INTERNATIONAL STANDARD

ISO/IEC 24707

First edition
2007-10-01

Information technology — Common Logic (CL): a framework for a family of logic-based languages

Technologies de l'information — Logique commune (CL): un cadre pour une famille de langages basés sur la logique

Reference number
ISO/IEC 24707:2007(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Page

Figures

Page

Tables

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

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

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

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 24707 was prepared by Joint Technical Committee ISO/IEC 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 classes or relations while preserving a first-order model theory, and a semantics which allows theories to describe intensional entities such as classes 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 (CL): a framework for a family of logic-based languages

1 Scope

This International Standard specifies a family of logic languages designed for use in the representation and interchange of information and data among disparate computer systems.

The following features are essential to the design of this International 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 first-order logical sentences.
- Interchange of information among heterogeneous computer systems.

The following are within the scope of this International Standard:

- representation of information in ontologies and knowledge bases;
- specification of expressions that are the input or output of inference engines;
- formal interpretations of the symbols in the language.

The following are outside the scope of this International Standard:

- the specification of proof theory or inference rules;
- specification of translators between the notations of heterogeneous computer systems;
- computer-based operational methods of providing relationships between symbols in the logical "universe of discourse" and individuals in the "real world".

This International Standard describes Common Logic's syntax and semantics.

It 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 this International Standard. The purpose is to facilitate interchange of first-order logic-based information between systems.

Issues relating to computability using this International Standard (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 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 14977:1996, *Information technology — Syntactic metalanguage — Extended BNF*

3 Terms and definitions

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

3.1
atom
sentence form which has no subsentences as syntactic components

NOTE Can be either an equation, or an atomic sentence consisting of a predicate applied to an argument sequence.

3.2
axiom
any sentence which is assumed to be true, from which others are derived, or by which they are entailed

NOTE In a computational setting, an axiom is a sentence which is never posed as a goal to be proved, but only used to prove other sentences.

3.3
Common Logic Interchange Format
CLIF
KIF-based syntax that is used for illustration purposes in this International Standard

NOTE It is one of the concrete syntaxes as described in Annex A. The name “KIF” is not used for this syntax in order to distinguish it from the commonly used KIF dialects. No assumptions are made in this International Standard with respect to KIF semantics; in particular, no equivalence between CLIF and KIF is intended.

3.4
conceptual graph
CG
graphical or textual display of symbols arranged according to the style of conceptual graph theory

3.5
Conceptual Graph Interchange Format
CGIF
text version of conceptual graphs whose rules of formation conform to Annex B of this International Standard

NOTE 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.

3.6
conceptual graph theory
form of first-order logic which represents existential quantification and conjunction via the assertion of logical constructs called concepts and relations, which are arranged in an abstract or visually displayed graph

NOTE Conceptual graph theory was introduced by John Sowa [1].

3.7
denotation
relationship holding between a name or expression and the thing to which it refers

NOTE Also used, with "of," to mean the entity being named, i.e. the referent of a name or expression.

3.8 dialect
concrete instance of Common Logic syntax that shares (at least some of) the uniform semantics of Common Logic

NOTE A dialect may be textual or graphical or possibly some other form. A dialect by definition is also a conforming language (see 7.1 for further details).

3.9
discourse name
name whose interpretation is in the universe of discourse

NOTE There is no assumption that different names are interpreted as different individuals. A single individual in the universe of discourse may be denoted by two or more distinct names.

3.10
domain of discourse
See universe of discourse.

3.11
eXtensible Common Logic Markup Language
XCL
XML-based syntax for Common Logic

3.12
individual
one element of the universe of discourse

NOTE The universe of discourse is the set of all individuals.

3.13
Internationalized Resource Identifier
IRI
string of Unicode characters conforming to the syntax described in [2] and intended for use as an Internet network identifier syntax which can accommodate a wide variety of international character forms

NOTE Intended to replace **Uniform Resource Identifier** as an Internet standard for network identifiers.

3.14
interpretation
formal specification of the meanings of the names in a vocabulary of a Common Logic dialect in terms of a universe of reference.

NOTE 1 An interpretation in turn determines the semantic values of all complex expressions of the dialect, in particular the truth values of its sentences.

NOTE 2 See 6.2 for a more precise description of how an interpretation is defined.

3.15
Knowledge Interchange Format
KIF
text-based first order formalism, using a LISP-like list notation

NOTE 1 KIF, introduced by Mike Genesereth [3], originated with the Knowledge Sharing Effort sponsored by the US DARPA.

NOTE 2 KIF forms the basis for one of the three Common Logic dialects included in this International Standard.

3.16
operator
distinguished syntactic role played by a specified component within a functional term

NOTE The denotation of a functional term in an interpretation is determined by the functional extension of the denotation of the operator together with the denotations of the remaining components.

3.17
predicate
⟨Common Logic⟩ distinguished syntactic role played by exactly one component within an atomic sentence

NOTE The truth value of an atomic sentence in an interpretation is determined by the relational extension of the denotation of the predicate together with the denotations of the remaining components.

3.18
segregated dialect
dialect in which some names are non-discourse names

NOTE In an interpretation of a segregated dialect, the denotations of the non-discourse names are in the universe of reference, but not in the universe of discourse.

3.19
sentence
⟨Common Logic⟩ unit of logical text which is true or false, i.e. which is assigned a truth-value in an interpretation

3.20
sort
any subset of the universe of discourse over which some quantifier is allowed to range

NOTE Related to the definition of "type" (see 3.24). Generally used to mean a proper subset of the individuals in the universe of discourse.

3.21
sorted logic
logic system (whether first-order or not) which requires that all nonlogical symbols be assigned to a sort

3.22
term
⟨Common Logic⟩ expression which denotes an individual, consisting of either a name or, recursively, a function term applied to a sequence of arguments, which are themselves terms

3.23
traditional first-order logic
TFOL
traditional mathematical formulations of logic as introduced chiefly by Russell, Whitehead, Peano, Frege, Peirce and Tarski dealing with *n*-ary predication, the Boolean operators (including negation) and quantification, and in which every proposition is either determinately true or determinately false

NOTE Languages for traditional first-order logic specifically exclude predicate quantifiers and the use of the same name in both predicate and argument position in atomic sentences, both of which are permitted (though not required) in Common Logic. Languages for traditional first-order logic fall within the category of segregated dialects in CL (see 6.1.3).

3.24
type
logical framework in which expressions in the logic are classified into syntactic or lexical categories (types) and restricted to apply only to arguments of a fixed type

NOTE 1 In practice, a type represents a class of individuals."Type theory" usually refers to a particular class of such logics in which relation symbols are separated into orders, with relations of order n applying only to those of lower orders.

NOTE 2 A type is more restricted than a sort in that a type imposes intensional or categorical constraints on which individuals are members of the type category, whereas a sort refers only to any subset of individuals in the domain over which some quantifier is presumed to operate.

3.25
universe of discourse
domain of discourse
set of all the individuals in an interpretation, i.e. the set over which the quantifiers range

NOTE Required to be a subset of the universe of reference, and may be identical to it.

3.26
universe of reference
set of all the entities needed to define the meanings of logical expressions in an interpretation

NOTE 1 Required to be a superset of the universe of discourse, and may be identical to it.

NOTE 2 Segregated dialects are commonly described to have a universe of discourse, without mentioning the universe of reference; and for non-segregated dialects the universes of discourse and of reference are identical. The distinction makes it possible to provide a single semantics which can cover both styles of dialect. Non-segregated dialects which treat the universes of discourse and of reference as identical may simply refer to 'the universe' of an interpretation.

3.27
Uniform Resource Identifier
URI
sequence of ASCII characters conforming to the syntax forms defined in [4]

NOTE At the time of writing, the Internet standard syntax for network identifiers. It is likely to be obsoleted by **Internationalized Resource Identifier**.

4 Symbols and abbreviations

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

4.1 Symbols

Some of these symbols represent terms which are defined in clause 3.

fun_I a mapping from UR_I to functions from UD_I* to UD_I

I an interpretation, in the model-theoretic sense

int_I a mapping from names in a vocabulary V to UR_I; informally, a means of associating names in V to referents in UR_I

rel_I a mapping from UR_I to subsets of UD_I*

seq_I a mapping from sequence markers in V to UD_I*

V a vocabulary, which is a set of names and sequence markers

UD_I the universe of discourse; a non-empty set of individuals that an interpretation I is "about" and over which the quantifiers are understood to range

UR_I the universe of reference, i.e. the set of all referents of names in an interpretation I

X* the set of finite sequences of the elements of X, for any set X

4.2 Abbreviations

These abbreviations are used in this International Standard. See clause 3 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)

EBNF Extended Backus-Naur Format, as in ISO/IEC 14977:1996.

FO First-order

IRI Internationalized Resource Identifier

KIF Knowledge Interchange Format

OWL Web Ontology Language

RDF Resource Definition Framework

RDFS Resource Definition Framework Schema

TFOL traditional first order logic

URI Uniform Resource Identifier

XCL eXtensible Common Logic Markup Language

XML eXtensible Markup Language

5 Requirements and design overview

This clause is informative. Its purpose is to briefly describe the purposes of Common Logic and the overall guiding principles and constraints on its content.

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. The use of "should" in the rest of clause 5 indicates a desired goal but is not required of either CL or its conforming dialect (in accordance with Annex H of ISO/IEC Directives – Part 2).

5.1.1 Common Logic should include full first-order logic with equality.

Common Logic syntax and semantics shall provide for the full range of first-order syntactic forms, with their usual meanings. Any conventional first-order syntax will be directly translatable into Common Logic without loss of information or alteration of meaning.

5.1.2 Common Logic should provide 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. There should be at least one compact, human-readable syntax defined which can be used to express the entire language.

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

a. The XML syntax should 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. URIs and URI references should be usable as names in the language.

c. URIs 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.

5.1.4 Common Logic should support open networks

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 entailments, 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 such universes.

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

5.1.5 Common Logic should not make arbitrary assumptions about semantics

a. Common Logic does not make gratuitous or arbitrary assumptions about logical relationships between different expressions.

b. If possible, Common Logic agents should express these assumptions in Common Logic directly.

5.2 A family of notations

This (informative) section describes what is meant by a “family” of languages and gives some of the rationale behind the development of Common Logic.

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 in this International Standard as *dialects*, may differ sharply in their surface syntax, but they have a single uniform semantics and can all be transcribed into the common abstract 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.

A Common Logic dialect called CLIF based on KIF (see Annex A) is used in giving examples throughout this International Standard. CLIF can be considered an updated and simplified form of KIF 3.0 [3], 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 form of first-order logic for machine processing; the CGIF language is specified in Annex B. An XML dialect using CL semantics is specified in Annex C.

6 Common Logic abstract syntax and semantics

This section describes the normative aspects of Common Logic’s syntax and semantics.

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*. Additional terms in the entries may identify sub-categories, or may identify constituent parts of the category. Those terms being defined here are underlined for clarity. Other terms may be found in the definitions of clause 3.

6.1.1.1 A text is a set, list, or bag of phrases. A piece of text shall optionally 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 shall preserve the structure of texts, since other applications are allowed to utilize the structure for other purposes, such as indexing. If a dialect imposes conditions on texts, these conditions shall be preserved by conforming applications. A text may be empty.*

6.1.1.2 A phrase is either a module, a sentence, an importation, or a text 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’ universe of discourse in which the text is understood; the exclusion set indicates any names in the text which are explicitly excluded from this local universe. 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 in clause 6.3.1 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 (i) a type, called a *quantifier*, (ii) a finite, nonrepeating sequence of names and sequence markers called the *binding sequence*, each element of which is called a *binding* of the quantified sentence, and (iii) a sentence called the *body* of the quantified sentence. Every Common Logic dialect shall distinguish the *universal* and the *existential* types of quantified sentence. A name or sequence marker which occurs in the binding sequence is said to be *bound in* the body. Any name or sequence marker which is not bound in the body is said to be *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 shall distinguish five types of Boolean sentences: *conjunctions* and *disjunctions,* which have any number of components, *implications* and *biconditionals*, which have exactly two components, and *negations*, which have exactly one component.

NOTE The current specification does not recognize any particular irregular sentence forms. This category is included in the abstract syntax to accommodate syntactic extensions to Common Logic whose semantics cannot be fully defined within Common Logic. Examples include modalities, non-monotonic connectives and imperative constructions.

6.1.1.9 An atom is either an *equation* containing two *arguments*, which are terms, or is an atomic sentence, which consists of a term, called the *predicate*, and a term sequence called the *argument sequence*, the elements of which are called *arguments* of the atom.

NOTE Dialects which use a name to identify equality may consider it to be a predicate, and treat an equation as an atomic sentence.

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*, the elements of which are called *arguments* of the functional term.

6.1.1.12 A term sequence is a finite sequence of terms or sequence markers.

NOTE Term sequences may be empty, but a functional term with an empty argument sequence shall not be identified with its operator, and an atomic sentence with an empty argument sequence shall not be identified with its predicate.

6.1.1.13 A *vocabulary* is a set of names and sequence markers.

6.1.1.14 Names and sequence markers are disjoint syntax categories, and each is disjoint from all other syntax categories.

This clause completely describes the abstract syntactic structure of Common Logic. Any fully conformant Common Logic dialect **shall** provide an unambiguous syntactic representation for each of the above types of recognized expressions, except for irregular sentences.

Sentence types are commonly indicated by the inclusion of explicit text strings, such as “forall” for universal sentence and “and” for conjunction. However, 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.

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 using UML class diagram notation [5].

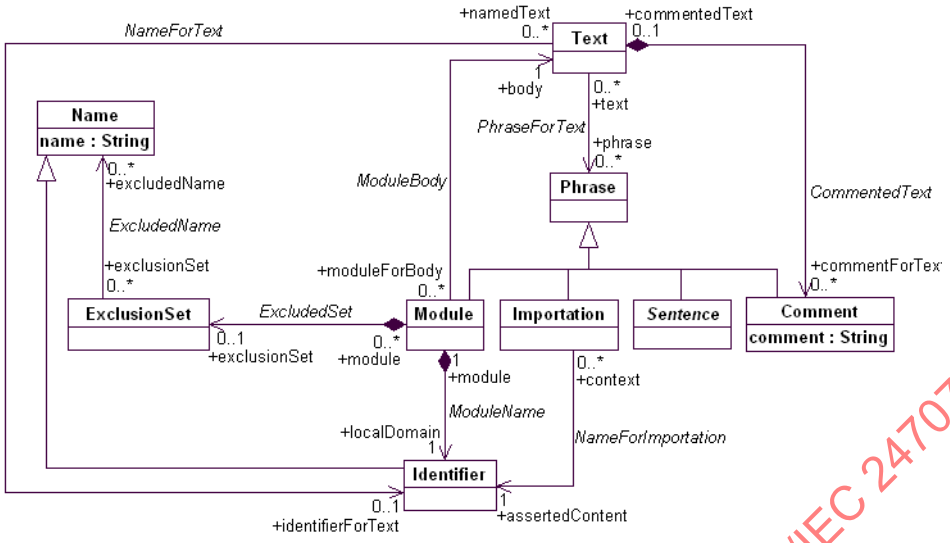

Figure 1 — Structure of a text and the taxonomy of the phrase category *text*

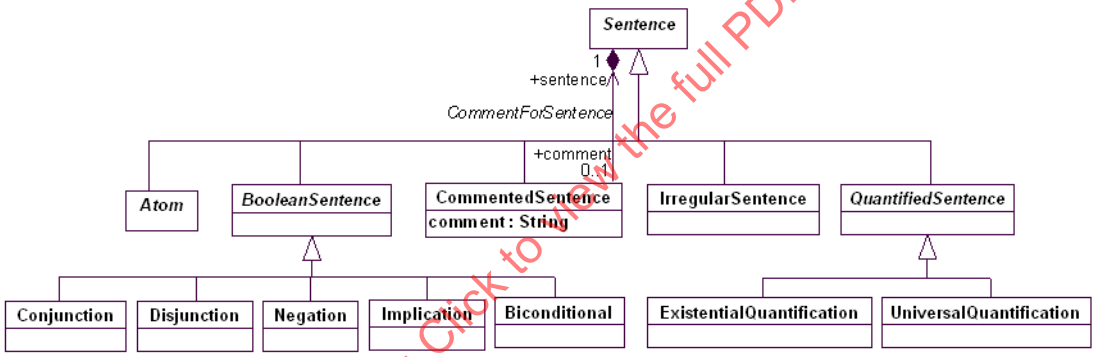

Figure 2 — Abstract syntax of sentence and its sub-categories

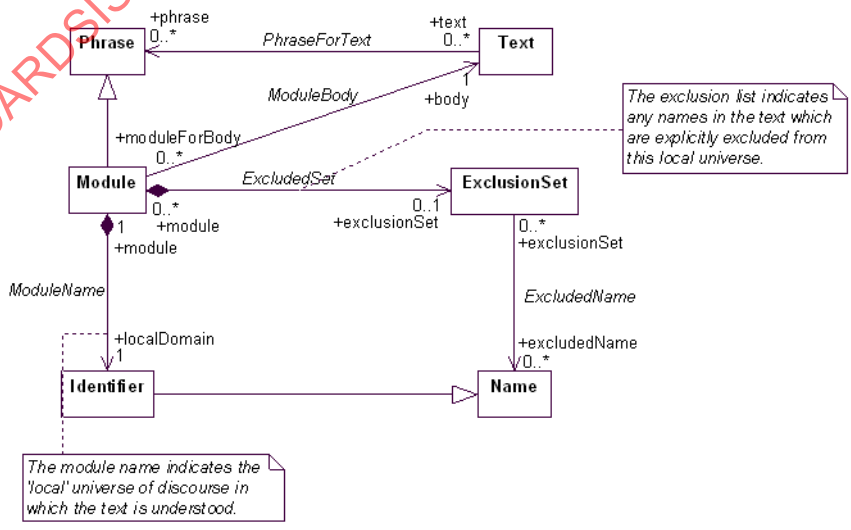

Figure 3 — Abstract syntax of a *module*

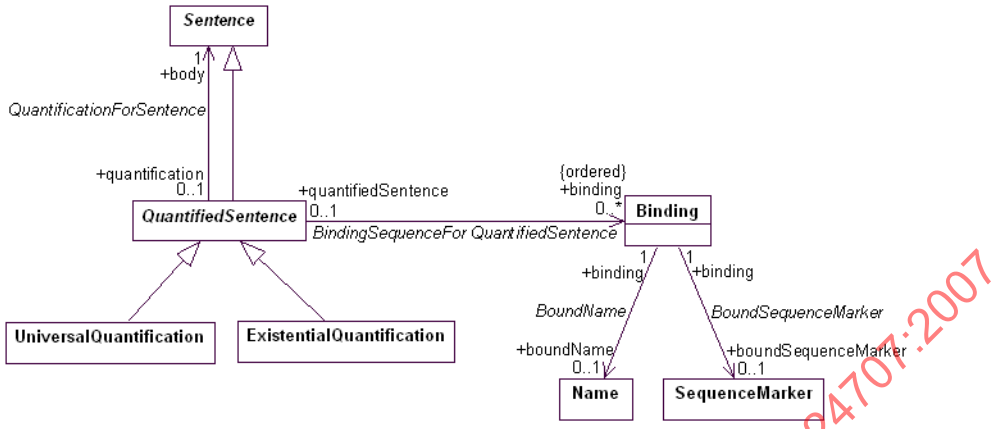

Figure 4 — Abstract syntax of a *quantified sentence*

Figure 4 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*.

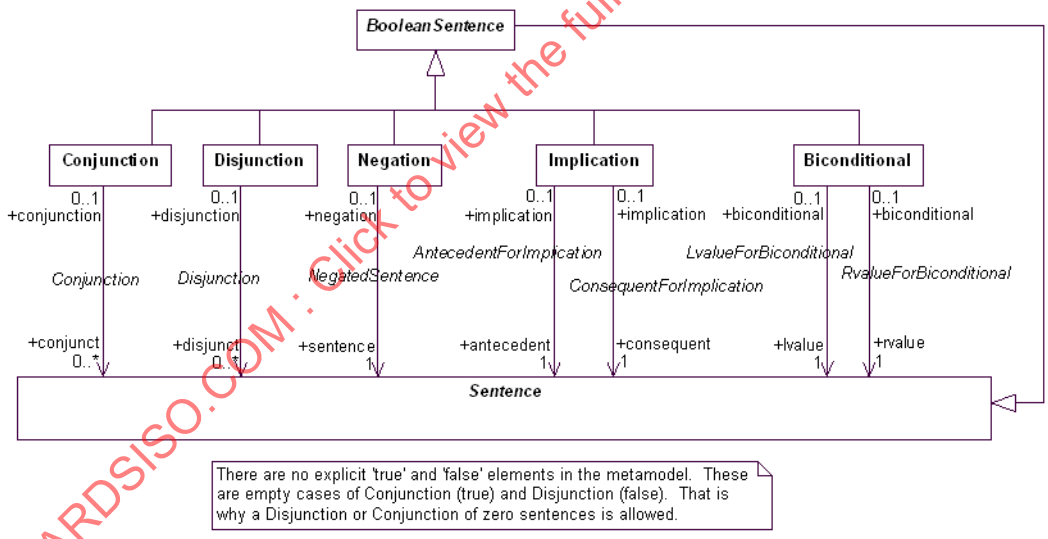

Figure 5 — Abstract syntax of a boolean sentence

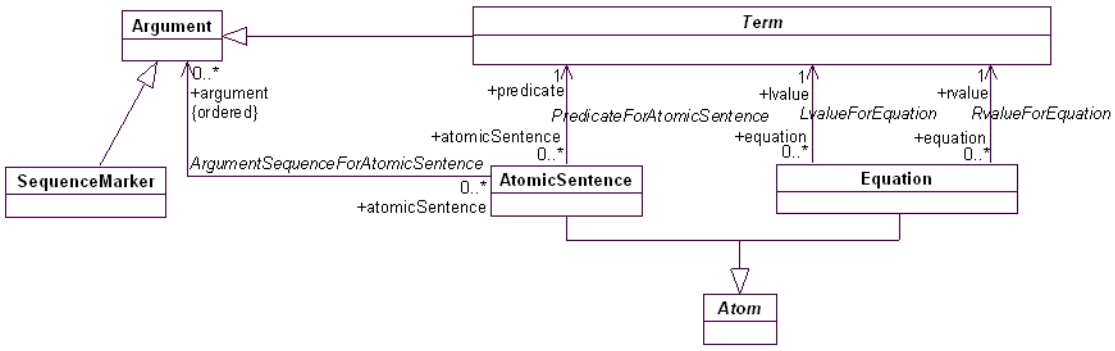

Figure 6 — Abstract syntax of an atom

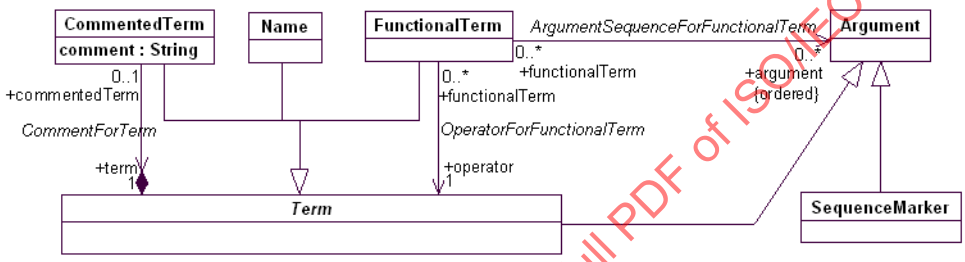

Figure 7 — Abstract syntax of a term and term sequence

6.1.3 Abstract syntactic structure of dialects

A dialect which provides only some types of the Common Logic expressions is said to be a *syntactically partial* Common Logic dialect, or *syntactically partially conformant*. In particular, a dialect that does not include sequence markers, but is otherwise fully conformant, is known as a *syntactically compact* dialect. See clause 7.1 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 shall 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 Annex A, contains a number of syntactic sugared forms for quantified and atomic sentences. (Other types of compliance are also recognized: see clause 7 for a full account of conformance.)

The only undefined terms in the abstract syntax clause are *name* and *sequence marker*. The only required syntactic constraint on the basic lexical categories of *name* and *sequence marker* are that they **shall be** exclusive. Dialects intended for transmission of content on a network **should not** impose arbitrary or unnecessary restrictions on the form of names, and **shall** provide for certain names to be used as identifiers of Common Logic texts; that is, character strings used as identifiers in a dialect shall be parseable as Common Logic names in that dialect. Dialects intended for use on the Web **should** allow Universal Resource Identifiers, International Resource Identifiers and URI references to be used as names [2] [4]. Common Logic dialects **should** define names in terms of Unicode (ISO/IEC 10646:2003) conventions.

There is no notion of 'bound variable' in the CL abstract syntax. Names that can occur bound are not required to be lexically distinguished from those that can (only) occur free, nor are names required to be partitioned into distinct classes such as relation, function or individual names. There are no 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 names that can occur bound (i.e., the variables of traditional first-order languages) be written with a special 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 traditional 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-discourse names*, which are understood not to denote entities in the universe of discourse. This requirement may be imposed, for example, by partitioning the vocabulary or by requiring names that occur in certain syntactic positions to be non-discourse. A dialect with non-discourse names is called *segregated*. Names which are not non-discourse names are called *discourse names*.

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

- Every name shall be classified as either discourse or as non-discourse.
- No name shall be classified as both discourse and non-discourse.
- No non-discourse name shall be an argument of an atom or functional term.
- No non-discourse name shall be bound in a quantified sentence.

As the presence of non-discourse names affects the semantics, special conditions apply to segregated dialects.

A dialect which is not segregated is called *non-segregated*. All names in a non-segregated dialect are discourse names.

6.2 Common Logic semantics

The semantics of Common Logic is defined in terms of a satisfaction relation between Common Logic text and mathematical structures called *interpretations*.

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

An *interpretation* I of a vocabulary V is a set UR_I , the *universe of reference*, with a distinguished nonempty subset UD_I, the *universe of discourse*, and four mappings:

- rel_I from UR_I to subsets of UD_I^* = {$<x_1,...,x_n>$ | $x_1,\ldots,x_n \in UD_I$} (i.e., the set of finite sequences of elements of UD_I). Note that the empty sequence is in UD_I^*, for any UD_I;
- fun_I from UR_I to total functions from UD_I^* into UD_I, that is, to functions that map each sequence in UD_I^* to a (unique) element of UD_I;
- int_I from names in V to UR_I, such that int_I(v) is in UD_I if and only if v is a discourse name;

  NOTE 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 truth values { true, false }.

- seq_I from sequence markers in V to UD_I^*.

Intuitively, UD_I is the universe or domain of discourse containing all the individual things the interpretation is 'about' and over which the quantifiers range. UR_I is a potentially larger set of things that might also contain entities which are not in the universe of discourse. In particular, UR_I might contain relations not in UD_I to serve as the interpretations of the non-discourse names in a segregated dialect. All names are interpreted in the same way, whether or not they are understood to denote something in the universe of discourse; that is why there is only a single interpretation mapping that applies to all names, regardless of their syntactic role. In particular, rel_I(x) is in UD_I* even when x is not in UD_I. When considering only segregated dialects, the elements of the universe of reference which are outside the universe of discourse may be identified with their corresponding values of the rel_I and fun_I mappings, which are then re-interpreted to be the identity mapping. The resulting construction maps predicates directly to relations and operators to functions, yielding a more traditional interpretation structure for the segregated syntax of traditional first-order logic. On the other hand, when considering only non-segregated dialects, the distinction between universes of reference and discourse is unnecessary, since they may be considered to be identical. The distinction is made here in order to give a uniform treatment of both segregated and non-segregated dialects.

Irregular sentences are treated as though they were arbitrary propositional variables. Note this does not affect the CL interpretations of any CL sentences which occur as syntactic components of an irregular sentence. Note also that, although sequence markers are mapped into finite sequences in an interpretation, these sequences are not denoted by names, and so are not required to be in the universe of reference.

The assignment of semantic values to complex expressions – notably, the assignment of truth values to sentences – requires some auxiliary definitions.

Let S be a subset of V. An interpretation J of V is an *S-variant of* I if it is exactly like I except that int_J and seq_J might differ with int_I and seq_I on what they assign to the members of S. More formally, J is an S-variant of I if $UR_J = UR_I$, $UD_J = UD_I$, $rel_J = rel_I$, $fun_J = fun_I$, $int_J(n) = int_I(n)$ for names $n \notin S$ and $seq_J(s) = seq_I(s)$ for sequence markers $s \notin S$.

If E is a subset of UD_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 $UD_K = E$, $rel_K(v)$ 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 **N** is a set of names, the *retraction* of I *from* **N**, [I<**N**], is the restriction of I to the set (UD_I – {$int_I(v)$: v in **N** }).

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>. In particular, s;<> = s for any sequence s.

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 marker **S**	seq_I(**S**)
E3	term sequence $\mathbf{T}_1$ … $\mathbf{T}_n$ with $\mathbf{T}_1$ a term	<I($\mathbf{T}_1$)>;I(<$\mathbf{T}_2$ … $\mathbf{T}_n$ >)
E4	term sequence $\mathbf{T}_1$ … $\mathbf{T}_n$ with $\mathbf{T}_1$ a sequence marker	I($\mathbf{T}_1$);I(<$\mathbf{T}_2$ … $\mathbf{T}_n$ >)
E5	term with operator **O** and argument sequence **S**	fun_I(I(**O**))(I(**S**))
E6	Atom which is an equation containing terms $\mathbf{T}_1$, $\mathbf{T}_2$	true if I($\mathbf{T}_1$) = I($\mathbf{T}_2$), otherwise false
E7	Atomic sentence with predicate **P** and argument sequence **S**	true if I(**S**) is in rel_I(I(**P**)), otherwise false
E8	boolean sentence of type negation and component **C**	true if I(**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 with bindings **N** and body **B**	true if for every **N**-variant J of I, J(**B**) is true; otherwise false
E14	quantified sentence of type existential with bindings **N** and body **B**	true if for some **N**-variant J of I, J(**B**) is true; otherwise false
E15	irregular sentence **S**	int_I(**S**)
E16	phrase which is a sentence **S**	I(**S**)
E17	phrase which is an importation containing name **N**	true if I($text$(I(**N**))) = true, otherwise false
E18	module with name **N**, exclusion set **L** and body text **B**	true if [I<**L**>](**B**) = true and rel_I(I(**N**)) = $UD_{[I<L]}$*, 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
E20	a text **T** with a name **N**	UR_I contains a named text value **t** with $text$(**t**) = **T** and $name$(**t**) = **N**

The meaning of the function *text* in clauses E17 for importation and E20, and the associated notion of a *named text value*, are described in the next section.

These are the basic logical semantic conditions which all conforming dialects must satisfy. A dialect may impose further semantic conditions in addition to these. A dialect with extra semantic conditions is a *semantic extension*. In particular, semantic extensions may impose syntactic and semantic conditions on irregular sentences, but **shall 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 with a comment and an empty text may be considered to be vacuously true; expressions with attached comments **shall** 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 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 phrase in Common Logic is understood to be a globally rigid identifier of that text as written (see next section), so that the same name **shall not** be used to refer to a different text, 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 [6].)

6.3.1 Importations and named phrases

The meaning of an importation phrase is that the name it contains shall be understood to identify some Common Logic text, and the importation is true just when that text 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 [7], but the definitions in this section apply to any such system of communicating agents. 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 [8] for more full 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 International Standard, 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 Common Logic semantics described in this section **shall** be applicable to all names used as network identifiers on a network on which Common Logic texts are published or transmitted.

The act of naming a text is distinct 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. The semantic entity corresponding to a named text is called a *named text value*. The exact nature of a named 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 UR_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. This requirement is included in Table 1, entry E20, which can be understood to mean:

	publication on the network of:	**requires that for any interpretation *I* of a text on the network:**
E20	a text **T** with a name **N**	UR_I contains a named text value **t** with *text*(**t**) = **T** and *name*(**t**) = **N**

Since the 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), this 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 shall be based on the assumption that the texts are mutually interpretable. For example, importing implies that the quantifiers in the imported text shall be interpreted to range over the same domain as those in the importing text. All texts which are published and identified on a network **shall** be mutually interpretable with all other texts on the network which can import them, over the same universe of reference 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 **shall not** treat communication errors or failures as an indication that a name does not denote or is a non-discourse name.

6.3.2 Mixed networks

Text may be published in more than one dialect on a single network. This International Standard refers to such a situation as a *mixed* network. Information exchange and publication on a mixed network should 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 to 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 **shall** be published in such a way that any importing of that text into another text written in an non-segregated dialect can express the content of the imported text in a way that allows mutual interpretability. This means in particular that a name shall be provided for the domain of discourse of text in any segregated text, and that any non-discourse names occurring in such text can be recognized efficiently by applications which process non-segregated text. The recommended practice in such cases is that the segregated text be replaced by non-segregated text in which all quantifiers are restricted or guarded by the segregated domain name, and all non-discourse 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-discourse names which occur in the text included in the exclusion list of the module. The module name **may** be used to identify a common universe of discourse associated with the dialect, or a local universe of discourse special to the text in the module.

Networks supporting segregated dialects which have lexical conventions for distinguishing domain from non-discourse 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 recognizes 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 Sequence markers, recursion and argument lists: discussion

Sequence markers take Common Logic beyond first-order expressivity. A sequence marker occurring in an argument sequence stands for an arbitrary finite sequence of arguments. A universal sentence binding a sequence marker has the same semantic import as the *infinite* conjunction of all the expressions obtained by replacing the sequence marker by a finite sequence of names, all bound by universal quantification.

This ability to represent infinite sets of sentences in a finite form means that Common Logic with sequence markers is not compact, and therefore not first-order; for clearly the infinite set of sentences corresponding in meaning to a single sentence quantifying a sequence marker 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 markers is to act as axiom schemata, rather than being posed as conclusions, and when they are restricted to this use the resulting logic is compact. This amounts to allowing sequence markers to be bound only by universal quantifiers at the the top phrase level of a text, and restricting these sentences to be used only as axioms, never posed as conclusions. This restriction is often appropriate for texts which are considered to be ‘ontologies’, i.e. authoritative information sources representing a conceptualization of some domain of application, intended to be applied to other data.

A compact dialect which does not support sequence markers can imitate much of the functionality provided by sequence markers, by the use of explicit argument lists, represented in Common Logic by terms built up from a list-constructing function. A sequence marker translates into the name of a list, and quantification over list names replaces quantification over sequence markers. The finiteness condition on sequences then corresponds to an implicit fixed-point assumption made on all 'standard' models of the list axioms. Such conventions are 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 domain 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, and the use of a compact base logic. 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.

6.6 Special cases and translations between dialects

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

An interpretation I is *flat* when $UD_I = UR_I$. It is *extensional* when rel_I and fun_I are the identity function on (UR_I - UD_I), so that the entities in the universe of reference outside the domain are the extensions of the non-discourse names. These are appropriate for, respectively, a non-segregated dialect, and a classical dialect. The general form of interpretation described above allows both kinds of dialects, and others, to be interpreted by a single construction.

For non-segregated 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 UR_J to be UD_I; for a non-segregated dialect, all names denote in UD_I so elements outside UD_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 x 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 predicates in a classical dialect influence the truth-conditions only through their associated extensions, this does not affect any truth-values. Formally, $UD_J = UD_I - \{I(v)$: v an operator or predicate in V $\}$, $int_J(x)=int_I(x)$ for discourse names, $int_J(x)=rel_I(int_I(x))$ for predicates x and $int_J(x)=fun_I(int_I(x))$ for operators x.

6.6.1 Translating between dialects

A *translation* is a mapping from expressions in a text in some dialect A, the source dialect, to expressions in a text in some dialect B, the target dialect, such that for every interpretation I of the vocabulary of the text in A there is an interpretation J of the vocabulary of the text in B, and for every interpretation J of the vocabulary of the text in B there is an interpretation I of the vocabulary of the text in A, with $I(E)=J(tr(E))$ for any expression E in the text in A, where tr is this translation. Since all Common Logic dialects have the same truth-conditions, translation is usually straightforward. Complications arise however in translating between segregated and non-segregated dialects.

Translation from a segregated dialect A into an non-segregated dialect B requires the translation to indicate which terms are non-discourse in A. Since all names in the non-segregated dialect denote entities in the domain, it is necessary for the translation to introduce a *discourse 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-discourse 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-discourse names of the text listed in the exclusion list of the module.

Translation from an non-segregated 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 reference of an interpretation of the non-segregated text. There is a general technique called the *holds-app translation* for translating any Common Logic dialect into a similar classical dialect. We assume that we have available a predicate *holds* and an operator *app* which do not occur in any vocabulary. Specifically (for non-segregated dialects), an atomic sentence with predicate P and argument 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. (The *holds-app* translation for segregated dialects is somewhat more complex to state but is no less obvious.)

Some dialects impose notational restrictions of various kinds, such as requiring bound names to have a particular lexical form, or requiring operator and predicates 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 are required to be made distinct in the target dialect, for example by adding suffixes to indicate the arity. In these cases also it may be necessary to introduce distinct *holds-n* and *app-n* predicates and operators for each arity. Applications which are required to faithfully translate multiple texts shall 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 **shall** be specified. Commonly this should be Unicode character strings (as specified in ISO/IEC 10646:2003), but other inscriptions e.g. diagrammatical representations such as directed graphs or structured images are possible. A method **shall** be specified for the dialect which will unambiguously parse any inscription in the set, or reject it as syntactically illegal. For Unicode character 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 clause 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 clause 6.1.1. For Common Logic conformity, dialects or sub-dialects whose parsings include other categories of sentences **shall** either (a) categorize them as irregular sentences or (b) specify how these categories to be mapped into the abstract syntax categories defined in 6.1.1. If a dialect conforms as in (a), such a dialect or sub-dialect shall be referred to as *semantic extensions* (see section 7.1.2 below). It is conformant as a ***syntactic sub-dialect*** if it recognizes at least one of the CL categories; but any dialect **shall** recognize some form of sentence category. One particular case of syntactic sub-dialect is identified, called a ***compact sub-dialect*** which is a dialect that recognizes all categories except sequence markers.

A dialect is ***syntactically segregated*** if the parsing requires a distinction to be made between lexical categories of CL names in order to check legality of an expression in that dialect. Segregated dialects **shall** 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 than the name itself.

7.1.2 Semantics

Any CL dialect **shall** 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 (except comment) or text in that dialect.

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

- For every dialect interpretation J of T, there exists a Common Logic interpretation I of T with I(T) = J (T)
- For any Common Logic interpretation I of T, there exists a dialect interpretation J of T with 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 clause 7.1.1 above) and meets the semantic conditions; that is, it recognizes only some parts of the full Common Logic and its interpretations are equivalent to the restrictions of a Common Logic interpretation to those parts.

A ***semantic extension*** is a dialect which satisfies the first condition, but does not satisfy the second 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. Any dialect which imposes non-trivial semantic conditions on irregular sentences is a semantic extension in this sense.

This allows a semantic extension to apply "external" semantic conditions to irregular sentences, in addition to the CL semantic conditions. CLIF is an example of a semantic extension, by virtue of the semantic conditions it imposes on numbers and quoted strings.

Semantic extensions **shall** 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 shall, 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-discourse in this sense, and classify it syntactically as a non-discourse name, 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 shall 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 discourse names, even those which are declared in a segregated dialect to be non-discourse.

7.2 Application conformance

"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 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.

All conformant applications **shall** 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, **shall** be capable of round-tripping any CL text; that is, they shall output or transmit the exact inscription that was input to them, without textual 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 which is not common-logic entailment. 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 Network conformance

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, and provides for network identifiers which satisfy the semantic conditions E17, E20 and as described in clause 6.2. Network transmission errors or failures which are indicated as error conditions do not count as distortion for purposes of conformance of a network.

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 [3] 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 markers, 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 shall 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 ’small’. 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.

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].

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 International Standard, and to distinguish it from various other dialects of KIF that may or may not be exactly compatible.

KIF and CLIF are similar in several ways. Both languages contain as sub-dialects a syntax for classical first-order (FO) logic. Both languages have notation for sequence variables (called sequence markers in this International Standard). Both languages use exclusively a prefix notational convention, and S-expression style syntax conventions. Both use parentheses as lexical delimiters. Both indicate quantifier restrictions similarly.

Some known differences between KIF and CLIF are as follows:

1. KIF requires ASCII encoding; CLIF uses Unicode encoding.
2. KIF has explicit notations for defining functions and relations, which CLIF does not.
3. KIF does not use the enclosed-name notation which CLIF has.

4. KIF uses the '@' symbol as a sequence variable prefix; CLIF uses the three-dot sequence for sequence markers.

5. KIF handles comments differently than CLIF and does not have the 'enclosing' construction.

6. KIF does not have the role-pair construction which CLIF has.

7. KIF does not have the notions of importation, texts, phrases, and modules which CLIF has.

8. KIF distinguishes variables from names, and requires quantifiers to bind only variables: CLIF does not make the distinction.

9. Free variables in KIF are treated as universally quantified. Free names in CLIF are simply names, and no quantification is implied.

10. KIF restricts operators and predicates to be names; CLIF allows general terms, and also allows these names to be bound by quantifiers.

11. KIF does not support the guarded quantifier construction.

A.2 CLIF Syntax

The following syntax is written using Extended Backus-Naur Form (EBNF), as specified by ISO/IEC 14977:1996. Literal chararacters are 'quoted', sequences of items are separated by commas, | indicates a separation between alternatives, { } indicates a sequence of zero or more expressions in the enclosed category, - 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 expression is encoded as a sequence of Unicode characters as defined in ISO/IEC 10646:2003. Any character encoding which supports the repertoire of ISO/IEC 10646:2003 may be used, but UTF-8 (ISO/IEC 10646:2003, 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 International Standard 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 2382-15:1999, clause 15.01 on lexical tokens). 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 shall 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. In both cases, a backslash is indicated by two backslashes \\ (U+005C, U+005C). Any other occurrence of the backslash character is an error. These inner-quote conventions apply in both ASCII and full Unicode renderings.

A.2.2 Lexical syntax

We make a distinction between lexical and syntactic constructs for convenience in dividing up the presentation into two parts. This sub-clause may help implementers in identifying logical tokens that make up syntactic expressions, as shown in the next sub-clause A.2.3. Implementations are not required to adhere to this distinction.

A.2.2.1 White space

```
whitechar = space U+0020 | tab U+0009 | line U+000A | page U+000C | return U+000D

white = whitechar |

      '/*' , {char - '*' | '*' , char - '/' | open | close | namequote | stringquote |
      backslash | whitechar }, ['*'] , '*/' |

       '//' {char | open | close | namequote | stringquote | backslash | space | tab },
      (page | line | return)  ;
```

This allows temporary comments to be inserted into CLIF text, following C++/Java conventions. Text on a line after `'//'`, and entire text blocks surrounded by `'/* ... */'`, are treated as whitespace by any CLIF parser.

The quoting sequences `'//'`, `'/*'` and `'*/'` trigger this production only when they occur outside a quoted string or enclosed name. Names in CLIF text which contain the character sequences `'//'`, `'/*'` or `'*/'` should therefore be written as enclosed names.

Temporary comments are distinct from CL comments, which are a permanent part of the CLIF parsed text. Since they are counted as whitespace, temporary comments act as lexical break characters.

A.2.2.2 Delimiters

Single quote (apostrophe) is used to delimit quoted strings, and double quote to delimit enclosed names, which obey special lexicalization rules. 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 =  '"' ;

backslash =  '\' ;
```

A.2.2.3 Characters

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' ;
```

A.2.2.4 Quoting within strings

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.

NOTE 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 \uxxxx and \Uxxxxxx 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 = '\"' ;

innerbackslash = '\\' ;

numeral = digit, { digit } ;
```

Sequence markers are a distinctive syntactic form with a special meaning in Common Logic. Note that a bare ellipsis without any text (i.e., '**...**') is itself a sequence marker.

```
seqmark = '...' , { 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 discourse 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. It is recommended to use the enclosed-name syntax when writing URIs, URI references and IRIs as names, since these Web identifiers may contain characters

which would otherwise break CLIF lexicalization: in particular, Xpath-compliant URI references will often end in a closing parenthesis.

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.

A.2.2.5 Quoted strings

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 a text or document 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 | innerbackslash }, stringquote ;

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

A.2.2.6 Reserved tokens

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 CLIF text.

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

A.2.2.7 Name character sequence

A *namecharsequence* is a lexical token which does not start with any of the special characters. Note that namecharsequences may not contain whitespace or parentheses, and may not start with a quote mark although they may contain them. Numerals and sequence markers are not namecharsequences.

```
namecharsequence = ( char , { char | stringquote | namequote | backslash } ) - (
     reservedelement  | numeral | seqmark ) ;
```

A.2.2.8 Lexical categories

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 '**'**'), sequence markers (which begin with '...'), enclosed names (which begin and end with '**"**') , and namesequences and reserved elements.

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

nonlexbreak = numeral | quotedstring | seqmark | reservedelement | namecharsequence |
     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.

A.2.3.1 Term sequence

Both terms and atomic sentences use the notion of a sequence of terms representing a vector of arguments to a function or relation. Sequence markers are used to indicate a subsequence of a term sequence; terms indicate single elements.

```
termseq = { term  | seqmark } ;
```

A.2.3.2 Name

A name is any lexical token which is understood to denote. We distinguish the names which have a fixed meaning from those which are given a meaning by an interpretation.

```
interpretedname =  numeral | quotedstring ;

interpretablename =  namecharsequence | enclosedname ;

name = interpretedname | interpretablename ;
```

A.2.3.3 Term

Names count as terms, and a complex (application) term consists of an operator, which is itself a term, together 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). Comment wrappers syntactically enclose the term they comment upon.

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

operator = term  ;
```

A.2.3.4 Equation

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 name.

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

A.2.3.5 Sentence

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 ;
```

A.2.3.6 Atomic sentence

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. Equations are considered to be atomic sentences, and an atomic sentence may be represented using role-pairs consisting of a role-name and a term.

```
atomsent = equation | atom ;

atom = ( open, predicate , termseq, close ) | ( open, term, open, 'roleset:' , { open,
     name, term, close }, close, close ) ;

predicate =  term ;
```

A.2.3.7 Boolean sentence

Boolean sentences require implication and biconditional to be binary, but allow conjunction and disjunction 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, ('if' | 'iff') ,
     sentence , sentence, close ) | ( open, 'not' , sentence, close ;
```

A.2.3.8 Quantified sentence

Quantifiers may bind any number of variables and may be guarded; and bound variables may be restricted to a category indicated by a term.

```
quantsent    = open, ('forall' | 'exists') , [ interpretablename ] , boundlist,
     sentence, close ;

boundlist    = open, { interpretablename |   seqmark | ( open, (interpretablename |
     seqmark), term, close )} , close ;
```

A.2.3.9 Commented sentence

A comment may be applied to any sentence; so comments may be attached to sentences which are subexpressions of larger sentences.

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

A.2.3.10 Module

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. The module name shall 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' , interpretablename ,   [open, 'cl:excludes' , {name}   ,
     close ] , cltext, close;
```

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

A.2.3.11 Phrase

CLIF text is a sequence of phrases, each of which is either a sentence, a module, an importation or a plain text with an attached comment. The commented text may be empty, or may be a single sentence. Text may be assigned a name in the same way as a module, but in this case the name serves only to identify the text and does not restrict the universe of discourse. A single module may also be treated as a text. Any name assigned to a named text or a module, and any name occurring inside an importation, shall be a network identifier. For Web applications at the time of writing, it should be an IRI [2]. Particular applications may impose additional conditions on names used as identifiers. The only nonterminal character for this grammar is `<code>cltext</code>`.

```
phrase = sentence | module | (open, 'cl:imports' , interpretablename , close) | (open,
     'cl:comment', quotedstring, cltext, close);

cltext = { phrase }  ;

namedtext = open, 'cl:text' interpretablename, text, close ;

cltext = module | namedtext | text ;
```

A.3 CLIF semantics

We will use both some notions and some notation that are defined in section 6.2, in particular the notation <..>;<..>.

Let IN be the set of all CLIF interpreted names, i.e. all decimal numerals and quoted strings, and let N be the set of all the natural numbers and all finite strings of Unicode characters. A CLIF vocabulary V = VN ∪ VS is a disjoint union of a set VN of interpretable names and VS of sequence markers.

A CLIF interpretation I of a vocabulary V is a structure consisting of a set U_I, called the *universe*, which is a superset of N, and two mappings rel_I from U_I to subsets of U_I* and fun_I from U_I to functions from U_I* to U_I; and a mapping int_I, on V from VN to U_I and from VS to U_I*. As in 6.2, for any subset S of V, an interpretation J of V is an *S-variant* of *I* if *J* is just like *I* except that int_I and int_J might differ on what they assign to the members of S.

NOTE CLIF does not distinguish between the universes of reference and discourse, since all names refer.

The interpretation of any expression of CLIF is then determined by the entries in Table A.1. The notation <$\mathbf{T}_1$ … $\mathbf{T}_n$> indicates a term sequence when referring to the syntax, and a sequence, i.e. an element of U_I*, when referring to the semantics.

The first column indicates links to rows in the CL semantics Table 1 in clause 6.2.

Table A.1 — CLIF Semantics

	If E is an expression of the form	**Then *I*(E) =**
E1	A decimal numeral	The natural number denoted by the decimal numeral.
E1	A quoted string '**s**'	The Unicode character string formed by removing the outer single quotes and replacing escaped inner substrings by their Unicode equivalents.
E1, E2	An interpretable name	int_I(**E**)
E3	A term sequence <$\mathbf{T}_1$ … $\mathbf{T}_n$> starting with a term $\mathbf{T}_1$	<I($\mathbf{T}_1$)>;I(<$\mathbf{T}_2$ … $\mathbf{T}_n$>)
E4	A term sequence $\mathbf{T}_1$ … $\mathbf{T}_n$ starting with a sequence marker $\mathbf{T}_1$	I($\mathbf{T}_1$); I(<$\mathbf{T}_2$ … $\mathbf{T}_n$>)
E5	A term (**O** $\mathbf{T}_1$ … $\mathbf{T}_n$)	fun_I(I(**O**))(I(<$\mathbf{T}_1$ … $\mathbf{T}_n$>)
	A term (cl:comment 'string' **T**)	I(**T**)
E6	An equation (= $\mathbf{T}_1$ $\mathbf{T}_2$)	true if I($\mathbf{T}_1$) = I($\mathbf{T}_2$), otherwise false
E7	An atomic sentence (**P** $\mathbf{T}_1$ … $\mathbf{T}_n$)	true if I(<$\mathbf{T}_1$ … $\mathbf{T}_n$>) is in rel_I(I(**P**)), otherwise false
E8	A boolean sentence (not **P**)	true if I(**P**)=false, otherwise false
E9	A boolean sentence (and $\mathbf{P}_1$ … $\mathbf{P}_n$)	true if I($\mathbf{P}_1$) =…I($\mathbf{P}_n$)= true, otherwise false
E10	A boolean sentence (or $\mathbf{P}_1$ … $\mathbf{P}_n$)	false if I($\mathbf{P}_1$) = …I($\mathbf{P}_n$) = false, otherwise true
E11	A boolean sentence (if **P** **Q**)	false if I(**P**) = true and I(**Q**) = false, otherwise true
E12	A boolean sentence (iff **P** **Q**)	true if I(**P**) = I(**Q**), otherwise false
	A sentence (cl:comment "string" **P**)	I(**P**)

E13	A quantified sentence (forall (**N**$_1$ … **N**$_n$) **B**) where **N** = {**N**$_1$, …, **N**$_n$} is the set of bindings for the sentence	true if for every **N**-variant *J* of *I*, *J*(**B**) = true, otherwise false.
E14	A quantified sentence (exists (**N**$_1$ … **N**$_n$) **B**) where **N** = {**N**$_1$, …, **N**$_n$} is the set of bindings for the sentence	true if for some **N**-variant *J* of *I*, *J*(**B**) = true, otherwise false.
	A phrase (cl:comment "string")	true
E17	A phrase (cl:imports **N**)	true if I(text(I(**N**))) = true, otherwise false
E19	A phrase (cl:text **T**$_1$ … **T**$_n$)	true if I(**T**$_1$)= … = I(**T**$_n$) = true, otherwise false
E20	(cl:text **N T**$_1$ …**T**$_n$)	true if there is a named text value **t** in U with text(**t**) = (cl:text **T**1 … **T**$_n$) , name(**t**)=**N**, and I(**N**)=**t** ; otherwise false

Not every CLIF syntactic form is covered by this table. The interpretation of the remaining syntactic cases is defined by mapping them to other CLIF expressions whose interpretation is defined by the above table. The translation is defined by Table A.2, which defines the translation **T[E]** of the expression **E**.

Table A.2 — Mapping from additional CLIF forms to core CLIF forms

If E is	**Then E translates to T[E] =**
An atomic sentence of the form (**T**$_0$ (roleset: (**N**$_1$ **T**$_1$) …(**N**$_n$ **T**$_n$)))	The sentence (exists (**X**)(and (**T**$_0$ **X**)(**N**$_1$ **X T**1)…(**N**$_n$ **X T**$_n$))) where **X** is a new name which does not occur in the atomic sentence or any containing sentence.
A quantified sentence (forall ((**N**$_1$ **T**$_1$) …) **B**)	The quantified sentence (forall (**N**$_1$) **T[** (forall (…) (if (**T**$_1$ **N**$_1$) **B**) **]**
A quantified sentence (exists ((**N**$_1$ **T**$_1$) …) **B**)	The quantified sentence (exists (**N**$_1$) **T[** (exists (…) (and (**T**$_1$ **N**$_1$) **B**) **]**
A quantified sentence (forall **G** (…) **B**)	The quantified sentence **T[** (forall (…)(if (**G X**$_1$ …**X**$_n$) **B**) **]** where **X**$_1$ … **X**$_n$ are all the names which occur free in **B**
A quantified sentence (exists **G** (…) **B**)	The quantified sentence **T[** (exists (…)(and (**G X**$_1$ …**X**$_n$) **B**) **]** where X1 … Xn are all the names which occur free in **B**
A module (cl:module **N** (cl:excludes **N**$_1$ ….. **N**$_n$) **T**)	The text (not (**N N**$_1$)) … (not (**N N**$_n$)) **T**[**T**']) Where **T**' is the text **T** in which every name or sequence marker **X** in the boundlist of a quantifier is replaced with (**X N**)

The forms on the left side of Table A.2 can be considered to be 'syntactic sugar' for their translations on the right, which are correspondingly referred to as their *sour* syntactic equivalents, and the subdialect of CLIF without these expression forms as *sour CLIF*.

A.4 CLIF conformance

The conformance of CLIF to Common Logic is demonstrated for two aspects of conformance – syntactic conformity and semantic conformity. This not only specifies CLIF's conformance itself, it also provides a guide to specifiers of other dialects so that they may see how conformance is demonstrated.

A.4.1 Syntactic conformity

The correspondence of CLIF syntax to the CL abstract syntax is indicated by the entries in the left column of the first table, which refer to the entries in Table 1, and from which the full syntactic conformance of sour CLIF can be determined by inspection. Note that both `interpretednames` and `interpretablenames` are considered to be CL `name`s. The syntactic conformity of CLIF then follows by virtue of the the mapping defined by the second table. Note that the CLIF comments syntax treats a commented expression as identical in meaning to the expression without the comment, so the comment can be considered to be 'attached' to the uncommented expression.

CLIF is syntactically segregated, by virtue of the restrictions it imposes on where interpreted names may occur in expressions; but it is not a segregated dialect in the sense of section 7.1

A.4.2 Semantic conformity

CLIF is a CL semantic extension. To show that CLIF is a CL semantic extension it is necessary to show that if I is a CLIF interpretation, then a CL interpretation J must exist which gives the same truth value to every sentence. This will be demonstrated by constructing J from I. using the notation and conventions from above when describing I, and from section 6.2 when describing J.

J has the same vocabulary as I, $UD_J = UR_J = U_I$, $rel_J = rel_I$ and $fun_J = fun_I$. The interpretation of interpretablenames is defined in the obvious way: $int_J(x) = int_I(x)$ for any interpretablename x. Since the interpretednames of a CLIF vocabulary are classified as CL names, we must also define $int_J(x)$ when x is an interpretedname, and clearly this is done to follow the first two entries in the CLIF semantic table, i.e. $int_J(x)$ = the integer denoted by x when x is a decimal numeral, and $int_J(x)$ = the Unicode character string denoted by x when x is a CLIF quoted string. It is then easy to see by a comparison of cases that J(s) = I(s) for any CLIF sentence s. If s is a module named N with an exclusion list L and a body B, then we need to show that J(s) = true just when [J<L](B) = true and rel(J(N)) = $UR_{[J<L]}$* (since $UD_J = UR_J$). It is easy to see that this is exactly equivalent to the truth in I of sentences in the sour translation of the module body text defined by the second table above, as described in section 6.2. (A formal proof would proceed by a structural induction on the sentences of the body text.) Hence, for any CLIF text t, J(t) = I(t).

It is not the case that if I is any CL interpretation of a CLIF text t, that there must be a CLIF interpretation J which gives t the same value, for since CLIF interpretednames are treated simply as names in CL, J may assign them a value which does not conform to their fixed interpretation in CLIF, e.g. J('a string') = 3 is not ruled out by the common logic semantics rules. This is a general phenomenon with any dialect which imposes predetermined, externally defined, meanings on some category of names, such as numerals or datatyped expressions. Such dialects may support inferences which cannot be expressed as CL axioms, and must be classified as external CL semantic extensions. The subdialect of CLIF which does not use numerals or quoted strings is exactly semantically conformant, as can be shown by inverting the avove construction of J from I.

Annex B
(normative)

Conceptual Graph Interchange Format (CGIF)

B.1 Introduction

This sub-clause summarizes conceptual graphs and then describes a set of transformation (rewrite) rules that will be used in the rest of this Annex to specify the description of the syntactic rules for CGIF.

B.1.1 Conceptual Graphs

A conceptual graph (CG) is a representation for logic as a bipartite graph with two kinds of nodes, called *concepts* and *conceptual relations*. The *Conceptual Graph Interchange Format* (CGIF) is a fully conformant dialect of Common Logic (CL) that serves as a serialized representation for conceptual graphs. This annex specifies the CGIF syntax and its mapping to the CL semantics. A nonnormative graphical notation, called the *CG display form*, is used in this International Standard only in examples that illustrate the CG structures. The first example, Figure B.1, shows the display form that represents the sentence *John is going to Boston by bus*.

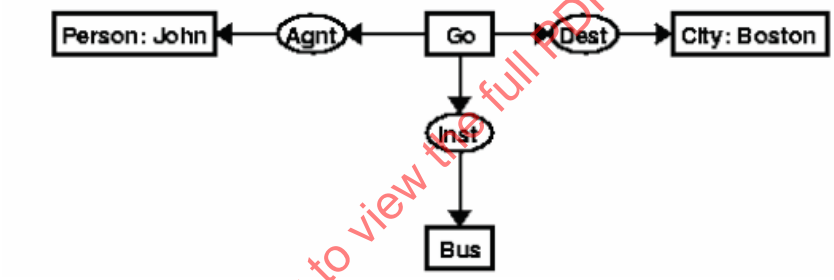

Figure B.1 — CG display form for John is going to Boston by bus

In the display form, rectangles or boxes represent concepts, and circles or ovals represent conceptual relations. An arc with an arrowhead pointing toward a circle marks the first *argument* of the relation, and an arc pointing away from a circle marks the last argument. If a relation has only one argument, the arrowhead is omitted. If a relation has more than two arguments, 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`, and `Bus`. Two of the concepts have *constants* that identify individuals: `John` and `Boston`. Each of the three conceptual relations has a type label that represents the type of relation: `Agnt` for the agent of going, `Inst` for the instrument, and `Dest` for the destination. The CG as a whole indicates that the person John is the agent of an instance of going with Boston as the destination and a bus as the instrument. Following is the CGIF representation of Figure B.1:

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

In CGIF, the concepts are represented by square brackets, and the conceptual relations are represented by parentheses. A character string prefixed with an asterisk, such as `*x`, is a *defining label*, which may be referenced by the *bound label* `?x`, which is prefixed with a question mark. These strings, which are called *coreference labels* in CGIF correspond to variables in Common Logic Interchange Format (CLIF). Unless prefixed with the symbol `@every`, a defining label is translated to an existential quantifier. Following is the equivalent CLIF representation of Figure B.1:

```
(exists ((x Go) (y Bus))
   (and (Person John) (city Boston)
        (Agnt x John) (Dest x Boston) (Inst x y)))
```

As this example illustrates, the differences between CGIF and CLIF result from the graph structure: the nodes of the graph have no implicit ordering, and the coreference labels such as `*x` or `?x` represent connections of nodes rather than variables. Note that CGIF uses the prefixes `*` and `?` to distinguish coreference labels from constants, but CLIF does not use any syntactic convention for distinguishing variables and constants.

Figure B.1 and its representation in CGIF illustrate the *extended syntax* of CGIF, which adds type labels on concepts and several other syntactic extensions to the *core syntax*. To convert the extensions of Figure B.1 to the core CGIF, the type labels in the concept nodes are replaced by relations linked to the nodes. The concept `[Go:*x]`, for example, becomes an untyped concept `[*x]` and a conceptual relation `(Go ?x)`. The concept `[Person: John]` becomes `[:John] (Person John)`, which may be simplified to just the relation `(Person John)`. Following is the core CGIF and the corresponding CLIF:

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

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

To illustrate *contexts* and logical operators, Figure B.2 shows the display form for the sentence *If a cat is on a mat, then it is a happy pet.* 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 the CG in the then-context. The `Attr` relation indicates that the cat, also called a pet, has an attribute, which is an instance of happiness.

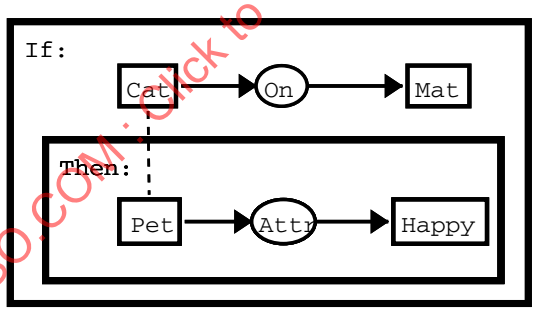

Figure B.2 — CG display form for "If a cat is on a mat, then it is a happy pet"

The dotted line connecting the concepts `[Cat]` and `[Pet]` is a *coreference link*, which indicates that they both refer to the same entity. In CGIF, the connection is shown by the defining label `*x` in the concept `[Cat: *x]` and the bound label `?x` in the concept `[Pet: ?x]`:

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

In core CGIF, the type labels `If` and `Then` are replaced by a negation symbol `~` in front of the opening bracket, and the type labels are replaced by monadic relations:

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

CLIF:

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

In core CGIF, the only quantifier is the existential. In extended CGIF, universal quantifiers may be used to represent the logically equivalent sentence *For every cat and every mat, if the cat is on the mat, then it is a happy pet*. In extended CGIF, the universal quantifier is represented as `@every`:

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

CLIF:

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

In CGs, functions are represented by conceptual relations called *actors*. Figure B.3 is the CG display form for the following equation written in ordinary algebraic notation:

```
y = (x + 7)/sqrt(7)
```

The three functions in this equation would be represented by three actors, which are drawn in Figure B.3 as diamond-shaped nodes with the type labels `Add`, `Sqrt`, and `Divide`. The boxes represent concept nodes, which contain the input and output values of the actors. The two empty concepts contain the output values of `Add` and `Sqrt`.

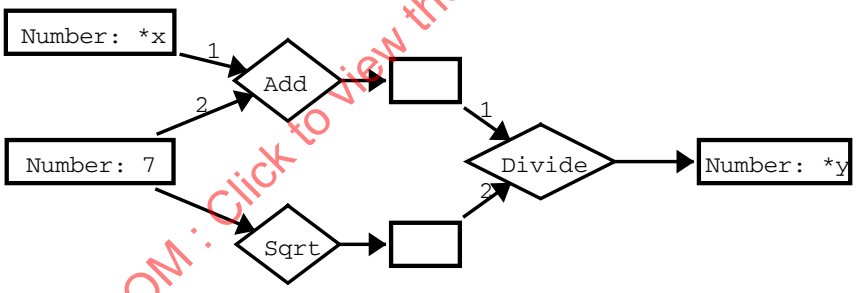

Figure B.3 — CL functions represented by actor nodes

In CGIF, actors are represented as relations with two kinds of arcs: a sequence of *input arcs* and a sequence of *output arcs*, which are separated by a vertical bar:

```
[Number: *x] [Number: *y] [Number: 7]
(Add ?x 7 | [*u]) (Sqrt 7 | [*v]) (Divide ?u ?v | ?y)
```

In the display form, the input arcs of `Add` and `Divide` are numbered 1 and 2 to indicate the order in which the arcs are written in CGIF. Following is the corresponding CLIF:

```
(exists ((x Number) (y Number))
   (and (Number 7) (= y (Divide (Add x 7) (Sqrt 7)))))
```

No CLIF variables are needed to represent the coreference labels `*u` and `*v` since the functional notation used in CLIF shows the connections directly.

All semantic features of CL, including the ability to quantify over relations and functions, are supported by CGIF. As an example, someone might say "Bob and Sue are related," but not say exactly how they are related. The following sentences in CGIF and CLIF state that there exists some familial relation *r* that relates Bob and Sue:

```
[Relation: *r] (Familial ?r) (#?r Bob Sue)

(exists ((r Relation)) (and (Familial r) (r Bob Sue)))
```

The concept `[Relation: *r]` states that there exists a relation *r*. The next two relations state that *r* is familial and *r* relates Bob and Sue. In CGIF, the prefix `#` indicates a bound coreference label used as a type label.

B.1.2 EBNF Syntax Rules for CGIF (informative)

In order to describe the syntax of CGIF, the EBNF notation is used, as referenced in ISO/IEC 14977:1996. The specifications in Annex B use only the following subset of the features specified by ISO/IEC 14977:1996. This section is intended as informative only, as ISO/IEC 14977:1996 should be considered the normative reference.

Terminal symbol. Any string enclosed in either single quotes or double quotes. Examples:

```
"This is a quoted string." 'and so is this'
```

Nonterminal symbol. A name of a category in a syntax rule. For example, the following syntax rule contains two nonterminal symbols, one terminal symbol `'";"'`, a defining symbol `"="`, a concatenation symbol `","`, and a terminator symbol `";"`.

```
syntaxRule   =  expression, ";";
```

Option. An expression enclosed in square brackets. It specifies zero or one occurrence of any string specified by the enclosed expression. Example:

```
[ "This string may or may not occur." ]
```

Iteration. An expression enclosed in curly braces. It specifies zero or more occurrences of any string specified by the enclosed expression. Example:

```
{ "This string may occur many times." }
```

Concatenation. Two or more terms separated by commas.

```
"Two kinds of quotes: ", "'", " and ", '"', "."
```

Exception. Two terms separated by a minus sign `-`, which specifies any string specified by the first term, but not the second. The following example specifies a sequence of zero or more digits that does not contain "6":

```
{digit} - 6
```

Group. An expression enclosed in parentheses and treated as a single term. The following group encloses an exception that specifies a sequence of one or more digits by excluding the empty term:

```
({digit} - )
```

Alternatives. Two or more concatenations separated by vertical bars. Example:

```
"cat", "dog" | "cow", "horse", "sheep" | wildAnimal
```

Special sequence. Any string enclosed by question marks. These sequences shall not affect the syntax specified by the syntax rules, but they may be used to copy strings analyzed by a syntax rule for later use by the rewrite rules specified in Section B.5.2. Example:

```
?sqn?
```

Syntax rule. A nonterminal symbol followed by "=" followed by an expression and ending with ";". The following syntax rules define the syntax of the syntax rules used in Annex B.

```
syntaxRule    =  expression, ";";
expression    =  alternative, {"|" alternative} | term, "-", term;
alternative   =  term [variable], {"," term [variable]};
term          =  terminal | nonterminal | "[", expression, "]"
                 | "{", expression, "}" | "(", expression, ")" | empty;
terminal      =  "'", ({character - '"'} - empty), "'"
                 | '"', ({character - "'"} - empty), '"';
nonterminal   =  identifier;
variable      =  "?", identifier, "?";
identifier    =  letter, {letter | digit | "_"};
empty         =  ;
```

These rules specify a subset of the syntax rules specified in Section 8.1 of ISO/IEC 14977:1996. The rules imply that "," has higher precedence than "|", which has higher precedence than "=". Parentheses may be used to override the precedence or to make the grouping more obvious.

B.1.3 Notation for Rewrite Rules

The syntax of both core (clause B.2 and extended CGIF (clause B.3) is defined by rules in Extended Backus-Naur Form (EBNF) rules as specified by ISO/IEC 14977:1996. To specify the translation from core CGIF to Common Logic, Section B.2 uses a combination of EBNF rules and mathematical notation supplemented with English. To specify the translation from extended CGIF to core CGIF, clause B.3 uses a combination of EBNF rules in this section and the rewrite rules defined in clause B.1.3.2. The syntax rules in Annex B presuppose a lexical analysis stage that has subdivided the text into *tokens* as in ISO/IEC 2382-15:1999 (clause 15.01 on lexical tokens); therefore at any point where a comma occurs in an EBNF rule, zero or more characters of white space may occur in the input text.

B.1.3.1 Transformation Rules

Each transformation rule shall define a function that analyzes an input string and returns a sequence of one or more output strings. A transformation rule shall have three parts: a *header*, a syntax rule as defined in B.1.2, and zero or more *rewrite rules*. The first string in a header shall specify the name of the function, which shall also be the name of the nonterminal symbol defined by the syntax rule. The header shall also specify a variable whose value shall be the *input string* to be analyzed by the syntax rule, and it shall specify a sequence of one or more *output variables*. If the syntax rule successfully analyzes the input string from beginning to end, the rewrite rules, if any, are executed. Following are the syntax rules that define the syntax of the transformation rules; transRule is the start symbol.

```
transRule     =  header, syntaxRule, {rewriteRule}, "end", ";";
header        =  nonterminal, "(", variable, ")", "->",
                 variable, {"," variable};
rewriteRule   =  assignment | conditional;
assignment    =  variable, "=", rewriteExpr, ";";
conditional   =  "if", condition, ({rewrite rule} - empty),
                 {"elif", condition, ({rewrite rule} - empty)},
                 ["else", ({rewrite rule} - empty)], "end;"
condition     =  "(", test, {"&", test}, ")";
test          =  rewriteTerm, ["~"], "=", rewriteTerm;
test          =  rewriteTerm, ["~"], "=", rewriteTerm;
rewriteExpr   =  rewriteTerm {"," rewriteTerm};
```

```
rewriteTerm  =  terminal | variable | funTerm;
funTerm      =  identifier, "(", [funTerm, {"," funTerm}, ")";
```

The following nonterminal symbols from ISO/IEC 24977 shall be defined as in B.1.2: syntaxRule, terminal, nonterminal, variable, identifier, empty.

The function defined by a transformation rule shall translate the input string to the sequence of values of the output variables by copying substrings from the input and executing rewrite rules to transform those strings. The execution shall be determined by the following procedure:

Any parsing algorithm may be used to the analyze the input string according to the specifications of the syntax rule. At the beginning of the analysis, all variables that occur in the transformation rule shall be initialized to the empty string. Although some parsing algorithms may assign values to variables during the parsing phase, the semantics shall not require those values to be accessible for executing any rewrite rules until after all parsing has finished.

Any variable *x* in the syntax rule shall occur immediately after some term *t* in that rule; there shall be no comma or other symbol separating *t* and *x*. The value assigned to *x* shall be the substring *s* of the input string that was matched to the pattern specified by *t*. If the alternative in which *t* occurs was not taken or if *t* matched the empty string, the value of *x* shall be empty.

After parsing has finished, the rewrite rules following the syntax rule are executed sequentially, unless one or more rewrite rules in the options of a conditional are skipped.

When an assignment is executed, the values of the terminals, variables, and functional terms on the right side of the rule shall be concatenated in the order in which they are written. The resulting string shall be assigned as the value of the variable on the left side of the rule.

A condition that occurs in a conditional is a conjunction of one or more tests for the equality or inequality of the values of two terms. An empty term, which is written as a blank, has the empty string as value. Therefore, the condition `(?x?= & ?y?~= )` shall be true if and only if `?x?` is empty and `?y?` is not empty.

When a conditional is executed, the conditions for the `if`, `elif`, and `else` options shall be evaluated sequentially. (The condition for `else` shall always be true.) When the first true condition is found, the rewrite rules following that condition shall be executed sequentially until the next occurrence of `elif`, `else`, or `end` for that rule is found. Then execution shall continue with the rewrite rule, if any, which occurs after the `end` marker for that conditional.

When the end marker for the transformation rule is reached, execution shall stop. Then the value of the function named in the header shall be a sequence of the values of all the output variables. Any output variable that had not been assigned a value shall have the value of the empty string. Any output variable that has the same identifier as some variable in the syntax rule shall have the value assigned to it from the input string. No assignment shall change the value of any variable after a value has been assigned to it.

According to this specification, some transformation rules may have no rewrite rules. The following rule, for example, defines an identity function, whose output is identical to its input:

```
identity(?s?) -> ?t?;
identity = {character} ?t?;
end;
```

The input string *s* is parsed by the syntax rule as a string of zero or more characters. That string is assigned to *t*, which becomes the output of the function.

The value assigned to a variable as a result of the parse is always some substring from the input. Except for the identity function, the output values generated by the rewrite rules for any syntactic category are often very different from any substring of the input. As an example, the transformation rule named `negation` translates a negation from extended CGIF to core CGIF:

```
negation(?b?) -> ?ng?;
negation  =  "~[", [comment] ?cm?, CG ?x?, [endComment] ?ecm?, "]";
?ng?  =  "~[", ?cm?, CG(?x?), ?ecm?, "]";
end;
```

The strings for the opening comment *cm* and the ending comment *ecm* are copied unchanged from input to output. But the nested CG, whose input string *x* is in extended CGIF, is very different from the core CGIF output of CG(*x*). The transformation rules for the syntactic categories of extended CGIF behave like compilers that translate input strings for extended CGIF categories to output strings in core CGIF.

B.1.3.2 Functions Used in Rewrite Rules

Any function defined by a transformation rule may be used in a rewrite rule. It may even be used recursively in the same transformation rule that defines it. In addition to the functions defined by transformation rules, the following seven functions shall be available for use in processing strings or sequences in any rewrite rule.

- **first**(*s*) shall return the first or only element of a sequence *s*. If length(*s*)="0", first(*s*) shall be empty.
- **gensym**() shall return a string that represents a CGname that shall be different from any other CGname in the current text. Each time gensym() is invoked, the string it returns shall also be different from any string it had previously returned.
- **length**(*s*) shall return the length of the sequence *s* as a string of one or more characters that represent the decimal digits of the length. If *s* is empty, length(*s*) shall be "0". If *s* is a single element, length(*s*) shall be "1".
- **map**(*f,s*) shall apply a function *f* to each element of a sequence *s* in order to return the sequence of values of *f*(*x*) for each *x* in *s*.
- **second**(*s*) shall return the second element of a sequence *s*. If length(*s*)<"2", second(*s*) shall be empty.
- **substitute**(*s,t,x*) shall return the result of substituting the string *s* for every occurrence of the string *t* in the string *x*. If *t* does not occur in *x*, substitute(*s,t,x*) shall be *x*.
- **third**(*s*) shall return the third element of a sequence *s*. If length(*s*)<"3", third(*s*) shall be empty.

The English phrase “CG name” shall refer to any syntactic token of the category “CGname”.

B.2 CG Core Syntax and Semantics

The CG abstract syntax is a notation-independent specification of the expressions and components of the *conceptual graph core*, which is the minimal CG subset capable of expressing the full CL semantics. The semantics of any expression *x* in the CG core syntax is specified by the function *cg2cl*(*x*), which maps *x* to a logically equivalent expression in the CL abstract syntax. The function *cg2cl* is recursive, since a CG or its components may be nested inside other components.

Sections 2.1 through 2.11 define the abstract CG syntax, the mapping of the abstract CG syntax to the abstract CL syntax, and the corresponding concrete syntax for CGIF core. Each clause includes a formal definition, a mapping to CL, a syntax rule for CGIF concrete syntax, and a comment with explanation and examples. The syntax rules are written in Extended Backus-Naur Form (EBNF) rules, as specified by ISO/IEC 14977:1996 and summarized in B.1.2. For each CGIF syntax rule, the lexical categories of Section A.2.2 shall be assumed. In Section A.2.3.2, the category *name* includes a category *enclosedname* of strings enclosed in quotes and a category *namesequence* of strings that are not enclosed. To avoid possible ambiguities, the category *CGname* requires that all CLIF name sequences except those in the CGIF category *identifier* shall be enclosed in quotes:

```
CGname  =  identifier | '"', (namesequence - identifier), '"'
         | numeral | enclosedname | quotedstring;
identifier  =  letter, {letter | digit | "_"};
```

When CGIF is translated to CL, any CGname shall be translated to a CLIF name by removing any quotes around a name sequence. CLIF does not make a syntactic distinction between constants and variables, but in CGIF any CGname that is not used as a defining label or a bound label shall be called a *constant*.

The start symbol for CGIF syntax shall be the category `text`, if the input is a complete text, or the category `CG`, if the input is a string that represents a conceptual graph.

B.2.1 actor

Definition: A conceptual relation *ac*=(*r*,*s*), in which *r* shall be a reference called the *type label* of *ac* and the arc sequence $s=s_1,s_2$ shall consist of an arc sequence s_1, called the *input arcs*, and a single arc s_2, called the *output arc*.

CL: *cg2cl*(*ac*) shall be an equation *eq*: the first term of *eq* shall be the name *cg2cl*(s_2), and the second term of *eq* shall be the functional term with operator *cg2cl*(*r*) and term sequence *cg2cl*(s_1) with an optional sequence marker *sqn*.

CGIF:
```
actor  =  "(", [comment], [ "#", "?"], CGname, arcSequence, "|", arc,
          [endComment], ")";
```

Like other conceptual relations, an actor node is enclosed in parentheses. The symbol `#` shall mark a bound coreference label that is used as a type label.

Comment: Although an actor is defined as a special case of a conceptual relation, the CG core syntax restricts an actor to exactly one output arc so that it may be mapped to a CL function. See Figure B.3 for examples of actors and their mapping to CGIF and CLIF. The input arcs may include a sequence marker at the end, but no sequence marker shall be used for the output arc. The extended CGIF syntax allows actors to have any number of output arcs.

B.2.2 arc

Definition: A reference *ar* that occurs in an arc sequence of some conceptual relation.

CL: *cg2cl*(*ar*) shall be the name *n* without the marker of the reference *ar*.

CGIF:
```
arc  =  [comment], reference;
```

Comment: The function *cg2cl* maps an arc to the name of the reference and omits any marker that distinguishes a bound label.

B.2.3 arcSequence

Definition: A pair *as*=(*s*,*sqn*) consisting of a sequence *s* of zero or more arcs followed by an optional sequence marker *sqn*.

CL: *cg2cl*(*as*) shall be a term sequence *ts*=*cg2cl*(*s*) and the sequence marker *sqn* if present in *as*. The term sequence *ts* shall be map(*cg2cl*,*s*), where *map* is a function that applies *cg2cl* to each arc of the sequence *s* to extract the name that becomes the corresponding element of the sequence *ts*.

CGIF:
```
arcSequence  =  {arc}, [[comment], "?", seqmark];
```

Any sequence marker in an arc sequence *as* shall be identical to the sequence marker in some existential concept that is directly contained in a context that contains the actor or conceptual relation that has the arc sequence *as*.

Comment: The option of having a sequence marker in an arc sequence implies that a conceptual relation may have a variable number of arcs.

B.2.4 comment

Definition: A string *cm*, which shall have no effect on the semantics of any CGIF expression *x* in which *s* occurs.

CL: *cg2cl*(*cm*) shall be the substring *s* of *cm* that does not include the delimiters "/*" and "*/" of a comment or the opening ";" of an end comment. The string *s* shall be included in a CL representation for a comment and shall be associated with the CL syntactic expression to which the CGIF expression *x* is translated. The syntax rules for comment and end comment are identical for core CGIF and extended CGIF.

CGIF:

```
comment = "/*", {(character-"*") | [ "*", (character-"/")]}, [ "*"], "*/";

endComment  =  ";", {character - ("]" | ")")};
```

The string enclosed by the delimiters "/*" and "*/" shall not contain a substring "*/". The string of an end comment may contain any number of ";", but it shall not contain "]" or ")".

Comment: A comment may occur immediately after the opening bracket of any concept, immediately after the opening parenthesis of any actor or conceptual relation, immediately before any arc, or intermixed with the concepts and conceptual relations of any conceptual graph. An end comment may occur immediately before the closing bracket of any concept or immediately before the closing parenthesis of any conceptual relation or actor. Since the syntax of comments is identical in core and extended CGIF, no additional sytax rules for comments shall be included in Section B.3.

B.2.5 concept

Definition: A pair *c*=(*R*,*g*) where *R* shall be either a defining label or a set of zero or more references, and *g* shall be a conceptual graph that is said to be *directly contained* in *c*.

CL: *cg2cl*(*c*) shall be the sentence *s* determined by one of the first three options below:

Context. If *R* is empty, then *s*=*cg2cl*(*g*). In this case, *c* shall be called a *context*.

Existential. If *g* is blank and *R* is a defining label, then the sentence *s* shall be a quantified sentence of type existential with a set of names {*cg2cl*(*R*)} and with a body consisting of a Boolean sentence of type conjunction and zero components. In this case, *c* shall be called an *existential concept*.

Coreference. If *g* is blank and *R* is a set of one or more references, then let *r* be any reference in *R*. The sentence *s* shall be a Boolean sentence of type conjunction whose components are the set of equations with first term *cg2cl*(*r*) and second term *cg2cl*(*t*) for every reference *t* in *R*–{*r*}. In this case, *c* shall be called a *coreference concept*.

Syntactically invalid. The case in which *g* is nonblank and *R* is not empty is not permitted in core CGIF, and no translation to CL is defined.

CGIF:

```
concept  =  context | existentialConcept | coreferenceConcept;

context  =  "[", [comment], CG, [endComment], "]";

existentialConcept  =  "[", [comment], "*", (CGname | seqmark),
                           [endComment], "]";

coreferenceConcept  =  "[", [comment], ":", {reference}-,
                           [endComment], "]";
```

A context shall be a concept that contains a CG; if the CG is blank, the context is said to be *empty*, even if it contains one or more comments. Any comment that occurs immediately after the opening bracket shall be part of the concept; any other comments shall be part of the nested CG. A coreference concept shall contain one or more constants or bound coreference labels; in EBNF, an iteration followed by a minus sign with nothing after it indicates at least one iteration.

Comment: A context is represented by a pair of brackets, which serve to limit the scope of quantifiers of the nested CG; an empty context `[ ]` is translated to CLIF as `(and)`, which is true by definition. An existential concept is represented by a concept such as `[*x]`, which is translated to CLIF as `(exists (x) (and))`; this sentence asserts that there exists some *x*. A coreference concept is represented by a concept that contains a set of constants or bound coreference labels, such as `[: ?x Cicero Tully ?abcd]`, which is translated to a conjunction of equations in CLIF:

```
(and (= x Cicero) (= x Tully) (= x abcd))
```

A coreference concept with just one reference, such as `[:?x]`, would become an empty conjunction `(and)`. Since it has no semantic effect, such a concept may be deleted.

B.2.6 conceptual graph (CG)

Definition: A triple *g*=(*C,R,A*), where *C* is a set of concepts, *R* is a set of conceptual relations, and *A* is the set of arcs that shall consist of all and only those arcs that occur in the arc sequence of some conceptual relation in *R*. If *C* and *R* are both empty, then *A* is also empty, and *g* is called a *blank* conceptual graph.

CL: Let *E* be the subset of *C* of existential concepts; and let *X* be the set of all concepts, conceptual relations, and negations of *g* except for those in *E*.

Let *B* be a Boolean sentence of type conjunction with components consisting of all the sentences *cg2cl*(*x*) for every *x* in *X*.

If *E* is empty, then *cg2cl*(*g*) is *B*.

If *E* is nonempty, then *cg2cl*(*g*) is a quantified sentence of type existential with the set of names consisting of the CGname of the defining coreference label of every *e* in *E* and with the body *B*.

CGIF:

```
CG  =  {concept | conceptualRelation | negation | comment};
```

A conceptual graph consists an unordered set of concepts, conceptual relations, negations, and comments. Formally, a negation is a pair consisting of a concept and a conceptual relation that are never separated in CGIF.

Comment: According to this specification, every CG maps to either a quantified sentence of type existential or to a Boolean sentence of type conjunction. If the conjunction has only one component, then the sentence could be simplified to an equality, an atomic sentence, or a Boolean sentence of type negation. If *g* is blank, the corresponding CLIF is `(and)`, which is true by definition. Although there is no required ordering of the nodes of a CG, some software that processes CGIF may run more efficiently if the defining coreference labels occur before the corresponding bound labels; the simplest way to ensure that condition is to move the existential concepts to the front of any context.

B.2.7 conceptual relation

Definition: A pair *cr*=(*r*,*s*), in which *r* shall be a reference called the *type label* of *cr* and *s* shall be an arc sequence.

CL: *cg2cl*(*ac*) shall be an atomic sentence whose predicate is *cg2cl*(*r*) and whose term sequence is *cg2cl*(*s*).

CGIF:

```
conceptualRelation  =  ordinaryRelation | actor;
```

```
ordinaryRelation  =  "(", [comment], [ "#", "?"], CGname, arcSequence,
                     [endComment], ")";
```

An ordinary conceptual relation has just one sequence of arcs. An actor partitions the sequence of arcs in two subsequences. A bound coreference label that is used as a type label shall begin with the string "#?" or “#?”.

Comment: By allowing the type label of a conceptual relation to be a bound label, CGIF supports the CL ability to quantify over relations and functions. As an example, see the CGIF at the end of section B.1.1 that represents the sentence "Bob and Sue are related."

B.2.8 negation

Definition: A pair *ng*=(*c*,*cr*), in which *c* shall be a concept and *cr* shall be a conceptual relation whose type label *r* shall be a constant with CGname `Neg`. The pair (*c*,*cr*) shall be treated as a single unit.

CL: *cg2cl*(*ng*) shall be a Boolean sentence of type negation with the component *cg2cl*(*g*).

CGIF:

```
negation  =  "~", context;
```

A negation shall begin with the symbol `~`. Although a negation is formally defined as a pair consisting of a context and a conceptual relation, the two elements of the pair shall not be expressed as separate nodes in CGIF.

Comment: A negation negates the proposition stated by the nested conceptual graph *g*. For examples, see the CGIF for Figure B.2. The negation of the blank CG, written `~[ ]`, is always false; the corresponding CLIF is `(not (and))`.

B.2.9 reference

Definition: A pair *r*=(*m*,*n*) where *n* is a CG name and *m* is a *marker* that shall designate a *constant* or a *bound label*.

CL: *cg2cl*(*r*) shall be the name *n*. The marker *m* shall be `?` for a bound label and the empty string `""` for a constant.

CGIF:

```
reference  =  ["?"], CGname;
```

This syntax of references is identical in core CGIF and extended CGIF. Any CG name that conists of a quoted namesequene shall be translated to a CL name by erasing the enclosing quotes; all other CG names are identical to the corresponding CL names. Sequence markers are identical in CLIF and CGIF.

Comment: Since references are identical in core and extended CGIF, no additional syntax rules for references are included in Section B.3.

B.2.10 scope

Definition: A set of contexts *S* associated with a concept *x* that has a defining label with CG name *n*.

The following terms are used in defining the constraints on defining labels in both core and extended CGIF:

- *constant*, a CG name without any prefix.
- *bound coreference label*, a CG name with the prefix `"?"`.
- *bound sequence label*, a sequence marker with the prefix `"?"`.

- *bound label*, a bound coreference label or a bound sequence label.
- *defining coreference label*, a CG name with the prefix "*".
- *defining sequence label*, a sequence marker with the prefix "*".
- *defining label*, either a defining coreference label or a defining sequence label.

According to this definition, a defining sequence label shall begin with the string "*…" and a bound sequence label shall begin with the string "?...".

Constraints: The verb *contains* shall be defined as the transitive closure of the relation *directly contains*, and it shall satisfy the following constraints in both core and extended CGIF:

B.2.10.1 If a context *c* directly contains a conceptual graph *g*, then *c* directly contains every node of *g* and every component of those nodes, except for those that are contained in some context of *g*.

B.2.10.2 If a context *c* directly contains a context *d*, then *c* indirectly contains everything that *d* contains.

B.2.10.3 The phrase "*c* contains *x*" is synonymous with "*c* directly or indirectly contains *x*".

B.2.10.4 If a concept *x* with a defining label with name *n* is directly contained in some context *c*, then *c* shall not contain any concept other than *x* with a defining label with the same CG name *n*, and *c* shall be in the scope S associated with the concept *x*.

B.2.10.5 If a context *c* is in the scope S associated with a concept *x*, then any context *d* directly contained in *c* shall also be in the scope S, unless *d* directly contains a concept *y* with a defining label with the same CG name as the defining label of *x*.

B.2.10.6 Every bound label with CG name *n* shall be in the scope associated with some concept with a defining label with CG name *n*.

B.2.10.7 No constant with CG name *n* shall be in the scope associated with some concept with a defining label with CG name *n*.

NOTE These constraints ensure that for every CGIF sentence s, the translation *cg2cl*(s) shall obey the CL constraints on scope of quantifiers. Since the constraints on scope are identical in core and extended CGIF, no additional constraints shall be included in Section B.3.

B.2.11 text

Definition: A context *c* that is not contained directly or indirectly in any context.

CL: *cg2cl*(*c*) shall be text consisting of the sentence *cg2cl*(*g*), where *g* is the conceptual graph directly contained in *c*. If a CG name *n* occurs immediately before *g* in the CGIF specification of the context *c*, then *n* shall be the name of the CL text.

CGIF:

```
text  = "[", [comment], "Proposition", ":", [CGname], CG,
        [endComment], "]";
```

Since a text is not contained in any context, it shall also be called the *outermost context*.

Comment: This syntax rule uses the syntax of extended CGIF, which allows a context to have a type label and a CG name. Since core CGIF syntax is a subset of extended CGIF syntax, text in core CGIF can be used by any processor that accepts extended CGIF. Context brackets may be used to group the concepts and relations of a text into units that correspond to CLIF sentences. That grouping is a convenience that has no effect on the semantics.

B.3 Extended CGIF Syntax

Extended CGIF is a superset of core CGIF, and every syntactically correct sentence of core CGIF is also syntactically correct extended CGIF. Its most prominent feature is the option of a *type label* or a *type expression* on the left side of any concept. In addition to types, extended CGIF adds the following features to core CGIF:

- more options in concepts, including universal quantifiers;
- Boolean contexts for representing the operators or, if, and iff;
- the option of allowing concept nodes to be placed in the arc sequence of conceptual relations;
- the ability to import text into a text.

These extensions are designed to make sentences more concise, more readable, and more suitable as a target language for translations from natural languages and from other CL dialects, including CLIF. None of them, however, extend the expressive power of CGIF beyond the CG core, since the semantics of every extended feature is defined by its translation to core CGIF, whose semantics is defined by its translation to CL.

This section defines the concrete syntax of extended CGIF and the translation of each extended feature to core CGIF. This translation has the effect of specifying a function *CG*, which translates any sentence *s* of extended CGIF to a semantically equivalent sentence CG(*s*) of core CGIF. The combined functions *cg2cl*(CG(*s*)) translate *s* to a logically equivalent sentence in the CL abstract syntax.

The function CG and other functions for the other CGIF categories are defined by *transformation rules* whose notation is specified in clause B.1.3.1. Two categories, `comment` and `reference`, have identical syntax in core and extended CGIF; for any comment *cm* in extended CGIF, comment(*cm*)=*cm*; and for any reference *r* in extended CGIF, reference(*r*)=*r*. For any other category `X` of core CGIF, the strings of category X are a proper subset of the extended CGIF strings of the same category.

Since the definitions in Section B.2 specified the conceptual graph abstract syntax and its mapping to the abstract syntax of Common Logic, they used notation-independent constructs, such as sets. The definitions below specify the mapping from the concrete syntax of extended CGIF to the concrete syntax of core CGIF. Therefore, they are defined in terms of strings and functions that transform strings.

B.3.1 actor

Definition: A string *ac* that shall contain a comment *cm*, a reference *r* called the *type label*, an arc sequence s_1 called the *input arcs*, an arc sequence s_2 called the *output arcs*, and an optional end comment *ecm*. The output arcs s_2 shall not contain a sequence marker.

Translation: A conceptual graph *g*.

```
actor(?ac?) -> ?g?;
   actor  =  "(", [comment] ?cm?, (["#", "?"], CGname) ?r?,
             arcSequence ?s1?, "|", {arc} ?s2?, [endComment] ?ecm?, ")";
   ?z1?   =  first(arcSequence(?s1?));
   ?z2?   =  first(arcSequence(?s2?));
   ?sqn?  =  third(arcsequence(?s1?));
   if (length(?s2?)="0")
      ?cr? = "(", ?cm?, ?r?, ?z1?, ?sqn?, ?ecm?, ";0-output actor", ")";
   elif (length(?s2?)="1")
      ?cr? = "(", ?cm?, ?r?, ?z1?, ?sqn?, "|", ?z2?, ?ecm?, ")";
   else ?cr? = "(", ?cm?, ?r?, ?z1?, ?sqn?, "/*|*/", ?z2?, ?ecm?, ")";
   end;
   ?g?  =  second(arcSequence(?s1?)), second(arcSequence(?s2?)), ?cr?;
   end;
```

If *s2* has no output arcs, *cr* shall be an ordinary conceptual relation, as defined in Section B.3.7; but to show that *cr* was derived from an actor, an end comment "0-output actor" is inserted. If *s2* has one output arc, *cr* shall be an actor, but *cr* differs from *ac* because the arcs are translated to core CGIF. If *s2* has two or more output arcs, *cr* shall be an ordinary conceptual relation, but the comment `"/*|*/"` is inserted to distinguish the input arcs from the output arcs. The final rewrite rule puts *cr* after any conceptual graphs derived from the arc sequences.

Comment: As an example, the combined effect of the transformation rules for actors, arcs, arc sequences, and concepts would translate the following actor node

```
(IntegerDivide [Integer: *x] [Integer: 7] | *u *v)
```

to a six-node conceptual graph consisting of three concepts and three conceptual relations:

```
[*x] (Integer ?x) (Integer 7) [*u] [*v]
(IntegerDivide ?x 7 /*|*/ ?u ?v)
```

The comment `/*|*/` has no semantic effect in core CGIF or CL, but if preserved, it would enable a mapping back to extended CGIF to distinguish the input arcs from the output arcs. If the distinction is important for some application, axioms may be used to state the functional dependencies of the outputs on the inputs. For example, the CL relation that results from the translation of an actor of type `IntegerDivide` would satisfy the following constraint stated in CLIF:

```
(exists (Quotient Remainder) (forall (x1 x2 x3 x4)
   (iff (IntegerDivide x1 x2 x3 x4)
        (and (= x3 (Quotient x1 x2)) (= x4 (Remainder x1 x2))))))
```

This sentence asserts that there exist functions `Quotient` and `Remainder` that determine the values of the of the third and fourth arguments of the relation `IntegerDivide`. The translation rules would not generate that axiom automatically, but it could be stated by a CGIF sentence that would be translated to the CLIF sentence:

```
[*Quotient] [*Remainder]
[[@every*x1] [@every*x2] [@every*x3] [@every*x4]
[Equiv: [Iff: (IntegerDivide ?x1 ?x2 | ?x3 ?x4)]
        [Iff: (#?Quotient ?x1 ?x2 | ?x3) (#?Remainder ?x1 ?x2 | ?x4)]]]
```

To show that the existential quantifiers for `[*Quotient]` and `[*Remainder]` take precedence over the universal quantifiers for the four arguments, a pair of context brackets is used to enclose the concept nodes with universal quantifiers.

B.3.2 arc

Definition: A string *ar* that shall contain an optional comment *cm* and either a reference *r*, a defining label with CG name *n*, or a concept *c*.

Translation: A pair (*x*,*g*) consisting of a an arc *x* and a conceptual graph *g*.

```
arc(?ar?) -> ?x?, ?g?;
arc = [comment] ?cm?, (reference ?r? | "*", CGname ?n? | concept ?c?);
if   (?r?~= )  ?x?  =  ?ar;  ?g?  =  ;
elif (?n?~= )  ?x?  =  ?cm?, "?", ?n?;  ?g?  =  "[*", ?n?, "]";
else           ?x?  =  ?cm?, first(concept(?c?));
               ?g?  =  third(concept(?c?));
end; end;
```

If *ar* is a reference, *x* shall be *ar* unchanged, and *g* shall be blank. If *ar* contains a defining label, *x* shall be the result of replacing the marker `*` in *ar* with `?`, and *g* shall be the concept `[*n]`. If *ar* contains a concept *c*, *x* shall be the result of replacing the concept *c* in *ar* with a reference *r*, and *g* shall be third(concept(*c*)).

Comment: As an example, if the arc *ar* is `[Integer]`, the value of concept(`[Integer]`) would be a CG name, such as `g00023`, and arc(`[Integer]`) would be the pair consisting of the reference `?g00023` and the conceptual graph `[*g00023] (Integer ?g00023).`

B.3.3 arcSequence

Definition: A string *as* that shall contain a sequence *s* of zero or more arcs followed by an optional sequence marker *sqn*.

Translation: A triple (*rs,g,sqn*) consisting of a sequence of references *rs*, a conceptual graph *g*, and the sequence marker *sqn*.

```
arcSequence(?as?) -> ?rs?, ?g?, ?sqn?;
arcSequence  =  {arc} ?s?, [[comment], "?", seqmark] ?sqn?;
?rs? = map(first,map(arc,?s?));
?g? = map(second,map(arc,?s?));
end;
```

Comment: The function `map(arc,?s?)` applies arc to each arc of *s* to generate a sequence of pairs consisting of a reference and a concept. Then `map(first,map(arc,?s?))` extracts the sequence of references from the first element of each pair. Finally, `map(second,map(arc,?s?))` extracts the sequence of concepts from the second element of each pair. The option of having a sequence marker in an arc sequence implies that a conceptual relation may have a variable number of arcs. An actor may have a variable number of input arcs, but the number of output arcs shall be fixed; therefore, the output arcs shall not have a sequence marker.

B.3.4 boolean

Definition: A string *b* that shall contain a context *bc*, which shall not directly contain a reference or a defining label. The context *bc* shall have either a prefix `"~"` and no type label or no prefix and one of the following constants as type label: `Either, Equiv, Equivalence, If, Iff, Then.`

Translation: A negation *ng* that shall be negation(*b*), eitherOr(*b*), ifThen(*b*), or equiv(*b*).

```
boolean  =  negation | eitherOr | ifThen | equiv;

negation(?b?) -> ?ng?;
negation  =  "~[", [comment] ?cm?, CG ?x?, [endComment] ?ecm?, "]";
?ng?  =  "~[", ?cm?, CG(?x?), ?ecm?, "]";
end;

ifThen(?b?) -> ?ng?;
ifThen  =  "[", [comment] ?cm1?, "If", [":"], CG ?ante?,
             "[", [comment] ?cm2?, "Then", [":"], CG ?conse?,
                [endComment] ?ecm1?, "]", [endComment] ?ecm2?, "]";
?ng?    =  "~[", ?cm1?, CG(?ante?),
             "~[", ?cm2?, CG(?conse?), ?ecm1?, "]", ?ecm2?, "]";
end;

equiv(?b?) -> ?ng?;
equiv = "[", [comment] ?cm1?, ("Equiv" | "Equivalence"), [":"],
        "[", [comment] ?cm2?, "Iff", [":"], CG ?g1?,
```

```
                [endComment] ?ecm2? "]",
          "[", [comment] ?cm3?, "Iff", [":"], CG ?g2?,
                [endComment] ?ecm3? "]", [endComment] ?ecm1? "]";
   ?ng? = "[", ?cm1?, "~[", ?cm2?, CG(?g1?),
                       "~[", CG(?g2?), "]", ?ecm2?, "]",
                ?cm2?, "~[", ?cm3?, CG(?g2?),
                       "~[", CG(?g1?), "]", ?ecm3?, "]", ?ecm1?, "]";
   end;

   eitherOr(?b?) -> ?ng?;
   eitherOr = "[", [comment] ?cm?, "Either", [":"],
                   {[comment], nestedOrs} ?ors?, [endComment] ?ecm?, "]";
   ?ng?  =  "~[", ?cm?, nestedOrs(?ors?), ?ecm?, "]";
   end;

   nestedOrs(?ors?) -> ?g?;
   nestedOrs  = ( "[", [comment] ?cm?, "Or" ?first?, [":"], CG ?ng?,
                      [endComment] ?ecm?, "]", nestedOrs ?more?
                | );
   if (?first?= )  ?g? = ;
   else ?g? = "~[", ?cm?, CG(?ng?), ?ecm?, "]", nestedOrs(?more?);
   end; end;
```

The rule for nestedOrs recursively processes a sequence of zero or more boolean contexts of type `Or`. If *b* contains zero nested Ors, `eitherOr(b)` shall be `~[ ]`, which is false; the corresponding CLIF sentence (or) is defined to be false.

Comment: The scope of quantifiers in any of the Boolean contexts shall be determined by the nesting of their translations to core CGIF. Any defining label in a context of type `If` shall have the nested context of type `Then` within its scope. For any two contexts directly contained in a context of type `Either`, `Equivalence`, or `Equiv`, neither one shall have the other within its scope.

B.3.5 concept

Definition: A string *c* consisting of four substrings, any or all of which may be omitted: an opening comment *cm*, a type field, a referent field, and an end comment *ecm*.

The referent field of *c* may contain a defining sequence label with sequence marker *sqn*. If so, the type field of *c* shall be empty, the defining sequence label may be preceded by "@every", and there shall not be any references or any conceptual graph in the referent field of *c*.

If no *sqn*, the type field of *c* shall contain either a type expression *tx* and a colon ":" or an optional reference *ty* called a *type label* and an optional colon ":". If no *sqn*, the referent field of *c* shall contain an optional defining label with CG name *df* (which may be preceded by "@every"), a sequence of zero or more references *rf*, and a conceptual graph *g*, which may be blank. If all the options are omitted, the concept *c* shall be the string `"[]"`.

Translation: A triple (*r,q,g*) consisting of a reference or a bound sequence label *r*, a quantifier *q*, which shall be "@every" or the empty string, and a conceptual graph *g*, which shall contain at least one concept.

```
concept = "[", [comment] ?cm?,
          ( (typeExpression ?tx?, ":"
            | [["#" , "?"], CGname] ?ty?, [":"]),
            [["@every"] ?q?, "*", CGname ?df?], {reference} ?rf?, CG ?x?
          | ["@every"] ?q?, "*", seqmark ?sqn?
          ), [endComment] ?ecm?, "]";
   if (?sqn?~= )  ?r? = "?", ?sqn?; ?g1? = "[", ?cm?, "*", ?sqn?, ?ecm?];
   elif (?df?~= ) ?r? = "?", ?df?; ?g1? = "[", ?cm?, "*", ?df?, ?ecm?];
                  if (?rf?~= ) ?g2? = "[", ":", ?r?, ?rf?, "]"; end;
```

```
elif (?rf?~= ) ?r? = first(?rf?);
               ?g2? = "[", ?cm?, ":", ?rf?, ?ecm?, "]";
else           ?df? = gensym(); ?r? = "?", ?df?;
               ?g1? = "[", ?cm?, "*", ?df?, ?ecm?, "]";
end;
if (?tx?~= )   ?b?  = first(typeExpression(?tx?));
               ?gx? = second(typeExpression(?tx?));
               ?g3? = substitute(?r?,?b?,?gx?);
elif (?ty?~= ) ?g3? = "(", ?ty?, ?r?, ")"; end;
if (?x?~= )  ?g4? = "[", CG(?x?), "]";
end;
?g? = ?g1?, ?g2?, ?g3?, ?g4?;
end;
```

Four options are permitted in the type field: a type expression *tx*, a bound coreference label prefixed with "#", a constant, or the empty string; a colon is required after *tx*, but optional after the other three. The rewrite rules move features from the concept *c* to four strings, which are concatenated to form the conceptual graph *g*: *g1* is an existential concept with the defining label from *c* or with a label generated by gensym() if no defining label or reference occurs in *c*; *g2* is a coreference concept if any references occur in *c*; *g3* is either a conceptual relation with a type label *ty* or a conceptual graph generated from a type expression *tx*; and *g4* is a context containing any nonblank CG *x*. Any comments *cm* and *ecm* are placed in the first nonblank concept, which shall be either *g1* or *g2*.

Comment: To illustrate the translation, the sentence *A pet cat Yojo is on a mat* could be represented in extended CGIF with two concept nodes in the arc sequence of a conceptual relation:

```
(On [@*x (Pet ?x) (Cat ?x): Yojo] [Mat])
```

To generate the equivalent core CGIF, the concepts are removed from the arc sequence. In their place, references are left to link them to the concepts, which are expanded by the above rewrite rules. Following is the resulting core CGIF:

```
[: Yojo] (Pet Yojo) (Cat Yojo)
[*g00238] (Mat ?g00238) (On Yojo ?g00238)
```

The CG name `Yojo` is the reference for the first concept, and the CG name `g00238` for the mat is generated by gensym(). See Section B.3.9 for a discussion of the type expression and its translation. The translation by *cg2cl* would translate the core CGIF to the abstract syntax, which would be expressed by the following CLIF:

```
(exists (g00238) (and (= Yojo Yojo) (Pet Yojo) (Cat Yojo)
        (Mat ?g00238) (On Yojo ?g00238)))
```

A coreference concept with only one reference, such as `[: Yojo]`, has no effect on the truth or falsity of the sentence. It could be deleted by an optimizing compiler, unless it is needed as a container for comments.

B.3.6 conceptual graph (CG)

Definition: A string *cg* consisting of an unordered sequence of substrings that represent concepts, conceptual relations, booleans, and comments.

Translation: A conceptual graph *g*.

```
CG(?cg?) -> ?g?;
CG  =  {concept | conceptualRelation | boolean | comment};
if (first(sortCG(?cg?)~= )
      ?g?  =  "~", "[", first(sortCG(?cg?)),
              "~", "[", second(sortCG(?cg?), "]", "]";
else  ?g?  =  second(sortCG(?cg?));
end; end;
```

sortCG(*cg*) shall be the pair (*g1*,*g2*), where *g1* is the conceptual graph derived from all the universally quantified concepts in *cg* and *g2* is the conceptual graph derived from all other concepts, conceptual relations, and comments in *cg*.

```
sortCG(?cg?) -> ?g1?,?g2?;
sortCG = ( (concept ?c? | conceptualRelation ?x?
          | boolean ?x? | comment ?x?), sortCG ?rem?
         | );
if (?c?= ) ?cg2? = CG(?x?));
elif (second(concept(?c?)) = "@every")
          ?cg1? = third(concept(?c?));
else      ?cg2? = third(concept(?c?));
end;
?g1? = ?cg1?, first(sortCG(?rem?)); ?g2? = ?cg2, second(sortCG(?rem?));
end;
```

Comment: If there are no concepts containing universal quantifiers in the input string, the result shall be a single string in core CGIF that concatenates the results of translating each node independently of any other node. But if the input string contains any universal concepts, the output string shall be a nest of two negations. The outer context shall contain the translations of all the universal concepts, and the inner context shall contain the translations of all other nodes in the input.

B.3.7 conceptual relation

Definition: A string *cr* that represents an ordinary conceptual relation or an actor.

Translation: A conceptual graph *g*, which shall be either ordinaryRelation(*cr*) or actor (*cr*).

```
conceptualRelation = ordinaryRelation | actor;

ordinaryRelation(?cr?) -> ?g?;
   ordinaryRelation  =  "(", [comment] ?cm?, (["#", "?"], CGname) ?r?,
                        arcSequence ?s?, [endComment] ?ecm?, ")";
   ?g?  =  second(arcSequence(?s?)),
           "(", ?cm?, ?r?, first(arcSequence(?s?)),
           third(arcSequence(?s?)), ?ecm?, ")";
   end;
```

The first line of the rewrite rule extracts a conceptual graph from the arc sequence *s*. The second line adds the opening comment, type label, and arc sequence of a conceptual relation. The third line adds the sequence marker, if any, the end comment, and the closing parenthesis of the conceptual relation.

Comment: As an example, the conceptual relation `(On [Cat: Yojo] [Mat])` would be translated by the rules for conceptual relations, arcs, arc sequences, and concepts to generate a conceptual graph expressed in core CGIF, such as the following:

```
[: Yojo] (Cat Yojo) [*g00719] (Mat ?g00719) (On Yojo ?g00719)
```

B.3.8 text

Definition: A context *c* that is not contained directly or indirectly in any context.

Translation: A context *cx*.

```
text(?c?) -> ?cx?;
text = "[", [comment] ?cm?, "Proposition", ":", [CGname] ?n?,
       CG ?g?, [endComment] ?ecm?, "]";
?cx? = "[", ?cm?, "Proposition", ":", ?n?, CG(?g?), ?ecm?, "]";
end;
```

Comment: CGIF does not provide an explicit syntax for modules. Instead, any CL module shall first be translated to a text in core CLIF according to the specification in Table A.2 of section 0. Then the result of that translation shall be translated to a text in extended CGIF according to the function *cl2cg*, which is defined in section B.4.

B.3.9 type expression

Definition: A string *tx* containing a CG name *n* and a conceptual graph *g*.

Translation: A pair (*b,g*), consisting of a bound label *b* and a conceptual graph *g*.

```
typeExpression(?tx?) -> ?b?,?g?;
typeExpression  =  "@", "*", CGname ?n?, CG ?g?;
?b?  =  "?", ?n?;
end;
```

If a concept *c* contains a type expression, the rewrite rules that specify concept(*c*) use the function `substitute(?r?,?b?,?g?)` to substitute some reference *r* for every occurrence of *b* in *g*.

Comment: A type expression corresponds to a lambda expression in which the CG name *n* specifies the formal parameter, and the conceptual graph *g* is the body of the expression. If a concept *c* contains a type expression, the transformation rules that process *c* shall substitute a reference derived from *c* for every occurrence of the bound label `?`*n* that occurs in *g*.

B.4 CGIF conformance

This annex has specified the syntax of three CL dialects, an abstract syntax for conceptual graphs, a concrete syntax for core CGIF, and a concrete syntax for extended CGIF. All three of these languages are fully conformant CL dialects in the sense that every CL sentence can be translated to a semantically equivalent sentence in each of them, and every sentence in any of these three dialects can be translated to a semantically equivalent sentence in CL. The semantic equivalence is established by definition: the semantics of every sentence in extended CGIF is defined by a translation to a sentence in core CGIF, the semantics of every sentence in core CGIF is defined by a translation to a sentence in the abstract CG syntax, and the semantics of every abstract CG sentence is defined by its translation to the abstract syntax of CL.

To demonstrate full conformance, this clause specifies the function *cl2cg*, which shall translate any sentence s in CL to a sentence *cl2cg*(*s*) in extended CGIF, which shall have the same truth value as *s* under every interpretation for CL. For most CL expressions, the mapping to some expression in extended CGIF is straightforward. The translation of functional terms from CL to CGIF, however, requires more than one step. Any CL function application can be translated to an actor that represents the function plus a reference to some concept whose referent is the value of that function. In order to translate a sequence of CL terms to an arc sequence in extended CGIF, the actor node shall be enclosed inside the concept node.

As an example, let (F X1 X2) be a CLIF term with an operator F applied to arguments X1 and X2, where the names X1 and X2 are bound by quantifiers, but F is not. When that term is translated by *cl2cg*, the gensym() function shall be used to generate a CG name, such as g00592. When prefixed with "?", that name becomes a bound coreference label, which shall be used as the output arc of an actor that represents the function F. The result of translating the original CLIF term by *cl2cg* shall be (F ?X1 ?X2 | ?g00592). The defining label *g00592 shall be placed in a concept, such as [*g00592], and the actor shall be placed inside that concept as a nested conceptual graph: [*g00592 (F X1 X2 | ?g00592)]. This concept shall be the result of *cl2cg* when applied to the functional term. It may appear as an arc in an arc sequence of some actor or conceptual relation.

Since the predicate of a CL relation or the operator of a CL function may be a functional term, the same transformation shall be used to translate the predicate or the operator to a concept. As an example, let ((F X1 X2) Y1 Y2) be a CLIF atomic sentence whose predicate is the same functional term that appeared in the previous example. Therefore, the bound label "?g00592", which represents the value of the function, shall be the type label of the corresponding conceptual relation. If both Y1 and Y2 are bound by quantifiers in CL, the conceptual relation shall be (#?g00592 ?Y1 ?Y2). In order to generate a single syntactic unit as the value of

cl2cg, this conceptual relation shall be placed inside the concept that represents the functional term, immediately before "]": [*g00592 (F X1 X2 | ?g00592) (#?g00592 ?Y1 ?Y2)]. This concept shall be the result of *cl2cg* when applied to the original atomic sentence. It may appear as a node of a conceptual graph that results from the translation of a larger CL sentence that contains the original atomic sentence.

For every CL expression **E**, Table B.1 specifies the extended CGIF expression that defines *cl2cg*(**E**). In order to ensure that the CL constraints on quantifier scope are preserved in the translations by *cl2cg*, context brackets, "[" and "]", are used to enclose the translations for expressions of type E13 and E14. In some cases, these brackets are unnecessary, and they may be ignored.

The first column of Table B.1 indicates links to rows in the CL semantics in Section 6. The second column uses the metalanguage and conventions used to define the CL abstract syntax. The third column mixes that metalanguage with the notation used for rewrite rules in clause B.1.3.2. That combination defines a function *cg2cl*, which translates any sentence *s* of core CGIF to a logically equivalent sentence *cg2cl*(*x*) of Common Logic.

Table B.1 — Mapping from CL abstract syntax to extended CGIF syntax

	If E is a CL expression of the form	**Then *cl2cg*(E) =**
E1	A numeral 'n'	The numeral 'n'
E1	A quoted string ‘s’	The quoted string 's'
E1	A interpretable name 'n'	The name 'n' shall be enclosed in quotes if it is not a CG identifier. If it occurs in the quantifier of some CL sentence, it shall be prefixed with "*". If it is bound by a quantifier, it shall be prefixed with "?".
E2	Sequence marker *S*	*S*
E3	A term sequence <T1 … Tn> starting with a term T1	An arc sequence: *cl2cg*(T1) … *cl2cg*(Tn)
E4	A term sequence T1 … Tn starting with a sequence marker T1	An arc sequence: *cl2cg*(T2), ..., *cl2cg*(Tn), *cl2cg*(T1)
E5	A term (O T1 … Tn)	A concept with a generated name 'n' that contains a nested actor: "[", "*", 'n', "(", *cl2cg*(O), *cl2cg*(T1, … Tn), ")", "|", "?", 'n', ")", "]"
	A term (cl:comment ‘string’ T)	An arc with a comment: "/*", 'string', "*/", *cg2cl*(T)
E6	An equation (= T1 T2)	A CG consisting of one, two, or three concepts. If both T1 and T1 are names, one concept: "[:", *cl2cg*(T1), *cl2cg*(T2), "]" If both T1 and T2 are functional terms, three concepts: *cg2cl*(T1), *cg2cl*(T2), "[", "?", 'n1', "?", 'n2', "]" where 'n1' is the name generated for T1 and 'n2' is the name generated for T2. If Ti is a functional term (where i=1 or i=2) and the other term Tj is a name, two concepts: *cl2cg*(Ti), "[", "?", 'ni', *cl2cg*(Tj), "]" where 'ni' is the name generated for Ti.

E7	An atomic sentence (P T1 … Tn)	A CG consisting of either a conceptual relation or a concept. If P is a name, a conceptual relation: "(", *cl2cg*(P), *cl2cg*(T1 ...Tn), ")" If P is a functional term, a concept: *cl2cg*(P) as modified by inserting the following conceptual relation immediately before the closing "]": "(", 'n', *cl2cg*(T1 ... Tn), ")" where 'n' is the name generated for *cl2cg*(P).
E8	A boolean sentence (not P)	A negation: "~", "[", *cl2cg*(P), "]"
E9	A boolean sentence (and P1 … Pn)	A CG: *cl2cg*(P1), ..., *cl2cg*(Pn)
E10	A boolean sentence (or P1 … Pn)	A CG: "[", "Either", "[", "Or", *cl2cg*(P1), "]", ..., "[", *cl2cg*(Pn), "]", "]"
E11	A boolean sentence (if P Q)	A CG: "[", "If", *cl2cg*(P), "[", "Then", *cl2cg*(Q), "]", "]"
E12	A boolean sentence (iff P Q)	A CG: "[", "Equiv", ":", "[", "Iff", *cl2cg*(P), "]", "[", "Iff", *cl2cg*(Q), "]", "]"
	A sentence (cl:comment 'string' P)	A comment and a CG: "/*", 'string', "*/", *cl2cg*(P)
E13	A quantified sentence (forall (N1 … Nn) B) where N1 through Nn are names or sequence markers	A CG: "[", "[", "@every", "*", *cl2cg*(N1), "]", ..., "[", "@every", "*", *cl2cg*(Nn), "]", *cl2cg*(B), "]"
E14	A quantified sentence (exists (N1 … Nn) B) where N1 through Nn are names or sequence markers	A CG: "[", "[", "*", *cl2cg*(N1), "]", ..., "[", "*", *cl2cg*(Nn), "]", *cl2cg*(B), "]"
	A phrase (cl:comment "string")	A comment: "/*", 'string', "*/"
E17	A phrase (cl:imports N)	A concept: "[", "cg_Imports", *cl2cg*(N), "]"
E18	A module with name N, exclusion list N1 ... Nn, and text T	If M is the translation to core CL specified in Table A.2 of Section A.3, then a text: "[", "Proposition", ":", *cl2cg*(M), "]"
E19	A phrase (cl:text T1 … Tn)	A text: "[", "Proposition", *cl2cg*(T1 ... Tn), "]"
E20	(cl:text N T1 …Tn)	A text: "[", "Proposition", ":", *cl2cg*(N), *cg2cl*(T1 ... Tn), "]"

To specify the translation from extended CGIF to core CGIF, Section B.3 uses a combination of EBNF syntax rules plus the rewrite rules specified in clause B.1.3.2 to define a function ex2cor, which translates any sentence *s* of extended CGIF to a logically equivalent sentence CG(*s*) of core CGIF.

This completes the description of CGIF semantics for the purposes of this annex and conformance.

Annex C

(normative)

eXtended Common Logic Markup Language (XCL)

C.1 Introduction

XCL is an XML notation for Common Logic. It is the intended interchange language for communicating Common Logic across a network. It is a straightforward mapping of the CL abstract syntax and semantics into an XML form.

C.2 XCL Syntax

Since XCL's lexical syntax is the same as XML itself, the syntax of XCL is described by a Document Type Definition (DTD), which is usually accessed in electronic form. For completeness and standardization purposes, the DTD is provided here in its entirety.

```
<!-- ..................................................................... -->
<!-- XML Common Logic 1.0 DTD  ............................................ -->
<!-- file: xcl1.dtd
-->
```



```
<!--  XML Common Logic 1.0 DTD
```

This is XCL, a formulation of Common Logic as an XML application.

Authors: Murray M. Altheim <m.altheim@open.ac.uk>
Pat Hayes <phayes@ihmc.us>

Status: Draft

Revision: $Id: xcl1c.dtd,v 1.8 2005/12/05 23:11:15 altheim Exp $

This DTD has the following formal public identifiers:

```
"ISO/IEC 24707:2006//DTD XML Common Logic (XCL) 1.0//EN"

"-//purl.org/xcl//DTD XML Common Logic (XCL) 1.0//EN"
```

The DTD may be invoked by one of the following declarations:

```
<!DOCTYPE text PUBLIC
    "ISO/IEC 24707:2006//DTD XCL Markup Language//EN">
    "xcl1.dtd">

<!DOCTYPE text PUBLIC
    "-//purl.org/xcl//DTD XML Common Logic (XCL) 1.0//EN"
```

```
                "xcl1.dtd">

        where the system identifier ("xcl1.dtd") may be customized as
        necessary to specify the location of the DTD.

        If there is any perceived difference between the prose of the ISO
        standard and the XCL DTD, the former should be considered
        authoritive.
-->

<!-- Comments in the DTD

        The comments in this DTD which use the expressions "must", "shall" or
        "shall not" are normative requirements of this International Standard.
        Comments which use the expression "should" or "should not" are
        recommendations of this International Standard. Comments which use
        the verbs "recommend" or "deprecate" are recommendations and
        deprecations of this International Standard.
-->

<!-- a Uniform Resource Identifier, see clause 3.27 of this International Standard and [8]
-->
<!ENTITY % URI.datatype "CDATA" >

<!-- XML namespace support ..................................... -->

<!-- The XML Namespace URI for XCL 1.0 is

       "http://purl.org/xcl/1.0/"
-->

<!ENTITY XCL1.xmlns "http://purl.org/xcl/1.0/" >

<!-- 1. General Syntax ..................................................... -->

<!-- 1.1 Content Models ..................... -->

<!ENTITY % Quantified.class
    "( quantified | forall | exists )"
>

<!ENTITY % Boolean.class
    "( boolean | and | or | implies | iff | not )"
>

<!ENTITY % Atomic.class
    "( atomic | relation | equal )"
>

<!ENTITY % Sentence.class
   "( %Quantified.class; | %Boolean.class; | %Atomic.class; )"
>

<!ENTITY % Comment.class
   "comment"
>

<!-- 1.2 Attributes ............................... -->

<!-- 1.2.1 Common Attributes

        The following attributes are declared on all XCL element types (though
        are not included in the descriptive text within the notes).
```

xmlns (optional) All XCL elements have a declared, optional 'xmlns' attribute whose fixed, default value matches the XML Namespace for XCL 1.0. XML processors may imply this attribute when not explicitly present in the document instance.

id (optional) All XCL elements have a declared, optional 'id' attribute whose value must match XML Name (production 5 of [XML]). When present, the ID value serves as the means of uniquely identifying a specific element within an XCL document. Note that this operates at the XML syntax level and has no semantic significance within CL. Each 'id' value must be unique within an XCL document.

```
-->

<!ENTITY % XCL.xmlns.attrib
    "xmlns          %URI.datatype;      #FIXED '&XCL1.xmlns;'"
>

<!ENTITY % id.attrib
    "id             ID                  #IMPLIED"
>

<!ENTITY % Common.attrib
    "%XCL.xmlns.attrib;
     %id.attrib;"
>

<!-- 1.2.2 CL Dialect Attribute ..... -->

<!--  Name:         dialect
      URI:          http://purl.org/xcl/1.0/#dialect
      Declares:     http://purl.org/xcl/1.0/#dialect-xcl
                    http://purl.org/xcl/1.0/#dialect-clif
                    http://purl.org/xcl/1.0/#dialect-cgif
      Label:        CL Dialect
      Description:  an identifier for the CL dialect of the
                    element's content; see clauses 3.8 and 7.1 of this International
Standard
-->
<!-- Notes:
```

The 'dialect' attribute is used to indicate the dialect of its element's content. 'dialect' is a linking attribute whose value (a URI reference) contains a reference to one of the fixed set of CL dialect identifiers:

```
http://purl.org/xcl/1.0/#dialect-xcl
http://purl.org/xcl/1.0/#dialect-clif
http://purl.org/xcl/1.0/#dialect-cgif
```

For other concrete syntax representations, a suitable URI indicating the dialect should be used. In all XCL elements for which the 'dialect' attribute is declared, its absence indicates the default: the XCL dialect defined by this DTD.

Note that the presence of a 'dialect' attribute overrides any 'dialect' attributes on parent elements; however, such parent-child dialect clashes are deprecated.

This attribute is declared on the <text>, <module>, <import>, and <phrase> elements.

Example:

```
            <text dialect="http://purl.org/xcl/1.0/#dialect-clif">
              (forall ex:romanceNovel ((x man)) (exists ((y woman))
              (and (loves x y) (not (loves y x))) ))
            </text>
-->

<!ENTITY XCL.dialect  "http://purl.org/xcl/1.0/#dialect-xcl"  >
<!ENTITY CLIF.dialect "http://purl.org/xcl/1.0/#dialect-clif" >
<!ENTITY CGIF.dialect "http://purl.org/xcl/1.0/#dialect-cgif" >

<!ENTITY % dialect.attrib
  "dialect         %URI.datatype;      '&XCL.dialect;'"
>


<!-- 1.3 Comments ..................................... -->

<!--  Name:         comment
      URI:          http://purl.org/xcl/1.0/#comment
      Label:        Comments
      Description:  Inserts a comment. <comment> elements can
                    be included within any XCL element and are
                    considered as comments on their immediate
                    parent element; see clause 6.1.1.3 of this International
Standard
-->
<!-- Notes:
```

When well-formed XML processing is acceptable (see the section on XCL conformance), `<comment>` elements can comprise any text, can be mixed content, and can have any user-defined attributes; they are ignored by logical processors, but conforming XCL applications are required to preserve them and their position relative to other elements. Comments inside other comments are considered to be comments on the comment. In most cases, XCL content models include comments as the last children of the parent element.

Note that XCL markup inside a comment is not considered to be part of the XCL containing element, and must also be suitably escaped.

For situations where rich comment markup is desired but valid XCL is required, comments may contain a link to an external documentation source using the '`href`' attribute:

```
<comment href="http://www.acme.com/docs/sec7.html"/>
```

If both element content and the '`href`' attribute are present, the latter is considered optional, i.e., traversing the link is not considered essential to ascertain the contents of the comment.

With appropriate XML Namespace declarations, the `%Comment.class;` parameter entity can be redeclared to contain alternative XML content, e.g., XHTML or DocBook.

```
            <!ENTITY % Comment.class
                "( xhtml:div | comment )"
            >
-->
```

```
<!ENTITY % Comment.content
   "( #PCDATA | %Comment.class; )*"
>
<!ELEMENT comment  %Comment.content; >
<!ATTLIST comment
   %Common.attrib;
   href            %URI.datatype;      #IMPLIED
>

<!-- 2. Top Level Elements ..................................................... -->

<!-- 2.1 XCL Document Element ....................... -->

<!--  Name:          text
      URI:           http://purl.org/xcl/1.0/#text
      Label:         XCL document element
      Description:   Used to surround any piece of XCL content, as the
                     delimiters of an XCL (i.e., XML) document. Text inside
                     this element must be valid XCL. It need not be a module
                     (ontology) See the XCL Conformance section for details
                     on well-formedness constraints. See clause 6.1.1.1 of this International
Standard.
-->

<!-- Notes:

     Attributes:

        xml:base (optional) Indicates the document base URI.

        dialect  (optional) see description in 1.2.2 of this section. When not
                 explicitly specified, this attribute defaults to the value indicating
                 the XCL 1.0 (XML) syntax.

        href (optional) Used to assign an "importing name" to a text. This is a
                 URI reference or IRI, and often it will be the same as xmlns
                 default namespace and/or the URL of the containing
                 document. However, this coincidence of naming is not
                 required. No logical relationship is assumed between names
                 based on their URI or XML namespace structure, so it is
                 acceptable to use a URI reference containing a fragment ID to
                 name a text.


     Children:

        Zero or more <module>, <phrase>, and/or <comment> elements in
        any order.

     Example:

          <text dialect="http://purl.org/xcl/1.0/#dialect-xcl">
            <phrase>
              ...
            </phrase>
          </text>
-->

<!ENTITY % Text.content
   "( module | phrase | %Comment.class; )*"
>
```