LOOM

Users Guide

Version 1.4

ISX Corporation

9 August 1991

Contents

Chapter 1

Introduction

This guide is an introduction to the LOOM[1] system and includes a brief overview of the theory of knowledge representation on which LOOM is based. LOOM is a high-level programming language and environment intended for use in constructing expert systems and other intelligent application programs. This guide will provide an introduction to LOOM concepts, definitions of key commands, and examples of usage. This guide is intended for readers who are looking for a conceptual grasp of the LOOM system. For example, people who wish to evalutate LOOM may want to start by reading this guide. It gives an overview of all major components of the LOOM system as well as a simple example of a LOOM application. The end of this introduction describes other available documentation on LOOM .

Chapter 2 describes the basic concept of operation for LOOM and provides an overview of the knowledge representation theory underlying LOOM. Chapter 3 completes the introductory material by offering a short example to illustrate a typical LOOM application. Chapter 4 introduces the knowledge representation modelling structures used in LOOM and defines concepts key to understanding and using the system. Chapter 5 continues the discussion of system elements with a description of the reasoning mechanisms provided by LOOM and Chapter 6 introduces the facilities available for building procedural and control elements within LOOM. Chapter 7 discusses the use of knowledge base packages within LOOM , and the final chapter introduces advanced LOOM concepts. Several appendices are provided including a glossary of terms, selected technical papers, and an example LOOM program.

This document is written for a technical audience, well versed in knowledge based system concepts. Readers are expected to have had experience with other knowledge representation systems and techniques. The information provided to users wishing to construct new knowledge based applications with LOOM presumes prior experience with the development of knowledge based systems and their architectural issues.

In addition to this User's Manual, there are a number of other documents included in the LOOM Distribution Package that contain important information. These other documents include:

[1]LOOM: "A frame ...for interlacing ...sets of threads or yarns to form a cloth." Webster's.

- The LOOM Reference Manual — detailed technical information on all aspects of the implemented LOOM system.
- The LOOM Tutorial — guided examples illustrating LOOM programming.
- The LOOM Installation Guide — detailed instructions for installing LOOM on supported hardware platforms and preparing it for use.

Chapter 2

An Overview of LOOM Concepts

2.1 The LOOM System

LOOM is a high-level programming language and environment intended for use in constructing expert systems and other intelligent application programs. The LOOM language targets a programming methodology that places heavy emphasis on the specification of explicit, declarative domain models. The LOOM system is built around a knowledge representation system that provides efficient support for on-line deductive query processing. In its inception, the LOOM system was designed as a self-contained, logic-based knowledge representation (KR) system. From observing early applications based on LOOM it was concluded that this "black box" approach to building a KR system was wrong: for most applications a great deal of effort was expended in developing useful programming interfaces to the KR system. The response to this was to extend the scope of the LOOM languages to incorporate several programming paradigms which are directly connected to the KR system. Now, instead of being a black box to be queried, LOOM represents an environment within which application programs can be written.

The LOOM language provides:

- A precise and expressive declarative model-specification language.
- Powerful deductive support, including both strict and default reasoning, and automatic consistency checking.
- Multiple programming paradigms that interface smoothly with a declarative model specification.
- Knowledge-base facilities: a full first-order query language; multiple knowledge bases; dumping and loading of knowledge base objects (persistent objects).

The present LOOM language is designed to capture the best features among the following programming paradigms: object-oriented programming, data-driven programming, problem solving, and constraint programming. In order to make multiple paradigms work together it

is essential that a common representational framework is used, and that the mechanisms that implement control-flow should complement one another; in particular, they should be *orthogonal* (non-overlapping). The knowledge representation framework in LOOM was derived from the language KL-ONE and forms the common basis for integrating the paradigms.

Unlike expert system shells or commercial knowledge-base tools, LOOM does not include a graphic user interface or interface construction tools. LOOM does not follow the tool-box metaphor which seeks to provide a user with a collection of special purpose modules. Instead, LOOM offers a highly expressive representation system which is tightly integrated to powerful reasoning mechanisms. In this way, LOOM users may make use of any of several common programming paradigms as appropriate to their application while relying on a powerful KR system. Future releases of LOOM may include graphical interfaces to the KR system.

While we expect many users of LOOM to be interested in the system as a basis for the development of practical knowledge-based applications to be quickly fielded, there are a number of additional roles in which LOOM may be of use. Researchers in the field of knowledge representation will find the techniques embedded in LOOM as an interesting example of a self contained, logic-based knowledge representation system that has been extended to incorporate powerful reasoning techniques. Researchers working in the areas of natural language, machine learning, and other areas having extensive KR requirements will find LOOM to be a powerful environment for experimentation.

2.2 Knowledge Representation Theory in LOOM

LOOM's architecture achieves a tight integration between rule-based and frame-based paradigms, by providing the capability to use *terminological reasoning* within the pattern matching and control components of a rule processing system architecture. By terminological reasoning, we mean the ability to represent knowledge about the defining characteristics of concepts, and to use that knowledge to automatically infer and maintain a complete and accurate taxonomic lattice of logical subsumption relations between concepts. LOOM is a descendent of the KL-ONE family of languages that feature efficient automatic classifiers to compute subsumption relationships.

A number of knowledge representation systems are appearing that incorporate a description classifier (also called a *term classifier*) into their deductive reasoning component. LOOM is based on the notion that the ability to define and reason with *descriptions* is basic to the task of knowledge representation. The frame component present in many of todays expert system tools provides a means for introducing terms present in the model of an application domain, and for attaching constraint knowledge to individual terms. In LOOM, a description language provides a principled means to describe the knowledge commonly associated with the frame component of a knowledge representation system.

The inheritance mechanism found in many frame systems is actually a simple form of deductive inference, but a specialized technology for reasoning with descriptions has been developed that extends the class of useful inferences far beyond simple inheritance schemes. LOOM captures this technology in the form of an inference engine, the *classifier*. LOOM

supports both a description language and a rule language, and uses its classifier to help bridge the gap between the description component and the rule component. The additional deductive power provided by the classifier enables LOOM to provide inference capabilities not found in current generation KR tools.

LOOM's modelling language is a hybrid consisting of two sublanguages, a *definition* language and an *assertion* language.[1] The definition language expresses knowledge about unary relations (referred to as concepts within LOOM) and binary and n-ary relations (referred to simply as relations in LOOM). The assertion language is used to specify constraints on concepts and relations, and to assert facts about individuals. If the assertions about an individual I collectively satisfy the definition of a concept C, then I is recognized as an instance of C. Each concept P is associated with a pattern $P(x)$; thus matching an individual to a pattern corresponds to recognizing an instantiation relationship between the individual and the corresponding concept.

The integration of terminological capabilities with rules is intended to address three problems with rule-based systems that critics have identified as hindering system maintenance and limiting the ability to generate high-quality explanations and justification. First, the meaning of the terminology used within rules is often ill-defined. This makes it difficult to determine when rules are, or should be, relevant to some shared abstraction—which, in turn, makes it difficult to find and change abstractions. Second, it is difficult to structure large rule sets. This makes it difficult to decompose the set into smaller, more comprehensible and maintainable subsets. Third, rules fail to explicitly separate different kinds of knowledge; different clauses in the same rule may implicitly serve to represent contexts, affect control, or capture structural knowledge. Because the intent behind them is unclear, it is hard to explain rules and difficult to determine how to correctly add or revise them.

The LOOM language attacks the problems just described from several different angles. LOOM's term definition facility enables users to construct definitions for domain concepts that are both rich and precise. By automatically validating concept definitions and organizing them into an abstraction hierarchy, the LOOM classifier supports the development of large-scale concept taxonomies. Once a deep abstraction hierarchy has been constructed for a domain, it is relatively easy for users to compose rule conditions to match against very specific situations. LOOM makes a sharp distinction between purely deductive rules, and side-effecting, procedural rules (LOOM calls the latter class of rules *production* rules. The LOOM language is designed so that that part of a behavioral specification that determines when to fire a production rule is distinct from a specification that determines how to select among multiple actions/responses to a rule invocation. In summary, LOOM differentiates among several kinds of knowledge that traditional rule-based systems lump together as "rules", partitioning them into distinct classes that reflect the various uses and purposes that knowledge.

[1] This distinction is standard in KL-ONE based KR systems.

Chapter 3

An Example LOOM Program

This chapter is intended to give the reader an overview of the LOOM programming language. The basic features of the language are introduced without a formal specification; their formal specifications are given in later sections. Each LOOM statement is followed by an English annotation. By reading the individual LOOM declarations and comparing each of them with the corresponding English annotations, the reader should rapidly develop the ability to read a LOOM model specification.

The LOOM language draws an explicit boundary between declarative, modelling, knowledge and procedural, behavioral, knowledge. The declarative knowledge in LOOM consists of definitions, implications, facts, and default rules. Procedural knowledge consists of production rules, actions, and methods. The distinction that LOOM draws between modelling declarations and behavior declarations is analogous to the distinction made in procedural languages between type declarations and procedure declarations. However, the model- driven style of programming exemplified by LOOM advocates that a model specification should do more than simply define a type lattice and bind field-like objects to record-like objects. A comprehensive model specification will contain assertions about the logical relationships (constraints) that hold between different conceptual entities.

To clarify these concepts and provide insight into the look and feel of LOOM and a typical LOOM application, the remaining portion of this chapter introduces a short LOOM example constructed from the domain of a robot driven factory. The factory will contain entities such as *robots*, *lathes*, *widgets*, etc. In this section we implement the following scenario: *factory-parts* (*widgets*, *gizmos*, and *doohickeys*) are generated at a location called the *Assembly-Site*. Finished parts are placed in a *box*. When a box becomes *almost-full*, the box is moved to another location called the Warehouse. At the Warehouse, the parts are removed from the box. Doohickies are fragile, and extra care must be taken when moving a box containing one or more doohickies.

3.1 The Modelling Language

The subset of the LOOM language used to specify definitions, declarative rules, and facts is referred to as the *modelling language*.

3.1.1 Definitions: Concepts, Relations, and Intervals

Definitions provide a formal means for introducing the terminology used in a particular application domain. Collectively, the set of definitions in a model specification establish the conceptual components relevant to that domain. A definition binds a symbol to a concept (an abstract class of individuals) or to a relation (an abstract relationship between individuals). Concepts in LOOM are used to represent such declarative statements as "There exists a class of Robot entities" or "A Factory-Robot is a Robot with exactly two arms." The LOOM notion of a concept is similar to the notion of a "class" found in many object-oriented systems. Here are two examples of concepts declarations:

```
(defconcept Robot)
(defconcept Factory-Robot :is (:and Robot (:exactly 2 robot-arm)))
```

The first declaration above declares a new (primitive) class of individuals named `Robot`. The second declaration declares a concept named `Factory-Robot`, comprised of those instances of the concept `Robot` for which there are exactly two fillers of the role `robot-arm`. The following LOOM statements define concepts for our factory domain and introduces the use of the `:constraints` keyword. We start with the taxonomy of concepts that define factory-parts, consisting of widgets, gizmos, and doohickeys:

```
[1]   (defconcept Physical-Object
         :constraints (:and (:exactly 1 weight) (:exactly 1 has-location)))
```

"A Physical-Object has exactly one weight and one location."

```
[2]   (defconcept Fragile-Thing :is-primitive Physical-Object)
```

"A Fragile-Thing is a basic kind of Physical-Object."

```
[3]   (defconcept Factory-Part :is-primitive Physical-Object)
```

"A Factory-Part is a kind of Physical-Object. It is not defined in terms of more basic concepts."

```
[4]   (defconcept Widget :is-primitive Factory-Part
         :constraints (:the weight-in-kilos 3))
```

"A Widget is a kind of Factory-Part. All instances of Widgets weigh of 3 kilos."

```
[5]   (defconcept Gizmo :is-primitive Factory-Part
         :constraints (:the weight-in-kilos 2))
```

"A Gizmo is a kind of Factory-Part. All instances of Gizmos weigh 2 kilos."

```
[6]   (defconcept Doohickey :is-primitive Factory-Part
         :constraints (:the weight-in-kilos 1))
```

"A Doohickey is a kind of Factory-Part. All instances of Doohickeys weigh 1 kilo."

Definitions for terms are expressed using the keywords `:is` and `:is-primitive`. For example, LOOM statement [2] defines the term `Fragile-Thing`. The keyword `:is-primitive` is employed to introduce a new property (e.g. "fragility") into a knowledge base. A simple way to think of primitiveness is that it designates concepts that are not or cannot be fully defined by references to other concepts. The designation of `:is-primitive` has the semantics of declaring that there is important hidden information that separates this concept from others in the knowledge base; this information is left unspecified and is not known to LOOM. In simple frame systems such as CLOS, all concepts are primitive in this sense; there is no mechanism for defining concepts whose instances can be recognized by their description in terms of other concepts. See Section 4.1.2 for an in-depth discussion on primitive concepts. Also, see the LOOM tutorial for more examples and discussion of primitiveness. Declarations that omit the `:is` clause, [1], are taken to define new primitive concepts or relations.

Constraints are implications introduced within a `defconcept` or `defrelation` declaration by one of the keywords `:constraints`, `:domain`, or `:range`. A constraint states a necessary condition that applies to instance matching a definition, e.g., "If W is an instance of Widget, then necessarily W weighs 3 kilos"[4].

To complete the definition of the Factory-Part taxonomy, we must define how the roles `weight-in-kilos` and `has-location` can be filled. Relations define how the roles of a concept instance can be filled. The following interval and relation definitions complete the model for factory parts.

```
[7]   (defconcept Weight-in-Kilos :is
         (:and Number (:through 0 +INFINITY)))
```

"Weight-in-Kilos is a kind of non-negative number."

```
[8]   (defrelation weight-in-kilos
         :domain Physical-Object :range Weight-in-Kilos
         :attributes :single-valued)
```

"A Physical-Object's role weight-in-kilos can be filled with an instance of Weight-in-Kilos. weight-in-kilos is a single valued relation."

```
[9]   (defrelation has-location :domain Physical-Object)
```

"Has-location is a legal role of Physical-Object."

Intervals in LOOM are concepts, and represent a (possibly infinite) sequence of scalar elements. The built-in interval Number represents the sequence of real numbers from minus infinity to plus infinity. [7] defines Weight-in-Kilos as a subset of Number containing only numbers from zero to plus infinity.

3.1.2 Implications and More Complex Definitions

Above, we have defined the concept of a Fragile-Thing [2]. Now we would like to imply that a Doohickey is a Fragile-Thing as well as a Factory-Part.

```
[10]   (implies Doohickey Fragile-Thing)
```

"All instances of Doohickey are Fragile-Things."

We will now complete the definitions of concepts and implications needed for our factory example.

```
[11]   (defconcept Physical-Container
          :is-primitive (:and Physical-Object Generalized-Container))
```

"A Physical-Container is a kind of Physical-Object and a kind of Generalized-Container."

```
[12]   (defrelation has-item
          :is-primitive container--containee
          :domain Physical-Container
          :range Physical-Object
          :attributes :closed-world)
```

"The relation has-item is a kind of container–containee relation."
"A Physical-Container's role has-item is filled by an instance of a Physical-Object."

```
[13]   (defrelation weight-of-items
          :is (:satisfies  (?container ?weight)
                           (sum (weight (has-item ?container)) ?weight))
          :domain Physical-Container
          :range Weight-in-Kilos)
```

"A Physical-Container's role weight-of-items is filled by an instance of Weight-in-Kilos." "The relation weight-of-items is computed as the sum of all the weights of the items in the container."

```
[14]   (defrelation capacity-in-kilos :range Weight-in-Kilos)
```

"The role capacity-in-kilos is filled by an instance of Weight-in-Kilos."

```
[15]   (defconcept Box-of-Parts :is (:and Physical-Container :primitive)
          :constraints (:and (:all has-item Factory-Part)
                             (:exactly 1 capacity-in-kilos))
          :defaults    (:filled-by capacity-in-kilos 5))
```

"A Box-of-Parts is a kind of Physical-Container."
"All fillers of the role has-item are Factory-Parts."
"A Box-of-Parts has exactly one capacity-in-kilos."
"By default, the capacity of a Box-of-Parts is 5 kilos."

```
[16]   (defconcept Almost-Full-Box
          :is (:and  Box-of-Parts
                 (:at-least 3 has-item)))
```

"An Almost-Full-Box is a kind of Box-of-Parts which has at least three items."

Definition [13] shows how the LOOM programming language incorporates pattern matching and computation with declarative knowledge. A symbol preceded with a ? represents a variable. The relation `weight-of-items` [13] uses a `:satisfies` clause to define the composition of the binary relations `has-item`, `weight`, and `sum`.

3.1.3 Open and Closed-World Semantics:

To enable LOOM to compute values for the weight-of-items function applied to specific containers, we have declared that closed-world semantics[1] apply to the relation `has-item` [12]. This localized closed-world assumption implies that those instances explicitly asserted to be items in a container C are in fact all of the items in C, and hence makes it possible for LOOM to compute the function which returns the set of of items in a container. If instead, open-world semantics applied to the `has-item` relation, then it would not be possible in general to determine the entire set of items for any particular container. The default setting for relations in LOOM is to assume an open-world semantics. The concept `Almost-Full-Box` [16] relies on the count of the `has-item` role fillers to determine which instances of `Box-of-Parts` are `Almost-Full-Boxes`.

3.1.4 Facts:

Facts are used to tell LOOM about an instance of a concept. LOOM is designed not only to define models but also to dynamically match object instances in the database against defined models. That is, the LOOM environment provides an active classifier in addition to a modelling language. The LOOM construct `tell` is used to "tell" LOOM about an instance

[1] See Appendix A: A Glossary of LOOM Terms for a definition of closed-world semantics

to be added to a database. The construct `tellm` invokes the matcher as well as adding an instance of a concept to the database.

```
[17]   (tellm  (Box-of-Parts box1)
               (capacity-in-kilos 7))
```

"box1 is an instance of a Box-of-Parts; box1 has a capacity of 7 kilos."

```
[18]   (tellm  (Widget w1)
               (has-item box1 w1))
```

"w1 is an instance of a Widget; w1 is placed in box1."

```
[19]   (tellm  (Doohickey d1)
               (has-item box1 d1))
```

"d1 is an instance of a Widget; d1 is placed in box1."

The first database event [17] is the creation of the instance `box1` having a capacity of 7 kilos. Next, an instance of `Widget` identified as `w1` is created and placed into `box1` by asserting that `box1 has-item w1`. The subsequent code creates a `doohickey` and another `widget`, both are put in `box1`.

The concluding operation of a `tellm` function is to alert the LOOM matcher to the presence of database modifications. At this time, the pattern matcher checks the database to see if new pattern matches can be inferred. After the first three calls to `tellm`, the following facts have been asserted or deduced:

```
(Widget w1)
(has-item box1 w1)
(Doohickey d1)
(Fragile-Thing d1)            ;; deduced fact
(has-item box1 d1)
(weight-of-items box1 4)    ;; deduced fact
```

3.2 The Behavior Language

The behavior language is employed to capture the procedurally-expressed aspects of both the domain and the application program. A behavioral specification consists of declarations of actions, methods, and production rules.

3.2.1 Actions and Methods:

An action declaration introduces a generic operation that can be invoked to accomplish some specific tasks. A LOOM action is implemented by a set of methods, each of which specifies a situation-specific implementation for that action. Each method is associated with a pattern that identifies the situation (set of states) when that particular method may be applied. In the case that an operator is to be applied and more than one method is applicable, LOOM compares the patterns for each of the candidate methods and chooses for execution the method with the most-specific matched pattern. The assumption governing this behavior is that methods which are tied to more specific situations are to be preferred over methods tied to more general situations. This type of control strategy, which represents a generalization of the techniques used for dispatching methods in object-oriented languages, provides LOOM programmers with a very powerful control heuristic.

Continuing with the factory example, we will implicitly define the action `move-object` by defining two `move-object` methods, one which is applicable to any object, the second applicable only to boxes which contain one or more fragile parts.

```
[20]   (defmethod move-object (?object ?to)
          :situation (Physical-Object ?object)
          :response
              ((format t "Move ~S to ~ location ~S~%" ?object ?to)
               (tell (has-location ?object ?to)))
```

Method: move-object.
Used when ?object is a kind of Physical-Object.

```
[21]   (defmethod move-object (?box ?to)
          :situation (:and (Box-of-Parts ?box)
                           (:about ?box (:some has-item Fragile-Thing)))
          :response
             ((format t
                "CAREFULLY Move ~S to ~ location ~S~%" ?box ?to)
              (tell (has-location ?box ?to))
```

Method: move-object.
Used when ?box is a kind of Box-of-Parts that contains a Fragile-Thing.

The `move-object` method in [20] is applicable when the instance bound to the formal parameter ?object is a kind of `Physical-Object`. This method can move any object from one place to another. The `move-object` method defined by [21] responds to the more specific situation when the object to be moved is a box which contains one or more fragile parts. In the case, the object (the box) is moved carefully. Hence, whenever an object to be moved is a box containing a fragile part, the second method [21] will be executed.

We will now define two more actions for our factory example: `remove-full-box` and `unload-box`.

```
[22]   (defmethod remove-full-box (?box)
          :response
             ((perform (move-object ?box (get-instance 'Warehouse)))
              (perform (unload-box ?box)))
```

Method: remove-full-box.

Move a box of parts from the Assembly site to the Warehouse and unload the box.

```
[23]   (defmethod unload-box (?box)
          :situation (Box-of-Parts ?box)
          :response
             ((format t "Unload the contents of the box ~S~%" ?box)
               ;; remove each item in the box "?box" and set its
               ;; location to the box's current location
              (do-retrieve ?part (has-item ?box ?part)
                 (forget (has-item ?box ?part))
                 (tell (has-location ?part (has-location ?box)))))
```

Method: unload-box

Unload a Box-of-Parts. For each item in the box, remove it from the box and assign the box location to the `has-location` role of the part.

The function `get-instance` (such as `(get-instance 'Warehouse)` in [22]) retrieves an instance from the database. In this case, `(get-instance 'Warehouse)` retrieves the instance of a location bound to the symbol `Warehouse`.

3.2.2 Productions:

A LOOM production rule specifies a task to be invoked whenever a particular event occurs.

```
[24]   (defproduction P1
          :when (Almost-Full-Box ?box)
          :schedule (remove-full-box ?box))
```

The production `P1` [24], defines a rule that detects the event that a box becomes almost full using the pattern `(Almost-Full-Box ?box)` (defined by declaration [13]). The effect of the production `P1` is to schedule a task to apply the operator `remove-full-box` to a box in the event that the box becomes almost full.

To show the interaction between asserting facts and the behavioral portion of the factory model we add a third item to `box1`.

```
[25]   (tellm (Widget w2)
              (has-item box1 w2))
```

"w2 is an instance of a Widget, w2 is placed in box1."

This causes the database to now have the following form.

```
(Widget w1)
(has-item box1 w1)
(Doohickey d1)
(Fragile-Thing d1)            ;; deduced fact
(has-item box1 d1)
(Widget w2)
(has-item box1 w2)
(weight-of-items box1 7)    ;; deduced fact
(Almost-Full-Box box1)        ;; deduced fact
```

When the `(Almost-Full-Box box1)` fact is deduced, the production rule `P1` [24] fires due to its `:when` condition `(Almost-Full-Box ?box)`. `P1` then schedules a task to apply the operator `remove-full-box` to the object `box1`. Because `box1` contains a doohickey `d1` that is fragile, the test conditions of both move-object methods are satisfied. Because the second method's pattern is more specific (applies to a more specific concept in the concept taxonomy) than the first method's pattern, the second method will be selected and `box1` will be moved `carefully` to the Warehouse.

Chapter 4

Modelling in LOOM

This Chapter introduces the syntax and semantics of the constructs that declare definitions, implications, and default rules. These constructs are all based upon a syntactic variant of first-order logic, called a *terminological logic*, that facilitates the expression of the most frequently referenced forms of descriptive knowledge.[1]

The Lisp operators `defconcept`, `defrelation`, and `defset` declare the following kinds of objects: concepts, relations, and sets defined as follows:

> **concept:** A concept is the LOOM specific construct that binds a name to a description of a class of entities. Concepts are used similar to frames in other knowledge representation systems.
>
> **relation:** A relation defines a linkage between elements of represented knowledge. A link between two concepts is a binary relation. The LOOM description language for relations provides particular support for defining binary relations. Unary relations are defined as concepts.
>
> **set:** In LOOM, a set is a specialized kind of concept that identifies a collection of constants. The set may be finite or infinite. Examples of constants are the color red or the number 5.

Expressions representing concepts and relations are either primary or compound. Some primary expressions are

`(:at-least 1 has-arm)`	the concept whose individuals all have at least one arm
`(:range Integer)`	the relation whose range fillers have type Integer

[1] Because a different syntax (closer to that of standard first-order logic) is used for asserting and retracting facts, we have placed the presentation of the fact language in its own Chapter, 5.

The logical operators `:and`, `:or`, and `:not`[2] can be employed to synthesize compound expressions. They have the semantics of intersection, union, and relative complement, respectively, e.g., if C_1 and C_2 are concepts, then `(:and` C_1 C_2`)` represents the concepts formed by intersecting C_1 and C_2.

4.1 Definitions

This section opens by describing the role that definitions play in the declaration of a concept or relation. We pay special attention to the semantic distinction between primitive and defined definitions. The remaining subsections present the syntantic constructs used to define concepts, relations, and sets, respectively.

4.1.1 What is a "Definition"

The primary function of a `defconcept` (or `defrelation`) function is to associate a name with a LOOM concept or relation. For example, to associate the name `2-Armed-Robot` with a LOOM concept representing "a Robot with exactly two arms" we might declare

```
(defconcept 2-Armed-Robot
   :is (:and Robot (:exactly 2 has-arm)))
```

In this `defconcept` declaration, the term `(:and Robot (:exactly 2 has-arm))` refers to (defines) a concept, and the keyword `:is` states that the symbol `2-Armed-Robot` denotes that concept. The term introduced by the `:is` keyword represents a necessary and sufficient condition for membership in the concept. Let us assume that closed-world semantics[3] apply to the relation `has-arm`. Then if we make the assertions

```
(tell (Robot Daneel)
      (has-arm Daneel A1)
      (has-arm Daneel A2))
```

then `Daneel` satisfies the condition `(:and Robot (:exactly 2 has-arm))` and hence the system will deduce

```
==>  (2-Armed-Robot Daneel).
```

Conversely, if we assert

```
(tell (2-Armed-Robot Giscard))
```

the system will deduce the necessary condition

```
==>  (:about Giscard (:exactly 2 has-arm))
```

[2]The `:not` operator is currently implemented only for the case of finite sets.

[3]See Appendix A: A Glossary of LOOM Terms for a definition of closed-world semantics.

meaning that "Giscard has two arms" (but the identity of each arm may not be known).

By attaching a necessary and sufficient condition to the symbol `2-Armed-Robot` we are telling the system "what it means to be a `2-Armed-Robot`." For example, if we evaluate the query

```
(retrieve ?x (:about ?x Robot
                     (:at-least 2 has-arm)))
```

which translates as "retrieve all robots which have at least two arms" the system will return a set containing both `Daneel` and `Giscard`. To generate this response, the system must infer for the case of the robot `Giscard` that "a robot with exactly two arms" is necessarily "a robot with at least two arms."

A primary purpose of LOOM's definitional facility is that it allows a programmer to define complex terms, and to make assertions using those terms.[4] It remains the responsibility of the LOOM system (specifically, of its deductive component) to correctly interpret the implications or consequences inherent in each new term definition.

To avoid complicating the semantics, LOOM requires that definitions of concepts and relations not be cyclic. For example, the following definition is self-referential, and therefore illegal:[5]

```
(defconcept Part
   :is-primitive (:and Physical-Object (:all has-part Part)))
```

We can repair this definition by moving the constraint `(:all has-part Part)` out of the definition of `Part`, and into a separate constraint clause:

```
(defconcept Part :is-primitive Physical-Object
   :constraints (:all has-part Part))
```

With respect to LOOM's pattern-matching and retrieval operations, this latter definition exhibits the same behavior as would be the case for the former (circular) definition (if it were legal). This technique of repairing circular definitions by substituting constraints is a general one. The constraint language thus compensates for the inability of the definition language to express circularly-defined concepts.[6]

4.1.2 Primitive and Defined Concepts

It is important to understand the semantic import that accompanies the distinction between *primitive* concepts (ones which either explicitly or implicitly contain the term `:is-primitive` in their definitions) and *defined* concepts (ones which don't). This section first illustrates the

[4] Contrast this with the view mechanisms found in relational database systems, which allow views to participate in retrieval expressions, but place severe limitations on how views can be used in assertional (update) expressions.

[5] The `:satisfies` clause is an exception to this rule, i.e., it can be self-referential.

[6] We have not yet encountered an application for LOOM which would benefit materially by introducing circularly-defined concepts.

difference conceptually, and then presents an example illustrating the difference in behavior associated with the two kinds of concepts.

Many words used to describe real-world objects defy exact description. Words such as "rock", "chair", or "bird" denote object classes such that the most exact thing we can say is "I know one when I see it." Such concepts are sometimes referred to as "natural kinds". In LOOM these are examples of *primitive* concepts. Other concepts, such as "a smooth red rock" or "a 3-legged chair", can be defined with precision, given that the terms referenced in their definitions ("smooth", "red", "rock", "leg", "chair") are independently defined. These two composite concepts are examples of *defined* concepts.[7] Another way of thinking of primitive concepts is that a primitive concept has unrepresented features that characterize an object as an instance of that concept. In this sense, the name of the concept itself carries the semantics necessary to distinguish it from other concepts, even though there may not be defined attributes and features to separate it. LOOM will merge defined concepts that appear to be identical, but primitive concepts are never merged since LOOM realizes there are features important to their definitions that will alway be unknown to the system.

Consider the following two concept definitions, whose semantics differ only in the use of the `:is-primitive` (rather than `:is`) in the second definition:

```
(defconcept 5-Kilo-Part
   :is (:and Part
             (:the weight-in-kilos 5)))
(defconcept Primitive-5-Kilo-Part
   :is-primitive (:and Part
                       (:the weight-in-kilos 5)))
```

An instance B is recognized by the LOOM pattern matcher as a `5-Kilo-Part` if it is a `Part` that weights 5 kilos. However, B is not recognized as a `Primitive-5-Kilo-Part` unless it is directly asserted to be one. More concretely, suppose we make the factual assertions

```
(tell (Part B1)
      (weight-in-kilos B1 5)
      (Primitive-5-Kilo-Part B2))
```

The query `(retrieve ?x (5-Kilo-Part ?x))` will return a set containing both `B1` and `B2`, because for each of the instances LOOM will recognize that their `weight-in-kilos` is 5 and will classify them as instances of `5-Kilo-Parts`. In constrast, the query `(retrieve ?x (Primitive-5-Kilo-Part ?x))` will return a set containing only `B2`, since `B1` was not directly asserted to be a `Primitive-5-Kilo-Part`.

[7] An explicit representation of primitiveness is absent in nearly all of the currently-available knowledge-based programming systems. This is because those systems lack any facility for representing a *defined* concept, i.e., the concepts represented by their frames, classes, sorts, etc. are *all primitive*; hence, attaching the notation *primitive* to a frame, class, etc. would be redundant.

4.2 Concept-Expressions

Below, we present the atomic concept-forming expressions. In these expressions, the symbol k represents a non-negative integer. The symbols R and R_j stand for relation expressions (either a unary or a binary relation). The term *role* is often used to refer to a relation R which appears within a concept expression. Each role R maps an instance of a concept to a set of instances which are called the *fillers* of that role, e.g., the items in a box of parts are considered to be fillers of the role `has-item` with respect to that box. The symbol T below stands for a concept expression.

(:and C_1 ... C_n) — INTERSECTION

Defines a concept representing the intersection of the concepts C_1, ..., C_n.

Example: `(:and Physical-Container Fragile-Thing)`
describes an object that is a physical container and is also fragile.

(:or C_1 ... C_n) — UNION

Defines a concept representing the union of the concepts C_1, ..., C_n.

Example: `(:or Widget Gizmo)`
Describes an object that is either a widget or a gizmo.

Comment1: The performance of the LOOM classifier will degrade if significant numbers of concepts are created that are not themselves primitive, and that do not inherit any primitiveness. A disjunction of two widely dissimilar concepts may create one of these undesirable concepts. Disjunctions of concepts that all specialize a common primitive ancestor do not exhibit this negative property. For example, both of the concepts `Widget` and `Gizmo` specialize the primitive concept `Part`, and hence their union `(:or Widget Gizmo)` also specializes `Part`.

Comment2: The `:or` operator has not been implemented for relation expressions (because doing so is non-trivial, and because users don't seem to miss it).

(:not C) — COMPLEMENT

Defines a concept representing the set of objects that are not instances of C.[8]

Example 1: `(:and Part (:not Gizmo))`
Describes an object that is a part but that is not a gizmo.
Example 2: `(:not Gizmo)`
Describes an object that is not a gizmo.

Comment: It is strongly recommended (for performance reasons) that the `:not` operator only be used to define a *relative complement* (of a concept other than `Thing`). By this we mean that a `:not` expression always be contained within an enclosing conjunction expression. The expression `(:and Part (:not Gizmo))` represents a correct use of `:not`, since it defines the complement of `Gizmo` *relative* to the *complement* `Part`. Explicit introduction of the concept `(:not Gizmo)` is undesirable because it defines `Gizmo` relative to everything, i.e., relative to `Thing`.

[8]Implementation Note: At present the `:not` operator has only been implemented for set concepts.

`(:at-least` k R`)` — Min Restriction
`(:at-most` k R`)` — Max Restriction
`(:exactly` k R`)` — Min-Max Restriction

Defines a concept C such that each instance of C has at-least/at-most/exactly k fillers for the role R.

Example1: `(:and Box-of-Parts (:at-most 0 has-item))`
Describes a box with no items.
Example2:
```
(:and Robot
    (:exactly 4 (:compose has-arm has-finger)))
```
Describes a robot which possesses a total of four fingers.

`(:all` R T`)` — Type Restriction

Defines a concept C such that for each instance of C, all fillers of the role R have type T.

Example: `(:and Robot (:all has-arm (:exactly 2 has-finger)))`
Describes a robot each of whose arms has exactly two fingers.

`(:some` R T`)` — Typed Existential

Defines a concept C such that for each instance of C, R has at least one filler of type T.

Example: `(:and Box-of-Parts (:some has-item Gizmo))`
Describes a box containing at least one gizmo.

`(:the` R T`)` — Singular Type Restriction

Defines a concept C such that for each instance of C, R has exactly one filler, and R's filler has type T.

Example:
```
(:and Lathe
      (:the (:compose has-bit thickness)
            (:through 30 50)))
```
Describes a lathe which has a (single) bit whose thickness is between 30 and 50 (inclusive).

`(:filled-by` R T`)` — Role Filler

Defines a concept C such that for each instance of C, R has a filler whose value is T.

Example:
```
(tellm (:about Joe
          (:filled-by parent Fred Mary)))
```
For the concept Joe, the parent slot is given the values Fred and Mary.

(`:same-as` R_1 R_2) — EQUIVALENCE (SAME FILLERS)

Defines a concept C such that for each instance of C, the sets of fillers of the roles R_1 R_2 are the same.

Example:
```
(:and Robot (:same-as (:compose left-arm location)
                      (:compose right-arm location)))
```
Describes a robot whose left and right arms are (currently) in the same location.

(REL R_1 R_2) — COMPARISON
(REL R_1 *const*)
(REL *const* R_2)

Defines a concept C such that for each instance of C, the j-place or 2-place predicate REL is true when applied to the sets of fillers of the roles R_1 ... R_j (or to a role R_i and a constant *const*). REL must belong to the set of built-in relations {`<`, `>`, `<=`, `>=`, `=`, `/=`, `:subset`}. In the usual case that some or all of the arguments to REL are required to be single values rather than sets, LOOM coerces the corresponding (singleton) sets of role values into non-set-valued values, e.g., the set "{3}" becomes the value "3".

Example1:
```
(:and Box-of-Parts
      (> weight-of-items capacity-in-kilos))
```
Describes a box of parts whose items collectively weigh more than its capacity.

Example2:
```
(:and Box-of-Parts
      (>= weight-of-items 10))
```
Describes a box of parts whose items collectively weigh more than ten kilos of its capacity.

(`:satisfies` (*arg*) *query*) — SATISFIES QUERIES

Defines a concept C of all instances such that the query *query* is satisfied when the variable *arg* is bound to that instance. *arg* is a symbol beginning with

the character '?', and *query* is an open sentence (expressed in the LOOM query language) having *arg* as its only free variable.

Example:
```
(:and Box-of-Parts
      (:satisfies ?box
          (:for-some (?m1 ?m2)
              (:and (has-item ?box ?m1)
                    (has-item ?box ?m2)
                    (/= (weight-in-kilos ?m1)
                        (weight-in-kilos ?m2))))))
```
Describes a box of parts containing two items of unequal weight.

(:predicate (*arg*) *body*) — Compiled Lisp Predicate

Defines a concept C of all instances such that *body*, a list of Lisp forms, returns a non-nil value when evaluated with the variable *arg* bound to that instance.

Example:
```
(:and Number
      (:predicate ?number
          (integerp ?number)))
```
Describes a number that is an integer ("integerp" is a built-in Lisp function).

Comment: The `:predicate` operator provides a formal way to introduce Lisp-defined predicates into a LOOM knowledge base. Concepts whose definitions include a `:predicate` operator are treated as pseudo-primitive concepts because LOOM cannot reason about the contents of the bodies of these definitions.

(:changed *R*) — Changed Values

Defines a concept C such that for each instance of C, the set of fillers of R has changed during the last match cycle.[9]

Example:
```
(:and Box
      (:changed location))
```
Describes a box whose location has just changed.

Comment: Because LOOM does not yet include a representation of time, the `:changed` construct operates by assuming that each call to the LOOM matcher represents a time increment. The `:changed` feature represents a first small step in the direction of a temporal calculus for LOOM.

[9] Implementation Note: The `:changed` construct has not been implemented yet.

4.3 Relation-Expressions

This section illustrates the operators available for defining new relations. Each relation expression corresponds either to a binary predicate whose first argument is called its *domain*, and whose second argument is called its *range*, or to an n-ary predicate whose first n-1 arguments are called its *domains*, and whose last argument is called its *range*.

(`:and` R_1 ... R_n) — INTERSECTION

Defines a n-ary relation representing the intersection of the relations R_1, ..., R_n.

Example:
```
(defrelation length-in-meters :is-primitive
   (:and unit-of-length metric-measurement))
```
Defines a `length` relation that is both a kind of `unit-of-length` and a kind of `metric-measurement`.

(`:range` C) — RANGE RESTRICTION

Defines a binary relation R such that for each instance <d,r> of R, r has type C.

Example:
```
(defrelation has-arm
   :is (:and has-part (:range Robot-Arm)))
```
Defines a `has-part` relation whose range fillers are robot arms.

Comment: Given the assertions `(Robot-Arm A1)` and `(has-part Shakey A1)`, the above definition allows LOOM to infer `(has-arm Shakey A1)`. The `:range` of a relation is also checked when using the `set-value` function.

(`:domain` C) — DOMAIN RESTRICTION

Defines a binary relation R such that for each instance <d,r> of R, d has type C.

Comment: Domain restrictions occur very seldom in user's relation definitions. However, they do appear occasionally in a system-defined definition, e.g., if an inverse of the relation `has-part` called `part-of` were defined, then upon defining the range-restricted relation `has-arm` (as in the previous example) LOOM would (on its own) construct an inverse of the relation `has-arm` whose definition would be:

```
(defrelation part-of-1
   :is (:and part-of (:domain Robot-Arm))).
```

(:inverse *RI*) — INVERSE RELATION

Defines a binary relation R such that for each instance $<d,r>$ of *RI*, $<r,d>$ is an instance of R.

Example: `(defrelation item-of :is (:inverse has-item))`
Defines the inverse of the relation `has-item`

Comment: Suppose we want to define a concept `Boxed-Part` to indicate a factory part which is an item of some box. This is easy if we have the inverse to `has-item` available:

```
(defconcept Boxed-Part
   :is (:and Factory-Part (:some item-of Box-of-Parts))).
```

(:compose R_1 ... R_n) — COMPOSITION

Defines a binary relation R as the composition of the relations R_j, i.e., $<i_1,i_n>$ is an instance of R if and only if there exists i_2 ... i_{n-1} such that for each j between 1 and n-1, $<i_j,i_{j+1}>$ satisfies R_j.

Example:
```
(defrelation has-in-hand
   :is (:compose has-arm has-hand grasp))
```
Defines a composition of the relations `has-arm`, `has-hand`, and `grasp`

(:satisfies (*args*) *query*) — SATISFIES QUERY

Defines an n-ary relation R whose arity equals the number of arguments listed in *args*. *query* is an open sentence (expressed in the LOOM query language) having free variables corresponding to the members of the list *args*. A tuple $<i_1,i_2,...,i_k>$ is an instance of R if and only if *query* is satisfied when its free variables are bound (in the order listed in *args*) to the instances i_1, i_2, ... ,i_k.

Example:
```
(defrelation has-item*
   :is (:satisfies (?x ?z)
          (:or (has-item ?x ?z)
               (:for-some ?y
                  (:and (has-item ?x ?y)
                        (has-item* ?y ?z))))))
```
Defines a relation named `has-item*` representing the transitive closure of the relation `has-item`.

Comment: The restriction that concept or relation definitions cannot be self-referential is lifted inside of a `:satisfies` clause.

(`:predicate` (*args*) *body*) — COMPILED LISP PREDICATE

Defines an n-ary relation R whose arity equals the number of arguments listed in *args*. LOOM generates a compiled Lisp function with arguments *args* and body *body* which is used to recognize instances of R.

Example:
```
(defrelation >=
   :is (:predicate (?v1 ?v2) (>= ?v1 ?v2))
   :domain Number :range Number)
```
Defines a relation representing the Lisp predicate "`>=`".

Comment: The `:predicate` operator provides a formal way to introduce Lisp-defined predicates into a LOOM knowledge base. Relations whose definitions include a `:predicate` operator are treated as pseudo-primitive relations because LOOM cannot reason about the contents of the bodies of these definitions.

(`:function` (*args*) *body*) — COMPILED LISP FUNCTION

Defines an n-ary relation R whose arity equals one more than the number of arguments listed in *args*. LOOM generates a compiled Lisp function with arguments *args* and body *body* which is used to generate instances of the range of R, when applied to a set of instances from the domains of R.

Example:
```
(defrelation +
   :is (:function (?v1 ?v2) (+ ?v1 ?v2))
   :domains (Number Number) :range Number)
```
Defines a relation representing the Lisp function "`+`".

Comment: The `:function` operator provides a formal way to introduce Lisp-defined functions into a LOOM knowledge base. Relations whose definitions include a `:function` operator are treated as pseudo-primitive relations because LOOM cannot reason about the contents of the bodies of these definitions.

4.4 Sets and Intervals

Sets and intervals are viewed as a special type of concept. In LOOM , a set is considered to be a group of constants and an interval is a set with an ordering of "`<`" over the members of the set. Named sets are declared with the use of the `defset` operator. Also, LOOM provides two set constructors `:one-of` to build a set from a collection of symbolic literals, and `:the-ordered-set` to build a set of symbolic literals which contain an ordering from the first element to the last element. Intervals are declared using `defconcept` and use the constructor `:through`. The examples below illustrate the use of declared sets, constructed sets, and intervals.

```
(defset Tire-Pressure :is (:the-ordered-set FLAT LOW NORMAL))
(defrelation tire-pressure :is-primitive
   :domain Wheel  :range Tire-Pressure
   :attributes :single-valued)
```

"For Wheels, the role tire-pressure must have exactly one filler taken from the interval Tire-Pressure and is either LOW, FLAT, or NORMAL."

```
(defconcept Low-Wheel
            :is (:and Wheel (:the tire-pressure (:through FLAT LOW))))
(defconcept Flat-Wheel
            :is (:and Wheel (:the tire-pressure (:one-of FLAT))))
```

"Low-Wheels are Wheels whose tire pressure is in the interval from FLAT to LOW. Flat-Wheels are Wheels whose tire pressure equals FLAT."

```
(defconcept Weight-in-Kilos :is
   (:and Number (:through 0 +INFINITY)))
```

"Weight-in-Kilos is a kind of non-negative number found in the interval from zero to infinity."

```
(defset Door-Status :is (:and (:one-of OPEN CLOSED) Symbol))
(defrelation door-status :is
   :domain Car    :range Door-Status
   :attributes :single-valued)
(defconcept Car-with-Open-Door
   :is (:and Car (:the door-status (:one-of OPEN))))
```

"Door-Status names the set of symbols {OPEN CLOSED }. A Car-With-Open-Door is a Car whose door-status role is filled by the symbol {OPEN}."

4.5 Implications/Constraint Rules

A constraint declaration attaches a rule of implication to a concept or relation. Constraints are attached to concepts using the keyword `:constraints`, and to relations using the keywords `:domain` and `:range`. The `:constraints` keyword introduces a 'necessary' qualifier to concept membership, that is, all instances of a concept must satisfy each of the constraint(s) associated with that concept. As an example, in the concept

```
(defconcept Object
   :constraints (:and (:exactly 1 weight) (:exactly 1 location)))
```

we see that an `Object` has two constraints: it can have precisely one weight and precisely one location. An alternative means for declaring constraint rules is supplied via the `implies` operator. The above definition of the concept Object could be restated using the `implies` operator as follows:

```
(defconcept Object)
(implies Object
   (:and (:exactly 1 weight) (:exactly 1 location)))
```

The `:domain` keyword restricts the type of objects that can appear as the first element of a relation. As an example of its use, the LOOM construct

```
(defrelation location :domain Object)
```

says that "objects may have locations". Note that no constraint has been placed on fillers of the role location.

To restrict the type of the fillers of a role, the keyword `:range` is used. For example, in order to define a numeric range ("range checking"), one can define an interval (using `defconcept`) that bounds the desired numeric range, and then use that defined interval as an argument to the `:range` keyword. To say that the weight of an object must be non-negative, we can first define an interval `Non-Negative-Number` as follows:

```
(defconcept Non-Negative-Number :is
   (:and Number (:through 0 +INFINITY)))
```

and then use the interval `Non-Negative-Number` in defining the concept `Weight`:

```
(defrelation Weight :domain Object :range Non-Negative-Number)
```

This LOOM construct is read as "An `Object` may have filler(s) for the role `Weight`. Fillers of a `Weight` role must be instances of `Non-Negative-Number`."

4.6 Default Rules

LOOM provides two syntactic forms for representing default rules, the keyword `:defaults` within a `defconcept` declaration and default rules using the macro `default` (no preceding colon). Defaults cannot be attached to relations. Essentially both the `:defaults` keyword and the `default` macro form an associative rule between a concept and a default expression. The default expression can be a concept, an `:at-least`, `:at-most`, `:exactly`, `:all`, or `:filled-by` clause, or a conjunction of these clauses or concepts.

`:defaults` (*default* – – – *expression*) — DEFAULT EXPRESSION

Keyword on `defconcept`. For all instances of the concept C, the $default - expr$ is held to be TRUE unless explicitly changed.

Example:
```
(defconcept 2-Armed-Robot
   :is (:and Robot (:exactly 2 has-arm)))
   :defaults (:filled-by Weight 500)
```
Declares that 2-Armed-Robots are assumed to have a weight of 500.

Comment: The `:defaults` keyword clause specifices the consequent to a rule whose antecedent is the concept being defined.

`(default` *antecedent consequent*`)` — DEFAULT RULE

A macro declaring that for concept expressions matching *antecedent*, the *consequent* is held to be true unless it is inconsistent with the current state of the knowledge base.

Example:
```
(default Part (:not Fragile-Thing))
```
Declares that if p is a part, and p is not known to be fragile, it is assumed that p is not fragile.

Comment: A default rule expresses an conditional implication that applies only when its consequent is consistent with current state of the knowledge base.

4.7 Attributes

In many frame and object based representation systems, the term *attribute* is used to denote a role or relation describing the concept; the attributes define the class. In LOOM however, the word *attributes* is reserved for describing special properties associated with concepts and relations. The `:attributes` keyword is used with `defrelation` or `defconcept` and can take on any or all of the following values:

`:backward-chaining` Indicates that instances do not classify under a concept during the normal match cycle. Instead, instantiation relationships are deduced in response to queries from either LOOM or the user. This often makes the matching process more efficient.

`:cache-computation` Used with relations whose definition includes the `:function` operator. Causes computed values to be cached as ordinary role fillers, so that subsequent accesses do not require recomputation. The cached values are not truth-maintained, but they may be explicitly retracted.

`:clip-roles` Used with (`:single-value`) relations, this attribute causes clipping to be performed, in the case that clipping has been turned off globally. By default, clipping is on.

`:closed-world` Over the range of this concept or relation, all propositions not proven TRUE are taken to be FALSE. The default condition for LOOM is `:open-world` semantics; see Section 3.1.3.

`:monotonic` Indicates that any instance filler that is asserted to belong to this concept/relation will always belong to it. Accelerates performance of truth maintenance[10].

`:multiple-valued` Used with relations whose definition includes the `:function` operator; declares that more than one value will be returned when the relation is used as a role. By default `:function` relations are `:single-valued` and all other relations are `multiple-valued`. Used with relations only.

`:predicate-specializes-parent` Used in concepts and relations whose definition includes a `:predicate` or `:satisfies` operator. Indicates the concept or relation does not need to inherit predicates from its parents since its own predicates imply those of its parents.

`:single-valued` This relation describes a role that can have only one value as a filler at a time. This declaration has three effects: 1) *clipping*, which causes each new role filler to supersede the previous role value; 2) in conjunction with `:same-as`, causes fillers of equivalent single-valued roles to be merged; and 3) making it easy to "close" a role. Used with relations only.

`:sequence` R's role fillers form a sequence[11]. Used with relations only.

`:symmetric` Indicates that a relation is its own inverse: $R(x, y) = R(y, x)$. Used with relations only.

Additionally, LOOM concepts and relations may have the following LOOM-assigned attributes:

`:incoherent` Indicates that the current definition of the concept or relation includes an inconsistency. For example a concept that specializes two disjoint concepts would be marked `:incoherent`.

`:system-defined` Indicates the concept or relation was created by LOOM (it has a definition, but no user-supplied name).

`:undefined` Used to mark a concept or relation which has been referenced but not yet defined (it has a name, but no definition).

[10] This attribute is only partially implemented.

[11] Not yet implemented

4.8 Partitions and Coverings

By default, the extensions of any pair of concepts in LOOM are assumed to overlap unless those concepts are explicitly or implicitly declared to be disjoint. Two concepts are considered to be disjoint (from each other) if no single individual can simultaneously be an instance of both concepts. The library function `disjoint-concepts-p` provides a means for testing the disjointness of pairs of concepts. During a match cycle, LOOM automatically applies a test to see if an instance belongs to two or more disjoint concepts. Whenever it is proved that an instance belongs to a pair of disjoint concepts, the instance is labelled `:incoherent`. A concept that specializes two or more disjoint concepts is also marked `:incoherent`.

Earlier versions of LOOM (Version 1.3.1 and earlier) used the keyword `:disjoint-covering` to divide a concept into a group of disjoint subconcepts. Beginning with LOOM 1.4, this has been replaced by two constructs: `:partitions` and `:exhaustive-partitions`, which designate partial and complete disjoint coverings respectively.

Often it is helpful to partition some or all instances of a concept into exclusive groups. The set of subordinate concepts then forms a *partition* of the domain of the concept. If the partitions that are known suffice to cover all possible instances of the concept, this is called an *exhaustive partition*. However, we frequently would like to use partitions to classify objects as being an instance of only one subconcept at a time without having to explicitly represent all of the possible partitions. For example:

```
(defconcept Part
   ...
   :partitions $PartTypes$)
(defconcept Widget
   :is (:and Part
   ... )
   :in-partition $PartType$)
(defconcept Gizmo
   :is (:and Part
   ... )
   :in-partition $PartType$)
(defconcept Doohickey
   :is (:and Part
   ... )
   :in-partition $PartType$)
```

declares that any given instance of `Part`, `P1` may be either a `Widget`, a `Gizmo`, or a `Doohickey`. `P1` could be some other type of part not yet known, but it cannot be both a `Gizmo` *and* a `Doohickey`. The use of the `:partition/:in-partition` syntax enables the user to add additional partitions of the parent concept at any time. In contrast, the statements:

```
(defconcept Part :is-primitive
   ...
   :exhaustive-partitions $PartTypes$)
(defconcept Widget
   :is-primitive (:and Part
   ... )
   :in-partition $PartType$)
(defconcept Gizmo
   :is-primitive (:and Part
   ... )
   :in-partition $PartType$)
(defconcept Doohickey
   :is-primitive (:and Part
   ... )
   :in-partition $PartType$)
```

declares that *all* instances of `Part` will be exactly one of the set `[Widget Gizmo Doohickey]`.

Section 5.4 shows how open and closed world semantics can interact with the behavior of LOOM with disjoint-coverings. The examples below show how exhaustive and non-exhaustive partitions may be combined in to enable reasoning over complex knowledge structures.

```
(defrelation has-feature :domain animal)
(defconcept Animal
"The Animal Kingdom"
     :is-primitive Thing
     :exhaustive-partition (Protozoan Metazoan))
(defconcept Metazoan
 "The Subkingdom Metazoa:  all animals with more than one cell"
    :is-primitive (:and Animal Many-celled-organism)
     :partitions $Metazoan-Phylii$)
(defconcept Protozoan
 "The Subkingdom Protozoa:  the single-celled animals"
    :is-primitive (:and Animal Single-celled-organism)
     :partitions $Protozoan-Phylii$)
```

All instances of animals will either be single-celled protozoa, or multi-celled animals called metazoa. By using the exhaustive partition form, we enable LOOM to infer membership in the set that is not eliminated by known facts. If we assert that Fred does not have more that one cell, we know that he is a protozoa.

When we do not know all of the disjoint partitions of a class, or when we do not wish to represent all of them, the use of `:partitions` allows us to make inferences of exclusion. In the example below, we have chosen to represent only four of the seven vertabrate classes (leaving out Agnatha, Chondricthyes, and Amphibia).

```
(defconcept Chordate :is-primitive Metazoan
:in-partition $Metazoan-Phylii$)
(defconcept Vertebrate
"The Subphylum Vertebrata:  animals with a
 backbone and skeletal structure"
:is-primitive (:and Chordate (:the has-feature Backbone))
    :partitions $Vertebrate-Classes$)
(defconcept Fish
"The Class of Fishes: cold-blooded, marine
 vertebrates that have gills"
:is-primitive (:and Vertebrate
(:the has-feature gills)
(:the has-feature cold-blooded)
(:the has-feature marine))
:in-partition $Vertebrate-Classes$)
(defconcept Mammal
"The Class of Mammals:  warm-blooded vertebrates
 whose females bear live offspring and give milk"
:is-primitive (:and Vertebrate
(:the has-feature female-gives-milk)
(:the has-feature lungs)
(:the has-feature warm-blooded)
(:the has-feature live-young))
:in-partition $Vertebrate-Classes$)
(defconcept Bird
"The Class of Birds:  warm-blooded
 vertebrates, covered with feathers,
 breathe with lungs and have wings"
:is-primitive (:and Vertebrate
(:the has-feature wings)
(:the has-feature lungs)
(:the has-feature warm-blooded)
(:the has-feature feathered))
:in-partition $Vertebrate-Classes$)
(defconcept Reptile
"The Class of Reptiles:  cold-blooded
 vertebrates that have lungs and scales"
:is-primitive (:and Vertebrate
(:the has-feature lungs)
(:the has-feature cold-blooded)
(:the has-feature scalely-skin))
:in-partition $Vertebrate-Classes$)
```

```
(defconcept warm-blooded :is-primitive Thing)
(defconcept female-gives-milk :is-primitive Thing)
(defconcept live-young :is-primitive Thing)
(defconcept wings :is-primitive Thing)
(defconcept marine :is-primitive Thing)
(defconcept gills :is-primitive Thing)
(defconcept Backbone :is-primitive Thing)
(defconcept Many-celled-organism :is-primitive Thing)
(defconcept Single-celled-organism :is-primitive Thing)
(defconcept feathered :is-primitive Thing)
(defconcept lungs :is-primitive Thing)
(defconcept cold-blooded :is-primitive Thing)
(defconcept scalely-skin :is-primitive Thing)
```

Chapter 5

Reasoning about LOOM Instances

LOOM's assertion, retraction, and query languages are all object-based. We use the term *instance* to refer to a knowledge base object that represents a single individual. The assertion language is limited to asserting facts about instances, and asserting relationships that hold between instances. Because the full expressive power of the concept definition language is available to the assertional language, facts about individuals can be arbitrarily complex. Knowledge base retrievals return sets of instances, or sets of lists of instances.

5.1 Assertion: "tell"

The only kinds of facts that can be entered into a LOOM knowledge base are either (1) assertions that an instance belongs to a particular concept, (2) assertions that some binary relationship holds between two knowledge base instances, or (3) assertions of the equivalence of two single-valued (functional) role chains.[1] Thus, there are some kinds of facts that one cannot state in LOOM's assertion language. For example, one cannot state something like "either widget `w1` is in the box, or widget `w2` is fragile."

Assertions are made using the Lisp function `tell` [2]. Inside of a `tell`, the proposition `(Robot Robby)` states that "`Robby` is an instance of the concept `Robot`," while the proposition `(has-arm Robby a1)` states that "the relation `has-arm` is satisfied by the instances `Robby` and `a1`" (this last proposition can be rephrased as "the role `has-arm` attached to the instance `Robby` contains the filler `a1`"). These two propositions are asserted by the form

```
(tell (Robot Robby)
      (has-arm Robby a1))
```

Within a `tell`, a non-keyword symbol that is not prefixed with one of the characters '?' or '$' is assumed to refer either to a specific concept, relation, or instance. The LOOM parser determines by context which of the three types of objects is being referenced. In the example just above, `Robot` refers to a concept, `has-arm` refers to a relation, and `Robby` and `a1` refer

[1] See the discussion of `:same-as` below for an explanation of role chain equivalence.

[2] The form `tellm` invokes the matcher following the assertion.

to an instances. Concepts and relations referenced by name within a `tell` must be defined at the time the `tell` is compiled.[3] LOOM evaluates each instance identifier by first looking for an already existing instance with that identifier. If none is found, LOOM creates a new instance, and attaches the identifier to it.

Within a `tell` clause, the keyword `:about` can be employed to escape into the terminological syntax used to define concepts. For example, the statement

```
(tell (:about Robby
              (:exactly 2 has-arm)
              (:all has-arm Double-Jointed)))
```

asserts that "the role `has-arm` on the instance `Robby` has exactly two fillers, and both (all) fillers are instances of the concept `Double-Jointed`." The `:about` construct also permits an abbreviated syntax for asserting grounded atomic formulae, e.g., the meaning of the statement

```
(tell (:about Robby
              Robot
              (has-arm a1)))
```

is identical to that in our first example of `tell`. In case we wish to assert several fillers for the same role, the keyword `:filled-by` can be used within an `:about` clause to further abbreviate the syntax. The following three `tell` statements are equivalent:

```
(tell (has-arm Robby a1) (has-arm Robby a2))
(tell (:about Robby (has-arm a1) (has-arm a2)))
(tell (:about Robby (:filled-by has-arm a1 a2)))
```

When role fillers are asserted into a previously empty role using the `:filled-by` syntax, LOOM guarantees that the the *order* of role fillers will be preserved. For example, one can depend on the following behavior:

```
(forget (has-arm Robby *))
(retrieve ?a (has-arm Robby ?a))                  ==> ()
(tell (:about Robby (:filled-by has-arm a3 a4)))
(retrieve ?a (has-arm[2] Robby ?a))               ==> (|I|A4)
```

The fillers of a role can be equated with the value of a Lisp expression using the keyword `:filled-by-list` inside of an `:about` clause. For example, the statements

```
(setq ?arms (list (get-instance 'a3) (get-instance 'a4)))
(tell (:about Robby (:filled-by-list ?arms)))
```

represents yet another way to provide `Robby` with two arms. The LOOM function `set-value` is a simpler way to accomplish the same effect:

```
(set-value 'Robby 'has-arm '(a3 a4))
```

[3]Implementation Note: This restriction *may* be lifted in a future LOOM release.

The `:filled-by-list` operator is destructive—it has the side-effect of retracting any previously-asserted fillers from a role. This behavior is somewhat analogous to the "clipping" behavior described in the next section.

The `tell` syntax permits a formula (a composition of one or more functions, applied to instances) to be substituted in place of a reference to an instance. For example, if the weight of instance `Robby` is known, we can assert that "the weight of `Nomad` is one greater than the weight of `Robby`" by stating

```
(tell (weight-in-kilos Nomad (+ (weight-in-kilos Robby) 1)))
```

In this example, the references to `+` and `weight-in-kilos` in second argument to the predicate `weight-in-kilos` references refer to binary relations that are interpreted as one-argument functions. The syntax for these formulae is the same as that for the arguments to predicates inside of a query statement (see section 5.3).

The keyword `:same-as` can be used inside of a tell to assert that two instances are equivalent, or that the fillers of two "role chains" are equivalent. If `Robby` and `robot-2` identify two knowledge base instances, then the statement

```
(tell (:same-as Robby robot-2))
```

causes them to be merged into a single instance that combines all assertions made about each of them individually. All references (pointers) to `Robby` and `robot-2` from other instances are relinked to point to the merged instance. A *role chain* refers to a binary relation, defined as the composition of other binary relations, that is attached to some concept or instance. We will call a role chain *single-valued* if all of the relations in its composition are single-valued (i.e., functional). The `:same-as` construct can be used to assert that two single-valued role chains are equivalent.[4] To assert that "the thing that `Robby` is looking at is the same as the thing that it is grasping in its right hand" we state

```
(tell (:same-as (look Robby)
                (grasp (has-right-hand Robby))))
```

If the fillers of both of these role chains are known, (i.e., are represented by actual knowledge base instances), then they are merged into a single instance. Otherwise, this `:same-as` assertion acts as a constraint that can be used during deduction to infer facts about one or both of the unknown fillers.[5]

It is important to note that within an `:about` clause the syntax for `:same-as` mirrors that for concept expressions, and hence differs from the syntax just described. The following assertion is semantically equivalent to the one in the previous example.

```
(tell (:about Robby (:same-as look
                              (:compose has-right-hand grasp))))
```

[4]Implementation Note: At present, the role chains referenced within a `:same-as` must emanate from the same instance. In a future release we plan to remove this restriction.

[5]In the case that neither role chain filler is known, LOOM creates a *skolem* instance to represent the common unknown filler. However, if `:same-as` is asserted for role chains that are not single-valued, skolem instancs are not generated, and merging will not occur.

Within a `tell`, any symbol prefixed by the character '?' is interpreted as a Lisp variable. The following code binds the Lisp variable `?r` to the instance `Robby`, and then asserts that "`Robby` is squeaky."

```
(setq ?r (get-instance 'Robby))
(tell (Squeaky ?r))
```

Variables can appear within a `tell` anywhere that a reference to a concept, instance, or relation would be valid. The following code binds the Lisp variable `?concept` to the concept `Squeaky` and then asserts that "`Robby` is squeaky."

```
(setq ?concept (get-concept 'Squeaky))
(tell (?concept Robby))
```

LOOM provides two additional means for creating a new knowledge base instance. The form

```
(create 'R2D2 'Robot)
```

creates an instance with identifier `R2D2`, and asserts that it is an instance of the concept `Robot`[6]. Within a `tell`, we can cause LOOM to generate a new knowledge base instance having a system-generated identifier by referencing a symbol prefixed by the character '$'. Subsequent references to that symbol within the same `tell` form refer to the same instance. The following form creates a new instance, asserts that it is a robot, and creates two additional instances filling the roles of its left and right arms.

```
(tell (Robot $r)
      (has-left-arm $r $a)
      (has-right-arm $r $b))
```

When an `:about` clause is the final form in a `tell`, the first argument to the `:about` keyword becomes the return value of the `tell`, e.g., the code

```
(tell (:about $r Robot (has-left-arm $a) (has-right-arm $b)))
```

returns the instance created to fill the variable `$r`.

5.2 Retraction: "forget"

Facts are retracted using the Lisp function `forget`. The syntax for `forget` is the same as that for `tell` (except that '$' variables, which implicitly cause assertions, cannot appear within a `forget` form). The effect of retracting a fact F is to withdraw explicit support for that fact. This causes LOOM to compute a new knowledge base state equivalent to what the state would have been if F had not been previously asserted.

[6] The form `createm` invokes the matcher following the creation.

```
(tell (Robot Robby))
(ask (Robot Robby))            ==> t
(forget (Robot Robby))
(ask (Robot Robby))            ==> nil
```

Note that the effect of a retraction in LOOM is to withdraw support for some particular fact—it does *not* have the effect of negating that fact. Assume that an `Android` is a special kind of `Robot`. Then in the following example, the retraction of explicit support for the proposition `(Robot Robby)` does not cause that proposition's truth value to become unknown, because after the retraction its truth is still derivable from the proposition `(Android Robby)`.

```
(tell (Robot Robby))
(tell (Android Robby))
(ask (Robot Robby))            ==> t
(forget (Robot Robby))
(ask (Robot Robby))            ==> t
```

The side-effect of creating new instances that sometimes accompanies an assertion is *not* retracted by a `forget` statement /indexget-instance:

```
(tell (Robot Robby))
(forget (Robot Robby))
(get-instance 'Robby)            ==> |I|ROBBY
```

The function `forget-all-about` can be called to retract all facts known about an instance, *and* to delete that instance from a knowledge base. `forget-all-about` takes a single argument, which must be a knowledge base instance.[7]

```
(tell (has-item Box5 Widget3))
(retrieve ?x (has-item Box5 ?x))         ==> (|I|WIDGET3)
(forget-all-about 'Widget3)
(retrieve ?x (has-item Box5 ?x))         ==> ()
(get-instance 'Widget3)                  ==> nil
```

In order to retract all facts about an instance without destroying the instance, one calls `forget-all-about` with the :dont-unintern-p option set to t, for example

```
(tell (has-item Box5 Widget3))
(forget-all-about 'Widget3 :dont-unintern-p t)
(get-instance 'Widget3)                  ==> |I|WIDGET3
```

The symbol "`*`" can be used in place of a role value in order to retract *all* fillers of a role:

```
(tell (:about Box5 (:filled-by has-part Widget3 Gizmo2)))
(forget (has-part Box5 *))
(retrieve ?p (has-part Box5 ?p))         ==> ()
```

[7] Unlike the `tell` and `forget` functions, the `forget-all-about` function does *not* implicitly quote symbols not prefixed by '?'.

When two (or more) different instances are asserted to be fillers of the same single-valued role, LOOM will automatically retract all but the last assertion. Hence, if the relation `color` is single-valued, then the statement sequences

```
(tell (color W3 GREY))
(tell (color W3 RED))
```

and

```
(tell (color W3 GREY))
(forget (color W3 GREY))
(tell (color W3 RED))
```

are equivalent. This behavior (automatic retraction of singleton role fillers) is called *clipping*. The automatic clipping feature can be disabled by evaluating the form:

```
(unset-features :auto-clip)
```

If clipping disabled, you can cause LOOM to selectively apply clipping to a specific relation by including the attribute `:auto-clip` in that relation's definition, e.g.,

```
(defrelation color :range Color
    :attributes (:single-valued :auto-clip))
```

5.3 Queries: "retrieve" and "ask"

The functions `retrieve` and `ask` provide the interface to LOOM's deductive query facility. `retrieve` is used for retrieving facts (instances) from a knowledge base, while `ask` is used to determine whether or not a proposition is true with respect to the currently stated rules and facts.

A query has one of the forms

`(retrieve` $?v_1$ *query*`)`
`(retrieve (`$?v_1$ ... $?v_n$`)` *query*`)`
`(ask` *query*`)`

The $?v_j$ are called *output variables*, and *query* is an open sentence in which the output variables appear unbound (unquantified). *query* can be an arbitrary expression in the first-order predicate calculus. The output variables must be prefixed with the character '?'. Section 5.3.1 below lists the different keyword expressions that can be used to form a query.

The next few examples assume a knowledge base containing the definitions and facts listed in figure 5.1.

A `retrieve` statement having one output variable, with no parentheses around that variable, returns a list of zero or more instances that satisfy the corresponding query. A `retrieve` statement that includes parentheses around its output variables returns a list of lists such that each sublist contains as many instances as there are output variables. The three queries below illustrate this behavior.

```
(defconcept Workstation)
(defconcept Lathe)
(defrelation lathe
   :domain Workstation :range Lathe
   :attributes :closed-world)
(defrelation time-since-last-servicing
   :domain Lathe  :range Integer)

(tell (lathe Workstation-2 Lathe-12)
      (lathe Workstation-2 Lathe-15)
      (lathe Workstation-5 Lathe-3)
      (time-since-last-servicing Lathe-12 50)
      (time-since-last-servicing Lathe-15 60))
```

Figure 5.1: Workstation/Lathe Model

```
(retrieve ?l (lathe Workstation-2 ?l))
   ==> (|I|LATHE-12 |I|LATHE-15)
(retrieve (?l) (lathe Workstation-2 ?l))
   ==> ((|I|LATHE-12) (|I|LATHE-15))
(retrieve (?w ?l) (lathe ?w ?l))
   ==> ((|I|WORKSTATION-2 |I|LATHE-12)
        (|I|WORKSTATION-2 |I|LATHE-15)
        (|I|WORKSTATION-5 |I|LATHE-3))
```

Any variable (i.e., symbols with the prefix '?') inside of a query that is not referenced either as an output variable or as a quantified variable (using one of the quantifiers `:for-some` or `:for-all`) is assumed to be bound externally by the Lisp environment that encloses the query. For example, the statements

```
(setq ?w (get-instance 'Workstation-2))
(retrieve ?l (lathe ?w ?l))
```

retrieve lathes belonging to `Workstation-2`. External variables (Lisp variables) cannot be used to represent unary or binary predicates, e.g., the statement `(retrieve ?x (?p ?x))` is *not* legal[8].

LOOM determines by context whether a symbol refers to a unary predicate, a binary predicate, a (single- or set-valued) function, or an instance, as illustrated by the following examples:

`(retrieve ?l (lathe ?l))` — `lathe` is a unary predicate

[8]This restriction will be removed in LOOM version 2.0.

```
(retrieve ?w (lathe ?w Lathe-15))            lathe is a binary predicate
(retrieve ?loc (location (lathe Workstation-2) ?loc))  lathe is a function
```

When a function references a binary relation that is not a single-valued relation, then the result of evaluating that function may be a set of instances rather than a single value. If a predicate whose domain does not specialize the built-in concept `Collection` is applied to a set of instances, the predicate is applied separately to each element of the set, yielding multiple sets of bindings to its arguments. The following queries are equivalent:

```
(retrieve ?t (time-since-last-servicing (lathe Workstation-2) ?t))
   ==> (50 60)
(retrieve ?t (:for-some ?lathe
                  (:and (lathe Workstation-2 ?lathe)
                        (time-since-last-servicing ?lathe ?t))))
   ==> (50 60)
```

In the first query, the expression `(lathe Workstation-2)` returns the set of instances `(|I|LATHE-12 |I|LATHE-15)`. The predicate `time-since-last-servicing`, is applied repeatedly to each of the elements in that set, yielding two bindings for the variable `?t`. The introduction of the existentially-quantified variable `?lathe` in the second query permits us to phrase an equivalent query such that all references to relations take the form of predicates.

LOOM permits the definition of relations whose domains are set-valued rather than single-valued. For example, the built-in relation `max` has the definition

```
(defrelation max
    :is (:function (?numbers) (apply (function max) ?numbers))
    :domain Collection  :range Number)
```

If a function whose domain does *not* specialize the built-in concept `Collection` is applied to a set of instances, it returns a set representing the union of applications of that function to individual elements of the set. These two kinds of semantics are illustrated in the following query:

```
(retrieve ?t
   (max (time-since-last-servicing (lathe Workstation-2)) ?t))
   ==> (60)
```

The expression `(lathe Workstation-12)` returns the set of instances `(|I|LATHE-12 |I|LATHE-15)`. The application of the function `time-since-last-servicing` to that set yields the set `(50 60)`. The predicate `max` computes the maximum of the set `(50 60)`, and returns a list containing the number 60.

5.3.1 Query Expression Constructors

This section presents the various kinds of constructs that LOOM provides for composing a query. The examples refer in most cases to the models defined in subsections 3.1.1 and 3.1.2.

(`:and` $term_1$ ...$term_n$) — Logical And

Returns true if each of the terms $term_j$ is satisfied.

Example:
```
(:and (Physical-Container ?x) (Fragile-Thing ?x))
```
Returns true if `?x` is both a `?Physical-Object` and a `Fragile-Thing`.

(`:or` $term_1$ ...$term_n$) — Logical Inclusive Or

Returns true if at least one of the terms $term_j$ is satisfied.

Example:
```
(:or (Widget ?x) (Gizmo ?x))
```
Returns true if `?x` is either a `Widget` or a `Gizmo`.

(`:for-some` ($?v_1$ ...$?v_n$) $term$) — Existential Quantification

Returns true if there exist values for the variables $?v_1$ through $?v_n$ that cause the boolean expression $term$ to be satisfied. The symbols $?v_j$ must be prefixed with the character '?'. If the variables list has only one variable, then the inner parentheses can be dropped.

Example:
```
(:for-some (?i1 ?i2)
    (:and (has-item ?box ?i1) (has-item ?box ?i2)
          (>= (* (weight-in-kilos ?i1) 2)
              (weight-in-kilos ?i2))))
```
Returns true if there exist two items in the box `?box` such that one item weighs at least twice as much as the other.

(`:for-all` ($?v_1$ ...$?v_n$) (`:implies` $term1$ $term2$)) — Universal Quantification

Returns true if only if all sets of bindings of the variables $?v_1$ through $?v_n$ that satisfy the boolean expression $term1$ also satisfy the boolean expression $term2$. The symbols $?v_j$ must be prefixed with the character '?'. If the variables list has only one variable, then the inner parentheses can be dropped.

Example:
```
(:for-all (?item)
    (:implies (has-item ?box ?item)
         (<= (weight-in-kilos ?item)
             (/ (weight-of-items ?box) 10))))
```
Returns true if no item in the box bound to the variable `?box` weighs more than 10 percent of the total weight of items in that box.

Comment: The use of an `:implies` clause at top level within a `:for-all` expression is mandatory.[9] The reason for this springs from the fact that the LOOM syntax does not require that variables be "typed." The universally quantified variables $?v_j$ must all appear within $term1$, the first argument of the (top-level) `:implies` clause.

(`:not` *term*) — PROVABLY FALSE

Returns true if LOOM can prove that the boolean expression *term* is not satisfiable. If LOOM can neither prove nor disprove *term*, then the entire expression returns false.[10]

Example:
```
(:not (weight-in-kilos ?part 5))
```
Returns true if the weight of `?part` is not 5 kilos.

```
(:and (Robot ?x) (:not (Android ?x)))
```
Returns true if `?x` is a `Robot` and `?x` cannot possibly be an `Android`.

Comment: Unbound variables within the negated term *term* must be bound during the time that *term* is evaluated. For example, the query `(retrieve ?x (:not (Robot ?x)))` is *not* legal.

(`:fail` *term*) — NOT PROVABLY TRUE

Returns true if LOOM cannot prove that the boolean expression *term* is satisfiable. The effect of the `:fail` operator is to introduce the semantics of "negation as failure" into a query.

Example:
```
(:and (Robot ?x) (:fail (Android ?x)))
```
Returns true if `?x` is a `Robot` and `?x` is not known to be an `Android`.

Comment: Unbound variables within the negated term *term* must be bound during the time that *term* is evaluated. For example, the query `(retrieve ?x (:fail (Robot ?x)))` is *not* legal.

(`:one-of` $item_1$ ... $item_n$) — SET CONSTRUCTOR

Returns a set containing the items $item_j$.

Example:
```
(member-of (color ?robot)
  (:one-of 'Red 'Blue 'Green))
```
Returns true if the color of the instance that `?robot` is bound to is one of red, blue, or green.

[9] This restriction will be dropped in LOOM version 2.0

[10] More precisely, the value returned is not `false` but `unknown`.

(`:about` *instance* $expr_1$ ... $expr_n$) — TRUE ABOUT INSTANCE

Returns true if the instance bound to the expression *instance* satisfies each of the concept expressions $expr_k$.

Example:
```
(:about ?r Robot
   (:all has-arm (:exactly 2 has-finger)))
```
Returns true if the instance bound to `?r` is a `Robot` and each of its arms has exactly two fingers.

Comment: The `:about` clause is provided as a means for introducing concept expressions into a query. Its syntax is the same as that used within a `tell` or `forget` statement.

R[i] — EVALUATE I'TH FILLER

Evaluates to the i^{th} filler of the role represented by R.

Example:
```
(:same-as (location (has-finger[1] ?h))
          (location (has-finger[2] ?h)))
```
Returns true the first and second fingers attached to `?h` are in the same place (i.e., are touching).

Comment: This construct provides a way to index into the set of fillers of a role. If the *order* of the fillers is significant, the role fillers should be asserted using one of the constructs `:filled-by` or `:filled-by-list`.

(`:collect` *?v term*) — COLLECT SATISFYING VALUES (COMPUTED SET)

Returns the set of items i such that *term* is satisfied when i is bound to *?v*.[11]

Example:
```
(max (:collect ?w
         (:for-some ?part
             (:and (has-item ?box ?part)
                   (Widget ?part)
                   (weight-in-kilos ?part ?w)))))
```
Returns the maximum weight among the set of items of the box `?box` that are widgets.

Comment1: As illustrated in the example, the need for the `:collect` construct arises when it is necessary to evaluate a complex expression in order to generate a set of values as the argument to a predicate or function (recall that the domain of the relation `max` is the concept `Set`).

Comment2: The `:collect` construct represents a recursive invocation of the retrieve function.

[11] `:collect` will appear in LOOM version 2.0.

5.4 Open and Closed World Semantics

Logical deduction in LOOM assumes an open world semantics. This means that the "law of the excluded middle" does not apply—if LOOM cannot prove or disprove a proposition, then it assigns that proposition the value UNKNOWN.

The correct interpretation of the `ask` function is that a non-nil return value means TRUE, while a nil return value means "FALSE or UNKNOWN." Suppose in our factory domain all parts are either widgets, gizmos, or doohickeys:

```
(defconcept Part
   ...
   :disjoint-covering (Widget Gizmo Doohickey))
```

and suppose that a part `P3` is known to be either a widget or a gizmo:

```
(tell (:about P3 (:or Widget Gizmo)))
(ask (Doohickey P3))           ==> nil
(ask (:not (Doohickey P3)))    ==> t
```

When we ask LOOM if `P3` is an instance of `Doohickey`, LOOM returns `nil`, which means "maybe." When we ask if `P3` is *not* an instance of `Doohickey`, LOOM returns `t`, which means "yes."

LOOM's `ask` facility was designed to return two-valued (rather than three-valued) responses (1) because application programmers have indicated a strong preference for a binary response, and (2) because a ternary version of `ask` would require a search for a disproof of a proposition each time that a proof of that proposition could not be found (i.e., the computational overhead of a ternary `ask` is likely to be significantly greater than that for a binary `ask`). Frequently, LOOM application programs interpret a `nil` value from `ask` to mean FALSE. This represents an *implicit* assumption of a *closed-world* semantics, effectively saying "If a proposition P cannot be proved TRUE, then assume that P is FALSE."

LOOM supports *explicit* assumption of closed-world semantics for concepts and relations. Consider the following sequence of definitions and assertions:

```
(defrelation has-item)
(tell (has-item B3 W5) (Box B3) (Widget W5))
(ask (:about B3 (:all has-item Widget)))   ==> nil
(defrelation has-item :attributes :closed-world)
(ask (:about B3 (:all has-item Widget)))   ==> t
(tell (:about B3 (:at-least 2 has-item)))
(ask (:about B3 (:all has-item Widget)))   ==> nil
```

Initially, open-world semantics (the default) applies to the relation `has-item`, and the query "Are all of `B3`'s items widgets?" returns `nil` (UNKNOWN) because LOOM assumes that there might exist items other than `W5` that belong to `B3`. When `has-item` is redefined to assume closed-world semantics, the same query returns the answer `t` (TRUE) because LOOM is now willing to assume that `W5` is the *only* item of `B3`. Continuing, when we explicitly state

that B3 has at least two items, then the closed-world assumption applied to the role has-item on B3 is revised, leading to the conclusion that B3 has exactly two items (one of which may or may not be a widget). This example illustrates that the use of closed world semantics moves us into the realm of non-monotonic logic. Formally, the assumption of closed-world semantics represents a limited application of *circumscription*.

Indiscriminate application of the closed-world assumption can lead to anomolous results. For example, suppose that the closed-world assumption is applied to the concept Part, and also to the concepts Widget, Gizmo, and Doohickey, which we again assume collectively constitute a disjoint covering of Part. If all we know about a part P5 is that it is an instance of Part, then the closed-world semantics infers (:not (Widget P5)), (:not (Gizmo P5)), and (:not (Doohickey P5)), implying that P5 is *not* an instance of Part, i.e., we have a contradiction. However, LOOM's default rule facility provides an alternative means for closing concepts. For example, the default rule:

```
(default Part (:not Fragile-Thing))
```

can be read as "If p is a part, and p is not known to be fragile, then assume that p is not fragile," i.e., the rule closes the concept Fragile-Thing.

Anomolous results analogous to that just illustrated in the Part-Widget-Gizmo-Doohickey scenario can be achieved by injudicious application of closed-world semantics to hierarchies of binary relations. Hence, if users want to hang themselves by making too many closed-world assumptions, LOOM will accomodate them.

Certain *syntactic* forms within the query language dictate a localized assumption of closed-world semantics (valid only while evaluating the form). For example, when evaluating the query

```
(ask (:for-all ?r (:implies (Robot ?r) (2-Armed-Robot ?r))))
```

LOOM circumscribes the set of instances of Robot [12]. For the query

```
(retrieve ?t
   (max (time-since-last-servicing (lathe Workstation-2)) ?t))
```

LOOM circumscribes the set represented by (time-since-last-servicing (lathe Workstation-2)) in order to provide a *definite* input to the function max.

5.5 The Query Optimizer

Instead of implementing a query *interpreter*, LOOM *compiles* every query into equivalent Lisp code. During query compilation, LOOM's query optimizer applies a variety of transformations to a query, looking for an evaluation strategy that will optimize the query's run-time

[12] LOOM version 2.0 eliminates the localized closed-world assumption within universally quantified expressions, e.g., in order to ask this query, LOOM would require that the closed-world assumption apply to the concept Robot

performance. Unlike database query optimizers, whose decisions are based principally on statistical features of the database, the decisions made by LOOM's query optimizer are based on *semantic* knowledge about the concepts and relations referenced within a query. For example, the optimizer evaluates the domain and range of each relation, compares the relative specificity of pairs of concepts, considers whether relations are single-valued or set-valued, and considers whether or not relations have inverses. In many cases, the code produced by the LOOM optimizer is as efficient as hand-coded Lisp.

The functions `retrieve` and `ask` are implemented Lisp macros that invoke the query optimizer during macro expansion. This means that *all concepts and relations referenced within a query must be fully-defined before the Lisp interpreter or compiler macro-expands that query.*[13]

5.6 Queries as Generators

The function `do-retrieve` facilitates the use of queries as *generators*. A query generator has the form:

(`do-retrieve` *variables query*
LispForms)

`do-retrieve` uses the query *query* to generate sets of bindings for the variables *variables*, and evaluates the code in *LispForms* once for each set of variable bindings. For example, the `unload-box` method defined in section 3.2.1 contained the following generator:

```
(do-retrieve ?part
             (has-item ?box ?part)
  (forget (has-item ?box ?part))
  (tell (location ?part (location ?box))))
```

In this code, the query "`(has-item ?box ?part)`" successively binds the variable `?part` to each of the parts that are items of the box `?box`. The body of the `do-retrieve` (the functions `forget` and `tell`) is evaluated once for each binding of the variable `?part`.

5.7 Invoking the Matcher: "tellm" and "forgetm"

In order to better utilize LOOM's reasoning capabilities, it is useful for a programmer to have some awareness of the deductive processing that takes place during the execution of a LOOM application. This section provides a few details on the LOOM matcher. Internally, LOOM manages a set of data structures that enable it to efficiently compute answers to deductive queries. These data structures usually become invalid when new facts are asserted or retracted. The process of updating these data structures is called *matching*. The name

[13] This restriction may be removed in a future LOOM release.

derives from the fact that a significant fraction of the match process is devoted to the computation of instantiation relationships between instances and concepts, i.e., determining which instances match which concepts.

In order to guarantee the validity/currency of internal knowledge base structures, functions such as `retrieve` and `ask` automatically invoke the matcher (prior to executing a query) in the case that the matcher hasn't already been invoked subsequent to the most recent update (assertion, retraction or revision of a definition or rule) to the knowledge base. Hence, even if the user never invokes the matcher explicitly, the query facility will still function correctly. Unfortunately, the production rule facility does not share this property:

The firing of production rules occurs only at the end of a match cycle. Hence, in order to assure the timely firing of productions, it is necessary that the matcher be invoked frequently. The simplest strategy for guaranteeing that productions are fired in a timely fashion is to call the matcher immediately after each update. This can be accomplished by using the functions `tellm`, `forgetm` and `forget-all-about-m` in place of `tell`, `forget`, and `forget-all-about`, and by calling the function `new-time-stamp`[14] after each call to other update functions such as `forget-all-about` and `define-concept`.

There are several reasons why continually invoking the matcher might not be desirable, some having to do with the semantics of transactions, and some having to do with performance. Consider a transaction T defined as a sequence $<U_1, \ldots, U_n>$ of assertions and retractions. For some situations, the sequential application of the updates U_j could result in the generation of intermediate knowledge base states that are in some way invalid. These invalid states might trigger production rules that should not have been fired. Each call to the matcher effectively causes a transition to a new knowledge base state. If T is programmed so that the matcher is called only once, at the end of the transaction, then LOOM will not generate any "intermediate" knowledge base states (effectively, all updates are made in parallel), and hence certain kinds of undesirable side-effects can be avoided.

Suppose the effect of some application process is to make a series of assertional updates to a knowledge base (for example, this process might be loading a file of assertions). In many cases LOOM's overall performance will be faster if the matcher is called once at the end of the updates, rather than calling it in between each update. Summarizing, when it can be reliably determined that the matcher need not or should not be invoked directly after a knowledge base update, then its preferable to avoid explicit calls to the matcher. Otherwise, the matcher should be invoked after each knowledge base update.

While on the subject of performance, we also note that while it is perfectly acceptable to interleave concept and rule declarations with factual assertions, LOOM will tend to perform better whenever concept and rule declarations collectively precede any factual assertions.

[14] Not yet implemented.

Chapter 6

Actions and Methods

In addition to its powerful classification mechanisms, LOOM provides a number of behavioral operators to support knowledge-based inferencing. The primary elements are *actions*, *methods*, and *productions*. Each of these operators, when invoked, creates a *task* whose execution may be controlled.

6.1 Actions

An action, in LOOM, is an object that specifies a procedural operation, or a set of operations. An action is considered to be a generic operation, analogous to, e.g., a CLOS generic function, that may take on widely different meanings in different contexts. Each action is defined by a `defaction` declaration which specifies the name of the action and lists its formal parameters. Optional keyword parameters can specify filters to aid method selection and error-handling. The specific functions to be applied when an action is invoked are defined in one or more *methods* of the same name as the action. Typically, users will wish to define actions to represent all physical actions in the domain being modelled and all operations upon conceptual entities being modelled in LOOM .

An action may be defined to take zero or more arguments; the methods which implement the action must have the same number of arguments as the `defaction` statement. If the action accepts arguments, these arguments will be dynamically bound to LOOM objects to be operated upon by the action. If a method is defined (using *defmethod*) before the corresponding action is declared, LOOM will create a system-defined action using default values.

`(defaction` *name parameters* `&key` *filters missing – methods*`)` — DEFINE AN ACTION

> Defaction declares an operator that will be implemented by one or more methods of the same name.

Example:
```
(defaction move-object (?object ?to)
  :filters (:overrides :most-specific :last-one)
  :missing-method :no-op)
```
"Define an action called move-object and select among move-object methods by considering override relations, specificity, and aging".

When applying this action, LOOM will consider all methods (defined by `defmethod`) with the name `move-object`. From the full list, it will consider only those methods whose `:situation` patterns are satisfied by the arguments passed to the action. From this list, it eliminate methods whose titles were listed in the `:overrides` argument of another surviving method. Then it will select those methods with the most specific `:situation` patterns[1]. If there remains more than one method remaining, LOOM will select the method most recently defined and evaluate its `:response` argument. If LOOM can find no methods for this action, it will simply perform no operation and return nil.

6.2 Methods

In the preceding section, we saw that the knowledge base developer could define actions and associate them with specific sets of operations in the knowledge base. The operations that are associated with an action are termed its methods. Invoking an action, therefore, actually means setting into motion the operations associated with the action as defined in one of its associated methods.

6.2.1 Defining Methods

Methods are defined using the `defmethod` macro. Multiple methods may be defined with the same name, corresponding to the name of a generic action the method implements. If no action of the same name has been previously defined, the use of the `defmethod` macro will cause LOOM to create a system-defined action with that name using default values for the keywords.

Within the definition of the method, the method is given an optional title to distinguish it from other methods of the same name. The title field is useful whenever users need to refer to that particular method. The `:situation` keyword is the primary mechanism by which methods are matched to the appropriate context. The value of the `:situation` keyword is a query-forming expression using the same syntax as the argument to a `retrieve` statement (see Chapter 5). The query expression can be used to identify those situations for which this particular method is an appropriate one to execute. When the `:most-specific` filter is specified for an action (the default case), the relative specificity of each situation field will be examined by the action to select among the possible methods to evaluate. While passed in

[1] In the current implementation, specificity is computed only between patterns of the form `(C ?X)`, where C is a concept and ?x is a variable. This restriction will be removed in a future release

parameters may be used freely in the `:situation` clause, new variables must be introduced by a `:for-some` or `:for-all`. The `:response` form is performed once for each binding of the variables.

The `:overrides` keyword allows the users to directly specify methods which should be subordinate to the current method. All methods referred to by title in the `:overrides` list will be removed from consideration if the `:situation` clause of this method is valid. In other words, the continued presence of this method in the action's candidate methods list will cause these less preferred methods to be dropped from the list.

The `:response` clause forms the body of the method. This clause is a list of Lisp forms which are to be evaluated when the method is invoked.

`(defmethod` *name parameters* `&key` *title situation overrides response*`)` — DEFINE A METHOD

Defmethod declares an operator which implements an associated action of the same name.

Example:
```
(defmethod move-object (?object ?to)
  :title "Move an Object"
  :situation (Physical-Object ?object)
  :response ((format t "Move  S to
       location  S %" ?object ?to)
      (tell (location ?object ?to)))
```
"Define a move-object method with the title given. This method may be used if the object is a Physical-Object. When invoked, issue the format statement and execute a tell changing the location role for the ?object to the value ?to".

Comment: This definition of move-object applies only when ?object is bound to an instance of the concept Physical-Object. If the `:situation` keyword were not given, this method would be the default method, to be used when no other move-object method is appropriate.

6.2.2 Method Selection

As described above, LOOM allows an action to have multiple implementations, i.e. multiple methods. When LOOM is asked to invoke a multiple method action, it resolves the conflict by choosing a method according to the filters given in the definition of the action. The default and most commonly used filters are `:most-specific` and `:last-one`. Together, these filters select the methods that are most specifically defined through the `:situation` field and among these select the one that is the most recently defined.

Here is an example. Suppose we have these following definitions:

```
[1]   (defmethod move-object (?object ?to)
          :response
              ((format t "Move ~S to ~ location ~S~%" ?object ?to)
               (tell (location ?object ?to)))
[2]   (defmethod move-object (?object ?to)
          :situation (Physical-Object ?object)
          :response
              ((format t "Move ~S to ~ location ~S~%" ?object ?to)
               (tell (location ?object ?to)))
```

When LOOM is asked to do the action `move-object`, it will go through the situation fields and try to pick the definition that most closely matches the current situation. Since definition [1] has no `:situation` field, we can think of that as the "default" situation, matching anything not previously matched. Definition [2] will work if the object is a `Box` that doesn't have any `Fragile` members.

Suppose, however, we want to have another version of `move-to`, to be invoked if we are moving a box with any fragile items. In this case, we want to indicate that the box should be moved carefully. To accomplish this, we will need to indicate to LOOM the additional specifics of the situation – namely, that the box contains fragile items. This is done through use of the `:situation` keyword, as follows:

```
(defmethod move-to (?box ?to)
   :situation (:and (Box-of-Parts ?box)
                    (:about ?box (:some has-item Fragile-Thing)))
   :response ((format t "CAREFULLY moving box")
              (tell (location ?box ?to))))
```

This gives a more specific situation to LOOM – "if the object is a `Box-of-Parts` and it has one or more `Fragile-Things` in it". LOOM uses the pattern following the `:situation` keyword to resolve conflicts among multiple candidate methods.

6.3 Productions

A LOOM production is a data-directed (recognize-and-act) rule used to detect a particular event and to cause a task (one or more actions) to be invoked. The `defproduction` operator is used to define or redefine such a construct within LOOM. The operator defines a production using the following arguments: 1) a symbol specifying the name of the to-be-defined production, 2) the keyword `:when` followed by a predicate pattern to be matched, 3) one of the keywords `:perform` or `:schedule` followed by a LOOM method to be invoked, and 4) if the keyword `:schedule` were used, the optional keyword `:priority` followed by one of `:low`, `:medium`, or `:high`.

The actions of a production can trigger other productions, allowing for a data-directed chain of reasoning. Since the actions executed by the production can be any LOOM construct,

a very close coupling exists between the behavioral constructs in LOOM and its declarative representations.

`(defproduction` *name* *:when* *:perform* *:schedule* *:priority*`)` — DEFINE A PRODUCTION

> Defines a production with name *name* that will cause a particular LOOM method to be invoked, when the specified situation occurs.
>
> Example:
> ```
> (defproduction move-full-box
>   :when (Full-box ?object)
>   :schedule (move-object ?object
>              (get-instance 'Warehouse))
>   :priority :high)
> ```
> "Define a production, move-full-box, as follows: When any object satisfies the concept Full-Box, Schedule the task of moving that object to the Warehouse, and make it a high-priority task."
>
> Comment: Note that, while the task is a high-priority one, it is placed on a task queue. Therefore, the task would not be done immediately if there were other high-priority tasks already queued. If we wanted the task `move-object` to be invoked immediately, we would replace the `:schedule` keyword with `:perform` and remove the (now irrelevant) `:priority` argument. Also note that, unlike `defconcept` and `defrelation`, any patterns or concepts used in the premise of the production must already be defined.

A production is triggered when its premise is satisfied. The `:when` keyword is used to indicate a premise to be satisfied by one or more instances in the current knowledge base.

When a production's premise is satisfied, LOOM looks at the conclusion of the premise to decide when its actions should be taken. If the keyword `:schedule` begins the conclusion, then the actions should be placed at the end of a queue, called the *task queue*.

Consider the LOOM production:

```
(defproduction P1
   :when (Almost-Full-Box ?box)
   :schedule (remove-full-box ?box))
```

> "When a box is an `Almost-Full-Box`, add to the task queue the task of removing that box using `remove-full-box`."

Tasks that are so scheduled will be performed at the end of the current match cycle, after all productions have fired.

6.4 Tasks

A LOOM task is an application of a specific action to one or more objects. A task consists of an operator (which causes a particular method to be invoked as in 6.2) and one or more objects over which to apply this operator.

Tasks may be started immediately (performed) or scheduled. Tasks that are scheduled are placed onto a task queue. Queued tasks will be executed, in order of their placement onto the queue, when the operator `perform-all` is invoked.

LOOM's behavioral operators permit tasks (collections of operations) to be done immediately or to be queued. We have seen that the conclusion field of a production may contain either the `:perform` (immediate) or the `:schedule` (queue) keywords. The task being done may itself use the perform and/or schedule keywords, without leading colons, to initiate or queue additional tasks.

`(schedule (`*task*`) :`*priority*`)` — SCHEDULE A TASK FOR LATER OPERATION

The schedule keyword (or the `:schedule` field in a production conclusion) causes LOOM to schedule a task to be done later. In addition to whatever arguments are necessary for the task, the schedule keyword takes and additional keyword `:priority` followed by one of the following keywords: `:high`, `:medium`, `:low`.

Example: `(schedule (move-box ?box) :priority :medium` puts the task `move-box` on the task queue, with priority medium.

`(perform` *task*`)` — DO A TASK IMMEDIATELY

The perform keyword causes LOOM to immediately execute the specified operation with the current variable instantiations, if any.

Example: `(perform (unload-box ?box))`
Causes the operation `unload-box` to be immediately performed with the current binding of the free variable `?box`.

In the previous section on productions, we saw the following LOOM construct:

```
(defproduction move-full-box
   :when (Almost-Full-box ?object)
   :schedule (move-object ?object (get-instance 'Warehouse))
   :priority :high)
```

The use of the `:schedule` keyword placed the task `move-object` on the task queue. Its arguments are the current object and the current `Warehouse`. This task will be actually

done at the end of the current match cycle (see Chapter 5.7. Had the keyword `:perform` been used instead of `:schedule`, the task would have been done immediately.

A task may also be done immediately by invoking the `perform` operator (no leading colon) wihtin a method. As an example, the LOOM construct: `(perform (unload-box ?box))` causes the following task to be performed immediately: "Apply the operator `unload-box` to the instance bound to the variable `box`".

Chapter 7

Using Knowledge Bases

The set of objects that are defined in LOOM at any given point are contained in a set of LOOM knowledge bases. Since LOOM assumes an open world of objects and knowledge, the state of these knowledge bases at any point in time may be thought of as LOOM current image of the world. LOOM provides functions for defining and manipulating snapshots of this image (that is, for manipulating knowledge bases).

The `defkb` operator associates a new symbol name with a new knowledge base. A pathname is optionally associated with the new knowledge base and the objects stored to that file. From that point on, the knowledge base can be considered as another type of LOOM object and can be accessed via LOOM functions.

LOOM provides many operators for manipulating all or part of a knowledge base. LOOM knowledge bases consist of four partitions: `concepts`, `relations`, `instances`, and `behaviors`. In the following examples, a `:partition` argument may be followed by a list of one or more partitions. If no `:partition` argument is supplied, then all partitions are operated on.

The operators described below provide the basis of LOOM's knowledge base capability.

`(defkb` *name parents* `:` *pathname*`)` — Define a Knowledge Base

Defines a new knowledge base with name *name* and parents *parents*. LOOM permits multiple knowledge bases to be loaded simultaneously, and further allows these multiple knowledge bases to be hierarchically structured.

Example: `(defkb 'incore-kb () :pathname "on-disk")`
Defines a knowledge base that will be referred to within LOOM as `incore-kb`. This knowledge base will be saved to disk in the file named `on-disk`.

Comment: LOOM automatically creates a "top-level" knowledge base called `upper-structure-kb`. Knowledge bases created using `defkb` are by default made descendants of `upper-structure-kb`.

Comment2: The `in-core` and `on-disk` names could have been the same. We wanted to make the potentially different names explicit.

(`save-kb` *name* :*partitions* :*path – name*) — SAVE A LOOM KNOWLEDGE BASE

The `save-kb` operator is used to write out the source code for all or part of a LOOM knowledge base. The developer may specify how much of the knowledge base is to be saved, and where the saving should take place.

Example1: `(save-kb)`
Saves the (entire) current knowledge base to the default filename (associated with this knowledge base at creation – see `defkb`).

Example2:
```
(save-kb 'incore-kb :partitions :instances
  :path-name "ergo:/loom/incore-inst.lisp")
```
Saves the instances from the knowledge base `incore-kb` to the file `/loom/incore-inst.lisp` on host `ergo`.

(`load-kb` *name path – name*) — LOAD A LOOM KNOWLEDGE BASE

The `load-kb` operator loads a specified LOOM knowledge base from definitions contained in the specified file. Partitions need not (and in fact, cannot) be supplied to the `load-kb` command. The user should, as a reminder, include some reference to the type of LOOM objects being saved in the knowledge base name.

Example:
```
(load-kb 'incore-kb :path-name
          "ergo:/loom/incore-inst.lisp")
```
Will load the definitions in host `ergo`'s file `/loom/incore-inst.lisp` into the LOOM knowledge base `incore-kb`.

(`clear-kb` *name* :*partitions*) — CLEAR A LOOM KNOWLEDGE BASE

The `clear-kb` operator deletes all (or portions) of a specified knowledge base. If no knowledge base name is specified, the current knowledge base is used.

Example1: `(clear-kb)`
Clears the entire contents of the current knowledge base.

Example2: `(clear-kb 'incore-kb :partitions :relations)`
Clears out all the relations from `incore-kb`.

(`list-knowledge-bases`) — LIST ALL LOOM KNOWLEDGE BASES

The `list-knowledge-bases` operator lists all currently loaded LOOM knowledge bases.

(`list-kb` *name* :*partitions* :*sort* − *p*) — LIST A KNOWLEDGE BASE

The `list-kb` operator lists all or a portion of the specified knowledge base onto the screen. The operator's default behavior is to list all objects in the current knowledge base, in order of their creation or entry into the knowledge base.

Example1: `(list-kb)`
Lists the current knowledge base.

Example2:
```
(list-kb 'incore-kb :partitions :concepts
  :sort-p 't)
```
Lists all the concepts defined in the knowledge base `incore-kb`. The concepts are printed out in alphabetical order.

(`in-kb` *knowledge* − *base*) (`change-kb` *knowledge* − *base*) — RESET THE KNOWLEDGE BASE

The `in-kb` operator is a macro that resets the LOOM knowledge base to the specified knowledge base at either run time or compile time. The `change-kb` operator is a function that accomplishes the same purpose at run time only. Additionally, `change-kb` takes the key `:no-checking-p` which, when non-nil, causes LOOM to assume the argument is a valid object and no error checking is performed.

Example1: `(in-kb foo-kb)`
Resets the knowledge base and changes it to `foo-kb`.

Example2: `(change-kb 'bar-kb :no-error-checking-p t)`
Changes the knowledge base to the object `bar-kb` when called at run time without the normal error checking.

Appendix A
Grammar for TBox Language

The following grammar describes the concept-forming and relation-forming expressions that are used in TBox definitions.

```
concept-expr ::=
    ConceptName |
    ( {:AND | :OR} concept-expr+ ) |
    ( :NOT concept-expr ) |
    ( :ONE-OF {InstanceId | Constant}+ ) |
    ( :THE-ORDERED-SET Scalar+ ) |
    ( :THROUGH Scalar Scalar ) |
    ( {:AT-LEAST | :AT-MOST | :EXACTLY} Integer relation-expr ) |
    ( {:ALL | :SOME | :THE} relation-expr concept-expr ) |
    ( :FILLED-BY relation-expr {InstanceId | Constant}+ ) |
    ( {:SAME-AS | :SUBSET | < | > | <= | >= | = | /=}
          {relation-expr+ | relation-expr Constant} ) |
    ( :SATISFIES ( ?Var ) Query-Expr ) |
    ( :PREDICATE ( Var ) Form ) ;
relation-expr ::=
    RelationName |
    ( :AND relation-expr+ ) |
    ( :DOMAIN concept-expr ) |
    ( :RANGE concept-expr ) |
    ( :INVERSE relation-expr ) |
    ( :COMPOSE relation-expr+ ) |
    ( :SATISFIES ( ?Var+ ) Query-Expr ) |
    ( :PREDICATE ( Var+ ) Form ) |
    ( :FUNCTION ( Var+ ) Form ) ;
```

In this grammar, **?Var** is a symbol beginning with the character **?**. **ConceptName**, **RelationName**, and **InstanceId** are symbols which name LOOM concepts, relations and instances. **Scalar** is a number, or a symbol representing a discrete quantity. **Constant** is a Lisp symbol, string, or number. **Integer** and **Form** are Lisp integers and forms. **Query-Expr** is an expression in the LOOM database language (see Appendix B).

Appendix B
Grammar for ABox Language

The following grammar has been generalized to cover both assertion and retrieval. Certain constructs (see comments below) can be used in one mode but not the other.

```
statement ::=
    ( ASK query-expr ) | ( RETRIEVE ?varlist query-expr ) |
    ( {TELL | TELLM | FORGET | FORGETM} term* ) ;
query-expr ::=
    term |
    ( {:AND | :OR} query-expr+ ) |
    ( {:NOT | :FAIL} query-expr ) |
    ( :IMPLIES query-expr query-expr ) |
    ( {:FOR-SOME | :FOR-ALL} ?varList query-expr ) ;
term ::=
    ( concept instance ) |
    ( relation instance+ value ) |
    ( :SAME-AS instance instance ) |
    ( :ABOUT instance about-term* ) ;
about-term ::=
    concept | ( concept ) |
    ( relation value ) |
    ( :FILLED-BY relation value* ) |
    ( :FILLED-BY-LIST relation List ) |
    ( {:AT-LEAST | :AT-MOST | :EXACTLY} Integer relation ) |
    ( {:ALL | :SOME | :THE} relation concept ) ;
?varlist ::=
    ?Var | ( ?Var+ ) ;
concept ::=
    ConceptName | ?Var ;
relation ::=
    RelationName | RelationName[ Scalar ] | ?Var ;
instance ::=
    InstanceId | ?Var | $Var | ( relation instance ) ;
value ::=
    instance | Constant | ( :ONE-OF Constant+ ) | * ;
```

In this grammar, **?Var** and **$Var** are symbols beginning with the characters **?** and **$** respectively. **ConceptName**, **RelationName**, and **InstanceId** are symbols which name LOOM concepts, relations and instances. **Scalar** is an integer, or a symbol representing a discrete quantity. **Constant** is a Lisp symbol, string, or number. **List** is a Lisp list.

The grammar is constrained as follows: 1) assertions and retractions cannot contain **:SOME**, **:THE**, or **:ONE-OF** operators, or indexed relations; 2) the * value is only meaningful in retractions; 3) queries cannot contain **:SAME-AS** or **:FILLED-BY-LIST** operators, or **$Var** variables, or **?Var** variables used as concepts or relations.[1]

In assertions and retractions, if a **?Var** variable is used in place of a concept, relation, or instance, it should be bound to a concept, relation, or instance object rather than the name of such an object.

[1] LOOM version 2.0 will remove some of these restrictions.

Appendix C

Glossary of LOOM Terms

This Glossary includes all significant terms used in the LOOM User's Manual. Specific LOOM functions, keywords, and constructs (such as `defrelation`, `defconcept`, `:is`, etc.) are described in the LOOM Reference Manual.

Action An action is a generic operation that can be invoked to accomplish some specific task, implemented as a set of methods.

Assertion A statement of a particular fact or constraint that describes a given domain.

Assertional Language An assertional language is designed to state constraints or facts that apply to a particular domain or world.

Behavior Language The portion of LOOM used to represent the procedurally-expressed aspects of both the application domain and the application program. Includes declarations of actions, methods, and production rules.

Binary Relation A relation between exactly two concepts.

Classification-Based Knowledge Representation Classification-based knowledge representation systems are characterized by three common architectural features: 1. they are logic-based (ie. they appeal to first-order logic for their semantics), 2. they draw a distinction between terminological and assertional knowledge, and each system implements its own specialized term-forming language, and 3. they include a classifier that organizes terms (concepts) into a taxonomy, based on the subsumption relationships between terms. LOOM is a classification-based knowledge representation system.

Classifier A classifier is a type of an inference engine that can reason with descriptive knowledge. Key features of a classifier are its ability to determine subsumption relationships between descriptions (when one description is implied by another) and to combine (unify) sets of descriptions to form new descriptions.

Clipping When two or more different instances are asserted to be fillers of the same single-valued role, LOOM will automatically retract all but the last assertion. This behavior is called `clipping`, and may be toggled on/off in LOOM .

Closed-World Semantics Under closed-world semantics it is assumed that "if proposition `P` cannot be proved `TRUE`, then assume that `P` is `FALSE`." LOOM allows programmers to explicitly declare that concept or a relation operates under the assumption of closed-world semantics (See also Open-World Semantics).

Concept A concept is the LOOM knowledge base construct that binds a symbol to a definition. Concepts are used similar to frames or classes in other knowledge representation systems. A concept is also considered as a unary relation.

Constraint A necessary condition that applies to all instances of a concept or to all tuples in a relation. (See Constraint Rule).

Constraint Rule A rule of the form "if $< a >$ then $< b >$ defines a constraint on the class of entities satisfying the expression $< a >$. If the expression $< a >$ denotes membership in a concept or relation A, then this rule is said to represent a constraint on A.

Data-Driven Programming A programming style emphasizing the use of condition / action rules for which the existence of the condition will cause the performance of the action.

Default A default rule expresses an conditional implication that applies only when its consequent is consistent with current state of the knowledge base.

Defined Concept A defined concept is a concept that is described in terms of one or more other concepts and/or relations. All concepts that are not primitive are considered to be defined concepts.

Definition A definition binds a predicate symbol to a description. The LOOM knowledge base object representing this definition is called a concept or relation.

Description A description is a lambda expression that defines a class of individuals or tuples. LOOM concepts are an example of descriptions and are analogous to frames or schemata in other knowledge representation systems.

Description Language The term description language refers to the set of LOOM constructs used to represent declarative knowledge. LOOM's description language provides a principled means for describing much of the knowledge that is commonly associated with the frame component of a knowledge representation tool. The description language in LOOM is designed to facilitate the construction of expressions that describe classes of individuals.

Dispatching A dispatch mechanism has the function of filtering a set of possible actions and selecting a subset (typically a singleton) for execution.

Domain The first argument of a binary predicate.

Fact In LOOM a fact asserts either an instantiation relationship between a concept and an individual or a relationship between two individuals.

Filler An instance of a concept that is pointed to by a role within another concept (See Role).

Implication An implication relationship defines a non-definitional constraint, i.e., one that must be true for some particular model or models, but is not necessarily true in all models. In LOOM, the `implies` operator permits one to state conditions for a concept that are necessary but not sufficient (or vice versa).

Instantiation, Instance The term instance refers to a knowledge base object that represents a single individual. The process of creating an instance of a representational class or concept is called instantiation. For example if `Widget` is a defined concept in LOOM, the process of representing a particular widget and giving it a name such as `Widget-1` is the process of instantiation.

Interval A set for which there is total ordering of "<" over members of the set.

Method An element of procedural code attached to an object. Methods are selected for execution via a dispatching mechanism (typically message-passing or a generalization upon it).

Model-Driven Programming A programming style in which the declarative knowledge present in an application program (the model) actively controls the behavior of a running application.

Modelling Language The subset of LOOM used to specify definitions, rules, and facts.

Object-Oriented Programming A programming style which represents information in frame-like data objects arranged in a hierarchy of inheritance relationships with procedural methods that are activated by a message passing paradigm. The use of message passing and inheritance typically characterize object-oriented programming.

Open-World Semantics LOOM assumes an open-world semantics. Under this assumption, if LOOM cannot prove or disprove a proposition, then it assigns that proposition the value `UNKNOWN` (See also Closed-World Semantics).

Pattern-Directed Dispatching Pattern-directed dispatching represents a generalization of the message-passing mechanism found in object-oriented languages. In pattern-directed dispatching, a pattern that defines the situation for a LOOM method typically

references one or more of the method's formal parameters, so that the choice of which action methods to invoke is sensitive to the types and attributes of the objects passed as actual parameters to an action.

Primitive Concept A concept or relation is primitive if it cannot be completely characterized in terms of other concepts (relations) and is unique.

Production Rule An condition/action pair.

Query Language The interface to LOOM's deductive query facility, provided by the `retrieve` and `ask` constructs. The query is an argument to either of these functions and may be an arbitrary expression in the first-order predicate calculus.

Range The second argument of a binary predicate, or last argument of a n-ary predicate.

Recognizer A recognizer is a assertional reasoner whose function is to maintain the current values for the types of all individuals in a knowledge base. LOOM refers to its recognizer as "the matcher".

Relation A relation establishes a semantically instantiated relationship between elements of represented knowledge. A relationship between two concepts is a binary relation and is the most common usage of the term relation.

Retraction The withdrawing of a particular previously asserted fact.

Role The term role is often used to refer to a particular relation that appears within a concept expression. A role is a semantic link to other concept instances which are called the fillers of the role. In other words, a role establishes a mapping between an instance of that concept and a set of instances (fillers) that fill that role.

Role Chain A role chain refers to a binary relation, defined as the composition of other binary relations, that is attached to some concept or instance.

Role-Relating Concept A role-relating concept expression specifies a relationship which constrains the fillers of two or more of the concept's roles. For example, the expression `(= input-voltage output-voltage)` specifies that the fillers of the roles input-voltage and output-voltage have the same value.

Role-Restricting Concept A role-restricting concept is a concept defined by a restriction placed on a particular relation. For example, the expression `(:at-least 2 child)` denotes the concept for which the role "child" must have at least two fillers. Both role-restricting and role-relating concepts are types of constraints.

Set In LOOM, a set is a specialized kind of concept that identifies a group of constants. The set may be a finite or an infinite group of atomic individuals. Examples of constants are the color red or the number 5.

Situation A set of states described as a pattern that indicates when a particular method is applicable.

Subsumption Subsumption describes the relative generality of two concepts. A concept `A` subsumes a concept `B` if the definitions of `A` and `B` logically imply that members of `B` must also be members of `A`.

Task A binding between a LOOM action and a set of arguments.

Term Classification Term Classification is an equivalent way of describing Classification-Based Knowledge Representation.

Terminological Language A terminological language is designed to facilitate the construction of expressions that describe classes of individuals.

Terminological Reasoning The ability to represent knowledge about the defining characteristics of concepts, and to use that knowledge to automatically infer and maintain a complete and accurate taxonomic lattice of logical subsumption relations between concepts.

Truth-Maintenance Mechanism The function of a truth-maintenance system (TMS) is to ensure that at any point in time all facts inferrable by the matcher have been inferred, and that logical support exists for each inferred fact. LOOM performs truth maintenance as a byproduct of its instance-matching process.

Type In LOOM the type of a database object is held as a dynamic expression which is maintained by the pattern matcher. The matcher keeps track of all `instantiates` relationships between an individual object and the concepts whose definition matches. The intersection of all concepts matched by a particular individual is called the `type` of the individual.

Unification, Unify Unification is the formal term for the operation that combines the structural components of facts and rules. LOOM uses the classifier to perform unification over descriptions. The unification routine in LOOM places fewer restrictions on the syntax of unifiable facts and rules than in typical current generation systems.

Index