1

FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS

2

3

4

5 # FIPA SL Content Language Specification

6

Document title	FIPA SL Content Language Specification		
Document number	SC00008I	Document source	FIPA TC Communication
Document status	Standard	Date of this status	2002/12/03
Supersedes	FIPA00003		
Contact	fab@fipa.org		
Change history	See *Informative Annex B — ChangeLog*		

7

8

9

10

11

12

13

14

15

16

17

Foreword

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

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

The FIPA specifications are developed through direct involvement of the FIPA membership. The status of a specification can be either Preliminary, Experimental, Standard, Deprecated or Obsolete. More detail about the process of specification may be found in the FIPA Document Policy [f-out-00000] and the FIPA Specifications Policy [f-out-00003]. A complete overview of the FIPA specifications and their current status may be found on the FIPA Web site.

FIPA is a non-profit association registered in Geneva, Switzerland. As of June 2002, the 56 members of FIPA represented many countries worldwide. Further information about FIPA as an organization, membership information, FIPA specifications and upcoming meetings may be found on the FIPA Web site at http://www.fipa.org/.

Contents

1   Scope

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

72 ## 2   Grammar FIPA SL Concrete Syntax

73 This content language is denoted by the normative constant `fipa-sl` in the `:language` parameter of an ACL
74 message. See Section 6 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    TermOrIE TermOrIE ")"
101                       | "(" PredicateSymbol TermOrIE+ ")"
102                       | "true"
103                       | "false".
104
105 BinaryTermOp          = "="
106                       | "result".
107
108 Quantifier            = "forall"
109                       | "exists".
110
111 ModalOp               = "B"
112                       | "U"
113                       | "PG"
114                       | "I".
115
116 ActionOp              = "feasible"
117                       | "done".
118
119 TermOrIE¹             = Term
120                       | IdentifyingExpression.
121
122 Term                  = Variable
123                       | FunctionalTerm
124                       | ActionExpression
125                       | Constant
126                       | Sequence
127                       | Set.
128
129 IdentifyingExpression = "(" ReferentialOperator TermOrIE Wff ")".
130
```

---

¹ Note that this grammar rule is used to group and represent both Terms and Identifying Expressions.

```
131   ReferentialOperator     = "iota"
132                           | "any"
133                           | "all".
134
135   FunctionalTerm          = "(" FunctionSymbol TermOrIE* ")"
136                           | "(" FunctionSymbol Parameter* ")".
137
138   Constant                = NumericalConstant
139                           | String
140                           | DateTime.
141
142   NumericalConstant       = Integer
143                           | Float.
144
145   Variable                = VariableIdentifier.
146
147   ActionExpression        = "(" "action" Agent TermOrIE ")"
148                           | "(" "|" ActionExpression ActionExpression ")"
149                           | "(" ";" ActionExpression ActionExpression ")".
150
151   PropositionSymbol       = String.
152
153   PredicateSymbol         = String.
154
155   FunctionSymbol          = String.
156
157   Agent                   = TermOrIE.
158
159   Sequence                = "(" "sequence" TermOrIE* ")".
160
161   Set                     = "(" "set" TermOrIE* ")".
162
163   Parameter               = ParameterName ParameterValue.
164
165   ParameterValue          = TermOrIE.
166
167
```

2.1 Lexical Definitions

All white space, tabs, carriage returns and line feeds between tokens should be skipped by the lexical analyser. See Section 6 for an explanation of the used notation.

```
172   String                  = Word
173                           | ByteLengthEncodedString
174                           | StringLiteral.
175
176   ByteLengthEncodedString = "#" DecimalLiteral+ "\"" <byte sequence>.
177
178   Word                    = [~ "\0x00" - "\0x20", "(", ")", "#", "0" - "9", ":", "-", "?"]
179                             [~ "\0x00" - "\0x20", "(", ")"]*.
180
181   ParameterName           = ":" String.
182
183   VariableIdentifier      = "?" String.
184
185   Sign                    = [ "+" , "-" ].
186
187   Integer                 = Sign? DecimalLiteral+
188                           | Sign? "0" ["x", "X"] HexLiteral+.
189
190   Dot                     = "."
191
192   Float                   = Sign? FloatMantissa FloatExponent?
```

```
193                              | Sign? DecimalLiteral+ FloatExponent.
194
195  FloatMantissa           = DecimalLiteral+ Dot DecimalLiteral*
196                              | DecimalLiteral* Dot DecimalLiteral+.
197
198  FloatExponent           = Exponent Sign? DecimalLiteral+.
199
200  Exponent                = ["e","E"].
201
202  DecimalLiteral          = ["0" - "9"].
203
204  HexLiteral              = ["0" - "9", "A" - "F", "a" - "f"].
205
206  StringLiteral           = "\""( [~ "\""]
207                              | "\\\"" )*"\"".
208
209  DateTime                = Sign? Year Month Day "T" Hour Minute
210                               Second MilliSecond TypeDesignator?.
211
212  Year                    = DecimalLiteral DecimalLiteral DecimalLiteral DecimalLiteral.
213
214  Month                   = DecimalLiteral DecimalLiteral.
215
216  Day                     = DecimalLiteral DecimalLiteral.
217
218  Hour                    = DecimalLiteral DecimalLiteral.
219
220  Minute                  = DecimalLiteral DecimalLiteral.
221
222  Second                  = DecimalLiteral DecimalLiteral.
223
224  MilliSecond             = DecimalLiteral DecimalLiteral DecimalLiteral.
225
226  TypeDesignator          = ["a" - "z" , "A" - "Z"].
227
```

3   Notes on FIPA SL Semantics

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

3.1   Grammar Entry Point: FIPA SL Content Expression

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

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

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

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

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

3.2   Well-Formed Formulas

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

- `(not <Wff>)`
  Negation. The truth value of this expression is false if `Wff` is true. Otherwise it is true.

- `(and <Wff0> <Wff1>)`
  Conjunction. This expression is true iff[2] well-formed formulae `Wff0` and `Wff1` are both true, otherwise it is false.

- `(or <Wff0> <Wff1>)`
  Disjunction. This expression is false iff well-formed formulae `Wff0` and `Wff1` are both false, otherwise it is true.

- `(implies <Wff0> <Wff1>)`
  Implication. This expression is true if either `Wff0` is false or alternatively if `Wff0` is true and `Wff1` is true. Otherwise it is false. The expression corresponds to the standard material implication connective `Wff0`   `Wff1`.

- `(equiv <Wff0> <Wff1>)`
  Equivalence. This expression is true if either `Wff0` is true and `Wff1` is true, or alternatively if `Wff0` is false and `Wff1` is false. Otherwise it is false.

- `(forall <variable> <Wff>)`
  Universal quantification. The quantified expression is true if `Wff` is true for every value of value of the quantified variable.

- `(exists <variable> <Wff>)`

---

[2] If and only if.

277   Existential quantification. The quantified expression is true if there is at least one value for the variable for which
278   `Wff` is true.
279

280   •   `(B <agent> <expression>)`
281   Belief. It is true that `agent` believes that `expression` is true.
282

283   •   `(U <agent> <expression>)`
284   Uncertainty. It is true that `agent` is uncertain of the truth of `expression`. `Agent` neither believes `expression`
285   nor its negation, but believes that `expression` is more likely to be true than its negation.
286

287   •   `(I <agent> <expression>)`
288   Intention. It is true that `agent` intends that `expression` becomes true and will plan to bring it about.
289

290   •   `(PG <agent> <expression>)`
291   Persistent goal. It is true that `agent` holds a persistent goal that `expression` becomes true, but will not
292   necessarily plan to bring it about.
293

294   •   `(feasible <ActionExpression> <Wff>)`
295   It is true that `ActionExpression` (or, equivalently, some event) can take place and just afterwards `Wff` will be
296   true.
297

298   •   `(feasible <ActionExpression>)`
299   Same as `(feasible <ActionExpression> true)`.
300

301   •   `(done <ActionExpression> <Wff>)`
302   It is true that `ActionExpression` (or, equivalently, some event) has just taken place and just before that `Wff` was
303   true.
304

305   •   `(done <ActionExpression>)`
306   Same as `(done <ActionExpression> true)`.
307

308 3.3   Atomic Formula

309 The atomic formula represents an expression which has a truth value in the language of the domain of discourse. Three
310 forms are defined:
311

312   •   A given propositional symbol may be defined in the domain language, which is either true or false,
313

314   •   Two terms may or may not be equal under the semantics of the domain language, or,
315

316   •   Some predicate is defined over a set of zero or more arguments, each of which is a term.
317

318 The FIPA SL representation does not define a meaning for the symbols in atomic formulae: this is the responsibility of
319 the domain language representation and ontology. Several forms are defined:
320

321   •   `true false`
322   These symbols represent the true proposition and the false proposition.
323

324   •   `(= Term1 Term2)`
325   `Term1` and `Term2` denote the same object under the semantics of the domain.
326

327 Other predicates may be defined over a set of arguments, each of which is a term, by using the `(PredicateSymbol`
328 `Term+)` production.
329

330 The FIPA SL representation does not define a meaning for other symbols in atomic formulae: this is the responsibility of
331 the domain language representation and the relative ontology.
332

3.4 Terms

334 Terms are either themselves atomic (constants and variables) or recursively constructed as a functional term in which a
335 functor is applied to zero or more arguments. Again, FIPA SL only mandates a syntactic form for these terms. With
336 small number of exceptions (see below), the meanings of the symbols used to define the terms are determined by the
337 underlying domain representation.
338

339 Note that, as mentioned above, no legal well-formed expression contains a free variable, that is, a variable not declared
340 in any scope within the expression. Scope introducing formulae are the quantifiers (`forall`, `exists`) and the
341 reference operators `iota`, `any` and `all`. Variables may only denote terms, not well-formed formulae.
342

3.5 Referential Operators

3.5.1 Iota

345 • (iota <term> <formula>)
346   The `iota` operator introduces a scope for the given expression (which denotes a term), in which the given
347   identifier, which would otherwise be free, is defined. An expression containing a free variable is not a well-formed
348   FIPA SL expression. The expression `(iota x (P x))` may be read as "the `x` such that `P` [is true] of `x`". The `iota`
349   operator is a constructor for terms which denote objects in the domain of discourse.
350

351   Notice that, unlike a term, an identifying expression can have different interpretations by different agents because
352   its formal definition depends on the KB.
353

354 • **Formal Definition**
355   A `iota` expression can only be evaluated with respect to a given theory. Suppose KB is a knowledge base such
356   that T (KB) is the theory generated from KB by a given reasoning mechanism. Formally, $\iota(\tau, \phi)=\theta\tau$ iff $\theta\tau$ is a term
357   that belongs to the set $\Sigma=\{\theta\tau: \theta\phi\in$T (KB)$\}$ and Σ is a singleton; or $\iota(\tau, \phi)$ is undefined if Σ is not a singleton. In this
358   definition θ is a most general variable substitution, $\theta\tau$ is the result of applying θ to τ, and $\theta\phi$ is the result of applying
359   θ to ϕ. This implies that a failure occurs if no object or more than one object satisfies the condition specified in the
360   `iota` operator.
361

362   If $\iota(\tau, \phi)$ is undefined then any term, identifying expression or well-formed formula containing $\iota(\tau, \phi)$ is also
363   undefined.
364

365 • **Example 1**
366   This example depicts an interaction between agent A and B that makes use of the `iota` operator, where agent A is
367   supposed to have the following knowledge base KB={P(A), Q(1, A), Q(1, B)}.
368

```
(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
```

383   `:in-reply-to query1)`

384

385   The only object that satisfies proposition P(x) is `a`, therefore, the `query-ref` message is replied by the `inform`

386   message as shown.

387

388   • **Example 2**

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

390

```
391    (query-ref
392      :sender (agent-identifier :name B)
393      :receiver (set (agent-identifier :name A))
394      :content
395        "((iota ?x (q ?x ?y)))"
396      :language fipa-sl
397      :reply-with query2)
398
399    (inform
400      :sender (agent-identifier :name A)
401      :receiver (set (agent-identifier :name B))
402      :content
403        "((= (iota ?x (q ?x ?y)) 1))"
404      :language fipa-sl
405      :in-reply-to query2)
```

406

407   The most general substitutions θ such that $\theta Q(x, y)$ can be derived from KB are $\theta_1=\{x/1, y/A\}$ and $\theta_2=\{x/1, y/B\}$.

408   Therefore, the set $\Sigma=\{\theta\tau: \theta\phi\in T(KB)\}=\{\{x/1, y/A\}x, \{x/1, y/B\}x \}=\{1\}$ is a singleton and hence `(iota ?x (q ?x ?y))`

409   represents the object 1.

410

411   • **Example 3**

412   Finally, this example shows an unsuccessful interaction using the `iota` operator. In this case, agent A cannot

413   evaluate the `iota` expression and therefore a failure message is returned to agent B

414

```
415    (query-ref
416      :sender (agent-identifier :name B)
417      :receiver (set (agent-identifier :name A))
418      :content
419        "((iota ?y (q ?x ?y)))"
420      :language fipa-sl
421      :reply-with query3)
422
423    (failure
424      :sender (agent-identifier :name A)
425      :receiver (set (agent-identifier :name B))
426      :content
427        "((action (agent-identifier :name A)
428          (inform-ref
429            :sender (agent-identifier :name A)
430            :receiver (set (agent-identifier :name B))
431            :content
432              \"((iota ?y (q ?x ?y)))\"
433            :language fipa-sl
434            :in-reply-to query3))"
435        more-than-one-answer)
436      :language fipa-sl
437      :in-reply-to query3)
```

438

439   The most general substitutions that satisfy Q(x, y) are $\theta_1=\{x/1, y/a\}$ and $\theta_2=\{x/1, y/b\}$, therefore, the set $\Sigma=\{\theta\tau:$

440        $\theta\phi\in T(KB)\}=\{\{x/1, y/A\}y, \{x/1, y/B\}y\}=\{A, B\}$, which is not a singleton. This means that the `iota` expression used in

441   this interaction is not defined.

442

443 **3.5.2   Any**

444 • (any <term> <formula>)
445 The `any` operator is used to denote any object that satisfies the proposition represented by `formula`.

446
447 Notice that, unlike a term, an identifying expression can have different interpretations by different agents because
448 its formal definition depends on the KB.

449
450 • **Formal Definition**
451 An `any` expression can only be evaluated with respect to a given theory. Suppose KB is a knowledge base such
452 that T(KB) is the theory generated from KB by a given reasoning mechanism. Formally, any(τ, ϕ)=$\theta\tau$ iff $\theta\tau$ is a term
453 that belongs to the set Σ={$\theta\tau$: $\theta\phi \in$ T(KB)}; or any(τ, ϕ) is undefined if Σ is the empty set. In this definition θ is a most
454 general variable substitution, $\theta\tau$ is the result of applying θ to τ, and $\theta\phi$ is the result of applying θ to ϕ.

455
456 If the set Σ is empty then any term, identifying expression or well-formed formula containing any(τ, ϕ) is undefined.

457
458 If the set Σ is not empty, then for any formula ψ containing any(τ, ϕ) let ψ' be the formula obtained from ψ by
459 replacing any(τ, ϕ) with a variable x (not occurring in ψ) and let s_k be a new Skolem constant. Then ψ is true when
460 {x/s_k}ψ' element_of T(KB union {τ/s_k}ϕ), ψ is false when {x/s_k}not(ψ') element_of T(KB union {τ/s_k}ϕ), and
461 otherwise ψ is undefined.

462
463 In other words if ψ contains any(τ, ϕ), ψ is true if a modified form of ψ obtained by replacing the any expression in it
464 with a new constant s_k can be inferred based on the assumption that phi holds of s_k. ψ is false if not(ψ) inferred
465 in a similar way. This definition is needed to avoid the following contradiction:

466
467 ```
(implies
468   (and (= Stephen (any ?x (fipa-member ?x)))
469   (= Farooq (any ?x (fipa-member ?x))))
470   (= Stephen Farooq))
```

471
472  This definition implies that failures only occur if there are no objects satisfying the condition specified as the second
473  argument of the `any` operator.

474
475  If any($\tau$, $\phi$) is undefined then any term, identifying expression or well-formed formula containing any($\tau$, $\phi$) is also
476  undefined.

477
478  • **Example 4**
479  Assuming that agent A has the following knowledge base KB={P(A), Q(1, A), Q(1, B)}, this example shows a
480  successful interaction with agent A using the `any` operator.

481
482  ```
(query-ref
483    :sender (agent-identifier :name B)
484    :receiver (set (agent-identifier :name A))
485    :content
486       "((any (sequence ?x ?y) (q ?x ?y)))"
487    :language fipa-sl
488    :reply-with query1)
```

489
490 ```
(inform
491   :sender (agent-identifier :name A)
492   :receiver (set (agent-identifier :name B))
493   :content
494   "((= (any (sequence ?x ?y) (q ?x ?y)) (sequence 1 a)))"
495   :language fipa-sl
496   :in-reply-to query1)
```

497

498    The most general substitutions θ such that θQ(x, y) can be derived from KB are {x/1, y/A} and {x/1, y/B}, therefore
499    Σ={θSequence(x, y): θQ(x, y)∈T(KB)}={Sequence(1, A), Sequence(1, B)}. Using this set, agent A chooses the first
500    element of Σ as the appropriate answer to agent B.

501

502    • **Example 5**
503    This example shows an unsuccessful interaction with agent A, using the `any` operator.

504

```
505    (query-ref
506      :sender    (agent-identifier :name B)
507      :receiver (set (agent-identifier :name A))
508      :content
509        "((any ?x (r ?x)))"
510      :language fipa-sl
511      :reply-with query2)

512

513    (failure
514      :sender (agent-identifier :name A)
515      :receiver (set (agent-identifier :name B))
516      :content
517        "((action (agent-identifier :name A)
518            (inform-ref
519               :sender (agent-identifier :name A)
520               :receiver (set (agent-identifier :name B))
521               :content
522                 \"((any ?x (r ?x)))\"
523               :language fipa-sl
524               :in-reply-to query2))
525         (unknown-predicate r))"
526      :language fipa-sl
527      :in-reply-to query2)
```

528

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

532

533   3.5.3   All

534   • `(all <term> <formula>)`
535   The `all` operator is used to denote the set of all objects that satisfy the proposition represented by `formula`.

536

537   Notice that, unlike a term, an identifying expression can have different interpretations by different agents because
538   its formal definition depends on the KB.

539

540   • **Formal Definition**
541   An `all` expression can only be evaluated with respect to a given theory. Suppose KB is a knowledge base such
542   that T(KB) is the theory generated from KB by a given reasoning mechanism. Formally, all(τ, φ)={θτ: θφ∈T(KB)}.
543   Notice that all(τ, φ) may be a singleton or even an empty set. In this definition θ is a most general variable
544   substitution, θτ is the result of applying θ to τ, and θφ is the result of applying θ to φ.

545

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

548

549   • **Example 6**
550   Suppose agent A has the following knowledge base KB={P(A), Q(1, A), Q(1, B)}. This example shows a successful
551   interaction between agent A and B that make use of the `all` operator.

552

```
553    (query-ref
554      :sender (agent-identifier :name B)
```

```
555        :receiver (set (agent-identifier :name A))
556        :content
557          "((all (sequence ?x ?y) (q ?x ?y)))"
558        :language fipa-sl
559        :reply-with query1)
560
561      (inform
562        :sender (agent-identifier :name A)
563        :receiver (set (agent-identifier :name B))
564        :content
565          "(( = (all (sequence ?x ?y) (q ?x ?y)) (set(sequence 1 a)(sequence 1 b))))"
566        :language fipa-sl
567        :in-reply-to query1)
```

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

```
576      (query-ref
577        :sender (agent-identifier :name B)
578        :receiver (set (agent-identifier :name A))
579        :content
580          "((all ?x (q ?x c)))"
581        :language fipa-sl
582        :reply-with query2)
583
584      (inform
585        :sender (agent-identifier :name A)
586        :receiver (set (agent-identifier :name B))
587        :content
588          "((= (all ?x (q ?x c))(set)))"
589        :language fipa-sl
590        :in-reply-to query2)
```

Since there is no possible substitution for x such that Q(x, C) can be derived from KB, then 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)`. The encoder is required to adopt the following criteria to select which form to use in order to represent a functional term. The first form, that is, the position-dependent form, should be used to encode all those functional terms for which the ontology does not specify the names of the parameters (for example, all the functions of the `fipa-agent-management` ontology). The second form, that is, the parameter-name dependent form, must be used to encode all those functional terms for which the ontology does specify the names of the parameters but not their position (for example, all the object descriptions of the `fipa-agent-management` ontology). 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:

11

```
613
614   (vehicle
615     :colour red
616     :max-speed 100
617     :owner (Person
618       :name Luis
619       :nationality Portuguese))
620
```

621 Some ontologies may decide to give a description of some concepts only in one or both of these two forms, that is by
622 specifying, or not, a default order to the arguments of each function in the domain of discourse. How this order is
623 specified is outside the scope of this specification.
624
625 Functional terms can be constructed by a domain functor applied to zero or more terms.
626

3.7   Result Predicate

628 A common need is to determine the result of performing an action or evaluating a term. To facilitate this operation, a
629 standard predicate `result`, of arity two, is introduced to the language. `result/2` has the declarative meaning that the
630 result of evaluating a term, or equivalently of performing an action, encoded by the first argument term, is the second
631 argument term. However, it is expected that this declarative semantics will be implemented in a more efficient,
632 operational way in any given FIPA SL interpreter.
633
634 A typical use of the `result` predicate is with a variable scoped by `iota`, giving an expression whose meaning is, for
635 example, "the `x` which is the result of agent `i` performing `act`":
636
```
637   (iota x (result (action i act) x)))
638
```

3.8   Actions and Action Expressions

640 Action expressions are a special subset of terms. An action itself is introduced by the keyword `action` and comprises
641 the agent of the action (that is, an identifier representing the agent performing the action) and a term denoting the action
642 which is [to be] performed.
643
644 Notice that a specific type of action is an ACL communicative act (CA). When expressed in FIPA SL, syntactically an
645 ACL communicative act is an action where the agent of the action is the `sender` of the CA, and the term denotes the
646 CA including all its parameters where the performative should be used as a function symbol, as referred by the used
647 ontology. Example 5 includes an example of an ACL CA, encoded as a `String`, whose content embeds another CA.
648
649 Two operators are used to build terms denoting composite CAs:
650
651 •   The sequencing operator (`;`) denotes a composite act in which the first action (represented by the first operand) is
652   followed by the second action, and,
653
654 •   The alternative operator (`|`) denotes a composite act in which either the first action occurs, or the second, but not
655   both.
656

3.9   Notes on the Grammar Rules

658 1.   The standard definitions for integers and floating point are assumed. However, due to the necessarily unpredictable
659   nature of cross-platform dependencies, agents should not make strong assumptions about the precision with which
660   another agent is able to represent a given numerical value. FIPA SL assumes only 32-bit representations of both
661   integers and floating point numbers. Agents should not exchange message contents containing numerical values
662   requiring more than 32 bits to encode precisely, unless some prior arrangement is made to ensure that this is valid.
663
664 2.   All keywords are case-insensitive.

3. A length encoded string is a context sensitive lexical token. Its meaning is as follows: the message envelope of the token is everything from the leading # to the separator " (inclusive). Between the markers of the message envelope is a decimal number with at least one digit. This digit then determines that *exactly* that number of 8-bit bytes are to be consumed as part of the token, without restriction. It is a lexical error for less than that number of bytes to be available.

4. Note that not all implementations of the ACC (see [FIPA00067]) will support the transparent transmission of 8-bit characters. It is the responsibility of the agent to ensure, by reference to internal API of the ACC, that a given channel is able to faithfully transmit the chosen message encoding.

5. Strings encoded in accordance with [ISO2022] may contain characters which are otherwise not permitted in the definition of `Word`. These characters are ESC (`0x1B`), SO (`0x0E`) and SI (`0x0F`). This is due to the complexity that would result from including the full [ISO2022] grammar in the above EBNF description. Hence, despite the basic description above, a word may contain any well-formed [ISO2022] encoded character, other (representations of) parentheses, spaces, or the # character. Strings must be enclosed between quote symbols. If the quote symbol itself needs to be part of the String, then it must be escaped by a `\` character.

6. The format for time tokens is defined in section 3.10.

7. An agent is represented by its agent-identifier using the standard format from [FIPA00023].

3.10 Representation of Time

Time tokens are based on [ISO8601], with extension for relative time and millisecond durations. Time expressions may be absolute, or relative. Relative times are distinguished by the sign character + or – appearing as the first character in the token. If no type designator is given, the local time zone is then used. The type designator for UTC is the character `Z`; UTC is preferred to prevent time zone ambiguities. Note that years must be encoded in four digits. As an example, 8:30 am on 15th April, 1996 local time would be encoded as:

```
19960415T083000000
```

The same time in UTC would be:

```
19960415T083000000Z
```

while one hour, 15 minutes and 35 milliseconds from now would be:

```
+00000000T011500035
```

704   # 4   Reduced Expressivity Subsets of FIPA SL

705   The FIPA SL definition given above is a very expressive language, but for some agent communication tasks it is
706   unnecessarily powerful. This expressive power has an implementation cost to the agent and introduces problems of the
707   decidability of modal logic. To allow simpler agents, or agents performing simple tasks, to do so with minimal
708   computational burden, this section introduces semantic and syntactic subsets of the full FIPA SL content language for
709   use by the agent when it is appropriate or desirable to do so. These subsets are defined by the use of profiles, that is,
710   statements of restriction over the full expressive power of FIPA SL. These profiles are defined in increasing order of
711   expressivity as FIPA-SL0, FIPA-SL1 and FIPA-SL2.
712
713   Note that these subsets of FIPA SL, with additional ontological commitments (that is, the definition of domain predicates
714   and constants) are used in other FIPA specifications.
715

716   ## 4.1   FIPA SL0: Minimal Subset

717   Profile 0 is denoted by the normative constant `fipa-sl0` in the `language` parameter of an ACL message. Profile 0 of
718   FIPA SL is the minimal subset of the FIPA SL content language. It allows the representation of actions, the
719   determination of the result a term representing a computation, the completion of an action and simple binary
720   propositions. The following defines the FIPA SL0 grammar:
721

```
722   Content             = "(" ContentExpression+ ")".
723
724   ContentExpression   = ActionExpression
725                       | Proposition.
726
727   Proposition         = Wff.
728
729   Wff                 = AtomicFormula
730                       | "(" ActionOp ActionExpression ")".
731
732   AtomicFormula       = PropositionSymbol
733                       | "(" "result"         Term Term ")"
734                       | "(" PredicateSymbol Term+ ")"
735                       | "true"
736                       | "false".
737
738   ActionOp            = "done".
739
740   Term                = Constant
741                       | Set
742                       | Sequence
743                       | FunctionalTerm
744                       | ActionExpression.
745
746   ActionExpression    = "(" "action" Agent Term ")".
747
748   FunctionalTerm      = "(" FunctionSymbol Term* ")"
749                       | "(" FunctionSymbol Parameter* ")".
750
751   Parameter           =  ParameterName ParameterValue.
752
753   ParameterValue      = Term.
754
755   Agent               = Term.
756
757   FunctionSymbol      = String.
758
759   PropositionSymbol   = String.
760
761   PredicateSymbol     = String.
```

```
762
763   Constant            = NumericalConstant
764                       | String
765                       | DateTime.
766
767   Set                 = "(" "set" Term* ")".
768
769   Sequence            = "(" "sequence" Term* ")".
770
771   NumericalConstant   = Integer
772                       | Float.
773
```

774   The same lexical definitions described in Section 2.1 apply for FIPA SL0.
775

776   **4.2   FIPA SL1: Propositional Form**

777   Profile 1 is denoted by the normative constant `fipa-sl1` in the `language` parameter of an ACL message. Profile 1 of
778   FIPA SL extends the minimal representational form of FIPA SL0 by adding Boolean connectives to represent
779   propositional expressions. The following defines the FIPA SL1 grammar:

```
780
781   Content             = "(" ContentExpression+ ")".
782
783   ContentExpression   = ActionExpression
784                       | Proposition.
785
786   Proposition         = Wff.
787
788   Wff                 = AtomicFormula
789                       | "(" UnaryLogicalOp  Wff ")"
790                       | "(" BinaryLogicalOp Wff Wff ")"
791                       | "(" ActionOp        ActionExpression ")".
792
793   UnaryLogicalOp      = "not".
794
795   BinaryLogicalOp     = "and"
796                       | "or".
797
798   AtomicFormula       = PropositionSymbol
799                       | "(" "result"        Term Term ")"
800                       | "(" PredicateSymbol Term+ ")"
801                       | "true"
802                       | "false".
803
804   ActionOp            = "done".
805
806   Term                = Constant
807                       | Set
808                       | Sequence
809                       | FunctionalTerm
810                       | ActionExpression.
811
812   ActionExpression    = "(" "action" Agent Term ")".
813
814   FunctionalTerm      = "(" FunctionSymbol Term* ")"
815                       | "(" FunctionSymbol Parameter* ")".
816
817   Parameter           = ParameterName ParameterValue.
818
819   ParameterValue      = Term.
820
821   Agent               = Term.
822
```

```
823    FunctionSymbol        = String.
824
825    PropositionSymbol     = String.
826
827    PredicateSymbol       = String.
828
829    Constant              = NumericalConstant
830                          | String
831                          | DateTime.
832
833    Set                   = "(" "set" Term* ")".
834
835    Sequence              = "(" "sequence" Term* ")".
836
837    NumericalConstant     = Integer
838                          | Float.
839
```

840   The same lexical definitions described in Section 2.1 apply for FIPA SL1.

841

4.3 FIPA SL2: Decidability Restrictions

843   Profile 2 is denoted by the normative constant `fipa-sl2` in the `language` parameter of an ACL message. Profile 2 of
844   FIPA SL allows first order predicate and modal logic, but is restricted to ensure that it must be decidable. Well-known
845   effective algorithms exist that can derive whether or not an FIPA SL2 Wff is a logical consequence of a set of Wffs (for
846   instance KSAT and Monadic). The following defines the FIPA SL2 grammar:

847
```
848    Content               = "(" ContentExpression+ ")".
849
850    ContentExpression     = IdentifyingExpression
851                          | ActionExpression
852                          | Proposition.
853
854    Proposition           = PrenexExpression.
855
856    Wff                   = AtomicFormula
857                          | "(" UnaryLogicalOp  Wff ")"
858                          | "(" BinaryLogicalOp Wff Wff ")"
859                          | "(" ModalOp         Agent PrenexExpression ")"
860                          | "(" ActionOp        ActionExpression ")"
861                          | "(" ActionOp        ActionExpression PrenexExpression ")".
862
863    UnaryLogicalOp        = "not".
864
865    BinaryLogicalOp       = "and"
866                          | "or"
867                          | "implies"
868                          | "equiv".
869
870    AtomicFormula         = PropositionSymbol
871                          | "(" "="           TermOrIE TermOrIE ")"
872                          | "(" "result"      TermOrIE TermOrIE ")"
873                          | "(" PredicateSymbol TermOrIE+ ")"
874                          | "true"
875                          | "false".
876
877    PrenexExpression      = UnivQuantExpression
878                          | ExistQuantExpression
879                          | Wff.
880
881    UnivQuantExpression   = "(" "forall" Variable Wff ")"
882                          | "(" "forall" Variable UnivQuantExpression ")"
883                          | "(" "forall" Variable ExistQuantExpression ")".
```

```
884
885   ExistQuantExpression  = "(" "exists" Variable Wff ")"
886                         | "(" "exists" Variable ExistQuantExpression ")".
887
888   TermOrIE              = Term
889                         | IdentifyingExpression.
890
891   Term                  =  Variable
892                         | FunctionalTerm
893                         | ActionExpression
894                         | Constant
895                         | Sequence
896                         | Set.
897
898   IdentifyingExpression = "(" ReferentialOp TermOrIE Wff ")".
899
900   ReferentialOp         = "iota"
901                         | "any"
902                         | "all".
903
904   FunctionalTerm        = "(" FunctionSymbol TermOrIE* ")"
905                         | "(" FunctionSymbol Parameter* ")".
906
907   Parameter             = ParameterName ParameterValue.
908
909   ParameterValue        = TermOrIE.
910
911   ActionExpression      = "(" "action" Agent TermOrIE ")"
912                         | "(" "|" ActionExpression ActionExpression ")"
913                         | "(" ";" ActionExpression ActionExpression ")".
914
915   Variable              = VariableIdentifier.
916
917   Agent                 = TermOrIE.
918
919   FunctionSymbol        = String.
920
921   Constant              = NumericalConstant
922                         | String
923                         | DateTime.
924
925   ModalOp               = "B"
926                         | "U"
927                         | "PG"
928                         | "I".
929
930   ActionOp              = "feasible"
931                         | "done".
932
933   PropositionSymbol     = String.
934
935   PredicateSymbol       = String.
936
937   Set                   = "(" "set" TermOrIE* ")".
938
939   Sequence              = "(" "sequence" TermOrIE* ")".
940
941   NumericalConstant      = Integer
942                          | Float.
943
944
945   The same lexical definitions described in Section 2.1 apply for FIPA SL2.
946
```

947   The `Wff` production of FIPA SL2 no longer directly contains the logical quantifiers, but these are treated separately to
948   ensure only prefixed quantified formulas, such as:
949
950   (forall ?x1
951   (forall ?x2
952   (exists ?y1
953   (exists ?y2
954   (Phi ?x1 ?x2 ?y1 ?y2)))))
955
956   Where `(Phi ?x1 ?x2 ?y1 ?y2)` does not contain any quantifier.
957
958   The grammar of FIPA SL2 still allows for quantifying-in inside modal operators. For example, the following formula is
959   still admissible under the grammar:
960
961   (forall ?x1
962   (or
963   (B i (p ?x1))
964   (B j (q ?x1))))
965
966   It is not clear that formulae of this kind are decidable. However, changing the grammar to express this context
967   sensitivity would make the EBNF form above essentially unreadable. Thus, the following additional mandatory
968   constraint is placed on well-formed content expressions using FIPA SL2: Within the scope of an `SLModalOperator`
969   only closed formulas are allowed, that is, formulas without free variables.
970

5   References

971

972   [FIPA00023]   FIPA Agent Management Specification. Foundation for Intelligent Physical Agents, 2000.
973   `http://www.fipa.org/specs/fipa00023/`
974   [FIPA00037]   FIPA Agent Communication Language Overview. Foundation for Intelligent Physical Agents, 2000.
975   `http://www.fipa.org/specs/fipa00037/`
976   [ISO8601]   Date Elements and Interchange Formats, Information Interchange-Representation of Dates and Times.
977   International Standards Organisation, 1998.
978   `http://www.iso.ch/cate/d15903.html`
979

980 ## 6   Informative Annex A — Syntax and Lexical Notation

981 The syntax is expressed in standard EBNF format. For completeness, the notation is given in *Table 2*.
982

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".`

983
984 **Table 2:** EBNF Rules
985

986 Some slightly different rules apply for the generation of lexical tokens. Lexical tokens use the same notation as above,
987 with the exceptions noted in *Table 3*.
988

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"]`

989
990 **Table 3:** Lexical Rules
991

7   Informative Annex B — ChangeLog

7.1   2002/11/01 - version H by TC X2S

994	Entire document:	Fixed bugs in the examples, by adding quotes and converting symbols into lower case
995 996	Entire document:	Added new non-terminal symbol `TermOrIE` and replaced all occurrences of `Term` with `TermOrIE`
997	Page 2, line 72:	Added symbol identifying `fipa-sl` content language
998	**Page 2, lines 104-112:**	**Removed superfluous binary term operators**
999	**Page 3, lines 139-149:**	**Removed superfluous functional term operators**
1000	**Page 3, lines 180-184:**	**Removed superfluous arithmetic operators**
1001	**Page 4, line 224:**	**Added optional `Sign` symbol to represent relative time**
1002	Pages 6, lines 342-373:	Removed description of superfluous equality operators
1003	Page 8, line 398:	Added note on interpretation of `iota` identifying expression
1004	Page 8, line 406:	Added note on interpretation of `iota` identifying expression
1005	Page 9, line 488 :	Added note on interpretation of `any` identifying expression
1006	Page 9, line 494:	Improved the definition of `any` identifying expression
1007	Page 9, line 497:	Improved the definition of `any` identifying expression
1008	Page 10, line 556:	Added note on interpretation of `all` identifying expression
1009	**Page 11, line 619:**	**Added requirement on encoding functional terms**
1010	Page 12, line 639:	Removed Table 1 on description of superfluous functional operators
1011	Page 12, lines 660-662:	Removed ambiguity in representing communicative acts in SL
1012	Page 12, line 664:	Added description of the actor of an ACL Message
1013	Page 13, lines 672-674:	Removed section on agent identifiers
1014	Page 13, lines 375-380:	Extended the section on Numerical Constants to incorporate more details on Grammar Rules
1015	Page 13, lines 682-692 :	Extended the section on Date and Time Constants to add a description of relative time

7.2   2002/12/03 - version I by FIPA Architecture Board

1018	Entire document:	Promoted to Standard status