NISTIR 7868

A Restricted English for Constructing Ontologies (RECON)

Edward Barkmeyer
Andreas Mattas

http://dx.doi.org/10.6028/NIST.IR.7868

National Institute of Standards and Technology
U.S. Department of Commerce

NISTIR 7868

A Restricted English for Constructing Ontologies (RECON)

Edward Barkmeyer
Systems Integration Division
National Institute of Standards and Technology

Andreas Mattas
Aristotle University
Thessaloniki, Greece

http://dx.doi.org/10.6028/NIST.IR.7868

July 2012

U.S. Department of Commerce
Rebecca Blank, Acting Secretary

National Institute of Standards and Technology
Patrick D. Gallagher, Under Secretary of Commerce for Standards and Technology and Director

A Restricted English for Constructing Ontologies (RECON)

Ed Barkmeyer, NIST, Andreas Mattas, Aristotle U. Thessaloniki

Abstract:

Verifying correctness of stated facts, rules, and term definitions for an industrial domain of work requires the contributions of experts in the domain. Conversely, use of automated reasoning technologies to assist in making industry decisions and validating industry information demands that facts, rules and the definitions of terms be stated in a formal logic language. Such languages are very difficult for engineers and operations experts to understand, and generally difficult to use. A restricted English is a language that looks like English, but is carefully restricted in grammar, so that it has an unambiguous equivalent in a formal logic language. This paper presents the grammar for RECON – a restricted English language whose purpose is to capture industry concepts, facts and rules in such a way as to permit conversion to an extended first-order logic language (IKL). The vocabulary of the language – the nouns, verbs and adjectives – is defined by the industry experts. The grammar regulates the forms of definitions and sentences that use those terms. The RECON grammar supports the use of complex noun phrases involving individuals, groups, classifiers and properties, quantifying expressions and ad hoc classifiers that are qualified by adjectives and properties. It supports verb phrases involving verbs and prepositional phrases, and it supports simple, compound and conditional sentence structures. RECON English can be read by experts in the industry domain, and can be written more easily by analysts who are capturing the knowledge of those experts. It facilitates the transfer of human knowledge to computational forms.

Contents

Introduction

This report is a reference manual for RECON – a restricted English language whose purpose is to capture industry concepts, facts and rules, in a form that is readable by industry experts and that has, at the same time, an unambiguous interpretation in first-order logic. The intent is to allow an industry expert to state a fact or rule and to know that exactly what s/he stated will be used directly by certain software implementations.

Background

The NIST Engineering Data Quality Measurement project is concerned with the accuracy, consistency, completeness, and timeliness of information received from business partners via electronic messages. The approach used in the project is to capture the relevant concepts of the industry domain in a form suitable for machine reasoning – an ontology, and to capture the associated rules for the completeness and consistency of information sets in those terms. In practice, the received messages are converted into formal statements about the individual "things of the manufacturing business" in terms of the concepts in the ontology. Then an automated reasoner is used to evaluate the completeness of the information received against the stated completeness and timeliness rules, and to evaluate the consistency of the information received from multiple sources with the background knowledge of the manufacturing business unit.

Capturing term definitions, rules and basic facts for an industrial domain requires the contributions of experts in the domain. But for the formal ontology, the facts, rules and definitions of terms must be stated in a formal logic language. Such languages are very difficult for engineers and operations experts to understand, and generally difficult to use. So, to facilitate capturing the knowledge of the industry experts, the project developed an intermediate language – a "restricted English" called RECON ("Restricted English for Constructing Ontologies").

A *restricted English* is a language that looks like English, but is carefully restricted in grammar, so that every statement and most definitions have an unambiguous equivalent in a formal logic language. The experts in the industry domain may require the assistance of a knowledge engineer to state their intent in the restricted English language, but it is most important that they can read the English formulation and verify that it captures their intent. The stated definitions, facts, and rules will be used directly in validating incoming information.

RECON is a restricted English language whose purpose is to capture industry concepts, facts and rules, in such a way as to be readable by industry experts and to be unambiguously converted to an extended first-order logic language (IKL) (Hayes 2006). The vocabulary of the language –- the nouns, verbs and adjectives – is defined by the industry experts. This report specifies the grammar of the language, which regulates the forms of definitions and sentences that use those terms.

The RECON grammar defines an *interpretation* for the language – each RECON structure or combination of structures corresponds to a particular combination of IKL structures, which have a well-defined formal semantics. This report describes the interpretation of RECON constructs, but the formal interpretation is a set of algorithms whose details are to be presented in a separate paper.

The real semantics of each RECON statement – what it actually means – depends primarily on the terms used in the statement. Those terms are the ones the expert declares, and RECON renders them directly into IKL predicates, essentially 1-to-1. The meaning of the terms and the predicates is what the user defines it to be. Some terms can be defined formally in the RECON language, and those definitions will be rendered formally into IKL definitions of the predicates. Some set of terms, however, must be *primitive* – they have no formal definitions; they mean what the user thinks they mean; and all of the machine-interpretable meaning is ultimately based on them.

The RECON grammar supports the use of complex noun phrases involving individuals, groups, classifiers and properties, quantifying expressions and ad hoc classifications – nouns that are qualified by adjectives and properties. It supports verb phrases involving verbs and prepositional phrases, and it supports simple, compound and conditional sentence structures. All of this is described in detail below.

The fundamental idea is that the RECON language is English, can be read by experts in the industry domain, and can be written more easily than formal logic by analysts who are capturing the knowledge of those experts. At the same time, it has a well-defined mapping to a formal logic, and thus it facilitates the transfer of human knowledge to computational forms.

Related work

It is important to distinguish restricted English from natural language processing. The objective of natural language processing is to produce a formal interpretation of text as published. Natural language is in many cases ambiguous, and software for interpreting natural English is currently about 80% reliable in the best of cases. The objective of restricted English is to ensure that the text written in the language can be converted in every case to a particular formal language by a particular algorithm.

For our purposes, the target formal language is a formal logic language that can be used by emerging computational reasoning software. There are three general categories of computational reasoning tools currently in common use: first-order logic reasoners, description logic reasoners, and production rules engines. There are now also international standards for languages of each kind: ISO Common Logic (International Organization for Standardization 2004) for first-order logics, the Web Ontology Language (OWL) (World Wide Web Consortium (W3C) 2004) for description logics, and the Rules Interchange Format (RIF) (W3C 2010). These languages determine the nature of the formal logic statements that a controlled English tool can export, and thus limit what the controlled English language can usefully express. First-order logic is the most expressive logic language. The other two are subsets of first-order logic, and thus intrinsically less expressive (and correspondingly more efficient).

The earliest "restricted English" or "controlled English" languages were "macro" or "template" languages, in which the knowledge engineer defines a set of sentence forms with parameter markers and also the interpretation of each such sentence into some formal language with slots to be filled by the parameters. These are now commonly called "domain-specific (text) languages" (DSLs) and are supported by tooling such as XTEXT (The Eclipse Foundation 2011). In most such languages, the knowledge engineer predefines the verbs in each sentence form, and the domain expert introduces the nouns. In many industry domains, however, the domain expert must be able to introduce new verbs as well, and often needs to create conditional sentences or

more complex noun phrases for the "noun" parameters, and those requirements are beyond the capabilities of a template language.

The RABBIT language (Hart 2007) is a controlled English language that is designed to be rendered into the OWL language. So, RABBIT enables one to introduce a noun term that represents an OWL "class," and to say how classes are related, or to say that an individual is a member of a class. Similarly, RABBIT allows one to introduce verbs that are interpreted as OWL "properties" that relate pairs of individuals, and to use such properties in defining classes. Sentences in RABBIT are limited to simple forms, but nouns can have compound qualifiers that use defined "properties," i.e., verbs. RABBIT does not have the RECON "property" concept, but it does do something similar for "part of" relationships. Unlike the other controlled English languages we have studied, RABBIT tends to require statements to use OWL terminology, and that makes it only a slight improvement over the Manchester Syntax (W3C 2009) – a standard OWL representation – for readability by industry experts.

The Attempto Controlled English project (ACE) (Fuchs 1999) is a major academic project whose primary purpose was to develop a formal language for knowledge representation that could be rendered into formal languages for machine reasoning and other computational purposes. A major research thrust of ACE was support for natural language patterns, including "discourse representation" – references to the same things by pronouns and other terms in a sequence of sentences. The controlled English was developed to eliminate aspects of natural English that interfered with unambiguous interpretation of sentences. For example, ACE insists that every term consists of exactly one word, hyphenated if necessary. Because of its natural language bent, ACE understands verbs only as transitive or intransitive (binary or unary). It considers indirect objects and prepositional phrases to be adverbial clauses. The consequence is that the formal logic model of ACE sentences often involves *nominalization* of simple subject-verb(-object) forms, so that they can be operands of pseudo-verbs that attach the adverbial phrases. For example, "Mary goes to the store" becomes "The situation 'Mary goes' (is)to the store." "The situation 'Mary goes'" is a noun-like thing that is the nominalization of the proposition "Mary goes" and is then the subject of the pseudo-verb "(is)to." Such constructs are not inconvenient for description logic reasoners, but they are nearly useless for first-order logic reasoning.

The CELT language (Pease 2003) is a controlled English language that is designed to express facts and queries, and possibly new concepts, based on an existing general upper ontology (SUMO (Niles 2001)). CELT exports formal statements in KIF, a dialect of ISO Common Logic, and thus in a useful first-order logic form. Unlike RECON, whose objective is to process statements in domain-specific terminology, the purpose of CELT is to process natural language terminology and interpret it under the SUMO ontology, using a ready-made set of mappings from usage cues to word sense and word sense to SUMO concepts. As a consequence, the formal logic output of CELT is phrased entirely in terms of predicates defined in SUMO. Among other things, this means CELT turns most expressions of relationships or events into instances of relationship or event classes that are then related to the things that play roles in them by predicates that represent SUMO-defined roles in generic categories of relationships and events. By comparison, RECON creates a specific predicate for each verb declared by the user and places the role player objects in the corresponding argument slots. The CELT grammar is not published, but it is said to use two sophisticated formulation technologies, which is in keeping with the attempt to process word sequences to meaning and thence to formulation. That said, the limited description of CELT syntax suggests that it supports most of the grammatical features of

RECON. On the other hand, the CELT idea of interpreting most verbs into event/relationship nouns makes the CELT handling of negation problematic. Interestingly, the CELT publications suggest that CELT and RECON also make different interpretations of some (controlled) English usages into logical quantifications.

The Common Logic Controlled English language (CLCE) (Sowa, Common Logic Controlled English 2004) is, as the name implies, intended to express arbitrary first-order logic propositions that can be rendered into Common Logic. The objective of CLCE is very similar to that of RECON – formulating statements that can be rendered directly into formal logic – and it has a formal grammar. Like RECON, CLCE has explicit notions for common nouns, proper names, "relational nouns" (RECON "properties") adjectives, and verbs. It supports logical quantification expressions, but not quantity expressions, and it supports explicit variables and simple back references. The online publication is said to be incomplete, and it leaves some serious questions unanswered. While the concept of a noun phrase including a qualifying clause introduced by "that" is mentioned, it is not apparent in the grammar. Similarly, although verb concepts that correspond to logical predicates with three or more arguments are mentioned in examples, it is unclear how these are declared, but the grammar provides for the attachment of prepositional phrases to simple sentence forms, which may or may not be the same concept as "particle phrases" in RECON. In so many words, the CLCE document appears to cover most of the RECON technical ground, but the sketchy specification makes the truth/depth of that coverage inaccessible. There is apparently a major difference in the handling of terms and dealing with homographs; CLCE is intentionally simplistic in this area, while RECON attempts to support real domain terminologies.

The Semantics of Business Vocabulary and Rules specification (SBVR) (Object Management Group (OMG) 2010) is an international standard for capturing arbitrary vocabularies, and facts and rules that use terms in those vocabularies. SBVR introduces a restricted English language called Structured English, which is used throughout the specification, but, curiously, is not actually standardized by the specification. The Structured English language is described less than formally in an annex, but the exported logical form is an enhanced first-order logic language that is formally described. The logic enhancements include proposition nominalization and modalities for necessity/possibility and obligation/permission. SBVR supports multi-word terms and it allows the definition of verb usage patterns that involve prepositional phrases, thus supporting "verb concepts" of arbitrary arity. As near as one can tell from the description of SBVR Structured English, it can express almost anything RECON English can. It allows statements about statements and statements about situations, it allows complex qualifications of noun terms, and it has a notion of attribute or property of a noun term (although it is strangely defined). Unfortunately, SBVR Structured English seems to rely on vocabulary terms being marked up in the text, so that the parser can recognize them. So "Mary goes to the store" is written "Mary *goes to* the store." How a tool might interact with a user to make this markup less than onerous is not discussed. The absence of a formal grammar for the language also makes the limitations on expression, or the possibility of ambiguity, unclear.

RECON explicitly allows for terms that include multiple words without markup. It provides verb usage templates and supports verbs of arbitrary arity. RECON explicitly supports attributes (or "properties"), and consequently, it supports genitive pronouns. It supports complex qualifications on nouns that may involve intermediate objects. And unlike any other controlled English we have seen, it supports compound noun phrases, which we have found to be common in the write-

ups of usage of messages. The most comprehensive restricted English language we have found is SBVR Structured English, and anything written in SBVR Structured English can be written in RECON, and more easily. And unlike SBVR Structured English, RECON is formally defined.

This report is that formal definition.

Structure of this report

This report defines the RECON language. The RECON language has two components:

- vocabulary terms: terms created, and possibly defined, for a particular domain of use. These are the terms – nouns, verbs, adjectives, etc. – that have meaning to the domain practitioners.
- grammatical structures: the language rules for composing definitions and sentences from vocabulary terms

The three principal sections of this report describe the general form of a RECON document, the detailed forms of vocabulary declarations, and the grammar for formal definitions and sentences, respectively.

The vocabulary declarations and the grammatical structures of the RECON language are formally defined in the Backus Naur form that is described in Appendix A: Form of the Grammar.

This report does not describe the generation of formal logic statements (IKL) from the definitions and sentences contained in a RECON document. That will be described in a separate report.

Status of the work

The RECON language presented in this report is deemed to be final. It is unlikely that there will be a need to add grammatical features to the language. There may be some fixed vocabularies that will have special meaning to future tooling, but such additions will not change the grammar.

The current RECON tooling supports all the language features described in this report, and generates formal IKL interpretations for all definitions and sentences. The generated IKL is based on some specific decisions about formal models for quantities, collections, situations and nominalized concepts. Those decisions may change, with the consequence that future versions of RECON may generate somewhat different IKL. Such changes, however, will not affect the grammar described here.

The current RECON tooling supports only the mechanisms for vocabulary declarations that are described in this report. That is, the current tooling only supports a text interface for such declarations. Some tooling to support importing vocabularies in standard forms, such as OWL and SBVR, is planned, and other forms may be supported in the more distant future.

Additional tooling is in work to provide a more interactive mechanism for the capture of bodies of knowledge with RECON, using a graphical user interface (GUI).

We see immediate applications for RECON in two areas of NIST work: capturing engineering requirements, and validating operational message flows in terms of content, consistency and completeness, that is, “fitness for purpose.”

Structure of a RECON document

A RECON document is an *ontology* – a representation of knowledge in a form that can be processed by a machine reasoning tool. The RECON tool transforms the RECON document into a document in a standard formal logic language.

A RECON document can be, or include, a *vocabulary* – a set of terms that are used to describe things of interest for a particular application, together with definitions of the terms.

A RECON document can also be, or include, a *model* – a body of statements in the RECON language that state facts, rules, and required relationships for a particular industrial domain.

```
reconDocument = { model | vocabulary } ;
model = { sentence | comment | directive } ;
vocabulary = { entry | comment | directive } ;
```

Vocabularies are described below under Vocabularies, Terms and Declarations.

Models are described below under Models, Sentences, and Definitions.

Directives and comments

A `directive` is a line beginning with a dollar sign character ($) and ending with an end-of-line character. It provides some directions to the RECON tool about processing resources containing vocabularies and models. Directives are described in Appendix B: RECON Directives.

A `comment` is a line beginning with two solidus characters (//), followed by any text, and ending with an end-of-line. The RECON tool ignores comments.

Vocabularies, Terms and Declarations

Domain vocabularies introduce specialized concepts and denote them by specific words or phrases, which we call *terms*. The RECON language grammar assumes the existence of a domain vocabulary. That is, it assumes that every term in the language is somehow declared as part of the vocabulary of the language. The only terms the RECON tool understands directly are those of the grammar itself.

RECON provides syntax for declaring and defining vocabulary terms. This section describes the vocabulary concepts and the declaration syntax. Other representations of a domain vocabulary may also be supported by the RECON tooling, but they are not addressed in this report.

The entries in a vocabulary are *terminological entries*. The fundamental element of a terminological entry is a concept of some kind, called a *vocabulary item*. A terminological entry consists of one or more *declarations*, each of which provides either a term for the vocabulary item or a definition of the vocabulary item in some language.

```
vocabulary = { entry | comment | directive } ;

entry = nounEntry | verbEntry | propertyEntry | adjectiveEntry |
        nameEntry | unitEntry | axiom ;
```

The general form of a declaration is a single character that specifies the type of declaration, followed by whitespace (space or tab characters) and then the body of the declaration. Declarations are described in detail below.

An end-of-line is considered to be the end of a declaration, unless it is followed by a space or tab character. End-of-line followed by a space or tab is treated as a space.

A terminological entry always begins with the declaration of a term for the vocabulary item, called the *primary term*. The primary term declaration may be followed by synonyms and definitions, and perhaps other declarations, that are part of the terminology entry and are associated with the primary term.Technically, a vocabulary item has exactly one terminological entry that contains all the declarations for the item. It is, however, possible to create another terminological entry whose formal definition makes it equivalent.

Words

A *lexical element* of a language is a sequence of characters that is a unit of interpretation of text in the language. The basic lexical element of the RECON language is a `wordForm`. A `wordForm` is any sequence of characters that does not include any of the following "breaking characters":

- space, tab, end-of-line
- comma (,), colon (:), semicolon (;), left parentheses ((), right parenthesis ())
- quotation mark (")
- asterisk (*), solidus (/), circumflex (^) and dollar sign ($).

A period (.) is permitted within a `wordForm`, but a period that is followed by a breaking character is treated as a separate `wordForm`.

In creating and recognizing terms, the RECON parser uses the more general concept `word`. A `word` that is an English word may appear as several different `wordForm`s, depending on the part of speech. So the concept `word` is further distinguished as:

`keyword` = a word that is explicitly used in the grammar

`noun` = a word recognized as a noun, having singular and plural forms

`verb` = a word recognized as a verb, having multiple number and tense forms

`adjective` = a word recognized as an adjective

`adjunct` = a recognized English word that is none of the above, and has only one form

`symbol` = a punctuation mark or other special character that has grammatical significance. The RECON symbols are: period (.), comma (,), colon (:), semicolon (;), parentheses (()), question mark (?), asterisk (*), solidus (/), circumflex (^), and end-of-line.

`name` = a `wordForm` that is not in the dictionary, treated as a word that has only one form

A term declaration specifies the words that make up the term. Different forms of a word are recognized as the same word and thus the same term.

Note: keywords are not necessarily "reserved." Some are reserved to their English usage, like `if` and `each`; others, like `and`, `at`, and `of`, can also be parts of defined terms.

Text

`text` is a lexical element that is not decomposed into words and other symbols. It consists of any sequence of characters followed by an end-of-line.

It is used for the body of natural language definitions and for comments.

Vocabulary Nouns

A *vocabulary noun* (`vocNoun`) is a term that denotes a category of things. There are two kinds of vocabulary nouns – type nouns and mass nouns – and they are similar in most regards.

```
nounEntry = ( typeDeclaration | massDeclaration ),
          { nounSynonym | definition | nounFormulation } ;
```

The general form of a vocabulary noun term is a sequence of words:

```
vocNoun = word, { word } ;
```

The vocabulary noun term being declared may include any combination of words, all of which are part of the term and must be present for the term to be recognized in use. Multiple forms of the same word will be recognized as the same `word` (see General Recognition Rules).

A vocabulary noun declaration of either kind begins a terminological entry. The entry may also contain synonyms and formal or informal definitions.

Type Nouns

A *type noun* is a term that denotes a category of individual things. The declaration for a type noun has the form:

```
typeDeclaration = "T", vocNoun ;
```

Examples:

```
T vehicle
T principal engineer
T system of units
```

Mass Nouns

A *mass noun* is a term that refers to a kind of substance, a concept that is not instantiated by individual things. The declaration for a mass noun has the form:

```
massDeclaration = "M", vocNoun ;
```

Examples:

```
M hydraulic fluid
M 20-gauge wire
M rope
```

Note: Declaring a vocabulary noun to be a mass noun only affects the formal interpretation of quantity expressions where the noun is the term for the substance/things being quantified. A mass noun can be used to refer to individual things, but not to quantities of them. For example, if "rope" is declared to be a mass noun, one can refer to "10 meters of rope," and "a rope," but not to "10 ropes."

Noun synonyms

A declaration of a synonym for a `vocNoun` occurs within the terminology entry, and has the form:

```
nounSynonym = "S", vocNoun ;
```

Noun definitions

The terminology entry for a vocabulary noun should contain an informal definition or a formal definition. It may contain more than one of each.

An informal definition declaration has the form:

```
definition = "D", text ;
```

The `text` is recorded and assumed to have meaning in some natural language.

A formal definition in the RECON language has the form:

```
nounFormulation = "R", ( nounPhrase | unionPhrase ) ;
unionPhrase = vocNoun, { "or" vocNoun } ;
```

When the `nounFormulation` is a `nounPhrase`, the new vocabulary noun is defined to designate the category/kind of thing described by the `nounPhrase`, usually a more general base classifier with one or more delimiting qualifiers. When the `nounFormulation` is a

unionPhrase, the new vocabulary noun is defined to be any thing that is an instance of any of the categories denoted by the vocNouns in the list.

Example: The definition of "shop foreman" might be:

```
R a person who manages a fabrication shop
```

where "person" and "fabrication shop" are vocabulary nouns and "manages" is a vocabulary verb. The term "person" itself may have only an informal definition.

Example: The definition of "shop item" might be:

```
R workpiece or tool
```

Note: a special case of the unionPhrase is the declaration that the term means the same thing as a vocNoun that was declared in a separate entry. The formal definition consists solely of the separately defined term. For example:

```
T work-in-progress
R workpiece
```

Vocabulary Verbs

A *vocabulary verb* (vocVerb) is a term that describes relationships, situations, or actions that involve things in different roles. A vocabulary verb (a part of a verb declaration) has the form:

```
vocVerb =   word, { word } ;
```

The term for a vocabulary verb should somewhere contain an English verb, but it can involve any number of words before or after the actual verb. The additional words can be adverbs or fixed adverbial phrases of any kind.

Example: "etch," "apply pressure to," "is stable at," "is between," "does windows" are valid vocVerbs.

Except for articles, every word of the vocVerb must appear verbatim and in sequence in each usage of the vocabulary verb. Multiple forms of the same word will be recognized as the same word (see General Recognition Rules).

An article (a, an, the) appearing in a vocVerb declaration is treated as an "optional article" in matching a use of the vocVerb in a VerbPhrase. For example, the vocabulary verb declared as "enters into a contract with" will be recognized in "XYZCo enters into contract with the IUEW," and in "George entered into the contract with XYZCo in 2004."

Special verb forms: be, have, do

There are a number of special cases of vocabulary verbs that have forms beginning with "is" (isForms). (Note: these productions are not used, they appear here simply to define the special case of vocVerb.)

```
isForm =    isVerb, { word } ;
isVerb =    "be" | "is" | "are" ;
```

The verb is alone is treated as stating identity. The vocabulary verb is a followed by a nounPhrase is treated as stating that the subject is an instance of the concept described by the nounPhrase. Actual uses of "is" followed by a participle are treated as forms of the verb that

has that participle. In a user-defined `isForm`, "is" must always be followed by a word that is not a participle of any verb used in a declared term. The "is" may be followed by an article, as long as there is some additional word following the article. Some special cases are defined for properties and adjectives below.

Similarly, there are special cases of verb forms that begin with "has."

```
hasForm =   hasVerb, { word } ;
hasVerb =   "have" | "has" ;
```

The verb has alone is treated as denoting ownership, and has special interpretation when followed by property phrases (see Vocabulary Properties). Auxiliary forms consisting of "has" followed by a perfect participle are treated as forms of the verb that has that participle. In a user-defined `hasForm` "has" must always be followed by a word that is not a perfect participle of a verb used in a declared term.

Finally, the verb do is ignored in parsing a verb phrase; it is treated as a meaningless auxiliary. For example, in "if the package does fail the test, …," the `verbPhrase` "does fail" will match "fail." In a user-defined `doForm` "do" must always be followed by an additional word or words. The group of words in the `doForm`, including "do," will be matched as a term.

```
doForm =    doVerb word { word } ;
doVerb =    "do" | "does" ;
```

Verb Syntax

The terminological entry for a vocabulary verb begins with a verb declaration. The entry may also contain alternative verb syntaxes and formal or informal definitions.

```
verbEntry = verbDeclaration,
         { alternativeSyntax | definition | verbFormulation } ;
```

The declaration for a vocabulary verb has a more complex structure than the verb term itself. It has the form:

```
verbDeclaration = "V", verbSyntax ;
```

A `verbSyntax` is a pattern for the use of the vocabulary verb in a sentence. The `verbSyntax` structure is the pattern that is actually recognized as a use of the vocabulary verb in a RECON sentence, and matching the `vocVerb` per se is only a part of that recognition process.

```
verbSyntax =   subjectType, vocVerb, [ objectType ],
         { particle, [ partType ] } ;
subjectType = roleType ;
objectType =  roleType | "()" ;
partType =    roleType ;
particle = word, { word } ;
roleType = "(", [ "+" ], typeName, [ ":", roleName ], ")" ;
typeName = vocNoun ;
roleName = word, { word } ;
```

The subjectType, objectType, and partType elements refer to things that play roles in uses of the vocabulary verb.

The verbSyntax will match a usage in which each of the subjectType, objectType and any partTypes is a valid nounPhrase, and in which the verbPhrase matches the vocVerb as described above. If particle elements are present in the declaration, they must also be matched, and the particle itself must match verbatim. If multiple particle elements are present in the verbSyntax declaration, the ordering of the particle structures will be irrelevant in recognizing usages.

Examples:

```
V (person: operator) operates (machine)
V (machine) consumes electric power
V (machine) requires (time) to perform (process) on (material)
```

Every particle structure other than the final particle in the verbSyntax is required to have a partType, that is, to include a role. The final particle is not required to have a partType when it represents a required part of the vocabulary verb that is separated from the nominal vocVerb in actual usage.

Example: `(party: customer) receives (shipment) on consignment`
The particle "on consignment" is a required part of the intended verb that happens to follow the direct object in actual usage.

In qualified noun phrases (see Qualifiers), a particle element can precede the subject and verb elements of the verbSyntax.

Example: The vocabulary verb that is declared by:
`(person:supervisor) assigns (task) to (person:subordinate)`
will be recognized in: "The mechanic to whom the foreman assigns the inspection …"

The special case in which the objectType is empty parentheses ("`()`") is used only when the vocabulary verb has no direct object, and thus no actual objectType role, but does have separable particles.

Example: `(thing) is used () for (purpose) in (situation)`
will be recognized in: "The purpose for which the spike is used in setup is …," and in "… is a situation in which the insertion tool can be used for extracting the pin." But
`(thing) is used for (purpose) in (situation),`
where the "()" does not separate "for" from "is used", will only be recognized in the latter sentence, because this declaration makes "is used for" the vocVerb, and it does not appear in the former sentence.

Particles can be keywords, like and and for. If the word is matched as a particle, it will not be interpreted as a keyword, as in the above example.

Particles appearing in a verbSyntax are effectively a part of the term for the vocabulary verb. The same English verb with different particle elements can be a different vocabulary verb with a different meaning.

Example: `(tool) is used () on (machine) for (time period)`
will not be confused with the vocabulary verb `is used` ... in the previous example, because a

`verbPhrase` containing the `on` particle will not match the `verbSyntax` containing the `in` particle, and vice versa.

Examples:

"All machining tasks require certification" is an instance of the (unary) vocabulary verb *requires certification*, with the form "(operation) *requires certification*."

"Mr. Berg is certified to perform high-speed turning operations" is an instance of the vocabulary verb *is certified to perform*, with the form: "(person) *is certified to perform* (operation)."

"Mr. Smith is authorized to perform high-precision machining on tool steel" is an instance of "(person) *is authorized to perform* (operation) *on* (material)." In this case, "on" is a `particle` and "material" is a `partType`.

"Mr. Berg supervises Mr. Smith" is an instance of the vocabulary verb *supervises*, with the form: "(person: supervisor) *supervises* (person: subordinate)," where the words "supervisor" and "subordinate" are role names that distinguish the two "persons" involved.

"The city in which the ZZZ factory is located is Pittsburgh" is an instance of the vocabulary verb *is*, i.e., Pittsburgh is the same as "the city in which the ZZZ factory is located." Further, that `nounPhrase` involves a constraint on the vocabulary noun "city" using the vocabulary verb "is located in." "is located in" has the form "(thing) *is located* () *in* (city)." The "()" indicates that the "in (city)" part of the verb phrase can be separated from the "is located" part of the verb phrase in a sentence, as it is in this example. If the empty `objectType` were omitted in the declaration, it would read: "(thing) *is located in* (city)," and the given statement would have to be rewritten as: "The city that the ZZZ factory is located in is Pittsburgh," in order for "is located in" to be matched verbatim.

"No operator shall perform an operation for which the operator is not certified." This involves two vocabulary verb forms: "(person) performs (operation)" and "(person) is certified () for (operation)." "shall perform" is a mandatory form of the vocabulary verb "performs" and "is not certified" is a negative form of the vocabulary verb "is certified."

Verb Roles

Each `roleType` in a `verbSyntax` defines a role in the meaning of the verb – the specific behavior of a single thing participating in a situation that is described using the verb.

```
roleType = "(", [ "+" ], typeName, [ ":", roleName ], ")" ;
typeName = vocNoun ;
roleName = word, { word } ;
```

Every verb has at least one role – the subject of the verb – and thus a corresponding `roleType`. The number of roles a verb has is called its *arity* (a jargon term, derived from the terms unary, binary, ternary, etc., for specific numbers of roles).

The `typeName` is a `vocNoun` that identifies the *range* of the role, i.e., the kind of thing that can play that role. The plus symbol (+) before the `typeName` means that the role has a *meta-range*, which is discussed below.

Every role has a name. If no `roleName` is provided, the `typeName` is also used as the name for the role. The `roleName` can be added to a `roleType` to provide a name for the role that is different from the `typeName`.. A `roleName` need not be a dictionary word. The name for the role is used to refer to the role in other declarations within the terminological entry, but nowhere else. The role name will never appear as such in an actual use of the vocabulary verb. The `typeName`, however, as the `vocNoun` that is a term for a kind of thing, may.

In a formal or informal definition of the vocabulary verb, the role name refers to the thing that plays the role in a given instance of the situation described by the vocabulary verb. For example, in

```
(party: supplier) supplies (goods) to (party: customer)
```

a formal definition will refer to "supplier" and "customer," to distinguish the two "parties" involved. In this example, "goods" is a `typeName` that is also used as a role name. In the definition of this vocabulary verb, "goods" refers to the thing that plays this role, not to the general category that the `vocNoun` "goods" refers to.

A `roleName` may be used again with a different meaning in a different terminological entry. In the above example, the term "supplier" may be separately defined as a vocabulary noun or property, but this use of "supplier" is unrelated to any such declaration. This term "supplier" is only the symbol for the subject of "supplies" and is recognized only within declarations for that vocabulary verb.

Meta-Roles

A *meta-role* is a role that is played by a category of things, rather than by a thing or group of things. Such roles commonly appear in verbs of change and verbs that describe ordering, buying or needing. In such cases, there is no individual thing that plays the role. The noun phrase that is used in the role cannot be a direct reference to an instance, and it doesn't necessarily refer to any current or specific individuals of the kind the noun phrase nominally refers to. Instead, it refers to things that may be or will be of that kind.

Example: The customer orders 10 widgets. The customer did not point out 10 specific widgets on the shelf. The customer ordered some collection of 10 things that are widgets, some or all of which may not exist at the time the order is placed. So the `verbSyntax` is:

```
(party: customer) orders (+product)
```

The + indicates that `product` is a meta-role. The customer orders some unspecified, possibly future, instances of the category "product." Alternatively, one might define this verb as:

```
(party: customer) orders (product type)
```

Example: The supplier and customer agree to change the delivery date. If the existing delivery date is "30 June 2010," the parties did not agree to change the day that is "30 June 2010" into some other day. Rather, what they agreed to change is the instance of the category "date" that is playing the role "delivery date." They are changing which day is the delivery date. So the `verbSyntax` is written:

```
(party) changes (+thing)
```

What is changed is a kind of thing, in this case the kind of thing that is the delivery date. Some other date will become the new delivery date.

Use of a meta-role is semantically similar to references to queries as instances (see Queries as instances), but the meta-role is used when the verb always refers to the kind of thing rather than the thing.

Alternative verb forms

A terminological entry may include alternate usage syntaxes for the same verb. An alternate `usageSyntax` is an alternative pattern for the use of the vocabulary verb in a sentence, and it may involve an entirely different term for the vocabulary verb. For example, an alternate form of a verb in the active voice may be a form in the passive voice. (The version 1 RECON tooling does not automatically recognize passive forms.)

The alternate form declaration has the form:

```
alternativeSyntax = "F", usageSyntax ;

usageSyntax =  subjectRole, vocVerb, [ objectRole ],
         { particle, [ partRole ] } ;

subjectRole = roleUsage ;

objectRole =  roleUsage | "()" ;

particle =    word, { word } ;

partRole =    roleUsage ;

roleUsage =   "(", roleName, ")" ;

roleName =    word, { word } ;
```

The alternative syntax declaration declares the `vocVerb` with its `particle`s, if any, to be a new term for the vocabulary verb. The `roleUsages` refer to the roles declared in the primary verb declaration. The `roleName` in the `roleUsage` refers to the role whose name matches it (regardless of whether the role name was taken from a `roleName` or a `typeName`). The declared role names can appear in `roleUsage`s anywhere within the `usageSyntax`, as long as the following hold:

- each `roleType` that appears in the `verbSyntax` in the primary verb declaration is referred to by exactly one `roleUsage` in the alternate form, and
- the `roleName` of each `roleUsage` matches one of the role names in the primary verb declaration.

Reusing the role names makes it possible for the interpreter to understand how the two forms represent the same meaning.

Example: the vocabulary verb declared as:

```
V (party: supplier) ships (shipment) to (party: customer)
```

may have the alternate form declared by:

```
F (customer) receives (shipment) from (supplier)
```

"Customer" and "supplier" are `roleName`s in the vocabulary verb declaration, and are used to refer to those roles in the alternative form. Note that "shipment" is the `typeName` in the original `verbSyntax`, but it is used as the name of the role, and therefore as the `roleName` in the alternative form.

Note: The original verb declaration is used to construct the IKL predicate. That predicate is used in the generated IKL formulation of a sentence or definition, regardless of which form of the verb actually appears in the sentence or definition. The alternative form is not declared as an IKL predicate. So:

```
X receives widgets from Y
```

will become IKL:

```
(supplier.ships.shipment.to.customer Y widgets X)
```

Verb Definitions

The terminology entry for a vocabulary verb should contain either an informal definition or a formal definition. It may contain more than one of each.

An informal definition declaration has the form:

```
definition = "D", text ;
```

The `text` is recorded and assumed to have meaning in some natural language.

A formal definition has the following form:

```
verbFormulation = "R", sentence ;
```

The `sentence` specifies the meaning of the vocabulary verb in terms of other vocabulary verbs, where each of the role elements in the `verbSyntax` (`subjectType`, `objectType`, `partType`) is considered to designate a specific thing that plays the role in an occurrence of the vocabulary verb. In the defining `sentence`, the thing that plays a given role in the vocabulary verb is identified by its role name.

Example:

```
V (person) is certified for (operation)
```

may be defined by the sentence:

```
R the person has completed the training that is required for
certification in the operation, and the situation where the person
maintains proficiency in the operation has occurred within the
previous year.
```

In that sentence, the terms “person” and “operation” refer to the individual things that play the roles in a usage of the vocabulary verb – an actual occurrence of certification. The formal definition requires that several other vocabulary verbs have been defined: “(person) completes (training),” (training) is required for certification in (activity),” “(person) maintains proficiency in (operation),” and “(situation) occurs within (time interval).”

Vocabulary Adjectives

A *vocabulary adjective* (`vocAdjective`) is a term that can modify a vocabulary noun term or a vocabulary property term. It is treated as a further descriptive constraint on the things that will satisfy the `nounPhrase` in which it occurs.

A terminological entry for a vocabulary adjective begins with an adjective declaration. The entry may also contain synonyms, alternative verb syntaxes and formal or informal definitions. The declaration for a vocabulary adjective has the form:

```
adjectiveEntry = adjectiveDeclaration,
        { adjectiveSynonym | definition | adjectiveFormulation } ;
adjectiveDeclaration = "A", roleType, "is", vocAdjective ;
```

The form of a vocabulary adjective is:

```
vocAdjective = word, { word } ;
```

The sequence of words of a vocabulary adjective must occur verbatim, in order for the term to be recognized. A `vocAdjective` may contain words that are keywords, but it may not begin with a quantifier keyword.

Examples:

```
A (thing) is early
```

declares the vocabulary adjective "early", to be applicable to arbitrary things.

```
A (coin) is very fine
```

declares the vocabulary adjective "very fine", as it applies to coins.

```
A (fee) is reasonable and customary
```

declares the vocabulary adjective "reasonable and customary", as it applies to fees.

```
A (thing) is late Roman
```

declares the vocabulary adjective "late Roman", to be applicable to arbitrary things.

The verb form that is the structure of the terminological entry will be recognized as a verb form that means the adjective. But the primary purpose is to declare the `vocAdjective` for use in its usual position before nouns.

Example: Given the vocabulary adjectives in the example above, "a very fine late Roman coin" will be recognized as "a coin that is late Roman and that is very fine." Similarly, a reference to "a reasonable and customary charge for the service" will be interpreted as "a charge for the service that is reasonable and customary."

The `vocNoun` that represents the range in the `roleType` limits the use (and interpretation) of the vocabulary adjective to instances in which the `vocAdjective` term modifies a `nounPhrase` that refers to instances of the type designated by the `vocNoun`. This allows the same adjective term to be declared and used for two `vocAdjectives` that are differently defined.

Example: Given the `vocAdjective` declarations:

```
A (coin) is very fine
```

and

```
A (mesh) is very fine
```

with appropriate definitions, a reference to "a very fine coin" will be interpreted as a coin that is in the condition called "very fine," while a reference to "a very fine mesh" will be interpreted as a mesh that has a maximum opening of 200 microns.

Adjective Synonyms

A synonym declaration for a `vocAdjective` has the form:

```
adjectiveSynonym = "S", vocAdjective ;
```

The implied alternative verb form for the synonym is:

```
roleType, "is", vocAdjective
```

where the `roleType` is exactly the same as in the original vocabulary adjective declaration, but the `vocAdjective` is the term in the synonym declaration.

Example: for the "late Roman" example above, the synonym declaration:

```
S 4th-century Roman
```

will be interpreted as declaring "4th-century Roman" as an adjective that modifies "coin" and has the alternative verb form declared by:

```
F (coin) is 4th-century Roman
```

and means the same thing as "late Roman".

As indicated above for vocabulary verbs, a synonym is replaced by the base predicate in the IKL output.

Adjective Definitions

The terminology entry for a vocabulary adjective should contain either an informal definition or a formal definition. It may contain more than one of each.

An informal definition declaration has the form:

```
definition = "D", text ;
```

The `text` is recorded and assumed to have meaning in some natural language.

A formal definition has the following form:

```
adjectiveFormulation = "R", sentence ;
```

The `sentence` characterizes the things that possess the characteristics designated by the adjective, using the role name for the subject role to refer to the thing within the sentence.

Example: `A (mesh) is very fine`
declares the adjective "very fine" that applies to a filter mesh. It may have a formal definition, such as:

```
R the diameter of each opening in the mesh is less than or equal
  to 0.2 mm
```

where the term "mesh" in the definition refers to the role whose name and type are "mesh."

In lieu of a synonym declaration, it is also possible to define an adjective in terms of another. For example, the synonym declaration for "4th century Roman" above could instead be a separate adjective:

```
A (coin) is 4th century Roman
R the coin is late Roman
```

Vocabulary Properties

A *vocabulary property* (`vocProperty`) is a term that refers to an attribute or a property of a given thing. It is used to refer to the things that are related in a specific way to a thing called the *domain* of the property. The form of a vocabulary property is:

```
vocProperty = word, { word } ;
```

The term for a vocabulary property has the form of a vocabulary noun. Each usage of the `vocProperty` term in text must match verbatim, in order to be recognized.

The terminological entry for a vocabulary property begins with a vocabulary property declaration. The entry may also contain synonyms, alternative verb syntaxes and formal or informal definitions. The declaration for a vocabulary property has the form:

```
propertyEntry = propertyDeclaration,
         { propertySynonym | alternativeSyntax
           | definition | propertyFormulation } ;

propertyDeclaration = "P", propertySyntax ;

propertySyntax = propertyType, "is", [ article ], propertyForm ;

propertyForm = vocProperty, "()" propertyLink, domainType ;

domainType = roleType ;

propertyType = roleType ;

propertyLink = particle ;
```

The `vocNoun` that appears in the `domainType` identifies the "domain" of the property – the kind of thing that has the property. In each use of the vocabulary property, the `domainType` element will be replaced by a reference to an instance or instances of the type designated by the `typeName` in the `domainType`.

The vocabulary noun that appears in the `propertyType` identifies the "range" of the property – the kind of thing that the property refers to. Most uses of the property, however, use only the `propertyForm`. The `propertyForm` is used in a `nounPhrase` to refer to instances of the type designated by range of the `propertyType`.

Example: "size of a mesh," declared:

```
P (length) is the size () of (mesh)
```

where "size" is the `vocProperty` – the property term,
"length" is a `vocNoun` that refers to a kind of measurement – what a "size" is,
"mesh" is the domain of the property – the kind of thing that has a "size," and
"of" is the `propertyLink` – a reference to the property "size" will be to "size <u>of</u>" something.
A typical use of the vocabulary property would be: "The size of the filter mesh must be less than or equal to 1 millimeter." "size of the filter mesh" is a reference to a thing that is an instance of "length."

Example: "required training for a certification" is declared:

```
P (training) is the required training () for (certification),
```

where "required training" is the `vocProperty` – the property term,
"training" is a `vocNoun` that refers to the kind of thing the property "required training" is,
"certification" is the kind of thing that has the "required training" attribute/property, and
"for" is the `propertyLink` – a reference to the "required training" property will be to "required training <u>for</u>" something.

Implicit Property Declarations

The `propertySyntax` above can be restated as a `verbSyntax`, of the form:

```
propertyType "is" [ article ] vocProperty "()" propertyLink
         domainType
```

When this form of the property is used, the `propertyType` is the `subjectType` of the verb form, the `propertyLink` is a `particle`, and the `domainType` is the corresponding `partType`. The usage is interpreted as an occurrence of this `verbSyntax` pattern.

Note: When this verb form of the property is used as a verb, the `propertyLink` has all the behaviors described above for verb particles. In particular, the particle structure is separable, and the `propertyLink` word(s) may distinguish two properties that use the same `vocProperty` term. For example, the training of a person is different from the training for an operation.

In addition to the verb form that appears in the property declaration, a second verb form of the property is implicitly declared. It has the form:

```
domainType "has", [ "as", article ], vocProperty, propertyType ;
```

This is a simple verb syntax in which the `domainType` is the `subjectType` and the `propertyType` is the `objectType`. It is just an alternative form of the `verbSyntax` above, in which the domain of the property is the subject.

Example: John *is the owner of* the vehicle that *has VIN number* "2T3YK123456768." In this example, *owner* and *VIN number* are properties of the domain *vehicle*. "John is the owner of the vehicle that …" is an example of the property `verbSyntax` for property *owner* (of vehicle). The qualifier "that has VIN number '2T3YK123456768'" is an example of the alternative verb syntax of *VIN number* (of vehicle).

The most important implicitly declared form, however, is the `propertyForm` that appears in the declaration. The `propertyForm` is what makes a vocabulary property different from a verb – it is the form that is used when the property appears in place of a noun, and denotes things.

Example: When the title to a vehicle is transferred, the owner of the vehicle must sign the record of transfer that is on the back of the title document for the title. In this rule, three properties are used in their `propertyForms`, and none of the verb forms is used.

The "title to a vehicle" is a use of some vocabulary property such as:

```
(title) is the title () to (goods).
```

The "owner of the vehicle" is a use of some vocabulary property such as:

```
(party) is the owner () of (goods)
```

The "title document for the title" is a use of some vocabulary property such as:

```
(document) is the title document () for (title)
```

Each is used in place of a noun, and refers to a specific thing.

Property synonyms

A synonym declaration for a `vocProperty` has the form(s):

```
propertySynonym = "S", vocProperty, "()", propertyLink
         | "S", vocProperty
         | "S", "()", propertyLink ;
```

That is, a synonym declaration for a property can declare a synonym for the property term, a synonym for the link term, or a synonym that is a new combination of property term and link term. If only a new property term (`vocProperty`) is provided, it will be recognized as a term for the property in all its forms. If a new property term is provided together with a `propertyLink` term, it will only be recognized in conjunction with that link term. If only a

propertyLink term is provided, it will be recognized as an alternative propertyLink in conjunction with the property term and any of its synonyms.

Alternative verb syntax for properties

A propertyEntry can also contain alternativeSyntax declarations for the vocabulary verb that is implicitly declared (by treating the propertySyntax as a verbSyntax), as described above. These are alternative verb forms only.

Example:

```
P (party) is the owner () of (goods)
F (party) owns (goods)
```

This effectively declares "owns" to mean the same thing as "is the owner of".

Alternative propertyForms for the property must be declared as synonyms, as described above.

Property definitions

The terminology entry for a vocabulary property should contain either an informal definition, or a formalized definition. It may contain more than one of each.

An informal definition declaration has the form:

```
definition = "D", text ;
```

The text is recorded and assumed to have meaning in some natural language.

The formalized definition has the following form:

```
propertyFormulation = "R", sentence ;
```

The sentence characterizes the relationship between a thing that is in the domain of the property and a thing that is a range value of the property for the thing in the domain. Within the sentence, the roleNames for the domainType and propertyType given in the declaration refer to the domain and range things, respectively. A given thing in the domain may have more than one such property value (the attribute may be multi-valued), but the sentence defines the one-on-one relationship. (If the number of values for the property for a given domain thing is constrained in some way, e.g., to only 1, that must be stated as a rule. See below.)

Example: The vocabulary property "shop foreman for (fabrication shop)" may be declared as

```
(person: foreman) is shop foreman () for (fabrication shop),
```

and defined by the sentence:

"the foreman is the person who manages the fabrication shop."

In this definition, the term *fabrication shop* is the role name for the domainType (fabrication shop) in the declaration (it is the typeName, but no other roleName is given); it refers to whatever shop is the domain of a given use of the property. The term *foreman* is the roleName given to the propertyType in the declaration. This definition requires that the vocabulary verb "(person) manages (organization)," or something similar, have been defined.

Note: Contrast the definition of "shop foreman for fabrication shop" with the definition of the vocabulary noun "shop foreman" under Vocabulary Nouns. "shop foreman" is there defined to be a person who manages some/any fabrication shop – a category of person. "shop foreman for (fabrication shop)" is defined to refer to the person who manages a given fabrication shop, as in

"the shop foreman for the Body Panel Shop." Unlike the vocabulary noun, each use of the property takes an argument – the shop in question.

Note: When two properties are related as synonyms or inverses, it may be preferable to define them as separate properties in separate entries, and then to define the second one formally in terms of the other. For example,

```
P (party) is the sender () of (message)
F (party) sends (message)
P (message) is a message () from (party)
R The party is the sender of the message
```

Vocabulary Names

A vocabulary name (`vocName`) is a term that is defined to refer to a specific thing, a "proper noun." It refers to that thing.

```
vocName = word, { word } ;
```

A vocabulary name can be declared in a terminology entry. It has the form:

```
nameEntry = nameDeclaration,
         { nameSynonym | definition | nameFormulation } ;

nameDeclaration = "N", vocName ;
```

The terminology entry can contain synonyms and definitions for the proper name.

Name Synonyms

A synonym declaration has the form:

```
nameSynonym = "S", vocName ;
```

Name Definitions

An informal definition declaration has the form:

```
definition = "D", text ;
```

The `text` is recorded and assumed to have meaning in some natural language. It describes the thing the proper name refers to.

A formal definition has the form:

```
nameFormulation = "R", nounPhrase ;
```

The `nounPhrase` should describe an individual thing, the unique thing that possesses some set of characteristics, or it may just refer to some specific `instance` (see Instances).

Examples:

```
N John Smith
N AT&T
N Statue of Liberty
```

Note: It is not always, or even usually, necessary to declare vocabulary names. See Proper Names.

Vocabulary Measurement Units

A *vocabulary measurement unit* (`vocUnit`) is a vocabulary name that is interpreted as a simple measurement unit. The base measurement units in the International System of Weights and Measures (SI) are predefined in the RECON vocabulary, along with some commonly used derived units and some commonly used English measurement units (see Appendix A). Domain vocabularies may need to declare additional measurement units.

The form of a vocabulary measurement unit is:

```
vocUnit = word, { word } ;
```

A vocabulary measurement unit behaves exactly like a vocabulary name in all usages except quantifiers. A number that is followed by a `vocUnit` will be recognized as a `quantity`.

The declaration for a vocabulary measurement unit has the form:

```
unitEntry = unitDeclaration,
         { unitSynonym | definition | unitFormulation } ;

unitDeclaration = "U", vocUnit, [ : vocProperty ] ;
```

followed by any synonyms and definitions.

The `vocProperty` is the kind of quantity that the unit measures, e.g., length, mass, time, temperature. If no `vocProperty` is specified in the unit declaration, the general property "quantity" is assumed. The `vocProperty` is used in interpreting a quantifying expression that involves the measurement unit. For example,

```
U meter: length
```

declares the `vocProperty` "length" to be the vocabulary property associated with the measurement unit "meter."Then "`5 meters of rope`" is interpreted as "an amount of rope whose length is 5 meters."

Unit Synonyms

Like a vocabulary name, a vocabulary unit can have synonyms, declared by:

```
unitSynonym = "S", vocUnit ;
```

In most cases the synonyms for a vocabulary unit are symbols for the unit. For example, the SI unit "kilometer" might be declared by:

```
U kilometer : length
S kilometre
S km
```

Unit Definitions

Like vocabulary names, a vocabulary unit may have a text definition and formal definitions. It may contain more than one of each.

An informal definition declaration has the form:

```
definition = "D", text ;
```

The `text` is recorded and assumed to have meaning in some natural language. It describes the characteristic of some natural phenomenon that is the basis for the unit.

The form of a formal definition of a vocabulary unit is:

```
unitFormulation = "R", nounPhrase ;
```

The `nounPhrase` may describe the unit in terms of a reference measurement. (The SI units themselves are defined by reference measurements of specific physical phenomena.) A `quantityValue` is a special case of `nounPhrase` that describes the unit in terms of other measurement units (see Quantity values).

Examples: The "kilometer" declaration under Unit Synonyms above could be followed by a formal definition:

```
R 1000 metres
```

The derived measurement unit "newton" could be formulated as:

```
U newton : force
R 1 kg * m / sec^2
```

Axioms

A terminology entry that contains no definition, or only informal definitions, declares a *primitive term* – a term whose definition is not in a form that can be processed by software. From a formal logic point of view, a primitive term is "undefined." But it is usually possible to characterize the relationship of primitive concepts to other concepts of interest, simply by making statements about the required properties of the things that satisfy the concept. These statements are called *axioms*.

In a RECON vocabulary, axiom declarations have the form:

```
axiom = "." sentence
```

All terms used in an axiom sentence must have been declared previously.

Axioms are not explicitly associated with any other terminology entry or declaration. They are independent terminology entries. But they express required behaviors of things corresponding to the terms used in them.

Example:

```
. Each shipment consists of exactly one product type
. Each shipment is on exactly one schedule
```

Axioms have exactly the same meaning as sentences that are parts of a `model`. This is just a syntactic mechanism for including such sentences in a `vocabulary` document.

Models, Sentences, and Definitions

The purpose of RECON is to enable the statement of definitions, facts and rules in a language that is readable by experts in the industry domain. The vocabulary structures above provide one means of creating the vocabulary for such statements. This section defines the grammar for expressing formal definitions, facts, and rules, using terms defined in a vocabulary.

A `model` is a set of facts and rules about the domain of interest, phrased as a sequence of `sentence`s in the RECON language.

```
model = { sentence | comment | directive } ;
```

A period symbol (.) is considered to be the end of a sentence, unless it is followed by a character that is not a space, tab or end-of-line.

Example:

```
//Shipment rules
$M (model mode)
Each shipment consists of exactly one product type.
Each shipment is on exactly one schedule.
If any container contains part of a shipment, it contains no other
shipment.
```

Note: compare this example with the similar example under Axioms (above).

General Recognition Rules

Every term that is used in a formal definition or a sentence must be declared as part of the vocabulary in use. Where a grammar rule below refers to a vocabulary item – a `vocNoun`, `vocVerb`, `vocProperty`, `vocAdjective`, `voc Name`, `vocUnit`, or `particle` – it is matched by a sequence of `word`s that is declared to be a term of the specified kind.

A term declaration specifies the words that make up the term. Different forms of a word are recognized as the same word and thus the same term.

Nouns occurring anywhere in a term will be properly recognized in either singular or plural form. E.g., The word "party" is recognized in "all parties to the contract," and thus the property "(organization) is party () to (contract)" is recognized as well.

Basic verb forms (infinitive, present singular, present plural) will all be recognized as the same verb. Present and past participles (with auxiliary verbs) are distinguished from basic forms. For example, "(thing) arrives" is recognized in "at least two shipments arrive" and "no shipment arrives."

Articles (the, a, an) occurring anywhere within a declared term will be recognized and considered to be optional in recognizing the term. Further, in recognizing the term, any article appearing in the text will match any article in the term. For example, "(party) reports a problem" will be recognized in "The person who reports the problem …" and in "if the customer reports problems …"

Note: Quantifiers other than articles will not match articles that are part of a term: "if the customer reports no problems …" will not be recognized as an occurrence of "(party) reports a

problem." It involves a word ("no") that does not appear in the sequence of words that makes up the verb term "reports a problem."

Sentences

The nature of sentences is what distinguishes different formal logic forms. In a description logic, all the sentences are either simple facts or relationships among classifications. In a first-order logic, one can write simple sentences, but one can also describe complex relationships among things and kinds of things in sentences that involve arbitrarily complex quantifications, conjunctions, negations, and implications. In IKL, one can make a sentence, or its meaning (a situation), a thing that can play roles in other sentences. Most importantly, in first-order logics, a set of sentences about a thing or a kind of thing may be sufficient to distinguish it from other things or kinds of things, without its having a formal definition. Such a set of sentences is called an *axiomatic characterization*. RECON provides a set of sentence structures and noun phrase structures that is powerful enough to enable axiomatic characterization of things and situations, in addition to the statement of simple facts.

A *declarative sentence* is a statement of fact. A RECON `sentence` is a formal structure for representing declarative sentences. The general form of a `sentence` is a set of *clauses* that are linked by connectives and conditionals. Each clause is based on exactly one verb, and is itself a `simpleForm` sentence.

Simple forms

The simplest form of a sentence is based on a single vocabulary verb. The verb appears in one of its syntax forms, with noun phrases playing all its roles.

```
simpleForm = [ subject ], verbPhrase, [ object ], { partPhrase } ;
subject = nounPhrase ;
object = nounPhrase ;
partPhrase = particle, [ nounPhrase ] ;
```

The `particle` of each `partPhrase` must match a particle defined in the `verbSyntax` form that is used in the `verbPhrase`. In a `partPhrase`, the `nounPhrase` is required unless the `particle` does not have a `partRole` (see Vocabulary verbs).

Except in the special case of omitted subjects (in Compound forms below), the `subject` of a `simpleForm` sentence or clause is always required.

Example: "Every shipment that is shipped by rail must be tracked using the NRTA system." This sentence has a `simpleForm`. "Every shipment that is shipped by rail" is a `nounPhrase` that plays the subject role in the sentence. The verb phrase "must be tracked using" is a `mandatedVerb`.

Note: A "rule" is a sentence in which the primary `verbPhrase` is a `mandatedVerb` or a `prohibitedVerb`.

General sentence forms

A sentence can be more complex than a `simpleForm`. RECON supports compound sentence forms, implication forms, and what it calls domain forms, all of which are described in detail below.

```
sentence = basicForm | implicationForm ;
basicForm = compoundForm | domainForm ;
```

Compound forms

The simplest `compoundForm` is a `simpleForm`. `simpleForms` can be connected by `or` and `and` to form compound sentences.

```
compoundForm = andForm ;
andForm = orForm, { "and", orForm } ;
orForm = simpleForm, { "or", simpleForm } ;
```

Example: "Every shipment must be associated with a purchase order, and the internal routing of the shipment must be the routing that is specified in the routing attachment to the purchase order." This is a compound sentence – an `andForm` – that is made up of two `simpleForm` clauses.

Note: In RECON, s1 and s2 or s3 is interpreted "s1 and (s2 or s3)." The reason for this is that the most common uses of compound sentence structures are in the antecedents to implications. Compound statements of fact are rare (and are often written with compound noun phrases, which are separately supported by RECON). An implication of the form "if s1 or (s2 and s3) then s4" can be rewritten "if s1 then s4. if s2 and s3 then s4." But an implication of the form "if (s1 or s2) and s3 then s4" cannot easily be rewritten. So the RECON grammar is designed to enable the expression of the latter structure of antecedent.

Example: If a shipment contains category 4 materials, and the shipment is late or the shipment is damaged, the shipment must be unloaded using the Special Handling for Defective Category 4 Shipments guidelines. In this example, "the shipment is late or the shipment is damaged" is one `orForm` condition that is "anded" with the "`orForm`" (a `simpleForm`) "a shipment contains category 4 materials" in interpreting the compound antecedent.

Omitted subjects: A special parsing exception for compound sentences allows the `subject` of a later clause to be omitted when it is the same as the subject of the previous clause. This is the only case in which the `subject` of a `simpleForm` can be omitted.

Example: "The workpiece must be positioned against bulkhead A and be fastened as shown in diagram D." The implied subject of "be fastened …" is "the workpiece."

Domain forms

A `domainForm` is a sentence structure that resembles the quantification structure in formal logic languages. These forms are common in engineering specifications, and they are common ways of clarifying rules in other domains.

```
domainForm = forForm | thereForm ;
```

```
forForm = "for", forList, [ "does" ], sentence ;

forList = forQuantifier, domain, { "and", forQuantifier, domain } ;

forQuantifier = quantification | quantity ;

thereForm = "there", thereVerb, thereList,
        [ "such that", sentence ] ;

thereVerb = "is" | "are" | "must", [ "not" ], "be" ;

thereList = [ thereQuantifier ], domain,
        { "and", [ thereQuantifier ], domain } ;

thereQuantifier = "a" | "an" | "some" | "no" | quantity ;

domain = [ adjectiveModifier ], localNoun, [ qualifier ] ;

localNoun = vocNoun | vocNoun, ":", shortName | longNoun ;
```

Each `domainForm` introduces a quantified type – the `domain` – and possibly a local name for the instances, and specifies a sentence that must be true for the quantified instances of the domain type. Within the sentence, those instances are referred to by either a *back reference* to the domain type (see Back references) or the local name (if declared, see Local Names).

Each `domain` of a `domainForm` is essentially a `typeNoun` (see Type Noun phrases), but the quantifiers are limited. Each `domain` can be constrained by adjectives and qualifiers. With respect to quantifiers, the grammar above is not entirely correct: when either a `forQuantifier` or a `thereQuantifier` is a `quantity`, the `quantityTerm` it contains must be an `integer`; it cannot be a `quantityValue`.

Example: "For every shipment that is late, an alert for the shipment must be raised, and the supplier of the shipment must be notified." In this example, each instance in the domain "shipment" is later referred to (twice) by the back reference "the shipment."

Example: "For each machined surface: s1, if there is a surface: s2 that adjoins s1, the angle between s1 and s2 must not be less than 18 degrees." In this example, the domain "machined surface" is given a local name (s1) and each instance in the domain is later referred to (twice) by the local name.

Example: "For each time interval:1 and each time interval:2, time interval:1 is strictly before time interval:2 if and only if there is a time interval:3 such that time interval:1 is before time interval:3 and time interval:2 is after time interval:3." This example is a `domainForm` in which the `forList` has two domains, both of which are of type "time interval." They are distinguished by the local names "time interval:1" and "time interval:2." The `sentence` that is the content of the `forForm` is an `equivalence`, in which the second `basicForm` is a `thereForm`.

Example: "For any standard part, there must be at least 2 authorized suppliers who supply the standard part." This is two nested `domainForm`s. The `sentence` in the `forForm` is a `thereForm`. The `thereQuantifier` of the domain in the `thereForm` is the `quantity` "at least 2."

Implication forms

Many sentences, particularly rules, are conditional in some way. Implication forms are sentences (or clauses) that relate condition clauses, called *antecedents*, to consequences of the condition

being true, called *consequents*. But English, and RECON, allow these to be stated in several different ways.

```
implicationForm = implication | equivalence |
         reverse | converse | inverse ;

implication = "if", antecedent, [ "then" ], consequent,
         [ ( "else" | "otherwise" ), alternative ] ;

reverse = antecedent, "only if", consequent ;

converse = consequent, "if", antecedent ;

equivalence = basicForm, "if and only if", basicForm ;

inverse = consequent, "unless", restriction ;

antecedent = basicForm ;

consequent = basicForm ;

alternative = sentence ;

restriction = basicForm ;
```

The `implication` form is interpreted as in English: when the `antecedent` clause is true, the `consequent` clause is true. If an `alternative` clause is included, it is true whenever the antecedent is false.

Example: "if an overheat is detected, the machine stops automatically and sounds an alarm." The antecedent is the `simpleForm` "an overheat is detected"; the consequent is the `andForm` "the machine stops automatically and (it) sounds an alarm."

A `reverse` form is just a different phrasing of an `implication`. People tend to think of the `consequent` of the `reverse` as the precursor to the `antecedent` situation, but the logical interpretation is exactly the same as for the `implication` form.

Example: "The alarm sounds only if an overheat is detected" is understood as "that an overheat is detected is the precursor situation that enables the alarm to sound." But the sentence does not say that the alarm sounds every time an overheat is detected; it says that when the alarm sounds, you know that an overheat was detected. The alarm implies the overheated state.

The `converse` form assigns the same meaning to the `antecedent` and `consequent` as the implication does, but the *ordering* of the antecedent and consequent are reversed, and therefore, the keyword `if` appears in the middle.

Example: "The alarm sounds if an overheat is detected" means exactly the same thing as: "If an overheat is detected, then the alarm sounds."

The `equivalence` form couples the `converse` form and the `reverse` form to mean that each state implies the other.

Example: "The alarm sounds if and only if an overheat is detected" means "If an overheat is detected, then the alarm sounds; and if the alarm sounds, then an overheat has been detected."

The `inverse` form is a logical implication, but the meaning is that when the `restriction` is false, the `consequent` is true. It is important to recognize that this relationship between the

consequent and the restriction is symmetric, that is, when the consequent is false, the restriction is true, as well.

Example: "The machine operates automatically unless the control switch is set to 'manual'." This is interpreted as: "If the control switch is not set to 'manual,' the machine operates automatically." But it is logically equivalent to: "If the machine is not operating automatically, the control switch is set to 'manual'."

Note: The RECON grammar has the "dangling else" problem. A basicForm can be a domainForm whose scope is an arbitrary sentence. It is thus possible that an implication has a consequent that is a domainForm that contains another implication followed by else and an alternative. It is not clear from the grammar to which implication the alternative is associated. The behavior of the parser is to associate it with the innermost implication, since it will be recognized as a valid syntactic continuation of that implication.

Example: "If the robot is operating in automatic mode, there is a safety interlock such that if anything enters the work volume, the safety interlock will stop the robot, otherwise there is no active safety interlock." It is not clear whether the keyword otherwise in "otherwise there is no active safety interlock" means "if no intruder enters the work volume" or "if the robot is not operating in automatic mode." The parser will associate the otherwise with the nearest if, and interpret it as "if no intruder enters the work volume," which is not what was meant. The solution is to make this two sentences, instead of using otherwise.

Verb Phrases

Verb phrases are the foundations of sentences and clauses. Every sentence has one primary verb phrase, and every subordinate clause has one verb phrase. Each verb phrase that appears in a sentence is based on a vocabulary verb. In a sentence or clause, a vocabulary verb is recognized by matching one of the usage syntaxes that is declared for that term. While the verbPhrase per se is not considered to include any of the particles that appear in the clause, those particles are used to determine which vocabulary verb is intended by the verb term. If there is no match on the particles, the verb term itself will not be recognized.

Four forms of verb phrase are supported by RECON:

- affirmed: the positive sense of the verb is the intent
- negated: the denial of the sense of the verb is the intent
- mandated: the positive sense of the verb is a requirement, not a fact
- prohibited: the denial of the sense of the verb is a requirement, not a fact

```
verbPhrase = affirmedVerb | negatedVerb
         | mandatedVerb | prohibitedVerb ;

affirmedVerb = [ doVerb ], [ "always" ], vocVerb ;

negatedVerb = [ doVerb ], ( "never" | "not" ), vocVerb ;

mandatedVerb = "must", [ "always" , vocVerb ;

prohibitedVerb = "must", ("not" | "never"), vocVerb ;
```

Examples: Every shipping container always has a standard label. (affirmedVerb "has"). Some shipping containers do not arrive on time. (negatedVerb "arrives on time") A shipping container that has no manifest must not be unloaded. (prohibitedVerb "is unloaded")

The keywords not, never, and always do not interfere with vocabulary verb recognition. A special case parsing rule is applied to verbs that begin with the auxiliary verbs has and is. When these auxiliary verbs are followed by always, not, or never – the verbPhrase is effectively rephrased using the auxiliary verb "do."

Example: "xxx is not a shipment for the order and has never contained widgets" is actually parsed as "xxx does not be a shipment for the order and does never have contained widgets." This allows the vocabulary verb "is a shipment for" to be recognized in the text as the vocVerb, and it allows "have contained" to be recognized as the perfective tense of the vocVerb "contains."

Unless the verb do is recognized as part of a vocabulary term, it is ignored. E.g., "(thing) arrives" is recognized in "if the shipment does not arrive," and in "if the shipment does arrive." But "(organization) does business () with (organization)" will cause "do" to be recognized as part of the vocabulary verb "does business" in "ABCco does not do business with Asian firms."

Noun phrases

Noun phrases are the other basic building block of definitions and sentences. Each noun phrase refers to zero or more individual things, or to some amount of a mass noun. Noun phrases fill the roles in verbs, and some definitions are just noun phrases.

Noun phrases are based on nouns or properties that are defined terms, and may include qualifying adjectives and delimiting clauses. The combination of terms, adjectives and qualifiers restricts the collection of things the noun phrase refers to.

```
nounPhrase = simpleNounPhrase | groupPhrase ;

simpleNounPhrase = typeNoun | propertyNoun | instance
         | roleNoun | pronoun | localName ;
```

Type Noun phrases

A typeNoun refers to some things of the kind described by a vocabulary noun (vocNoun). The grammatical localNoun specifies the vocNoun, which can be restricted by modifiers and qualifiers. The localNoun can specify a name for the instance(s) to which the vocNoun refers (see Local Names).

```
typeNoun = [ modifier ], localNoun, [ qualifier ]  ;

localNoun = vocNoun | vocNoun, ":", shortName | longNoun ;
```

Example: "Every piece of equipment that is used by XYZCo and that is not owned by XYZCo is covered by a terms of use agreement." The nounPhrase "every piece of equipment that is used by XYZCo and that is not owned by XYZCo" is a typeNoun that is based on the vocNoun "piece of equipment." The nounPhrase "a terms of use agreement" is also a typeNoun that is based on the vocNoun "terms of use agreement."

Property Noun phrases

A `propertyNoun` refers to things that are related to domain instances by a vocabulary property (`vocProperty`). The `localProperty` specifies the `vocProperty` (see Local names) and may declare a local name for the referents of the usage, as restricted by any modifiers. The domain instances are specified either by the `nounPhrase` in the `propertyPhrase`, or by a `genitive` pronoun. .

```
propertyNoun = [ modifier ], propertyPhrase
         | genitive, [ adjectiveModifier ], localProperty ;

propertyPhrase = localProperty, propertyLink, nounPhrase ;

localProperty = vocProperty | vocProperty, ":", shortName
         | longProperty ;
```

Example: “The ASN message must specify the gross weight of the shipment that it describes.” “the gross weight of the shipment that it describes” is a `propertyNoun` that is based on the `vocProperty` “(mass) is the gross weight () of (thing).” The domain instance is the `typeNoun` “the shipment that it describes.”

```
genitive = "its" | "his" | "her" | "their" ;
```

If the genitive pronoun does not appear in the subject of the clause, the genitive pronoun refers to the thing that is the subject of the clause in which the `propertyNoun` occurs. If the genitive pronoun appears in the subject of the clause as a member of a group phrase, but not the first member, the genitive pronoun refers to the first member of the group phrase. Otherwise, the genitive pronoun refers to the subject of the preceding clause. The number and gender of the pronoun is disregarded.

Note: an occurrence of `"her"` that is not followed by a `propertyNoun` is not a genitive pronoun.

Example: “Every employee who has an operator’s license and his supervisor must attend the meeting.” “his supervisor” is a `propertyNoun` that is based on the `vocProperty` “(person) is the supervisor of (person),” using the genitive pronoun “his” to refer to the domain instance “every employee who has an operator’s license.”

Role Noun phrases

A `roleNoun` can only appear in the context of a RECON formal definition. It refers to the thing that plays the role designated by the `roleName` in a given situation that is described by the vocabulary item that declares the role.

```
roleNoun = [ "the" ], roleName ;
```

See the examples under Verb Definitions.

Note: When a term for a `vocNoun` is used as a role name in the declaration of the vocabulary item, the term is recognized within the definition as a `roleNoun` and not as a `typeNoun`.

Pronouns

RECON supports four kinds of pronouns. Genitive pronouns are discussed under Property Noun phrases above, because they are only used in conjunction with properties. Anaphor pronouns are

discussed under Qualifiers, because they are only used in qualifiers. Quantified pronouns and reference pronouns are discussed in this section, because they are nounPhrases.

```
pronoun = quantifiedPronoun | referencePronoun ;
```

A quantifiedPronoun is an abbreviation for a quantifier and a general category of things.

```
quantifiedPronoun = everyPronoun | somePronoun | anyPronoun
         | noPronoun  ;
everyPronoun = "everything" | "everyone" | "everybody" ;
somePronoun = "something" | "someone" | "somebody" ;
anyPronoun  = "anything" | "anyone" | "anybody" ;
noPronoun = "nothing" | "noone" | "nobody" ;
```

The most general category of things is called *thing*.
Every everyPronoun is treated as "every thing" – a universally quantified typeNoun.
Every somePronoun is treated as "some thing" – an existentially quantified typeNoun.
Every anyPronoun is treated as "any thing" – a quantified typeNoun.
Every nonePronoun is treated as "no thing" – a quantified typeNoun.
(See Quantifiers.)

A referencePronoun is a pronoun that refers to a thing or things that have been previously identified within the statement or definition.

```
referencePronoun = basePronoun | selfPronoun ;
basePronoun = "it" | "he" | "him" | "she" | "her"
         | "they" | "them" ;
selfPronoun = "itself" | "himself" | "herself" | "themselves" ;
```

The rules for determining the referent of a basePronoun are somewhat complicated.
If a basePronoun appears in a qualifier in a nounPhrase that is not the subject of the clause in which it occurs, the basePronoun refers to the subject of the clause that contains the nounPhrase. Otherwise, the basePronoun should only appear as, or in, the subject of a clause, and it refers to the subject of the preceding clause. The number and gender of the pronoun is disregarded.

Example: "If any container contains part of a shipment, it contains no other shipment." In this sentence, the pronoun it refers to *any container*, which is the subject of the clause that precedes the clause containing it.

Example: "Every person who signs out equipment is responsible for the equipment that is assigned to him." In this sentence, the pronoun him refers to "*every person who signs out equipment*," which is the subject of the clause that contains the qualifier "*that is assigned to him*," which, in turn, contains him.

A selfPronoun never appears as the subject of a clause and always refers to the subject of the clause in which it appears.

Example: "No person is permitted to sign an authorization for himself." In this sentence, the pronoun himself refers to "*no person*", which is the subject of the clause.

Example: "No tool may contain a battery that can damage it." Here, the pronoun `it` refers to "*no tool*," as described above. Consider: "No tool may contain a battery that can damage itself." Here, the pronoun `itself` refers to the subject of the clause, i.e., to "*that*," which in turn refers to "*a battery*."

Local Names

A local name is a symbol that is created within a statement or formal definition to refer to the instances of a particular concept that appears in the statement/definition. A local name is typically used in a definition or rule that involves several references to things with a complex description, or to different things of the same kind. The syntax that creates the local name also identifies the things it refers to.

There are two forms of local name declaration: short names and long names.

```
localName = shortName | longName ;

shortName = name ;

longName = longNoun | longProperty ;
```

In addition, a local name refers to referents of a `vocNoun` or of a `VocProperty`.

```
localNoun = vocNoun | vocNoun, ":", shortName | longNoun ;

longNoun = vocNoun, ":", integer ;

localProperty = VocProperty | VocProperty, ":", shortName
         | longProperty ;

longProperty = vocProperty, ":", integer ;
```

The forms:

```
vocNoun, ":", shortName
vocProperty, ":", shortName
```

declare the `shortName` to be a local name within the body of the definition or statement in which the declaration appears. Subsequent occurrences of the `localName` symbol within that statement or definition (only) refer to the instances of the `vocNoun` or `vocProperty` that correspond to the usage that contained the declaration.

Example: "For each machined surface: s1, if there is a surface: s2 that adjoins s1, the angle between s1 and s2 must not be less than 18 degrees." In this sentence, "s1"and "s2" are local names (`shortName`s) that refer to the things that are the referents of "each machined surface" and "a surface that adjoins s1," respectively.

The forms:

```
vocNoun ":" integer
vocProperty ":" integer
```

declare the whole `longName`, inclusive of the vocabulary term, the colon, and the integer, to be a local name within the body of the definition or statement in which the declaration appears.

Statements and definitions are parsed left to right. The leftmost (first) occurrence of a `longName` is recognized as a `localNoun` or `localProperty` – a local name declaration for the `longName`. Any subsequent occurrence of the same `longName` within that definition or

statement (only) is recognized as a reference to the localName. It refers to the instances of the vocNoun or vocProperty that correspond to the usage that contained the declaration.

Example: "For each machined surface:1, if there is a surface:2 that adjoins machined surface:1, the angle between machined surface:1 and surface:2 must not be less than 18 degrees." In this sentence, "machined surface:1" and "surface:2" are local names that refer to the things that are the referents of "each machined surface" and "a surface that adjoins machined surface:1," respectively. This example is just a rewrite of the previous example, using the alternative longName forms.

Note: The longName form was added to the grammar, to accommodate a subscripted form of local name that is used in certain formal specifications.

Modifiers

Modifiers are of two kinds: quantifiers, which describe the number of things of a kind a noun or property refers to, and adjectives, which are vocabulary terms that denote properties in their own right. The modifier in a given typeNoun or propertyNoun phrase consists of one optional specialModifier followed by any number of vocabulary adjectives.

```
modifier = adjectiveModifier
         | specialModifier, [ adjectiveModifier ] ;

adjectiveModifier = vocAdjective { vocAdjective } ;
```

Each vocabulary adjective that appears is interpreted as a separate qualifier on the noun or property, and it is conjoined with any other qualifiers on a typeNoun.

Example: If "tempered steel" and "approved" are vocabulary adjectives, the sentence: "An approved tempered steel shaft must be inserted in the socket." refers to "a shaft that is tempered steel and that is approved." Similarly, "an approved supplier who commits to the delivery date" refers to "a supplier who is approved and who commits to the delivery date."

```
specialModifier = quantifier, [ "other" ]
         | "the", "same" | "another" | "a", "different" ;
```

The most common special modifiers are quantifiers, discussed under Quantifiers below.

The special adjectives same and other refer to the thing meant by a previous use of the same term (a kind of *back reference*, see Back references).

the same indicates that whatever things satisfy the vocNoun or vocProperty term that has the modifier must also be identical to the things that satisfied the previous use of the term.

Example: "If a cutting tool is used in machining the housing, the same cutting tool may be used in machining the flange." "The same cutting tool" refers to "*the cutting tool that is used in machining the housing*" – the previous use of "cutting tool."

other, another, and a different indicate that the thing that satisfies the vocNoun or vocProperty term that has the modifier must not be identical to the thing(s) that satisfied the previous use of the term. Other can itself be quantified, that is, it is possible to speak of "2 other" or "at least one other."

Example: “One cutting tool is used in machining the housing, and another cutting tool is used in machining the flange.” “Another cutting tool” means a cutting tool that is not the same as the “*cutting tool that is used in machining the housing*.”

Example: “For each widget that is used in the xxx assembly, 3 other widgets must be purchased as replacements.” “3 other widgets” is interpreted as “3 widgets, each of which is not the “widget that is used in the xxx assembly.”

Quantifiers

A `quantifier` is an expression that determines how many individuals of a kind, or how much of a substance, are referred to by a `nounPhrase`. There are three kinds of quantifiers: articles, explicit logical quantifications, and quantity expressions.

```
quantifier = article | quantification | quantity ;
```

When no `quantifier` appears in a `simpleNounPhrase`, the implicit quantifier is `any`.

Articles

```
article = "a" | "an" | "the"  ;
```

Each `article` that appears as a `quantifier` in a `nounPhrase` is interpreted as a logical quantification. The articles `a` and `an` refer to at least one instance of the concept they modify; they are always interpreted as existential quantifications. The article `the` refers to *all* instances of the concept it modifies, although the expectation is that there will be exactly one. `the` is always interpreted as a universal quantification.

Quantifications

```
quantification = "every" | "no" |
          ( "any" | "some" | "each" | "all" | "none" ),
            [ "of", [ "the" ] ] ;
```

Each `quantification` is interpreted as a logical quantification:

`every`, `each`, and `all` are *universal* quantifiers – they refer to *all* instances of the concept they modify.

`some` is an *existential* quantifier – it refers to at least one instance of the concept it modifies, perhaps more than one, and it makes no difference which one.

`any` is interpreted as universal (all) when it modifies a `nounPhrase` that plays a role in the primary clause, or when there is a back reference (see Back references) in a later clause to the `nounPhrase` it modifies. `any` is interpreted as existential (some) in all other cases.

`no` and `none of` are interpreted as a *negated existential* quantifier – “there does not exist an instance of” the concept they modify.

Note: No distinction is made between `every` and `all`. If the intent is to reference the extension of the concept as a group, the keyword `together` must be used.

Note: When the keywords `of` `the` are embedded in a quantification expression, they are ignored.

Example: “Every active workstation is assigned to a qualified operator.” “Every active workstation” is a universally quantified reference to “active workstation.” It means all such

workstations in the facility. “a qualified operator” is considered to be an existentially quantified reference to the concept “qualified operator.” It means any one such person.

Example: “None of the parts in the Model 219 housing assembly is steel.” “None of the parts in the Model 219 housing assembly” is a negated existential quantification of the `propertyNoun` “parts in the Model 219 housing assembly.” It means that there is no such part that is made of steel. One could also have written “There are no steel parts in the Model 219 housing assembly.” The keywords `of` `the` in “none of the parts” are ignored. The keyword `the` in “the Model 219 housing assembly” is an `article` that is a `quantifier` and is treated as universal – it means every Model 219 housing assembly, although there is probably only one.

Note: Use of `a`, `some`, `any`. The article `a` (or `an`) and the quantifier `some` are *always* treated as referring to some one unspecified thing. “A person who opens a safe must lock the safe” means “some person who opens a safe must lock it, but other persons who open safes may not be required to.” By comparison, “any person who opens a safe must lock the safe” means “every person who opens a safe must lock the safe.” One should not use `a` when the intent is “any and all.” What is really meant here is: “Every person who opens any safe must lock the safe.” By comparison, “A record of student employment must be filed for any project that employs a student” has exactly the same meaning as “a record of student employment must be filed for any project that employs any student.” The record of student employment must be filed if the project employs at least one student, but “any project that …” means “every project that …,” because the keyword `any` modifies a `nounPhrase` that plays a role in the primary verb of the sentence.

General quantities as quantifiers

```
quantity = ( prebound | postbound ), [ "of", [ "the" ] ] ;

prebound = ( prelower | preupper | "exactly" ), quantityTerm ;

prelower =  "at least" | "no less than" | "no fewer than"
         | "more than" ;

preupper = "at most" | "only" | "less than" | "fewer than"
         | "no more than" ;

postbound = quantityTerm, [ "or more" | "or less" | "or fewer" ] ;

quantityTerm = number | quantityValue ;

quantityValue = number, measurementUnit ;
```

Quantities are really of two kinds: those in which the `quantityValue` is the number `1` or `one`, and all others. The grammatical distinction is purely syntactic.

Every `quantity` in which the `quantityTerm` is `one` or the number `1`, with one exception, represents a logical quantification or constraint:

- `at least` `one`, `one` `or more`, `no less than` `one`, and `no fewer than` `one` are interpreted as existential quantifiers, synonyms for `some`.
- `one` or `exactly` `one` is interpreted as an existential quantification plus a uniqueness constraint – there cannot be a different one.
- `at most` `one`, `only` `one`, `one` `or fewer`, or `one` `or less` are interpreted as a uniqueness constraint, without any associated logical quantification. They only assert that all the things

that are instances of the concept they modify are the same thing. They don't require any such thing to exist.

- `fewer than one` or `less than one`, should anyone use them, are interpreted as negative existential quantifications, synonyms for `no`.
- `more than one` is the exception. It means "two or more" and is interpreted as a general quantity construct, as specified below.

Any other `quantity` construct, whether it has a `measurementUnit` or not, is considered to be a general quantity. A quantity whose `quantityTerm` does not involve a measurement unit is considered to be a count of whatever `nounPhrase` it modifies; the count is treated as an amount of the concept to which the `nounPhrase` refers, treated as a generic substance. That is, there is no distinction in kind between "3 shipments and 3.5 tons of scrap iron." "3 shipments" is "3 count of shipment" – the implicit measurement unit is "count." "3 shipments" is interpreted as "a quantity of shipments whose count equals 3 (count)." "3.5 tons of scrap steel" is interpreted as "a quantity of scrap steel whose mass equals 3.5 tons." When the quantity structure involves `at least` or `at most` (or their equivalents), the verb "equals" is replaced by the appropriate mathematical relationship.

Note: Future editions of RECON may deal with "a quantity of X whose count is …" as "a set of X whose cardinality is …."

To be recognized as a `quantifier`, a `quantityValue` should always be followed by the keyword `of`. Otherwise, it may be interpreted as the `instance` that is just the abstract quantity denoted by the `quantityValue (see` Quantity values`)`.

Example: If more than one modem is connected to the cable, and the distance between modems is not at least 2 m, protocol failures may occur. In this case, "2m" is a `quantityValue`, but "at least 2m" is *not* a `quantifier` – it does not modify "protocol failures." It is interpreted as "a quantity/length that is at least 2 metres." By comparison, in "If more than one modem is connected to the cable, at least 2m of cable must connect the modems," "at least 2m of" will be interpreted as a `quantifier` of the `nounPhrase` "cable," i.e., as "a length of cable that is at least 2m."

Back references

A back reference is a use of a `vocNoun` or `vocProperty` term to refer to the/an instance of that kind of thing that satisfies a use of that term earlier in the sentence or clause.

An occurrence of the article `the` followed by a `vocNoun` with no other modifiers or qualifiers is interpreted as a back reference to the previous instance of that `vocNoun`.

Example: For each shipment that has not arrived by the due date for the shipment, the agent shall send a delinquency notice to the supplier for the shipment. The references to "the shipment" in "the due date for the shipment" and "the supplier for the shipment" are back references to the shipment that "each shipment that has not arrived" refers to.

An occurrence of the article `the` followed by a `vocProperty` term with no other modifiers and no `propertyLink` occurrence is interpreted as a back reference to the previous instance of that `vocProperty`.

Example: When the supplier for any shipment ships the shipment, the supplier must send an Advanced Shipment Notification message to the customer for the shipment. The reference to "the supplier" in "the supplier must send …" is a back reference to the thing that satisfies "the supplier for any shipment." This assumes that the "supplier for a shipment" is a property of the shipment, declared as: `(party) is the supplier () for (shipment).` Similarly, the references to "the shipment" in "ships the shipment" and "customer for the shipment" are back references to the shipment that is "any shipment."

Qualifiers

A *qualifier* is a clause that delimits a term so as to refer to a narrower concept. In the previous sentence, for example, "a clause that delimits a term" includes the qualifier "that delimits a term," and narrows the concept "clause." A qualifier always includes some word that refers to instances of the more general term, such as "that" or "who." Linguistically, words used to refer to a thing that has already been mentioned are said to be *anaphors*.

The anaphor word usually comes first in the qualifier clause, but its role in the qualifying verb depends on the grammatical structure of the clause. The various bound forms below distinguish the role of the anaphor in the qualifying clause: subject, direct object, or object of adverbial (prepositional) phrase.

```
simpleQualifier = boundSubjectForm | boundObjectForm |
        boundParticleForm ;

boundSubjectForm = boundTerm, verbPhrase, [ object ],
        { partPhrase } ;

boundObjectForm = boundTerm, subject, verbPhrase, { partPhrase } ;

boundParticleForm = particle, boundTerm, subject, verbPhrase,
        [ object ], { partPhrase } ;

subject = nounPhrase ;

object = nounPhrase ;

partPhrase = particle, [ nounPhrase ] ;

boundTerm =  anaphor | boundProperty | boundQuantity ;

anaphor = ( "that" | "which" | "who" | "whom" ), [ "together" ] ;

boundProperty = "whose", { adjective }, vocProperty ;

boundQuantity = quantification,
        ( "which" | "whom" | boundProperty ) ;
```

The `particle` of each `partPhrase`, and the particle in the `boundParticleForm`, must match a particle defined in the `verbSyntax` form that is used in the `verbPhrase`. In a `partPhrase`, the `nounPhrase` is required unless the `particle` does not have a `partRole` (see Vocabulary verbs).

There are two kinds of references to the qualified concept: the anaphor pronoun (`that`, `which`, `who`, `whom`) that plays some verb role directly, and the possessive pronoun `whose`, followed by a term for some property of the concept being qualified.

Example: "Any delivery that arrives at the wrong destination …" involves the qualifier "that arrives at the wrong destination," in which the subject of the qualifier clause is the anaphor pronoun `that`, referring to the delivery.

Example: "Any delivery to which no order corresponds …" involves the qualifier "to which no order corresponds" – a `boundParticleForm`. The anaphor pronoun `which` is the object of the preposition "to," which is a particle in the `verbSyntax`:

```
(order) corresponds () to (delivery)
```

Note that the empty parentheses in the syntax declaration make the "to" particle separable, and thus allow the "to which" construct to be recognized.

Example: "Any shipment whose shipping label does not conform to the MH10 standard …" involves the qualifier "whose shipping label does not conform to the MH10 standard." This is a `boundSubject` form. The subject of the qualifier clause is a `boundProperty` that refers to the property "shipping label of (shipment);" the qualified concept is "shipment" in "any shipment whose ..."

The `boundQuantity` supports situations in which the qualified concept refers (potentially) to a set of things, rather than a single thing. The `boundQuantity` determines how many members of the set must satisfy the qualifier clause: all, at least 1, etc.

Example: "Any shipment that comprises one or more packages, at least one of which contains hazardous material …" contains two qualifier clauses. "At least one of which contains hazardous material" qualifies "packages". It is a `boundSubject` form in which the `boundTerm` is a `boundQuantity` – "at least one of which." Since "packages" is plural, it is necessary to specify how many of them must be hazardous for the rest of the rule to apply to the shipment. "A shipment that comprises one or more packages which contain hazardous material" is interpreted to refer to a set of "package that contains hazardous material," and relies on the interpretation of "comprises" to determine what is meant. The `boundQuantity` "at least one of which" specifies that if even one of the set of packages contains hazardous material, the rule applies to the shipment.

Example: "A shipment one or more of whose packages is damaged…" uses the `boundQuantity` to quantify a `boundProperty`. It refers to the instances of the property "package in (shipment)" that must be damaged for the rule to apply to the shipment.

The keyword `together`, following the anaphor word, is somewhat similar to the `boundQuantity`. It means the group of all things to which the anaphor refers play the role in the qualifier clause *as a group* (jointly), rather than each individually.

Example: The surface is bounded by two planes which together are 2.5 cm apart. Here the idea is that it is the group of two planes that are 2.5 cm apart, rather than two instances of "plane that is 2.5cm apart." Without the keyword `together`, the latter (erroneous) interpretation would be made.

The general form of `qualifier` is one or more `simpleQualifiers` connected by `and` and `or`. The `simpleQualifiers` are used together to qualify the concept to which the base `typeNoun` refers.

```
qualifier = complexQualifier, { "or", complexQualifier }  ;
```

```
complexQualifier = simpleQualifier, { "and", simpleQualifier },
        [ condition ] ;
```

The grammar rules specify that, if when ands and ors are mixed in a qualifier expression, the "joint"qualifiers connected by and are treated as the components of the alternatives that are connected by or.

Example: "a delivery that arrives at the correct destination and that is late or that arrives at the wrong destination" is interpreted as "a delivery that both arrives at the correct destination and is late, or a delivery that arrives at the wrong destination."

Note: Each qualifier in a qualifier expression that uses and or or must begin with a boundTerm or a particle followed by a boundTerm. In, "a delivery that arrives at the correct destination and is late," "and is late" will not be recognized as a part of the qualifier. The anaphor that, in "and that is late," must be present.

Conditional Qualifiers

When a concept is qualified by multiple alternative qualifiers (linked by or), it is sometimes the case that each of the alternatives applies under only some conditions. The result is a kind of "decision tree" or "qualifier chain," in which each qualifier has an attached condition. The body of each condition (the antecedent) is a simple statement of fact.

```
condition = complexCondition, { "or", complexCondition }
        | "otherwise" ;

complexCondition = simpleCondition, { "and", simpleCondition } ;

simpleCondition = "if", simpleForm ;
```

Note: simpleForm is defined under Simple forms.

The decision tree interpretation is that any condition that is satisfied causes the qualifier it modifies to be applied in qualifying the instances of the qualified typeNoun or propertyNoun. If no condition is true, then no qualifier is applied, and the typeNoun or propertyNoun refers to all of the instances of the base vocabulary item. For this reason, it is not uncommon to have the condition on the last qualifier in the chain be otherwise, which is interpreted as "if none of the other conditions is satisfied, the qualifier whose condition is otherwise is applied."

Example: A formal definition of February: a calendar month that follows a January and that has as duration a duration that is 29 days, if the calendar month occurs in a leap year, or that is 28 days otherwise. In this example, the clause "that follows a January and that has a duration…" is a complexQualifier that qualifies "calendar month." It is the "and" of two simpleQualifiers. Also, "that is 29 days, if the calendar month occurs in a leap year, or that is 28 days otherwise" is a qualifier chain with two conditional alternatives. It qualifies the "a duration." In this case, otherwise is interpreted: "if the calendar month does not occur in a leap year."

Note: Yes, "has as duration a duration" is ugly, and one would like to support "has a duration." Unfortunately, "duration" is also both a vocNoun (a length of time) and a vocProperty of "time interval" (the amount of time in that interval), and that makes "has a duration" ambiguous. This may be improved in a later edition of RECON.

Group phrases

A group phrase is a compound noun phrase that is used to place multiple things, or a choice of things, in the same role.

```
groupPhrase = bothPhrase | eitherPhrase | neitherPhrase |
         insteadPhrase ;

bothPhrase = "both", simpleNounPhrase, { ",", simpleNounPhrase },
         "and", simpleNounPhrase, [ "together" ] ;
```

A `bothPhrase` that does not end in `together` refers to two or more `nounPhrases` that independently play the role in which the `groupPhrase` appears. It is a short form for the "and" of two or more `simpleForm` clauses.

Example: "Everybody and his dog came to the festival" is interpreted as: Every person came to the festival and the dog of that person came to the festival. Similarly, "the clerk must record the name of both the customer and the representative of the customer" is interpreted as: The clerk must record the name of the customer and the clerk must record the name of the representative of the customer.

When `together` appears at the end of a `bothPhrase`, the phrase refers to the group consisting of all the `simpleNounPhrases` as a single object.

Example: "The suspension assembly consists of both a spring and a shock absorber together." This is interpreted: "The suspension assembly consists of a group whose members are a spring and a shock absorber." By comparison, "The suspension assembly consists of both a spring and a shock absorber." is interpreted as "The suspension assembly consists of a spring and the suspension assembly consists of a shock absorber," which is not what is intended.

Note: "The suspension assembly consists of a spring and a shock absorber" will not parse, because the "and" appears to join clauses (see Compound forms), and there is no verb following "shock absorber." The expected pattern is "The suspension assembly consists of a spring, and a shock absorber *does something*." The keyword `both` is required, in order to avoid confusing the intent of "and".

```
eitherPhrase = "either", simpleNounPhrase,
         { ",", simpleNounPhrase },"or", simpleNounPhrase ;
```

An `eitherPhrase` refers to a choice between two or more `nounPhrases` independently. It is interpreted as an instance of a concept that is defined by the corresponding `unionPhrase`.

Example: "Either the clerk or the clerk's supervisor must sign the form." is interpreted as: A person that is the clerk or that is the clerk's supervisor must sign the form." Similarly, "each customer must send either an acceptance response or a rejection response" is interpreted as: "each customer must send a message that is an acceptance response or that is a rejection response.

Note: Unlike the "and" case, which causes all of the choices to apply, the "or" case must allow the individual choice in each case covered by the "each customer" quantifier. It is not the intent that "either every customer must send an acceptance response or every customer must send a rejection response."

```
neitherPhrase = "neither", simpleNounPhrase,
        "nor", simpleNounPhrase ;
```

A neitherPhrase negates the verb for both nounPhrases independently. It is a short form for the "and" of two simpleForm sentences in which the verb is negated.

Example: "Neither the carrier nor the forwarder is permitted to break the container seal." This is interpreted as: "The carrier is not permitted to break the container seal and the forwarder is not permitted to break the container seal."

```
insteadPhrase = simpleNounPhrase, ( "instead of" | "but not" ),
        simpleNounPhrase ;
```

An insteadPhrase causes the clause that contains it to be interpreted as a compound sentence in which one subclause asserts the verb for the first nounPhrases and the second subclause negates the verb for the second nounPhrase.

Example: "If the shipment fails the customs inspection, the supplier, instead of the forwarder, is responsible for the temporary warehousing costs." This will be interpreted as: "If the shipment fails the customs inspection, the supplier is responsible for the temporary warehousing costs and the forwarder is not responsible for the temporary warehousing costs." Such statements are typically exceptions to more general rules.

Instances

An instance is a term or expression that refers directly to an individual thing. .

```
instance = properName | textString | number | dateTime
        | quantityValue | nominalization | query ;
```

Proper names, text strings, and numbers are explicit terms for individual things. Quantity values and date/time values can be more complex expressions that refer to individual things. Nominalizations and queries are complex expressions that refer to abstract things.

Proper Names

A properName is a use of a vocabulary name as a nounPhrase. It is a term for a specific thing.

```
properName = vocName ;
```

A properName need not always refer to a vocName that has been formally declared. Any sequence of capitalized words that are not declared to be terms will be recognized as a properName.

Example: "Every piece of equipment that is used by XYZCo and that XYZCo does not own is covered by a terms of use agreement." "XYZCo" is a properName.

Example: "John Smith" is recognized as a properName, without declaration, and it will be entered as a vocabulary name (vocName) when it is encountered in a statement or definition. "Statue of Liberty," however, is not automatically recognized as a vocabulary name, because it contains the non-capitalized word "of." Only "Statue" is recognized as the properName. The term "Statue of Liberty" would have to be declared as a vocabulary name:

```
N Statue of Liberty
```

Text strings

A `textString` is a sequence of characters delimited by quotation marks or apostrophes. It refers to itself, that is, to the sequence of characters that appears between the quotation marks or apostrophes.

Example: `"3 rings"` refers to the sequence of characters: 3, space, r, i, n, g, s.

Numbers

A `number` is a lexical construct that refers to a mathematical value. It has the form:

```
number = integer, [ fraction ] ;
integer = [ "-" | "+" ], digit, { digit } ;
fraction = ".", digit, { digit }, [ powerof10 ] ;
powerof10 = "E", integer ;
```

Note: RECON distinguishes numbers that have no fraction as integers, and there are some usages that require integers.

Technically, a `number` is a special case of a quantity value (see below), and some are treated that way.

Date/time identifers

A `dateTime` identifier is a lexical construct that refers to a date or a time of day. It has the form:

```
dateTime = time | date, [ ":", time ] ;
date = month, "/", [ day, "/" ], year
         | day, ".", month, ".", year ;
month = digit, [ digit ] ;
day = digit, [ digit ] ;
year = digit, { digit }  ;
time = hour, ":", minute, [ ":", second ] ;
hour = digit, [ digit ] ;
minute = digit, digit ;
second = digit, digit, [ ".", digit, { digit } ] ;
```

Quantity values

A *quantity value* is a representation of an amount of something. It consists of a number and a measurement unit. Measurement units can be simple SI units or English units, or products of units, or ratios of units or products of units, such as km/hr or kg·m/sec^2.

```
quantityValue = number, measurementUnit ;
measurementUnit = unitProduct, [ ( "/" | "per" ), unitProduct ] ;
unitProduct = dimension, { "*", dimension } ;
```

```
dimension = vocUnit, [ "^", exponent ] ;
exponent = integer ;
```

The general form of a measurement unit is a ratio of products of simple measurement units. Simple measurement units are represented by vocabulary measurement units (vocUnits). The simplest form of a measurement unit is a single vocabulary measurement unit.

A dimension is a simple measurement unit, or a simple measurement unit with an associated exponent, such as m^2 (square metres). The exponent must be an integer. The use of exponents is treated as a shorthand representation for a product or ratio or both. E.g., sec^{-2} is interpreted as 1 / (sec * sec).

Examples: 3 metres, 5.2 kg, 120 km/hr, 32 ft/sec^2.

Example: "The weight of each engine is 300 kg." "300 kg" is a quantityValue, used as an instance.

Note: A quantity value that is followed by of and a term for a substance or a category of thing is a form of quantity, that is, it is treated as a quantifier for the substance or category concept (see Quantifiers). That is, it is not treated as an instance in that case.

Nominalizations

A nominalization is a use of a sentence to refer to an idea or situation that plays a role in some outer sentence. In some cases the nominalized sentence refers to itself as the expression of an idea. In other cases, the nominalized sentence refers to the situation it describes.

```
nominalization = statement | question | situation ;
```

Statements

```
statement = "that", sentence ;
```

In a statement, the sentence (see Sentences) refers to itself, to the idea it expresses.

Example: The message from XYZCo says that the shipment is in Los Angeles. The embedded sentence "the shipment is in Los Angeles" refers to itself. It is the thing that is the content of the message.

Example: The forwarder denies that the shipment was routed to Memphis. The thing that is being denied is the sentence "the shipment was routed to Memphis."

Questions

```
question = ( "whether" | "why" | "how" ), basicForm ;
```

A question is a role of a basicForm (see General sentence forms) with respect to a verb that has the sense of a question. In more formal terms, one may say that the verb means a speech act that adds an interrogative force to the meaning of the basicForm sentence.

Example: The receiving agent must determine whether the shipment matches the order. The agent must determine whether the sentence "the shipment matches the order" is true. In general, the question aspect is carried by the verb, e.g., determine, but the particular nature of the question is carried by the interrogative term whether, as distinct from why, or how.

Verb constructs "says that," "denies that," "determines whether" are all about the sentence as a proposition having a truth value. By comparison, why and how are semantically much more like situations. They are references to situations that relate to the situation that the sentence describes.

Situations

A situation is a *use* of a sentence that refers to the kind of situation or event that is meant by the sentence, or to actual situations of that kind.

```
situation = situationWord, "where", basicForm ;
situationWord = "event" | "situation" | "case" ;
```

All of the situationWords, when followed by where, are treated as introducers of a situation. Unlike the question words, they do not have different meanings. These words are not "reserved;" they can be used as, or in, vocabulary terms.

Example: The customer must pay for the ticket before the voyage departs. This can be formalized as: "The event where the customer pays for the ticket must occur before the event where the voyage departs," assuming "(event) occurs before (event)" is a verb in the vocabulary. It is clear in such a case that the sentence is being used to describe actual event instances.

Example: The rule: "The inspector must prevent passengers from carrying weapons on board the aircraft" may be written: "The inspector must prevent the situation where a passenger carries a weapon on board the aircraft." "A passenger carries a weapon on board the aircraft" is the kind of event that must be prevented. This assumes that "prevent" is defined as

```
(thing: agent) prevents (+situation)
```

Note: The distinction between a statement and a situation is more than syntactic. It is a distinction between the *mention* of the sentence (when it refers to itself, making it, or its meaning, a thing in the domain of discourse) and the *use* of the sentence (when it refers to events or situations that it characterizes – a different thing in the domain of discourse). These distinctions are based on the intent of the verb.

In some cases, whether the verb takes a use role or a mention role may be debatable. The verb "prevents" is particularly tricky in this regard. In the second example above, the suggested syntax for the vocabulary verb "prevents" uses "situation" as a *meta-role* (see Meta-Roles). That is, the thing that plays the role is not a situation but rather a kind of situation. The whole idea is that the situation thing that is prevented doesn't ever exist. The agent prevents the existence of an instance of some kind of situation. Now, every sentence represents a kind of situation. So the example sentence may also be written: "The inspector must prevent that a passenger carries a weapon on board the aircraft." That is, the nominalized meaning – the idea that a passenger carries a weapon on board the aircraft – is a kind of situation. Sowa (Sowa, Conceptual Graphs 2008) treats "prevents" as taking a "theme" argument, which is in our terms the statement, as distinct from the situation.

Note: The occurrences of statements almost always directly follow verb phrases in sentences. So the use of the keyword that to introduce a statement is not ambiguous. It can never be confused with the anaphor pronoun that, which must follow a noun – "the thing that …" References to situations, however, may occur in any position that permits a noun phrase.

Requiring `situation where` avoids ambiguity in those cases where a `situation` construct appears. Restricted English is somewhat unnatural in these cases.

Queries as instances

In general, a term or expression for a concept is used to refer to the specific things that it denotes. "The package that is on Dock 4" refers to a specific package; "Employee who works the second shift" refers to any one of a specific set of people. In statements about specifications, questions, answers and changes, on the other hand, the abstraction "whatever the concept refers to" is the subject of the discourse. For example, in "The manager must specify the employees who work the second shift, the noun phrase "employees who work the second shift" refers to an abstract list of things, but the verb is about putting things on that list, not about the things that are on the list at the time. We call the abstract "list of things" the *intension* of the term – the kind of things that are or will be meant by it – and we treat a reference to the intension as different from a reference to the things themselves. The intension is usually represented in the RECON grammar by a `query`:

```
query = subjectQuery | objectQuery | partQuery ;

subjectQuery = interrogative, verbPhrase, [ object ],
         { partPhrase } ;

objectQuery = interrogative, subject, verbPhrase, { partPhrase } ;

partQuery = particle, interrogative, subject, verbPhrase,
         [ object ], { partPhrase } ;

interrogative = queryTerm, [ adjectiveModifier ],
         ( typeQuery | propertyQuery ) ;

queryTerm =  "what" | "which of"
         | "how much", [ "of" [ "the" ] ]
         | "how many", [ "of" [ "the" ] ] ;

typeQuery = localNoun, [ qualifier ] ;

propertyQuery = propertyPhrase | genitive, localProperty
```

Note: An `interrogative` phrase is just a `typeNoun` phrase or `propertyNoun` phrase in which the `queryTerm` is used instead of a `quantifier`. (See Noun phrases.)

Example: The agent must determine what order authorizes the shipment. This is a simple `subjectQuery`. The object of "determine" is the intension of the concept "the order that authorizes the shipment." The agent does not act on the order that is the referent of "the order that authorizes the shipment" itself; what the agent determines is which order is the one meant. The idea "which order is meant" is the abstraction that is the `query` notion.

Example: The delivery date cannot be changed after the order is placed. This does not appear to involve any interrogative form, and thus does not appear to involve a `query`. If, however, this sentence is formally interpreted as it stands, the interpretation will not be what was intended. Suppose the delivery date on the order is July 25, 2012. The interpretation is that "July 25, 2012" cannot be changed after the order is placed, because "July 25, 2012" is the referent of "delivery date on the order." But the intent is not that the day that is July 25, 2012 cannot be changed into something else. What is meant is "what date the delivery date for the order *is*

cannot be changed after the order is placed." That formulation involves the `objectQuery` "what date the delivery date for the order is."

Example: The message must specify both how much oil is on order and how much oil is in the shipment. This is two simple uses of `subjectQuery`, in which the interrogative is "how much oil" and it means "what the quantity of oil … *is*". The message doesn't do anything to the oil itself.

Example: The agent must inform the shipper how many days in advance of departure the booking must be confirmed. This requires a complex circumlocution. What is being requested is the amount of time between two events. Assuming the underlying verb form is: `(event:1) precedes (event:2) by (duration)`, the formulation must be something like: The agent must inform the shipper by how many days confirmation of the booking must precede the departure of the vessel. So, "how many days" plays the "duration" role and is thus a `partQuery`. On the other hand, this could be written: The agent must tell the shipper the amount of time by which confirmation of the booking must precede the departure of the vessel. This formulation refers to a specific duration ("the amount of time") and does not appear to require a `query`. However, one does not "tell" someone an amount of time; one tells someone a fact, an answer to a query – what the amount of time is.

Example: The agent must tell the customer who made the arrangements. This statement will be misinterpreted to refer to "the customer who made the arrangements," which is not the intended sense. It must be formulated: The agent must tell the customer what person made the arrangements.

Example: The agent must ask the customer to whom the package is to be sent. This statement will also be misinterpreted, and in the same way as the one above. It will be interpreted to refer to "the customer to whom the package is to be sent." It must be formulated: The agent must ask the customer to what person the package is to be sent. Note that in each case, the `query` formulates an explicit question.

Example: The agent must tell the customer where the shipment is. And similarly: The agent must tell the customer when the shipment will arrive. If `where` and `when` are interpreted as "at what place" and "at what time," respectively, then these statements can be parsed. In designing RECON, we elected not to support `where` and `when` as interrogatives. RECON requires the explicit wording: The agent must tell the customer what place the shipment is in. And similarly: The agent must tell the customer at what time the shipment will arrive.

Appendix A: Form of the Grammar

This specification formally defines the grammar for the RECON language using a dialect of the Backus-Naur form (BNF) that is defined in ISO 14977 (International Organization for Standardization 1996). A BNF grammar consists of *terminal symbols* and *non-terminal symbols*.

A *non-terminal symbol* identifies a grammatical construct that can appear in other grammatical constructs and has a rule, called a *production rule*, for how a valid language instance that matches it is formed. The lexical elements defined above (`word`, `number`, `string`) are the only non-terminal symbols that do not have formal production rules. Each non-terminal symbol is represented in the text and in the production rules as a sequence of letters in Courier font. E.g., `sentence`.

A *terminal symbol* is a specific sequence of characters that must appear in a particular place in a given language construct. A terminal symbol is represented in a production rule by that sequence of characters enclosed in straight quotation marks, e.g., `"if"`. In general, terminal symbols are keywords, and they appear in the text of this specification in Courier font with underscores, e.g., `if`.

A *production rule* relates a non-terminal symbol to the rule for forming it. It has the form:

```
non-terminal-symbol = (rule elements)
```

A *rule element* can be:

- a terminal symbol, which means that that sequence of characters must appear at that point, or
- a non-terminal symbol, which means that a construct that matches the production rule for that symbol must appear at that point, or
- a sequence of rule elements, separated by the comma (`,`) character, which means that constructs that match each of those elements must appear in exactly that order, or
- a set of rule elements separated by vertical bar (`|`), and possibly enclosed in parentheses, which means that a construct matching exactly one of those sequences must appear at that point, or
- a sequence of rule elements in square brackets (`[]`), which means that a construct matching that sequence of rule elements may appear at that point, but is not required, or
- a sequence of rule elements in curly braces (`{}`), which means that any number of occurrences of constructs matching that sequence of rule elements may appear at that point, but none is required.

Example:

```
inverse = consequent, "unless", restriction
```

means that the non-terminal symbol `inverse` is matched by: a construct that matches the non-terminal symbol `consequent`, followed by the keyword (terminal symbol) `unless`, followed by a construct that matches the non-terminal symbol `restriction`.

Example:

```
article = "a" | "an" | "the"
```

means that the non-terminal symbol `article` is matched by a construct that is exactly one of the keywords `a`, `an`, or `the`.

Example:

```
typeNoun = [ modifier ], localNoun, [ qualifier ]
```

means that the non-terminal symbol `typeNoun` is matched by a sequence of constructs that may begin with a construct that matches `modifier`, always followed by a construct that matches `localNoun` (even if there is no match for `modifier`), and possibly followed by a construct that matches `qualifier`.

According to this production rule, `typeNoun` is matched by "person." It is also matched by "at least one person who can read," assuming that "at least one" is a valid sequence of constructs for `modifier`, and "who can read" matches `qualifier`.

Example:

```
vocNoun = word, { word }
```

means that the non-terminal symbol `vocNoun` is matched by a construct that matches `word`, optionally followed by any number of additional `word`s.

Example:

```
implication = "if", antecedent, [ "then" ], consequent,
      [ ( "else" | "otherwise" ), alternative ]
```

means that the non-terminal symbol `implication` is matched by a construct beginning with the keyword `if`, followed by a construct matching `antecedent`, optionally followed by the keyword `then`, followed by a construct matching `consequent`, optionally followed by a sequence consisting of one of the keywords `else` and `otherwise` followed by a construct matching `alternative`.

According to this production rule, `implication` is matched by "If a shipment arrives, the shipment is unloaded." It is also matched by: "If a shipment arrives, then it is unloaded, otherwise a delinquency notice is sent."

Appendix B: RECON Directives

As of this writing, the RECON tool supports three directives: import document, set vocabulary mode, set model mode.

```
directive = importDocument | setVocabularyMode | setModelMode
```

Import Document

The `importDocument` directive specifies that a given file/resource is to be processed at this point. It has the form:

```
importDocument = "$I" filename
```

where filename is a URI or local file name. The contents of the named resource is processed in the current mode at this point, and then processing the current document continues in the current mode specified for the current document, even if the imported file changed the mode.

Example: If the tool is processing in vocabulary mode, and it encounters a directive to import a document that begins with a `setModelMode` directive, the tool will process the imported document in model mode, but it will return to vocabulary mode when it returns to processing the original document (following the import directive).

SetVocabularyMode

The `setVocabularyMode` directive specifies that the current resource is to be processed in the "vocabulary mode" from this point on. In the vocabulary mode, each line is expected to be a declaration (or a comment or a directive). Axioms must have the "axiom declaration form": period, space, `sentence`.

```
setVocabularyMode = "$V" text
```

SetModelMode

The `setModelMode` directive specifies that the current resource is to be processed in the "model mode" from this point on. In the model mode, each line is expected to be a sentence (or a comment or a directive). There is no "declaration type" character at the beginning of a line.

```
setModelMode = "$M" text
```

References

Fuchs, N., U. Schwertel, R. Schwitter. *Attempto Controlled English (ACE) Language Manual, Version 3.0.* Zurich: Department of Computer Science, University of Zurich, 1999.

Hart, G., Dolbear, C., Goodwin, J., and Kovacs, K. "Lege Feliciter: Using Structured English to represent a Topographic Hydrology Ontology." *Proceedings of the OWL Experiences and Directions Third International Workshop.* Innsbruck, Austria, 2007.

Hayes, P., and Menzel, C. "IKL Specification Document." July 2006. http://www.ihmc.us/users/phayes/IKL/SPEC/SPEC.html (accessed April 2012).

International Organization for Standardization (ISO). *ISO/IEC 14977 Information technology – Syntactic metalanguage – Extended BNF.* Geneva, 1996.

International Organization for Standardization. *ISO 24707 Information technology – Common Logic (CL) – A framework for a family of logic-based languages.* Geneva: International Standards Organization, 2004.

Niles, I and Pease A. "Towards A Standard Upper Ontology." *Proceedings of Formal Ontology in Information Systems (FOIS).* Ogunquit, Maine, 2001.

Object Management Group (OMG). "Semantics of Business Vocabulary and Rules v1.0." 2010. http://www.omg.org/spec/SBVR/1.0 (accessed April 2012).

Pease, A., and Murray, W. "An English to Logic Translator for Ontology-based Knowledge Representation." *Proceedings of the 2003 IEEE International Conference on Natural Language Processing and Knowledge Engineering.* Beijing. 777-783.

Sowa, John F. "Common Logic Controlled English." February 2004. http://www.jfsowa.com/clce/specs.htm (accessed April 2012).

Sowa, John F. "Conceptual Graphs." In *Handbook of Knowledge Representation*, edited by F. van Harmelen, V. Lifschitz and B. Porter. London: Elsevier, 2008.

The Eclipse Foundation. "Xtext Overview." 2011. http://www.eclipse.org/Xtext/documentation/2_1_0/000-introduction.php (accessed April 2012).

World Wide Web Consortium (W3C). "OWL 2 Web Ontology Language Manchester Syntax." October 2009. http://www.w3.org/TR/owl2-manchester-syntax (accessed April 2012).

—. "OWL Web Ontology Language Reference." February 2004. http://www.w3.org/TR/2004/REC-owl-ref-20040210/ (accessed April 2012).

—. "Rules Interchange Format (RIF) Production Rule Dialect." June 2010. http://www.w3.org/TR/rif-prd (accessed April 2012).