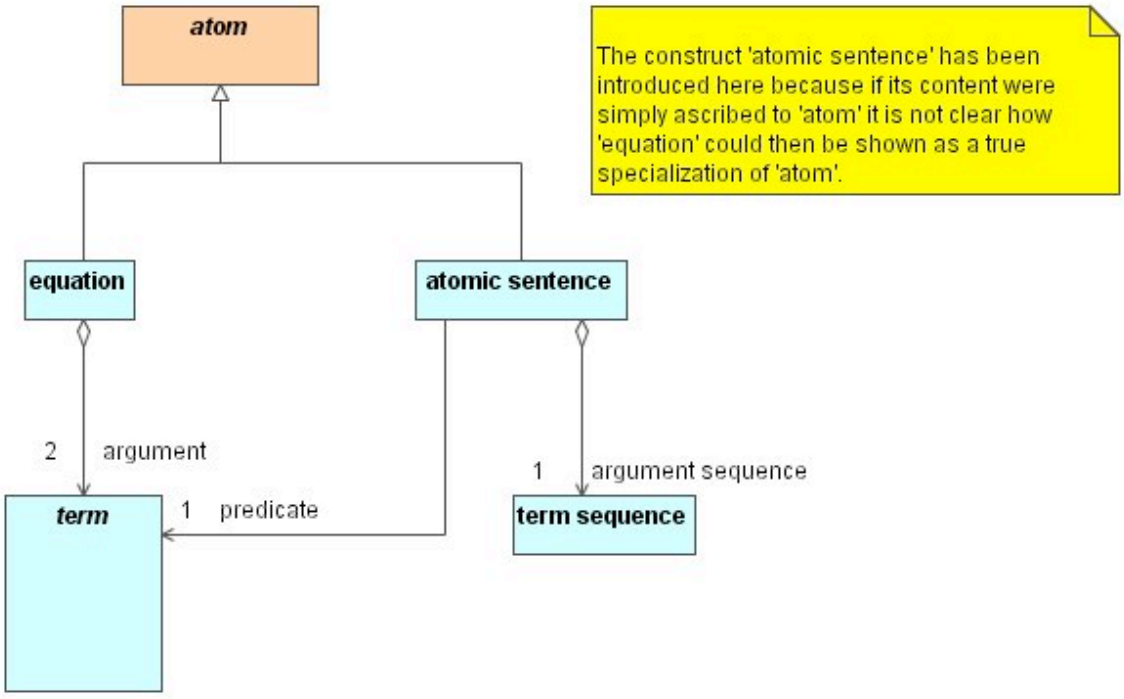

Figure 5 – Abstract syntax of an atom

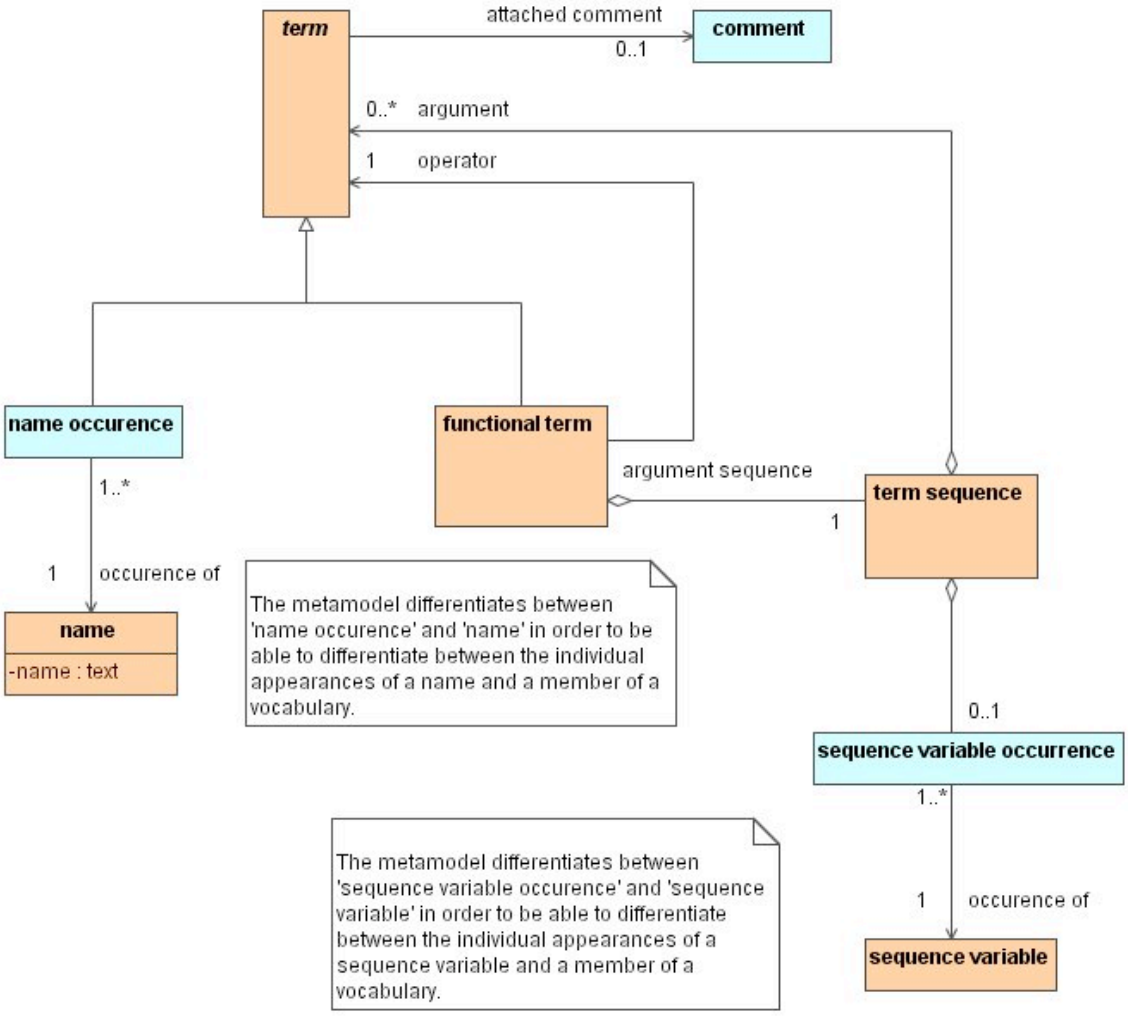

Figure 6 – Abstract syntax of a term and term sequence

6.1.3 Abstract syntactic structure of dialects

This section completely describes the abstract syntactic structure of Common Logic. Any fully conformant Common Logic dialect **must** provide an unambiguous syntactic representation for each of the above types of recognized expressions. No conditions are imposed on how the various syntactic categories are represented in the surface forms of a dialect. In particular, expressions in a dialect are not required to consist of character strings.

A dialect which provides only some types of the Common Logic expressions may be described as a *syntactically partial* Common Logic dialect, or as *syntactically partially conformant*. In particular, a dialect which does not provide for term sequences with a sequence variable, but is otherwise fully conformant, is a *syntactically compact* dialect, and may be described as a *syntactically (fully) conformant compact dialect* if it provides for all other constructions in the abstract syntax. See clause 7 for a description of some relationships between syntactic and semantic conformance.

Dialects **may** in addition provide for other forms of sentence construction not described by this syntax, but in order to be fully conformant such constructions must either be new categories defined in terms of these categories, or be extensions of these categories (e.g. new kinds of Boolean sentence, or kinds of quantifier) which are equivalent in

meaning to a construction using just this syntax, interpreted according to the Common Logic semantics; that is, they can be considered to be systematic abbreviations, or macros; also known as “syntactic sugar”. The CLIF dialect, described in sec. 6, contains a number of syntactic sugared forms for quantified and atomic sentences. (Other types of compliance are also recognized: see section 7 for a full account of conformance.)

The only undefined terms in the above are *name* and *sequence variable*. The only required constraint on these is that they **must be** exclusive. Dialects intended for transmission of content on a network **should not** impose arbitrary or unnecessary restrictions on the form of names, and **must** provide for names to be used as identifiers of Common Logic texts. Dialects intended for use on the Web **should** allow Universal Resource Identifiers and URI references [3] to be used as names. Common Logic dialects **should** define names in terms of Unicode (ISO/IEC 10646) conventions.

Common Logic does not require names to be distinguished from variables, nor does it require names to be partitioned into distinct classes such as relation, function or individual names, or impose sortal restrictions on names. Particular Common Logic dialects **may** make these or other distinctions between subclasses of names, and impose extra restrictions on the occurrence of types of names or terms in expressions – for example, by requiring that bound names be written with a special variable prefix, as in KIF, or with a particular style, as in Prolog; or by requiring that operators be in a distinguished category of relation names, as in conventional first-order syntax.

A dialect **may** impose particular semantic conditions on some categories of names, and apply syntactic constraints to limit where such names occur in expressions. For example, the CLIF syntax treats numerals as having a fixed denotation, and prohibits their use as identifiers.

A dialect **may** require some names to be *non-denoting names*. This requirement may be imposed by, for example, partitioning the vocabulary, or by requiring names which occur in certain syntactic positions to be non-denoting. A dialect with non-denoting names is called *segregated*.

A segregated dialect **must** provide sufficient syntactic constraints to guarantee that in any syntactically legal text of the dialect:

- every name shall be classified as either denoting or as non-denoting;
- no name shall be classified as both denoting and non-denoting;
- no non-denoting name shall be bound by a quantifier;
- the result of substituting a non-denoting name for the bound name in the body of any quantified expression shall not be a legal expression.
- No equation shall contain both a denoting and a non-denoting name

As the presence of non-denoting names affects the semantics, special conditions apply to segregated dialects. The semantic constraints on segregated dialects are described in the next section.

6.2 Common logic semantics

The semantics of Common Logic is defined in terms of a satisfaction relation between Common Logic text and structures called *interpretations*. All dialects **must** apply these semantic conditions to all Common Logic expressions, that is, to any of the forms listed in the previous section. They **may** in addition apply further semantic conditions to subclasses of Common Logic expressions, or to other expressions.

A *vocabulary* is a set of names and sequence variables. The vocabulary of a Common Logic text is the set of names and sequence variables which occur in the text. In a segregated dialect, vocabularies are partitioned into denoting names and non-denoting names.

An *interpretation* I of a vocabulary V is a set U_I , the *universe*, with a distinguished nonempty subset D_I, the domain of discourse, or simply *domain*, and four mappings:

- rel_I from U_I to subsets of D_I*

- fun_I from U_I to functions D_I*->D_I, (which we will also consider to be the set D_I* x D_I)

- int_I from names in V to U_I. If the dialect is segregated, then int_I(x) is in D_I if and only if x is a denoting name. If the dialect recognizes irregular sentences, then they are treated as names of propositions, and int_I also includes a mapping from the irregular sentences of a text to the truthvalues {true, false}.

- seq_I from sequence variables in V to D_I*.

Intuitively, D_I is the domain of discourse containing all the individual things the interpretation is 'about' and over which the quantifiers range. U_I is a potentially larger set of things which might also contain entities which are not in the universe of discourse. All names are interpreted in the same way, whether or not they are understood to denote something in the domain of discourse; this is why there is only a single interpretation mapping applying to all names regardless of their syntactic role. In particular, rel_I(x) is in D_I* even when x is not in D. When considering only segregated dialects, the universe outside the domain may be considered to contain names and can be ignored; when considering only unsegregated dialects, the distinction between universe and domain is unnecessary. The distinction is required in order to give a uniform treatment of all dialects. Irregular sentences are treated as though they were arbitrary propositional variables.

The truth-conditions require some auxiliary definitions.

A *name map* on a vocabulary V (relative to an interpretation I) is any mapping from V into D_I. A *sequence map* on a set of sequence variables is any mapping from those variables to the set D_I*={<x_1,...,x_n>: x_1,...,x_n all in D_I) of finite sequences of elements of D_I. (Note that D* contains the empty sequence <>, for any D.) If A is a name map or sequence map on S, I[A] is the interpretation which is like A on names or sequence variables in S, but otherwise is like I: formally, $U_{I[A]}$ = U_I, $D_{I[A]}$ = D_I, $rel_{I[A]}$ = rel_I, $fun_{I[A]}$ = fun_I, and $int_{I[A]}$(v) = A(v) when v is in S, otherwise $int_{I[A]}$(v)= int_I(v). Note that if x is in the domain of B then $int_{(I[A])[B]}$(x)=B(x).

If E is a subset of D_I, then the *restriction* of I to E is an interpretation K of the same vocabulary and over the same universe and with int_K = int_I and seq_K = seq_I, but where D_K = E, rel_K(v) is the restriction of rel_I(v) to E* and fun_K(v) is the restriction of fun_I(v) to E*->E, for all v in the vocabulary of I. If **S** is a set of names, the *retraction* of I *from* **S**, [I<**S**], is the restriction of I to the set (D_I – {int_I(x): x in **S** }).

If s = <s_1, ..., s_n> and t = <t_1, ..., t_m> are finite sequences, then s;t is the concatenated sequence <s_1, ..., s_n, t_1, ..., t_m>.

The value of any expression E in the interpretation I is given by following the rules in Table 1.

Table 1 – Interpretations of Common Logic Expressions.

	If E is an expression of the form	**then I(E) =**
E1	name **N**	int_I(**N**)
E2	sequence variable **S**	seq_I(**S**)
E3	term sequence $\mathbf{T}_1$ … $\mathbf{T}_n$	<I($\mathbf{T}_1$), … I($\mathbf{T}_n$)>
E4	term sequence $\mathbf{T}_1$ … $\mathbf{T}_n$ with sequence variable **S**	<I($\mathbf{T}_1$), … I($\mathbf{T}_n$)>;I(**S**)
E5	term which is an equation containing terms $\mathbf{T}_1$, $\mathbf{T}_2$	true if I($\mathbf{T}_1$) = I($\mathbf{T}_2$), otherwise false

E6	term with operator **O** and term sequence **S**	$fun_I(I(\mathbf{O}))(I(\mathbf{S}))$ i.e. the x such that <$I(\mathbf{S})$, x> is in $fun_I(I(\mathbf{O}))$
E7	atom with predicate **P** and term sequence **S**	true if $I(\mathbf{S})$ is in $rel_I(I(\mathbf{P}))$, otherwise false
E8	boolean sentence of type negation and component **C**	true if $I(\mathbf{C})$ = false, otherwise false
E9	boolean sentence of type conjunction and components $\mathbf{C}_1$ ... $\mathbf{C}_n$	true if $I(\mathbf{C}_1)$ = … = $I(\mathbf{C}_n)$ = true, otherwise false
E10	boolean sentence of type disjunction and components $\mathbf{C}_1$ ... $\mathbf{C}_n$	false if $I(\mathbf{C}_1)$ = … = $I(\mathbf{C}_n)$ = false, otherwise true
E11	boolean sentence of type implication and components $\mathbf{C}_1$, $\mathbf{C}_2$	false if $I(\mathbf{C}_1)$ = true and $I(\mathbf{C}_2)$ = false, otherwise true
E12	boolean sentence of type biconditional and components $\mathbf{C}_1$, $\mathbf{C}_2$	true if $I(\mathbf{C}_1)$ = $I(\mathbf{C}_2)$, otherwise false.
E13	quantified sentence of type universal and set of names **N** and body **B**	true if for every name map A on **N**, I[A](**B**) is true; otherwise false.
E14	quantified sentence of type existential and set of names **N** and body **B**	false if for every name map A on **N**, I[A](**B**) is false; otherwise true.
E15	irregular sentence **S**	$int_I(\mathbf{S})$
E16	phrase which is a sentence **S**	true if for every sequence map B on the set of sequence variables in **S** , I[B](**S**) is true; otherwise false.
E17	phrase which is an importation containing name **N**	true if $I(text(I(\mathbf{N})))$ = true, otherwise false.
E18	module with name **N**, exclusion set **L** and body text **B**	true if [I<**L**](**B**) = true and ext(I(**N**)) = $D_{[I<\mathbf{L}]}$*, otherwise false.
E19	text containing phrases $\mathbf{S}_1$ … $\mathbf{S}_n$	true if $I(\mathbf{S}_1)$ = … = $I(\mathbf{S}_n)$ = true, otherwise false.

The meaning of the function *text* in the clause for importation is described in the next section.

These are the *basic* logical semantic conditions. In addition, dialects may impose other semantic conditions. A dialect with extra semantic conditions is a *semantic extension*. In particular, semantic extensions may impose syntactic and semantic conditions on irregular sentences, but **must not** use irregular sentence forms to represent content that is expressible in Common Logic text.

A semantic extension which fixes the meanings of certain special names (such as datatypes), or specifies relationships between Common Logic and other naming conventions, such as network identification conventions, is called *external*. External semantic constraints may refer to conventions or structures which are defined outside the model theory itself. For example, the CLIF dialect refers to numbers. The semantics of importations, described in the next section, is external and normative.

Table 1 specifies no interpretation for comments. Phrases consisting of a comment may be considered to be vacuously true; expressions with attached comments **must** have identical truth-conditions as the same expressions with the comments not attached. Thus, adding or deleting comments does not change the truth-conditions of any piece of Common Logic text. Nevertheless, comments are part of the formal syntax and applications **should** preserve them when transmitting, editing or re-publishing Common Logic text. In particular, a name used to identify a piece of Common Logic text is understood to be a globally rigid identifier of that text as written (see next section), so that to use the same name to refer to a different text is an error, even if the texts have the same meaning.

6.3 Importing and identification on a network.

This section applies only to dialects which support importations and/or named texts. It is normative when it applies. (This treatment of naming and identifying is partly based on that in [7].)

The meaning of an importation phrase is that the name it contains shall be understood to identify some Common Logic content, and the importation is true just when that content is true. Thus, an importation amounts to a virtual 'copying' of some Common Logic content from one 'place' to another. This idea of 'place' and 'copying' can be understood only in the context of deploying logical content on a communication network. A *communication network*, or simply a *network*, is a system of agents which can store, publish or process Common Logic text, and can transmit Common Logic text to one another by means of information transfer protocols associated with the network. The most widely used network is the World Wide Web [8], but other networks are possible. In particular, a subset of Web nodes which uses special conventions for communication may be considered to be a Common Logic network. A network is presumed to support communication and publication of Common Logic content in some subset of dialects. XCL is intended to be a general-purpose dialect for distributing Common Logic content on any network which supports XML.

Names used to name texts on a network are understood to be *rigid* and to be *global* in scope, so that the name can be used to identify the thing named – in this case, the Common Logic text – across the entire communication network. (See [3] for further discussion.) A name which is globally attached to its denotation in this way is an *identifier*, and is typically associated with a system of conventions and protocols which govern the use of such names to identify, locate and transmit pieces of information across the network on which the dialect is used. While the details of such conventions are beyond the scope of this document, we can summarize their effect by saying that the act of publishing a named Common Logic text is intended to establish the name as a rigid identifier of the text, and Common Logic acknowledges this by requiring that *all* interpretations shall conform to such conventions when they apply to the network situation in which the publication takes place.

Named texts are not required to be in 1:1 correspondence to documents, files or other units of data storage. Dialects or implementations may provide for texts to be distributed across storage units, or for multiple named texts to be stored in one unit. The naming conventions for text may be related to the addressing conventions in use for data units, but this is not required. Texts may also be identified by external naming conventions, for example by encoding the text in documents or files which have network identifiers; the semantics described in this section **must** be applicable to all names used as network identifiers.

It is important here to distinguish the act of naming from that of asserting the truth of the text itself. Publishing a named text does not, in itself, necessarily make any claim about the truth of the text; but it does make a claim about the denotation of the name of the text.

In order to state semantic conditions on identifiers we need to assume appropriate values to exist in the universe of discourse. We will call the semantic entity corresponding to a named text a *named text value*. The exact nature of a text value is unimportant, but the semantics considers them to be pairs consisting of a name and a Common Logic text: **t** = <*name*(**t**), *text*(**t**)>. The rigid identifier convention is an external semantic condition which *all* interpretations of texts published on the communication network are required to satisfy. The global rigidity of the naming is captured by the universality of this requirement. Note that this is an external semantic condition since it refers to a structure defined by the network protocols. It may be considered to be a semantic condition on the network.

if **t** is a text value in U_I and name(**t**) is in V, then int_I(name(**t**)) = **t**.

The publication of a text with a name on a communication network is considered to be an assertion of the existence of an appropriate named text value, with global scope, i.e. one that *all* interpretations of *any* text available on the network are required to acknowledge:

publication on the network of:	**requires that for any interpretation *I* of a text on the network:**
a text **T** with a name **N**	U_I contains a named text value **t** with *text*(**t**) = **T** and *name*(**t**) = **N**

This notion of importation amounts to a virtual copying of one piece of text into another. (In fact, it is a virtual copying of the *importation closure*, since one has to consider the case where the imported text itself contains an importation of another text.) Such a relationship between texts makes an implicit assumption that the texts can be interpreted together, and the truth-conditions given above reflect this by applying the interpretation of the importing phrase directly to the imported text. This means, in effect, that any use of this notion of textual importing must be based on the assumption that the texts are mutually interpretable. For example, importing implies that the quantifiers in the imported text will be interpreted to range over the same universe as those in the importing text. All texts which are published and identified on a network **must** be mutually interpretable with all other texts on the network which can import them, over the same universe and domain of discourse, and with their vocabularies merged. This condition applies to all texts which might possibly import other texts, even if they do not in fact do so in a particular state of the network.

Real networks, being implementations, are subject to errors or breakdowns. The rigid naming conventions described in the section are understood to apply even under such failure conditions. Thus for example if a URI is used on the Web to be a rigid identifier of some text, then it remains an identifier even when an attempt to use it in an HTTP *get* protocol produces a 404 error. Applications **must not** treat communication errors or failures as an indication that a name is non-denoting.

6.3.1 Mixed networks

Text may be published in more than one dialect on a single network. This document refers to such a situation as a *mixed* network. Information exchange and publication on a mixed network must be conducted in such a way that all agents can represent content written in any text in use on the network. One way to achieve this is to use the most permissive dialect for information transmission and require agents to express their content in this dialect.

In order to maintain mutual interpretability, any text in a segregated dialect which is published on a mixed network **must** be published in such a way that any importing of that text into another text written in an unsegregated dialect can express the content of the imported text in a way that allows mutual interpretability. This means in particular that a name must be provided for the domain of discourse of text in any segregated text, and that any non-denoting names occurring in such text can be recognized efficiently by applications which process unsegregated text. The recommended practice in such cases is that the segregated text be replaced by unsegregated text in which all quantifiers are restricted or guarded by the segregated domain name, and all non-denoting names are asserted to be outside that domain. Modules provide a general-purpose technique for such publication; the segregated text can be published as the body text of a module, with the non-denoting names which occur in the text included in the exclusion list of the module. The module name **may** be used to identify a common domain of discourse associated with the dialect, or a local domain of discourse special to the text in the module.

Networks supporting segregated dialects which have lexical conventions for distinguishing denoting from non-denoting names **may** require agents to recognize such lexical distinctions even when using segregated text, and apply suitable translations where needed, as part of the transfer protocol. However, such conventions cannot support information exchange outside that network, so are not considered to be fully conformant.

6.4 Satisfaction, validity and entailment.

A Common Logic set of sentences, or text, T is *satisfied* by an interpretation I just when I(S)=true for every S in T. A text is *satisfiable* if there is an interpretation which satisfies it, otherwise it is *unsatisfiable*, or *contradictory*. If every interpretation which satisfies S also satisfies T, then S *entails* T.

Common logic interpretations treat irregular sentences as opaque sentence variables. In a dialect which supports irregular sentences, the above definitions are used to refer to interpretations determined by the semantics of the dialect; however, when qualified by the prefixing adjective or adverb "common-logic", as in "common-logic entails", they shall be understood to refer to interpretations which conform exactly to the Common Logic semantic conditions. For example, a dialect might support modal sentences, and its semantics support the entailment *(Necessary P) entails P*; but this would not be a common-logic entailment, even if the language was conformant as a Common Logic extension. However, the entailment *(Necessary P) entails (Necessary P)* is a common-logic entailment.

Several of the later discussions consider restricted classes of interpretations. All the above definitions may be qualified to apply only to interpretations in a certain restricted class. Thus, S *foo*-entails T just when for any interpretation I in the class *foo*, if I satisfies S then I satisfies T. Entailment (or unsatisfiability) with respect to a class of interpretations implies entailment (or unsatisfiability) with respect to any subset of that class.

When describing entailment of T from S, S is referred to as the *antecedent*, and T the *conclusion*, of the entailment

6.5 Summary of CLIF

The Common Logic core syntax (referred to herein as CLIF) is an unsegregated dialect, based on KIF [2], that is used to give examples in this document. A full description of CLIF is given in Annex A. Here CLIF forms for the Common Logic classes are briefly indicated. [**T**] indicates the CLIF representation of the expression **T** . This is **not** a normative or complete description of the CLIF syntax. The normative description is specified in Annex A.

Table 2 – Summary of the CLIF syntax with respect to Common Logic abstract syntax.

sequence variable **S**	a string of Unicode characters, other than '(', ')' or white space, beginning with '…'
name **N**	a string of Unicode characters, other than '(', ')' or white space, and which is not a sequence variable.
term sequence $\mathbf{T}_1$ … $\mathbf{T}_n$	a sequence of terms written left to right, separated by white space: [$\mathbf{T}_1$] … [$\mathbf{T}_n$]
term sequence $\mathbf{T}_1$ … $\mathbf{T}_n$ with sequence variable **S**	as above, with a sequence variable at the right: [$\mathbf{T}_1$] … [$\mathbf{T}_n$] [**S**]
term which is an equation containing terms $\mathbf{T}_1$, $\mathbf{T}_2$	(= [$\mathbf{T}_1$] [$\mathbf{T}_2$])
term with operator **O** and term sequence **S**	([**O**] [**S**])
atom with predicate **P** and term sequence **S**	([**P**] [**S**])

boolean sentence of type negation and component **C**	(not [**C**])
boolean sentence of type conjunction and components $\mathbf{C}_1$ ... $\mathbf{C}_n$	(and [$\mathbf{C}_1$] … [$\mathbf{C}_n$])
boolean sentence of type disjunction and components $\mathbf{C}_1$ ... $\mathbf{C}_n$	(or [$\mathbf{C}_1$] … [$\mathbf{C}_n$])
boolean sentence of type implication and components $\mathbf{C}_1$, $\mathbf{C}_2$	(implies [$\mathbf{C}_1$] [$\mathbf{C}_2$])
boolean sentence of type biconditional and components $\mathbf{C}_1$, $\mathbf{C}_2$	(iff [$\mathbf{C}_1$] [$\mathbf{C}_2$])
quantified sentence of type universal and set of names **N** and body **B**	(forall ([**N**]) [**B**])
quantified sentence of type existential and set of names **N** and body **B**	(exists ([**N**]) [**B**])
phrase which is a sentence **S**	[**S**]
phrase which is an importation containing name **N**	(cl:imports [**N**])
module with name **N**, exclusion set **L** and body text **B**	(cl:module [**N**] (cl:excludes [**L**]) [**B**])
text containing phrases $\mathbf{S}_1$ … $\mathbf{S}_n$	a sequence of phrases written left to right, separated by white space: [$\mathbf{S}_1$] … [$\mathbf{S}_n$]
Named text with name **N** and phrases $\mathbf{S}_1$ … $\mathbf{S}_n$	(cl:text [**N**] [$\mathbf{S}_1$ … $\mathbf{S}_n$])

6.6 Sequence variables, recursion and argument lists: discussion

Sequence variables take Common Logic beyond first-order expressivity. A sequence variable stands for an arbitrary sequence of arguments. Since sequence variables are implicitly universally quantified, any expression containing a sequence variable has the same semantic import as the *infinite* conjunction of all the expressions obtained by replacing the sequence variable by a finite sequence of names, all universally quantified at the top (phrase) level.

This ability to represent infinite sets of sentences in a finite form means that Common Logic with sequence variables is not compact, and therefore not first-order; for clearly the infinite set of sentences corresponding in meaning to a simple atomic sentence containing a sequence variable is logically equivalent to that sentence and so entails it, but no finite subset of the infinite set does. However, the intended use of sentences containing sequence variables is to act as axiom schemata, rather than being posed as goals to be proved, and when they are restricted to this use the resulting logic is compact. Also, even without this restriction, Common Logic is finitely complete, in the sense there are inference schemes which can derive T from S if S entails T and S is finite. Since Common Logic sentences can express the same content as infinite sets of conventional first-order sentences, the limitation to finite antecedents is less restrictive than it might seem; in fact, this completeness is a strengthening of Gödel's classical first-order completeness result.

A truncated dialect which does not support sequence variables can imitate much of the functionality provided by sequence variables, by the use of explicit argument lists, represented in Common Logic by terms with a special operator, which itself can then be regarded as part of the syntax of the dialect. This convention is widely used in logic programming applications and in RDF and OWL. The costs of this technique are a considerable reduction in syntactic clarity and readability, the need to allow lists as entities in the universe of discourse, and possibly the reliance on external software to manipulate the lists. The advantage is the ability of rendering arbitrary argument sequences using only a small number of primitives. Implementations based on argument-list constructions are often limited to conventional first-order expressivity, and fail to support all inferences involving quantification over lists. This may be considered either as an advantage or as a disadvantage.

6.7 Special cases and translations between dialects.

A segregated dialect in which all operators and predicates are non-denoting names is called a *classical* dialect.

An interpretation I is *flat* when $D_I = U_I$. It is *extensional* when rel_I and fun_I are the identity function on $(U_I\text{-}D_I)$, so that the entities in the universe outside the domain are the extensions of the non-denoting names. Both flat and extensional interpretations can be described with mentioning the universe. These are appropriate for, respectively, an unsegregated dialect, and a classical dialect. The general form of interpretation described above allows both kinds of dialect, and others, to be interpreted by a single construction.

For unsegregated dialects, only flat interpretations need be considered: for given any interpretation I there is a flat interpretation J which satisfies the same expressions of any text of the dialect as I does. J may be obtained by simply declaring U_J to be D_I; for an unsegregated dialect, all names denote in D_I so elements outside D_I are irrelevant to the truth-conditions.

For classical dialects, only extensional interpretations need be considered: for given any interpretation I there is an extensional interpretation J which satisfies the same expressions of any text of the dialect as I does. J may be obtained by replacing I(x) by $fun_I(I(x))$ for every operator and by $rel_I(I(x))$ for every predicate x in the vocabulary, and removing them from the domain if they are present. Since all operator and predicate names in a classical dialect influence the truth-conditions only through their associated extensions, this does not affect any truth-values. Formally, $D_J = D_I$ - {I(v): v an operator or predicate in V}, $int_J(x)=int_I(x)$ for denoting names, $int_J(x)=rel_I(int_I(x))$ for predicates x and $int_J(x)=fun_I(int_I(x))$ for operators x.

6.7.1 Translating between dialects

A *translation* is a mapping from texts in a dialect A to texts in a dialect B, the target dialect, such that for every interpretation I of A there is an interpretation J of B, and for every interpretation J of B there is an interpretation I of A, with I(A)=J(B). Since all Common Logic dialects have the same truth-conditions, translation is usually straightforward. Complications arise however in translating between segregated and unsegregated dialects.

Translation from a segregated dialect A into an unsegregated dialect B requires the translation to indicate which terms are non-denoting in A. Since all names in the unsegregated dialect denote, it is necessary for the translation to introduce a *domain name* whose extension in B is the domain of an interpretation of A, and the for the translation to restrict all quantifiers in the text to range over this domain, and assert that non-denoting names of the segregated dialect denote entities outside this domain. No other translation is required. The module construction provides a general-purpose technique for such translations: text in A has the same meaning as a module in B named with the domain name and with the non-denoting names of the text listed in the exclusion list of the module.

Translation from an unsegregated dialect B into a segregated dialect A requires that names are used so as to respect the restrictions of the dialect. This may require adding axioms to the translations in order to ensure that the domain of an interpretation of the segregated translation of any text corresponds to the universe of an interpretation of the unsegregated text. There is a general technique called the *holds-app translation* for translating any Common Logic dialect into a similar classical dialect. An atomic sentence with predicate P and argument term sequence S_1 … S_n translates into an atomic sentence with predicate *holds* and argument sequence P S_1 … S_n. A term with operator

O and argument sequence S_1 … S_n translates into a term with operator *app* and argument sequence O S_1 … S_n . The introduced predicate and operator require no other axioms: their only role is to allow the operators and predicates of the B dialect to denote entities in the domain of the A dialect translation.

Some dialects impose notational restrictions of various kinds, such as requiring bound names to have a particular lexical form, or requiring operator and predicate names to be used with a particular length of argument sequence (conventionally called the *arity* of the operator or predicate). Translation into a dialect with such restrictions can usually be done by re-writing names to conform to the restrictions and by 'de-punning' occurrences of a name which must be made distinct in the target dialect, for example by adding suffices to indicate the arity. Applications which are required to faithfully translate multiple texts must maintain consistency between such name re-writings.

7 Conformance

There are three kinds of conformance that can be specified for Common Logic. There can be conditions on a dialect (i.e., the specification of a language), conditions on an application (that conforms to the standard) and conditions on a network.

7.1 Dialect Conformance

These are really conditions on a *specification* of a language or notation, in order for it to count as a CL dialect. Conformance is specified in two ways: syntactic and semantic. A dialect's syntactic and semantic conformance can be specified separately, although not all combinations may be useful or meaningful.

7.1.1 Syntax

A dialect is defined over some set of inscriptions, which **must** be specified. (Commonly this will be Unicode character strings, but other inscriptions eg diagrammatical representations such as directed graphs or structured images) are possible. A method **must** be specified for the dialect which will unambiguously parse any inscription in the set or reject it as illegal. (For Unicode string inscriptions, a grammar in EBNF is a sufficiently precise specification.) A *parsing* is an assignment of each part of a legal inscription into its corresponding CL abstract syntax category in section 6.1.1, and the parsed inscription is an *expression*.

A dialect is ***syntactically fully conformant*** if its parsings recognize expressions for every category of the abstract syntax in section 6.1.1. Conformant dialects or sub-dialects whose parsings include other categories of sentences **should** categorize them as irregular sentences for Common Logic conformity; such dialects or sub-dialects will be referred to as *semantic extensions* (see section 7.1.2 below). It is conformant as a ***syntactic sub-dialect*** if it recognizes only some of the CL categories; but any dialect **must** recognize some form of sentence category. We recognize one particular case we call ***non-sequence sub-dialect*** which is a dialect that recognizes all categories except the sequence variable.

A dialect is *syntactically segregated* if the parsing requires a distinction to be made between categories of CL names in order to check legality of an expression in that dialect. Segregated dialects **must** specify criteria which are sufficient to enable an application to detect the category of a name in the dialect without performing operations on any structure other then the name itself. (Note, this may be done 'invisibly', by defining names in the dialect to be pairs of an inscription and a hidden lexical type code, for example.) Every possible name in a segregated dialect **must** be unambiguously classifiable into exactly one lexical category of that dialect.

7.1.2 Semantics

Any CL dialect must have a model-theoretic semantics, defined on a set of interpretations, called *dialect interpretations*, which assigns one of the two truth-values *true* or *false* to every sentence, phrase or text in that dialect.

A dialect is ***exactly semantically conformant*** when, for any syntactically legal sentence, phrase or text T in that dialect, the following two (separate) conformance conditions are true:

- For every dialect interpretation $\mathcal{J}$ of T, there exists a Common Logic interpretation $\mathcal{I}$ of T such that $\mathcal{I}$(T) = $\mathcal{J}$ (T)

- For any Common Logic interpretation I of T, there exists a dialect interpretation J of T such that I(T) = J (T)

It follows that the notions of satisfiability, contradiction and entailment corresponding to the dialect interpretations, and to Common Logic interpretations, are identical for an exactly conforming dialect.

Syntactically segregated dialects may be required to satisfy additional conditions, see below.

The simplest way to achieve exact semantic conformance is to adopt the CL model theory as the model-theoretic semantics for the dialect, but the definition is phrased so as to allow other ways of formulating the semantic meta-theory to be used if they are preferred for mathematical or other reasons, provided only that satisfiability, contradiction and entailment are preserved.

A *semantic sub-dialect* is a syntactic sub-dialect (see section 7.1.1 above) and meets the first condition; that is, it covers only some parts of the full Common Logic but its interpretations are identical to Commmon Logic's for those parts.

A ***semantic extension*** is a dialect which satisfies the second condition, but does not satisfy the first condition. In other words, a semantic extension dialect has some part(s) whose interpretation is more constrained than they would be by a CL interpretation. Such parts are to be considered irregular sentences as described in Table 1 and section 6.2.

This allows a semantic extension to apply "external" semantic conditions, for example to irregular sentences, in addition to the CL semantic conditions. CLIF is an example of a semantic extension, by virtue of the conditions on numbers and quoted strings and datatypes, for example **`(= (xsd:integer "3") 3)`** is logically true in CLIF but has CL interpretations in which it is false.

Semantic extensions should be referred to as "conforming semantic extension" or "conforming extension", rather than as exactly conformant or simply as "conformant". For sentences, phrases and texts of a conforming extension, contradiction and entailment with respect to the Common Logic semantics implies respectively contradiction and entailment with respect to the dialect semantics, but not vice versa; and satisfaction with respect to the dialect semantics implies satisfaction with respect to Common Logic semantics, but not vice versa. This means that inference engines which perform Common Logic inferences will be correct, but may be less complete, for the dialect.

A ***segregated dialect*** is a syntactically segregated dialect which requires names in one or more categories to not denote entities in the set over which its quantifiers range. For example, traditional first-order logic syntax may be interpreted in a way which requires that relation names not denote individuals. In order to be conformant, segregated dialects must, in addition to being semantically conformant, (a) provide syntactic criteria which are sufficient to enable an application to detect that a dialect name is non-denoting, and (b) as part of the publication of any published sentences, phrases or texts of the dialect, provide a name which can be used by other dialects to refer to the universe of discourse of the published sentences, phrases or texts. That is, the dialect must specify, as a semantic condition in all dialect interpretations, that the relational extension of this name, when used as a predicate, shall be true of precisely the entities in the domain of the interpretation. The module construct in the abstract syntax is intended to facilitate this conformance requirement.

No dialect may restrict the range of quantification of a different dialect. Other dialects may treat all names as denoting names, even those which are declared in a segregated dialect to be non-denoting.

7.2 Applications

"Application" means any piece of computational machinery (software or hardware, or a network) which performs any operations on CL text (even very trivial operations like inputting it and storing it for later re-transmission.)

Conformance of applications is defined relative to a collection of dialects, called the *conformance set*. Applications which are conformant for the XCL dialect may be referred to as 'conformant' without qualification. (This is because the idea of XCL is that it can include text from any other dialect, so it's a kind of universal solvent.)

All conformant applications **must** be capable of processing all legal inscriptions of the dialects in the conformance set. Applications which input, output or transmit CL text, even if embedded inside text processed using other textual conventions, **must** be capable of round-tripping any CL text; that is, they must output or transmit the exact inscription that was input to them, without alteration.

Applications which detect entailment relationships between CL texts in the conformance set are **correct** when, for any texts T and S in dialects in the conformance set, if the application detects the entailment of T from S then S common-logic entails T (that is, for any Common Logic interpretation I, if I(S) =true then I(T)=true). The application is **complete** when, for any texts T and S in dialects in the conformance set, if S common-logic entails T then the application can detect the entailment of T from S. (Note this requires completeness 'across' dialects in the conformance set).

Completeness does not require that the application can detect entailment in a semantic extension. If a dialect is a semantic extension, then an application is **dialect complete** for that dialect if, for any dialect interpretation I of that dialect, I(T)=true whenever I(S)=true, then the application detects the entailment of T by S. Dialect completeness for D implies completeness for {D}, but not vice versa.

7.3 Networks

Conformance of communication networks is defined relative to a collection of dialects, called the conformance set. A network is conformant when it transmits all expressions of all dialects in the conformance set without distortion from any node in the network to any other node. Network transmission errors or failures which are indicated as error conditions do not count as distortion for purposes of conformance of a network.

Needs some stuff here on treating identifiers as globally rigid. This might need some thought, eg do we allow dynamically changing texts on a network, such as might be produced by a news feed? The Web stuff in RFC 2396 and REST (see http://webservices.xml.com/pub/a/ws/2002/02/06/rest.html for a nice overview) treats 'resources' as functions from times to states but requires identification of a resource to be rigid, which is an interesting way to try to get the best of both. This needs some work.

Annex A
(normative)

Common Logic Interchange Format (CLIF)

A.1 Introduction

Historically, the Common Logic project arose from an effort to update and rationalize the design of KIF [2], which was first proposed as a 'knowledge interchange format' over a decade ago and, in a simplified form, has become a *de facto* standard notation in many applications of logic. Several features of Common Logic, most notably its use of sequence variables, are explicitly borrowed from KIF. However, the design philosophy of Common Logic differs from that of KIF in various ways, which we briefly review here.

First, the goals of the languages are different. KIF was intended to be a common notation into which a variety of other languages could be translated without loss of meaning. Common Logic is intended to be used for information interchange over a network, as far as possible without requiring any translation to be done; and when it must be done, Common Logic provides a single common *semantic* framework, rather than a syntactically defined interlingua.

Second, largely as a consequence of this, KIF was seen as a 'full' language, containing representative syntax for a wide variety of forms of expressions, including for example quantifier sorting, various definition formats and with a fully expressive meta-language. The goal was to provide a single language into which a wide variety of other languages could be directly mapped. Common Logic, in contrast, has been deliberately kept 'minimal': the kernel in particular is a tiny language. This makes it easier to state a precise semantics and to place exact bounds on the expressiveness of subsets of the language, and allows extended languages to be defined as encodings of axiomatic theories expressed in Common Logic, or by reduction to the Common Logic kernel.

Third, KIF was based explicitly on LISP. KIF syntax was defined to be LISP S-expressions; and LISP-based ideas were incorporated into the semantics of KIF, for example in the way that the semantics of sequence variables were defined. Although the CLIF surface syntax retains a superficially LISP-like appearance in its use of a nested unlabelled parentheses, and could be readily parsed as LISP S-expressions, Common Logic is not LISP-based and makes no basic assumptions of any LISP structures. The recommended Common Logic interchange notation is based on XML, a standard which was not available when KIF was originally designed.

Finally, many of the 'new' features of Common Logic have been motivated directly by the ideas arising from new work on languages for the semantic web [9], and in particular, Common Logic is intended to be useable as a slightly improved Lbase [10], i.e. as serving the role of a semantic reference language for other SW notations.

The name chosen for Common Logic's KIF-like syntax is the Common Logic Interchange Format (CLIF). This is primarily to identify it as the version being prescribed in this standard, and to distinguish it from various other dialects of KIF that may or may not be exactly compatible.

A.2 CLIF Syntax

The following syntax is written using *Extended Backus-Naur Form* (EBNF), as specified by ISO/IEC 14977. Literal chararacters are **'quoted'**, sequences of items are separated by commas, **|** indicates a separation between alternatives, **{ }** indicates a sequence of expressions, **-** indicates an exception, **[ ]** indicates an optional item, and parentheses **( )** are used as grouping characters. Productions are terminated with **;**.

The syntax is written to apply to ASCII encodings. It also applies to full Unicode character encodings, with the change noted below to the category *nonascii*.

The syntax is presented here in two parts. The first deals with parsing character streams into lexical items: the second is the logical syntax of CLIF, written assuming that lexical items have been isolated from one another by a lexical analyser. This way of presenting the syntax allows the expression syntax to ignore complications arising from whitespace handling.

A.2.1 Characters

Any CLIF, or core, expression is encoded as a sequence of Unicode characters as defined in ISO/IEC 10646. **[KIF requires ASCII encodings.]** Any character encoding which supports the repertoire of ISO 10646 may be used, but UTF-8 (ISO 10646 Annex D) is preferred. Only characters in the US-ASCII subset are reserved for special use in CLIF itself, so that the language can be encoded as an ASCII text string if required. This document uses ASCII characters. Unicode characters outside the ASCII range are represented in CLIF ASCII text by a character coding sequence of the form \u*nnnn* or \U*nnnnnn* where *n* is a hexadecimal digit character. When transforming an ASCII text string to a full-repertoire character encoding, or when printing or otherwise rendering the text for maximum accessibility for human readers, such a sequence **may** be replaced by the corresponding direct encoding of the character, or an appropriate glyph. Moreover, these coding sequences are understood as denoting the corresponding Unicode character when they occur in quoted strings (see below).

The syntax is defined in terms of disjoint blocks of characters called *lexical tokens* (in the sense used in ISO/IEC 2382-15). A character stream can be converted into a stream of lexical tokens by a simple process of lexicalisation which checks for a small number of *delimiter* characters, which indicate the termination of one lexical token and possibly the beginning of the next lexical token. Any consecutive sequence of whitespace characters acts as a *separator* between lexical tokens (except within quoted strings and names, see below). Certain characters are reserved for special use as the first character in a lexical item. The double-quote(U+0022) character is used to start end names which contain delimiter characters, the single-quote (apostrophe U+002C) character is used to start and end quoted strings, which are also lexical items which may contain delimiter characters, and the equality sign must be a single lexical item when it is the first character of an item.

The backslash \ (reverse solidus U+005C) character is reserved for special use. Followed by the letter u or U and a four- or six-digit hexadecimal code respectively, it is used to transcribe non-ASCII Unicode characters in an ASCII character stream, as explained above. Any string of this form in an ASCII string rendering plays the same Common Logic syntactic role as a single ordinary character. The combination \' (U+005C, U+002C) is used to encode a single quote inside a Common Logic quoted string, and similarly the combination \" (U+005C, U+0022) indicates a double quote inside a double-quoted enclosed name string. These inner-quote conventions apply in both ASCII and full Unicode renderings. Any other uses of the backslash are treated simply as single U+005C characters.

A.2.2 Lexical syntax

```
white = space U+0020 | tab U+0009 | line U+0010 | page U+0012 | return U+0013 ;
```

Single quote (apostrophe) is used to delimit quoted strings, and double quote to delimit enclosed names, which obey special lexicalization rules. **[KIF does not use the enclosed-name device.]** Quoted strings and enclosed names are the only CLIF lexical items which can contain whitespace and parentheses. Parentheses elsewhere are self-delimiting; they are considered to be lexical tokens in their own right. Parentheses are the primary grouping device in CLIF syntax.

```
open = '(' ;

close = ')' ;

stringquote = ''' ;

namequote=  '"'
```

char is all the remaining ASCII non-control characters, which can all be used to form lexical tokens (with some restrictions based on the first character of the lexical token). This includes all the alphanumeric characters.

```
char = digit | '~' | '!' | '#' | '$' | '%' | '^' | '&' | '*' | '_' | '+' | '{' | '}' | '|' |
':' |   '<' | '>' | '?' | '`' | '-' | '=' | '[' | ']' | '\' | ';'| ',' | '.' | '/' | 'A' |
'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' |
'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | 'a' | 'b' | 'c' | 'd' | 'e' |
'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' |
'u' | 'v' | 'w' | 'x' | 'y' | 'z' ;

digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;

hexa = digit | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' ;
```

Certain character sequences are used to indicate the presence of a single character. *nonascii* is the set of characters or character sequences which indicate a Unicode character outside the ASCII range. For input using a full Unicode character encoding, this production should be ignored and *nonascii* should be understood instead to be the set of all non-control characters of Unicode outside the ASCII range which are supported by the character encoding. The use of the \u*xxxx* and \U*xxxxxx* sequences in text encoded using a full Unicode character repertoire is **deprecated**.

innerstringquote is used to indicate the presence of a single-quote character inside a quoted string. A quoted string can contain any character, including whitespace; however, a single-quote character can occur inside a quoted string only as part of an *innerstringquote*, i.e. when immediately preceded by a backslash character. The occurrence of a single-quote character in the character stream of a quoted string marks the end of the quoted string lexical token unless it is immediately preceded by a backslash character. Inside enclosed name strings, double quotes are treated exactly similarly. *Innernamequote* is used to indicate the presence of a double-quote character inside an enclosed name.

```
nonascii = '\u' , hexa, hexa, hexa, hexa | '\U' , hexa, hexa, hexa, hexa, hexa, hexa ;

innerstringquote ='\'';

innernamequote ='\"';

numeral = digit , { digit } ;
```

Sequence variables are a distinctive syntactic form with a special meaning in Common Logic. Note that a bare ellipsis **'...'** is a sequence variable. (Note, this document uses ellipses conventionally in text and when indicating repetitive infinite structures, but their use inside CLIF syntax, indicated by the special font, should be understood as part of the formal syntax.) **[KIF uses the '@' symbol as a sequence variable prefix.]**

```
seqvar = '...' , { char } ;
```

Single quotes are delimiters for quoted strings; double quotes for enclosed names. An enclosed name is simply a name which may contain characters which would break the lexicalization, such as "Mrs Norah Jones" or "Girl(interrupted)"; like any other name, it may denote anything. The surrounding double-quote marks are not considered part of the denoting name, which is defined to be the character string obtained by removing the enclosing double-quote marks and replacing any internal occurrences of an innernamequote by a single double-quote character. A quoted string, in contrast, is an expression with a fixed semantic meaning: it *denotes* a text string similarly related to the string inside the quotes. The combination \' indicates the presence of a single quote mark in the denoted string. With these conventions, a quoted string denotes the string enclosed in the quote marks. For example, the quoted string 'a\(b\'c' denotes the string a\(b'c, and '\'a (b\\\') c \\" denotes the string: 'a (b\\') c \'.

Quoted strings and enclosed names require a different lexicalization algorithm than other parts of CLIF text, since parentheses and whitespace do not break a quoted text stream into lexical tokens.

When CLIF text is enclosed inside markup which uses character escaping conventions, the Common Logic quoted string conventions here described are understood to apply to the text described or indicated by the conventions in use, which should be applied first. Thus for example the content of the XML element: **<cl-text>'a\'b<c&apos</cl-text>** is the CLIF syntax quoted string 'a\'b<c' which denotes the five-character text string a'b<c . Considered as bare CLIF text, however, **'a\'b<c&apos** would simply be a rather long name.

```
quotedstring = stringquote, { white | open | close | char | nonascii | namequote |
innerstringquote }, stringquote ;

 enclosedname   = namequote, { white | open | close | char | nonascii | stringquote |
innernamequote }, namequote ;
```

reservedelement consists of the lexical tokens which are used to indicate the syntactic structure of Common Logic expressions. These may not be used as names in core text.

```
reservedelement = '=' | 'and' | 'or' | 'iff' | 'implies' | 'forall' | 'exists' | 'not' | 'roleset:' | ‘cl:text’ |
'cl:imports'|'cl:excluded'|'cl:module'|'cl:comment';
```

A *namesequence* is a lexical token which does not start with any of the special characters. Note that namesequences may not contain whitespace or parentheses, and may not start with a quote mark although they may contain them; and that numerals are syntactically similar to namesequences but are distinguished for semantic reasons.

```
namesequence = ( char , { char | stringquote | namequote} ) - ( reservedelement | numeral |
seqvar ) ;
```

The task of a lexical analyser is to parse the character stream into consecutive, non-overlapping lexbreak and nonlexbreak strings, and to deliver the lexical tokens it finds as a stream of tokens to the next stage of syntactic processing. Lexical tokens are divided into eight mutually disjoint categories: the open and closing parentheses, numerals, quoted strings (which begin and end with '`'), seqvars (which begin with '...'), reserved elements, enclosed names (which begin and end with "") , and namesequences.

```
lexbreak  = open | close | white , { white } ;

nonlexbreak   = numeral | quotedstring | seqvar | reservedelement | namesequence   |
enclosedname ;

lexicaltoken  = open | close | nonlexbreak ;

charstream = { white }, { lexicaltoken,  lexbreak } ;
```

A.2.3 Expression syntax

This part of the syntax is written so as to apply to a sequence of Common Logic lexical tokens rather than a character stream.

Both terms and atomic sentences use the notion of a sequence of terms representing a vector of arguments to a function or relation. A term sequence may optionally end in a seqvar.

```
termseq = { term } , seqvar? ;
```

A *name* is any lexical token which is understood to denote.

```
name = namesequence | numeral | enclosedname | quotedstring ;
```

Names count as terms, and a complex (application) term consists of a function with a vector of arguments. Terms may also have an associated comment, represented as a quoted string (in order to allow text which would otherwise break the lexicalization). Comments enclose the term they comment upon. **[KIF handles comments differently and does not have the 'enclosing' construction.]**

```
term = name | ( open, term , termseq, close ) | ( open, 'cl:comment', quotedstring , term, close ) ;
```

Equations are distinguished as a special category because of their special semantic role, and special handling by many applications. The equality sign is not a term.

```
equation = open, '=', term, term, close
```

Atomic sentences are similar in structure to terms, but in addition the arguments to an atomic sentence may be represented using role-pairs consisting of a role-name and a term. **[KIF does not have the role-pair construction.]** Equations are considered to be atoms, and an atomic sentence may be represented using role-pairs consisting of a role-name and a term.

```
atomsent = equation | ( open, term , termseq, close ) | ( open, term, open, 'roleset:' , { open, name, term, close }, close, close ) ;
```

Boolean sentences require implication and *iff* to be binary, but allow *and* and *or* to have any number of arguments, including zero; the sentences (and) and (or) can be used as the truth-values true and false respectively.

```
boolsent = ( open, ('and' | 'or') , { sentence }, close ) | ( open, ('implies' | 'iff') , sentence , sentence, close ) | ( open, 'not' , sentence, close ;
```

Quantifiers may bind any number of variables and may be guarded; and bound variables may be restricted to a named category.**[KIF does not have the guarded quantifier construction.]**

```
quantsent = open, ('forall' | 'exists') , [ name ] , open, { name | ( open, name, term, close )} , close, sentence, close ;
```

A comment may be applied to sentences which are subexpressions of larger sentences.

```
commentsent = open, 'cl:comment', ( quotedstring , sentence ), close ;
```

Like terms, sentences may have enclosing comments. Note that comments may be applied to sentences which are subexpressions of larger sentences.

```
sentence = atomsent | boolsent | quantsent | commentsent
```

Modules are named text segments which represent a text intended to be understood in a 'local' context, where the name indicates the domain of the quantifiers in the text. **[KIF does not have the module construction.]** The module name must not be a numeral or a quoted string. A module may optionally have an exclusion list of names whose denotations are considered to be excluded from the domain. Note that text and module are mutually recursive categories, so that modules may be nested.

```
module =  open, 'cl:module' , (namesequence | enclosedname) ,  [open, 'cl:excludes' , {name} , close ] , cltext, close;
```

CLIF text is a sequence of phrases, each of which is either a sentence, a module, an importation, or a bare comment. Phrases may be wrapped in comments just like sentences. Text may be given a name in the same way as a module; if the text has a name it must occur as the first symbol in the text, without parentheses.

```
phrase = sentence | module | (open,'cl:imports', name , close) | (open, 'cl:comment',
quotedstring, [phrase], close);

cltext = { phrase } ;

clnamedtext  = open, [namesequence |enclosedname], cltext,  close ;
```

A module without an exclusion list is *not* a named text.

The name occurring inside an importation **must** be interpretable as an identifier of a piece of core text, or other Common Logic content which can be rendered as core text without change of meaning. For Web applications, it should be a URI or URI reference identifying a document or document part containing Common Logic text, or a URI or URI reference which is recognized as naming the text by some external conventions known to the application. Particular applications may impose additional conditions on names used as identifiers.

A.3 CLIF Semantics

As described in section A.2, each of the syntactic expressions has specific semantics with respect to Common Logic. This section summarizes each of the CLIF expresssions and their semantics. Refer to Table 1 for the corresponding Common Logic abstract semantics.

CLIF expression from section A.2	**Common Logic abstract semantics (from Table 1)**	
name	E1	name **N**
seqvar	E2	sequence variable **S**
termseq	E3	term sequence $\mathbf{T}_1$ … $\mathbf{T}_n$
	E4	term sequence $\mathbf{T}_1$ … $\mathbf{T}_n$ with sequence variable **S**
equation	E5	term which is an equation containing terms $\mathbf{T}_1$, $\mathbf{T}_2$
boolsent	E8	boolean sentence of type negation and component **C**
	E9	boolean sentence of type conjunction and components $\mathbf{C}_1$ … $\mathbf{C}_n$
	E10	boolean sentence of type disjunction and components $\mathbf{C}_1$ … $\mathbf{C}_n$
	E11	boolean sentence of type implication and components $\mathbf{C}_1$, $\mathbf{C}_2$
quantsent	E13	quantified sentence of type universal and set of names **N** and body **B**
	E14	quantified sentence of type existential and set of names **N** and body **B**
module	E18	module with name **N**, exclusion set **L** and body text **B**
phrase	E16	phrase which is a sentence **S**
	E17	phrase which is an importation containing name **N**

A.4 CLIF Conformance

CLIF is a syntactically fully conformant dialect that is exactly semantically conformant to Common Logic.

Annex B
(normative)

Conceptual Graph Interchange Format (CGIF)

B.1 Introduction

A conceptual graph (CG) is a representation for logic as a bipartite graph with two kinds of nodes, called *concepts* and *conceptual relations*. For computational purposes, the CG structure enables graph-based algorithms to be applied directly to the logical notation. For human factors, the graphs can be displayed in a two or three-dimensional form that is more readable than most linear notations for logic. But for data storage and transmission, the graphs must be serialized in a linear form. The *Conceptual Graph Interchange Format* (CGIF) is a serialized representation for CGs that serves as a concrete notation for the full semantics of Common Logic; it is also the first step toward an XML representation called XCG. This annex specifies the CGIF syntax and its mapping to the CL semantics. CGIF is a normative notation for CL, but the graphical representation, called the *CG display form*, is a nonnormative notation, which appears in this document only in examples. To illustrate the graphical notation and its mapping to CGIF and CLIF, Figure B.1 shows the display form of a conceptual graph that represents the sentence *"John is going to Boston by bus."*

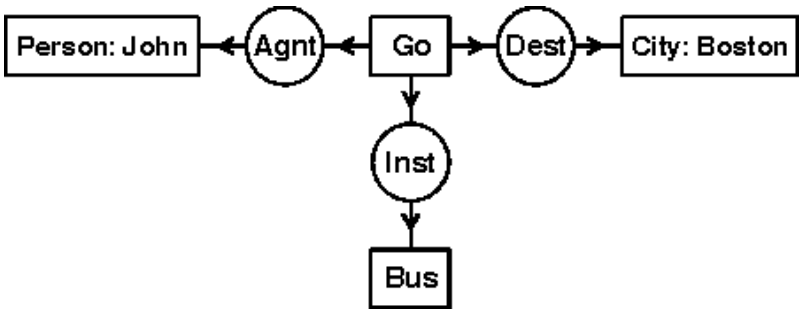

Figure B.1 – CG display form for "John is going to Boston by bus."

In the display form, rectangles represent *concepts*, and circles or ovals represent *conceptual relations*. An arc with an arrowhead pointing toward a circle marks the first argument of the conceptual relation, and an arc pointing away from a circle marks the last argument. If a conceptual relation has only one argument, the arrowhead is omitted. If a conceptual relation has *n* arguments for *n*>2, the arrowheads are replaced by integers 1,...,*n*.

The CG in Figure B.1 has four concepts, each with a *type label* that represents the type of entity to which the concept refers: **`Person, Go, Boston, Bus`**. Two of the concepts have *names*, which identify the referent: **`John, Boston`**. Each of the three conceptual relations has a type label that represents the type of relation: **`Agnt, Inst, Dest`**. The CG as a whole indicates that the person John is the agent **`(Agnt)`** of some instance of going, the city Boston is the destination **`(Dest)`**, and the bus is the instrument **`(Inst)`**. Following is a CGIF representation of Figure B.1:

```
[Go *x] [Person: John *y] [City: Boston *z] [Bus *w]
(Agnt ?x ?y) (Dest ?x ?z) (Inst ?x ?w)
```

In CGIF, the concepts are represented by square brackets, and the conceptual relations are represented by parentheses. An identifier prefixed with an asterisk, such as **`*x`**, marks a *defining node*, which may be referenced by the same identifier prefixed with a question mark, such as **`?x`**. These identifiers, which are called *coreference labels* in conceptual graphs, are mapped to variables with the identifier in CLIF. Following is the equivalent CLIF representation of Figure B.1:

```
(exists ((x Go) (y Person) (z City) (w Bus))
   (and (Agnt x y) (Dest x z) (Inst x w)
        (name y 'John') (name z 'Boston') ))
```

As this example illustrates, the differences between CGIF and CLIF are purely syntactic. The CG display form is a graphical representation that is usually more readable for humans, but it differs from CGIF and CLIF only in syntax. Any semantic information expressed in any one of these notations can be translated to the others without loss or distortion. Formatting information, such as the size or layout of nodes in the display form, has no effect on semantics, but it may be important for esthetics or readability. To preserve the formatting, layout information may be indicated in comments included in the CGIF or CLIF notation.

Although CGIF and CLIF look similar, at least in the previous example, there are two fundamental differences: first, CGIF is a serialized form of a graph; second, the labels such as **`x`** or **`y`** represent connections between nodes in CGIF, but variables in CLIF. Since the nodes of a graph, have no inherent ordering, CGIF also represents an unordered collection, except when explicitly bracketed. Therefore, the list of nodes could be arbitrarily permuted without changing the semantics; the following would be an equivalent representation of Figure B.1:

```
(Inst ?x ?w) [Person: John *y] [Bus *w] (Agnt ?x ?y)
[Go *x] (Dest ?x ?z) [City: Boston *z]
```

Note that the CLIF operator `and` does not occur in CGIF because the conjunction of nodes within any bracketed area is implicit. Since the CGIF labels show connections of nodes, labels may be omitted when they are not needed. One way to omit at least some of the labels is to move concept nodes inside the parentheses of the relation nodes:

```
[Go *x]
   (Agnt ?x [Person: John])
   (Dest ?x [City: Boston])
   (Inst ?x [Bus ])
```

When written in this way, CGIF looks like a frame notation. It is, however, much more general than frames, since it can represent the full semantics of CL.

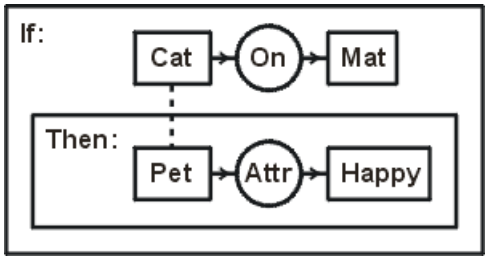

Figure B.2 – CG display form for "If a cat is on a mat, then it is a happy pet."

As another example that illustrates the similarities and differences with CLIF, Figure B.2 displays a CG that represents *"If a cat is on a mat, then it is a happy pet."* The relation **`(Attr ?x ?z)`** indicates that the cat, also called a pet, has an attribute, which is an instance of happiness. As in Figure B.1, the rectangles represent concept nodes, but the two large rectangles contain nested conceptual graphs. Any concept that contains a nested CG is called a *context*; in this example, the type labels **`If`** and **`Then`** indicate that the proposition stated by the CG in the if-context implies the proposition stated by then-context. The dotted line connecting the concepts **`[Cat ]`** and **`[Pet]`** is called a *coreference link*. It indicates that both concepts refer to the same entity. Following is the equivalent CGIF form:

```
[If: [Cat *x] [Mat *y] (On ?x ?y)
   [Then: [Pet ?x] [Happy ?z] (Attr ?x ?z) ]]
```

This example shows that the same coreference labels used to link concept nodes to relation nodes may also be used to link concept nodes to other concept nodes. Any concept that contains a node marked by a question mark, such as **`[Pet ?x]`** is called a *bound node*, which must be linked to a defining node in the same context or some containing context.

The type labels **`If`** and **`Then`** were chosen for readability, but they are not primitive logical operators in CGIF or CLIF. Before they can be mapped to CLIF, they must be expanded to their definitions; both of them actually have the same logical definition: a negation symbol **~** in front of the enclosing bracket:

```
~[ [Cat *x] [Mat *y] (On ?x ?y)
   ~[ [Pet ?x] [Happy ?z] (Attr ?x ?z) ]]
```

This so-called expansion is actually shorter than the original, but it is not as readable. However, it maps directly to CLIF:

```
(not (exists ((x Cat) (y Mat)) (and (On x y)
   (not (and (Pet x) (exists ((z Happy)) (Attr x z)))) )))
```

The CLIF representation for Figure B.2 is not one that most logicians would write directly in CLIF or in many other notations for logic. One difference is the conceptual relation **`(Attr ?x ?z)`**, which indicates that the cat, also called a pet, has an attribute, which is an instance of happiness. This reified treatment of attributes and actions, which is often used in linguistics, is avoided by many philosophers who think of physical objects as ontologically fundamental. Another difference is in the nesting of the then-clause inside the if-clause in Figure B.2. That choice is also designed to simplify the translations from natural languages to logic, and it is isomorphic to structures used in *discourse representation theory* [ref Kamp]. Although these choices are commonly used for conceptual graphs that represent English sentences, CGIF can also represent any choice of ontology or logical operators used in CLIF, as in the following:

```
(forall ((x Cat) (y Mat))
   (if (On x y) (and (Pet x) (Happy x))) )
```

This CLIF statement may be read *For every cat x and mat y, if x is on y then x is a happy pet.* It can be mapped to the following statement in CGIF:

```
[Cat @every*x] [Mat @every*y]
[If: (On ?x ?y) [Then: (Pet ?x) (Happy ?x)]]
```

The flexibility of the CL semantics, which allows quantifiers to range over relations and types, can support the representation of many different ontologies and the rules for translating from one ontology to another.

B.2 CG Terms and Definitions

The following terms and definitions refer specifically to syntactic and semantic features of conceptual graphs and CGIF. Some terms may be shortened to acronyms or abbreviations, which are listed in parentheses after the term. The term "conceptual relation", for example, may be abbreviated "relation".

B.2.1 actor

A conceptual relation in which functional dependencies are explicitly indicated. Formally, the arcs of any actor are explicitly marked as the disjoint union of a set of *input arcs* and *output arcs*, and every output arc is functionally dependent on some subset of the input arcs. An actor that has exactly one output arc is called a *function*.

B.2.2 arc

The connection between a conceptual relation *r* and a concept *c*. In CGIF, the arcs of the relation *r* are represented in a list; each member of the list is either the corresponding concept *c* or a bound label whose defining label is contained in the referent field of *c*.

B.2.3
blank graph

An empty CG containing no concepts or conceptual relations.

B.2.4
bound label

An identifier prefixed with the character "?". The identifier must be identical to the identifier of some defining label in the same context as the bound label or in a context that contains the context of the bound label.

B.2.5
concept

A node that refers to an entity, a sequence of entities, or a range of entities.

B.2.6
conceptual graph
graph

An abstract representation for logic with nodes called concepts and conceptual relations, which are connected by arcs and coreference links.

B.2.7
conceptual graph interchange form (CGIF)

A normative concrete representation for conceptual graphs.

B.2.8
conceptual relation
relation

A node in a CG that has zero or more arcs, each of which links the conceptual relation to some concept.

B.2.9
context

A concept that contains a nonblank CG that is used to describe the referent of the concept.

B.2.10
coreference label

A defining label or a bound label used to identify the concepts that belong to a particular coreference set.

B.2.11
coreference link

Any pair of concepts (c_1,c_2) such that c_1 has a defining label, c_2 has a bound label with the same identifier as the defining label of c_1, and c_1 is either in the same context as c_2 or in a context that contains the context of c_1.

B.2.12
coreference set

A set of concepts in a CG that refer to the same entity or entities. Exactly one member *c* of the coreference set has a defining label, and for every other member *d* of the coreference set, the pair (*c*,*d*) is a coreference link.

B.2.13
defining label

An identifier prefixed with the character "*" that marks the defining concept of a coreference set.

B.2.14
designator

A symbol in the referent field of a concept that determines the referent of the concept by showing its literal form, specifying its location, or describing it by a nested CG.

B.2.15
display form (DF)

A non-normative concrete representation for conceptual graphs that uses graphical displays for better human readability than CGIF.

B.2.16
entity

Anything that exists, has existed, or may exist.

B.2.17
formal parameter
parameter

A concept in a lambda expression whose referent is not defined until the concept is linked to some coreference set whose referent is defined.

B.2.18
functional dependency

A property of some type *t* of conceptual relation. Formally, an arc *a* of a conceptual relation *r* of type *t* is said to be *functionally dependent* on some subset of one or more arcs $S=\{a_1,...,a_n\}$ of *r* where *a* is not in *S*, provided that the following condition is true:

For any possible sequence of entities $e_1,...,e_n$ that may occur as referents of concepts linked to the arcs $a_1,...,a_n$ respectively, there exists one and only one entity e that may occur as the referent of the concept linked to arc a.

If this condition is true, the relation *r* is said to have a functional dependency of arc *a* on the arcs in *S*.

B.2.19
identifier

A character string beginning with a letter or the underscore character "_" and continuing with zero or more letters, digits, or underscores. Case is significant: two identifiers that differ only in the case of one or more letters are considered distinct.

B.2.20
negation

A context of type Proposition to which is attached a monadic conceptual relation of type Neg or its abbreviation by the character "~" or "¬".

B.2.21
nested conceptual graph

A CG that is used as a designator in the referent field of some concept.

B.2.22
outermost context

A collection of conceptual graphs that is not nested in any context.

B.2.23
quantifier

A symbol in the referent field of a concept that determines the range of entities in the referent of the concept. The default quantifier is the existential, which is represented by a blank; all others are defined quantifiers.

B.2.24
referent

The entity or entities to which a concept refers.

B.2.25
referent field

The area in a concept where the referent is specified.

B.2.26
simple graph

A conceptual graph that contains no contexts, negations, or defined quantifiers.

B.2.27
singleton graph

A CG that contains a single concept and no conceptual relations.

B.2.28
star graph

A CG that contains a single conceptual relation and the concepts that are attached to each of its arcs.

B.2.29
type field

The area in a concept node where the concept type is specified.

B.2.30
type label

An identifier that specifies a type of concept, conceptual relation, or actor.

B.2.31
valence

A nonnegative integer that specifies the number of arcs that link a conceptual relation to concepts. All conceptual relations with the same type label have the same valence. A conceptual relation or relation label of valence *n* is said to be *n-adic*; *monadic* is synonymous with 1-adic, *dyadic* with 2-adic, and *triadic* with 3-adic.

B.3 CGIF Syntax

B.3.1 Lexical Categories

The lexical categories of CGIF are defined as sequences of characters as specified in ISO/IEC 10646-1. Characters outside the 7-bit subset of ISO/IEC 10646-1 occur only in strings that represent identifiers, names, literals, and comments. By using escape sequences in such strings, it is possible to encode and transmit CGIF in the 7-bit subset of ISO/IEC 10646-1. Numerical values are represented by the conventions of ANSI X3.42. The following lexical categories may be used in EBNF rules in this chapter or in the translation rules in chapters 8 and 9:

The CGIF lexical categories can be recognized by a finite-state tokenizer or preprocessor. No characters of white space (blanks or other nonprinting characters) are permitted inside any lexical item other than delimited strings (names, comments, or quoted strings). Zero or more characters of white space may be inserted or deleted between any lexical categories without causing an ambiguity or changing the syntactic structure of CGIF. The only white space that should not be deleted is inside delimited strings.

B.3.1.1 Comment.

A comment is a delimited string with a semicolon ";" as the delimiter.

```
Comment  ::=  DelimitedStr(";")
```

B.3.1.2 DelimitedStr(D).

A delimited string is a sequence of two or more characters that begin and end with a single character D called the delimiter. Any occurrence of D other than the first or last character must be doubled.

```
DelimitedStr(D)  ::=  D (AnyCharacterExcept(D) | D D)* D
```

B.3.1.3 Exponent.

An exponent is the letter E in upper or lower case, an optional sign ("+" or "-"), and an unsigned integer.

```
Exponent ::= ("e" | "E") ("+" | "-")? UnsignedInt
```

B.3.1.4 Floating.

A floating-point number is a sign ("+" or "-") followed by one of three options: (1) a decimal point ".", an unsigned integer, and an optional exponent; (2) an unsigned integer, a decimal point ".", an optional unsigned integer, and an optional exponent; or (3) an unsigned integer and an exponent.

```
Floating ::= ("+" | "-") ("." UnsignedInt Exponent?
                          | UnsignedInt ("." UnsignedInt? Exponent?
                                        | Exponent ) )
```

B.3.1.5 Identifier.

An identifier is a string beginning with a letter or underscore "_" and continuing with zero or more letters, digits, or underscores.

```
Identifier  ::=  (Letter | "_") (Letter | Digit | "_")*
```

Identifiers beginning with "_" followed by one or more digits are generated by the Gensym rule defined in Section 8.2. Such identifiers should be avoided unless they have been generated by a call to the Gensym rule.

B.3.1.6 Integer.

An integer is a sign ("+" or "-") followed by an unsigned integer.

```
Integer ::= ("+" | "-") UnsignedInt
```

B.3.1.7 Name.

A name is a delimited string with a single quote "'" as the delimiter.

```
Name  ::=  DelimitedStr("'")
```

B.3.1.8 Number.

A number is an integer or a floating-point number.

```
Number ::= Floating | Integer
```

B.3.1.9 QuotedStr.

A quoted string is a delimited string with a double quote '"' as the delimiter.

```
QuotedStr  ::=  DelimitedStr('"')
```

B.3.1.10 UnsignedInt.

An unsigned integer is a string of one or more digits.

```
UnsignedInt ::= Digit+
```

B.3.2 Syntactic Categories

Every syntactic category of CGIF is first described in English and then defined formally by an EBNF rule, as defined in Annex B. Each EBNF rule is followed by an English statement of possible constraints and implications. Context-sensitive constraints, which are determined by the abstract syntax defined in Chapter 6 or by the extended syntax defined in Chapter 8, are stated with a reference to the governing section in Chapter 6 or 8. In all questions of interpretation, the EBNF rules take precedence over the English descriptions and statements. The following syntactic categories may be used in EBNF rules and translation rules:

The CGIF syntactic categories are defined by a context-free grammar that can be processed by a recursive-descent parser. Zero or more characters of white space (blanks or other nonprinting characters) are permitted between any two successive constituents of any grammar rule that defines a syntactic category.

B.3.2.1 Actor.

An actor begins with "<" followed by a type. It continues with zero or more input arcs, a separator "|", zero or more output arcs, and an optional comment. It ends with ">".

```
Actor  ::=  "<" Type(N) Arc* "|" Arc* Comment? ">"
```

The arcs that precede the vertical bar are called *input arcs*, and the arcs that follow the vertical bar are called *output arcs*. The valence N of the actor type must be equal to the sum of the number of input arcs and the number of output arcs.

B.3.2.2 Arc.

An arc is a concept or a bound label.

```
Arc  ::=  Concept | BoundLabel
```

B.3.2.3 BoundLabel.

A bound label is a question mark "?" followed by an identifier.

```
BoundLabel  ::=  "?" Identifier
```

B.3.2.4 CG.

A conceptual graph is a list of zero or more concepts, conceptual relations, actors, special contexts, or comments.

```
CG  ::=  (Concept | Relation | Actor | SpecialContext | Comment)*
```

The alternatives may occur in any order provided that any bound coreference label must occur later in the CGIF stream and must be within the scope of the defining label that has an identical identifier. The definition permits an empty CG, which contains nothing. An empty CG, which says nothing, is always true.

B.3.2.5 CGStream.

A CG stream is a list of zero or more conceptual graphs, each followed by a period ".".

```
CGStream  ::=  (CG ".")*
```

Since a CG may be empty, a string of periods, such as "....", is a valid CG stream.

B.3.2.6 Concept.

A concept begins with a left bracket "[" and an optional monadic type followed by optional coreference links and an optional referent in either order. It ends with an optional comment and a required "]".

```
Concept  ::=  "[" Type(1)? {CorefLinks?, Referent?} Comment? "]"
```

If the type is omitted, the default type is Entity. This rule permits the coreference labels to come before or after the referent. If the referent is a CG that contains bound labels that match a defining label on the current concept, the defining label must precede the referent.

For example, the concept [Person: Mary] could be written in CGIF as `[Person:'Mary'*x]`; the coreferent concept `[T]` could be written `[?x]`, and its implict type would be Entity.

B.3.2.7 Conjuncts.

A conjunction list consists of one or more type terms separated by "&".

```
Conjuncts(N)  ::=  TypeTerm(N) ("&" TypeTerm(N))*
```

The conjunction list must have the same valence N as every type term.

B.3.2.8 CorefLinks.

Coreference links are either a single defining coreference label or a sequence of zero or more bound labels.

```
CorefLinks  ::=  DefLabel | BoundLabel*
```

If a dominant concept node, as specified in Section 6.9, has any coreference label, it must be either a defining label or a single bound label that has the same identifier as the defining label of some co-nested concept.

B.3.2.9 DefLabel.

A defining label is an asterisk "*" followed by an identifier.

```
DefLabel  ::=  "*" Identifier
```

The concept in which a defining label appears is called the *defining concept* for that label; a defining concept may contain at most one defining label and no bound coreference labels. Any defining concept must be a dominant concept as defined in Section 6.9.

Every bound label must be resolvable to a unique defining coreference label within the same context or some containing context. When conceptual graphs are imported from one context into another, however, three kinds of conflicts may arise:

1. A defining concept is being imported into a context that is within the scope of another defining concept with the same identifier.

2. A defining concept is being imported into a context that contains some nested context that has a defining concept with the same identifier.

3. Somewhere in the same module there exists a defining concept whose identifier is the same as the identifier of the defining concept that is being imported, but neither concept is within the scope of the other.

In cases (1) and (2), any possible conflict can be detected by scanning no further than the right bracket "]" that encloses the context into which the graph is being imported. Therefore, in those two cases, the newly imported

defining coreference label and all its bound labels must be replaced with an identifier that is guaranteed to be distinct. In case (3), there is no conflict that could affect the semantics of the conceptual graphs or any correctly designed CG tool; but since a human reader might be confused by the similar labels, a CG tool may replace the identifier of one of the defining coreference labels and all its bound labels.

B.3.2.10 Descriptor.

A descriptor is a structure or a nonempty CG.

```
Descriptor  ::=  Structure | CG
```

A context-free rule, such as this, cannot express the condition that a CG is only called a descriptor when it is nested inside some concept.

B.3.2.11 Designator.

A designator is a literal, a locator, or a quantifier.

```
Designator  ::=  Literal | Locator | Quantifier
```

B.3.2.12 Disjuncts.

A disjunction list consists of one or more conjunction lists separated by "|".

```
Disjuncts(N)  ::=  Conjuncts(N) ("|" Conjuncts(N))*
```

The disjunction list must have the same valence N as every conjunction list.

B.3.2.13 FormalParameter.

A formal parameter is a monadic type followed by a optional defining label.

```
FormalParameter  ::=  Type(1) [DefLabel]
```

The defining label is required if the body of the lambda expression contains any matching bound labels.

B.3.2.14 Indexical.

An indexical is the character "#" followed by an optional identifier.

```
Indexical  ::=  "#" Identifier?
```

The identifier specifies some implementation-dependent method that may be used to replace the indexical with a bound label.

B.3.2.15 IndividualMarker.

An individual marker is the character "#" followed by an integer.

```
IndividualMarker  ::=  "#" UnsignedInt
```

The integer specifies an index to some entry in a catalog of individuals.

B.3.2.16 Literal.

A literal is a number or a quoted string.

```
Literal  ::=  Number | QuotedStr
```

B.3.2.17 Locator.

A locator is a name, an individual marker, or an indexical.

```
Locator  ::=  Name | IndividualMarker | Indexical
```

B.3.2.18 Negation.

A negation begins with a tilde "~" and a left bracket "[" followed by a conceptual graph and a right bracket "]".

```
Negation  ::=  "~[" CG "]"
```

A negation is an abbreviation for a concept of type Proposition with an attached relation of type Neg. It has a simpler syntax, which does not permit coreference labels or attached conceptual relations. If such options are required, the negation can be expressed by the unabbreviated form with an explicit Neg relation.

B.3.2.19 Quantifier.

A quantifier consists of an at sign "@" followed by an unsigned integer or an identifier and an optional list of zero or more arcs enclosed in braces.

```
Quantifier  ::=  "@" (UnsignedInt | Identifier ("{" Arc* "}")?)
```

The symbol **`@some`** is called the *existential quantifier*, and the symbol **`@every`** is called the *universal quantifier*. If the quantifier is omitted, the default is **`@some`**.

B.3.2.20 Referent.

A referent consists of a colon ":" followed by an optional designator and an optional descriptor in either order.

```
Referent  ::=  ":" {Designator?, Descriptor?}
```

B.3.2.21 Relation.

A conceptual relation begins with a left parenthesis "(" followed by an N-adic type, N arcs, and an optional comment. It ends with a right parenthesis ")".

```
Relation  ::=  "(" Type(N) Arc* Comment? ")"
```

The valence N of the relation type must be equal to the number of arcs.

B.3.2.22 SpecialConLabel.

A special context label is one of five identifiers: "if", "then", "either", "or", and "sc", in either upper or lower case.

```
SpecialConLabel  ::=  "if" | "then" | "either" | "or" | "sc"
```

The five special context labels and the two identifiers "else" and "lambda" are reserved words that may not be used as type labels.

B.3.2.23 SpecialContext.

A special context is either a negation or a left bracket, a special context label, an optional colon, a CG, and a right bracket.

```
SpecialContext  ::=  Negation | "[" SpecialConLabel ":"? CG "]"
```

B.3.2.24 Type.

A type is a type expression or an identifier other than the reserved labels: "if", "then", "either", "or", "sc", "else", "lambda".

```
Type(N)  ::=  TypeLabel(N) | TypeExpression(N)
```

A concept type must have valence N=1. A relation type must have valence N equal to the number of arcs of any relation or actor of that type. The type label or the type expression must have the same valence as the type.

B.3.2.25 TypeExpression.

A type expression is either a lambda expression or a disjunction list enclosed in parentheses.

```
TypeExpression(N)  ::=  LambdaExpression(N) | "(" Disjuncts(N) ")"
```

The type expression must have the same valence N as the lambda expression or the disjunction list.

B.3.2.26 TypeLabel(N).

A type label is an identifier.

```
TypeLabel(N)  ::=  Identifier
```

The type label must have an associated valence N.

B.4 CGIF Semantics

To be completed. *This section will contain the mappings from CGIF expressions to the abstract syntax of Common Logic.*

Annex C
(normative)

eXtended Common Logic Markup Language (XCL)

C.1 XCL and dialects

XCL is an XML notation for Common Logic designed to serve three inter-related roles. It is the intended interchange language for communicating Common Logic across a network; it is the primary abstract syntax for Common Logic into which all other Common Logic dialects can be translated; and it is a generic notation for incorporating Common Logic text written in any Common Logic dialect.

C.2 XCL Syntax

The last of these is the simplest to deal with, so we consider it first. Any piece of Common Logic text written in any named dialect, treated as XML content and enclosed in an **<XCL:text> </XCL:text>** element, with the property value of the property **Common Logic-dialect** identifying the dialect, is a legal XCL element, called a *dialect element*. Other attributes may be attached to an **XCL:text** element, but the **Common Logic-dialect** value is required. Dialect elements allow XCL to transmit Common Logic text between applications tuned to a particular dialect, with minimal changes to the text.

The name of the core-syntax dialect is **coreSyntax**. Thus, the following is a legal XCL element:

```
<XCL:text cl-dialect="coreSyntax"> (forall (x) (implies (Boy x) (exists (y) (and (Girl y)(Kissed x y) )) )) </XCL:text>
```

> Note that this convention assumes that XML escaping is applied to the enclosed Common Logic text. For example,
>
> ```
> <XCL:text cl-dialect="cl:coreSyntax">
>
> (cl:String 'this is <forbidden> in XML & is considered "naughty"')
>
> </XCL:text>
> ```
>
> should be understood as indicating the CLIF syntax expression:
>
> ```
> (cl:String 'this is <forbidden> in XML & is considered "naughty"')
> ```

The use of **'** to escape a single-quote mark is deprecated in the body of a core syntax dialect element, due to potential confusion with the CLIF string-quoting conventions. If the syntactic specification of a dialect prohibits the use of certain characters or character sequences, the use of XML escaping to encode these characters or character sequences is illegal inside an **XCL:text** element with that dialect property. Such uses may be reported as syntactic errors by a conforming XCL application. There is no XCL requirement that the enclosed Common Logic dialect text be well-formed dialect Common Logic, although applications may report ill-formed dialect Common Logic as a syntactic error.

C.3 XCL Semantics

Common Logic can be rendered into XCL directly, in a form which displays the Common Logic syntax directly in the XML markup. This is analogous to "content markup" in MathML [11]. The elements required for the CLIF *will be* described here, linked to the corresponding clauses of the EBNF Common Logic Interchange Format in Annex A.

C.4 XCL Conformance

This section will be provided later.

Bibliography

1. **Sowa, J.F., *Conceptual Structures: Information Processing in Mind and Machine*. 1984, Reading, Mass.: Addison-Wesley. 481.**
2. **Genesereth, M.R. and R.E. Fikes, *Knowledge Interchange Format Version 3.0 Reference Manual*. 1992, Computer Science Department, Stanford University.**
3. **Berners-Lee, T., et al., *Uniform Resource Identifiers*. 1998, IETF. http://www.ietf.org/rfc/rfc2396.txt.**
4. **Brickley, D. and R.V. Guha, *RDF Vocabulary Description Language 1.0: RDF Schema*, in *W3C Recommendation*. 2004. http://www.w3.org/TR/2004/REC-rdf-schema-20040210.**
5. **Chen, W., et al., *HILOG: A Foundation for Higher-Order Logic Programming*. Journal of Logic Programming, 1993. 15(3): p. 187-230.**
6. **Rumbaugh, J., I. Jacobsen, and G. Booch, *Unified Modeling Language Reference Manual*. 1998, Reading, MA U.S.A.: Addison-Wesley. 480.**
7. **Carrol, J.J., et al., *Named graphs, provenance and trust*, in *14th Intl. Conf. on World Wide Web*. 2005, ACM Press: Chiba, Japan. http://portal.acm.org/citation.cgm?doid=1060745.1060835.**
8. **Fielding, R.T., *Architectural Styles and the Design of Network-Based Software Architectures*, in *Information and Computer Science*. 2000, University of California, Irvine: Irvine, CA, USA. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.**
9. **Berners-Lee, T., J. Hendler, and O. Lassila, *The Semantic Web*, in *Scientific American*. 2001.**
10. **Guha, R.V. and P. Hayes, *LBase: Semantics for Languages of the Semantic Web*. 2003(W3C Working Group Note 10 October 2003).**
11. **Ausbrooks, R., et al., *Mathematical markup Language (MathML) Version 2.0*, in *W3C Recommendation*. 2003, W3C. http://www.w3.org/TR/MathML2.**

Index