FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS

FIPA SL Content Language Specification

Document title	FIPA SL Content Language Specification		
Document number	XC00008G	**Document source**	FIPA TC C
Document status	Experimental	**Date of this status**	2001/08/10
Supersedes	FIPA00003		
Contact	fab@fipa.org		
Change history			
2000/09/28	Approved for Experimental		
2001/08/10	Line numbering added		

Foreword

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

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

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

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

Contents

1 Scope

This 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 specified for message content given in [FIPA00037].

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

2 Grammar FIPA SL Concrete Syntax

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

```
Content                 = "(" ContentExpression+ ")".

ContentExpression       = IdentifyingExpression
                        | ActionExpression
                        | Proposition.

Proposition             = Wff.

Wff                     = AtomicFormula
                        | "(" UnaryLogicalOp  Wff ")"
                        | "(" BinaryLogicalOp Wff Wff ")"
                        | "(" Quantifier      Variable Wff ")"
                        | "(" ModalOp         Agent Wff ")"
                        | "(" ActionOp        ActionExpression ")"
                        | "(" ActionOp        ActionExpression Wff ")".

UnaryLogicalOp          = "not".

BinaryLogicalOp         = "and"
                        | "or"
                        | "implies"
                        | "equiv".

AtomicFormula           = PropositionSymbol
                        | "(" BinaryTermOp    Term Term ")"
                        | "(" PredicateSymbol Term+ ")"
                        | "true"
                        | "false".

BinaryTermOp            = "="
                        | "\="
                        | ">"
                        | ">="
                        | "<"
                        | "=<"
                        | "member"
                        | "contains"
                        | "result".

Quantifier              = "forall"
                        | "exists".

ModalOp                 = "B"
                        | "U"
                        | "PG"
                        | "I".

ActionOp                = "feasible"
                        | "done".

Term                    = Variable
                        | FunctionalTerm
                        | ActionExpression
                        | IdentifyingExpression
                        | Constant
                        | Sequence
                        | Set.
```

```
IdentifyingExpression  = "(" ReferentialOperator Term Wff ")".

ReferentialOperator    = "iota"
                       | "any"
                       | "all".

FunctionalTerm         = "(" "cons"         Term Term ")"
                       | "(" "first"        Term ")"
                       | "(" "rest"         Term ")"
                       | "(" "nth"          Term Term ")"
                       | "(" "append"       Term Term ")"
                       | "(" "union"        Term Term ")"
                       | "(" "intersection" Term Term ")"
                       | "(" "difference"   Term Term ")"
                       | "(" ArithmeticOp   Term Term ")"
                       | "(" FunctionSymbol Term* ")"
                       | "(" FunctionSymbol Parameter* ")".

Constant               = NumericalConstant
                       | String
                       | DateTime.

NumericalConstant      = Integer
                       | Float.

Variable               = VariableIdentifier.

ActionExpression       = "(" "action" Agent Term ")"
                       | "(" "|" ActionExpression ActionExpression ")"
                       | "(" ";" ActionExpression ActionExpression ")".

PropositionSymbol      = String.

PredicateSymbol        = String.

FunctionSymbol         = String.

Agent                  = Term.

Sequence               = "(" "sequence" Term* ")".

Set                    = "(" "set" Term* ")".

Parameter              = ParameterName ParameterValue.

ParameterValue         = Term.

ArithmeticOp           = "+"
                       | "-"
                       | "*"
                       | "/"
                       | "%".
```

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, Annex A — Syntax and Lexical Notation* for an explanation of the used notation.

```
String                 = Word
                       | StringLiteral.

Word                   = [~ "\0x00" - "\0x20", "(", ")", "#", "0" - "9", ":", "-", "?"]
                         [~ "\0x00" - "\0x20", "(", ")"]*.
```

```
ParameterName          = ":" String.

VariableIdentifier     = "?" String.

Sign                   = [ "+" , "-" ].

Integer                = Sign? DecimalLiteral+
                       | Sign? "0" ["x", "X"] HexLiteral+.

Dot                    = "."

Float                  = Sign? FloatMantissa FloatExponent?
                       | Sign? DecimalLiteral+ FloatExponent.

FloatMantissa          = DecimalLiteral+ Dot DecimalLiteral*
                       | DecimalLiteral* Dot DecimalLiteral+.

FloatExponent          = Exponent Sign? DecimalLiteral+.

Exponent               = ["e","E"].

DecimalLiteral         = ["0" - "9"].

HexLiteral             = ["0" - "9", "A" - "F", "a" - "f"].

StringLiteral          = "\""( [~ "\""]
                       | "\\\"" )*"\"".

DateTime               = Year Month Day "T" Hour Minute
                         Second MilliSecond TypeDesignator?.

Year                   = DecimalLiteral DecimalLiteral DecimalLiteral DecimalLiteral.

Month                  = DecimalLiteral DecimalLiteral.

Day                    = DecimalLiteral DecimalLiteral.

Hour                   = DecimalLiteral DecimalLiteral.

Minute                 = DecimalLiteral DecimalLiteral.

Second                 = DecimalLiteral DecimalLiteral.

MilliSecond            = DecimalLiteral DecimalLiteral DecimalLiteral.

TypeDesignator         = ["a" - "z" , "A" – "Z"].
```

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 `Wffs`) 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[1] 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>)

[1] If and only if.

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

`(B <agent> <expression>)`
Belief. It is true that `agent` believes that `expression` is true.

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

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

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

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

`(feasible <ActionExpression>)`
Same as `(feasible <ActionExpression> true)`.

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

`(done <ActionExpression>)`
Same as `(done <ActionExpression> true)`.

3.3 Atomic Formula

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

a given propositional symbol may be defined in the domain language, which is either true or false,

two terms may or may not be equal under the semantics of the domain language, or,

some predicate is defined over a set of zero or more arguments, each of which is a term.

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

`true false`
These symbols represent the true proposition and the false proposition.

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

`(\= Term1 Term2)`
`Term1` and `Term2` do not denote the same object 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 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 `(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, (,) iff is a term that belongs to the set { : T(KB)} and is a singleton; or (,) is undefined if is not a singleton. In this definition is a most general variable substitution, is the result of applying to , and is the result of applying to . 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-idenfier :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 such that Q(x, y) can be derived from KB are $_1$ {x/1, y/A} and $_2$ {x/1, y/B}. Therefore, the set { : T(KB)} {{x/1, y/A}x, {x/1, y/B}x } {1} is a singleton and hence `(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

```
(query-ref
  :sender (agent-identifier :name B)
  :receiver (set (agent-identifier :name A))
  :content
    ((iota ?y (q ?x ?y)))
  :language FIPA-SL
  :reply-with query3)

(failure
  :sender (agent-identifier :name A)
  :receiver (set (agent-identifier :name B))
  :content
    ((action (agent-identifier :name A)
             (inform-ref
              :sender (agent-identifier :name A)
              :receiver (set (agent-identifier :name B))
              :content
                 "((iota ?y (q ?x ?y)))"
              :language FIPA-SL
              :in-reply-to query3))
    more-than-one-answer)
  :language FIPA-SL
  :in-reply-to query3)
```

The most general substitutions that satisfy Q(x, y) are $_1$ {x/1, y/a} and $_2$ {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, any(,) iff is a term that belongs to the set { : T(KB)}; or any(,) is undefined if is the empty set. In this definition is a most general variable substitution, is the result of applying to , and is the result of applying to .

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.

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

(inform
  :sender (agent-identifier :name A)
```

```
  :receiver (set (agent-identifier :name B))
  :content
    ((= (any (sequence ?x ?y) (q ?x ?y)) (sequence 1 a)))
  :language FIPA-SL
  :in-reply-to query1)
```

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 { Sequence(x, y): Q(x, y) T(KB)}={Sequence(1, A), 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.

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

(failure
  :sender (agent-identifier :name A)
  :receiver (set (agent-identifier :name B))
  :content
    ((action (agent-identifier :name A)
             (inform-ref
               :sender (agent-identifier :name A)
               :receiver (set (agent-identifier :name B))
               :content
                 "((any ?x (r ?x)))"
               :language FIPA-SL
               :in-reply-to query2))
    (unknown-predicate r))
  :language FIPA-SL
  :in-reply-to query2)
```

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(KB) is the theory generated from KB by a given reasoning mechanism. Formally, all(,) { : T(KB)}. Notice that all(,) may be a singleton or even an empty set. In this definition is a most general variable substitution, is the result of applying to , and is the result of applying to .

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 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.

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

(inform
  :sender (agent-identifier :name A)
  :receiver (set (agent-identifier :name B))
  :content
    (( =   (all (sequence ?x ?y) (q ?x ?y)) (set(sequence 1 a)(sequence 1 b))))
  :language FIPA-SL
  :in-reply-to query1)
```

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 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.

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

(inform
  :sender (agent-identifier :name A)
  :receiver (set (agent-identifier :name B))
  :content
    ((= (all ?x (q ?x c))(set)))
  :language FIPA-SL
  :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)`. 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:

```
(vehicle
```

```
  :colour red
  :max-speed 100
  :owner (Person
    :name Luis
    :nationality Portuguese))
```

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

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

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.

Table 1: Functional Operators

3.7 Result Predicate

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

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

```
(iota x (result (action i act) x)))
```

3.8 Actions and Action Expressions

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

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

Two operators are used to build terms denoting composite CAs:

- the sequencing operator (`;`) denotes a composite act in which the first action (represented by the first operand) is followed by the second action, and,

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

3.9 Agent Identifiers

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

3.10 Numerical Constants

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

3.11 Date and Time Constants

Time tokens are based on [ISO8601], with extension for millisecond durations. 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
```

4 Reduced Expressivity Subsets of FIPA SL

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

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

4.1 FIPA SL0: Minimal Subset

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

```
Content            = "(" ContentExpression+ ")".

ContentExpression  = ActionExpression
                   | Proposition.

Proposition        = Wff.

Wff                = AtomicFormula
                   | "(" ActionOp ActionExpression ")".

AtomicFormula      = PropositionSymbol
                   | "(" "result"        Term Term ")"
                   | "(" PredicateSymbol Term+ ")"
                   | "true"
                   | "false".

ActionOp           = "done".

Term               = Constant
                   | Set
                   | Sequence
                   | FunctionalTerm
                   | ActionExpression.

ActionExpression   = "(" "action" Agent Term ")".

FunctionalTerm     = "(" FunctionSymbol Term* ")"
                   | "(" FunctionSymbol Parameter* ")".

Parameter          =  ParameterName ParameterValue.

ParameterValue     = Term.

Agent              = Term.

FunctionSymbol     = String.

PropositionSymbol  = String.

PredicateSymbol    = String.
```

```
Constant          = NumericalConstant
                  | String
                  | DateTime.

Set               = "(" "set" Term* ")".

Sequence          = "(" "sequence" Term* ")".

NumericalConstant = Integer
                  | Float.
```

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

4.2 FIPA SL1: Propositional Form

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

```
Content              = "(" ContentExpression+ ")".

ContentExpression    = ActionExpression
                     | Proposition.

Proposition          = Wff.

Wff                  = AtomicFormula
                     | "(" UnaryLogicalOp  Wff ")"
                     | "(" BinaryLogicalOp Wff Wff ")"
                     | "(" ActionOp        ActionExpression ")".

UnaryLogicalOp       = "not".

BinaryLogicalOp      = "and"
                     | "or".

AtomicFormula        = PropositionSymbol
                     | "(" "result"        Term Term ")"
                     | "(" PredicateSymbol Term+ ")"
                     | "true"
                     | "false".

ActionOp             = "done".

Term                 = Constant
                     | Set
                     | Sequence
                     | FunctionalTerm
                     | ActionExpression.

ActionExpression     = "(" "action" Agent Term ")".

FunctionalTerm       = "(" FunctionSymbol Term* ")"
                     | "(" FunctionSymbol Parameter* ")".

Parameter            = ParameterName ParameterValue.

ParameterValue       = Term.

Agent                = Term.
```

```
FunctionSymbol         = String.

PropositionSymbol      = String.

PredicateSymbol        = String.

Constant               = NumericalConstant
                       | String
                       | DateTime.

Set                    = "(" "set" Term* ")".

Sequence               = "(" "sequence" Term* ")".

NumericalConstant      = Integer
                       | Float.
```

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

4.3 FIPA SL2: Decidability Restrictions

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

```
Content                = "(" ContentExpression+ ")".

ContentExpression      = IdentifyingExpression
                       | ActionExpression
                       | Proposition.

Proposition            = PrenexExpression.

Wff                    = AtomicFormula
                       | "(" UnaryLogicalOp  Wff ")"
                       | "(" BinaryLogicalOp Wff Wff ")"
                       | "(" ModalOp         Agent PrenexExpression ")"
                       | "(" ActionOp        ActionExpression ")"
                       | "(" ActionOp        ActionExpression PrenexExpression ")".

UnaryLogicalOp         = "not".

BinaryLogicalOp        = "and"
                       | "or"
                       | "implies"
                       | "equiv".

AtomicFormula          = PropositionSymbol
                       | "(" "="             Term Term ")"
                       | "(" "result"        Term Term ")"
                       | "(" PredicateSymbol Term+ ")"
                       | "true"
                       | "false".

PrenexExpression       = UnivQuantExpression
                       | ExistQuantExpression
                       | Wff.

UnivQuantExpression    = "(" "forall" Variable Wff ")"
                       | "(" "forall" Variable UnivQuantExpression ")"
                       | "(" "forall" Variable ExistQuantExpression ")".
```

```
ExistQuantExpression  = "(" "exists" Variable Wff ")"
                      | "(" "exists" Variable ExistQuantExpression ")".

Term                  =  Variable
                      | FunctionalTerm
                      | ActionExpression
                      | IdentifyingExpression
                      | Constant
                      | Sequence
                      | Set.

IdentifyingExpression = "(" ReferentialOp Term Wff ")".

ReferentialOp         = "iota"
                      | "any"
                      | "all".

FunctionalTerm        = "(" FunctionSymbol Term* ")"
                      | "(" FunctionSymbol Parameter* ")".

Parameter             = ParameterName ParameterValue.

ParameterValue        = Term.

ActionExpression      = "(" "action" Agent Term ")"
                      | "(" "|" ActionExpression ActionExpression ")"
                      | "(" ";" ActionExpression ActionExpression ")".

Variable              = VariableIdentifier.

Agent                 = Term.

FunctionSymbol        = String.

Constant              = NumericalConstant
                      | String
                      | DateTime.

ModalOp               = "B"
                      | "U"
                      | "PG"
                      | "I".

ActionOp              = "feasible"
                      | "done".

PropositionSymbol     = String.

PredicateSymbol       = String.

Set                   = "(" "set" Term* ")".

Sequence              = "(" "sequence" Term* ")".

NumericalConstant      = Integer
                       | Float.
```

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

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

```
(forall ?x1
  (forall ?x2
    (exists ?y1
      (exists ?y2
        (Phi ?x1 ?x2 ?y1 ?y2)))))
```

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

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

```
(forall ?x1
  (or
    (B i (p ?x1))
    (B j (q ?x1))))
```

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

Within the scope of an `SLModalOperator` only closed formulas are allowed, that is, formulas without free variables.

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

6 Annex A — Syntax and Lexical Notation

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

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".`

Table 2: EBNF Rules

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

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"]`

Table 3: Lexical Rules