

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.

19 **Foreword**

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

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

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

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

38 **Contents**

39	1	Scope	1
40	2	FIPA KIF Specification	2
41	2.1	Syntax	2
42	2.1.1	Introduction	2
43	2.1.2	Characters	3
44	2.1.3	Lexemes	3
45	2.1.4	Expressions	5
46	2.2	Basics	8
47	2.2.1	Introduction	8
48	2.2.2	Bottom	9
49	2.2.3	Functional Terms	9
50	2.2.4	Relational Sentences	9
51	2.2.5	Equations and Inequalities	9
52	2.2.6	True and False	9
53	2.3	Logic	10
54	2.3.1	Logical Terms	10
55	2.3.2	Logical Sentences	10
56	2.3.3	Quantified Sentences	10
57	2.3.4	Definitions	11
58	2.4	Numbers	12
59	2.4.1	Introduction	12
60	2.4.2	Functions on Numbers	12
61	2.4.3	Relations on Numbers	14
62	2.5	Lists	14
63	2.6	Characters and Strings	16
64	2.6.1	Characters	16
65	2.6.2	Strings	17
66	2.7	Meta Knowledge	17
67	2.7.1	Naming Expressions	17
68	2.7.2	Types of Expressions	18
69	2.7.3	Changing Levels of Denotation	19
70	3	References	21
71	4	Informative Annex A — Examples	22
72			

72 **1 Scope**

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

75

76 Expression of objects as terms.

77

78 Expression of propositions as sentences.

79

80 FIPA KIF currently has no specific way to expresses actions.

81

81 2 FIPA KIF Specification

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

85
86 FIPA KIF is *not* intended as a primary language for interaction with human users (though it can be used for this
87 purpose). Different computer systems can interact with their users in whatever forms are most appropriate to their
88 applications (for example, Prolog, conceptual graphs, natural language and so forth).

89
90 FIPA KIF is also *not* intended as an internal representation for knowledge *within* computer systems or within closely
91 related sets of computer systems (though the language can be used for this purpose as well). Typically, when a
92 computer system reads a knowledge base in FIPA KIF, it converts the data into its own internal form (specialized
93 pointer structures, arrays, etc.) and all computation is done using these internal forms. When the computer system
94 needs to communicate with another computer system, it maps its internal data structures into FIPA KIF before message
95 transfer.

96
97 The following categorical features are essential to the design of FIPA KIF:

98
99 The language has declarative semantics. It is possible to understand the meaning of expressions in the language
100 without appeal to an interpreter for manipulating those expressions. In this way, FIPA KIF differs from other
101 languages that are based on specific interpreters, such as Emycin and Prolog.

102
103 The language is logically comprehensive. At its most general, it provides for the expression of arbitrary logical
104 sentences. In this way, it differs from relational database languages (like SQL) and logic programming languages
105 (like Prolog).

106
107 The language provides for the representation of knowledge about knowledge. This allows the user to make
108 knowledge representation decisions explicit and permits the user to introduce new knowledge representation
109 constructs without changing the language.

110
111 In addition to these essential features, FIPA KIF is designed to maximize the following additional features (to the extent
112 possible while preserving the preceding features):

113
114 **Implementability.** Although FIPA KIF is not intended for use within programs as a representation or
115 communication language, it should be usable for that purpose if so desired.

116
117 **Readability.** Although FIPA KIF is not intended primarily as a language for interaction with humans, human
118 readability facilitates its use in describing representation language semantics, its use as a publication language for
119 example knowledge bases, its use in assisting humans with knowledge base translation problems, etc.

120
121 Unless otherwise stated, all terms and definitions are taken from [ISO10646] and [ISO14481].
122

123 2.1 Syntax

124 2.1.1 Introduction

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

130
131 In this section, the syntax of FIPA KIF is presented using a modified BNF notation. All nonterminals and BNF
132 punctuation are written in boldface, while characters in FIPA KIF are expressed in plain font. The notation {x1, ..., xn}

133 means the set of terminals x_1, \dots, x_n . The notation **[nonterminal]** means zero or one instances of **nonterminal**;
 134 **nonterminal*** means zero or more occurrences; **nonterminal+** means one or more occurrences; **nonterminal ^ n**
 135 means **n** occurrences. The notation **nonterminal1 - nonterminal2** refers to all of the members of **nonterminal1** except
 136 for those in **nonterminal2**. The notation **int (n)** denotes the decimal representation of integer **n**. The nonterminals
 137 **space**, **tab**, **return**, **linefeed** and **page** refer to the characters corresponding to ASCII codes 32, 9, 13, 10, and 12,
 138 respectively. The nonterminal **character** denotes the set of all 128 ASCII characters. The nonterminal **empty** denotes
 139 the empty string.
 140

141 2.1.2 Characters

142 The alphabet of FIPA KIF consists of 7 bit blocks of data. In this document, we refer to FIPA KIF data blocks via their
 143 usual ASCII encodings as characters as given in [ISO646].
 144

145 FIPA KIF characters are classified as upper case letters, lower case letters, digits, alpha characters (non-alphabetic
 146 characters that are used in the same way that letters are used), special characters, white space, and other characters
 147 (every ASCII character that is not in one of the other categories):
 148

```

149 upper      ::=  A | B | C | D | E | F | G | H | I | J | K | L | M |
150             N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
151
152 lower      ::=  a | b | c | d | e | f | g | h | i | j | k | l | m |
153             n | o | p | q | r | s | t | u | v | w | x | y | z |
154
155 digit      ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
156
157 alpha      ::=  ! | $ | % | & | * | + | - | . | / | < | = |   | ? |
158             @ | _ | ~ |
159
160 special    ::=  " | # | ' | ( | ) | , | \ | ^ | '
161
162 white      ::=  space | tab | return | linefeed | page
163

```

164 A normal character is either an upper case character, a lower case character, a digit, or an alpha character.
 165

```

166 normal     ::=  upper | lower | digit | alpha
167

```

168 2.1.3 Lexemes

169 The process of converting characters into lexemes is called lexical analysis. The input to this process is a stream of
 170 characters, and the output is a stream of lexemes.
 171

172 The function of a lexical analyser is cyclic. It reads characters from the input string until it encounters a character that
 173 cannot be combined with previous characters to form a legal lexeme. When this happens, it outputs the lexeme
 174 corresponding to the previously read characters. It then starts the process over again with the new character. White
 175 space causes a break in the lexical analysis process but otherwise is discarded.
 176

177 There are five types of lexemes in FIPA KIF: special lexemes, words, character references, character strings and
 178 character blocks. Each special character forms its own lexeme. It cannot be combined with other characters to form
 179 more complex lexemes, except through the escape' syntax described below.
 180

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

```

182
183 word ::= normal | word normal | word\character
184

```

185 It is possible to include the character \ in a word by preceding it by another occurrence of \, that is, two contiguous
 186 occurrences of \ are interpreted as a single occurrence. For example, the string A\\\'B corresponds to a word
 187 consisting of the four characters A, \, ', and B.
 188

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

193

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

196

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

198

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

201

```
202 string ::= "quotable"
```

203

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

205

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

207

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

213

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

215

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

218

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

221

```
222 variable ::= indvar | seqvar
```

223

```
224 indvar ::= ?word
```

225

```
226 seqvar ::= @word
```

227

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

229

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

231

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

233

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

235

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

237

```
238 termop ::= value | listof | quote | if
```

239

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

242

```
243 defop ::= defobject | defunction | defrelation | deflogical |  
244 := | :- | :<= | :=
```

245

246 All other words are called **constants**:

247
 248 constant ::= word - variable - operator
 249

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

251 **Object constants** are used to denote individual objects.

252 **Function constants** denote functions on those objects.

253 **Relation constants** denote relations.

254 **Logical constants** express conditions about the world and are either true or false.

255
 256
 257
 258 FIPA KIF is unusual among logical languages in that there is no syntactic distinction among these four types of
 259 constants; any constant can be used where any other constant can be used. The differences between these categories
 260 of constants is entirely semantic.
 261
 262
 263

264 2.1.4 Expressions

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

267 **Terms** are used to denote objects in the world being described.

268 **Sentences** are used to express facts about the world.

269 **Definitions** are used to define constants.

270
 271
 272 There are nine types of terms in FIPA KIF: individual variables, constants, character references, character strings,
 273 character blocks, functional terms, list terms, quotations, and logical terms. Individual variables, constants, character
 274 references, strings and blocks were discussed earlier.
 275
 276

277 term ::= indvar | constant | charref | string | block |
 278 funterm | listterm | quoterm | logterm
 279
 280

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

284 funterm ::= (constant term* [seqvar])
 285
 286

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

289 funterm ::= (value term term* [seqvar])
 290
 291

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

294 listterm ::= (listof term* [seqvar])
 295
 296

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

301 quoterm ::= (quote listexpr) | 'listexpr
 302

```

303
304     listexpr    ::=  atom | (listexpr*)
305
306     atom       ::=  word | charref | string | block
307

```

308 **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.

```

312
313     logterm     ::=  (if logpair+ [term])
314
315     logpair    ::=  sentence term
316
317     logterm     ::=  (cond logitem*)
318
319     logitem    ::=  (sentence term)
320

```

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

```

323
324     sentence   ::=  constant | equation | inequality |
325                   relsent | logsent | quantsent
326

```

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

```

328
329     equation   ::=  (= term term)
330
331     inequality ::=  (/= term term)
332

```

333 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.

```

336
337     relsent    ::=  (constant term* [seqvar])
338

```

339 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.

```

341
342     relsent    ::= (holds term term* [seqvar])
343

```

344 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.

348 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.

```

356
357     logsent    ::= (not sentence) |
358                   (and sentence*) |
359                   (or sentence*) |
360                   (= sentence* sentence) |
361                   (<= sentence sentence*) |

```

362 (<= sentence sentence)

363

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

368

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

370

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

372

373

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

380

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

386

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

388

```
389     unrestricted ::= (defobject constant [string] sentence*)
390                  | (deffunction constant [string] sentence*)
391                  | (defrelation constant [string] sentence*)
392                  | (deflogical constant [string] sentence*)
```

390

391

392

393

```
394     complete    ::= (defobject constant [string] := term)
395                  | (deffunction constant (indvar* [seqvar]) [string] := term)
396                  | (defrelation constant (indvar* [seqvar]) [string] := sentence)
397                  | (deflogical constant [string] := sentence)
```

395

396

397

398

```
399     partial     ::= (defobject constant [string] :- indvar <= sentence)
400                  | (defobject constant [string] :- indvar := sentence)
401                  | (deffunction constant (indvar* [seqvar])
402                    [string] :- indvar <= sentence)
403                  | (deffunction constant (indvar* [seqvar])
404                    [string] :- indvar := sentence)
405                  | (defrelation constant (indvar* [seqvar])
406                    [string] <= sentence)
407                  | (defrelation constant (indvar* [seqvar])
408                    [string] := sentence)
409                  | (deflogical constant [string] <= sentence)
410                  | (deflogical constant [string] := sentence)
```

399

400

401

402

403

404

405

406

407

408

409

410

411

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

413

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

414

415

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

419

420 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,
 421 not a *sequence*; and, therefore, the order of forms within a knowledge base is unimportant. Order *may* have heuristic

422 value to deductive programs by suggesting an order in which to use those sentences; however, this implicit approach to
 423 knowledge exchange lies outside of the definition of FIPA KIF.
 424

425 2.2 Basics

426 2.2.1 Introduction

427 The basis for the semantics of FIPA KIF is a conceptualization of the world in terms of objects and relations among
 428 those objects.
 429

430 A *universe of discourse* is the set of all objects presumed or hypothesized to exist in the world. The notion of object
 431 used here is quite broad. Objects can be concrete, for example, a specific carbon atom, Confucius, the Sun or abstract,
 432 such as the number 2, the set of all integers or the concept of justice. Objects can be primitive or composite, for
 433 example, a circuit that consists of many sub circuits. Objects can even be fictional, for example, a unicorn, Sherlock
 434 Holmes, etc.
 435

436 Different users of a declarative representation language, like FIPA KIF, are likely to have different universes of
 437 discourse. FIPA KIF is conceptually promiscuous in that it does not require every user to share the same universe of
 438 discourse. On the other hand, FIPA KIF is conceptually grounded in that every universe of discourse is required to
 439 include certain basic objects.
 440

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

442 All numbers, real and complex.

443 All ASCII characters.

444 All finite strings of ASCII characters.

445 Words and the things they represent.

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

447 Bottom. A distinguished object that occurs as the value of a partial when that function is applied to arguments for
 448 which the function make no sense.
 449

450 Remember, that to these basic elements, the user can add whatever non-basic objects seem useful.
 451

452 In FIPA KIF, relationships among objects take the form of relations. Formally, a relation is defined as an arbitrary set of
 453 finite lists of objects (of possibly varying lengths). Each list is a selection of objects that jointly satisfy the relation. For
 454 example, the < relation on numbers contains the list <2,3>, indicating that 2 is less than 3.
 455

456 A function is a special kind of relation. For every finite sequence of objects (called the arguments), a function associates
 457 a unique object (called the value). More formally, a function is defined as a set of finite lists of objects, one for each
 458 combination of possible arguments. In each list, the initial elements are the arguments, and the final element is the
 459 value. For example, the $1+$ function contains the list <2, 3>, indicating that integer successor of 2 is 3.
 460

461 Note that both functions and relations are defined as sets of lists. In fact, every function is a relation. However, not
 462 every relation is a function. In a function, there cannot be two lists that disagree on only the last element, since this
 463 would be tantamount to the function having two values for one combination of arguments. By contrast, in a relation,
 464 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
 465 $1+$ 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
 466 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
 467 forth.
 468
 469
 470
 471
 472
 473
 474

475 Many mathematicians require that functions and relations have fixed arity, that is, they require that all of the lists
 476 comprising a relation have the same length. The definitions here allow for relations with variable arity; it is perfectly
 477 acceptable for a function or a relation to contain lists of different lengths. For example, the relation `<` contains the lists
 478 `<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
 479 flexibility is not essential, but it is extremely convenient and poses no significant theoretical problems.
 480

481 2.2.2 Bottom

482 In FIPA KIF, all functions are total, that is, there is a value for every combination of arguments. In order to allow a user
 483 to express the idea that a function is not meaningful for certain arguments, FIPA KIF assumes that there is a special
 484 "undefined" object in the universe and provides the object constant `bottom` to refer to this object.
 485

486 2.2.3 Functional Terms

487 The value of a functional term without a terminating sequence variable is obtained by applying the function denoted by
 488 the function constant in the term to the objects denoted by the arguments.
 489

490 For example, the value of the term `(+ 2 3)` is obtained by applying the addition function (the function denoted by `+`) to
 491 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
 492 the object constant `5`.
 493

494 If a functional term has a terminating sequence variable, the value is obtained by applying the function to the sequence
 495 of arguments formed from the values of the terms that precede the sequence variable and the values in the sequence
 496 denoted by the sequence variable.
 497

498 Assume, for example, that the sequence variable `@1` has as value the sequence 2, 3, 4. Then, the value of the term `(+
 499 1 @1)` is obtained by applying the addition function to the numbers 1, 2, 3, and 4 to obtain the value 10, which is the
 500 value of the object constant `10`.
 501

502 2.2.4 Relational Sentences

503 A simple relational sentence without a terminating sequence variable is true if and only if the relation denoted by the
 504 relation constant in the sentence is true of the objects denoted by the arguments. Equivalently, viewing a relation as a
 505 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
 506 arguments is a member of the set of tuples denoted by the relation constant.
 507

508 If a relational sentence terminates in a sequence variable, the sentence is true if and only if the relation contains the
 509 tuple consisting of the values of the terms that precede the sequence variable together with the objects in the sequence
 510 denoted by the variable.
 511

512 2.2.5 Equations and Inequalities

513 An equation is true if and only if the terms in the equation refer to the same object in the universe of discourse. An
 514 inequality is true if and only if the terms in the equation refer to distinct objects in the universe of discourse.
 515

516 2.2.6 True and False

517 The truth-value of true is `true`, and the truth-value of false is `false`.
 518

519 2.3 Logic

520 2.3.1 Logical Terms

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

523
524 If none of the embedded sentences of a logical term involving the `if` operator is true and there is an isolated term at the
525 end, the value of the conditional term is the value of that isolated term. For example, if the object constant `a` denotes a
526 number, then the term `(if (a 0) a (- a))` denotes the absolute value of that number.

527
528 If none of the embedded sentences is true and there is no isolated term at the end, the value is undefined (i.e. bottom).
529 In other words, the term `(if (p a) a)` is equivalent to `(if (p a) a bottom)`.

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

532
533 If none of the embedded sentences is true, the value is undefined. In other words, the term `(cond ((p a) a))` is
534 equivalent to `(cond ((p a) a) (true bottom))`.

535

536 2.3.2 Logical Sentences

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

538

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

540

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

542

543 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
544 any of the antecedents is false, then the implication as a whole is true, regardless of the truth-value of the consequent.

545

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

547

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

549

550 2.3.3 Quantified Sentences

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

553

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

556

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

562

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

565

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

568

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

571 In other words, the meaning of free variables is determined by the way in which FIPA KIF is used. It cannot be
 572 unambiguously defined within FIPA KIF itself. To be certain of the usage in all contexts, use explicit quantifiers.
 573

574 2.3.4 Definitions

575 The definitional operators in FIPA KIF allow us to state sentences that are true "by definition" in a way that distinguishes
 576 them from sentences that express contingent properties of the world. Definitions have no truth-values in the usual
 577 sense; they are so because we say that they are so.
 578

579 On the other hand, definitions have content: sentences that allow us to derive other sentences as conclusions. In FIPA
 580 KIF, every definition has a corresponding set of sentences, called the content of the definition.
 581

582 The `defobject` operator is used to define objects. The legal forms are shown below, together with their content. In the
 583 first case, the content is the equation involving the object constant in the definition with the defining term. In the second
 584 case, the content is the conjunction of the constituent sentences.
 585

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

```
589 (defobject s p1 ... pn)
590   (and p1 ... pn)
```

```
592 (defobject s :- v := p)
593   (= (= s v) p)
```

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

597
 598 The `deffunction` operator is used to define functions. Again, the legal forms are shown below, together with their
 599 defining axioms. In the first case, the content is the equation involving the term formed from the function constant in the
 600 definition and the variables in its argument list and the defining term. In the second case, as with object definitions, the
 601 content is the conjunction of the constituent sentences.
 602

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

```
606 (deffunction f p1 ...pn)
607   (and p1 ...pn)
```

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

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

614
 615 The `defrelation` operator is used to define relations. The legal forms are shown below, together with their defining
 616 axioms. In the first case, the content is the equivalence relating the relational sentence formed from the relation
 617 constant in the definition and the variables in its argument list and the defining sentence. In the second case, as with
 618 object and function definitions, the content is the conjunction of the constituent sentences.
 619

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

```
623 (defrelation r p1 ...pn)
624   (and p1 ...pn)
```

```
626 (defrelation r (v1 ... vn) := p)
627   (= (r v1 ... vn) p))
```

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

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

630
631

632 2.4 Numbers

633 2.4.1 Introduction

634 The referent of every numerical constant in FIPA KIF is assumed to be the number for which that constant is the base
635 10 representation. Among other things, this means that we can infer inequality of all distinct numerical constants, i.e. for
636 every t_1 and distinct t_2 the following sentence is true.

637
638 (/= t1 t2)
639

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

642 Note that this does mean that it is incorrect to apply these functions and relations to terms other than numbers. For
643 example, a non-numerical term may refer to a number, for example, the term two may be defined to be the same as the
644 number 2 in which case it is perfectly proper to write (+ two two).
645

646 The user may also want to extend these functions and relations to apply to objects other than numbers, for example,
647 sets and lists.
648
649

650 2.4.2 Functions on Numbers

651 *
652 If t_1, \dots, t_n denote numbers, then the term ($* t_1 \dots t_n$) denotes the product of those numbers.

653 +
654 If t_1, \dots, t_n are numerical constants, then the term ($+ t_1 \dots t_n$) denotes the sum t of the numbers
655 corresponding to those constants.
656

657 -
658 If t and t_1, \dots, t_n denote numbers, then the term ($- t t_1 \dots t_n$) denotes the difference between the number
659 denoted by t and the numbers denoted by t_1 through t_n . An exception occurs when $n=0$, in which case the term
660 denotes the negation of the number denoted by t .
661

662 /
663 If t_1, \dots, t_n are numbers, then the term ($/ t_1 \dots t_n$) denotes the result t obtained by dividing the number
664 denoted by t_1 by the numbers denoted by t_2 through t_n . An exception occurs when $n=1$, in which case the term
665 denotes the reciprocal t of the number denoted by t_1 .
666

667 1+
668 The term ($1+ t$) denotes the sum of the object denoted by t and 1.
669

670 (deffunction 1+ (?x) := (+ ?x 1))
671

672 1-
673 The term ($1- t$) denotes the difference of the object denoted by t and 1.
674

675 (deffunction 1- (?x) := (- ?x 1))
676

677 abs
678 The term ($abs t$) denotes the absolute value of the object denoted by t .
679

680 (deffunction abs (?x) := (if (= ?x 0) ?x (- ?x)))
681

682

683 `ceiling`

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

686

687 `denominator`

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

689

690 `expt`

691 The term `(expt t1 t2)` denotes the object denoted by t_1 raised to the power the object denoted by t_2 .

692

693 `floor`

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

695

696 `gcd`

697 The term `(gcd t1 ... tn)` denotes the greatest common divisor of the objects denoted by t_1 through t_n .

698

699 `imagpart`

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

701

702 `lcm`

703 The term `(lcm t1 ... tn)` denotes the least common multiple of the objects denoted by t_1, \dots, t_n .

704

705 `log`

706 The term `(log t1 t2)` denotes the logarithm of the object denoted by t_1 in the base denoted by t_2 .

707

708 `max`

709 The term `(max t1 ... tk)` denotes the largest object denoted by t_1 through t_n .

710

711 `min`

712 The term `(min t1 ... tk)` denotes the smallest object denoted by t_1 through t_n .

713

714 `mod`

715 The term `(mod t1 t2)` denotes the root of the object denoted by t_1 modulo the object denoted by t_2 . The result
 716 will have the same sign as denoted by t_1 .

717

718 `numerator`

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

720

721 `realpart`

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

723

724 `rem`

725 The term `(rem t1 t2)` denotes the remainder of the object denoted by t_1 divided by the object denoted by t_2 .

726

726 The result has the same sign as the object denoted by t_2 .

727

728 `round`

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

731

732 `sqrt`

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

734

735 `truncate`

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

737

738 **2.4.3 Relations on Numbers**

739 integer

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

741

742 real

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

744

745 complex

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

747

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

749

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

751

752 `(defrelation rational (?x) :=`753 `(exists (?y) (and (integer ?y) (integer (* ?x ?y))))))`

754

755 approx

756 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`.

757

758

759

760

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

762

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

764

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

766

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

768

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

770

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

772

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

774

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

776

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

778

779 **2.5 Lists**

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

781

782 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`.

783

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

785

786 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.

787

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

789

790

791

792

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

```
796 (defobject nil := (listof))
797
798 (defrelation null (?l) := (= ?l (listof)))
799
800 (defrelation single (?l) := (exists (?x) (= ?l (listof ?x))))
801
802 (defrelation double (?l) := (exists (?x ?y) (= ?l (listof ?x ?y))))
803
804 (defrelation triple (?l) := (exists (?x ?y ?z) (= ?l (listof ?x ?y ?z))))
805
```

806
 807 The functions `first`, `rest`, `last` and `butlast` each take a single list as argument and select individual items or sub
 808 lists from those lists.

```
809 (deffunction first (?l) := (if (= (listof ?x @items) ?l) ?x)
810
811 (deffunction rest (?l) :=
812   (cond ((null ?l) ?l)
813         ((= ?l (listof ?x @items)) (listof @items))))
814
815 (deffunction last (?l) :=
816   (cond ((null ?l) bottom) ((null (rest ?l)) (first ?l))
817         (true (last (rest ?l)))))
818
819 (deffunction butlast (?l) :=
820   (cond ((null ?l) bottom) ((null (rest ?l)) nil)
821         (true (cons (first ?l) (butlast (rest ?l))))))
822
```

823
 824 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
 825 `t1` is either the first item of that list or an item in the rest of the list.

```
826 (defrelation item (?x ?l) :=
827   (and (list ?l) (not (null ?l))
828        (or (= ?x (first ?l)) (item ?x (rest ?l)))))
829
```

830
 831 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
 832 `t2`.

```
833 (defrelation sublist (?l1 ?l2) :=
834   (and (list ?l1) (list ?l2)
835        (or (= ?l1 ?l2) (sublist ?l1 (rest ?l2)))))
836
```

837
 838 The function `cons` adds the object specified as its first argument to the front of the list specified as its second argument.

```
839 (deffunction cons (?x ?l) :=
840   (if (= ?l (listof @l)) (listof ?x @l)))
841
```

842
 843 The function `append` adds the items in the list specified as its first argument to the list specified as its second
 844 argument. The function `revappend` is similar, except that it adds the items in reverse order.

```
845 (deffunction append (?l1 ?l2) :=
846   (cond ((null ?l1) (if (list ?l2) ?l2))
847         ((list ?l1) (cons (first ?l1) (append (rest ?l1) ?l2)))))
848
849 (deffunction revappend (?l1 ?l2) :=
850   (cond ((null ?l1) (if (list ?l2) ?l2))
851         ((list ?l1) (revappend (rest ?l1) (cons (first ?l1) ?l2)))))
852
```

853

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

856

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

858

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

861

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

863

```
864 (deffunction remove (?x ?l) :=
```

865

```
      (cond ((null ?l) nil) ((and (= ?x (first ?l)) (list ?l))
```

866

```
        (remove ?x (rest ?l)))
```

867

```
        ((list ?l) (cons ?x (remove ?x (rest ?l))))))
```

868

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

871

```
872 (deffunction subst (?x ?y ?z) :=
```

873

```
      (cond ((= ?y ?z) ?x) ((null ?z) nil)
```

874

```
        ((list ?z) (cons (subst ?x ?y (first ?z))
```

875

```
          (subst ?x ?y (rest ?z))))
```

876

```
        (true ?z)))
```

877

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

881

```
882 (deffunction length (?l) :=
```

883

```
      (cond ((null ?l) 0)
```

884

```
        ((list ?l) (1+ (length (rest ?l))))))
```

885

```
886 (deffunction nth (?l ?n) :=
```

887

```
      (cond ((= ?n 1) (first ?l))
```

888

```
        ((and (list ?l) (positive ?n)) (nth (rest ?l) (1- ?n))))))
```

889

```
890 (deffunction nthrest (?l ?n) :=
```

891

```
      (cond ((= ?n 0) (if (list ?l) ?l))
```

892

```
        ((and (list ?l) (positive ?n)) (nthrest (rest ?l) (1- ?n))))))
```

893

894

2.6 Characters and Strings

895

2.6.1 Characters

896

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

899

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

908

```
(= (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.

961 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,
 962 equivalently, (quote john). To refer to the expression (p a b), we use the term '(p a b) or, equivalently, (quote
 963 (p a b)).

964

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

967

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

969

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

972

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

976

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

978

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

981

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

987

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

  989 (believes lisa (listof 'material ?x ?y)))
```

990

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

996

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

  998 (believes lisa ^(material ,?x ,?y)))
```

999

1000 2.7.2 Types of Expressions

1001 In order to facilitate the encoding of knowledge about FIPA KIF, the language includes type relations for the various
 1002 syntactic categories defined in *Section 2.1, Syntax*.

1003

1004 For every individual variable *v*, there is an axiom asserting that it is indeed an individual variable. Each such axiom is a
 1005 defining axiom for the *indvar* relation.

1006

```
1007 (indvar (quote v))
```

1008

1009 For every sequence variable *s*, there is an axiom asserting that it is a sequence variable. Each such axiom is a defining
 1010 axiom for the *seqvar* relation.

1011

```
1012 (indvar (quote s))
```

1013

1014 For every word *w*, there is an axiom asserting that it is a word. Each such axiom is a defining axiom for the *word*
 1015 relation.

1016

(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 w_{tr} (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 $(w_{tr} 'p)$ 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 w_{tr} 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)))))))

1072 No matter how we interpret this sentence, we get a contradiction. If we assume the sentence is true, then we have a
 1073 problem because the sentence asserts its own falsity. If we assume the sentence is false, we also have a problem
 1074 because the sentence then is necessarily true.

1075

1076 Fortunately, we can circumvent such paradoxes by slightly modifying the proposed definition of `wtr`. In particular, we
 1077 have the following axiom schema for all `p` that do not contain any occurrences of `wtr`. For all `p` that do contain
 1078 occurrences, `wtr` is false.

1079

1080 `(=<= (wtr 'p) p)`

1081

1082 With this modified definition, the paradox described above disappears, yet we retain the ability to write virtually all useful
 1083 axiom schemata as meta-level axioms.

1084

1085 From the point of view of formalizing truth, `wtr` is a not particularly useful, since it fails to cover those interesting cases
 1086 where sentences contain the `truth` predicate. However, from the point of view of capturing axiom schemata not
 1087 involving the `truth` predicate, it works just fine. Furthermore, unlike the solutions to the problem of formalizing truth,
 1088 the framework presented here is easy for users to understand, and it is easy to implement.

1089

1090 Two other constants round out FIPA KIF's level-crossing vocabulary. The term `(denotation t)` denotes the object
 1091 denoted by the object denoted by `t`. A quotation denotes the quoted expression; the denotation of any other object is
 1092 `bottom`. As with `wtr`, the denotation of a quoted expression is the embedded expression, provided that the expression
 1093 does not contain any occurrences of denotation. Otherwise, the value is undefined.

1094

1095 `(= (denotation 't) t)`

1096

1097 The term `(name t)` denotes the standard name for the object denoted by the term `t`. The standard name for an
 1098 expression `t` is `(quote t)`; the standard name for a non-expression is at the discretion of the user. (Note that there are
 1099 only a countable number of terms in FIPA KIF, but there can be worlds with uncountable cardinality; consequently, it is
 1100 not always possible for every object to have a unique name.)

1101

3 References

- 1101
- 1102 [FIPA00061] FIPA ACL Message Structure Specification. Foundation for Intelligent Physical Agents, 2000.
- 1103 <http://www.fipa.org/specs/fipa00061/>
- 1104 [ISO646] Information Technology – ISO 7-bit Coded Character Set for Information Interchange, ISO 646:1991.
- 1105 International Standards Organisation, 1991.
- 1106 <http://www.iso.ch/cate/d4777.html>
- 1107 [ISO10646] Information Technology – Universal Multiple-Octet Coded Character Set (UCS), ISO 10646-1:1993.
- 1108 International Standards Organisation, 1993.
- 1109 <http://www.iso.ch/cate/d18741.html>
- 1110 [ISO14481] Information Technology – Conceptual Schema Modeling Facilities (CSMF), ISO 14481:1998.
- 1111 International Standards Organisation, 1998.
- 1112
- 1113

1113 4 Informative Annex A — Examples

- 1114 1. The following FIPA ACL message with the content in FIPA KIF informs that database-agent1 specializes
 1115 handling the sentence '(price ,?x ,?y) where ?x is a constant and ?y is a number. Note that the
 1116 communicative act inform takes a proposition as its content.

```
1117 (inform
1118   :sender
1119     (agent-identifier
1120      :name database-agent1)
1121   :receiver
1122     (agent-identifier
1123      :name facilitator1)
1124   :language FIPA-KIF
1125   :ontology ec-ontology
1126   :content
1127     (<= (specialist agent1 '(price ,?x ,?y))
1128        (constant ?x)
1129        (number ?y)))
```

- 1130 2. This message informs that database-agent1 conforms to the conformance profile database-system (see
 1131 [ANSkif] for conformance details).

```
1132 (inform
1133   :sender
1134     (agent-identifier
1135      :name database-agent1)
1136   :receiver
1137     (agent-identifier
1138      :name facilitator1)
1139   :language FIPA-KIF
1140   :ontology ec-ontology
1141   :content
1142     (conformance-profile databae-agent1 database-system))
```

- 1143 3. This message informs that database-agent1's conformance dimensions are horn, non-recursive, simple,
 1144 first-order, universal and baselevel (see [ANSkif] for conformance details).

```
1145 (inform
1146   :sender
1147     (agent-identifier
1148      :name database-agent1)
1149   :receiver
1150     (agent-identifier
1151      :name facilitator1)
1152   :language FIPA-KIF
1153   :ontology ec-ontology
1154   :content
1155     (conformance-dimension databae-agent1
1156      (horn non-recursive simple first-order universal baselevel)))
```

- 1160 4. This message denies the message of the example in 1. Note that the communicative act disconfirm takes a
 1161 proposition as its content.

```
1162 (disconfirm
1163   :sender
1164     (agent-identifier
1165      :name database-agent1)
1166   :receiver
1167     (agent-identifier
1168      :name facilitator1)
```

```

1172 :language FIPA-KIF
1173 :ontology ec-ontology
1174 :content
1175   (<= (specialist agent1 '(price ,?x ,?y))
1176     (constant ?x) (number ?y)))
1177
1178 5. This message expresses a query by the agent, facilitator1 to the agent, database-agent1. Note that the
1179 communicative act query-ref takes an object as its content.
1180
1181 (query-ref
1182   :sender
1183     (agent-identifier
1184       :name facilitator1)
1185   :receiver
1186     (agent-identifier
1187       :name database-agent1)
1188   :language FIPA-KIF
1189   :ontology ec-ontology
1190   :content
1191     (kappa (?make ?door ?price)
1192       (and (car ?car) (make ?car ?make)
1193         (doors ?car ?doors) (price ?car ?price))))
1194
1195 6. This message expresses the answer to the query of the previous example by the agent, database-agent1 to the
1196 agent, facilitator1:
1197
1198 (inform
1199   :sender
1200     (agent-identifier
1201       :name database-agent1)
1202   :receiver
1203     (agent-identifier
1204       :name facilitator1)
1205   :language FIPA-KIF
1206   :ontology ec-ontology
1207   :content
1208     (= (kappa (?make ?door ?price)
1209       (and (car ?car) (make ?car ?make)
1210         (doors ?car ?doors) (price ?car ?price)))
1211       '((Mercedes 4 100,000) (Honda 2 20,000) (Toyota 4 25,000))))

```