

# FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS

## FIPA SL Content Language Specification

<b>Document title</b>	FIPA SL Content Language Specification		
<b>Document number</b>	XC00008G	<b>Document source</b>	FIPA TC C
<b>Document status</b>	Experimental	<b>Date of this status</b>	2001/08/10
<b>Supersedes</b>	FIPA00003		
<b>Contact</b>	fab@fipa.org		
<b>Change history</b>			
2000/09/28	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	Grammar FIPA SL Concrete Syntax .....	2
41	2.1	Lexical Definitions .....	3
42	3	Notes on FIPA SL Semantics .....	5
43	3.1	Grammar Entry Point: FIPA SL Content Expression .....	5
44	3.2	Well-Formed Formulas .....	5
45	3.3	Atomic Formula .....	6
46	3.4	Terms .....	7
47	3.5	Referential Operators .....	7
48	3.5.1	iota .....	7
49	3.5.2	Any .....	9
50	3.5.3	All .....	10
51	3.6	Functional Terms .....	11
52	3.7	Result Predicate .....	12
53	3.8	Actions and Action Expressions .....	12
54	3.9	Agent Identifiers .....	13
55	3.10	Numerical Constants .....	13
56	3.11	Date and Time Constants .....	13
57	4	Reduced Expressivity Subsets of FIPA SL .....	14
58	4.1	FIPA SL0: Minimal Subset .....	14
59	4.2	FIPA SL1: Propositional Form .....	15
60	4.3	FIPA SL2: Decidability Restrictions .....	16
61	5	References .....	19
62	6	Annex A — Syntax and Lexical Notation .....	20
63			

## 63 **1 Scope**

64 This specification defines a concrete syntax for the FIPA Semantic Language (SL) content language. This syntax and  
65 its associated semantics are suggested as a candidate content language for use in conjunction with the FIPA Agent  
66 Communication Language (see [FIPA00037]). In particular, the syntax is defined to be a sub-grammar of the very  
67 general s-expression syntax specified for message content given in [FIPA00037].  
68

69 This content language is included in the specification on an informative basis. It is not mandatory for any FIPA  
70 implementation to implement the computational mechanisms necessary to process all of the constructs in this  
71 language. However, FIPA SL is a general purpose representation formalism that may be suitable for use in a number of  
72 different agent domains.

73

## 73 2 Grammar FIPA SL Concrete Syntax

74 See Section 6, Annex A — *Syntax and Lexical Notation* for an explanation of the used syntactic notation.

75		
76	Content	= "(" ContentExpression+ ")".
77		
78	ContentExpression	= IdentifyingExpression
79		ActionExpression
80		Proposition.
81		
82	Proposition	= Wff.
83		
84	Wff	= AtomicFormula
85		"(" UnaryLogicalOp Wff ")"
86		"(" BinaryLogicalOp Wff Wff ")"
87		"(" Quantifier Variable Wff ")"
88		"(" ModalOp Agent Wff ")"
89		"(" ActionOp ActionExpression ")"
90		"(" ActionOp ActionExpression Wff ")".
91		
92	UnaryLogicalOp	= "not".
93		
94	BinaryLogicalOp	= "and"
95		"or"
96		"implies"
97		"equiv".
98		
99	AtomicFormula	= PropositionSymbol
100		"(" BinaryTermOp Term Term ")"
101		"(" PredicateSymbol Term+ ")"
102		"true"
103		"false".
104		
105	BinaryTermOp	= "="
106		"\="
107		">"
108		">="
109		"<"
110		"<="
111		"member"
112		"contains"
113		"result".
114		
115	Quantifier	= "forall"
116		"exists".
117		
118	ModalOp	= "B"
119		"U"
120		"PG"
121		"I".
122		
123	ActionOp	= "feasible"
124		"done".
125		
126	Term	= Variable
127		FunctionalTerm
128		ActionExpression
129		IdentifyingExpression
130		Constant
131		Sequence
132		Set.
133		

```

134 IdentifyingExpression = "(" ReferentialOperator Term Wff ")".
135
136 ReferentialOperator   = "iota"
137                       | "any"
138                       | "all".
139
140 FunctionalTerm        = "(" "cons"          Term Term ")"
141                       | "(" "first"         Term ")"
142                       | "(" "rest"          Term ")"
143                       | "(" "nth"           Term Term ")"
144                       | "(" "append"        Term Term ")"
145                       | "(" "union"         Term Term ")"
146                       | "(" "intersection"  Term Term ")"
147                       | "(" "difference"    Term Term ")"
148                       | "(" ArithmeticOp   Term Term ")"
149                       | "(" FunctionSymbol  Term* ")"
150                       | "(" FunctionSymbol  Parameter* ")".
151
152 Constant              = NumericalConstant
153                       | String
154                       | DateTime.
155
156 NumericalConstant     = Integer
157                       | Float.
158
159 Variable              = VariableIdentifier.
160
161 ActionExpression      = "(" "action" Agent Term ")"
162                       | "(" "|" ActionExpression ActionExpression ")"
163                       | "(" ";" ActionExpression ActionExpression ")".
164
165 PropositionSymbol     = String.
166
167 PredicateSymbol       = String.
168
169 FunctionSymbol        = String.
170
171 Agent                 = Term.
172
173 Sequence              = "(" "sequence" Term* ")".
174
175 Set                   = "(" "set" Term* ")".
176
177 Parameter             = ParameterName ParameterValue.
178
179 ParameterValue        = Term.
180
181 ArithmeticOp          = "+"
182                       | "-"
183                       | "*"
184                       | "/"
185                       | "%".
186

```

## 187 2.1 Lexical Definitions

188 All white space, tabs, carriage returns and line feeds between tokens should be skipped by the lexical analyser. See  
 189 *Section 6, Annex A — Syntax and Lexical Notation* for an explanation of the used notation.

```

190
191 String                = Word
192                       | StringLiteral.
193
194 Word                  = [~ "\0x00" - "\0x20", "(", ")", "#", "0" - "9", ":", "-", "?"]
195                       [~ "\0x00" - "\0x20", "(", ")", ""]*.

```

```

196
197 ParameterName      = ":" String.
198
199 VariableIdentifier  = "?" String.
200
201 Sign                = [ "+" , "-" ].
202
203 Integer             = Sign? DecimalLiteral+
204                    | Sign? "0" ["x", "X"] HexLiteral+.
205
206 Dot                 = "."
207
208 Float               = Sign? FloatMantissa FloatExponent?
209                    | Sign? DecimalLiteral+ FloatExponent.
210
211 FloatMantissa       = DecimalLiteral+ Dot DecimalLiteral*
212                    | DecimalLiteral* Dot DecimalLiteral+.
213
214 FloatExponent       = Exponent Sign? DecimalLiteral+.
215
216 Exponent            = ["e", "E"].
217
218 DecimalLiteral      = ["0" - "9"].
219
220 HexLiteral          = ["0" - "9", "A" - "F", "a" - "f"].
221
222 StringLiteral       = "\""( [~ "\""]
223                    | "\\\"")*\"".
224
225 DateTime            = Year Month Day "T" Hour Minute
226                    Second MilliSecond TypeDesignator?.
227
228 Year                = DecimalLiteral DecimalLiteral DecimalLiteral DecimalLiteral.
229
230 Month               = DecimalLiteral DecimalLiteral.
231
232 Day                 = DecimalLiteral DecimalLiteral.
233
234 Hour                = DecimalLiteral DecimalLiteral.
235
236 Minute              = DecimalLiteral DecimalLiteral.
237
238 Second              = DecimalLiteral DecimalLiteral.
239
240 MilliSecond         = DecimalLiteral DecimalLiteral DecimalLiteral.
241
242 TypeDesignator      = ["a" - "z" , "A" - "Z"].
243
244

```

## 244 3 Notes on FIPA SL Semantics

245 This section contains explanatory notes on the intended semantics of the constructs introduced in above.  
246

### 247 3.1 Grammar Entry Point: FIPA SL Content Expression

248 An FIPA SL content expression may be used as the content of an ACL message. There are three cases:  
249

250 A proposition, which may be assigned a truth value in a given context. Precisely, it is a well-formed formula (Wff)  
251 using the rules described in the `wff` production. A proposition is used in the `inform` communicative act (CA) and  
252 other CAs derived from it.  
253

254 An action, which can be performed. An action may be a single action or a composite action built using the  
255 sequencing and alternative operators. An action is used as a content expression when the act is `request` and  
256 other CAs derived from it.  
257

258 An identifying reference expression (IRE), which identifies an object in the domain. This is the Referential operator  
259 and is used in the `inform-ref` macro act and other CAs derived from it.  
260

261 Other valid content expressions may result from the composition of the above basic cases. For instance, an action-  
262 condition pair (represented by an `ActionExpression` followed by a `wff`) is used in the `propose` act; an action-  
263 condition-reason triplet (represented by an `ActionExpression` followed by two `wffs`) is used in the `reject-`  
264 `proposal` act. These are used as arguments to some ACL CAs in [FIPA00037].  
265

### 266 3.2 Well-Formed Formulas

267 A well-formed formula is constructed from an atomic formula, whose meaning will be determined by the semantics of  
268 the underlying domain representation or recursively by applying one of the construction operators or logical connectives  
269 described in the `wff` grammar rule. These are:  
270

271 (not <Wff>)

272 Negation. The truth value of this expression is false if `wff` is true. Otherwise it is true.  
273

274 (and <Wff0> <Wff1>)

275 Conjunction. This expression is true iff<sup>1</sup> well-formed formulae `wff0` and `wff1` are both true, otherwise it is false.  
276

277 (or <Wff0> <Wff1>)

278 Disjunction. This expression is false iff well-formed formulae `wff0` and `wff1` are both false, otherwise it is true.  
279

280 (implies <Wff0> <Wff1>)

281 Implication. This expression is true if either `wff0` is false or alternatively if `wff0` is true and `wff1` is true. Otherwise  
282 it is false. The expression corresponds to the standard material implication connective `wff0` `wff1`.  
283

284 (equiv <Wff0> <Wff1>)

285 Equivalence. This expression is true if either `wff0` is true and `wff1` is true, or alternatively if `wff0` is false and  
286 `wff1` is false. Otherwise it is false.  
287

288 (forall <variable> <Wff>)

289 Universal quantification. The quantified expression is true if `wff` is true for every value of value of the quantified  
290 variable.  
291

292 (exists <variable> <Wff>)

---

<sup>1</sup> If and only if.



293 Existential quantification. The quantified expression is true if there is at least one value for the variable for which  
294 wff is true.

295  
296 (B <agent> <expression>)  
297 Belief. It is true that agent believes that expression is true.

298  
299 (U <agent> <expression>)  
300 Uncertainty. It is true that agent is uncertain of the truth of expression. Agent neither believes expression  
301 nor its negation, but believes that expression is more likely to be true than its negation.

302  
303 (I <agent> <expression>)  
304 Intention. It is true that agent intends that expression becomes true and will plan to bring it about.

305  
306 (PG <agent> <expression>)  
307 Persistent goal. It is true that agent holds a persistent goal that expression becomes true, but will not  
308 necessarily plan to bring it about.

309  
310 (feasible <ActionExpression> <Wff>)  
311 It is true that ActionExpression (or, equivalently, some event) can take place and just afterwards wff will be  
312 true.

313  
314 (feasible <ActionExpression>)  
315 Same as (feasible <ActionExpression> true).

316  
317 (done <ActionExpression> <Wff>)  
318 It is true that ActionExpression (or, equivalently, some event) has just taken place and just before that wff was  
319 true.

320  
321 (done <ActionExpression>)  
322 Same as (done <ActionExpression> true).

323

### 324 3.3 Atomic Formula

325 The atomic formula represents an expression which has a truth value in the language of the domain of discourse.  
326 Three forms are defined:

- 327
- 328 a given propositional symbol may be defined in the domain language, which is either true or false,
  - 329
  - 330 two terms may or may not be equal under the semantics of the domain language, or,
  - 331
  - 332 some predicate is defined over a set of zero or more arguments, each of which is a term.
  - 333

334 The FIPA SL representation does not define a meaning for the symbols in atomic formulae: this is the responsibility of  
335 the domain language representation and ontology. Several forms are defined:

336  
337 true false  
338 These symbols represent the true proposition and the false proposition.

339  
340 (= Term1 Term2)  
341 Term1 and Term2 denote the same object under the semantics of the domain.

342  
343 (\= Term1 Term2)  
344 Term1 and Term2 do not denote the same object under the semantics of the domain.

345  
346 (> Constant1 Constant2)

The `>` operator relies on an order relation defined to be the usual numeric ordering for numerical constants and the usual alphabetical ordering for literal constants. Under this order relation, `Constant1` denotes an object that comes after the object denoted by `Constant2`, under the semantics of the domain.

```
(>= Constant1 Constant2)
```

The `>=` operator relies on an order relation defined to be the usual numeric ordering for numerical constants and the usual alphabetical ordering for literal constants. Under this order relation, `Constant1` denotes an object that comes after or is the same object as the object denoted by `Constant2`, under the semantics of the domain.

```
(< Constant1 Constant2)
```

The `<` operator relies on an order relation defined to be the usual numeric ordering for numerical constants and the usual alphabetical ordering for literal constants. Under this order relation, `Constant1` denotes an object that comes before the object denoted by `Constant2`, under the semantics of the domain.

```
(=< Constant1 Constant2)
```

The `=<` operator relies on an order relation defined to be the usual numeric ordering for numerical constants and the usual alphabetical ordering for literal constants. Under this order relation, `Constant1` denotes an object that comes before or is the same object as the object denoted by `Constant2`, under the semantics of the domain.

```
(member Term Collection)
```

The object denoted by `Term`, under the semantics of the domain, is a member of the collection (either a set or a sequence) denoted by `Collection` under the semantics of the domain.

```
(contains Collection1 Collection2)
```

If `Collection1` and `Collection2` denote sets, this proposition means the set denoted by `Collection1` contains the set denoted by `Collection2`. If the arguments are sequences, then the proposition means that all of the elements of the sequence denoted by `Collection2` appear in the same order in the sequence denoted by `Collection1`.

Other predicates may be defined over a set of arguments, each of which is a term, by using the `(PredicateSymbol Term+)` production.

The FIPA SL representation does not define a meaning for other symbols in atomic formulae: this is the responsibility of the domain language representation and the relative ontology.

### 3.4 Terms

Terms are either themselves atomic (constants and variables) or recursively constructed as a functional term in which a functor is applied to zero or more arguments. Again, FIPA SL only mandates a syntactic form for these terms. With small number of exceptions (see below), the meanings of the symbols used to define the terms are determined by the underlying domain representation.

Note that, as mentioned above, no legal well-formed expression contains a free variable, that is, a variable not declared in any scope within the expression. Scope introducing formulae are the quantifiers (`forall`, `exists`) and the reference operators `iota`, `any` and `all`. Variables may only denote terms, not well-formed formulae.

### 3.5 Referential Operators

#### 3.5.1 Iota

```
(iota <term> <formula>)
```

The `iota` operator introduces a scope for the given expression (which denotes a term), in which the given identifier, which would otherwise be free, is defined. An expression containing a free variable is not a well-formed

FIPA SL expression. The expression  $(\text{iota } x (P x))$  may be read as "the  $x$  such that  $P$  [is true] of  $x$ ". The *iota* operator is a constructor for terms which denote objects in the domain of discourse.

### Formal Definition

A *iota* expression can only be evaluated with respect to a given theory. Suppose  $KB$  is a knowledge base such that  $T(KB)$  is the theory generated from  $KB$  by a given reasoning mechanism. Formally,  $(\text{iota } x (P x))$  iff  $x$  is a term that belongs to the set  $\{t : T(KB)\}$  and  $\{x\}$  is a singleton; or  $(\text{iota } x (P x))$  is undefined if  $\{x\}$  is not a singleton. In this definition  $\sigma$  is a most general variable substitution,  $\sigma(P x)$  is the result of applying  $\sigma$  to  $P x$ , and  $\sigma(x)$  is the result of applying  $\sigma$  to  $x$ . This implies that a failure occurs if no object or more than one object satisfies the condition specified in the *iota* operator.

### Example 1

This example depicts an interaction between agent  $A$  and  $B$  that makes use of the *iota* operator, where agent  $A$  is supposed to have the following knowledge base  $KB=\{P(A), Q(1, A), Q(1, B)\}$ .

```
(query-ref
  :sender (agent-identifier :name B)
  :receiver (set (agent-identifier :name A))
  :content
    ((iota ?x (p ?x)))
  :language FIPA-SL
  :reply-with query1)

(inform
  :sender (agent-identifier :name A)
  :receiver (set (agent-identifier :name B))
  :content
    ((= (iota ?x (p ?x)) a))
  :language FIPA-SL
  :in-reply-to query1)
```

The only object that satisfies proposition  $P(x)$  is  $a$ , therefore, the *query-ref* message is replied by the *inform* message as shown.

### Example 2

This example shows another successful interaction but more complex than the previous one.

```
(query-ref
  :sender (agent-identifier :name B)
  :receiver (set (agent-identifier :name A))
  :content
    ((iota ?x (q ?x ?y)))
  :language FIPA-SL
  :reply-with query2)

(inform
  :sender (agent-identifier :name A)
  :receiver (set (agent-identifier :name B))
  :content
    ((= (iota ?x (q ?x ?y)) 1))
  :language FIPA-SL
  :in-reply-to query2)
```

The most general substitutions  $\sigma$  such that  $Q(x, y)$  can be derived from  $KB$  are  $\sigma_1 \{x/1, y/A\}$  and  $\sigma_2 \{x/1, y/B\}$ . Therefore, the set  $\{t : T(KB)\} \{\{x/1, y/A\}x, \{x/1, y/B\}x\} \{1\}$  is a singleton and hence  $(\text{iota } x (q ?x ?y))$  represents the object 1.

### Example 3

Finally, this example shows an unsuccessful interaction using the *iota* operator. In this case, agent A cannot evaluate the *iota* expression and therefore a failure message is returned to agent B

```

457 (query-ref
458   :sender (agent-identifier :name B)
459   :receiver (set (agent-identifier :name A))
460   :content
461     ((iota ?y (q ?x ?y)))
462   :language FIPA-SL
463   :reply-with query3)
464
465 (failure
466   :sender (agent-identifier :name A)
467   :receiver (set (agent-identifier :name B))
468   :content
469     ((action (agent-identifier :name A)
470              (inform-ref
471                :sender (agent-identifier :name A)
472                :receiver (set (agent-identifier :name B))
473                :content
474                  "((iota ?y (q ?x ?y)))"
475                :language FIPA-SL
476                :in-reply-to query3))
477              more-than-one-answer)
478   :language FIPA-SL
479   :in-reply-to query3)

```

The most general substitutions that satisfy  $Q(x, y)$  are  $\{x/1, y/a\}$  and  $\{x/1, y/b\}$ , therefore, the set  $\{ : T(KB) \} \{ \{x/1, y/A\}y, \{x/1, y/B\}y \} \{A, B\}$ , which is not a singleton. This means that the *iota* expression used in this interaction is not defined.

### 3.5.2 Any

```
(any <term> <formula>)
```

The *any* operator is used to denote any object that satisfies the proposition represented by *formula*.

#### Formal Definition

An *any* expression can only be evaluated with respect to a given theory. Suppose *KB* is a knowledge base such that  $T(KB)$  is the theory generated from *KB* by a given reasoning mechanism. Formally,  $\text{any}(t, \phi)$  iff  $t$  is a term that belongs to the set  $\{ : T(KB) \}$ ; or  $\text{any}(t, \phi)$  is undefined if  $\phi$  is the empty set. In this definition  $t$  is a most general variable substitution,  $\phi$  is the result of applying  $t$  to  $\phi$ , and  $\phi'$  is the result of applying  $t$  to  $\phi'$ .

This definition implies that failures only occur if there are no objects satisfying the condition specified as the second argument of the *any* operator.

#### Example 4

Assuming that agent A has the following knowledge base  $KB = \{P(A), Q(1, A), Q(1, B)\}$ , this example shows a successful interaction with agent A using the *any* operator.

```

503 (query-ref
504   :sender (agent-identifier :name B)
505   :receiver (set (agent-identifier :name A))
506   :content
507     ((any (sequence ?x ?y) (q ?x ?y)))
508   :language FIPA-SL
509   :reply-with query1)
510
511 (inform
512   :sender (agent-identifier :name A)

```

```

513     :receiver (set (agent-identifier :name B))
514     :content
515       ((= (any (sequence ?x ?y) (q ?x ?y)) (sequence 1 a)))
516     :language FIPA-SL
517     :in-reply-to query1)
518

```

The most general substitutions such that  $Q(x, y)$  can be derived from KB are  $\{x/1, y/A\}$  and  $\{x/1, y/B\}$ , therefore  $\{ \text{Sequence}(x, y): Q(x, y) \in T(\text{KB}) \} = \{ \text{Sequence}(1, A), \text{Sequence}(1, B) \}$ . Using this set, agent A chooses the first element of as the appropriate answer to agent B.

### Example 5

This example shows an unsuccessful interaction with agent A, using the `any` operator.

```

526 (query-ref
527   :sender (agent-identifier :name B)
528   :receiver (set (agent-identifier :name A))
529   :content
530     ((any ?x (r ?x)))
531   :language FIPA-SL
532   :reply-with query2)
533
534 (failure
535   :sender (agent-identifier :name A)
536   :receiver (set (agent-identifier :name B))
537   :content
538     ((action (agent-identifier :name A)
539              (inform-ref
540                :sender (agent-identifier :name A)
541                :receiver (set (agent-identifier :name B))
542                :content
543                  "((any ?x (r ?x)))"
544                :language FIPA-SL
545                :in-reply-to query2))
546              (unknown-predicate r))
547   :language FIPA-SL
548   :in-reply-to query2)
549

```

Since agent A does not know the  $r$  predicate, the answer to the query that had been sent by agent B cannot be determined, therefore a failure message is sent to agent B from agent A. The failure message specifies the failure's reason (i.e., `unknown-predicate r`)

### 3.5.3 All

```
(all <term> <formula>)
```

The `all` operator is used to denote the set of all objects that satisfy the proposition represented by `formula`.

#### Formal Definition

An `all` expression can only be evaluated with respect to a given theory. Suppose KB is a knowledge base such that  $T(\text{KB})$  is the theory generated from KB by a given reasoning mechanism. Formally,  $\text{all}(x, y) \{ : T(\text{KB}) \}$ . Notice that  $\text{all}(x, y)$  may be a singleton or even an empty set. In this definition  $x$  is a most general variable substitution,  $y$  is the result of applying  $x$  to  $y$ , and  $z$  is the result of applying  $x$  to  $z$ .

If no objects satisfy the condition specified as the second argument of the `all` operator, then the identifying expression denotes an empty set.

#### Example 6

Suppose agent A has the following knowledge base  $\text{KB} = \{P(A), Q(1, A), Q(1, B)\}$ . This example shows a successful interaction between agent A and B that make use of the `all` operator.

```

570
571 (query-ref
572   :sender (agent-identifier :name B)
573   :receiver (set (agent-identifier :name A))
574   :content
575     ((all (sequence ?x ?y) (q ?x ?y)))
576   :language FIPA-SL
577   :reply-with query1)
578
579 (inform
580   :sender (agent-identifier :name A)
581   :receiver (set (agent-identifier :name B))
582   :content
583     ((= (all (sequence ?x ?y) (q ?x ?y)) (set(sequence 1 a)(sequence 1 b))))
584   :language FIPA-SL
585   :in-reply-to query1)
586

```

The set of the most general substitutions such that  $Q(x, y)$  can be derived from KB is  $\{\{x/1, y/A\}, \{x/1, y/B\}\}$ , therefore  $\text{all}(\text{Sequence}(x, y), Q(x, y)) \{\text{Sequence}(1, A), \text{Sequence}(1, B)\}$ .

### Example 7

Following Example 6, if there is no possible answer to a query making use of the `all` operator, then the agent should return the empty set.

```

593
594 (query-ref
595   :sender (agent-identifier :name B)
596   :receiver (set (agent-identifier :name A))
597   :content
598     ((all ?x (q ?x c)))
599   :language FIPA-SL
600   :reply-with query2)
601
602 (inform
603   :sender (agent-identifier :name A)
604   :receiver (set (agent-identifier :name B))
605   :content
606     ((= (all ?x (q ?x c))(set)))
607   :language FIPA-SL
608   :in-reply-to query2)
609

```

Since there is no possible substitution for  $x$  such that  $Q(x, C)$  can be derived from KB, then  $\text{all}(x, Q(x, c))=\{\}$ . In this interaction the term `(set)` represents the empty set.

## 3.6 Functional Terms

A functional term refers to an object via a functional relation (referred by the `FunctionSymbol`) with other objects (that is, the terms or parameters), rather than using the direct name of that object, for example, `(fatherOf Jesus)` rather than `God`.

Two syntactical forms can be used to express a functional term. In the first form the functional symbol is followed by a list of terms that are the arguments of the function symbol. The semantics of the arguments is position-dependent, for example, `(divide 10 2)` where 10 is the dividend and 2 is the divisor. In the second form each argument is preceded by its name, for example, `(divide :dividend 10 :divisor 2)`. This second form is particularly appropriate to represent descriptions where the function symbol should be interpreted as the constructor of an object, while the parameters represent the attributes of the object.

The following is an example of an object, instance of a vehicle class:

```

626 (vehicle
627

```

```

628   :colour red
629   :max-speed 100
630   :owner (Person
631     :name Luis
632     :nationality Portuguese))
633

```

634 Some ontologies may decide to give a description of some concepts only in one or both of these two forms, that is by  
635 specifying, or not, a default order to the arguments of each function in the domain of discourse. How this order is  
636 specified is outside the scope of this specification.

637  
638 Functional terms can be constructed by a domain functor applied to zero or more terms. Besides domain functions,  
639 FIPA SL includes functional terms constructed from widely used functional operators and their arguments described in  
640 *Table 1*.  
641

Operator	Example	Description
+ - / % *	5 % 2	Usual arithmetic operations.
Union	(union ?s1 ?s2)	Represents the union of two sets.
Intersection	(intersection ?s1 ?s2)	Represents the intersection of two sets.
Difference	(difference ?s1 ?s2)	Represents the set difference between ?s1 and ?s2.
First	(first ?seq)	Represents the first element of a sequence.
Rest	(rest ?seq)	Represents sequence ?seq except its first element.
Nth	(nth 3 ?seq)	Represents the nth element of a sequence.
Cons	(cons a (sequence b c))	If its second argument is a sequence, it represents the sequence that results of inserting its first argument in front of its second argument. If its second argument is a set, it represents the set that has all elements contained in its second argument plus its first argument.
Append	(append ?seq (sequence c d))	Represents the sequence that results of concatenating its first argument with its second argument.

642  
643  
644

**Table 1:** Functional Operators

### 645 3.7 Result Predicate

646 A common need is to determine the result of performing an action or evaluating a term. To facilitate this operation, a  
647 standard predicate `result`, of arity two, is introduced to the language. `Result/2` has the declarative meaning that the  
648 result of evaluating a term, or equivalently of performing an action, encoded by the first argument term, is the second  
649 argument term. However, it is expected that this declarative semantics will be implemented in a more efficient,  
650 operational way in any given FIPA SL interpreter.

651  
652 A typical use of the `result` predicate is with a variable scoped by `iota`, giving an expression whose meaning is, for  
653 example, "the `x` which is the result of agent `i` performing `act`":

```

654 (iota x (result (action i act) x))
655
656

```

### 657 3.8 Actions and Action Expressions

658 Action expressions are a special subset of terms. An action itself is introduced by the keyword `action` and comprises  
659 the agent of the action (that is, an identifier representing the agent performing the action) and a term denoting the action  
660 which is [to be] performed.

661

662 Notice that a specific type of action is an ACL communicative act (CA). When expressed in FIPA SL, syntactically an  
663 ACL communicative act is an action where the term denotes the CA including all its parameters, as referred by the used  
664 ontology. Example 5 includes an example of an ACL CA, encoded as a `String`, whose content embeds another CA.

665  
666 Two operators are used to build terms denoting composite CAs:

667  
668 the sequencing operator (`:`) denotes a composite act in which the first action (represented by the first operand) is  
669 followed by the second action, and,

670  
671 the alternative operator (`|`) denotes a composite act in which either the first action occurs, or the second, but not  
672 both.

673

### 674 **3.9 Agent Identifiers**

675 An agent is represented by referring to its name. The name is defined using the standard format from [FIPA00023].  
676

### 677 **3.10 Numerical Constants**

678 The standard definitions for integers and floating point numbers are assumed. However, due to the necessarily  
679 unpredictable nature of cross-platform dependencies, agents should not make strong assumptions about the precision  
680 with which another agent is able to represent a given numerical value. FIPA SL assumes only 32-bit representations of  
681 both integers and floating point numbers. Agents should not exchange message contents containing numerical values  
682 requiring more than 32 bits to encode precisely, unless some prior arrangement is made to ensure that this is valid.  
683

### 684 **3.11 Date and Time Constants**

685 Time tokens are based on [ISO8601], with extension for millisecond durations. If no type designator is given, the local  
686 time zone is then used. The type designator for UTC is the character `z`; UTC is preferred to prevent time zone  
687 ambiguities. Note that years must be encoded in four digits. As an example, 8:30 am on 15th April, 1996 local time  
688 would be encoded as:

689  
690 `19960415T0830000000`

691  
692 The same time in UTC would be:

693  
694 `19960415T083000000Z`

695

696



## 696 4 Reduced Expressivity Subsets of FIPA SL

697 The FIPA SL definition given above is a very expressive language, but for some agent communication tasks it is  
 698 unnecessarily powerful. This expressive power has an implementation cost to the agent and introduces problems of the  
 699 decidability of modal logic. To allow simpler agents, or agents performing simple tasks, to do so with minimal  
 700 computational burden, this section introduces semantic and syntactic subsets of the full FIPA SL content language for  
 701 use by the agent when it is appropriate or desirable to do so. These subsets are defined by the use of profiles, that is,  
 702 statements of restriction over the full expressive power of FIPA SL. These profiles are defined in increasing order of  
 703 expressivity as FIPA-SL0, FIPA-SL1 and FIPA-SL2.

704  
 705 Note that these subsets of FIPA SL, with additional ontological commitments (that is, the definition of domain predicates  
 706 and constants) are used in other FIPA specifications.  
 707

### 708 4.1 FIPA SL0: Minimal Subset

709 Profile 0 is denoted by the normative constant FIPA-SL0 in the :language parameter of an ACL message. Profile 0  
 710 of FIPA SL is the minimal subset of the FIPA SL content language. It allows the representation of actions, the  
 711 determination of the result a term representing a computation, the completion of an action and simple binary  
 712 propositions. The following defines the FIPA SL0 grammar:

```

713 Content           = "(" ContentExpression+ ")".
714
715 ContentExpression = ActionExpression
716                  | Proposition.
717
718 Proposition       = Wff.
719
720 Wff               = AtomicFormula
721                  | "(" ActionOp ActionExpression ")".
722
723 AtomicFormula    = PropositionSymbol
724                  | "(" "result"      Term Term ")"
725                  | "(" PredicateSymbol Term+ ")"
726                  | "true"
727                  | "false".
728
729 ActionOp         = "done".
730
731 Term             = Constant
732                  | Set
733                  | Sequence
734                  | FunctionalTerm
735                  | ActionExpression.
736
737 ActionExpression = "(" "action" Agent Term ")".
738
739 FunctionalTerm   = "(" FunctionSymbol Term* ")"
740                  | "(" FunctionSymbol Parameter* ")".
741
742 Parameter       = ParameterName ParameterValue.
743
744 ParameterValue  = Term.
745
746 Agent           = Term.
747
748 FunctionSymbol  = String.
749
750 PropositionSymbol = String.
751
752 PredicateSymbol = String.
753
```

```

754
755 Constant          = NumericalConstant
756                   | String
757                   | DateTime.
758
759 Set                = "(" "set" Term* ")".
760
761 Sequence           = "(" "sequence" Term* ")".
762
763 NumericalConstant = Integer
764                   | Float.
765

```

The same lexical definitions described in *Section 2.1, Lexical Definitions* apply for FIPA SL0.

## 768 4.2 FIPA SL1: Propositional Form

769 Profile 1 is denoted by the normative constant FIPA-SL1 in the :language parameter of an ACL message. Profile 1  
770 of FIPA SL extends the minimal representational form of FIPA SL0 by adding Boolean connectives to represent  
771 propositional expressions. The following defines the FIPA SL1 grammar:

```

772
773 Content            = "(" ContentExpression+ ")".
774
775 ContentExpression = ActionExpression
776                   | Proposition.
777
778 Proposition        = Wff.
779
780 Wff                = AtomicFormula
781                   | "(" UnaryLogicalOp Wff ")"
782                   | "(" BinaryLogicalOp Wff Wff ")"
783                   | "(" ActionOp ActionExpression ")".
784
785 UnaryLogicalOp    = "not".
786
787 BinaryLogicalOp   = "and"
788                   | "or".
789
790 AtomicFormula     = PropositionSymbol
791                   | "(" "result" Term Term ")"
792                   | "(" PredicateSymbol Term+ ")"
793                   | "true"
794                   | "false".
795
796 ActionOp          = "done".
797
798 Term              = Constant
799                   | Set
800                   | Sequence
801                   | FunctionalTerm
802                   | ActionExpression.
803
804 ActionExpression  = "(" "action" Agent Term ")".
805
806 FunctionalTerm    = "(" FunctionSymbol Term* ")"
807                   | "(" FunctionSymbol Parameter* ")".
808
809 Parameter         = ParameterName ParameterValue.
810
811 ParameterValue    = Term.
812
813 Agent             = Term.
814

```

```

815 FunctionSymbol      = String.
816
817 PropositionSymbol   = String.
818
819 PredicateSymbol     = String.
820
821 Constant            = NumericalConstant
822                    | String
823                    | DateTime.
824
825 Set                 = "(" "set" Term* ")".
826
827 Sequence            = "(" "sequence" Term* ")".
828
829 NumericalConstant  = Integer
830                    | Float.

```

831  
832 The same lexical definitions described in *Section 2.1, Lexical Definitions* apply for FIPA SL1.  
833

### 834 4.3 FIPA SL2: Decidability Restrictions

835 Profile 2 is denoted by the normative constant FIPA-SL2 in the :language parameter of an ACL message. Profile 2  
836 of FIPA SL allows first order predicate and modal logic, but is restricted to ensure that it must be decidable. Well-known  
837 effective algorithms exist that can derive whether or not an FIPA SL2 Wff is a logical consequence of a set of Wffs (for  
838 instance KSAT and Monadic). The following defines the FIPA SL2 grammar:

```

839
840 Content              = "(" ContentExpression+ ")".
841
842 ContentExpression    = IdentifyingExpression
843                    | ActionExpression
844                    | Proposition.
845
846 Proposition          = PrenexExpression.
847
848 Wff                  = AtomicFormula
849                    | "(" UnaryLogicalOp Wff ")"
850                    | "(" BinaryLogicalOp Wff Wff ")"
851                    | "(" ModalOp Agent PrenexExpression ")"
852                    | "(" ActionOp ActionExpression ")"
853                    | "(" ActionOp ActionExpression PrenexExpression ")".
854
855 UnaryLogicalOp      = "not".
856
857 BinaryLogicalOp     = "and"
858                    | "or"
859                    | "implies"
860                    | "equiv".
861
862 AtomicFormula        = PropositionSymbol
863                    | "(" "=" Term Term ")"
864                    | "(" "result" Term Term ")"
865                    | "(" PredicateSymbol Term+ ")"
866                    | "true"
867                    | "false".
868
869 PrenexExpression     = UnivQuantExpression
870                    | ExistQuantExpression
871                    | Wff.
872
873 UnivQuantExpression = "(" "forall" Variable Wff ")"
874                    | "(" "forall" Variable UnivQuantExpression ")"
875                    | "(" "forall" Variable ExistQuantExpression ")".

```

```

876
877 ExistQuantExpression = "(" "exists" Variable Wff ")"
878                       | "(" "exists" Variable ExistQuantExpression ")".
879
880 Term                   = Variable
881                       | FunctionalTerm
882                       | ActionExpression
883                       | IdentifyingExpression
884                       | Constant
885                       | Sequence
886                       | Set.
887
888 IdentifyingExpression = "(" ReferentialOp Term Wff ")".
889
890 ReferentialOp          = "iota"
891                       | "any"
892                       | "all".
893
894 FunctionalTerm         = "(" FunctionSymbol Term* ")"
895                       | "(" FunctionSymbol Parameter* ")".
896
897 Parameter              = ParameterName ParameterValue.
898
899 ParameterValue         = Term.
900
901 ActionExpression       = "(" "action" Agent Term ")"
902                       | "(" "|" ActionExpression ActionExpression ")"
903                       | "(" ";" ActionExpression ActionExpression ")".
904
905 Variable               = VariableIdentifier.
906
907 Agent                  = Term.
908
909 FunctionSymbol         = String.
910
911 Constant               = NumericalConstant
912                       | String
913                       | DateTime.
914
915 ModalOp                = "B"
916                       | "U"
917                       | "PG"
918                       | "I".
919
920 ActionOp               = "feasible"
921                       | "done".
922
923 PropositionSymbol      = String.
924
925 PredicateSymbol        = String.
926
927 Set                    = "(" "set" Term* ")".
928
929 Sequence                = "(" "sequence" Term* ")".
930
931 NumericalConstant      = Integer
932                       | Float.
933
934

```

935 The same lexical definitions described in *Section 2.1, Lexical Definitions* apply for FIPA SL2.

936

937 The *Wff* production of FIPA SL2 no longer directly contains the logical quantifiers, but these are treated separately to  
 938 ensure only prefixed quantified formulas, such as:

```
939
940 (forall ?x1
941   (forall ?x2
942     (exists ?y1
943       (exists ?y2
944         (Phi ?x1 ?x2 ?y1 ?y2))))))
```

945  
946 Where (Phi ?x1 ?x2 ?y1 ?y2) does not contain any quantifier.

947  
948 The grammar of FIPA SL2 still allows for quantifying-in inside modal operators. For example, the following formula is  
949 still admissible under the grammar:

```
950
951 (forall ?x1
952   (or
953     (B i (p ?x1))
954     (B j (q ?x1))))
```

955  
956 It is not clear that formulae of this kind are decidable. However, changing the grammar to express this context  
957 sensitivity would make the EBNF form above essentially unreadable. Thus, the following additional mandatory  
958 constraint is placed on well-formed content expressions using FIPA SL2:

959  
960 Within the scope of an `S�ModalOperator` only closed formulas are allowed, that is, formulas without free variables.

961

962

962  
963  
964  
965  
966  
967  
968  
969  
970  
971

## 5 References

- [FIPA00023] FIPA Agent Management Specification. Foundation for Intelligent Physical Agents, 2000.  
<http://www.fipa.org/specs/fipa00023/>
- [FIPA00037] FIPA Agent Communication Language Overview. Foundation for Intelligent Physical Agents, 2000.  
<http://www.fipa.org/specs/fipa00037/>
- [ISO8601] Date Elements and Interchange Formats, Information Interchange-Representation of Dates and Times. International Standards Organisation, 1998.  
<http://www.iso.ch/cate/d15903.html>

## 971 6 Annex A — Syntax and Lexical Notation

972 The syntax is expressed in standard EBNF format. For completeness, the notation is given in *Table 2*.

973

Grammar rule component	Example
Terminal tokens are enclosed in double quotes	" ( "
Non terminals are written as capitalised identifiers	Expression
Square brackets denote an optional construct	[ ", " OptionalArg ]
Vertical bar denotes an alternative	Integer   Real
Asterisk denotes zero or more repetitions of the preceding expression	Digit *
Plus denotes one or more repetitions of the preceding expression	Alpha +
Parentheses are used to group expansions	( A   B ) *
Productions are written with the non-terminal name on the left-hand side, expansion on the right-hand side and terminated by a full stop	AnonTerminal = "an expansion".

974

975 **Table 2:** EBNF Rules

976

977

978 Some slightly different rules apply for the generation of lexical tokens. Lexical tokens use the same notation as above,  
979 with the exceptions noted in *Table 3*.

978

979

Lexical rule component	Example
Square brackets enclose a character set	[ "a", "b", "c" ]
Dash in a character set denotes a range	[ "a" - "z" ]
Tilde denotes the complement of a character set if it is the first character	[ ~ "(, )" ]
Post-fix question-mark operator denotes that the preceding lexical expression is optional (may appear zero or one times)	[ "0" - "9" ]? [ "0" - "9" ]

980

981

**Table 3:** Lexical Rules