

The IFF Syntax*

Robert E. Kent

November 6, 2005

Contents

1	Syntax Summary	1
2	Lexical Syntax	3
3	Grammatical Syntax	6
3.1	IFF Syntactical Terms	9
3.2	IFF Syntactical Expressions	9
3.3	IFF Syntactical Modules	11
4	Code Examples	12
4.1	Meta-Ontology: IFF-CORE	12
4.2	Meta-Ontology: IFF-CAT	13
4.3	Namespace: Categories	13
4.4	Namespace: Classifications and Infomorphisms	17
4.5	Ontology: Webster's Dictionary	25

1 Syntax Summary

The metashell follows standard logical syntactic conventions. The particular syntactic primitives chosen are a compromise between a ‘minimal’ set of logical primitives and a ‘full’ set. They embody several abbreviated forms, such as conjunctions and disjunctions with any number of arguments, restricted quantification, guards and definite description. Most IFF expressions can be described as an operator applied to one or two operands. We list the basic syntactic forms here for convenience.

*We owe a special debt of thanks to the common logic group, and specifically to Pat Hayes, since we have borrowed from some of the boilerplate in the common logic documents. However, the IFF grammar is a grammar in its own right, and in some respects a much simpler grammar.

Category	Description	Form	Example
meta-name	primitive reference for meta-objects	a character string	abc-123, x_2
object-name	primitive reference for real-world-objects	a complex ascii string	'name', http://www.abc.com/ cgi-bin/def?ghi=ijkl &mn=op&qr=12&st=uv
term	denotes the value of a function applied to an argument	a function symbol applied to a single term	(f τ), (f [τ_0 τ_1])
relation	asserts that a relation holds between two arguments	a relation symbol applied to two terms	(r τ_0 τ_1)
equation	asserts that its two arguments are equal	the equality symbol applied to two terms	(= τ_0 τ_1)
conjunction	asserts all of its component expressions are true	the conjunction operator applied to a list of conjuncts	(and ϕ_0 ϕ_1 ... ϕ_n)
disjunction	asserts at least one of its component expressions are true	the disjunction operator applied to a list of conjuncts	(or ϕ_0 ϕ_1 ... ϕ_n)
negation	asserts that its contained expression is false	the negation operator applied to a single expression	(not ϕ)
implication	asserts that the antecedent implies the consequent	the implication operator applied to two expressions	(implies ϕ_0 ϕ_1), (=> ϕ_0 ϕ_1)
equivalence	asserts that the two component expressions imply each other	the equivalence operator applied to two expressions	(iff ϕ_0 ϕ_1), (<=> ϕ_0 ϕ_1)
existential quantifier	asserts that its body is true for some guarded interpretation of the variables in its restricted bound variable list	the existential operator followed by a list of restricted bound variables followed by a list of guards followed by the body expression	(exists ($x_0(A_0 x_0)$... γ_0 ...) ϕ)
universal quantifier	asserts that its body is true for all guarded interpretations of the variables in its restricted bound variable list	the universal operator followed by a list of restricted bound variables followed by a list of guards followed by the body expression	(forall ($x_0(A_0 x_0)$... γ_0 ...) ϕ)

The current expression of the IFF is encoded as a sequence of ascii characters. The syntax is presented here in two parts: the first is concerned with lexical analysis (as performed by the lexer), and the second is concerned with syntactic analysis (as performed by the parser). The lexer transforms a character stream consisting of a stream of ascii characters into a token stream consisting of a stream of lexical items. The lexical items in the character stream are usually separated by whitespace, and hence the lexer deals with whitespace handling. The parser transforms the token stream into a parse tree. This parse tree represents the logical syntax of the IFF code. Various semantic actions, such as initializing internal data structures, populating a database, or performing type checking, might be associated with the production rules of the grammar, and hence with the internal nodes of the parse tree.

The IFF syntax is written using Extended Backus-Naur Form (EBNF), as specified by International Standard ISO/IEC 14977¹. Literal characters are ‘quoted’ with bold quote marks, sequences of items are separated by bold commas **, . . .**, a bold bar **|** indicates alternatives, a pair of bold braces **{ }** indicates any sequence of expressions, a bold minus **-** indicates an exception, a pair of bold brackets **[]** indicates an optional item, and a pair of bold parentheses **()** is used for grouping. Productions are terminated with a bold semicolon **;**.

The IFF syntax distinguishes between names in the meta part of the IFF (meta-names) from names in the object part of the IFF (object-names). Meta-names are relatively simple in form and nature. Object-names are not.

- A meta-name references a meta-object. A meta-object might be a set, a function, a relation, a number, a simple name for a number, or a composite, such as a category, an infomorphism, etc. All of these meta-names are of the form of a letter followed by a letter, a number, a dash or an underscore. We must avoid a period, a colon or a dollar sign in an unqualified meta-name, as these are used in namespace prefixes.
- An object-name references a real world object. A real world object might be a person, a machine, a process, a quotation, a notation, a word, or address, in particular, a URI, etc. Quotations and words need surrounding (single or double) quote marks. URIs need special internal characters (other than a letter or a number) such as the slash, the colon, the dash, the underscore, the per-cent symbol, the question-mark, the ampersand symbol, the equality symbol, etc.

2 Lexical Syntax

- A *white* character is either a space character or a tab character or a line character or a page character or a return character. *Whitespace* is any positive amount of white characters.

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

- A *delimiter* is either a single quote, either open or closed (apostrophe), a double quote, an open or closed parenthesis, or an open or closed bracket. A single quote is used to delimit quoted strings, and double quote is used to delimit object-names. Single and double quotes obey special lexicalization rules. Quoted strings and object-names are the only IFF lexical items that can contain whitespace and delimiters. Parentheses and brackets outside quoted strings and object-names are self-delimiting; they are considered to be lexical tokens in their own right. In the IFF, parentheses are the general grouping mechanism, whereas brackets are the special grouping characters for tuples.

```
open = ‘(’ ;
close = ‘)’ ;
```

¹<http://www.cl.cam.ac.uk/%7emgk25/iso-14977.pdf>

```

opentuple = '[' ;
closetuple = ']' ;
openstringquote = '"' ;
closestringquote = '"' ;
namequote = '\"' ;

```

- A *letter* is any alphabetic symbol of either upper or lower case.

```

letter = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' |
'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' |
'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' |
'x' | 'y' | 'z' ;

```

- A *digit* is one of the symbols zero through nine.

```

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

```

- An *alphanumeric* is either a letter, a digit, a dash or an underscore.

```

alphanumeric = letter | digit | '-' | '_' ;

```

- A *nonalphanumeric* is all the remaining nonalphanumeric ascii characters, other than open parenthesis, closed parenthesis, single open and close quote or double quote.

```

nonalphanumeric = '~' | '!' | '#' | '$' | '%' | '^' | '&' | '*' | '+' | '{' | '}' | '|' | ':' |
'<' | '>' | '?' | '=' | '\ ' | ';' | ',' | '.' | '/' ;

```

- A *character* is any of the ascii characters, other than the delimiters. These 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.

```

character = alphanumeric | nonalphanumeric ;

```

- A *meta-name* is a name in the IFF that represents a meta-object. A meta-name is a letter followed by a list of alphanumeric characters.

```

metaname = letter , { alphanumeric } ;

```

- An *inner string quote* 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 inner string quote, i.e. when immediately preceded by a backslash character. The occurrence of the 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 object-names, double quotes are treated exactly the same. An *inner name quote* is used to indicate the presence of a double quote character inside an object-name.

```

openinnerstringquote = '\ ' ;
closeinnerstringquote = '\ ' ;
innernamequote = '\"' ;

```

- Single quotes are the delimiters for quoted strings. A *quoted string* is a special form in the IFF syntax, which denotes the text string inside the quotes, except that the combinations `\'` and `\'` indicate the presence of single quote marks in the denoted string. Any occurrence of the backslash character `\` not immediately followed by the characters `'` or `'` simply indicates the backslash character itself. With these conventions, a quoted string denotes the string enclosed in the quote marks. Quoted strings require a different lexicalization algorithm than other parts of IFF text (except object-names), since whitespace does not break a quoted text stream into lexical tokens.

```
quotedstring = openstringquote , { whitespace | open | close | opentuple | closetuple |
namequote | openinnerstringquote | closeinnerstringquote | character } ,
closestringquote ;
```

- An *object-name* is a name in the IFF that represents a real world object. Object-names in the IFF syntax are expressed as a sequence of ascii characters enclosed by double quotemarks; that is, double quotes are the delimiters for object-names. For example, IFF reserved words or URIs can be represented as object-names in the IFF by simply enclosing them by double quotemarks. Object-names also require a different lexicalization algorithm than other parts of IFF text (except quoted strings), since whitespace does not break a quoted text stream into lexical tokens.

```
objectname = namequote , { white | open | close | opentuple | closetuple |
openstringquote | closestringquote | innernamenamequote | character } , namequote ;
```

- A *reserved word* consists of the lexical tokens which are used to indicate the syntactic structure of the IFF.

```
reservedword = '=' | 'and' | 'or' | 'not' | 'implies' | '=>' | 'iff' | '<=>' | 'forall' | 'exists'
| 'iff:title' | 'iff:metalevel' | 'iff:name' | 'iff:comment' | 'iff:ontology' |
'iff:namespace' | 'iff:meta-ontology' ;
```

- A *number* is a list of digits of positive length.

```
number = digit , { digit } ;
```

- A *name* is either a meta-name or an object-name. Reserved words and numbers may not be used as names in IFF text.

```
name = ( metaname | objectname ) - ( reservedword | number ) ;
```

- A *numerical variable* is a name preceded by the pound sign symbol `#`.

```
numericalvariable = '#', name ;
```

- A *numerical expression* is either a number or a numerical variable or an application term consisting of an addition (or a subtraction) symbol with two arguments that are numerical expressions.

```
numericalexpression = number | numericalvariable | open , numericalexpression , (
'+', '-' ) , numericalexpression , close ;
```

- A *level number* is either a meta-name or a numerical expression.

levelnumber = metaname | numericalexpression ;

- A *namespace prefix* is a list of (namespace) meta-names separated by periods and preceded by a level number. The periods in a namespace prefix are both separators and concatenators. As a concatenator, the lefthand period can coerce its lefthand argument from a number into a string.

namespaceprefix = levelnumber , '.' , metaname , { '.' , metaname } ;

- A *qualified meta-name* is a meta-name preceded by a namespace prefix. The recommended separator is the colon symbol ':'. However, we also use the dollar symbol '\$' for backward compatibility.

qualifiedmetaname = namespaceprefix , (':' | '\$') , metaname ;

- A *qualified object-name* is an object-name preceded by a namespace prefix. The recommended separator is the colon symbol ':'. However, we also use the dollar symbol '\$' for backward compatibility.

qualifiedobjectname = namespaceprefix , (':' | '\$') , objectname ;

- A *qualified name* is either a qualified meta-name or a qualified object-name.

qualifiedname = qualifiedmetaname | qualifiedobjectname ;

- A *variable* is a meta-name preceded by the question mark symbol '?'.

variable = '?' , metaname ;

- A *constant* is a name or a qualified name.

constant = name | qualifiedname ;

Lexical tokens are divided into nine mutually disjoint categories: parentheses, brackets, reserved words, meta-names, quoted strings (which begin with ‘‘ and end with ‘’), object-names (which begin with ‘’ and end with ‘’ and numbers. Lexical tokens other than parentheses and brackets are separated by whitespace, parentheses or brackets. Thus, whitespace adjacent to a parenthesis or bracket is optional. The task of the lexer is to analyze a character stream, and to deliver the lexical tokens it finds, in order, to the parser.

3 Grammatical Syntax

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

Note 1 *Since there are two kinds of names, names for meta-things and names for object-things, so also there should be two kinds of terms, meta-terms and object-terms. However, for simplicity and expediency, the current version of the IFF grammar only specifies a combined kind of term, which by starting from a constant, incorporates both meta-names and object-names.*

Terms: Here are some examples of terms taken from the metashell and core axiomatizations. Terms can be either constants (names), variables or composites (neither constants nor variables).

Constant

- The following equation is used for typing the composition function in the generic core namespace, where we see that the first term in the equation is an application term consisting of the function symbol '`(#n+1).ftn:composition`' with the argument tuple '`[composition source]`' consisting of constant terms '`composition`' and '`source`'.

```
(= ((#n+1).ftn:composition [composition source])
   ((#n+1).ftn:composition [function0 source]))
```

Variable

- In the following axiom, which asserts reflexivity of the subset endorelation in the meta namespace, the variable term '`?X`' fills both arguments of the relational expression '`(subset ?X ?X)`'.

```
(forall (?X (set ?X))
 (subset ?X ?X))
```

Composite

- The following axiom states that the extent of a relation is a subset of the binary product of the domain and codomain of the relation: $\text{ext}(r) \subseteq X_0 \times X_1$. The term '`(product [(set0 ?r) (set1 ?r)])`' is composite.

```
(forall (?r (relation ?r))
 (subset (extent ?r) (product [(set0 ?r) (set1 ?r)])))
```

- The following equation is used for typing the subset endorelation in the meta namespace. The first term is the composite '`(type:base subset)`' and the second term is the constant '`set`'.

```
(= (type:base subset) set)
```

Position: Here are some examples of atoms taken from the metashell axiomatization, which show terms in set/predicate/function/relation position other than names; that is, variables or composite terms.

Set

- The following axiom defines the binary subset endorelation between pairs of sets in the type namespace of the metashell: in the subset relationship $X \subseteq Y$, all elements of the smaller type set X are contained in the larger type set Y . It uses (quantification over) variables '`?X`' in set position in the declaration '`(?X ?x)`'.

```
(forall (?X (set ?X) ?Y (set ?Y))
 (<=> (subset ?X ?Y)
      (forall (?x (?X ?x)) (?Y ?x))))
```

- The following axiom defines the intersection operation in the type namespace of the metashell: the intersection of any pair of sets X_0 and X_1 is (and hence contains) the overlap. It uses the composite term '`(intersection [?X0 ?X1])`' in set position.

```
(forall (?X0 (set ?X0) ?X1 (set ?X1) ?x (?X0 ?x) (?X1 ?x))
 ((intersection [?X0 ?X1] ?x))
```

- The following axiom defines the powerset construction in the meta namespace of the metashell: for any set X , the power metaset $\wp X$ consists of the set of all subsets of X . It uses the composite term '`(power ?X)`' in set position.

```
(forall (?X (set ?X) ?Y (set ?Y))
 (<=> ((power ?X) ?Y) (subset ?Y ?X)))
```

Predicate

- The following axiom defines the differentia of a predicate in the type namespace of the metashell: the differentia is the subset of the genus set consisting of those elements that satisfy the predicate $x \in \text{diff}(p)$ iff $p(x)$ for any element $x \in X$. It uses quantification over variables in predicate position.

```
(forall (?p (predicate ?p) ?x ((genus ?p) ?x))
 (<=> ((differentia ?p) ?x)
      (?p ?x)))
```

Function

- The following axiom asserts that all functions in the iff namespace of the metashell are total (defined on all source elements) and functional (have only one value). It uses (quantification over) variables ‘?f’ in function position in the term ‘(?f ?x)’.

```
(forall (?f (function ?f))
 (forall (?x ((source ?f) ?x))
  (and (exists (?y ((target ?f) ?y))
        (= (?f ?x) ?y))
       (exists (?y0 ((target ?f) ?y0) ?y1 ((target ?f) ?y1))
         (= (?f ?x) ?y0) (= (?f ?x) ?y1)
         (= ?y0 ?y1))))))
```

- The following axiom defines the binary restriction endorelation between pairs of functions in the type namespace of the metashell: in the restriction relationship $f_1 \sqsubseteq f_2$ between two functions $f_1 : X_1 \rightarrow Y_1$ and $f_2 : X_2 \rightarrow Y_2$, (1) the source (target) of f_1 is a subset of the source (target) of f_2 , $X_1 \subseteq X_2$ and $Y_1 \subseteq Y_2$, and (2) the functions agree (on source elements of X_1); that is, the functions commute with the source/target inclusions. It uses (quantification over) variables ‘?f1’ in function position in the term ‘(?f1 ?x1)’.

```
(forall (?f1 (function ?f1) ?f2 (function ?f2))
 (<=> (restriction ?f1 ?f2)
      (and (subset (source ?f1) (source ?f2))
            (subset (target ?f1) (target ?f2))
            (forall (?x1 ((source ?f1) ?x1))
              (= (?f1 ?x1) (?f2 ?x1))))))
```

- The following axiom defines the injective predicate in the meta namespace of the metashell: a function $f : A \rightarrow B$ is an injection when any image value is from a unique source element. It uses the non-constant, non-variable term ‘(type:differentia injective)’ in set position. It also uses (quantification over) variables ‘?f’ in function position in the term ‘(?f ?x0)’.

```
(forall (?f (function ?f))
 (<=> ((type:differentia injective) ?f)
      (forall (?x0 ((source ?f) ?x0) ?x1 ((source ?f) ?x1))
        (= (?f ?x0) (?f ?x1))
        (= ?x0 ?x1))))
```

Relation

- The following axiom defines the binary abridgment endorelation between pairs of relations in the type namespace of the metashell: in the abridgment relationship $r \preceq s$ between two relations $r : X_0 \rightarrow X_1$ and $s : Y_0 \rightarrow Y_1$, the domain X_0 and codomain X_1 of r are subsets of the domain Y_0 and codomain Y_1 of s , respectively, $X_0 \subseteq Y_0$ and $X_1 \subseteq Y_1$, and for all $x_0 \in X_0$ and $x_1 \in X_1$, $r(x_0, x_1)$ iff $s(x_0, x_1)$. It uses (quantification over) variables ‘?r’ in relation position in the relational expression ‘(?r ?x0 ?x1)’.

```
(forall (?r (relation ?r) ?s (relation ?s))
 (<=> (abridgment ?r ?s)
      (and (subset (set0 ?r) (set0 ?s))
            (subset (set1 ?r) (set1 ?s))
            (forall (?x0 ((set0 ?r) ?x0) ?x1 ((set1 ?r) ?x1))
              (<=> (?r ?x0 ?x1) (?s ?x0 ?x1))))))
```

- The following axiom defines the extent of a relation in the type namespace of the metashell: the extent is the subset of the binary product of the two component type sets $\text{ext}(r) \subseteq X_0 \times X_1$, consisting of those pairs that satisfy the relation $(x_0, x_1) \in \text{ext}(r)$ iff $r(x_0, x_1)$ for any elements $x_0 \in X_0$ and $x_1 \in X_1$. It uses quantification over variables in relation position.

```
(forall (?r (relation ?r) ?x0 (set0 ?r) ?x0) ?x1 (set1 ?r) ?x1)
 (<=> ((extent ?r) [?x0 ?x1])
      (?r ?x0 ?x1)))
```

3.1 IFF Syntactical Terms

- *Sets, predicates, functions and relations* can be expressed as terms.

set = term ;
predicate = term ;
function = term ;
relation = term ;

- A *term* is either a constant or a variable or an application term consisting of a function term with a single argument.

term = constant | variable | (open , function , argument , close) ;

- An *argument* is represented by a single term or a term list within brackets.

argument = term | (opentuple , { term } , closetuple) ;

3.2 IFF Syntactical Expressions

- An *declaration* is a set or predicate term followed by a term.

declaration = open , (set | predicate) , term , close ;

- An *equation* is an equality symbol followed by two terms.

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

- An *relational expression* is a relation term followed by two terms.

relationalexpression = open , relation , term , term , close ;

- An *atom* is either a declaration, an equation or a relational expression.

atom = declaration | equation | relationalexpression ;

- A *conjunction* is the conjunction symbol followed by a list of expressions.

conjunction = open , 'and' , { expression } , close ;

- A *disjunction* is the disjunction symbol followed by a list of expressions.

disjunction = open , 'or' , { expression } , close ;

- A *negation* is the negation symbol followed by a single expression.

negation = open , 'not' , expression , close ;

- An *implication* is the implication symbol (or its abbreviation) followed by two expressions, with the antecedent first and the consequent second.

implication = open , ('implies' | '=>') , expression , expression , close ;

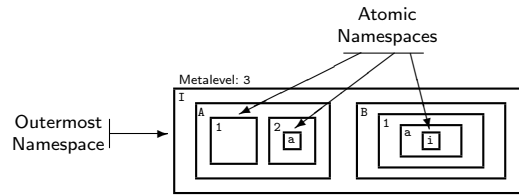


Figure 1: Nested Namespaces

- An *equivalence* is the equivalence symbol (or its abbreviation) followed by two expressions.

equivalence = open , ('iff' | '<=>') , expression , expression , close ;

- A *boolean expression* is either a conjunction, a disjunction, a negation, an implication or an equivalence.

booleanexpression = conjunction | disjunction | negation | implication | equivalence ;

- A *guard list* is any number of expressions (to be used as guards).

guardlist = { expression } ;

- In a *quantified expression*, the quantifiers (either universal or existential) may bind any number of variables and bound variables should be restricted to a named category. Guard expressions are allowed at the end of the list of bound variables.

boundvariablelist = { variable , open , set , variable , close } ;

quantifiedexpression = open , ('forall' | 'exists') , open , boundvariablelist , guardlist , close , expression , close ;

- In a *definite description*, the 'the' operator binds a variable restricted to a named category. Guard expressions are allowed after the bound variable.

definitedescription = open , 'the' , open , variable , open , set , variable , close , guardlist , close , expression , close ;

- A *expression* is either an atom, a boolean expression, a quantified expression or a definite description.

expression = atom | booleanexpression | quantifiedexpression | definitedescription ;

3.3 IFF Syntactical Modules

There are three kinds of modules in the IFF: ontologies, meta-ontologies and namespaces. An IFF ontology is an object-level module representing some aspect of the real world. Object-level IFF ontologies are created and maintained by using the facilities of some IFF metalanguage. An IFF meta-ontology is merely a collection of IFF namespaces. IFF meta-ontologies may overlap; that is, two meta-ontologies may share one or more namespaces. An IFF namespace is a named piece of text. IFF namespaces come in two kinds, either molecular or atomic. Molecular namespaces contain other (nested) namespaces, whereas atomic namespaces do not. Figure 1 illustrates namespace nesting: there are nine namespaces in total, consisting of three atomic and six molecular namespaces. Unlike physical atoms, namespace atoms can be located in only one molecule (molecular nesting). Each namespace, whether nested or not, must have a name, which contributes to its part of the full namespace prefix. The full namespace prefix for any particular namespace is its unique identifier; this is the concatenation of the metalevel number of the outermost namespace in the nesting, followed by the concatenation of each prefix name through the nesting, from the outermost to that particular namespace. The three atomic namespaces in Figure 1 have the full namespace prefixes (hence, unique identifiers) ‘3.I.A.1’, ‘3.I.A.2.a’ and ‘3.I.B.1.a.i’. The outermost namespace must have a title and a metalevel indicator (given by either a numerical expression or a meta-name). Each axiom in the natural part of the IFF (that is, not in the iff, type or meta namespaces) is precisely situated in a two-dimensional matrix with the metalevel number (0, 1, 2, \dots , ∞) along the vertical coordinate and the atomic namespace (containing the axiom) along the horizontal coordinate.

- A *title* is the ‘iff:title’ symbol followed by a quoted string.

```
title = open , ‘iff:title’ , quotedstring , close ;
```

- A *meta level* is the ‘iff:metalevel’ symbol followed by a level number.

```
metalevel = open , ‘iff:metalevel’ , levelnumber , close ;
```

- A *ontology name* is the ‘iff:name’ symbol followed by a name.

```
ontologyname = open , ‘iff:name’ , name , close ;
```

- A *namespace name* is the ‘iff:name’ symbol followed by a meta-name.

```
namespacename = open , ‘iff:name’ , metaname , close ;
```

- A *meta-ontology name* is the ‘iff:name’ symbol followed by a meta-name.

```
metaontologyname = open , ‘iff:name’ , metaname , close ;
```

- A *comment* is the ‘iff:comment’ symbol followed by a quoted string.

```
comment = open , ‘iff:comment’ , quotedstring , close ;
```

- A *ontology text* is a sequence of phrases, each being an atomic expression or a comment.

ontologytext = { atom | comment } ;

- A *namespace text* is a sequence of phrases, each being an expression, a comment or a nested namespace.

namespacetext = { expression | comment | nestednamespace } ;

- An *ontology* consists of the ‘iff:ontology’ symbol followed by an optional title followed by an ontology name followed by the text for the ontology.

ontology = open , ‘iff:ontology’ , [title] , ontologyname , ontologytext , close ;

- An *inner namespace* consists of the ‘iff:namespace’ symbol followed by an optional title followed by a namespace name followed by the text for the namespace. An inner namespace is not outermost in the nesting.

innernamespace = open , ‘iff:namespace’ , [title] , namespace , namespacetext , close ;

- An *outermost namespace* consists of the ‘iff:namespace’ symbol followed by an optional title followed by a namespace level number followed by a namespace name followed by the text for the namespace.

outermostnamespace = open , ‘iff:namespace’ , [title] , metalevel , namespace , namespacetext , close ;

- A *namespace entry* consists of the ‘iff:namespace’ symbol followed by a full namespace prefix.

namespaceentry = open , ‘iff:namespace’ , namespaceprefix , close ;

- A *meta-ontology* consists of the ‘iff:meta-ontology’ symbol followed by an optional title followed by a meta-ontology name followed by an optional comment followed by a list of the namespace entries, one for each outermost namespace included in that meta-ontology.

metaontology = open , ‘iff:meta-ontology’ , [title] , metaontologyname , [comment] , { namespaceentry } , close ;

- A *module* is either an ontology, a namespace or a meta-ontology.

module = ontology | innernamespace | outermostnamespace | metaontology ;

4 Code Examples

4.1 Meta-Ontology: IFF-CORE

Here is one code example of an IFF meta-ontology. The axiomatization represents the IFF Core (meta) Ontology (IFF-CORE). The name of the meta-ontology is ‘CORE’, hence its unique identifier is ‘IFF:CORE’.

```
(iff:meta-ontology
  (iff:title 'The IFF Core (meta) Ontology (IFF-CORE)')
  (iff:name CORE)
  (iff:comment 'The IFF Core (meta) Ontology (IFF-CORE) anchors the natural
part of the IFF. It consists of a kernel namespace for sets, functions and
relations, plus namespaces for diagrams, limits and exponents. In one sense
it represents the notion of a set-theoretic Cartesian-closed categories.
Eventually, it will represent the notion of a set-theoretic topos.')
  (iff:namespace #n.krnl) (iff:namespace ur.krnl)
  (iff:namespace #n.dgm) (iff:namespace ur.dgm)
  (iff:namespace #n.lim) (iff:namespace ur.lim)
  (iff:namespace #n.exp) (iff:namespace ur.exp)
)
```

4.2 Meta-Ontology: IFF-CAT

Here is another code example of an IFF meta-ontology. The axiomatization represents the IFF Category Theory (meta) Ontology (IFF-CAT). The name of the meta-ontology is 'CAT', hence its unique identifier is 'IFF:CAT'.

```
(iff:meta-ontology
  (iff:title 'The IFF Category Theory (meta) Ontology (IFF-CAT)')
  (iff:name CAT)
  (iff:comment 'The IFF Category Theory (meta) Ontology (IFF-CAT)
structures the natural part of the IFF. It consists of namespaces
for the basic concepts of category theory: categories, functors,
natural transformations, adjunctions, monads and Kan extensions.')
  (iff:namespace #n.cat) (iff:namespace ur.cat)
  (iff:namespace #n.func) (iff:namespace ur.func)
  (iff:namespace #n.nat) (iff:namespace ur.nat)
  (iff:namespace #n.adj) (iff:namespace ur.adj)
  (iff:namespace #n.mnd) (iff:namespace ur.mnd)
  (iff:namespace #n.kan) (iff:namespace ur.kan)
)
```

4.3 Namespace: Categories

Here is a code example for IFF namespaces. The axiomatization is generic (level n); it represents the most basic notion in category theory, that of a category. This namespace depends largely upon the sibling namespaces for graphs and graph morphisms. The name of the namespace is `cat`, hence its unique identifier is `#n.cat` (the variable `#n` needs to be bound to some number).

```
(iff:namespace
  (iff:title 'The Generic IFF Category Namespace')
  (iff:metalevel #n)
  (iff:name cat)
  (iff:comment 'This is the level n namespace for categories. It represents
level n categories - categories whose sets of objects and morphisms are
level n sets. The content of the namespace for categories is developed
in chapter 1 of the text "Category Theory for the Working Mathematician"
by Saunders Mac Lane (1971). All terms declared and axiomatized in this
namespace are listed in Table 1.')
)
```

(iff:comment 'A category C can be thought of as a special kind of graph |C| - a graph with monoidal properties (Figure 1). More precisely, a category C = (C, mu_C, eta_C) is a monoid in the 2-dimensional quasi-category Graph of (level n) graphs and graph morphisms. This means that it consists of an underlying graph |C|, a composition graph 2-cell mu_C : |C| o |C| --> |C| and an identity graph 2-cell eta_C : 1_obj(C) --> |C|, both with the identity object function id_obj(C). The 2-cell mu_C provides for a binary associative composition operation on morphisms in the category - its morphism function operates on any two morphisms that are composable, in the sense that the target object of the first is equal to the source object of the second, and returns a well-defined (composition) morphism. The 2-cell eta_C provides identities - its morphism function associates a well-defined (identity) morphism with each object in the category. Both mu and eta have the category C as a parameter. Categories are determined by their (graph, mu, eta) triples.')

```
((#n+1).set:set category)

((#n+1).ftn:function graph) ((#n+1).ftn:function underlying) (= underlying graph)
(= ((#n+1).ftn:source graph) category)
(= ((#n+1).ftn:target graph) #n.gph.obj:graph)

((#n+1).ftn:function mu)
(= ((#n+1).ftn:source mu) category)
(= ((#n+1).ftn:target mu) #n.gph.mor:2-cell)
(= ((#n+1).ftn:composition [mu #n.gph.mor:source])
    ((#n+1).ftn:composition
     [(((#n+1).rel:mediator #n.gph.obj:multipliable) [graph graph]
      #n.gph.obj:multiplication]))
    graph)
(= ((#n+1).ftn:composition [mu #n.gph.mor:target]) graph)

(KIF$function eta)
(= (KIF$source eta) category)
(= (KIF$target eta) #n.gph.mor:2-cell)
(= ((#n+1).ftn:composition [eta #n.gph.mor:source]) graph)
(= ((#n+1).ftn:composition [((#n+1).ftn:composition
    [graph #n.gph.obj:object]) #n.gph.obj:unit]))
(= ((#n+1).ftn:composition [eta #n.gph.mor:target]) graph)

(forall (?C1 (category ?C1) ?C2 (category ?C2))
  (=> (and (= (graph ?C1) (graph ?C2))
        (= (mu ?C1) (mu ?C2))
        (= (eta ?C1) (eta ?C2)))
    (= ?C1 ?C2)))
```

(iff:comment 'For convenience in the language used for categories, we rename the object and morphism classes, and the source and target functions in the setting of categories. These terms refer to the underlying graph of a category.')

```
((#n+1).ftn:function object)
(= ((#n+1).ftn:source object) category)
(= ((#n+1).ftn:target object) #n.set:set)
(= object ((#n+1).ftn:composition [graph #n.gph.obj:object]))

((#n+1).ftn:function morphism)
(= ((#n+1).ftn:source morphism) category)
(= ((#n+1).ftn:target morphism) #n.set:set)
(= morphism ((#n+1).ftn:composition [graph #n.gph.obj:morphism]))
```

```

((#n+1).ftn:function source)
(= ((#n+1).ftn:source source) category)
(= ((#n+1).ftn:target source) #n.ftn:function)
(= source ((#n+1).ftn:composition [graph #n.gph.obj:source]))

((#n+1).ftn:function target)
(= ((#n+1).ftn:source target) category)
(= ((#n+1).ftn:target target) #n.ftn:function)
(= target ((#n+1).ftn:composition [graph #n.gph.obj:target]))

(iff:comment 'For convenience in the language used for categories, we define
the relation of composable pairs of morphisms, and rename the zeroth and first
morphism functions in the setting of categories. These terms refer to the
horizontal multiplication of the underlying graph of a category with itself.

((#n+1).ftn:function composable-opspan)
(= ((#n+1).ftn:source composable-opspan) category)
(= ((#n+1).ftn:target composable-opspan) #n.dgm.ospn:opspan)
(= ((#n+1).ftn:composition [mu #n.gph.mor:source])
    ((#n+1).ftn:composition
    [((#n+1).rel:mediator #n.gph.obj:multipliable) [graph graph]
    #n.gph.obj:multiplication-opspan]))

((#n+1).ftn:function composable)
(= ((#n+1).ftn:source composable) category)
(= ((#n+1).ftn:target composable) REL$relation)
(= ((#n+1).ftn:composition [composable #n.rel.endo:base]) morphism)
(= composable
    ((#n+1).ftn:composition [composable-opspan #n.dgm.ospn:relation]))

((#n+1).ftn:function morphism-morphism)
(= ((#n+1).ftn:source morphism-morphism) category)
(= ((#n+1).ftn:target morphism-morphism) #n.set:set)
(= morphism-morphism
    ((#n+1).ftn:composition [composable-opspan #n.lim.pbk:pullback]))

((#n+1).ftn:function morphism0)
(= ((#n+1).ftn:source morphism0) category)
(= ((#n+1).ftn:target morphism0) #n.ftn:function)
(= ((#n+1).ftn:composition [morphism0 #n.ftn:source]) morphism-morphism)
(= ((#n+1).ftn:composition [morphism0 #n.ftn:target]) morphism)
(= morphism0
    ((#n+1).ftn:composition [composable-opspan #n.lim.pbk:projection0]))

((#n+1).ftn:function morphism1)
(= ((#n+1).ftn:source morphism1) category)
(= ((#n+1).ftn:target morphism1) #n.ftn:function)
(= ((#n+1).ftn:composition [morphism1 #n.ftn:source]) morphism-morphism)
(= ((#n+1).ftn:composition [morphism1 #n.ftn:target]) morphism)
(= morphism1
    ((#n+1).ftn:composition [composable-opspan #n.lim.pbk:projection1]))

(iff:comment 'The composition function provides for a binary associative
operation on morphisms in the category - it operates on any two morphisms that
are composable, in the sense that the target object of the first is equal to
the source object of the second, and returns a well-defined (composition)

```

morphism. The identity function provides identities - it associates a well-defined (identity) morphism with each object in the category. Both composition and identity have the category as a parameter. Categories are determined by their (underlying, mu, eta) triples, and hence by their (underlying, composition, identity) triples. By the definitions of graph morphisms, graph multiplication and graph units, for any category C, these operations satisfy the typing constraints listed in the Table:

```
o_C src_C = 0th_C src_C
o_C tgt_C = 1st_C tgt_C
1_C src_C = id_obj(C) = id_C tgt_C'
```

```
((#n+1).ftn:function composition)
(= ((#n+1).ftn:source composition) category)
(= ((#n+1).ftn:target composition) #n.ftn:function)
(= ((#n+1).ftn:composition [composition #n.ftn:source]) morphism-morphism)
(= ((#n+1).ftn:composition [composition #n.ftn:target]) morphism)
(= ((#n+1).ftn:composition
    [((#n+1).rel:mediator #n.ftn:composable) [composition source]
     #n.ftn:composition])
    ((#n+1).ftn:composition
    [((#n+1).rel:mediator #n.ftn:composable) [morphism0 source]
     #n.ftn:composition]))
(= ((#n+1).ftn:composition
    [((#n+1).rel:mediator #n.ftn:composable) [composition target]
     #n.ftn:composition])
    ((#n+1).ftn:composition
    [((#n+1).rel:mediator #n.ftn:composable) [morphism1 target]
     #n.ftn:composition]))
(= composition ((#n+1).ftn:composition [mu #n.gph.mor:morphism]))

((#n+1).ftn:function identity)
(= ((#n+1).ftn:source identity) category)
(= ((#n+1).ftn:target identity) #n.ftn:function)
(= ((#n+1).ftn:composition
    [((#n+1).rel:mediator #n.ftn:composable) [identity source]
     #n.ftn:composition])
    ((#n+1).ftn:composition [object #n.ftn:identity]))
(= ((#n+1).ftn:composition
    [((#n+1).rel:mediator #n.ftn:composable) [identity target]
     #n.ftn:composition])
    ((#n+1).ftn:composition [object #n.ftn:identity]))
(= identity ((#n+1).ftn:composition [eta #n.gph.mor:morphism]))
```

(iff:comment 'To each category C, there is an opposite category C^{op} = (C^{op}, tau_C,C mu_C^{op}, eta_C^{op}). Since all categorical notions have their duals, the opposite category can be used to decrease the size of the axiom set. The objects of C^{op} are the objects of C, and the morphisms of C^{op} are the morphisms of C. However, the source and target of a morphism are reversed: src_C^{op}(m) = tgt_C(m) and tgt_C^{op}(m) = src_C(m). The composition is defined by m2 o^{op} m1 = m1 o m2, and the identity is 1^{op} = 1.'

```
((#n+1).ftn:function opposite)
(= ((#n+1).ftn:source opposite) category)
(= ((#n+1).ftn:target opposite) category)
(= ((#n+1).ftn:composition [opposite graph])
    ((#n+1).ftn:composition [graph #n.gph.obj:opposite]))
(= ((#n+1).ftn:composition [opposite eta])
```

```

((#n+1).ftn:composition [eta #n.gph.mor:opposite]))
(= ((#n+1).ftn:composition [opposite mu])
  ((#n+1).ftn:composition
    [(((#n+1).rel:mediator #n.gph.mor:composable)
      [((#n+1).ftn:composition
        [(((#n+1).rel:mediator #n.gph.obj:multipliable) [graph graph])
          #n.gph.mor:tau])
        ((#n+1).ftn:composition [mu #n.gph.mor:opposite])])])
      #n.gph.mor:composition]))
)

```

4.4 Namespace: Classifications and Infomorphisms

Here is a code example for IFF namespaces. The axiomatization is generic (level n); it represents a somewhat reduced axiomatization for the set-theoretically n^{th} level category Clsn_n , the category of classifications and infomorphisms. The name of the namespace is `clsn`, hence its unique identifier is `#n.clsn` (the variable `#n` needs to be bound to some number).

The level 1 category Clsn_1 is the basic category defined and used in the theory of Information Flow [1]. Like many of the generic modules of the IFF, it contains one outermost molecular namespace `#n.clsn` containing axiomatizations for the categories, functors, natural transformations and adjunctions that center around Clsn_n , plus two atomic nested namespaces: `#n.clsn.obj` for classifications (the objects of Clsn_n) and `#n.clsn.mor` for infomorphisms (the morphisms of Clsn_n).

```

(iff:namespace
  (iff:title 'The Generic IFF Classification Namespace')
  (iff:metalevel #n)
  (iff:name clsn)
  (iff:comment 'This is the level n namespace for classifications and
infomorphisms. The terms introduced in this namespace are listed in Table 1.')

  (iff:comment 'There is a level n+1 category of classifications Clsn. The object
level n set of Clsn is the level n set of all classifications. The morphism
level n set of Clsn is the level n set of all infomorphisms. Composition in
Clsn is composition of infomorphisms.')

  ((#n+1).cat:category Classification)
  (= ((#n+1).cat:object language) #n.clsn.obj:classification)
  (= ((#n+1).cat:morphism language) #n.clsn.mor:infomorphism)
  (= ((#n+1).cat:source language) #n.clsn.mor:source)
  (= ((#n+1).cat:target language) #n.clsn.mor:target)
  (= ((#n+1).cat:composable language) #n.clsn.mor:composable)
  (= ((#n+1).cat:composition language) #n.clsn.mor:composition)
  (= ((#n+1).cat:identity language) #n.clsn.mor:identity)

  (iff:comment 'There are instance and type projection functors from the
category Clsn to the category Set.')

  ((#n+1).func:functor instance)
  (= ((#n+1).func:source instance) Classification)
  (= ((#n+1).func:target instance) #n.set:Set)
  (= ((#n+1).func:object instance) #n.clsn.obj:instance)

```

```

(= ((#n+1).func:morphism instance) #n.clsn.mor:instance)

((#n+1).func:functor type)
(= ((#n+1).func:source type) Classification)
(= ((#n+1).func:target type) #n.set:Set)
(= ((#n+1).func:object type) #n.clsn.obj:type)
(= ((#n+1).func:morphism type) #n.clsn.mor:type)

(iff:comment 'There is a contravariant instance power functor from the
category Set to the category Clsn, and there is a covariant type power
functor from the category Set to the category Clsn.')
```

```

((#n+1).func:functor instance-power)
(= ((#n+1).func:source instance-power) ((#n+1).cat:opposite #n.set:Set))
(= ((#n+1).func:target instance-power) Classification)
(= ((#n+1).func:object instance-power) #n.clsn.obj:instance-power)
(= ((#n+1).func:morphism instance-power) #n.clsn.mor:instance-power)

((#n+1).func:functor type-power)
(= ((#n+1).func:source type-power) #n.set:Set)
(= ((#n+1).func:target type-power) Classification)
(= ((#n+1).func:object type-power) #n.clsn.obj:type-power)
(= ((#n+1).func:morphism type-power) #n.clsn.mor:type-power)

(iff:comment 'The composition of instance power followed by (the
opposite of) instance is the identity functor on Set.')
```

```

(= ((#n+1).func:composition
    [instance-power ((#n+1).func:opposite instance)])
    ((#n+1).func:identity ((#n+1).cat:opposite #n.set:Set)))

(iff:comment 'There is a natural transformation whose source (and target)
category is Clsn, whose source functor is the identity functor on Clsn,
whose target functor is the composition of instance followed by instance
power, and whose A-th component infomorphism for classification A is
the extent infomorphism of A.')
```

```

((#n+1).nat:natural-transformation extent)
(= ((#n+1).nat:source-category extent) Classification)
(= ((#n+1).nat:target-category extent) Classification)
(= ((#n+1).nat:source-functor extent)
    ((#n+1).func:identity language))
(= ((#n+1).nat:target-functor extent)
    ((#n+1).func:composition
    [((#n+1).func:opposite instance) instance-power]))
(= ((#n+1).nat:component extent) #n.clsn.mor:extent)

(iff:comment 'There is an adjunction (actually, a reflection) between
classifications and sets, where the instance functor is the left adjoint,
the instance power functor is the right adjoint, the unit natural
transformation is the extent natural transformation, and the counit
natural transformation is the identity.')
```

```

((#n+1).adj:adjunction reflection)
(= ((#n+1).adj:underlying-category reflection) Classification)
(= ((#n+1).adj:free-category reflection)
    ((#n+1).cat:opposite #n.set:Set))

```

```

(= ((#n+1).adj:left-adjoint reflection)
  ((#n+1).func:opposite instance))
(= ((#n+1).adj:right-adjoint reflection) instance-power)
(= ((#n+1).adj:unit reflection) extent)
(= ((#n+1).adj:counit reflection)
  ((#n+1).nat:identity
    ((#n+1).func:identity ((#n+1).cat:opposite #n.set:Set))))

(iff:namespace
  (iff:name obj)
  (iff:comment 'A level n classification A = (inst(A), typ(A), |=A) (Figure 1)
    is identical to a level n binary relation. However, from a category-theoretic
    standpoint, the context of classifications is very different from the context
    of relations, since their morphisms are very different. A level n classification
    consists of a level n set of instances inst(A) identified with the domain or
    zeroth set of a binary relation, a level n set of types typ(A) identified with
    the codomain or first set of a binary relation, and a level n set of incidence
    or classification |=A identified with the extent set of a binary relation. The
    following is an IFF representation for the elements of a classification. The
    elements in the IFF representation are useful for the specification of a
    classification by declaration and population. The term \'classification\'
    allows one to declare classifications. The terms \'instance\', \'type\' and
    \'incidence\' resolve classifications into their parts, thus allowing one to
    populate classifications.')

  ((#n+1).set:set classification)
  (= classification #n.rel:relation)

  ((#n+1).ftn:function instance)
  (= ((#n+1).ftn:source instance) classification)
  (= ((#n+1).ftn:target instance) #n.set:set)
  (= instance #n.rel:set0)

  ((#n+1).ftn:function type)
  (= ((#n+1).ftn:source type) classification)
  (= ((#n+1).ftn:target type) #n.set:set)
  (= type #n.rel:set1)

  ((#n+1).ftn:function incidence)
  (= ((#n+1).ftn:source incidence) classification)
  (= ((#n+1).ftn:target incidence) #n.set:set)
  (= incidence #n.rel:extent)

  (iff:comment 'Associated with any classification A = (inst(A), typ(A), |=A)
    is a function int(A) : inst(A) --> pow(typ(A)) that produces the intent of
    an instance and a function ext(A) : typ(A) --> pow(inst(A)) that produces
    the extent of a type, both within the context of the classification.
    The intent of an instance i in inst(A) is defined by
      int(A)(i) = {t in typ(A) | i |=A t}.
    The extent of a type t in typ(A) is defined by
      ext(A)(t) = {i in inst(A) | i |=A t}.
    Intent and extent are synonymous with relational fibers (01 and 10, respectively).
    The following axioms specify the intent and extent functions.')

  ((#n+1).ftn:function intent)
  (= ((#n+1).ftn:source intent) classification)
  (= ((#n+1).ftn:target intent) #n.ftn:function)

```

```

(= intent #n.rel:fiber01)

((#n+1).ftn:function extent)
(= ((#n+1).ftn:source extent) classification)
(= ((#n+1).ftn:target extent) #n.ftn:function)
(= extent #n.rel:fiber10)

(iff:comment 'These axioms demonstrate that the relative instantiation-predication
represented by the incidence relation is compatible with, generalizes and
relativizes the absolute instantiation-predication - an instance is a member of
the extent of a type (or dually, a type is a member of the intent of an instance)
iff the instance is classified by the type.')
```

```

(= ((#n+1).ftn:composition [extent #n.ftn:source]) type)
(= ((#n+1).ftn:composition [extent #n.ftn:target])
  ((#n+1).ftn:composition [instance #n.set:power]))
(= ((#n+1).ftn:composition [intent #n.ftn:source]) instance)
(= ((#n+1).ftn:composition [intent #n.ftn:target])
  ((#n+1).ftn:composition [type #n.set:power]))
(forall (?A (classification ?A)
  ?i ((instance ?A) ?i) ?t ((type ?A) ?t))
  (and (<=> ((extent ?A) ?t) ?i
    (?A ?i ?t)))
  (<=> ((intent ?A) ?i) ?t
    (?A ?i ?t))))
```

```

(iff:comment 'For any classification A = (inst(A), typ(A), |=A), two instances
i1, i2 in inst(A) are indistinguishable in A (Barwise and Seligman, 1997),
written symbolically as i1 ~A i2, when int(A)(i1) = int(A)(i2). Two types
t1, t2 in typ(A) are coextensive in A, written symbolically as t1 ~A t2, when
ext(A)(t1) = ext(A)(t2). A classification A is separated when there are no two
distinct but indistinguishable instances, and extensional when there are no
distinct coextensive types. The terms \indistinguishable ?a\ and
\coextensive ?a\ represent the Information Flow notions of instance
indistinguishability and type coextension, respectively. The terms \separated\
and \extensional\ represent the Information Flow notions of classification
separateness and extensionality, respectively.')
```

```

((#n+1).ftn:function indistinguishable)
(= ((#n+1).ftn:source indistinguishable) classification)
(= ((#n+1).ftn:target indistinguishable) #n.rel.endo:equivalence-relation)
(= ((#n+1).ftn:composition [indistinguishable #n.rel.endo:base]) instance)
(forall (?A (classification ?A)
  ?i1 ((instance ?A) ?i1)
  ?i2 ((instance ?A) ?i2))
  (<=> ((indistinguishable ?A) ?i1 ?i2)
    (= ((intent ?A) ?i1)
      ((intent ?A) ?i2))))
```

```

((#n+1).ftn:function coextensive)
(= ((#n+1).ftn:source coextensive) classification)
(= ((#n+1).ftn:target coextensive) #n.rel.endo:equivalence-relation)
(= ((#n+1).ftn:composition [coextensive #n.rel.endo:base]) type)
(forall (?A (classification ?A)
  ?t1 ((type ?a) ?t1)
  ?t2 ((type ?a) ?t2))
  (<=> ((coextensive ?a) ?t1 ?t2)
```

```

(= ((extent ?a) ?t1)
    ((extent ?a) ?t2)))

((#n+1).dgm.ppr.obj:parallel-pair separated-parallel-pair)
(= ((#n+1).dgm.ppr.obj:source separated-parallel-pair) classification)
(= ((#n+1).dgm.ppr.obj:target separated-parallel-pair)
    #n.rel.endo:equivalence-relation)
(= ((#n+1).dgm.ppr.obj:function0 separated-parallel-pair) indistinguishable)
(= ((#n+1).dgm.ppr.obj:function1 separated-parallel-pair)
    (#n.ftn:composition [instance #n.rel.endo:identity]))

((#n+1).set:set separated)
((#n+1).set:subset separated classification)
(= separated ((#n+1).lim.equ.obj:equalizer separated-parallel-pair))

((#n+1).dgm.ppr.obj:parallel-pair extensional-parallel-pair)
(= ((#n+1).dgm.ppr.obj:source extensional-parallel-pair) classification)
(= ((#n+1).dgm.ppr.obj:target extensional-parallel-pair)
    #n.rel.endo:equivalence-relation)
(= ((#n+1).dgm.ppr.obj:function0 extensional-parallel-pair) coextensive)
(= ((#n+1).dgm.ppr.obj:function1 extensional-parallel-pair)
    (#n.ftn:composition [type #n.rel.endo:identity]))

((#n+1).set:set extensional)
((#n+1).set:subset extensional classification)
(= extensional ((#n+1).lim.equ.obj:equalizer extensional-parallel-pair))

(iff:comment 'To quote (Barwise and Seligman, 1997), "in any classification,
we think of the types as classifying the instances, but it is often useful
to think of the instances as classifying the types." For any classification
 $A = (\text{inst}(A), \text{typ}(A), |=A)$ , the opposite or dual of  $A$  is the opposite binary
relation; that is, the classification  $A^{\text{op}} = (\text{typ}(A), \text{inst}(A), |=A^{\text{op}})$ ,
whose instances are types of  $A$ , whose types are instances of  $A$ , and whose
incidence is:  $t |=A^{\text{op}} i$  when  $i |=A t$ .')

((#n+1).ftn:function opposite)
(= ((#n+1).ftn:source opposite) classification)
(= ((#n+1).ftn:target opposite) classification)
(= opposite #n.rel:opposite)

(iff:comment 'For any set  $A$  the instance power classification
 $\text{inst-pow}(A) = (A, \text{pow}(A), \text{in}(A))$  over  $A$  is defined as follows:
the instance set is  $A$ ; the type set is the power set  $\text{pow}(A)$ 
(so that a type is a subset of  $A$ ), and incidence is the
membership relation  $\text{in}(A)$ .

((#n+1).ftn:function instance-power)
(= ((#n+1).ftn:source instance-power) #n.set:set)
(= ((#n+1).ftn:target instance-power) classification)
(= ((#n+1).ftn:composition [instance-power instance]
    ((#n+1).ftn:identity #n.set:set)
    (#n+1).ftn:composition [instance-power type]) #n.set:power)
(forall (?A (#n.set:set ?A)
    ?x (?A ?x)
    ?Y ((#n.set:power ?A) ?Y))
    (<=> ((instance-power ?A) ?x ?Y)
        (?Y ?x)))

```

```

)

(iff:namespace
  (iff:name mor)
  (iff:comment 'Classifications are related through infomorphisms.
  An infomorphism  $f : A \rightarrow B$  from classification A to classification B
  (Figure 10) consists of a pair  $f = (inst(f), typ(f))$  of oppositely
  directed functions,
    - a function between instances
       $inst(f) : inst(B) \rightarrow inst(A)$  and
    - a function between types
       $typ(f) : typ(A) \rightarrow typ(B)$ ,
  which satisfy the fundamental property:
     $inst(f)(i) \models_A t$  iff  $i \models_B typ(f)(t)$ 
  for all instances  $i$  in  $inst(B)$  and all types  $t$  in  $typ(A)$ . The following
  is an IFF representation for the elements of an infomorphism. Such elements
  are useful for the definition of an infomorphism. In the same fashion as
  classifications, infomorphisms are specified by declaration and population.
  The term \‘infomorphism\’ allows one to declare infomorphisms themselves, and
  the two terms \‘source\’ and \‘target\’ allow one to declare their associated
  source and target classifications, respectively. The terms \‘instance\’ and
  \‘type\’ resolve infomorphisms into their parts, thus allowing one to populate
  infomorphisms. Let  $Clsn$  denote the category of classifications and infomorphisms.’)

  ((#n+1).set:set infomorphism)

  ((#n+1).ftn:function source)
  (= ((#n+1).ftn:source source) infomorphism)
  (= ((#n+1).ftn:target source) #n.clsn.obj:classification)

  ((#n+1).ftn:function target)
  (= ((#n+1).ftn:source target) infomorphism)
  (= ((#n+1).ftn:target target) #n.clsn.obj:classification)

  ((#n+1).ftn:function instance)
  (= ((#n+1).ftn:source instance) infomorphism)
  (= ((#n+1).ftn:target instance) #n.ftn:function)
  (= ((#n+1).ftn:composition [instance #n.ftn:source])
      ((#n+1).ftn:composition [target #n.clsn.obj:instance]))
  (= ((#n+1).ftn:composition [instance #n.ftn:target])
      ((#n+1).ftn:composition [source #n.clsn.obj:instance]))

  ((#n+1).ftn:function type)
  (= ((#n+1).ftn:source type) infomorphism)
  (= ((#n+1).ftn:target type) SET.FTN$function)
  (= ((#n+1).ftn:composition [type #n.ftn:source])
      ((#n+1).ftn:composition [source #n.clsn.obj:type]))
  (= ((#n+1).ftn:composition [type #n.ftn:target])
      ((#n+1).ftn:composition [target #n.clsn.obj:type]))

  (forall (?f (infomorphism ?f)
    ?i ((#n.clsn.obj:instance (target ?f)) ?i)
    ?t ((#n.clsn.obj:type (source ?f)) ?t))
    (<=> ((source ?f) ((instance ?f) ?i) ?t)
      ((target ?f) ?i ((type ?f) ?t))))

  (iff:comment 'The composition function operates on any two infomorphisms

```

that are composable in the sense that the target classification of the first is equal to the source classification of the second. Composition produces an infomorphism, whose instance function is the composition of the instance functions of the components and whose type function is the composition of the type functions of the components.’)

```

((#n+1).dgm.opsn.obj:opspan composable-opspan)
(= ((#n+1).dgm.opsn.obj:opzeroth composable-opspan) target)
(= ((#n+1).dgm.opsn.obj:opfirst composable-opspan) source)
(= ((#n+1).dgm.opsn.obj:opvertex composable-opspan) #n.clsn.obj:classification)

((#n+1).rel.endo:endo:relation composable)
(= ((#n+1).rel.endo:base composable) infomorphism)
(= ((#n+1).rel.endo:extent composable)
    ((#n+1).lim.pbk.obj:pullback composable-opspan))

((#n+1).set:set infomorphism-infomorphism)
(= infomorphism-infomorphism)
    ((#n+1).lim.pbk.obj:pullback composable-opspan))

((#n+1).ftn:function infomorphism0)
(= ((#n+1).ftn:source infomorphism0) infomorphism-infomorphism)
(= ((#n+1).ftn:target infomorphism0) infomorphism)
(= infomorphism0 ((#n+1).lim.pbk.obj:projection0 composable-opspan))

((#n+1).ftn:function infomorphism1)
(= ((#n+1).ftn:source infomorphism1) infomorphism-infomorphism)
(= ((#n+1).ftn:target infomorphism1) infomorphism)
(= infomorphism1 ((#n+1).lim.pbk.obj:projection1 composable-opspan))

((#n+1).ftn:function composition)
(= ((#n+1).ftn:source composition) infomorphism-infomorphism)
(= ((#n+1).ftn:target composition) infomorphism)
(= ((#n+1).ftn:composition [composition source])
    ((#n+1).ftn:composition [infomorphism0 source]))
(= ((#n+1).ftn:composition [composition target])
    ((#n+1).ftn:composition [infomorphism1 target]))
(= ((#n+1).ftn:composition [composition instance])
    ((#n+1).rel:mediator composable)
    [((#n+1).ftn:composition [infomorphism1 instance])
     ((#n+1).ftn:composition [infomorphism0 instance])]))
(= ((#n+1).ftn:composition [composition type])
    ((#n+1).rel:mediator composable)
    [((#n+1).ftn:composition [infomorphism0 instance])
     ((#n+1).ftn:composition [infomorphism1 instance])]))

(iff:comment ‘The identity function associates a well-defined identity
infomorphism with any classification, whose instance function is the identity
set function on instances and whose type function is the identity set function
on types.’)

((#n+1).ftn:function identity)
(= ((#n+1).ftn:source identity) #n.clsn.obj:classification)
(= ((#n+1).ftn:target identity) infomorphism)
(= ((#n+1).ftn:composition [identity source])
    ((#n+1).ftn:identity #n.clsn.obj:classification))
(= ((#n+1).ftn:composition [identity target])

```

```

((#n+1).ftn:identity #n.clsn.obj:classification))
(= ((#n+1).ftn:composition [identity instance])
  ((#n+1).ftn:composition [#n.clsn.obj:instance #n.ftn:identity]))
(= ((#n+1).ftn:composition [identity type])
  ((#n+1).ftn:composition [#n.clsn.obj:type #n.ftn:identity]))

(iff:comment 'Duality can be extended to infomorphisms. For any infomorphism
f : A --> B, the opposite or dual of f is the infomorphism f^op : B^op --> A^op
whose source classification is the opposite of the target classification of f,
whose target classification is the opposite of the source classification of f,
whose instance function is the type function of f, whose type function is the
instance function of f, and whose fundamental condition is equivalent to that
of f:
  typ(f )(t) |=^op i iff t |=^op inst(f)(i).')

(#n+1).ftn:function opposite)
(= ((#n+1).ftn:source opposite) infomorphism)
(= ((#n+1).ftn:target opposite) infomorphism)
(= ((#n+1).ftn:composition [opposite source])
  ((#n+1).ftn:composition [target #n.clsn.obj:opposite]))
(= ((#n+1).ftn:composition [opposite target])
  ((#n+1).ftn:composition [source #n.clsn.obj:opposite]))
(= ((#n+1).ftn:composition [opposite instance]) type)
(= ((#n+1).ftn:composition [opposite type]) instance)

(iff:comment 'For any set function f : B --> A the components of the instance
power infomorphism inst-pow(f) : inst-pow(A) --> inst-pow(B) over f (Figure 11)
are defined as follows: the source classification inst-pow(A) = (A,pow(A),in(A))
is the instance power classification over the target set A, the target
classification inst-pow(B) = (B,pow(B),in(B)) is the instance power classification
over the source set B, the instance function is f, and the type function is the
inverse image function f^-1 : pow(A) --> pow(B) from the powerset of A to the
powerset of B. Note the contravariance.')

(#n+1).ftn:function instance-power)
(= ((#n+1).ftn:source instance-power) SET.FTN$function)
(= ((#n+1).ftn:target instance-power) infomorphism)
(= ((#n+1).ftn:composition [instance-power source])
  ((#n+1).ftn:composition [#n.ftn:target #n.clsn.obj:power]))
(= ((#n+1).ftn:composition [instance-power target])
  ((#n+1).ftn:composition [#n.ftn:source #n.clsn.obj:power]))
(= ((#n+1).ftn:composition [instance-power instance])
  ((#n+1).ftn:identity #n.ftn:function))
(= ((#n+1).ftn:composition [instance-power type] #n.ftn:inverse-image)

(iff:comment 'It is a standard fact in Information Flow that from any
classification A there is a canonical extent infomorphism
h(A) : A --> inst-pow(inst(A)) (Figure 12) from A to the instance power
classification inst-pow(inst(A)) = (inst(A), pow(inst(A)), in(inst(A))) over
the instance set. This infomorphism is the A-th component of a natural
transformation called extent, the unit of the adjunction between the underlying
instance functor and the instance power functor. The instance function of extent
is the identity function on the instance set inst(A), and the type function of
extent is the extent function ext(A) : typ(A) --> pow(inst(A)).')

(#n+1).ftn:function extent)
(= ((#n+1).ftn:source extent) #n.clsn.obj:classification)

```

```

(= ((#n+1).ftn:target extent) infomorphism)
(= ((#n+1).ftn:composition [extent source])
    ((#n+1).ftn:identity #n.clsn.obj:classification))
(= ((#n+1).ftn:composition [extent target])
    ((#n+1).ftn:composition [#n.clsn.obj:instance #n.clsn.obj:instance-power]))
(= ((#n+1).ftn:composition [extent instance])
    ((#n+1).ftn:composition [#n.clsn.obj:instance #n.ftn:identity]))
(= ((#n+1).ftn:composition [extent type]) #n.clsn.obj:extent)
)
)

```

4.5 Ontology: Webster's Dictionary

Here is a code example for IFF ontologies. The axiomatization is object level (level 0); it represents Example 4.3 (page 70) in the text *Information Flow: The Logic of Distributed Systems* by Job Barwise and Jerry Seligman [1]. The name of the ontology is `webster`, hence its unique identifier is `0.webster`. This ontology is dependent upon the terminology and axiomatization of the level 1 namespace `'1.clsn.obj'` for classifications.

```

(iff:ontology
  (iff:title 'The Webster's Dictionary Classification')
  (iff:name webster)
  (iff:comment 'We can represent the classification of English words according to
    parts of speech as given in Webster's Dictionary using a classification table. We
    only give a small part of this table representing the classification, of course.')
```

```

  (1.clsn.obj:classification Webster)
  ((Webster 1.clsn.obj:instance) "bet")
  ((Webster 1.clsn.obj:instance) "eat")
  ((Webster 1.clsn.obj:instance) "fit")
  ((Webster 1.clsn.obj:instance) "friend")
  ((Webster 1.clsn.obj:instance) "square")
  ...
  ((Webster 1.clsn.obj:type) "NOUN")
  ((Webster 1.clsn.obj:type) "INT-VB")
  ((Webster 1.clsn.obj:type) "TR-VB")
  ((Webster 1.clsn.obj:type) "ADJ")
  ...
  (Webster "bet" "NOUN")
  (Webster "bet" "INF-VB")
  (Webster "bet" "TR-VB")
  (Webster "eat" "INF-VB")
  (Webster "eat" "TR-VB")
  (Webster "fit" "NOUN")
  (Webster "fit" "INF-VB")
  (Webster "fit" "TR-VB")
  (Webster "fit" "ADJ")
  (Webster "friend" "NOUN")
  (Webster "friend" "TR-VB")
  (Webster "square" "NOUN")
  (Webster "square" "TR-VB")
  (Webster "square" "ADJ")
  ...
)

```

References

- [1] Jon Barwise and Jerry Seligman. *Information Flow: The Logic of Distributed Systems*, volume 44 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1997.