FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS

FIPA KIF Content Language Specification

Document title	FIPA KIF Content Language Specification		
Document number	XC00010B	**Document source**	FIPA TC C
Document status	Experimental	**Date of this status**	2001/08/10
Supersedes	None		
Contact	fab@fipa.org		
Change history			
2000/08/22	Approved for Experimental		
2001/08/10	Line numbering added		

© 2000 Foundation for Intelligent Physical Agents - http://www.fipa.org/

Geneva, Switzerland

Notice

Use of the technologies described in this specification may infringe patents, copyrights or other intellectual property rights of FIPA Members and non-members. Nothing in this specification should be construed as granting permission to use any of the technologies described. Anyone planning to make use of technology covered by the intellectual property rights of others should first obtain permission from the holder(s) of the rights. FIPA strongly encourages anyone implementing any part of this specification to determine first whether part(s) sought to be implemented are covered by the intellectual property of others, and, if so, to obtain appropriate licenses or other permission from the holder(s) of such intellectual property prior to implementation. This specification is subject to change without notice. Neither FIPA nor any of its Members accept any responsibility whatsoever for damages or liability, direct or consequential, which may result from the use of this specification.

Foreword

The Foundation for Intelligent Physical Agents (FIPA) is an international organization that is dedicated to promoting the industry of intelligent agents by openly developing specifications supporting interoperability among agents and agent-based applications. This occurs through open collaboration among its member organizations, which are companies and universities that are active in the field of agents. FIPA makes the results of its activities available to all interested parties and intends to contribute its results to the appropriate formal standards bodies.

The members of FIPA are individually and collectively committed to open competition in the development of agent-based applications, services and equipment. Membership in FIPA is open to any corporation and individual firm, partnership, governmental body or international organization without restriction. In particular, members are not bound to implement or use specific agent-based standards, recommendations and FIPA specifications by virtue of their participation in FIPA.

The FIPA specifications are developed through direct involvement of the FIPA membership. The status of a specification can be either Preliminary, Experimental, Standard, Deprecated or Obsolete. More detail about the process of specification may be found in the FIPA Procedures for Technical Work. A complete overview of the FIPA specifications and their current status may be found in the FIPA List of Specifications. A list of terms and abbreviations used in the FIPA specifications may be found in the FIPA Glossary.

FIPA is a non-profit association registered in Geneva, Switzerland. As of January 2000, the 56 members of FIPA represented 17 countries worldwide. Further information about FIPA as an organization, membership information, FIPA specifications and upcoming meetings may be found at http://www.fipa.org/.

Contents

1 Scope

This document gives the specification the draft proposed American National Standard (ANSkif) for Knowledge Interchange Format (KIF) as a content language for FIPA ACL (see [FIPA00061]. This specification covers:

- Expression of objects as terms.

- Expression of propositions as sentences.

FIPA KIF currently has no specific way to expresses actions.

2 FIPA KIF Specification

The aim of this section is to specify KIF as a language for use in the interchange of knowledge among disparate computer systems (created by different programmers, at different times, in different languages, and so forth), especially among FIPA agents.

FIPA KIF is *not* intended as a primary language for interaction with human users (though it can be used for this purpose). Different computer systems can interact with their users in whatever forms are most appropriate to their applications (for example, Prolog, conceptual graphs, natural language and so forth).

FIPA KIF is also *not* intended as an internal representation for knowledge *within* computer systems or within closely related sets of computer systems (though the language can be used for this purpose as well). Typically, when a computer system reads a knowledge base in FIPA KIF, it converts the data into its own internal form (specialized pointer structures, arrays, etc.) and all computation is done using these internal forms. When the computer system needs to communicate with another computer system, it maps its internal data structures into FIPA KIF before message transfer.

The following categorical features are essential to the design of FIPA KIF:

- 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. In this way, FIPA KIF differs from other languages that are based on specific interpreters, such as Emycin and Prolog.

- The language is logically comprehensive. At its most general, it provides for the expression of arbitrary logical sentences. In this way, it differs from relational database languages (like SQL) and logic programming languages (like Prolog).

- The language provides for the representation of knowledge about knowledge. This allows the user to make knowledge representation decisions explicit and permits the user to introduce new knowledge representation constructs without changing the language.

In addition to these essential features, FIPA KIF is designed to maximize the following additional features (to the extent possible while preserving the preceding features):

- **Implementability**. Although FIPA KIF is not intended for use within programs as a representation or communication language, it should be usable for that purpose if so desired.

- **Readability**. Although FIPA KIF is not intended primarily as a language for interaction with humans, human readability facilitates its use in describing representation language semantics, its use as a publication language for example knowledge bases, its use in assisting humans with knowledge base translation problems, etc.

Unless otherwise stated, all terms and definitions are taken from [ISO10646] and [ISO14481].

2.1 Syntax

2.1.1 Introduction

As with many computer-oriented languages, the syntax of FIPA KIF is most easily described in three layers. First, there are the basic characters of the language. These characters can be combined to form lexemes. Finally, the lexemes of the language can be combined to form grammatically legal expressions. Although this layering is not strictly essential to the specification of FIPA KIF, it simplifies the description of the syntax by dealing with white space at the lexeme level and eliminating that detail from the expression level.

In this section, the syntax of FIPA KIF is presented using a modified BNF notation. All nonterminals and BNF punctuation are written in boldface, while characters in FIPA KIF are expressed in plain font. The notation **{**x1, ..., xn**}** means the set of terminals x1, ..., xn. The notation **[nonterminal]** means zero or one instances of **nonterminal**; **nonterminal*** means zero or more occurrences; **nonterminal+** means one or more occurrences; **nonterminal ^ n** means **n** occurrences. The notation **nonterminal1 - nonterminal2** refers to all of the members of **nonterminal1** except for those in **nonterminal2**. The notation **int (n)** denotes the decimal representation of integer n. The nonterminals **space**, **tab**, **return**, **linefeed** and **page** refer to the characters corresponding to ASCII codes 32, 9, 13, 10, and 12, respectively. The nonterminal **character** denotes the set of all 128 ASCII characters. The nonterminal **empty** denotes the empty string.

2.1.2 Characters

The alphabet of FIPA KIF consists of 7 bit blocks of data. In this document, we refer to FIPA KIF data blocks via their usual ASCII encodings as characters as given in [ISO646].

FIPA KIF characters are classified as upper case letters, lower case letters, digits, alpha characters (non-alphabetic characters that are used in the same way that letters are used), special characters, white space, and other characters (every ASCII character that is not in one of the other categories):

```
upper      ::=  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

lower      ::=  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

alpha      ::=  ! | $ | % | & | * | + | - | . | / | < | = |   | ? |
                @ | _ | ~ |

special    ::=  " | # | ' | ( | ) | , | \ | ^ | '

white      ::=  space | tab | return | linefeed | page
```

A normal character is either an upper case character, a lower case character, a digit, or an alpha character.

```
normal     ::=  upper | lower | digit | alpha
```

2.1.3 Lexemes

The process of converting characters into lexemes in called lexical analysis. The input to this process is a stream of characters, and the output is a stream of lexemes.

The function of a lexical analyser is cyclic. It reads characters from the input string until it encounters a character that cannot be combined with previous characters to form a legal lexeme. When this happens, it outputs the lexeme corresponding to the previously read characters. It then starts the process over again with the new character. White space causes a break in the lexical analysis process but otherwise is discarded.

There are five types of lexemes in FIPA KIF: special lexemes, words, character references, character strings and character blocks. Each special character forms its own lexeme. It cannot be combined with other characters to form more complex lexemes, except through the escape' syntax described below.

A **word** is a contiguous sequence of normal characters or other characters preceded by the escape character `\`.

```
word  ::=  normal | word normal | word\character
```

It is possible to include the character `\` in a word by preceding it by another occurrence of `\`, that is, two contiguous occurrences of `\` are interpreted as a single occurrence. For example, the string `A\\\'B` corresponds to a word consisting of the four characters `A`, `\`, ', and `B`.

Except for characters following `\`, the lexical analysis of words is case insensitive. The output lexeme for any word corresponds to the lexeme obtained by converting all letters not following `\` to their upper case equivalents. For example, the word `abc` and the word `ABC` map into the same lexeme. The word `a\bc` maps into the same lexeme as the word `A\bC`, which is not the same as the lexeme for the word `ABC`, since the second character is lower case.

A **character reference** consists of the characters `#`, `\`, and any character. Character references allow us to refer to characters as characters and differentiate them from one-character symbols, which may refer to other objects.

```
charref     ::=   #\character
```

A **character string** is a series of characters enclosed in quotation marks. The escape character `\` is used to permit the inclusion of quotation marks and the `\` character itself within such strings.

```
string      ::=   "quotable"

quotable    ::=   empty | quotable strchar | quotable\character

strchar     ::=   character - {",\}
```

Sometimes it is desirable to group together a sequence of arbitrary bits or characters without imposing escape characters, for example, to encode images, audio, or video in special formats. Character blocks permit this sort of grouping through the use of a prefix that specifies how many of the following characters are to grouped together in this way. A **character block** consists of the character `#` followed by the decimal encoding of a positive integer *n*, the character `q` or `Q` and then *n* arbitrary characters.

```
block       ::=   # int(n) q character^n | # int(n) Q character^n
```

For the purpose of grammatical analysis, it is useful to subdivide the class of words a little further, viz. as variables, operators and constants.

A **variable** is a word in which the first character is `?` or `@`. A variable that begins with `?` is called an **individual variable**. A variable that begins with an `@` is called a **sequence variable**.

```
variable    ::=   indvar | seqvar

indvar      ::=   ?word

seqvar      ::=   @word
```

Operators are used in forming complex expressions of various sorts. There are three types of operators in FIPA KIF:

- **Term operators** are used in forming complex terms.

- **Sentence operators** and user operators are used in forming complex sentences.

- **Definition operators** are used in forming definitions.

```
operator    ::=   termop | sentop | defop

termop      ::=   value | listof | quote | if

sentop      ::=   holds | = | /= | not | and | or | = | <= | <= |
                  forall | exists

defop       ::=   defobject | defunction | defrelation | deflogical |
                  := | :- | :<= | :=
```

All other words are called **constants**:

```
constant    ::=   word - variable - operator
```

Semantically, there are four categories of constants in FIPA KIF:

- **Object constants** are used to denote individual objects.
- **Function constants** denote functions on those objects.
- **Relation constants** denote relations.
- **Logical constants** express conditions about the world and are either true or false.

FIPA KIF is unusual among logical languages in that there is no syntactic distinction among these four types of constants; any constant can be used where any other constant can be used. The differences between these categories of constants is entirely semantic.

2.1.4 Expressions

The legal expressions of FIPA KIF are formed from lexemes according to the rules presented in this section. There are three disjoint types of expressions in the language:

- **Terms** are used to denote objects in the world being described.
- **Sentences** are used to express facts about the world.
- **Definitions** are used to define constants.

There are nine types of terms in FIPA KIF: individual variables, constants, character references, character strings, character blocks, functional terms, list terms, quotations, and logical terms. Individual variables, constants, character references, strings and blocks were discussed earlier.

```
term        ::=   indvar | constant | charref | string | block |
                  funterm | listterm | quoterm | logterm
```

A ***implicit functional term*** consists of a constant and an arbitrary number of argument terms, terminated by an optional sequence variable and surrounded by matching parentheses. Note that there is no syntactic restriction on the number of argument terms; arity restrictions in FIPA KIF are treated semantically.

```
funterm     ::=   (constant term* [seqvar])
```

A **explicit functional term** consists of the operator value and one or more argument terms, terminated by an optional sequence variable and surrounded by matching parentheses.

```
funterm     ::=   (value term term* [seqvar])
```

A **list term** consists of the `listof` operator and a finite list of terms, terminated by an optional sequence variable and enclosed in matching parentheses.

```
listterm    ::=   (listof term* [seqvar])
```

Quotations involve the quote operator and an arbitrary *list expression*. A list expression is either an *atom* or a sequence of list expressions surrounded by parentheses. An atom is either a word or a character reference or a character string or a character block. Note that the list expression embedded within a quotation need *not* be a legal expression in FIPA KIF.

```
quoterm     ::=   (quote listexpr) | 'listexpr
```

```
listexpr    ::=   atom | (listexpr*)

atom        ::=   word | charref | string | block
```

Logical terms involve the `if` and `cond` operators. The `if` form allows for the testing of a single condition or multiple conditions and an optional term at the end allows for the specification of a default value when all of the conditions are false. The `cond` form is similar but groups the pairs of sentences and terms within parentheses and has no optional term at the end.

```
logterm     ::=   (if logpair+ [term])

logpair     ::=   sentence term

logterm     ::=   (cond logitem*)

logitem     ::=   (sentence term)
```

The following BNF defines the set of legal sentences in FIPA KIF. There are six types of sentences (logical constants have already been introduced):

```
sentence    ::=   constant | equation | inequality |
                  relsent | logsent | quantsent
```

An **equation** consists of the = operator and two terms. An ***inequality*** consist of the `/=` operator and two terms.

```
equation    ::=   (= term term)

inequality  ::=   (/= term term)
```

An **implicit relational sentence** consists of a constant and an arbitrary number of argument terms, terminated by an optional sequence variable. As with functional terms, there is no syntactic restriction on the number of argument terms in a relation sentence.

```
relsent     ::=   (constant term* [seqvar])
```

A **explicit relational sentence** consists of the operator holds and one or more argument terms, terminated by an optional sequence variable and surrounded by matching parentheses.

```
relsent     ::=(holds term term* [seqvar])
```

It is noteworthy that the syntax of implicit relational sentences is the same as that of implicit functional terms. On the other hand, their meanings are different. Fortunately, the context of each such expression determines its type (as an embedded term in one case or as a top-level sentence or argument to some sentential operator in the other case); and so this slight ambiguity causes no problems.

The syntax of **logical sentences** depends on the logical operator involved. A sentence involving the not operator is called a negation. A sentence involving the and operator is called a conjunction, and the arguments are called conjuncts. A sentence involving the or operator is called a disjunction, and the arguments are called disjuncts. A sentence involving the = operator is called an implication, all of its arguments but the last are called antecedents which is called the consequent. A sentence involving the <= operator is called a reverse implication, its first argument is called the consequent and the remaining arguments are called the antecedents. A sentence involving the <= operator is called an equivalence.

```
logsent ::= (not sentence) |
            (and sentence*) |
            (or sentence*) |
            (= sentence* sentence) |
            (<= sentence sentence*) |
```

```
                  (<= sentence sentence)
```

There are two types of **quantified sentences**: a universally quantified sentence is signalled by the use of the `forall` operator, and an existentially quantified sentence is signalled by the use of the `exists` operator. The first argument in each case is a list of variable specifications. A variable specification is either a variable or a list consisting of a variable and a term denoting a relation that restricts the domain of the specified variable.

```
quantsent   ::=   (forall (varspec+) sentence) |
                  (exists (varspec+) sentence)

varspec     ::=   variable | (variable constant)
```

Note that, according to these rules, it is permissible to write sentences with free variables, that is, variables that do not occur within the scope of any enclosing quantifiers. The significance of the free variables in a sentence depends on the use of the sentence. When we assert the truth of a sentence with free variables, we are, in effect, saying that the sentence is true for all values of the free variables, i.e. the variables are universally quantified. When we ask whether a sentence with free variables is true, we are, in effect, asking whether there are any values for the free variables for which the sentence is true, i.e. the variables are existentially quantified.

The following BNF defines the set of legal FIPA KIF definitions. There are three types of definitions: unrestricted, complete and partial. Within each type, there are four cases, one for each category of constant. Object constants are defined using the `defobject` operator, function constants are defined using the `deffunction` operator, relation constants are defined using the `defrelation` operator and logical constants are defined using the `deflogical` operator.

```
definition  ::=   unrestricted | complete | partial

unrestricted::=   (defobject constant [string] sentence*)
            |     (deffunction constant [string] sentence*)
            |     (defrelation constant [string] sentence*)
            |     (deflogical constant [string] sentence*)

complete    ::=   (defobject constant [string] := term)
            |     (deffunction constant (indvar* [seqvar]) [string] := term)
            |     (defrelation constant (indvar* [seqvar]) [string] := sentence)
            |     (deflogical constant [string] := sentence)

partial     ::=   (defobject constant [string] :- indvar :<= sentence)
            |     (defobject constant [string] :- indvar := sentence)
            |     (deffunction constant (indvar* [seqvar])
                        [string] :- indvar :<= sentence)
            |     (deffunction constant (indvar* [seqvar])
                        [string] :- indvar := sentence)
            |     (defrelation constant (indvar* [seqvar])
                        [string] :<= sentence)
            |     (defrelation constant (indvar* [seqvar])
                        [string] := sentence)
            |     (deflogical constant [string] :<= sentence)
                        (deflogical constant [string] := sentence)
```

A **form** in FIPA KIF is either a sentence or a definition.

```
form        ::=   sentence | definition
```

It is important to note that definitions are top level constructs. While definitions contain sentences, they are not themselves sentences and, therefore, cannot be written as constituent parts of sentences or other definitions (unless they occur inside of a quotation.

A **knowledge base** is a finite set of forms. It is important to keep in mind that a knowledge base is a *set* of sentences, not a *sequence*; and, therefore, the order of forms within a knowledge base is unimportant. Order *may* have heuristic value to deductive programs by suggesting an order in which to use those sentences; however, this implicit approach to knowledge exchange lies outside of the definition of FIPA KIF.

2.2 Basics

2.2.1 Introduction

The basis for the semantics of FIPA KIF is a conceptualization of the world in terms of objects and relations among those objects.

A *universe of discourse* is the set of all objects presumed or hypothesized to exist in the world. The notion of object used here is quite broad. Objects can be concrete, for example, a specific carbon atom, Confucius, the Sun or abstract, such as the number 2, the set of all integers or the concept of justice. Objects can be primitive or composite, for example, a circuit that consists of many sub circuits. Objects can even be fictional, for example, a unicorn, Sherlock Holmes, etc.

Different users of a declarative representation language, like FIPA KIF, are likely to have different universes of discourse. FIPA KIF is conceptually promiscuous in that it does not require every user to share the same universe of discourse. On the other hand, FIPA KIF is conceptually grounded in that every universe of discourse is required to include certain basic objects.

The following basic objects must occur in every universe of discourse:

- All numbers, real and complex.

- All ASCII characters.

- All finite strings of ASCII characters.

- Words and the things they represent.

- All finite lists of objects in the universe of discourse.

- Bottom. A distinguished object that occurs as the value of a partial when that function is applied to arguments for which the function make no sense.

Remember, that to these basic elements, the user can add whatever non-basic objects seem useful.

In FIPA KIF, relationships among objects take the form of relations. Formally, a relation is defined as an arbitrary set of finite lists of objects (of possibly varying lengths). Each list is a selection of objects that jointly satisfy the relation. For example, the `<` relation on numbers contains the list <2,3>, indicating that 2 is less than 3.

A function is a special kind of relation. For every finite sequence of objects (called the arguments), a function associates a unique object (called the value). More formally, a function is defined as a set of finite lists of objects, one for each combination of possible arguments. In each list, the initial elements are the arguments, and the final element is the value. For example, the `1+` function contains the list <2, 3>, indicating that integer successor of 2 is 3.

Note that both functions and relations are defined as sets of lists. In fact, every function is a relation. However, not every relation is a function. In a function, there cannot be two lists that disagree on only the last element, since this would be tantamount to the function having two values for one combination of arguments. By contrast, in a relation, there can be any number of lists that agree on all but the last element. For example, the list <2, 3> is a member of the `1+` function, and there is no other list of length 2 with 2 as its first argument, that is, there is only one successor for 2. By contrast, the `<` relation contains the lists <2, 3>, <2, 4>, <2, 5>, and so forth, indicating that 2 is less than 3, 4, 5, and so forth.

Many mathematicians require that functions and relations have fixed arity, that is, they require that all of the lists comprising a relation have the same length. The definitions here allow for relations with variable arity; it is perfectly acceptable for a function or a relation to contain lists of different lengths. For example, the relation `<` contains the lists <2, 3> and <2, 3, 4>, reflecting the fact that 2 is less than 3 and the fact that 2 is less than 3 and 3 is less than 4. This flexibility is not essential, but it is extremely convenient and poses no significant theoretical problems.

2.2.2 Bottom

In FIPA KIF, all functions are total, that is, there is a value for every combination of arguments. In order to allow a user to express the idea that a function is not meaningful for certain arguments, FIPA KIF assumes that there is a special "undefined" object in the universe and provides the object constant `bottom` to refer to this object.

2.2.3 Functional Terms

The value of a functional term without a terminating sequence variable is obtained by applying the function denoted by the function constant in the term to the objects denoted by the arguments.

For example, the value of the term `(+ 2 3)` is obtained by applying the addition function (the function denoted by `+`) to the numbers 2 and 3 (the objects denoted by the object constants `2` and `3`) to obtain the value 5, which is the value of the object constant `5`.

If a functional term has a terminating sequence variable, the value is obtained by applying the function to the sequence of arguments formed from the values of the terms that precede the sequence variable and the values in the sequence denoted by the sequence variable.

Assume, for example, that the sequence variable `@l` has as value the sequence 2, 3, 4. Then, the value of the term `(+ 1 @l)` is obtained by applying the addition function to the numbers 1, 2, 3, and 4 to obtain the value 10, which is the value of the object constant `10`.

2.2.4 Relational Sentences

A simple relational sentence without a terminating sequence variable is true if and only if the relation denoted by the relation constant in the sentence is true of the objects denoted by the arguments. Equivalently, viewing a relation as a set of tuples, we say that the relational sentence is true if and only if the tuple of objects formed from the values of the arguments is a member of the set of tuples denoted by the relation constant.

If a relational sentence terminates in a sequence variable, the sentence is true if and only if the relation contains the tuple consisting of the values of the terms that precede the sequence variable together with the objects in the sequence denoted by the variable.

2.2.5 Equations and Inequalities

An equation is true if and only if the terms in the equation refer to the same object in the universe of discourse. An inequality is true if and only if the terms in the equation refer to distinct objects in the universe of discourse.

2.2.6 True and False

The truth-value of true is `true`, and the truth-value of false is `false`.

2.3 Logic

2.3.1 Logical Terms

The value of a logical term involving the if operator is the value of the term following the first true sentence in the argument list. For example, the term `(if (1 2) 1 (2 1) 2 0)` is equivalent to 2.

If none of the embedded sentences of a logical term involving the if operator is true and there is an isolated term at the end, the value of the conditional term is the value of that isolated term. For example, if the object constant a denotes a number, then the term `(if (a 0) a (- a))` denotes the absolute value of that number.

If none of the embedded sentences is true and there is no isolated term at the end, the value is undefined (i.e. bottom). In other words, the term `(if (p a) a)` is equivalent to `(if (p a) a bottom)`.
The value of a logical term involving the `cond` operator is the value of the term following the first true sentence in the argument list. For example, the term `(cond ((1 2) 1) ((2 1) 2))` is equivalent to 2.

If none of the embedded sentences is true, the value is undefined. In other words, the term `(cond ((p a) a))` is equivalent to `(cond ((p a) a) (true bottom))`.

2.3.2 Logical Sentences

A negation is true if and only if the negated sentence is false.

A conjunction is true if and only if every conjunct is true.

A disjunction is true if and only if at least one of the disjuncts is true.

If every antecedent in an implication is true, then the implication as a whole is true if and only if the consequent is true. If any of the antecedents is false, then the implication as a whole is true, regardless of the truth-value of the consequent.

A reverse implication is just an implication with the consequent and antecedents reversed.

An equivalence is equivalent to the conjunction of an implication and a reverse implication.

2.3.3 Quantified Sentences

A simple existentially quantified sentence (one in which the first argument is a list of variables) is true if and only if the embedded sentence is true for some value of the variables mentioned in the first argument.

A simple universally quantified sentence (one in which the first argument is a list of variables) is true if and only if the embedded sentence is true for every value of the variables mentioned in the first argument.

Quantified sentences with complicated variables specifications can be converted into simple quantified sentences by replacing each complicated variable specification by the variable in the specification and adding an appropriate condition into the body of the sentence. Note that, in the case of a set restriction, it may be necessary to rename variables to avoid conflicts. The following pairs of sentences show the transformation from complex quantified sentences to simple quantified sentences.

```
(forall (... (?x r) ...) s)
(forall (...  ?x    ...) (= (r ?x) s))

(exists (... (?x r) ...) s)
(exists (...  ?x    ...) (and (r ?x) s))
```

Note that the significance of free variables in quantifier-free sentences depends on context. Free variables in an assertion are assumed to be universally quantified. Free variables in a query are assumed to be existentially quantified.

In other words, the meaning of free variables is determined by the way in which FIPA KIF is used. It cannot be unambiguously defined within FIPA KIF itself. To be certain of the usage in all contexts, use explicit quantifiers.

2.3.4 Definitions

The definitional operators in FIPA KIF allow us to state sentences that are true "by definition" in a way that distinguishes them from sentences that express contingent properties of the world. Definitions have no truth-values in the usual sense; they are so because we say that they are so.

On the other hand, definitions have content: sentences that allow us to derive other sentences as conclusions. In FIPA KIF, every definition has a corresponding set of sentences, called the content of the definition.

The `defobject` operator is used to define objects. The legal forms are shown below, together with their content. In the first case, the content is the equation involving the object constant in the definition with the defining term. In the second case, the content is the conjunction of the constituent sentences.

```
(defobject s := t)
      (= s t)

(defobject s p1 ... pn)
      (and p1 ... pn)

(defobject s :- v  := p)
      (= (= s v) p)

(defobject s :- v  :<= p)
      (<= (= s v) p)
```

The `deffunction` operator is used to define functions. Again, the legal forms are shown below, together with their defining axioms. In the first case, the content is the equation involving the term formed from the function constant in the definition and the variables in its argument list and the defining term. In the second case, as with object definitions, the content is the conjunction of the constituent sentences.

```
(deffunction f (v1 ...vn) := t)
      (= (f v1 ...vn) t)

(deffunction f p1 ...pn)
      (and p1 ...pn)

(deffunction f (v1 ... vn) :- v := p)
      (= (= (f v1 ... vn) v) p)

(deffunction f (v1 ... vn) :- v :<= p)
      (<= (= (f v1 ... vn) v) p)
```

The `defrelation` operator is used to define relations. The legal forms are shown below, together with their defining axioms. In the first case, the content is the equivalence relating the relational sentence formed from the relation constant in the definition and the variables in its argument list and the defining sentence. In the second case, as with object and function definitions, the content is the conjunction of the constituent sentences.

```
(defrelation r (v1 ...vn) := p)
      (<= (r v1 ...vn) p)

(defrelation r p1 ...pn)
      (and p1 ...pn)

(defrelation r (v1 ... vn) := p)
      (= (r v1 ... vn) p))

(defrelation r (v1 ... vn) :<= p)
```

```
            (<= (r v1 ... vn) p))
```

2.4 Numbers

2.4.1 Introduction

The referent of every numerical constant in FIPA KIF is assumed to be the number for which that constant is the base 10 representation. Among other things, this means that we can infer inequality of all distinct numerical constants, i.e. for every `t1` and distinct `t2` the following sentence is true.

```
    (/= t1 t2)
```

We use the intended meaning of numerical constants in defining the numerical functions and relations in this section. In particular, we require that these functions and relations behave correctly on all numbers represented in this way.

Note that this does mean that it is incorrect to apply these functions and relations to terms other than numbers. For example, a non-numerical term may refer to a number, for example, the term two may be defined to be the same as the number 2 in which case it is perfectly proper to write `(+ two two)`.

The user may also want to extend these functions and relations to apply to objects other than numbers, for example, sets and lists.

2.4.2 Functions on Numbers

- `*`
  If `t1`, ..., `tn` denote numbers, then the term `(* t1 ... tn)` denotes the product of those numbers.

- `+`
  If `t1`, ..., `tn` are numerical constants, then the term `(+ t1 ... tn)` denotes the sum t of the numbers corresponding to those constants.

- `-`
  If `t` and `t1`, ..., `tn` denote numbers, then the term `(- t t1 ... tn)` denotes the difference between the number denoted by `t` and the numbers denoted by `t1` through `tn`. An exception occurs when `n=0`, in which case the term denotes the negation of the number denoted by `t`.

- `/`
  If `t1`, ..., `tn` are numbers, then the term `(/ t1 ... tn)` denotes the result `t` obtained by dividing the number denoted by `t1` by the numbers denoted by `t2` through `tn`. An exception occurs when `n=1`, in which case the term denotes the reciprocal `t` of the number denoted by `t1`.

- `1+`
  The term `(1+ t)` denotes the sum of the object denoted by `t` and `1`.

  ```
  (deffunction 1+ (?x) := (+ ?x 1))
  ```

- `1-`
  The term `(1- t)` denotes the difference of the object denoted by `t` and `1`.

  ```
  (deffunction 1- (?x) := (- ?x 1))
  ```

- `abs`
  The term `(abs t)` denotes the absolute value of the object denoted by `t`.

  ```
  (deffunction abs (?x) := (if (= ?x 0) ?x (- ?x)))
  ```

- `ceiling`
  If `t` denotes a real number, then the term `(ceiling t)` denotes the smallest integer greater than or equal to the number denoted by `t`.

- `denominator`
  The term `(denominator t)` denotes the denominator of the canonical reduced form of the object denoted by `t`.

- `expt`
  The term `(expt t1 t2)` denotes the object denoted by `t1` raised to the power the object denoted by `t2`.

- `floor`
  The term `(floor t)` denotes the largest integer less than the object denoted by `t`.

- `gcd`
  The term `(gcd t1 ... tn)` denotes the greatest common divisor of the objects denoted by `t1` through `tn`.

- `imagpart`
  The term `(imagpart t)` denotes the imaginary part of the object denoted by `t`.

- `lcm`
  The term `(lcm t1 ... tn)` denotes the least common multiple of the objects denoted by `t1`, ..., `tn`.

- `log`
  The term `(log t1 t2)` denotes the logarithm of the object denoted by `t1` in the base denoted by `t2`.

- `max`
  The term `(max t1 ... tk)` denotes the largest object denoted by `t1` through `tn`.

- `min`
  The term `(min t1 ... tk)` denotes the smallest object denoted by `t1` through `tn`.

- `mod`
  The term `(mod t1 t2)` denotes the root of the object denoted by `t1` modulo the object denoted by `t2`. The result will have the same sign as denoted by `t1`.

- `numerator`
  The term `(numerator t)` denotes the numerator of the canonical reduced form of the object denoted by `t`.

- `realpart`
  The term `(realpart t)` denotes the real part of the object denoted by `t`.

- `rem`
  The term `(rem t1 t2)` denotes the remainder of the object denoted by `t1` divided by the object denoted by `t2`. The result has the same sign as the object denoted by `t2`.

- `round`
  The term `(round t)` denotes the integer nearest to the object denoted by `t`. If the object denoted by `t` is halfway between two integers (for example 3.5), it denotes the nearest integer divisible by 2.

- `sqrt`
  The term `(sqrt t)` denotes the principal square root of the object denoted by `t`.

- `truncate`
  The term `(truncate t)` denotes the largest integer less than the object denoted by `t`.

2.4.3 Relations on Numbers

- `integer`
  The sentence `(integer t)` means that the object denoted by `t` is an integer.

- `real`
  The sentence `(real t)` means that the object denoted by `t` is a real number.

- `complex`
  The sentence `(complex t)` means that the object denoted by `t` is a complex number.

```
(defrelation number (?x) := (or (real ?x) (complex ?x)))

(defrelation natural (?x) := (and (integer ?x) (= ?x 0)))

(defrelation rational (?x) :=
            (exists (?y) (and (integer ?y) (integer (* ?x ?y)))))
```

- `approx`
  The sentence `(approx t1 t2 t)` is true if and only if the number denoted by `t1` is "approximately equal" to the number denoted by `t2`, that is, the absolute value of the difference between the numbers denoted by `t1` and `t2` is less than or equal to the number denoted by `t`.

- `<`
  The sentence `(< t1 t2)` is true if and only if the number denoted by `t1` is less than the number denoted by `t2`.

```
(defrelation > (?x ?y) := (< ?y ?x))

(defrelation =< (?x ?y) := (or (= ?x ?y) (< ?x ?y)))

(defrelation >= (?x ?y) := (or (> ?x ?y) (= ?x ?y)))

(defrelation positive (?x) := (> ?x 0))

(defrelation negative (?x) := (< ?x 0))

(defrelation zero (?x) := (= ?x 0))

(defrelation odd (?x) := (integer (/ (+ ?x 1) 2))

(defrelation even (?x) := (integer (/ ?x 2))
```

2.5 Lists

A list is a finite sequence of objects. Any objects in the universe of discourse may be elements of a list.

In FIPA KIF, we use the term `(listof t1 ... tk)` to denote the list of objects denoted by `t1`, ..., `tk`. For example, the following expression denotes the list of an object named `mary`, a list of objects named `tom`, `dick` and `harry`, and an object named `sally`.

```
(listof mary (listof tom dick harry) sally)
```

The relation list is the type predicate for lists. An object is a list if and only if there is a corresponding expression involving the listof operator.

```
(defrelation list (?x) := (exists (@l) (= ?x (listof @l))))
```

The object constant `nil` denotes the empty list and also tests whether or not an object is the empty list. The relation constants `single`, `double` and `triple` allow us to assert the length of lists containing one, two or three elements, respectively.

```
(defobject nil := (listof))

(defrelation null (?l) := (= ?l (listof)))

(defrelation single (?l) := (exists (?x) (= ?l (listof ?x))))

(defrelation double (?l) := (exists (?x ?y) (= ?l (listof ?x ?y))))

(defrelation triple (?l) := (exists (?x ?y ?z) (= ?l (listof ?x ?y ?z))))
```

The functions `first`, `rest`, `last` and `butlast` each take a single list as argument and select individual items or sub lists from those lists.

```
(deffunction first (?l) := (if (= (listof ?x @items) ?l) ?x)

(deffunction rest (?l) :=
            (cond ((null ?l) ?l)
                  ((= ?l (listof ?x @items)) (listof @items))))

(deffunction last (?l) :=
            (cond ((null ?l) bottom) ((null (rest ?l)) (first ?l))
                  (true (last (rest ?l)))))

(deffunction butlast (?l) :=
            (cond ((null ?l) bottom) ((null (rest ?l)) nil)
                  (true (cons (first ?l) (butlast (rest ?l))))))
```

The sentence `(item t1 t2)` is true if and only if the object denoted by `t2` is a non-empty list and the object denoted by `t1` is either the first item of that list or an item in the rest of the list.

```
(defrelation item (?x ?l) :=
            (and (list ?l) (not (null ?l))
                  (or (= ?x (first ?l)) (item ?x (rest ?l)))))
```

The sentence `(sublist t1 t2)` is true if and only if the object denoted by `t1` is a final segment of the list denoted by `t2`.

```
(defrelation sublist (?l1 ?l2) :=
            (and (list ?l1) (list ?l2)
                  (or (= ?l1 ?l2) (sublist ?l1 (rest ?l2)))))
```

The function `cons` adds the object specified as its first argument to the front of the list specified as its second argument.

```
(deffunction cons (?x ?l) :=
            (if (= ?l (listof @l)) (listof ?x @l)))
```

The function `append` adds the items in the list specified as its first argument to the list specified as its second argument. The function `revappend` is similar, except that it adds the items in reverse order.

```
(deffunction append (?l1 ?l2) :=
            (cond ((null ?l1) (if (list ?l2) ?l2))
                  ((list ?l1) (cons (first ?l1) (append (rest ?l1) ?l2)))))

(deffunction revappend (?l1 ?l2) :=
            (cond ((null ?l1) (if (list ?l2) ?l2))
                  ((list ?l1) (revappend (rest ?l1) (cons (first ?l1) ?l2)))))
```

The function `reverse` produces a list in which the order of items is the reverse of that in the list supplied as its single argument.

```
(deffunction reverse (?l) := (revappend ?l (listof)))
```

The functions `adjoin` and `remove` construct lists by adding or removing objects from the lists specified as their arguments.

```
(deffunction adjoin (?x ?l) := (if (item ?x ?l) ?l (cons ?x ?l)))

(deffunction remove (?x ?l) :=
             (cond ((null ?l) nil) ((and (= ?x (first ?l)) (list ?l))
                   (remove ?x (rest ?l)))
                   ((list ?l) (cons ?x (remove ?x (rest ?l))))))
```

The value of `subst` is the object or list obtained by substituting the object supplied as first argument for all occurrences of the object supplied as second argument in the object or list supplied as third argument.

```
(deffunction subst (?x ?y ?z) :=
             (cond ((= ?y ?z) ?x) ((null ?z) nil)
                   ((list ?z) (cons (subst ?x ?y (first ?z))
                   (subst ?x ?y (rest ?z))))
                   (true ?z)))
```

The function `length` gives the number of items in a list. The function `nth` returns the item in the list specified as its first argument in the position specified as its second argument. The function `nthrest` returns the list specified as its first argument minus the first *n* items, where *n* is the number specified as its second argument.

```
(deffunction length (?l) :=
             (cond ((null ?l) 0)
                   ((list ?l) (1+ (length (rest ?l))))))

(deffunction nth (?l ?n) :=
             (cond ((= ?n 1) (first ?l))
                   ((and (list ?l) (positive ?n)) (nth (rest ?l) (1- ?n)))))

(deffunction nthrest (?l ?n) :=
             (cond ((= ?n 0) (if (list ?l) ?l))
                   ((and (list ?l) (positive ?n)) (nthrest (rest ?l) (1- ?n)))))
```

2.6 Characters and Strings

2.6.1 Characters

A character is a printed symbol, such as a digit or a letter. There are 128 distinct characters known to FIPA KIF, corresponding to the 128 possible combinations of bits in the ASCII encoding. In FIPA KIF, there are two ways to refer to characters.

The first method is use of the `charref` syntax, that is, the characters `#` and `\`, followed by the character to be represented. While this method works for all 128 characters, it is less than ideal for documents like this one, because of the difficulty of writing out non-printing characters. Using this method, it is also difficult to assert properties of some classes of characters. For this reason, FIPA KIF supports an alternative method of specification, viz. the use of the 7 bit code corresponding to the character. The relationship between characters and their numerical codes is given via the functions char-code and code-char. The former maps the *n*th character *cn* into the corresponding 7-bit integer n, and the latter maps a 7-bit integer *n* into the corresponding character *cn*. The values of these functions on all other arguments are undefined.

```
(= (char-code #\cn) n)

(= (code-char n) #\cn)
```

The relation `character` is true of the characters of FIPA KIF and no other objects.

```
(defrelation character (?x) :=
        (exists ((?n natural-number)) (and (= ?n 0) (
```

2.6.2 Strings

A string is a list of characters. One way of referring to strings is through the use of the string syntax described in *Section 2.1.3, Lexemes*. In this method, we refer to the string `abc` by enclosing it in double quotes, such as, `"abc"`.

A second way is through the use of character blocks, the block syntax described in *Section 2.1.3, Lexemes*. In this method, we refer to the string `abc` by prefixing with the character `#`, a positive integer indicating the length, the letter `q`, and the characters of the string, for example, `#3qabc`.

A third way of referring to strings is to use the `listof` function. For example, we can denote the string `abc` by a term of the form `(listof #\a #\b #\c)`.
The advantage of the `listof` representation over the preceding representations is that it allows us to quantify over characters within strings. For example, the following sentence says that all 3 character strings beginning with `a` and ending with `a` are `nice`.

```
(= (character ?y) (nice (listof #\a ?y #\a)))
```

From this sentence, we can infer that various strings are `nice`.

```
(nice (listof #\a #\a #\a))
(nice "aba")
(nice #\Qaca)
```

2.7 Meta Knowledge

2.7.1 Naming Expressions

In formalizing knowledge about knowledge, we use a conceptualization in which expressions are treated as objects in the universe of discourse and in which there are functions and relations appropriate to these objects. In our conceptualization, we treat atoms as primitive objects with no subparts. We conceptualize complex expressions as lists of subexpressions (either atoms or other complex expressions). In particular, every complex expression is viewed as a list of its immediate subexpressions.

For example, we conceptualize the sentence `(not (p (+ a b c) d))` as a list consisting of the operator not and the sentence `(p (+ a b c) d)`. This sentence is treated as a list consisting of the relation constant `p` and the terms `(+ a b c)` and `d`. The first of these terms is a list consisting of the function constant `+` and the object constants `a`, `b` and `c`.

For Lisp programmers, this conceptualization is relatively obvious, but it departs from the usual conceptualization of formal languages taken in the mathematical theory of logic. It has the disadvantage that we cannot describe certain details of syntax such as parenthesization and spacing (unless we augment the conceptualization to include string representations of expressions as well). However, it is far more convenient for expressing properties of knowledge and inference than string-based conceptualizations.

In order to assert properties of expressions in the language, we need a way of referring to those expressions. There are two ways of doing this in FIPA KIF.

One way is to use the quote operator in front of an expression. To refer to the symbol john, we use the term `'john` or, equivalently, `(quote john)`. To refer to the expression `(p a b)`, we use the term `'(p a b)` or, equivalently, `(quote (p a b))`.

With a way of referring to expressions, we can assert their properties. For example, the following sentence ascribes to the individual named `john` the belief that the moon is made of a particular kind of blue cheese.

```
(believes john '(material moon stilton))
```

Note that, by nesting quotes within quotes, we can talk about quoted expressions. In fact, we can write towers of sentences of arbitrary heights, in which the sentences at each level talk about the sentences at the lower levels.

Since expressions are first-order objects, we can quantify over them, thereby asserting properties of whole classes of sentences. For example, we could say that Mary believes everything that John believes. This fact together with the preceding fact allows us to conclude that Mary also believes the moon to be made of blue cheese.

```
(= (believes john ?p) (believes mary ?p))
```

The second way of referring to expressions is FIPA KIF is to use the `listof` function. For example, we can denote a complex expression like `(p a b)` by a term of the form `(listof 'p 'a 'b)`, as well as `'(p a b)`.

The advantage of the `listof` representation over the quote representation is that it allows us to quantify over parts of expressions. For example, let us say that Lisa is more skeptical than Mary. She agrees with John, but only on the composition of things. The first sentence below asserts this fact without specifically mentioning moon or stilton. Thus, if we were to later discover that John thought the sun to be made of chili peppers, then Lisa would be constrained to believe this as well.

```
(= (believes john (listof 'material ?x ?y))
     (believes lisa (listof 'material ?x ?y)))
```

While the use of `listof` allows us to describe the structure of expressions in arbitrary detail, it is somewhat awkward. For example, the term `(listof 'material ?x ?y)` is somewhat awkward. Fortunately, we can eliminate this difficulty using the up arrow (`^`) and comma (`,`) characters. Rather than using the `listof` function constant as described above, we write the expression preceded by `^` and `,` in front of any subexpression that is not to be taken literally. For example, we would rewrite the preceding sentence as follows.

```
(= (believes john ^(material ,?x ,?y))
     (believes lisa ^(material ,?x ,?y)))
```

2.7.2 Types of Expressions

In order to facilitate the encoding of knowledge about FIPA KIF, the language includes type relations for the various syntactic categories defined in *Section 2.1, Syntax*.

For every individual variable *v*, there is an axiom asserting that it is indeed an individual variable. Each such axiom is a defining axiom for the `indvar` relation.

```
(indvar (quote v))
```

For every sequence variable *s*, there is an axiom asserting that it is a sequence variable. Each such axiom is a defining axiom for the `seqvar` relation.

```
(indvar (quote s))
```

For every word *w*, there is an axiom asserting that it is a word. Each such axiom is a defining axiom for the `word` relation.

```
(word (quote w))
```

Using this basic vocabulary and our vocabulary for lists, it is possible to define type relations for all types of syntactic expressions in FIPA KIF.

2.7.3 Changing Levels of Denotation

Logicians frequently use axiom schemata to encode (potentially infinite) sets of sentences with particular syntactic properties. As an example, consider the axiom schema shown below, where we are told that `r` stands for an arbitrary relation constant.

```
(= (and (r 0) (forall (?n) (= (r ?n) (r (1+ ?n))))) (forall (?n) (r ?n)))
```

This schema encodes infinitely many sentences, the principle of mathematical induction for named relations. The following sentences are instances:

```
(= (and (p 0) (forall (?n) (= (p ?n) (p (1+ ?n))))) (forall (?n) (p ?n)))

(= (and (q 0) (forall (?n) (= (q ?n) (q (1+ ?n))))) (forall (?n) (q ?n)))
```

Axiom schemata are differentiated from axioms due to the presence of meta-variables or other meta-linguistic notation (such as dots or star notation), together with conditions on the variables. They describe sentences in a language, but they are not themselves sentences in the language. As a result, they cannot be manipulated by procedures designed to process the language (presentation, storage, communication, deduction and so forth) but instead must be hard coded into those procedures.

As we have seen, it is possible in FIPA KIF to write expressions that describe FIPA KIF sentences. As it turns out, there is also a way to write sentences that assert the truth of the sentences so described. The effect of adding such meta-level sentences to a knowledge base is the same as directly including the (potentially infinite) set of described sentences in the knowledge base.

The use of such a language simplifies the construction of knowledge-based systems, since it obviates the need for building axiom schemata into deductive procedures. It also makes it possible for systems to exchange axiom schemata with each other and thereby promotes knowledge sharing.

The FIPA KIF truth predicate is called `wtr` (which stands for "weakly true"). For example, we can say that a sentence of the form `(= (p ?x) (q ?x))` is true by writing the following sentence.

```
(wtr '(= (p ?x) (q ?x)))
```

This may seem of limited utility, since we can just write the sentence denoted by the argument as a sentence in its own right. The advantage of the meta-notation becomes clear when we need to quantify over sentences, as in the encoding of axiom schemata. For example, we can say that every sentence of the form `(= p p)` is true with the following sentence. (The relation sentence can easily be defined in terms of `quote`, `listof`, `indvar`, `seqvar` and `word`.)

```
(= (sentence ?p) (wtr ^(= ,?p ,?p)))
```

Semantically, we would like to say that a sentence of the form `(wtr 'p)` is true if and only if the sentence `p` is true. Unfortunately, this causes serious problems. Equating a truth function with the meaning it ascribes to `wtr` quickly leads to paradoxes. The English sentence "This sentence is false" illustrates the paradox. We can write this sentence in FIPA KIF as shown below. The sentence, in effect, asserts its own negation.

```
(wtr (subst (name ^(subst (name x) ^x ^(truth ,x)))
      ^x
      ^(not (wtr (subst (name x) ^x ^(not (wtr ,x)))))))
```

No matter how we interpret this sentence, we get a contradiction. If we assume the sentence is true, then we have a problem because the sentence asserts its own falsity. If we assume the sentence is false, we also have a problem because the sentence then is necessarily true.

Fortunately, we can circumvent such paradoxes by slightly modifying the proposed definition of `wtr`. In particular, we have the following axiom schema for all `p` that do not contain any occurrences of `wtr`. For all p that do contain occurrences, `wtr` is false.

```
(<= (wtr 'p) p)
```

With this modified definition, the paradox described above disappears, yet we retain the ability to write virtually all useful axiom schemata as meta-level axioms.

From the point of view of formalizing truth, `wtr` is a not particularly useful, since it fails to cover those interesting cases where sentences contain the `truth` predicate. However, from the point of view of capturing axiom schemata not involving the `truth` predicate, it works just fine. Furthermore, unlike the solutions to the problem of formalizing truth, the framework presented here is easy for users to understand, and it is easy to implement.

Two other constants round out FIPA KIF's level-crossing vocabulary. The term `(denotation t)` denotes the object denoted by the object denoted by `t`. A quotation denotes the quoted expression; the denotation of any other object is `bottom`. As with `wtr`, the dentotation of a quoted expression is the embedded expression, provided that the expression does not contain any occurrences of denotation. Otherwise, the value is undefined.

```
(= (denotation 't) t)
```

The term `(name t)` denotes the standard name for the object denoted by the term t. The standard name for an expression `t` is `(quote t)`; the standard name for a non-expression is at the discretion of the user. (Note that there are only a countable number of terms in FIPA KIF, but there can be worlds with uncountable cardinality; consequently, it is not always possible for every object to have a unique name.)

3 References

[FIPA00061] FIPA ACL Message Structure Specification. Foundation for Intelligent Physical Agents, 2000. `http://www.fipa.org/specs/fipa00061/`

[ISO646] Information Technology – ISO 7-bit Coded Character Set for Information Interchange, ISO 646:1991. International Standards Organisation, 1991. `http://www.iso.ch/cate/d4777.html`

[ISO10646] Information Technology – Universal Multiple-Octet Coded Character Set (UCS), ISO 10646-1:1993. International Standards Organisation, 1993. `http://www.iso.ch/cate/d18741.html`

[ISO14481] Information Technology – Conceptual Schema Modeling Facilities (CSMF), ISO 14481:1998. International Standards Organisation, 1998.

4 Informative Annex A — Examples

1. The following FIPA ACL message with the content in FIPA KIF informs that `database-agent1` specializes handling the sentence `'(price ,?x ,?y)` where `?x` is a constant and `?y` is a number. Note that the communicative act `inform` takes a proposition as its content.

```
(inform
  :sender
    (agent-identifier
      :name database-agent1)
  :receiver
    (agent-identifier
      :name facilitator1)
  :language FIPA-KIF
  :ontology ec-ontology
  :content
    (<= (specialist agent1 '(price ,?x ,?y))
      (constant ?x)
      (number ?y)))
```

2. This message informs that `database-agent1` conforms to the conformance profile `database-system` (see [ANSkif] for conformance details).

```
(inform
  :sender
    (agent-identifier
      :name database-agent1)
  :receiver
    (agent-identifier
      :name facilitator1)
  :language FIPA-KIF
  :ontology ec-ontology
  :content
     (conformance-profile databae-agent1 database-system))
```

3. This message informs that `database-agent1`'s conformance dimensions are `horn`, `non-recursive`, `simple`, `first-order`, `universal` and `baselevel` (see [ANSkif] for conformance details).

```
(inform
  :sender
    (agent-identifier
      :name database-agent1)
  :receiver
    (agent-identifier
      :name facilitator1)
  :language FIPA-KIF
  :ontology ec-ontology
  :content
    (conformance-dimension databae-agent1
      (horn non-recursive simple first-order universal baselevel)))
```

4. This message denies the message of the example in 1. Note that the communicative act `disconfirm` takes a proposition as its content.

```
(disconfirm
  :sender
    (agent-identifier
      :name database-agent1)
  :receiver
    (agent-identifier
      :name facilitator1)
```

```
  :language FIPA-KIF
  :ontology ec-ontology
  :content
    (<= (specialist agent1 '(price ,?x ,?y))
      (constant ?x) (number ?y)))
```

5. This message expresses a query by the agent, `facilitator1` to the agent, `database-agent1`. Note that the communicative act `query-ref` takes an object as its content.

```
(query-ref
  :sender
    (agent-identifier
      :name facilitator1)
  :receiver
    (agent-identifier
      :name database-agent1)
  :language FIPA-KIF
  :ontology ec-ontology
  :content
    (kappa (?make ?door ?price)
      (and (car ?car) (make ?car ?make)
      (doors ?car ?doors) (price ?car ?price))))
```

6. This message expresses the answer to the query of the previous example by the agent, `database-agent1` to the agent, `facilitator1`:

```
(inform
  :sender
    (agent-identifier
      :name database-agent1)
  :receiver
    (agent-identifier
      :name facilitator1)
  :language FIPA-KIF
  :ontology ec-ontology
  :content
    (= (kappa (?make ?door ?price)
      (and (car ?car) (make ?car ?make)
        (doors ?car ?doors) (price ?car ?price)))
      '((Mercedes 4 100,000) (Honda 2 20,000) (Toyota 4 25,000))))
```