ISO/IEC JTC 1/SC 32

Data Management and Interchange

Secretariat: United States of America (ANSI)
Administered by Farance, Inc. on behalf of ANSI

<table>
<thead>
<tr>
<th>DOCUMENT TYPE</th>
<th>Working Draft Text (for information or comment)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TITLE</td>
<td>ISO/IEC WD 24707 Information technology -- Common Logic (CL) – A Framework for a Family of Logic-Based Languages</td>
</tr>
<tr>
<td>SOURCE</td>
<td>Harry Delugach (editor)</td>
</tr>
<tr>
<td>PROJECT NUMBER</td>
<td>1.32.25.01.00.00</td>
</tr>
<tr>
<td>STATUS</td>
<td>For review</td>
</tr>
<tr>
<td>REFERENCES</td>
<td></td>
</tr>
<tr>
<td>ACTION ID.</td>
<td>FYI</td>
</tr>
<tr>
<td>REQUESTED ACTION</td>
<td></td>
</tr>
<tr>
<td>DUE DATE</td>
<td></td>
</tr>
<tr>
<td>Number of Pages</td>
<td>54</td>
</tr>
<tr>
<td>LANGUAGE USED</td>
<td>English</td>
</tr>
<tr>
<td>DISTRIBUTION</td>
<td>P &amp; L Members</td>
</tr>
<tr>
<td></td>
<td>SC Chair</td>
</tr>
<tr>
<td></td>
<td>WG Conveners and Secretaries</td>
</tr>
</tbody>
</table>

Douglas Mann, Secretariat, ISO/IEC JTC 1/SC 32
Farance, Inc *, 360 Pelissier Lake Road, Marquette, MI 49855-9678, United States of America
Telephone: +1 906-249-9275; E-mail: MannD@battelle.org
available from the JTC 1/SC 32 WebSite  http://staging.jtc1sc32.org/
*Farance, Inc. administers the ISO/IEC JTC 1/SC 32 Secretariat on behalf of ANSI
Information technology — Common Logic (Common Logic) – Framework for a family of logic-based languages

Editor's note: Need a French title

Warning

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

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

This ISO document is a working draft or committee draft and is copyright-protected by ISO. While the reproduction of working drafts or committee drafts in any form for use by participants in the ISO standards development process is permitted without prior permission from ISO, neither this document nor any extract from it may be reproduced, stored or transmitted in any form for any other purpose without prior written permission from ISO.

Requests for permission to reproduce this document for the purpose of selling it should be addressed as shown below or to ISO’s member body in the country of the requester:

[Indicate:
the full address
telephone number
fax number
telex number
and electronic mail address

as appropriate, of the Copyright Manager of the ISO member body responsible for the secretariat of the TC or SC within the framework of which the draft has been prepared]

Reproduction for sales purposes may be subject to royalty payments or a licensing agreement.

Violators may be prosecuted.
Contents

1 SCOPE .............................................................................. 1

2 NORMATIVE REFERENCES .................................................. 2

3 TERMS AND DEFINITIONS ................................................ 2

3.1 DEFINITIONS ........................................................................ 2
  3.1.1 conceptual graph .......................................................... 2
  3.1.2 Conceptual Graph Interchange Format .................................. 2
  3.1.3 interpretation ............................................................... 2
  3.1.4 Knowledge Interchange Format ........................................ 2
  3.1.5 sori ........................................................................... 2
  3.1.6 sorted logic ................................................................. 3
  3.1.7 traditional first order logic ............................................... 3
  3.1.8 universe of discourse universe ....................................... 3
  3.1.9 URI ........................................................................... 3
  3.1.10 XML-based Common Logic markup language .................... 3

3.2 ABBREVIATED TERMS .................................................... 3
  3.2.1 Conceptual Graphs ......................................................... 3
  3.2.2 CGIF: Conceptual Graph Interchange Format ....................... 3
  3.2.3 Knowledge Interchange Format ....................................... 3
  3.2.4 Traditional First Order Logic ........................................ 3
  3.2.5 Uniform Resource Identifier .......................................... 3
  3.2.6 XML ................................................................. 3
  3.2.7 XCL .............................................................. 3

4 SYMBOLS AND ABBREVIATIONS ....................................... 4

4.1 SYMBOLS ........................................................................... 4
  4.2 ABBREVIATIONS ............................................................. 4

5 INTRODUCTION AND RATIONALE ...................................... 4

5.1 REQUIREMENTS .................................................................... 4

5.2 DESIGN OVERVIEW ............................................................ 5
  5.2.1 A family of notations ................................................... 5
  5.2.2 Background discussion and motivation ................................ 6
  5.2.3 On being first-order ....................................................... 8

6 COMMON LOGIC ABSTRACT SYNTAX AND SEMANTICS ............ 9

6.1 COMMON LOGIC ABSTRACT SYNTAX ................................ 9

6.2 COMMON LOGIC SEMANTICS ............................................. 11

6.3 IMPORTING AND IDENTIFICATION ON A NETWORK ............... 13

6.3.1 Mixed networks .......................................................... 14

6.4 SATISFACTION, VALIDITY AND ENTAILMENT ...................... 15

6.5 SUMMARY OF THE CORE SYNTAX ..................................... 15

6.6 SEQUENCE VARIABLES, RECURSION AND ARGUMENT LISTS: DISCUSSION .......................................................... 16

6.7 SPECIAL CASES AND TRANSLATIONS BETWEEN DIALECTS .......... 17

6.7.1 Translating between dialects .......................................... 17

7 CONFORMANCE .................................................................... 18

7.1 DIALECT CONFORMANCE .................................................. 18
  7.1.1 Syntax ..................................................................... 18
  7.1.2 Semantics ............................................................... 18

7.2 APPLICATIONS .................................................................... 19
7.3 NETWORKS ................................................................. 19

ANNEX A (NORMATIVE) KNOWLEDGE INTERCHANGE FORMAT (KIF) – SYNTAX ...................... 21

A.1 INTRODUCTION ........................................................................... 21
A.2 COMMON LOGIC CORE SYNTAX .................................................. 21
A.2.1 Characters ............................................................................. 22
A.2.2 Lexical syntax ......................................................................... 22
A.2.3 Expression syntax .................................................................. 24

ANNEX B (NORMATIVE) CONCEPTUAL GRAPH INTERCHANGE FORMAT (CGIF) – SYNTAX .......... 27

B.1 GENERAL .................................................................................. 27
B.2 TERMS AND DEFINITIONS ......................................................... 27
B.2.1 Terms ...................................................................................... 27
B.3 CGIF CONCRETE SYNTAX ........................................................ 27
B.4 LEXICAL CATEGORIES ............................................................ 27
B.4.1 Comment ............................................................................... 27
B.4.2 DelimitedStr(D) ................................................................... 27
B.4.3 Exponent .............................................................................. 28
B.4.4 Floating ............................................................................... 28
B.4.5 Identifier ............................................................................... 28
B.4.6 Integer .................................................................................. 28
B.4.7 Name .................................................................................... 28
B.4.8 Number ................................................................................. 28
B.4.9 QuotedStr ............................................................................ 28
B.4.10 UnsignedInt ......................................................................... 28

B.5 SYNTACTIC CATEGORIES ......................................................... 28
B.5.1 Actor ...................................................................................... 29
B.5.2 Arc ......................................................................................... 29
B.5.3 BoundLabel .......................................................................... 29
B.5.4 CG ......................................................................................... 29
B.5.5 CGStream ............................................................................ 29
B.5.6 Concept ................................................................................. 29
B.5.7 Conjuncts .............................................................................. 30
B.5.8 CorefLinks ........................................................................... 30
B.5.9 DefLabel ............................................................................... 30
B.5.10 Descriptor ............................................................................ 30
B.5.11 Designator .......................................................................... 31
B.5.12 Disjuncts ............................................................................ 31
B.5.13 FormalParameter ................................................................. 31
B.5.14 Indexical ............................................................................ 31
B.5.15 IndividualMarker ................................................................. 31
B.5.16 LambdaExpression(N) ....................................................... 31
B.5.17 Literal ................................................................................. 31
B.5.18 Locator ............................................................................... 32
B.5.19 Negation ............................................................................. 32
B.5.20 Quantifier ........................................................................... 32
B.5.21 Referent ............................................................................... 32
B.5.22 Relation ............................................................................... 32
B.5.23 Signature ............................................................................ 32
B.5.24 SpecialConLabel ................................................................. 32
B.5.25 SpecialContext ................................................................. 32
B.5.26 Structure ............................................................................ 33
B.5.27 Type .................................................................................... 33
B.5.28 TypeExpression ................................................................. 33
B.5.29 TypeLabel(N) ................................................................. 33
B.5.30 TypeTerm ........................................................................... 33
B.5.31 Discussion ................................................................. 33

B.6 MAPPING ABSTRACT SYNTAX TO CGIF ............................................. 34
B.6.1 Conceptual Graph .................................................................. 34
B.6.2 Concept................................................................................. 35
B.6.3 Conceptual Relation .......................................................... 35
B.6.4 Lambda Expression .......................................................... 35
B.6.5 Concept Type .................................................................. 35
B.6.6 Relation Type .................................................................. 36
B.6.7 Referent ........................................................................... 37
B.6.8 Context ........................................................................... 37
B.6.9 Coreference Set ............................................................... 37
B.6.10 Module ........................................................................... 38

ANNEX C (NORMATIVE) CONCRETE SYNTAX – EXTENDED COMMON LOGIC MARKUP LANGUAGE (XCL) ................................................................. 39
  C.1 XCL AND DIALECTS .......................................................... 39
  C.2 COMMON LOGIC DIALECT TEXT IN XCL ......................... 39
  C.3 COMMON LOGIC SYNTAX IN XCL ....................................... 40
  C.4 <CLMODULE> .................................................................. 40
  C.4.1 Discussion ................................................................... 40
  C.4.2 Attributes ................................................................. 40

ANNEX D (INFORMATIVE) MAPPING OF KNOWLEDGE INTERCHANGE FORMAT (KIF) TO COMMON LOGIC ABSTRACT SYNTAX ................................................. 41

ANNEX E (INFORMATIVE) MAPPING OF CONCEPTUAL GRAPH INTERCHANGE FORMAT (CGIF) TO COMMON LOGIC ABSTRACT SYNTAX ................................................. 42

ANNEX F (INFORMATIVE) MAPPING OF XML-BASED COMMON LOGIC (XCL) TO COMMON LOGIC ABSTRACT SYNTAX ............................................................... 43

ANNEX G (INFORMATIVE) RELATIONSHIP OF COMMON LOGIC TO OTHER LOGIC STANDARDS AND OTHER LOGIC FORMALISMS ......................................................... 44
Foreword

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

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

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

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

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

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

1 Scope

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

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

- The language has declarative semantics. It is possible to understand the meaning of expressions in the language without appeal to an interpreter for manipulating those expressions.
- The language is logically comprehensive—at its most general, it provides for the expression of arbitrary logical sentences.

The following are within the scope of this standard:

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

The following are outside the scope of this standard:

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

This document describes Common Logic’s syntax and semantics.

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

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

EDITOR’s note: this section is incomplete

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

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

3 Terms and definitions

EDITOR’s note: this section is very incomplete And no longer accurate. I will provide a glossary of the terminology introduced in the text I've written, but not before next week. -Pat

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

3.1 Definitions

3.1.1 conceptual graph

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

3.1.2 Conceptual Graph Interchange Format

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

3.1.3 interpretation

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

3.1.4 Knowledge Interchange Format

A text-based first order formalism, using a LISP-like list notation, originating with the Knowledge Sharing Effort sponsored by the U.S. at ???. One of the three

3.1.5 sort

For the purposes of this standard, sort means a type which is an attribute of a symbol.
3.1.6 sorted logic

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

3.1.7 traditional first order logic

The traditional algebraic (or mathematical) formulations of logic as introduced by Russell, Peano, and Frege. This specifically excludes reasoning over relations and uses the same name as both an individual and a relation name.

3.1.8 universe of discourse

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

3.1.9 URI

3.1.10 XML-based Common Logic markup language

3.2 Abbreviated Terms

3.2.1 Conceptual Graphs

A logic-based semantic network representation system for first-order facts; one of the concrete syntaxes which instantiate the abstract syntax of Common Logic.

3.2.2 CGIF: Conceptual Graph Interchange Format

A LISP-based logic syntax for describing first-order sentences; one of the concrete syntaxes which instantiate the abstract syntax of Common Logic.

3.2.4 Traditional First Order Logic

The first order logic generally described by Russell, Whitehead, Peano and Peirce dealing with quantification, negation, logical relations as expressed in propositions which are strictly true or false.

3.2.5 Uniform Resource Identifier

A locator string, generally designed for use over a network, that uniquely identifies a resource (e.g., a document) that exists somewhere on the network.

3.2.6 XML

3.2.7 XCL

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

4.1 Symbols

\[ V \]: a vocabulary of a set of names and sequence variables

\[ U_I \]: the universe of individuals represented by some interpretation \( I \)

4.2 Abbreviations

- **KIF**: Knowledge Interchange Format
- **OWL**: Ontology Markup Language
- **RDF**: Resource Definition Framework
- **RDFS**: Resource Definition Framework Schema
- **TFOL**: Traditional First Order Logic
- **URI**: Uniform Resource Identifier
- **XML**: eXtendable Markup Language

5 Introduction and rationale

This section is informative.

5.1 Requirements

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

1. Common Logic is full first-order logic with equality.

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

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

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

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

• a. The XML syntax must be compatible with the published specifications for XML, URI syntax, XML Schema, Unicode and other conventions relevant to transmission of information on the Web.

• b. URI references can be used as logical names in the language

• c. URI references can be used to give names to expressions and sets of expressions, in order to facilitate Web operations such as retrieval, importation and cross-reference.

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

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

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

• c. No agent can 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 relate it to other universes.

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

5.2 Design Overview

5.2.1 A family of notations

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

One Common Logic dialect plays a special role. The XML syntax for Common Logic, here called XCL, is designed to serve three functions. It is a standard intercommunication notation for machine use on a network; it is the single 'reference syntax' into which all other dialects can be translated or embedded [requirement 2a]; and it provides forms which can be used to express various extensions and relations to existing standards [requirements 2b, 2c, 2d ] and to support existing conventions regarding URI usage [requirements 2c, 3c, 3d and 4e ]. XCL syntax is intended to be unambiguous, extendable, capable of expressing a wide variety of logical forms, and to provide ways to access logically meaningful subexpressions using conventional XML technology; it is not intended to be compact or human-readable. This document also independently defines a more conventional dialect, Common Logic core, which is intended to be compact, human-readable [requirement 2e] and more similar to conventional machine-oriented logics, notably to KIF (see Annex A) Common Logic core syntax is used in giving examples throughout this document. The Common Logic core can be considered an updating and simplification of KIF 3.0, 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.
5.2.2 Background discussion and motivation

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

First, A and B may have used different surface syntactic forms to express their knowledge. This is a well-known problem and various proposals have been made to solve it, usually by defining a standard syntax into which others can be translated, such as KIF [refKIF]. Common Logic approaches this by requiring all Common Logic dialects to be translatable into XCL. The advent and widespread adoption of XML technology has made this problem easier to solve, by making it possible to divorce the 'structured' specification of the language from any particular surface form. XCL allows the same Common Logic logical structure to be assigned a variety of surface forms, and for various surface forms - dialects - to be transmitted unchanged within XCL markup if required.

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

Consider the situation of two agents A and B communicating information to C, and suppose that there is some concept used by both A and B but with different signatures. For example, A might use a relation married with two arguments – the people who are married – while B might use it with three – two people and a time-interval – or as a function from people to time-intervals. Cases like this can be handled very simply merely by allowing a symbol to play more than one role. Common Logic simply allows relation and function symbols to be variadic, i.e. to take any number of arguments, and it allows a relation name to be used as a function name and vice versa. Thinking in traditional terms, this amounts to allowing a kind of 'punni function from people to time

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

It is also possible that A uses a name such as \textit{married} as an individual name while B uses it as a relation name. For example, A might have an ontology of interpersonal states, perhaps categorized in some way. Identifying an individual with a relation may seem like a direct violation of first-order conventions, but it is important to understand that this is not necessarily an irreconcilable difference of opinion between A and B. The traditional Tarskian semantics for first-order logic requires that the universe of discourse be a nonempty set of things called 'individuals', but it does not mandate what \textit{kind} of things must be in this set. In particular, there is no requirement that individuals be in any sense inherently non-relational in nature; in fact, it is quite common to introduce relations into a first-order theory by treating them as arguments to a 'dummy' relation, writing expressions such as \((\text{Holds } R \ a \ b)\) rather than \((R \ a \ b)\). Thus, A's use of a name to denote an individual, and B's use of the same name to denote a relation, need not be considered a difference of opinion: they can both be right, and can both correctly use first-order reasoning. In this case, however, the agent C who wishes to make use of both their sentences is faced with a dilemma, since traditional first-order signature-based syntax is unable to accommodate both points of view at once. Common Logic relaxes this rigidity and allows 'punning' between relation and individual names, as between relation and function names.

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

\[(= \text{married} \ (\text{conjugalRelationBetween Joe Jane}))\]

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

\[(\text{married} \ Jack \ Jill)\]

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

\[(\text{conjugalRelationBetween Joe Jane} \ Jack \ Jill)\]

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

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

\[(\exists x \ ( = x \ (\text{conjugalRelationBetween Joe Jane})))\]

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

\[(\exists x \ (x \ Jack \ Jill))\]

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

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

\[
\begin{align*}
& \text{married (Jack Jill)} \\
& \text{(exists (x) (and (married x) (= Jack (husband x)) (= Jill (wife x))))} \\
& \text{(= (when (married Jack Jill)) (hour 3 (pm (thursday (week 12 (year 1997))))) )} \\
& \text{(= (wife (married 32456)) Jill)} \\
& \text{(ConjugalStatus married Jack)} \\
& \text{((ConjugalStatus Jack) Jill)}
\end{align*}
\]

These all express similar propositions about a Jack and Jill's being married, but in widely different ways. The logical name married appears here as a binary relation between people; a three-place relation between two persons and a time; a function from numbers to individuals; as an individual 'status', and finally, although not named explicitly, as the value of a function from persons to predicates on persons. In a fully expressive Common Logic dialect, such as Common Logic core, all these forms can be used at the same time.

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

\[(\text{rdfs:} \text{class rdfs:class})\]

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

5.2.3 On being first-order

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

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

6 Common Logic abstract syntax and semantics

This section is normative.

6.1 Common Logic abstract syntax.

We describe the syntax of common logic ‘abstractly’ here in order to not be committed to any particular dialect’s syntactic conventions.

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

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

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.

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

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

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.

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

A Boolean sentence has a type, called a connective, and a number of sentences called the components of the Boolean sentence. The number depends on the particular type. Every common logic dialect must distinguish the conjunction, disjunction, negation, implication and biconditional types with respectively any number, any number, one, two and two components.
The current specification does not recognize any irregular sentence forms. They are included in the abstract syntax to accommodate future extensions to Common Logic, such as modalities.

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

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

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

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

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

A dialect which provides only some of the common logic forms may be described as a partial common logic dialect, or as partially conformant. In particular, a dialect which does not provide for term sequences with a sequence variable, but is otherwise fully conformant, is a compact dialect, and may be described as a fully conformant compact dialect if it provides for all other constructions in the abstract syntax.

Dialects may in addition provide for other forms of sentence construction not described by this syntax, but in order to be fully conformant such constructions must either be new categories defined in terms of these categories, or be extensions of these categories (e.g. new kinds of Boolean sentence, or kinds of quantifier) which are equivalent in meaning to a construction using just this syntax, interpreted according to the Common Logic semantics; that is, they can be considered to be systematic abbreviations, or macros; also known as syntactic sugar. The core dialect, described in sec. 6, contains a number of syntactic sugared forms for quantified and atomic sentences. (Other types of compliance are also recognized: see section 7 for a full account of conformance.)

The only undefined terms in the above are name and sequence variable. The only required constraint on these is that they must be exclusive. Dialects intended for transmission of content on a network should not impose arbitrary or unnecessary restrictions on the form of names, and must provide for names to be used as identifiers of common logic texts. Dialects intended for use on the Web should allow Universal Resource Identifiers and URI references [refRFC2396] to be used as names. Common Logic dialects should define names in terms of Unicode (ISO/IEC 10646) conventions.

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

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

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

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

- every name shall be classified as either denoting or as non-denoting;
no name shall be classified as both denoting and non-denoting;

- no non-denoting name shall be bound by a quantifier;

- the result of substituting a non-denoting name for the bound name in the body of any quantified expression shall not be a legal expression.

- No equation shall contain both a denoting and a non-denoting name

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

6.2 Common logic semantics

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

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

An interpretation \( I \) of a vocabulary \( V \) is a set \( U_I \), the universe, with a distinguished nonempty subset \( D_I \), the domain of discourse, or simply domain, and four mappings: \( \text{rel}_I \) from \( U_I \) to subsets of \( D_I^* \), \( \text{fun}_I \) from \( U_I \) to functions \( D_I^* \rightarrow D_I \), (which we will also consider to be the set \( D_I^* \times D_I \)), \( \text{int}_I \) from names in \( V \) to \( U_I \), and \( \text{seq}_I \) from sequence variables in \( V \) to \( D_I^* \). If the dialect is segregated, then \( \text{int}_I(x) \) is in \( D_I \), and only if \( x \) is a denoting name. If the dialect recognizes irregular sentences, then they are treated as names of propositions, and \( \text{int}_I \) also includes a mapping from the irregular sentences of a text to the truthvalues \{true, false\}.

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

The truth-conditions require some auxiliary definitions.

A name map on a vocabulary \( V \) (relative to an interpretation \( I \)) is any mapping from \( V \) into \( D_I \). A sequence map on a set of sequence variables is any mapping from those variables to the set \( D_I^* \) of finite sequences of elements of \( D_I \). (Note that \( D_I^* \) contains the empty sequence \( <> \), for any \( D_I \).) If \( A \) is a name map or sequence map on \( S \), \( I[A] \) is the interpretation which is like \( A \) on names or sequence variables in \( S \), but otherwise is like \( I \). Formally, \( I[A] = U_I, D_I[A] = D_I, \text{rel}_I[A] = \text{rel}_I, \text{fun}_I[A] = \text{fun}_I, \text{int}_{I[A]}(v) = A(v) \) when \( v \) is in \( S \), otherwise \( \text{int}_{I[A]}(v) = \text{int}_I(v) \). Note that if \( x \) is in the domain of a \( B \) then \( \text{int}_{I[A][B]}(x) = B(x) \).

If \( E \) is a subset of \( D_I \), then the restriction of \( I \) to \( E \) is an interpretation \( K \) of the same vocabulary and over the same universe and with \( \text{int}_K = \text{int}_I \) and \( \text{seq}_K = \text{seq}_I \), but where \( \text{rel}_K = \text{rel}_I \) and \( \text{fun}_K(v) \) is the restriction of \( \text{fun}_I(v) \) to \( E^* \rightarrow E^* \), for all \( v \) in the vocabulary of \( I \). If \( S \) is a set of names, the retraction of \( I \) from \( S \), \( I[<S>] \), is the restriction of \( I \) to the set \( \{ D_I - \{ \text{int}_I(x) \mid x \in S \} \} \).

If \( s=<s_1, \ldots, s_m> \) and \( t=<t_1, \ldots, t_m> \) are finite sequences, then \( s;t \) is the concatenated sequence \( <s_1, \ldots, s_m, t_1, \ldots, t_m> \).

The value of any expression \( E \) in the interpretation \( I \) is then given by following the rules in the semantic table.
**Table 1. Interpretations of Common Logic Expressions.**

<table>
<thead>
<tr>
<th>Expression Type</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name N</td>
<td>int(I(N))</td>
</tr>
<tr>
<td>Sequence variable S</td>
<td>seq(I(S))</td>
</tr>
<tr>
<td>Term sequence T₁ … Tₙ</td>
<td>l₁(T₁), … lₙ(Tₙ)&gt;</td>
</tr>
<tr>
<td>Term sequence T₁ … Tₙ with sequence variable S</td>
<td>l₁(T₁), … lₙ(Tₙ)&gt;; l(S)</td>
</tr>
<tr>
<td>Term which is an equation containing terms T₁, T₂</td>
<td>true if l(T₁) = l(T₂), otherwise false</td>
</tr>
<tr>
<td>Term with operator O and term sequence S</td>
<td>fun₁(l(O)) l(S)</td>
</tr>
<tr>
<td>Boolean sentence of type negation and component C</td>
<td>true if l(C) = false, otherwise false</td>
</tr>
<tr>
<td>Boolean sentence of type conjunction and components C₁ … Cₙ</td>
<td>l₁(C₁) = … = l(Cₙ) = true, otherwise false</td>
</tr>
<tr>
<td>Boolean sentence of type disjunction and components C₁ … Cₙ</td>
<td>false if l(C₁) = … = l(Cₙ) = false, otherwise true</td>
</tr>
<tr>
<td>Boolean sentence of type implication and components C₁, C₂</td>
<td>true if l(C₁) = l(C₂), otherwise false.</td>
</tr>
<tr>
<td>Boolean sentence of type biconditional and components C₁, C₂</td>
<td>false if for every name map A on N, l<a href="B">A</a> is false; otherwise true</td>
</tr>
<tr>
<td>Quantified sentence of type universal and set of names N and body B</td>
<td>true if for every name map A on N, l<a href="B">A</a> is true; otherwise false</td>
</tr>
<tr>
<td>Irregular sentence S</td>
<td>int(I(S))</td>
</tr>
<tr>
<td>Phrase which is a sentence S</td>
<td>true if for every sequence map B on the set of sequence variables in S, l<a href="S">B</a> is true; otherwise false</td>
</tr>
<tr>
<td>Module with name N, exclusion set L and body text B</td>
<td>true if l[&lt;L]l(B) = true and ext(l(N)) = D₁[l&lt;2], otherwise false</td>
</tr>
<tr>
<td>Text containing phrases S₁ … Sₙ</td>
<td>true if l(S₁) = … = l(Sₙ) = true, otherwise false</td>
</tr>
</tbody>
</table>
The meaning of the function text in the clause for importation is described in the next section.

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

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

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

6.3 Importing and identification on a network.

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

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

Names used to name texts on a network are understood to be rigid and to be global in scope, so that the name can be used to identify the thing named – in this case, the Common Logic text – across the entire communication network. (See [refRFC2396] for further discussion.) A name which is globally attached to its denotation in this way is an identifier, and is typically associated with a system of conventions and protocols which govern the use of such names to identify, locate and transmit pieces of information across the network on which the dialect is used. While the details of such conventions are beyond the scope of this document, we can summarize their effect by saying that the act of publishing a named Common Logic text is intended to establish the name as a rigid identifier of the text, and Common Logic acknowledges this by requiring that all interpretations shall conform to such conventions when they apply to the network situation in which the publication takes place.

Named texts are not required to be in 1:1 correspondence to documents, files or other units of data storage. Dialects or implementations may provide for texts to be distributed across storage units, or for multiple named texts to be stored in one unit. The naming conventions for text may be related to the addressing conventions in use for data units, but this is not required. Texts may also be identified by external naming conventions, for example by encoding the text in documents or files which have network identifiers; the semantics described in this section must be applicable to all names used as network identifiers.
It is important here to distinguish the act of naming from that of asserting the truth of the text itself. Publishing a named text does not, in itself, necessarily make any claim about the truth of the text; but it does make a claim about the denotation of the name of the text.

In order to state semantic conditions on identifiers we need to assume appropriate values to exist in the universe of discourse. We will call the semantic entity corresponding to a named text a named text value. The exact nature of a text value is unimportant, but the semantics considers them to be pairs consisting of a name and a Common Logic text: \( t = \langle \text{name}(t), \text{text}(t) \rangle \). 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.

\[
\text{if } t \text{ is a text value in } U_I \text{ and } \text{name}(t) \text{ is in } V, \text{ then } \text{int}_I(\text{name}(t)) = t.
\]

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:

<table>
<thead>
<tr>
<th>publication on the network of:</th>
<th>requires that for any interpretation I of a text on the network:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a text ( T ) with a name ( N )</td>
<td>( U_I ) contains a named text value ( t ) with ( \text{text}(t) = T ) and ( \text{name}(t) = N )</td>
</tr>
</tbody>
</table>

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

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

6.3.1 Mixed networks

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

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

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

### 6.4 Satisfaction, validity and entailment.

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

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

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

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

### 6.5 Summary of the core syntax

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

<table>
<thead>
<tr>
<th>sequence variable $S$</th>
<th>a string of Unicode characters, other than ‘(’, ‘)’ or white space, beginning with ‘…”’</th>
</tr>
</thead>
<tbody>
<tr>
<td>name $N$</td>
<td>a string of Unicode characters, other than ‘(’, ‘)’ or white space, and which is not a sequence variable.</td>
</tr>
<tr>
<td>term sequence $T_1 \ldots T_n$</td>
<td>a sequence of terms written left to right, separated by white space: $[T_1] \ldots [T_n]$</td>
</tr>
<tr>
<td>term sequence $T_1 \ldots T_n$ with sequence variable $S$</td>
<td>as above, with a sequence variable at the right: $[T_1] \ldots [T_n] [S]$</td>
</tr>
<tr>
<td>term which is an equation containing terms $T_1, T_2$</td>
<td>$(= [T_1] [T_2])$</td>
</tr>
<tr>
<td>Term Description</td>
<td>CL Expression</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>term with operator $O$ and term sequence $S$</td>
<td>$([O] [S])$</td>
</tr>
<tr>
<td>atom with predicate $P$ and term sequence $S$</td>
<td>$([P] [S])$</td>
</tr>
<tr>
<td>boolean sentence of type negation and component $C$</td>
<td>(not $[C]$)</td>
</tr>
<tr>
<td>boolean sentence of type conjunction and components $C_1, \ldots, C_n$</td>
<td>(and $[C_1] \ldots [C_n]$)</td>
</tr>
<tr>
<td>boolean sentence of type disjunction and components $C_1, \ldots, C_n$</td>
<td>(or $[C_1] \ldots [C_n]$)</td>
</tr>
<tr>
<td>boolean sentence of type implication and components $C_1, C_2$</td>
<td>(iff $[C_1] [C_2]$)</td>
</tr>
<tr>
<td>boolean sentence of type universal and set of names $N$ and body $B$</td>
<td>(forall $(N) \ [B]$)</td>
</tr>
<tr>
<td>quantified sentence of type existential and set of names $N$ and body $B$</td>
<td>(exists $(N) \ [B]$)</td>
</tr>
<tr>
<td>phrase which is a sentence $S$</td>
<td>$[S]$</td>
</tr>
<tr>
<td>phrase which is an importation containing name $N$</td>
<td>(cl:imports $[N]$)</td>
</tr>
<tr>
<td>text containing phrases $S_1, \ldots, S_n$</td>
<td>a sequence of phrases written left to right, separated by white space: $[S_1] \ldots [S_n]$</td>
</tr>
<tr>
<td>Named text with name $N$ and phrases $S_1, \ldots, S_n$</td>
<td>(cl:text $[N] \ [S_1 \ldots S_n]$)</td>
</tr>
</tbody>
</table>

### 6.6 Sequence variables, recursion and argument lists: discussion

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

This ability to represent infinite sets of sentences in a finite form means that Common Logic with sequence variables is not compact, and therefore not first-order; for clearly the infinite set of sentences corresponding in meaning to a simple atomic sentence containing a sequence variable is logically equivalent to that sentence and so entails it, but no finite subset of the infinite set does. However, the intended use of sentences containing sequence variables is to

---

**ISO/WD 24707**

© ISO 2004. All rights reserved.
act as axiom schemata, rather than being posed as goals to be proved, and when they are restricted to this use the resulting logic is compact. Also, even without this restriction, Common Logic is finitely complete, in the sense there are inference schemes which can derive T from S if S entails T and S is finite. Since Common Logic sentences can express the same content as infinite sets of conventional first-order sentences, the limitation to finite antecedents is less restrictive than it might seem; in fact, this completeness is a strengthening of Gödel’s classical first-order completeness result.

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

6.7 Special cases and translations between dialects.

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

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

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

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

6.7.1 Translating between dialects

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

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

Translation from an unsegregated dialect B into a segregated dialect A requires that names are used so as to respect the restrictions of the dialect. This may require adding axioms to the translations in order to ensure that the domain of an interpretation of the segregated translation of any text corresponds to the universe of an interpretation.
of the unsegregated text. There is a general technique called the holds-app translation for translating any Common Logic dialect into a similar classical dialect. An atomic sentence with predicate \( P \) and argument term sequence \( S_1 \ldots S_n \) translates into an atomic sentence with predicate \( \text{holds} \) and argument sequence \( P \ S_1 \ldots S_n \). A term with operator \( O \) and argument sequence \( S_1 \ldots S_n \) translates into a term with operator \( \text{app} \) and argument sequence \( O \ S_1 \ldots S_n \). The introduced predicate and operator require no other axioms: their only role is to allow the operators and predicates of the B dialect to denote entities in the domain of the A dialect translation.

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

### 7 Conformance

**Editor's Note: This is still undergoing discussion.**

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.

##### 7.1.1 Syntax

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

A dialect is syntactically fully conformant if its parsings recognize expressions of every category in the abstract syntax. Conformant dialects or subdialects whose parsings include other categories of sentences should categorize them as irregular sentences for common logic conformity. It is conformant as a syntactic subdialect if it recognizes only some of the CL categories; but any dialect must recognize some form of sentence construct. Particular cases include compact dialects which recognizes all categories except the sequence variable, and….

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

##### 7.1.2 Semantics

Any CL dialect must have a model-theoretic semantics, defined on a set of interpretations, called dialect interpretations, which assigns one of the two truth-values true or false to every sentence, phrase or text in that dialect. (Multi-truth-valued or partial semantics are not acceptable.)

A dialect is weakly semantically conformant if for any dialect interpretation \( I \) of any sentence, phrase or text \( T \) in that dialect, there is a common logic interpretation \( J \) of \( T \) with \( J(T) = I(T) \).
If in addition it is the case that for any common logic interpretation \( I \) of \( T \), there is a dialect interpretation \( J \) with \( I(T) = J(T) \), then the dialect is \textit{exactly semantically conformant}. (The simplest way to achieve exact semantic conformance is to simply adopt the CL model theory.)

Subdialects are required to satisfy these conditions only on expressions which are legal for the dialect, in order to qualify as conformant.

A dialect which is weakly but not strongly semantically conformant is called a \textit{semantic extension}. (This allows a semantic extension to apply ‘external’ semantic conditions, for example to irregular sentences. The Common Logic core is an example of a semantic extension, by virtue of the conditions on numbers and quoted strings and datatypes, \( (= (\text{xsd:integer} \ '3') \ 3) \) is logically true in the core but might be false in a CL interpretation.)

A syntactically segregated dialect which requires names in one or more categories to \textit{not} denote entities in the set over which the quantifiers range is a \textit{segregated dialect}. Segregated dialects \textbf{must} provide criteria which are sufficient to enable an application to detect that a name is non-denoting in the dialect without performing operations on any structure other then the name itself, and \textbf{must} provide a name which can be used by other dialects to refer to the universe of discourse of the dialect; that is, the dialect must specify as a semantic condition in all dialect interpretations that the relational extension of this name, when used as a predicate, shall be \textit{true} of all entities in the domain of the interpretation. Other dialects may treat such names as denoting names.

### 7.2 Applications

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

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

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

Applications which detect entailment relationships between CL texts in the conformance set are \textbf{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 \( I(S) = \text{true} \) then \( I(T) = \text{true} \)). The application is \textbf{complete} when, for any texts \( T \) and \( S \) in dialects in the conformance set, if \( S \) common-logic entails \( T \) then the application can detect the entailment of \( T \) from \( S \). (Note this requires completeness ‘across’ dialects in the conformance set).

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

### 7.3 Networks

Conformance of communication networks is defined relative to a collection of dialects, called the conformance set. A network is conformant when it transmits all expressions of all dialects in the conformance set without distortion from any node in the network to any other node. Network transmission errors or failures which are indicated as error conditions do not count as distortion for purposes of conformance of a network.
Needs some stuff here on treating identifiers as globally rigid. This might need some thought, eg do we allow dynamically changing texts on a network, such as might be produced by a news feed? The Web stuff in RFC 2396 and REST (see http://webservices.xml.com/pub/a/ws/2002/02/06/rest.html for a nice overview) treats ‘resources’ as functions from times to states but requires identification of a resource to be rigid, which is an interesting way to try to get the best of both. This needs some work.
Annex A
(normative)
Knowledge Interchange Format (KIF) – Syntax

Editor’s note about the annexes

Annexes A, B, and C will have similar structure – they are intended to specify standard forms for four logic formalisms that have been specifically tailored to conform completely to the Common Logic standard: KIF, CGIF, and XCL. They are all normative.

Annexes D, E, and F will also have a similar structure to each other. They will demonstrate conformance for each of the standard formalisms in A through D. They are all informative.

A.1 Introduction

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

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

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

Third, KIF was based explicitly on LISP. KIF syntax was defined to be LISP S-expressions; and LISP-based ideas were incorporated into the semantics of KIF, for example in the way that the semantics of sequence variables were defined. Although the Common Logic core 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 [refSW], and in particular, Common Logic is intended to be useable as a slightly improved Lbase [refLbase], i.e. as serving the role of a semantic reference language for other SW notations.

A.2 Common Logic Core Syntax

The following syntax is written using Extended Backus-Naur Form (EBNF), as specified by ISO/IEC 14977. Literal characters are 'quoted', sequences of items are separated by commas, | indicates a separation between alternatives, { } indicates a sequence of expressions, - indicates an exception, [ ] indicates an optional item, and parentheses ( ) are used as grouping characters. Productions are terminated with ;.
The syntax is written to apply to ASCII encodings. It also applies to full Unicode character encodings, with the change noted below to the category nonascii.

The syntax is presented here in two parts. The first deals with parsing character streams into lexical items: the second is the logical syntax of Common Logic Core, 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 Common Logic Core, or core, expression is encoded as a sequence of Unicode characters as defined in ISO/IEC 10646. [KIF requires ASCII encodings.] Any character encoding which supports the repertoire of ISO 10646 may be used, but UTF-8 (ISO 10646 Annex D ) is preferred. Only characters in the US-ASCII subset are reserved for special use in the core itself, so that the language can be encoded as an ASCII text string if required. This document uses ASCII characters. Unicode characters outside the ASCII range are represented in Common Logic core ASCII text by a character coding sequence of the form \u00000nnn or \U00000nnnnn where n is a hexadecimal digit character. When transforming an ASCII text string to a full-repertoire character encoding, or when printing or otherwise rendering the text for maximum accessibility for human readers, such a sequence may be replaced by the corresponding direct encoding of the character, or an appropriate glyph. Moreover, these coding sequences are understood as denoting the corresponding Unicode character when they occur in quoted strings (see below).

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

The backslash \(\text{reverse solidus U+005C}\) 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 \(\text{\u00000} \text{U+0005C, U+002C}\) is used to encode a single quote inside a common logic quoted string, and similarly the combination \(\text{\u00000} \text{U+0005C, U+0022}\) indicates a double quote inside a double-quoted enclosed name string. These inner-quote conventions apply in both ASCII and full Unicode renderings. Any other uses of the backslash are treated simply as single U+005C characters.

A.2.2 Lexical syntax

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

Single quote (apostrophe) is used to delimit quoted strings, and double quote to delimit enclosed names, which obey special lexicalization rules. [KIF does not use the enclosed-name device.] Quoted strings and enclosed names are the only Common Logic core 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 Common Logic core syntax.

```
open = \ '{' ;
close = \ '}' ;
stringquote = \ ''' ;
amequote= \ ''
```
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';
```

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

```
hexa = 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';
```

Certain character sequences are used to indicate the presence of a single character. nonascii is the set of characters or character sequences which indicate a Unicode character outside the ASCII range. For input using a full Unicode character encoding, this production should be ignored and nonascii should be understood instead to be the set of all non-control characters of Unicode outside the ASCII range which are supported by the character encoding. The use of the \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 | '\U', hexa, hexa, hexa, hexa, hexa, hexa;
innerstringquote = '\'';
innernamequote = '\"';
numeral = digit, { digit };
```

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

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

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

Quoted strings and enclosed names require a different lexicalization algorithm than other parts of Common Logic core text, since parentheses and whitespace do not break a quoted text stream into lexical tokens.
When Common Logic Core text is enclosed inside markup which uses character escaping conventions, the Common Logic quoted string conventions here described are understood to apply to the text described or indicated by the conventions in use, which should be applied first. Thus for example the content of the XML element: `<cl-text>&apos;a\'&apos;b&lt;c&apos;</cl-text>` is the core syntax quoted string `a\'b&lt;c` which denotes the five-character text string `a'b<c`. Considered as bare core text, however, `&apos;a\'&apos;b&lt;&lt;\'&apos;c` would simply be a rather long name.

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

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

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

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

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

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

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

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

lexicaltoken  = open | close | nonlexbreak ;

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

### A.2.3 Expression syntax

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

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

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

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

```plaintext
name = namesequence | numeral | enclosedname | quotedstring ;
```
Names count as terms, and a complex (application) term consists of a function with a vector of arguments. Terms may also have an associated comment, represented as a quoted string (in order to allow text which would otherwise break the lexicalization). Comments enclose the term they comment upon. [KIF handles comments differently and does not have the ‘enclosing’ construction.]

\begin{verbatim}
term = name | ( open, term, termseq, close ) | ( open, 'cl:comment', quotedstring, term, close ) ;
\end{verbatim}

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

\begin{verbatim}
equation = open, '=', term, term, close
\end{verbatim}

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

\begin{verbatim}
atomsent = equation | ( open, term, termseq, close ) | ( open, open, 'roleset:', { open, name, term, close }, close, close ) ;
\end{verbatim}

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

\begin{verbatim}
boolsent = ( open, ('and' | 'or'), { sentence }, close ) | ( open, ('implies' | 'iff'), sentence, sentence, close ) | ( open, 'not', sentence, close ;
\end{verbatim}

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

\begin{verbatim}
quantsent = open, ('forall' | 'exists'), [ name ], open, { name | ( open, name, term, close )}, close, sentence, close ;
\end{verbatim}

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

\begin{verbatim}
sentence = atomsent | boolsent | quantsent | ( open, 'cl:comment', quotedstring, close ) ;
\end{verbatim}

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

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

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

\begin{verbatim}
phrase = sentence | module | (open,'cl:imports', name , close) | (open, 'cl:comment', quotedstring, [phrase], close);
cltext = { phrase } ;
clnamedtext = open, [namesequence |enclosedname], cltext, close ;
\end{verbatim}
A module without an exclusion list is not a named text.

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

B.1 General
This annex describes the syntax of CGIF which conforms to the standard.

B.2 Terms and Definitions
For the purpose of this annex, the terms and definitions given in ISO/IEC 10646 and the following list apply. Some terms may be shortened to acronyms or abbreviations, which are listed in parentheses after the term. The term "abstract syntax", for example, may be abbreviated "AS”, and "conceptual relation" may be abbreviated “relation”.

B.2.1 Terms

B.3 CGIF Concrete Syntax
The conceptual graph interchange form (CGIF) is a representation for conceptual graphs intended for transmitting CGs across networks and between IT systems that use different internal representations. All features defined by the abstract syntax of Chapter 6 and the extensions of Chapter 8 are representable in CGIF. The CGIF syntax ensures that all necessary syntactic and semantic information about a symbol is available before the symbol is used; therefore, all translations can be performed during a single pass through the input stream. The CGIF syntax is specified in terms of the Extended BNF (EBNF) rules and metalevel conventions, which are defined in Annex B.

B.4 Lexical Categories
The lexical categories of CGIF are defined as sequences of characters as specified in ISO/IEC 10646. Characters outside the 7-bit subset of ISO/IEC 10646 occur only in strings that represent identifiers, names, literals, and comments. By using escape sequences in such strings, it is possible to encode and transmit CGIF in the 7-bit subset of ISO/IEC 10646. Numerical values are represented by the conventions of ANSI X3.42. The following lexical categories may be used in EBNF rules in this chapter or in the translation rules in chapters 8 and 9:

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

B.4.1 Comment.
A comment is a delimited string with a semicolon “;” as the delimiter.

Comment ::= DelimitedStr(";")

B.4.2 DelimitedStr(D).
A delimited string is a sequence of two or more characters that begin and end with a single character D called the delimiter. Any occurrence of D other than the first or last character must be doubled.

DelimitedStr(D) ::= D (AnyCharacterExcept(D) | D D)* D
B.4.3 Exponent.

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

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

B.4.4 Floating.

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

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

B.4.5 Identifier.

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

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

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

B.4.6 Integer.

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

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

B.4.7 Name.

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

    Name ::= DelimitedStr("'")

B.4.8 Number.

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

    Number ::= Floating | Integer

B.4.9 QuotedStr.

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

    QuotedStr ::= DelimitedStr(""")

B.4.10 UnsignedInt.

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

    UnsignedInt ::= Digit+

B.5 Syntactic Categories

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

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

B.5.1 Actor.

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

\[
\text{Actor} :: \ "<" \ Type(N) \ Arc^* \ "|" \ Arc^* \ Comment? \ ">
\]

The arcs that precede the vertical bar are called input arcs, and the arcs that follow the vertical bar are called output arcs. The valence N of the actor type must be equal to the sum of the number of input arcs and the number of output arcs. For purposes of conformance and translation within the standard, an actor will be treated as a conceptual relation.

B.5.2 Arc.

An arc is a concept or a bound label.

\[
\text{Arc} :: \ \text{Concept} \ | \ \text{BoundLabel}
\]

B.5.3 BoundLabel.

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

\[
\text{BoundLabel} :: \ "?" \ Identifier
\]

B.5.4 CG.

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

\[
\text{CG} :: \ (\text{Concept} \ | \ \text{Relation} \ | \ \text{Actor} \ | \ \text{SpecialContext} \ | \ \text{Comment})^*
\]

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

B.5.5 CGStream.

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

\[
\text{CGStream} :: \ (\text{CG} \ ".")^*
\]

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

B.5.6 Concept.

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

\[
\text{Concept} :: \ "[" \ Type(1)? \ \{\text{CorefLinks?}, \ \text{Referent?}\} \ Comment? \ "]"
\]
If the type is omitted, the default type is Entity. This rule permits the coreference labels to come before or after the referent. If the referent is a CG that contains bound labels that match a defining label on the current concept, the defining label must precede the referent.

In Figure 4, for example, the concept [Person: Mary] could be written in CGIF as [Person:'Mary'*x]; the coreferent concept [\&top:] could be written [?x], and its implicit type would be Entity.

**B.5.7 Conjuncts.**

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

\[
\text{Conjuncts}(N) \ := \ \text{TypeTerm}(N) \ ("&" \ \text{TypeTerm}(N))^* \\
\]

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

**B.5.8 CorefLinks.**

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

\[
\text{CorefLinks} \ := \ \text{DefLabel} \ | \ \text{BoundLabel}^* \\
\]

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

**B.5.9 DefLabel.**

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

\[
\text{DefLabel} \ := \ "*" \ Identifier \\
\]

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

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

1. A defining concept is being imported into a context that is within the scope of another defining concept with the same identifier.
2. A defining concept is being imported into a context that contains some nested context that has a defining concept with the same identifier.
3. Somewhere in the same module there exists a defining concept whose identifier is the same as the identifier of the defining concept that is being imported, but neither concept is within the scope of the other.

In cases (1) and (2), any possible conflict can be detected by scanning no further than the right bracket "]" that encloses the context into which the graph is being imported. Therefore, in those two cases, the newly imported defining coreference label and all its bound labels must be replaced with an identifier that is guaranteed to be distinct. In case (3), there is no conflict that could affect the semantics of the conceptual graphs or any correctly designed CG tool; but since a human reader might be confused by the similar labels, a CG tool may replace the identifier of one of the defining coreference labels and all its bound labels.

**B.5.10 Descriptor.**

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

**B.5.11 Designator.**

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

Designator ::= Literal | Locator | Quantifier

**B.5.12 Disjuncts.**

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

Disjuncts(N) ::= Conjunctions(N) ("|" Conjunctions(N))*

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

**B.5.13 FormalParameter.**

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

FormalParameter ::= Type(1) [DefLabel]

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

**B.5.14 Indexical.**

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

Indexical ::= "#" Identifier?

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

**B.5.15 IndividualMarker.**

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

IndividualMarker ::= "#" UnsignedInt

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

**B.5.16 LambdaExpression(N).**

A lambda expression begins with "(" and the keyword "lambda", it continues a signature and a conceptual graph, and it ends with ")".

LambdaExpression(N) ::= "(" "lambda" Signature(N) CG ")"

A lambda expression with N formal parameters is called an N-adic lambda expression. The simplest example, represented "(lambda ())", is a 0-adic lambda expression with a blank CG.

**B.5.17 Literal.**

A literal is a number or a quoted string.

Literal ::= Number | QuotedStr
B.5.18 Locator.

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

\[
\text{Locator ::= Name | IndividualMarker | Indexical}
\]

B.5.19 Negation.

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

\[
\text{Negation ::= } \sim [\text{ CG }]\]

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

B.5.20 Quantifier.

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

\[
\text{Quantifier ::= } @ (\text{UnsignedInt | Identifier } \{\text{ Arc* }\})?
\]

The symbol @some is called the \textit{existential quantifier}, and the symbol @every is called the \textit{universal quantifier}. If the quantifier is omitted, the default is @some.

B.5.21 Referent.

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

\[
\text{Referent ::= } :: \{\text{Designator?, Descriptor?}\}
\]

B.5.22 Relation.

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

\[
\text{Relation ::= } \{(\text{ Type(N) Arc* Comment? }\})
\]

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

B.5.23 Signature.

A signature is a parenthesized list of zero or more formal parameters separated by commas.

\[
\text{Signature ::= } (\text{ FormalParameter }\{(\text{ FormalParameter)*}\})
\]

B.5.24 SpecialConLabel.

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

\[
\text{SpecialConLabel ::= } \text{"if" | "then" | "either" | "or" | "sc"}
\]

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

B.5.25 SpecialContext.

A special context is either a negation or a left bracket, a special context label, an optional colon, a CG, and a right bracket.
SpecialContext ::= Negation | "[" SpecialConLabel ":"? CG "]"

B.5.26 Structure.

A structure consists of an optional percent sign "%" and identifier followed by a list of zero or more arcs enclosed in braces.

Structure ::= ("%" Identifier)? "{" Arc* "}"

B.5.27 Type.

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

Type(N) ::= TypeLabel(N) | TypeExpression(N)
A concept type must have valence N=1. A relation type must have valence N equal to the number of arcs of any relation or actor of that type. The type label or the type expression must have the same valence as the type.

B.5.28 TypeExpression.

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

TypeExpression(N) ::= LambdaExpression(N) | "(" Disjuncts(N) ")"
The type expression must have the same valence N as the lambda expression or the disjunction list.

B.5.29 TypeLabel(N).

A type label is an identifier.

TypeLabel(N) ::= Identifier
The type label must have an associated valence N.

B.5.30 TypeTerm.

A type term is an optional tilde "~" followed by a type.

TypeTerm(N) ::= "~"? Type(N)
The type term must have the same valence N as the type.

B.5.31 Discussion.

Here is an example of a CGIF sentence that roughly corresponds to “a person is between a rock and a hard place.”

(Betw [Rock] [Place *x1] [Person]) (Attr ?x1 [Hard])

For more compact storage and transmission, all white space not contained in comments or enclosed in quotes may be eliminated:

(Betw[Rock][Place*x1][Person])(Attr?x1[Hard])

This translation takes the option of nesting all concept nodes inside the conceptual relation nodes. A logically equivalent translation, which uses more coreference labels, moves the concepts outside the relations:

[Rock *x1] [Place *x2] [Person *x3] (Betw ?x1 ?x2 ?x3)

The concept and relation nodes may be listed in any order provided that every bound label follows the defining node for that label.

Here is an example showing the embedding of literals of any kind within a CGIF sentence.

(Betw [Rock] [Place *x1] [Person]) (Attr ?x1 [Hard])
This example shows how literals of any kind may be represented in the referent field of a concept. The identifiers "WAV" and "JPEG" specify the method of encoding. In DF, the sound might be represented by a transcription such as "clink clankety Scrape" or it could be converted to audio when someone clicks a mouse on the concept node. To avoid storing multiple copies of large literals, such as sound or video, a single copy might be stored outside the CG system with only a locator in the referent field.

Here is another example graph.

\[\text{[Person: Tom} *\text{x1}] \text{[Believe *x2]} \text{(Expr} ?\text{x2} ?\text{x1})
\text{(Thme} ?\text{x2} \text{[Proposition}
\text{[Person: Mary} *\text{x3}] \text{[Want} *\text{x4]}
\text{(Thme} ?\text{x4} \text{[Situation}
\text{[Agnt [Marry} *\text{x5] ?\text{x3}] (Thme} ?\text{x5} \text{[Sailor]}) \text{])}\]

Note that the concept [T] which may be represented [?x3] in LF or CGIF, may also be omitted completely in CGIF since the coreference label [?x3] inside the conceptual relation of type Agnt is sufficient to show the connection. As these examples illustrate, CGIF is not as readable as DF, but it contains the equivalent semantic information. To reconstruct an exact image of the original DF, the comment fields in the concept and relation nodes may contain additional layout information to specify the style and placement of the nodes.

**B.6 Mapping Abstract Syntax to CGIF**

Whenever a feature of the abstract syntax (AS) specified in Chapter 6 has the same name as some CGIF category of Section 7.2, the AS feature is represented by a string defined by that CGIF category. The mapping of AS features to CGIF categories is not one to one because of the following exceptions:

1. **Comments.** No comments are represented in the abstract syntax. Therefore, the CGIF categories ActorComment, CGComment, ConComment, and RelComment do not correspond to anything in AS.

2. **Lexical Categories.** Since the AS features are independent of any notation or implementation, the lexical categories of Section 7.1, which are defined as character strings, do not have a direct mapping to AS. Some of them, such as WhiteSpace, do not correspond to anything in AS.

3. **Noncontiguous Constituents.** Every CGIF category defines a class of contiguous character strings. Some AS features, such as coreference sets, cannot be represented by a single contiguous string. Therefore, they must be mapped to a noncontiguous collection of strings, such as the defining and bound labels.

Each subsection from C.7.1 through C.7.10 specifies the CGIF strings that represent the AS features defined in the section from @@@@ respectively.

**B.6.1 Conceptual Graph.**

Every conceptual graph (CG) is represented by a string of category CG.

* Every arc \( x \) of a CG is a pair consisting of a relation \( r \) and a concept \( c \) that is linked to \( r \). The Relation string that represents \( r \) contains the Arc string that represents \( x \). That Arc string may either be the Concept string that represents \( c \) or a BoundLabel string whose identifier matches a DefLabel or BoundLabel in the Concept string that represents \( c \).

* A CG string may contain Concept strings that represent concepts that are not linked to any conceptual relation.

* Three kinds of conceptual graphs are given distinguished names:
1. The blank is represented by an empty string or a string that contains nothing but strings of category WhiteSpace or CGComment.

2. A singleton is represented by a string of category CG that contains exactly one string of category Concept, but no strings of category Relation.

3. A star is represented by a string of category CG that contains exactly one string of category Relation. Every Concept string in the CG string must represent one of the concepts attached to the conceptual relation of the star graph.

**B.6.2 Concept.**

Every concept cis represented by a Concept string that contains a Type string that represents the concept type of c and a Referent string that represents the referent of c.

**B.6.3 Conceptual Relation.**

Every conceptual relation r is represented by a Relation string or an Actor string. The relation type of r is the RelTypeLabel or the LambdaExpression contained in the Relation string or the Actor string. The valence of r is the valence of the RelTypeLabel or the LambdaExpression.

* The number of Arc strings in the Relation string or the Actor string is equal to the valence of r.
* The signature of r is the signature of the RelTypeLabel or the LambdaExpression contained in the Relation string or the Actor string.

A conceptual relation that is represented by an Actor string may have side effects that are not represented in the abstract syntax or the translation to predicate calculus defined in Chapter 8.

**B.6.4 Lambda Expression.**

Every lambda expression e is represented by a LambdaExpression string that contains n FormalParameter strings that represent the formal parameters of e and a CG string that represents the body of e.

* For each i from 1 to n, the i-th formal parameter of e is represented by the i-th FormalParameter string.
* The signature of e is the sequence of Type strings contained in the FormalParameter strings.

**B.6.5 Concept Type.**

Every type hierarchy T may be represented by a concept of type TypeHierarchy that contains a nested CG that defines the concept type labels of T and the partial ordering over T:

```plaintext
TypeHierarchy ::= "[" "TypeHierarchy"
   (TypeDefinition | TypeLabelOrdering)* "]"
```

A type definition is a star graph containing a conceptual relation of type Def that relates a type label to a lambda expression:

```plaintext
TypeDefinition ::= "(" "Def"
   "[" "TypeLabel" ">
   "[" "LambdaExpression" "]"
   "[" "ConTypeLabel"
   "]"
   "[" "ConTypeLabel"
   "]"
)"
```

A type label ordering is a star graph containing a conceptual relation of type EQ, GT, or LT that relates two type labels:

```plaintext
TypeLabelOrdering ::= "(" ("EQ" | "GT" | "LT")
   "[" "TypeLabel" ">
   "[" "ConTypeLabel"
   "]"
   "[" "ConTypeLabel"
   "]"
)"
```
For two type labels s and t, the type label ordering has a conceptual relation of type EQ iff s=t, of type GT iff s>t, and of type LT iff s<t.

The type definitions and type label orderings that define a type hierarchy must obey the following constraints:

* A type label that does not appear in a type definition is said to be primitive.

* Two primitive type labels are Entity and Absurdity. Every type hierarchy contains the star graph

\[(\text{GT } [\text{TypeLabel "Entity"}] [\text{TypeLabel "Absurdity"}]).\]

* A type label that appears in a type definition is said to be defined, and no type label may appear in more than one type definition.

For more concise storage and transmission, multiple star graphs for the type label ordering may be abbreviated by using a collection as defined in Section 8.2. As an example, the following star graph asserts that the type label “Entity” is related to each type label in the collection by the relation GT:

\[(\text{GT } [\text{TypeLabel "Entity"}] [\text{TypeLabel @Col{"Physical", "Abstract", "Independent", "Relative", "Mediating", "Continuant", "Occurrent"}}]).\]

The translation rules in Section 8.2 would expand this star graph to seven separate star graphs -- one for each type label in the collection.

### B.6.6 Relation Type.

Every relation hierarchy R may be represented by a concept of type RelationHierarchy that contains a nested CG that defines the relation type labels of R and the partial ordering over R:

```
RelationHierarchy ::= "{ "RelationHierarchy" (RelationDefinition | ValenceSpec | RelationLabelOrdering)* " ]"
```

A relation definition is a star graph containing a conceptual relation of type Def that relates a relation label to a lambda expression:

```
RelationDefinition ::= "{ "Def"
"[ "RelationLabel "\"" RelTypeLabel "\" ]"
"[ "LambdaExpression "\"" LambdaExpression "\" " ]"
"
```

A valence specification is a star graph containing a conceptual relation of type Has that relates a relation label to its valence:

```
ValenceSpec ::= "{ "Has"
"[ "RelationLabel "\"" RelTypeLabel "\" ]"
"[ "Valence Integer ]"
"
```

A relation label ordering is a star graph containing a conceptual relation of type EQ, GT, or LT that relates two type labels:

```
RelationLabelOrdering ::= "{ ("EQ" | "GT" | "LT")
"[ "RelationLabel "\"" RelTypeLabel "\" ]"
"[ "RelationLabel "\"" RelTypeLabel "\" " ]"
"
```

For two relation labels r and s, the relation label ordering has a conceptual relation of type EQ iff r=s, of type GT iff r>s, and of type LT iff r<s.

The most general relation type of valence n would be defined by a lambda expression (\lambda(Entity,...,Entity)), in which the signature is a list of n concepts of type Entity. A relation of that type would always be true when linked to n concepts of any type.
The relation definitions, valence specifications, and relation label orderings that define a relation hierarchy must obey the following constraints:

* Every primitive relation label must have exactly one valence specification and no relation definition.

* Every defined relation label must have exactly one relation definition and no valence specification.

For more concise storage and transmission, multiple star graphs for relation label orderings and valence specifications may be abbreviated by using collections. As an example, the following star graph asserts that each of the four type labels in the collection has the valence 2:

$$(\text{Has \{TypeLabel } @\text{Col\{"EQ", "GT", "LT", "Has"\}} \text{ [Valence 2]})$$

The translation rules in Section 8.2 would expand this star graph to four separate star graphs – one for each type label in the collection.

B.6.7 Referent.

The referent of a concept is represented by a quantifier string and a designator string.

* A quantifier string is one of the following two kinds:
  1. **Existential.** An existential quantifier is represented by the empty string "" or by the string "@exist".
  2. **Defined.** A defined quantifier is represented by the character "@" followed by an identifier or expression that is defined as a quantifier in Section 8.2.

* A designator is one of three kinds:
  1. **Literal.** A literal is represented by a Literal string.
  2. **Locator.** A locator is represented by an Indexical string, an IndividualMarker string, or a Name string.
  3. **Descriptor.** A descriptor is represented by a CG string containing at least one Concept string or Relation string.

B.6.8 Context

A context is represented by a SpecialContext string or by a Concept string that contains a CG string containing at least one Concept string or Relation string.

B.6.9 Coreference Set.

Every coreference set C is represented by an identifier i that matches the coreference labels of every Concept string that represent concepts in C.

* One of the dominant nodes d in C shall be selected as the defining node of C, and the Concept string that represents d shall contain the defining label *i and no other coreference labels.

* Every Concept string that represents a concept in C other than d shall contain a bound label ?i and no defining labels.

* No Concept string that represents any dominant concept in C may contain a coreference label that does not match i.

* Any Concept string that represents a nondominant concept in C may contain one or more bound labels other than ?i.

* If d is the only concept in C, the defining label *i is optional and may be omitted.
No concept within the scope of d that is not in C may be represented by a Concept string that contains a coreference label that matches i.

B.6.10 Module.

A module is represented by a context of type Module, which contains three contexts whose types are TypeHierarchy, CatalogOfIndividuals, and Assertion. Its syntax is defined by the following translation rule (see Annex B.2):

```
KB ::= "[" "Module" ":" 
    { "[" "TypeHierarchy" ":" CG.T "]", 
    "[" "CatalogOfIndividuals" ":" CG.C "]", 
    "[" " Assertion" ":" CG.O "]" } "]"
```

When the pattern part of this rule matches a CGIF string that represents a module, the CGs used as designators of the four nested concepts are assigned to the variables T, R, C, and O. Those CGs must obey the following constraints:

1. **Type hierarchy.** The conceptual graph T contains one concept of the form "[ConTypeLabel:" Identifier.t DefLabel "]" for each type label t. To specify the partial ordering, T also contains a collection of star graphs of the form "(PSbt BoundLabel BoundLabel ")", which specifies that the first bound label refers to a type label that has as proper subtype (PSbt) the type label referred to by the second bound label. For each defined type label, there is a star graph of the form "(Def BoundLabel [LambdaExpression:" " LambdaExpression.e "]")", which specifies that the bound label refers to a type label that is is defined (Def) by the quoted lambda expression in the following concept.

2. **Relation hierarchy.** The conceptual graph R contains one concept of the form "[RelTypeLabel:" Identifier.r DefLabel "]" for each relation type label r. To specify the partial ordering, R also contains a collection of star graphs of the form "(PSbt BoundLabel BoundLabel ")", which specifies that the first bound label refers to a relation type label that has as proper subtype the relation type label referred to by the second bound label. For each defined type label, there is a star graph of the form "(Def BoundLabel [LambdaExpression:" " LambdaExpression.e "]")", which specifies that the bound label refers to a relation type label that is is defined (Def) by the quoted lambda expression in the following concept.

3. **Catalog of individuals.** A concept whose designator is a conceptual graph g that contains global information about the individuals identified by individual markers in B. For each individual marker i that occurs in any concept of B, there is exactly one concept c immediately nested in g whose designator is i. The graph g may also contain other concepts and conceptual relations that describe the individuals in the catalog and the relationships among them.

4. **Outermost context.** A context A called the outermost context of B whose type label is Assertion and whose nested CG contains only type labels in T, relation labels in R, and individual markers in C.

Although CGIF requires that every bound coreference label be preceded by a matching defining coreference label, there is no such restriction on the type labels and the relation labels. Therefore, the definitions of type and relation labels may be recursive or mutually recursive. Any implementation of CGIF must be prepared to allocate internal storage space for type and relation labels before their definitions and partial ordering have been specified.

The CGs T and R specify the possible type and relation labels that may appear in any concept in the module together with their definitions and the partial orderings of the type and relation hierarchies. The CG C catalogs all the entities that represent individuals that may be explicitly referenced in the module; it may also assert some background knowledge about those individuals. All other assertions that are known or assumed to be true in the current module are stated by the CG O.

Since a module is also a concept, it may be linked to conceptual relations in a conceptual graph that makes a statement about the module, the CGs nested inside it, and the entities those CGs may refer to. Such a graph could be nested in the outermost context of another module, which would then be a metaknowledge base about the nested module. Modules may be nested to any depth; a metamodule may contain assertions about knowledge bases nested inside itself, but no module may contain references to a module in which it is nested.
Annex C
(normative)

Concrete Syntax – extended Common Logic Markup Language (XCL)

C.1 XCL and dialects

Editor's Note: the XCL syntax is still being discussed and modified!

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

C.2 Common Logic dialect text in XCL

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

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

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

Note that this convention assumes that XML escaping is applied to the enclosed Common Logic text. For example,

```
<XCL:text cl-dialect="cl:coreSyntax">
(cl:String 'this is &lt;forbidden&gt; in XML &amp; is considered &quot;naughty&quot;')
</XCL:text>
```

should be understood as indicating the Common Logic core syntax expression:

```
(cl:String 'this is <forbidden> in XML & is considered "naughty")
```

The use of `apos;` to escape a single-quote mark is deprecated in the body of a core syntax dialect element, due to potential confusion with the Common Logic core string-quoting conventions. If the syntactic specification of a dialect prohibits the use of certain characters or character sequences, the use of XML escaping to encode these characters or character sequences is illegal inside an `<XCL:text>` element with that dialect property. Such uses may be reported as syntactic errors by a conforming XCL application. There is no XCL requirement that the enclosed Common Logic dialect text be well-formed dialect Common Logic, although applications may report ill-formed dialect Common Logic as a syntactic error.
C.3 Common Logic Syntax in XCL

Common Logic can be rendered into XCL directly, in a form which displays the Common Logic syntax directly in the XML markup. This is analogous to "content markup" in MathML [refMathML]. The elements required for the Common Logic Core are described here, linked to the corresponding clauses of the EBNF Common Logic core syntax.

Other Common Logic dialects may extend these elements with others, provided that the new elements are reducible to the elements given here, in the sense described in section @@@Hmmm, not sure about this@@@.

The XCL elements are

cmodule
ctext
cphrase
sentence
equation
application
roleatom
term

C.4 <cmodule>

C.4.1 Discussion

The cmodule element represents a complete Common Logic module. Normally this will be used only once in a document, and may then be identified by the same URI as the document itself. However, a single document may contain any number of module elements. Following normal WWWWeb best practice, every module element must have a unique URI name different from the name of any other module. This is required in order to ensure consistency of the importing mechanism in other modules. The URI of a module should provide an access path to the definition of the module. Typically, the URI of a module in a document will be a URI reference of the form ex:base#modulename whose base is the URL of the document containing the module definition and whose extension is the internal anchor of the module in that document.

Any module must contain a name in the form of a URI and a some Common Logic text. It may also contain a header element.

C.4.2 Attributes

name
dialect
Annex D (informative)
Mapping of Knowledge Interchange Format (KIF) to Common Logic
Abstract Syntax

EDITOR’s note: this section will illustrate KIF’s mapping to the Common Logic abstract syntax and semantics. Since much of KIF’s recent development has been aimed for such conformance, the illustration should be straightforward. This serves two purposes: one, to “bless” the KIF syntax/semantics with respect to Common Logic and two, to serve as a guide to users of other logic systems who seek to demonstrate their conformance to the standard.
Annex E
(informative)

Mapping of Conceptual Graph Interchange Format (CGIF) to Common Logic Abstract Syntax

EDITOR’s note: this section will illustrate CGIF’s conformance to the Common Logic abstract syntax and semantics. Since much of CGIF’s recent development has been aimed for such conformance, the illustration should be straightforward. This serves two purposes: one, to “bless” the CGIF syntax/semantics with respect to Common Logic and two, to serve as a guide to users of other logic systems who seek to demonstrate their conformance to the standard.
Annex F
(informative)

Mapping of XML-based Common Logic (XCL) to Common Logic Abstract Syntax

EDITOR’s note: this section will illustrate XCL’s conformance to the Common Logic abstract syntax and semantics. Since much of XCL’s recent development has been aimed for such conformance, the illustration should be straightforward. This serves two purposes: one, to “bless” the XCL syntax/semantics with respect to Common Logic and two, to serve as a guide to users of other logic systems who seek to demonstrate their conformance to the standard.
Annex G
(informative)

Relationship of Common Logic to other logic standards and other logic formalisms

EDITOR's note: this section will describe Common Logic's relationship to other logic standards (most notably Prolog and Z) and distinguish its differences.
Bibliography