

Loom Tutorial

For **Loom** Release 2.1
Release Date: May 17, 1995

Document Version 2.1.0

Chapter 1 — The Basics of Using Loom	2
1.1 Getting Loom Started.....	2
1.2 Defining a New Knowledge Base.....	3
1.3 Viewing the Knowledge Base.....	4
1.4 Viewing More Complex Objects.....	5
1.5 Deleting and Changing Objects.....	8
1.6 Retrieval from the Knowledge Base.....	10
1.7 Saving and Restoring Loom Knowledge Bases.....	11
Chapter 2 — Concept Definitions and Classification-based Reasoning ...	12
2.1 Creating Instances of Objects.....	12
2.2 Classification-based Reasoning in Loom.....	13
2.3 Basic Mechanisms for Defining Objects.....	15
2.4 More on Object Definition: Primitive Concepts.....	17
Chapter 3 — More Complex Knowledge Base Definitions	19
3.1 The Domain Example: A Toy Problem in Crisis Planning.....	19
3.2 Defined (Non-Primitive) Concepts.....	21
3.3 Complex Knowledge Base Queries: RETRIEVE.....	24
3.4 The Use of Constraints in Concept Definition.....	25
3.5 Relations Between User-Defined Concepts.....	26
3.6 A More Complex Example of Defined Concepts.....	27
3.7 Non-Numeric Constraints in Concept Definition.....	28
3.8 Using Backward-Chaining Inference in Concept Matching.....	29
3.9 More Complex Terms in Concept And Relation Definition.....	29
Chapter 4 — Behavioral Programming: Actions and Methods	31
4.1 More on Object Specification.....	31
4.2 Basic Action Definition.....	33
4.3 Activating Methods.....	34
4.4 Defining Actions with Multiple Methods.....	34
4.5 Defining Actions with Different Filters.....	35
4.6 Knowledge Base Access within Methods.....	35
4.7 Actions Calling Actions.....	36
Chapter 5 — Behavioral Programming: Productions	39
5.1 Defining a Production.....	39
5.2 Executing a Production.....	42

The Loom Tutorial

This document is written for the beginning Loom user. It is intended to provide a hands-on introduction to the basic features in Loom and how they can be used to build systems. This document assumes a reader familiar with the CommonLisp language and environment, and having an awareness of the basic notions of knowledge representation and knowledge inference. The exercises given in this tutorial will require an installed and runnable Loom system.

Introduction

Loom provides both a high level programming language and an environment for knowledge based system construction. When compared to other knowledge representation systems, Loom is unique in several respects. Perhaps most significant of its characteristics is its support for a broad range of programming metaphors. Loom supports a “description” language for modelling objects and relationships. Procedural programming is supported through pattern-directed methods, while production-based and classification-based inference capabilities support deductive reasoning. All of these capabilities reside in a framework of query-based assertion and retrieval. This document is designed to work together with **Loom version 2.1**

For those familiar with the description logic branch of the knowledge representation research community, the concepts and features of Loom are old friends: Loom represents the next rung in an evolutionary ladder of knowledge representation system development. For others, Loom can initially present a formidable array of features. Even experienced knowledge system developers may at first have difficulty understanding just what the features of Loom provide and how to organize solutions to application problems using Loom.

This tutorial is intended to help the beginning Loom user overcome the initial problems of understanding what Loom features are available, how they can be applied to structure solutions to problems, and how they can be manipulated in the Loom programming environment. The examples given in this tutorial have been designed to demonstrate specific features in Loom, as clearly and as simply as possible. These basic constructs and operations will enable the user to build realistically complex application knowledge bases.

Chapter 1 — The Basics of Using Loom

This chapter will introduce the basic mechanics of interacting with the Loom environment.

The Loom tutorial has been developed to guide an exploration of Loom's features and capabilities. It is intended that the user participate in this activity by actually typing and executing the examples provided. While there is some tedium in typing examples, much of the familiarity and expertise in Loom's use will directly derive from this participation. Besides the basic benefit of "learning by doing," this approach leads the user to face the problems that occur when typos and similar errors are introduced. This tutorial offers opportunities for the user to customize and extend the basic lessons, and poses problems for the user to "flesh out" more fully the example problem's development. These suggested exercises are denoted by the use of a • (bullet). Should the user reach an impasse or wish to expedite the input process, (s)he may refer to the file "tutorial.lisp", which contains all the input for the examples and solutions for the exercises in this tutorial.

1.1 Getting Loom Started

The specific means of invoking a CommonLisp environment and loading Loom may vary greatly across machines. It may be necessary to contact the person who installed CommonLisp and Loom on your specific system for help. The Loom Installation Guide contains detailed, environment-specific instructions for creating a Loom environment from scratch. Lucid CommonLisp was used for the Loom image in this tutorial; you may see some slight differences in the system output or in the exact lisp commands if you are using a different system.

Start Loom appropriately for your system environment.

1.2 Defining a New Knowledge Base

We are about to address the first step in constructing an actual Loom knowledge base. The knowledge base that will be developed through the course of this tutorial focuses on the problem of planning a crisis response mission. This knowledge base will allow the user to specify characteristics about the mission requirements and the available resources, and will in return prepare a set of orders to marshal a coordinated crisis response. Naturally, this problem domain will be much simplified and idealized to meet the objectives of this tutorial.

There are four steps necessary to cleanly create and start working in a new knowledge base. The first four steps look like this: (Note that your typed input is shown in **boldface**, while the system-generated type is shown in standard face).

```

> (make-package "CRISIS-PLANNER")
#<Package "CRISIS-PLANNER" 405DB196>
> (in-package "CRISIS-PLANNER")
#<Package "CRISIS-PLANNER" 405DB196>
> (loom:use-loom "CRISIS-PLANNER")
|TH|CRISIS-PLANNER-THEORY
;;; Substitute your favorite path name in the next call
> (defkb "CRISIS-PLANNER-KB"
      nil :pathname "~/Loom/crisis-saved-kb")
|K|LOOM::CRISIS-PLANNER-KB

```

With the first command, we built a new package in the CommonLisp environment for this system. Remember that strings in CommonLisp are case-sensitive. Packages provide a mechanism for defining lexical context in LISP; further discussion is available in any of the CommonLisp manuals. Having defined this new package, we set the current package to it and then we called the **use-loom** function. This accomplished two things. First, all symbols needed to interact with Loom (e.g. function names, data structure variables, etc.) were imported into the package with the name specified. Second, **use-loom** created a new Loom context (theory) and knowledge base; later additional knowledge bases may be defined within this package. Finally, we call **defkb** to provide additional information about the knowledge base we just defined. **defkb** has two required arguments, the name of the knowledge base and the parents of the knowledge base (in this particular case, the crisis-planner knowledge base has no parents). The keyword argument **:pathname** specifies a path name for use in saving and restoring the new knowledge base. This parameter should be set to the path name you wish the knowledge base to be stored in; the path name in the example is supplied for illustrative purposes only. Information in Loom is organized into hierarchical “contexts”. Knowledge bases are used to add additional information and structure to the contexts.

The next step is simply to tell Loom which knowledge base should be the “current” knowledge base. This also makes the context “CRISIS-PLANNER-THEORY” the current context:

```

> (change-kb "CRISIS-PLANNER-KB")
|K|LOOM::CRISIS-PLANNER-KB

```

Since the goal of this tutorial is to demonstrate the widest extent of Loom’s reasoning, we need to use the most complete. That means selecting “classified” instances instead of Loom’s default of

“lite” instances. The following function call establishes the default creation type for the instances that will be used in this tutorial. (Lite instances do not have as much inference done, but they have a smaller storage footprint and should be quicker for most operations than classified instances).

```
> (creation-policy :classified-instance)
:CLASSIFIED-INSTANCE
```

1.3 Viewing the Knowledge Base

The interface to Loom is an application program interface that includes user-friendly commands. Each Loom command is basically a LISP S-expression which evaluates to something meaningful because it was defined as a function or macro in the Loom package, and has been imported into your working package. This interface offers some advantages. First, this form of interface is portable across environments. Second, such an interface allows Loom to essentially work as an extension to the LISP environment. Loom doesn't really care whether the query language utterances come from your keyboard, from some LISP functions that were built with embedded Loom queries, or more complex functions or macros that actually synthesize Loom queries at run time. As you interact directly with Loom in this tutorial, keep in mind that the steps you are taking might ultimately be examples of programming instructions in your own applications.

There are a few basic ways of interacting with Loom: functions that let you *examine* the Loom knowledge base, functions that let you *tell* things to the knowledge base, and functions that let you *ask* things about the knowledge base. Different examples of these functions are appropriate for talking to Loom about different types and complexities of knowledge base entries. This section will introduce a few of the simple and commonly used mechanisms. More complex examples of these interactions will be scattered throughout this tutorial as the example knowledge base grows more complex.

To begin, look at the current state of the knowledge base by examining the contents of its context:

```
> (list-context)
NIL
```

Naturally, since we have a newly initialized knowledge base, it is empty. We can begin to inform the knowledge base about objects that exist in the world, using the **create** function:

```
> (create 'ob-1 nil)
```

|I|OB-1

What you have just done is to tell Loom that there is a thing in the world, called **ob-1**. The first argument to **create** is a symbol or a string specifying the name of an object. The second argument is the type of the object. Since we don't care about the type of **ob-1**, we specify **nil**. Loom labels symbols representing knowledge base entries with a prefix enclosed in vertical bars: **|C|** for concepts, **|I|** or **|i|** for instances, **|R|** for relations, and **|K|** for knowledge bases. Another form of inference that Loom does when you tell it about something is to match it to the things it already knows about. Since we haven't yet embodied this knowledge base with any models of typical objects, **ob-1** is "classified" as an instance of **Thing**, the most general concept.

1.4 Viewing More Complex Objects

The instance of type **Thing** we have defined so far is not very interesting. Since the Loom knowledge base has not been told about any concepts more complex than **Thing**, there is no way to describe an instance of any more complex types. The definition of concepts and relations is covered more fully in Chapter 2, but let's create a simple knowledge base to demonstrate some of the ways to interact with Loom. (Recall that a "bullet" (•) indicates an exercise for the user.)

- List the knowledge base. Note the current set of entries.

Now tell the knowledge base about some basic entities in our air strike planner domain. Let's begin by defining a military-installation as a basic object in our domain. Then, we can define an air base, a type of a military installation that has a name and at least one runway length (obviously, this concept is simplified for the example). In Loom, this means we want a concept that has exactly one **name** relation to some other object (an instance of a **string** concept that contains the name), and at least one **runway-length** relation to some object that describes that concept.

```
> (defconcept military-installation)
.+
|C|MILITARY-INSTALLATION
> (defconcept air-base
   :is-primitive
   (and military-installation
        (exactly 1 name)
        (at-least 1 runway-length)))
|C|AIR-BASE
```

First, notice the output after the first **defconcept**. This output is typical of that generated when Loom accepts new knowledge base specifications and attempts to modify the knowledge base in response. The symbols tell the informed Loom observer (which you will become after reading later chapters in this tutorial) when Loom is classifying concepts (**.**), compiling constraints (**+**), compiling access functions (**!**), merging concepts (**M**), unclassifying (**-**), and classifying instances (*****). Don't worry if the number and order of the symbols on your system differs from that in this tutorial.

Now, if you list the knowledge base, you will see some new entries. More about those in a minute. There is a mechanism for examining concepts, relations, and instances in detail. The **pc** ("print concept") macro, for example, lets you look at a concept. Its only argument is the concept object's name.

```
> (pc air-base)
(defconcept Air-Base
  :is-primitive
    (:and Military-Installation
      (:exactly 1 Name)
      (:at-least 1 Runway-Length))
  :annotations (Thing))
```

Similarly, you can use **pr** for viewing relations, **pi** for looking at instances, **pb** for looking at actions or production rules (behaviors), and a more general macro, **po**, for looking at any type of Loom object. All of these macros except **po** will work with either an object or its name. **PO** must have an object.

Back to the knowledge base entries. You probably noticed that in addition to the concept **air-base**, two relations are also present. The existence of these relations was implied by their reference in the concept definition. If you examine these (and please do, using **pr**), you will see that they are system-defined placeholders. For now, take it on faith that these are not sufficient for the knowledge base to work. We must define these two *relations*. **Name** is intended to be a relation that relates any concept (by leaving the domain unspecified, it will default to the most general concept **Thing**) to a single **string**. **Runway-length** relates an airbase to one or more numbers (representing lengths of runways). In this case, the relation is meaningful only to concepts of type **air-base**.

```
> (defrelation name
  :range string)
.
|R|NAME
```

```
> (defrelation runway-length
      :domain air-base
      :range number)
..+++
|R|RUNWAY-LENGTH
```

- Use **pr** to view these objects.

Now we can create some instances of the air base concept we have defined. **Tellm** and its partner function **tell** will be explained in later chapters. For now, we will just create a few instances:

```
> (tellm (create ab-1 air-base)
      (name ab-1 "Atlanta NAS")
      (runway-length ab-1 12050)))

!!!!!!*****
Recognition changes at agent time 1:
entry: AB-1      |C|AIR-BASE
entry: AB-1      |C|MILITARY-INSTALLATION
2
```

You have just created an instance labeled **ab-1** of type **air-base** and given **ab-1** a **name** and a **runway-length**. Loom is able to infer that **ab-1** is a **military-installation** as well as an **air-base**, since by definition an **air-base** is a **military-installation**. **Tellm** returns 2, the number of the current knowledge base state. Note that since **ab-1** was created to be of type **air-base**, it will not be possible to retract this fact. The type used in the **create** function is treated differently by Loom than a type used in an assertion. The latter types can be retracted, while the creation type cannot.

- Try examining this instance with the **pi** function.

1.5 Deleting and Changing Objects

Let's experiment a bit with a new air base, ab-2. First, let's create the instance, and list the knowledge base to confirm its existence.

```
> (tellm (create ab-2 air-base)
      (name ab-2 "Mainland NAS")
      (runway-length ab-2 9000))
```

```

Recognition changes at agent time 2:
  entry: AB-2      |C|MILITARY-INSTALLATION
  entry: AB-2      |C|AIR-BASE
3
> (list-context)
(|I|OB-1 |I|AB-2 |I|AB-1 |C|AIR-BASE |C|MILITARY-INSTALLATION |R|NAME
|R|RUNWAY-LENGTH)

```

Now, let's tell Loom about some new runway lengths, and then print the instance. Note that we can use **tellm** to specify new values for instance relations:

```

> (tellm (runway-length ab-2 9000))
3
> (tellm (runway-length ab-2 10000))
4
> (pi ab-2)
(TELL (:ABOUT AB-2
      AIR-BASE
      (RUNWAY-LENGTH 9000)
      (RUNWAY-LENGTH 10000)
      (NAME "Mainland NAS")))

```

Note that Loom added the second **runway-length**, but not the first. This is because the the first **runway-length** is the same as an already existing runway length, so Loom assumes that you are merely repeating information that is already present in the knowledge base.

We can remove information from the knowledge base using **forgetm**. The syntax of **forgetm** is identical to **tellm**:

```

> (forgetm (runway-length ab-2 10000))
5
> (pi ab-2)
(TELL (:ABOUT AB-2
      AIR-BASE
      (RUNWAY-LENGTH 9000)
      (NAME "Mainland NAS")))

```

We can completely remove an instance from the knowledge base by using the **forget-all-about** function. This function requires either the instance *object*, or its identifier.

```
> (forget-all-about 'ab-2)
T
```

- List the knowledge base to verify the deletion.

Now, let's re-tell Loom about ab-2.

```
> (tellm (create ab-2 air-base)
      (name ab-2 "Mainland AFB")
      (runway-length ab-2 9000))
!!!!***
Recognition changes at agent time 6:
entry: AB-2      |C|MILITARY-INSTALLATION
entry: AB-2      |C|AIR-BASE
7
```

Finally, there is the problem of changing concepts and relations. The Loom knowledge base will respond to redefinitions of concepts and relations. For example, let's redefine an air-base and add a new relation.

```
> (defconcept air-base :is-primitive
      (and Military-Installation
            (exactly 1 name)
            (at-least 1 runway-length)
            (exactly 1 service-branch)))
-
|C|AIR-BASE
> (defset us-service-branch :is
      (one-of 'army 'air-force 'navy 'marine))
.
|C|US-SERVICE-BRANCH(ONE-OF 'ARMY 'AIR-FORCE 'NAVY 'MARINE)
> (defrelation service-branch
      :domain air-base
      :range us-service-branch)
.
|R|SERVICE-BRANCH
```

Notice the use of **defset**. **defset** is used to define a set of symbolic items. The items may be either ordered or unordered (in this particular case, they are unordered). Notice also that **defset** returns a concept; a set is a special kind of concept.

1.6 Retrieval from the Knowledge Base

All of this manipulation of instances does little good unless a mechanism is available to query the knowledge base about its state. This mechanism is provided through the **retrieve** function. Retrieval will be discussed more fully in later chapters, but for just a taste, try the following:

```
> (retrieve ?p (air-base ?p))
(|I|AB-1 |I|AB-2)
```

This retrieves all instances of an air base.

```
> (retrieve ?p (and (air-base ?p)
                    (for-some (?1) (and (runway-length ?p ?1)
                                        (> ?1 10000))))))
(|I|AB-1)
```

This case only returns air bases with at least one runway longer than 10000.

1.7 Saving and Restoring Loom Knowledge Bases

Saving the contents of a Loom knowledge base is accomplished as follows:

```
> (save-kb)

|C|US-SERVICE-BRANCH(ONE-OF (QUOTE ARMY) (QUOTE AIR-FORCE) (QUOTE NAVY)
  (QUOTE MARINE))
|R|NAME
|R|RUNWAY-LENGTH
|R|SERVICE-BRANCH
|C|AIR-BASE
|C|MILITARY-INSTALLATION
|I|OB-1
|I|AB-2
|I|AB-1
"~/Loom/crisis-saved-kb"
```

The knowledge base saved is the current one, selected with the **change-kb** (or **in-kb**) expression. The default can be overridden by supplying a knowledge base name as an optional variable. The path name for the save file is generated from the name given earlier in the

defkb expression. An alternate path name could have been specified at the time that the knowledge base was created, or it can be specified in the **save-kb** command as the value for the keyword argument **:pathname**.

Restoring the knowledge base is accomplished through the **load-kb** function, which requires the knowledge base name, and also offers the optional **:pathname** argument.

Chapter 2 — Concept Definitions and Classification-based Reasoning

This chapter will explore the basic means by which definitions and instances of concepts and relations are added to the Loom knowledge base, and how those representations are used to perform classification-based inference.

Earlier, in Chapter 1, we had an initial exposure to the **defconcept** and **defrelation** macros, the basic mechanisms by which knowledge structures are declared in Loom. We also caused an instance of an object to be created in the Loom knowledge base, and experimented a bit with ways to manipulate and view that object. This chapter of the Loom Tutorial will focus on a more complete picture of concepts and relations, and the ways that complex knowledge representation structures can be defined using these elements.

There are a number of variations in the way Loom knowledge base entries can be defined. Each variation allows the knowledge base developer to exploit specific features of Loom's inference capability. As definitions are added to the knowledge base, Loom performs a certain class of inference called *classification*. In classification, the definitions of concepts and relations are organized to form a domain-specific taxonomy of terms. This taxonomy is structured in accordance with the subsumption (superclass/subclass) relationships that exist between terms. When instances of objects are created in the knowledge base, they are classified into that taxonomy, either automatically or on request.

Since there are so many types of activity within Loom initiated by the definitions it is given, it is impossible to clearly separate examples of definition from examples of classification-based inference. For that reason, this chapter of the Loom Tutorial will explore both the declaration of objects in the modelling language and the related classification inferencing.

2.1 Creating Instances of Objects

The **tell** and **tellm** functions are used to tell the knowledge base about instances of defined concepts that exist in the world. There are a number of syntactic variations that cause equivalent kinds of behavior. One variation is in the use of **tell** vs. **tellm**. **Tell** is used to provide information to the knowledge base. **Tellm** performs the same function, and then tells Loom to update the state of the knowledge base. The **tellm** operation causes the knowledge base to perform any inferences it can based on the new information queued since the last state change. In the first example, we will tell the knowledge base about an instance of concept, and

about its role fillers. We will do each with explicit statements. At the end, we will cause Loom to update the state of the knowledge base.

```
> (tell (create ab-3 air-base))
OK
> (tell (runway-length ab-3 14000))
OK
> (tellm (name ab-3 "Centrada AFB"))
*
Recognition changes at agent time 7:
  entry: AB-3      |C|MILITARY-INSTALLATION
  entry: AB-3      |C|AIR-BASE
8
```

We could have done the same exercise using **tellm** for each statement. The only difference is that after each statement, the knowledge base would have been updated. In some circumstances, this causes Loom to make inferences that will change (or just be redundantly recomputed) as more information is given. Loom will always infer the “right things” based on its current information.

An alternative syntax would have embedded all these statements inside a single **tellm** statement. Try this yourself:

- Create an instance called **ab-4** with **name** “Nearby Shores NAS” and **runway-length** 9000.

Note how you were forced to redundantly type the instance name for each statement. This can be avoided by using the **about** construct inside your **tell** statements. This is illustrated in the next section.

2.2 Classification-based Reasoning in Loom

Let's create a new concept **small-air-base**, which is an air base with only 1 runway.

```
> (defconcept small-air-base
  :is (and air-base
        (exactly 1 runway-length)))
.+
|C|SMALL-AIR-BASE
```

Note the use of **:is** instead of **:is-primitive** as in our previous concept definitions. Since **small-air-base** is a “fully-characterizable” type of **air-base**, it is naturally conceptualized as a non-primitive concept. Primitive and non-primitive concepts will be discussed in more detail in a later section, but for now it is only important to realize that such a distinction exists. Now, let’s tell the knowledge base about an instance of an object which, like a small air base, has one **runway-length**. We will use yet another variation of the **tell** construct, even though any of our previous methods would work here as well.

```
> (tellm (about ab-5
          (runway-length 1100)))
!*
Recognition changes at agent time 9:
entry: AB-5      |C|MILITARY-INSTALLATION
entry: AB-5      |C|AIR-BASE
*
10
```

Notice that Loom has classified this instance as an **air-base** only, and has not classified the instance as a **small-air-base**, even though we have specified only one **runway-length**. **Ask** is a function that lets us pose queries to the knowledge base, in forms that in most ways resemble **tell** constructs. An **ask** will return either **NIL**, if the query is false or unknown, or **T** if the query is true. Let’s use the **ask** function to inquire whether **ab-5** is really a **small-air-base**:

```
> (ask (air-base ab-5))
T
> (ask (small-air-base ab-5))
NIL
```

Loom has no way of knowing that there are not additional facts of which it has not been told. In short, Loom assumes *open world semantics*. For example, nothing we have stated so far precludes the possibility that **ab-5** has more than one **runway-length**. Loom allows the user to define relations that assume *closed world semantics*. The inclusion of the **:closed-world** characteristic will affect only the objects to which it is added (relations or concepts); the rest of the knowledge base remains open world. The closed world assumption allows Loom to assume that what it has been told is complete with respect to that concept or relation.

Let’s redefine the relations in this example to use closed-world semantics:

```
> (defrelation runway-length
    :domain air-base
```

```

      :range number
      :characteristics :closed-world)
----...++++
|R|RUNWAY-LENGTH

```

- Use the **ask** function as before to see if this object can be classified as a **small-air-base**.

As you can see, Loom has been able to make a non-trivial inference to classify the instance as a **small-air-base** based on its roles, given the assumption of closed-world semantics.

The examples we will build later in this tutorial will draw heavily on this ability, and show how sophisticated and very useful classification based reasoning can be. In the meantime, let's continue to tease out some of the issues related to basic object definition and classification.

2.3 Basic Mechanisms for Defining Objects

We have seen how the **defconcept** and **defrelation** macros are used to define objects representing concepts and relations. We will explore a number of variations in the way these macros can be used. Let us begin by defining a fairly simple concept, **surveillance-plane**.

```

> (defconcept surveillance-plane :is
      (and airplane
            (at-least 1 surveillance-device)))
|C|SURVEILLANCE-PLANE
.+

```

We have described a **surveillance-plane** in terms of the concept **airplane** that it specializes, and in terms of *roles* (similar to *slots* in other languages) whose existence distinguishes this concept from other concepts. We have specified that “a surveillance-plane is an airplane with at least one filler of the role surveillance-device.” The key word in that phrase is the word **is**; this form of a concept definition is a fully-specified definition. We will explore a few variations that qualify that definition.

Our definition of a **surveillance-plane** is not quite complete. We have defined a simple concept, but we have yet to define how its roles can be satisfied. If we were to try to build an instance at this time, something like this would happen:

```

> (tellm (about a1))
>>Error: Attempt to seal the network when undefined concepts exist.

```

For example, `|C|AIRPLANE` is undefined.
 Execute ``(list-undefined-concepts)'` to retrieve all undefined concepts.

```
LOOM::WARN-OF-UNCLASSIFIED-CONCEPTS-DURING-SEALING:
  :C 0: Resume running LOOM
  :A 1: Abort to Lisp Top Level
-> :a
```

As the error message suggests, we can ask the knowledge base about objects whose definitions are not complete:

```
> (list-undefined-concepts)
(|C|AIRPLANE |R|SURVEILLANCE-DEVICE)
```

Remember, what Loom does with concept and relation definitions is to build a domain-specific taxonomy that can be used to classify any instance. Note that Loom generated an error without regard to the statement in the **tellm**. We did not specify the new instance to be related in any way to the concepts we were recently defining; the error was simply attempting to update the state of the knowledge base while there were unclassified concepts. Unclassified concepts are those for which insufficient information has been provided to allow their inclusion in the taxonomy: the knowledge base is “unsealable” because a consistent taxonomy cannot be built from these unclassified concepts. We must, in this case, complete the definitions for **airplane** and **surveillance-device** before the concept **surveillance-plane** can be fully classified.

```
> (defconcept airplane)
..
|C|AIRPLANE
> (defrelation surveillance-device
   :range surveillance-hardware
   :characteristics :closed-world)
..
|R|SURVEILLANCE-DEVICE

> (defset surveillance-hardware :is
   (one-of 'radar 'infrared-spy-camera
           'subsonic-message-unscrambler))
..++++
|C|SURVEILLANCE-HARDWARE(ONE-OF 'RADAR 'INFRARED-SPY-CAMERA
                                'SUBSONIC-MESSAGE-UNSCRAMBLER)
```

SURVEILLANCE-HARDWARE

- Check to verify that no unclassified concepts still exist.

2.4 More on Object Definition: Primitive Concepts

To use Loom correctly and efficiently, it is important that one understands Loom's notion of primitive concepts and relations. First, we will briefly explain what it means for a concept or relation to be primitive. Then, we will try some examples to see how Loom handles "primitiveness."

A concept or relation is *primitive* if its definition is incompletely specified. By declaring a concept to be primitive, we are in effect telling Loom there are hidden attributes about objects of that type that we are not representing. The choice of which concepts to be declared primitive depends largely on the intended use of the represented concept and will vary across applications. For example, in an application concerned with the migratory patterns of birds, the concept **bird** may need no further definition and be considered primitive, while **raptor** may be defined as "a bird that is a predator". If the application is concerned instead with representing relationships and interactions within an ecosystem, it may be important to know that a **bird** is "an animal which has wings and reproduces by laying eggs" and the concepts **animal**, **egg**, and **wing** are primitive. Concepts composed of combinations and specializations of other concepts, such as bird in the latter case, are called *defined* concepts.

Let's see how changing the primitiveness of the concept affects Loom inferences. First, tell Loom about an instance which matches the **surveillance-plane** definition.

```
> (tellm (create a1 airplane) (surveillance-device a1 `radar))
!!!*
Recognition changes at agent time 11:
  entry:  A1          |C|AIRPLANE
  entry:  A1          |C|SURVEILLANCE-PLANE
***
12
```

As expected, Loom correctly classifies the instances as an **surveillance-plane**. Now, let's redefine **surveillance-plane** to be a primitive concept, and then tell Loom to update the state of the knowledge base.

```
> (defconcept surveillance-plane :is-primitive
      (and airplane
```

```
                (at-least 1 surveillance-device)))
-.+
|C|SURVEILLANCE-PLANE
> (tellm)
!*
Recognition changes at agent time 12:
  exit:  A1          |C|SURVEILLANCE-PLANE
*
13
```

Loom has reclassified **a1** as an **airplane**. Note that **airplane** is implicitly primitive, as are all “top-level” concepts. If we would like Loom to classify **a1** as an **surveillance-plane**, we must directly assert this fact.

```
> (tellm (surveillance-plane a1))
*
Recognition changes at agent time 13:
  entry:  A1          |C|SURVEILLANCE-PLANE
14
```

Chapter 3 — More Complex Knowledge Base Definitions

This chapter will explore the more complex mechanisms available to specify object definitions in the Loom knowledge base. It will also explore the classification-based inference capabilities enabled by these more complex representations.

In the previous chapter, we explored the basic mechanisms by which objects and relations are defined in Loom, how objects based on those definitions are instantiated, and how Loom performs classification-based reasoning to infer a mapping of those objects to a domain-specific taxonomy. This chapter will expand on those basic ideas by presenting a more complete view of Loom's definitional constructs and related inferential capabilities.

While Chapter 2 introduced the power of the basic Loom approach, Chapter 3 will expose some of the richness and breadth of knowledge representation possible. Much of this breadth comes from aspects of concept and relation definition not yet seen: the use of constraints, and a more complete set of definitional terms. We will also explore *defined* concepts and relations, where new knowledge base definitions are defined as compositions and specializations of existing definitions. Finally, we will start to look at the use of backward chaining inference to determine object classification. Sprinkled through this chapter will be discussions of the mechanisms for retrieval of information from the knowledge base.

3.1 The Domain Example: A Toy Problem in Crisis Planning

To add clarity to the examples used in this and later chapters, we discuss our problem domain. Crisis planning is a task faced by military planners who develop transportation plans for the deployment of forces and resources. In fact, a large and complex organizational hierarchy is necessary to master the planning complexity and to capture the expertise needed to meet military transportation planning requirements. Planning in a crisis poses one of the most difficult planning tasks, forcing planning experts to respond to unexpected and rapidly changing situations. They must quickly develop transportation plans with little time for deliberation.

In this tutorial, we will be building examples related to a crisis planning problem. Among the objectives of our tutorial is the construction of "realistic" examples of the way Loom knowledge representation features can be used to construct knowledge based solutions to real world problems. Against this objective is the desire to pose clear, simple examples that make Loom's features as easy as possible to understand. We provide examples based on a toy version of a crisis planning problem, designed to illustrate the scope of knowledge based representation that

is possible in Loom. We have designed this toy problem around common sense knowledge, avoiding the problem of training the reader in the details of real world crisis planning.

Our toy domain is about planning transportation for an emergency relief effort. In general, our knowledge base will know about things like roads, airports, and cities. Air, ground, and sea transportation modes will be represented. Knowledge will be provided to develop an appropriate set of actions to transport construction and medical teams to appropriate locations in the face of damaged or otherwise limited transportation facilities.

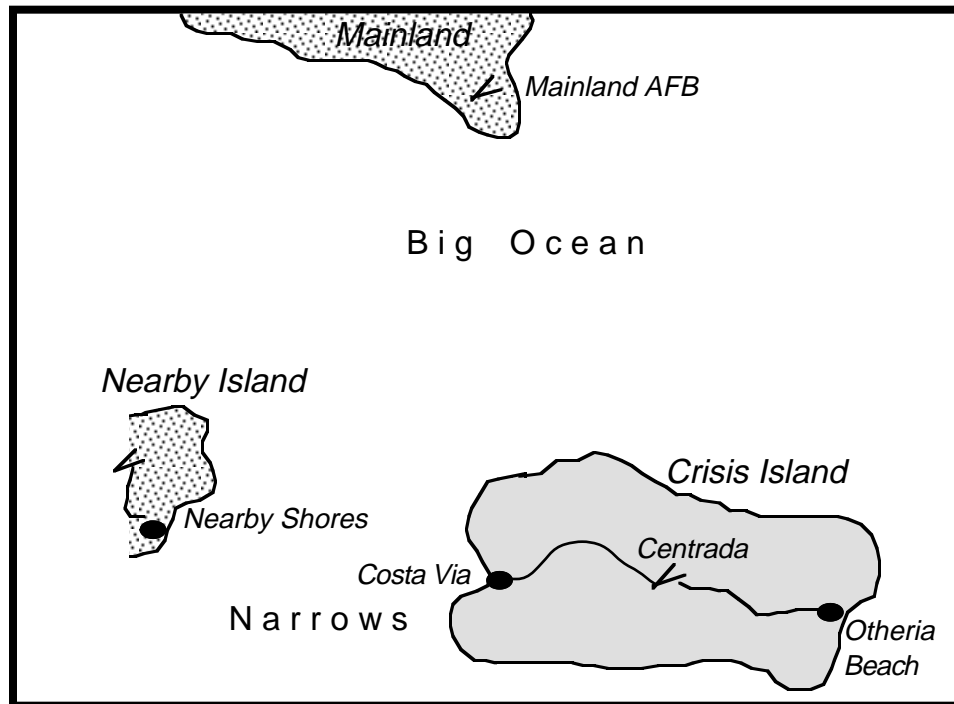


Figure 1: The Crisis Island World

To provide a bit more context for our examples, we have invented a mythical world that contains the basic elements of an interesting crisis planning problem. The instances of objects used in our examples are taken from this world (see map, Fig. 1). In our toy problem, Crisis Island, someplace in the Caribbean, has been hit by a natural disaster: roads, airport, and sea ports may have some level of damage, and cities may have some need for medical relief. Our job is to get medical units deployed to these cities despite the limited transportation facilities. In some cases, this may mean deploying construction teams ahead of the medical teams to restore roadways, repair port facilities, etc. It may also mean utilizing undamaged transportation facilities at Nearby Island.

Of course, in this tutorial we will only address a narrow range of issues related to creating such a crisis planning knowledge base. The specific pieces of knowledge will be discussed as they arise in examples. Our goal is to show examples of how Loom's most important features

work, both individually and in concert, to support the representation and manipulation of knowledge.

3.2 Defined (Non-Primitive) Concepts

Let us begin by defining a primitive concept in our domain, **location**. Three relations are associated with a **location**: a **latitude**, a **longitude**, and a **region**.

```
> (defconcept location)
.+
|C|LOCATION
> (defrelation latitude :domain location :range number)
.+
|R|LATITUDE
> (defrelation longitude :domain location :range number)
.+
|R|LONGITUDE
> (defrelation region :domain location :range string)
.+
|R|REGION
```

Notice that we did not explicitly identify the above relations in **location**'s concept definition. Instead, the relations are associated with **location** by virtue of the **:domain** field in the relation definitions. This is an example of *indirect role definition*.

Let's tell Loom about 3 different locations in Crisis Island.

```
> (tellm (create c1 location)
        (about c1
          (latitude 23.2)
          (longitude 75.9)
          (region "Crisis Island")))
!!!!*****
Recognition changes at agent time 15:
  entry:  C1      |C|LOCATION
16
> (tellm (create c2 location)
        (about c2
          (latitude 23.3)
          (longitude 75.6))
```

```

                (region "Crisis Island")))
*
Recognition changes at agent time 16:
  entry:  C2      |C|LOCATION
17
> (tellm (create c3 location)
      (about c3
          (latitude 23.1)
          (longitude 75.3)
          (region "Crisis Island")))
*
Recognition changes at agent time 17:
  entry:  C3      |C|LOCATION
18

```

Now, let's use **location** as a component of a more complex concept. **City** is a **location** that has additional roles that make it a unique new concept. This is accomplished by including the more general concept in the **:is** clause of the concept definition. For example:

```

> (defconcept city :is
    (and location
          (exactly 1 name)
          (exactly 1 population)))

```

```
|C|CITY
```

Of course, we still must define the relations that can fill these new roles.

```

> (defrelation name
    :range string
    :characteristics :closed-world)
----...
|R|NAME

```

Another aspect of roles is the *cardinality* of their fillers. Relations can be defined to restrict their filler to being *single valued* (the default is multi-valued). If a relation is known to be single valued, then asserting a new value for that relation will automatically cause the old value to be retracted. In Loom terminology, this behavior is referred to as “clipping”. Let's use this in defining our **population** relation:

```
> (defrelation population
    :range integer
    :characteristics :single-valued)
..+++++++
|R|POPULATION
```

Of course, the interesting part of this exercise lies in manipulating the instances. We have created three instances of locations. Refer back to the “Crisis World” map, and find the names of the three cities. Let’s fill in the name and population roles of **c1**, **c2**, and **c3**.

```
> (tellm (about c1 (name "Costa Via") (population 11000))
        (about c2 (name "Centrada") (population 8000))
        (about c3 (name "Otheria Beach") (population 4500)))
!!!!!!*
Recognition changes at agent time 18:
  entry:  C3      |C|CITY
*
Recognition changes at agent time 18:
  entry:  C2      |C|CITY
*
Recognition changes at agent time 18:
  entry:  C1      |C|CITY
**
19
```

Loom recognizes each instance qualifying to be classified as a **city** rather than just a **location**. Now, let’s “flesh out” this type of knowledge about the crisis world by creating instances for all the cities described on the “Crisis World” map.

```
> (tellm (create c4 location)
        (about c4
          (name "Nearby Shores")
          (population 17000)
          (latitude 20.9)
          (longitude 74.6)
          (region "Nearby Island")))
*
Recognition changes at agent time 19:
  entry:  C4      |C|LOCATION
```

```

    entry:  C4          |C|CITY
20
> (tellm (create c5 location)
    (about c5
      (name "Mainland AFB")
      (population 900)
      (latitude 23.0)
      (longitude 75.0)
      (region "Mainland")))
*
Recognition changes at agent time 20:
    entry:  C5          |C|LOCATION
    entry:  C5          |C|CITY
21

```

3.3 Complex Knowledge Base Queries: RETRIEVE

As a knowledge base develops, much of its value to an application user comes from the ability to query the knowledge base about the state of the world, including inferences made about the state of the world. Since we have begun to develop a body of knowledge about cities and locations, let's start to do some basic queries.

First, let's retrieve all the knowledge base entries about cities:

```

> (retrieve ?c (city ?c))
(|I|C3 |I|C2 |I|C1 |I|C4 |I|C5)

```

This retrieval asked for all instances of type city. The basic components of the retrieve statement are the pattern variables (the ?c) and the retrieval pattern. Such retrievals can be made more complex by using more than one pattern variable, or by combining multiple selection terms with the **and** construct. A more complex retrieval might ask for pairs of cities where one city's population is greater than 10,000, and the other's is between 5,000 and 10,000:

```

> (retrieve (?x ?y)
    (and (city ?x)
      (city ?y)
      (> (population ?x) 10000)
      (< (population ?y) 10000)
      (> (population ?y) 5000)))
((|I|C4 |I|C2) (|I|C1 |I|C2))

```

Of course, the results will depend on the populations you assigned to your own instances. If you got NIL as your response, lower the 5000 number. Notice that the result is a list of answers. Each answer has the same pattern as the pattern variable specification (in this case, two objects). When the elements of each answer are substituted for their corresponding pattern variables in the retrieval pattern, the retrieval pattern is true. To say that another way, each answer *satisfies* the retrieval pattern.

Imagine that you were building an application on top of Loom. Just as you have queried Loom's knowledge base, your application program could issue similar LISP calls to **retrieve** to acquire this list of knowledge base entries.

3.4 The Use of Constraints in Concept Definition

Often, when we want to specialize one concept to form a new concept, we want to do it by restricting the value of some role. A big city, for example, might be a city whose population is greater than some value. Loom provides mechanisms to build such restriction constraints.

One method of restriction is just to build the constraint into the **:is** clause that defines the concept. For example, let's redefine a **city** to be a location with a big population :

```
> (defconcept city :is
    (and location
      (exactly 1 name)
      (exactly 1 population)
      (>= population 5000)))
-.+
|C|CITY
> (tellm)
!*
Recognition changes at agent time 20:
  exit: C3      |C|CITY
*
Recognition changes at agent time 20:
  exit: C5      |C|CITY
21
```

By running **tellm** without any arguments, we force the knowledge base to be updated. Now, ask about any of your instances:

```
> (ask (city c3))
NIL
```

As you can see, our smaller cities lost their classification as **city** because they do not satisfy the constraint and revert to being merely a **location**.^{*} This potential problem can be avoided by creating a new concept for these instances:

```
> (defconcept town
  :is (and location
        (exactly 1 name)
        (exactly 1 population)
        (< population 5000)))
.+
|C|TOWN
> (tellm)

!*
Recognition changes at agent time 21:
  entry:  C5      |C|TOWN
*
Recognition changes at agent time 21:
  entry:  C3      |C|TOWN
*
22
```

- Check to see if each instance is a **city** or a **town**.

3.5 Relations Between User-Defined Concepts

We have cheated a bit in our examples so far. We have used examples of roles that are filled by instances of *built-in* concepts, such as **string** and **number**. A role defines a link from one concept to another. When we say that a city has a name, we are really saying that there is an instance of a **city** concept whose **name** role is filled by an instance of a **string** concept. Phew!

* It is also possible that the smaller “ex-cities” will be considered incoherent. This will occur if the **tellm** used to create them asserted that they were cities directly, rather than letting Loom deduce this from the values of the **latitude**, **longitude**, **population**, **name-is** and **region** roles. Since there is a direct user assertion that an object is a city, but the population value is no longer in range (since we changed the definition of city), an inconsistency is detected. You can correct this by using (**forgetm** (**city** *object-name*)).

In general, both the domain and the range of a relation/role are just concepts, either built-in or user defined.

To illustrate this idea, let's create two new concepts, called **damaged** and **damage-level**. Then, define the relation **damage-estimate** that links these concepts as a role owner / role filler.

```
> (defconcept damaged
    :is (exactly 1 damage-estimate))
|C|DAMAGED
> (defconcept damage-level
    :is-primitive (one-of normal structural-damage injuries
damage-and-injuries))
.
|C|DAMAGE-LEVEL(ONE-OF NORMAL STRUCTURAL-DAMAGE INJURIES DAMAGE-AND-
INJURIES)
> (defrelation damage-estimate
    :domain damaged
    :range damage-level
    :characteristics :single-valued)
..+++
|R|DAMAGE-ESTIMATE
```

Notice something else new. We have used the construct **one-of** to define what is in a concept (we could have also used **defset**). In other words, we have defined a primitive concept that is described by enumerating its instances, not by specifying a set of restrictions. If we had wanted to express the damage level in terms **low**, **medium**, and **high**, we could have used the expression **:the-ordered-set** instead.

3.6 A More Complex Example of Defined Concepts

Now that we have defined the concept **damaged**, we can use it to build damaged versions of other complex concepts. For example:

```
> (defconcept damaged-city :is
    (and city damaged))
.+
|C|DAMAGED-CITY
```

This example shows how a new concept can be defined by combining two concepts together. Let's say something about one of our city instances to cause it to be damaged:

```

> (tellm (damage-estimate c1 damage-and-injuries))
!!!!*
Recognition changes at agent time 22:
  entry:  NORMAL |C|DAMAGE-LEVEL(ONE-OF NORMAL STRUCTURAL-DAMAGE
INJURIES DAMAGE-AND-INJURIES)
*
Recognition changes at agent time 22:
  entry:  STRUCTURAL-DAMAGE |C|DAMAGE-LEVEL(ONE-OF NORMAL
STRUCTURAL-DAMAGE INJURIES DAMAGE-AND-INJURIES)
*
Recognition changes at agent time 22:
  entry:  INJURIES |C|DAMAGE-LEVEL(ONE-OF NORMAL
STRUCTURAL-DAMAGE INJURIES DAMAGE-AND-INJURIES)
*
Recognition changes at agent time 22:
  entry:  DAMAGE-AND-INJURIES
|C|DAMAGE-LEVEL(ONE-OF NORMAL STRUCTURAL-DAMAGE INJURIES
DAMAGE-AND-INJURIES)
****
23

```

3.7 Non-Numeric Constraints in Concept Definition

The restrictions we wish to include in the definition of a concept need not be limited to intervals on numeric values. Symbolic descriptions and set membership can also be used. For example:

```

> (defconcept city-needing-medical-help
  :is (and damaged-city
        (the damage-estimate (one-of injuries damage-and-
injuries))))

```

Experiment with the result of this definition. You should find that any city instance that you **tellm** about having an appropriate type of damage (e.g. injuries) will classify as a **city-needing-medical-help**. Note that **c1** will not be classified as a **city-needing-medical-help** until you updated the knowledge base again, since this definition was added after the instance classifier has already run.

- Try to build a **retrieve** that finds only large cities needing medical help.

3.8 Using Backward-Chaining Inference in Concept Matching

Loom provides a powerful mechanism by which the values of an objects roles can impact its classification. This mechanism involves the use of the **:satisfies** term. This mechanism specifies a condition under which an object instance can be classified as a certain type. It is used as a term within the **:is** or **:constraints** clauses of a concept definition, as in this example:

```
> (defconcept priority-1-medical-need :is
    (and city-needing-medical-help
         (satisfies (?x) (> (population ?x) 10000))))
```

We can test for an instance's classification as a **priority-1-medical-need** by using the **ask** or **retrieve** statements.

```
> (ask (priority-1-medical-need c1))
!*
T
```

3.9 More Complex Terms in Concept And Relation Definition

We have, for the most part, described very simple concepts whose roles are unique and single-valued. In fact, many real-world concepts may have a variable number of instances filling any particular role. Loom supports this type of concept role definition, using terms like **:at-least** and **:at-most**:

```
> (defconcept path)
.+
|C|PATH
> (defconcept transportation-path :is
    (and path
         (at-least 2 terminus)
         (exactly 1 name)))
|C|TRANSPORTATION-PATH
> (defrelation terminus
    :domain transportation-path
    :range location)
..++
|R|TERMINUS
```

- Define your own concept of a **medical-team**. Use **at-least** and **at-most** to restrict the size and composition of doctors, nurses, etc. Give it a status that can be either **available** or **committed**. Create an instance and fill its roles.

Chapter 4 — Behavioral Programming: Actions and Methods

This chapter will introduce ideas of behavioral specification using the object oriented programming paradigm in Loom. Mechanisms for defining Loom actions and methods to implement those actions will be explored.

We are about to take a major shift in our exploration of Loom. Thus far, we have exploited only one of Loom's sub-languages: the *Modelling Language*. This has provided a rich environment for defining a domain taxonomy and for describing specific instances of objects within that taxonomy. Loom's modelling language, with its powerful classification-based inference engine, is one feature that sets Loom apart from the pack of knowledge representation systems available. We are about to tackle a new collection of Loom features, known collectively as the *Behavior Language*.

Loom provides two paradigms for behavioral specification: *object oriented programming*, and *rule based programming*. This chapter will address object oriented programming features.

4.1 More on Object Specification

This section really will not introduce anything new. It is included to serve two purposes. First, it serves as a reminder that what we have been doing thus far is about defining objects. The instances of concept types we create are the objects whose behavior we are about to learn to specify. Second, this section will provide a place to define a few new knowledge base entries, to keep up with our crisis planning example. If you have not yet noticed the "tutorial.lisp" file in the Loom directory, now may be a good time to look for it. This file contains all of the source code for the examples in this tutorial and will save you quite a bit of typing in this chapter.

The basic data structure that a transportation planning expert deals with is called a TPFDD (pronounced "tip-fid"), for Time-Phased Force Deployment Data. This is a collection of records, each of which defines a single requirement to transport a unit of cargo or personnel. We can capture the most important characteristics of these data structures in the following definitions:

```
(defconcept tpfdd
  :constraints
  (and (exactly 1 name)
        (at-least 1 transportation-requirement)))
(defrelation transportation-requirement
  :range tpfdd-record
```

```

      :characteristics :closed-world)
(defconcept tpfdd-record)
(defrelation cargo-type
  :range (:the-set medical-team engineering-team)
  :characteristics single-valued)
(defrelation unit :range number
  :characteristics single-valued)
(defrelation stons :range number
  :characteristics single-valued)
(defrelation mtons :range number
  :characteristics single-valued)
(defrelation point-of-disembarcation :range location
  :characteristics single-valued)
(defrelation point-of-embarcation :range location
  :characteristics single-valued)
(defrelation transportation-path-used :range transportation-path
  :characteristics single-valued)
(defrelation destination :range location
  :characteristics single-valued)
(defrelation ready-to-load-date :range number
  :characteristics single-valued)
(defrelation earliest-arrival-date :range number
  :characteristics single-valued)
(defrelation latest-arrival-date :range number
  :characteristics single-valued)
(defrelation destination-date :range number
  :characteristics single-valued)

```

The meaning of the roles of a TPFDD record will be described as they are addressed in our examples, here and in Chapter 5. Note that we chose to specify each relation to be single-valued, overriding the default of multiple-valued relations. The function **set-features** could be used to declare all relations to use closed-world semantics by: (**set-features :closed-world-roles**).

Let's create an instance of a TPFDD object:

```

> (tellm (create tf1 tpfdd)
      (name tf1 "Crisis Island TPFDD"))

```

4.2 Basic Action Definition

An *action* in Loom represents a procedural behavior that is invoked on request. An action is implemented by a set of *methods* that implement the responses necessary to produce the desired behavior. An action represents the generic solution to a task request, while methods provide context-specific options for actually accomplishing that task. Task requests are made using the **perform** function, which acts much like the typical send functions in object-oriented programming.

The most basic mechanism for defining an action is simply to start creating methods. The action will then be built implicitly. Let's begin by building and executing a toy method:

```
> (defmethod build-tpfdd (?obj)
  :title "Build TPFDD if you are a TPFDD object."
  :situation (tpfdd ?obj)
  :response ((format t "~a is a TPFDD object.~%" ?obj)))
|METHOD|BUILD-TPFDD-"Build TPFDD if you are a TPFDD object."
```

Let's look at the components of this method definition. The first argument, **build-tpfdd**, is the operation (or, if you like, behavior or task) this method knows how to perform. More accurately, it is the name of the action that this method implements. It has an argument list, in the usual Lisp form, except that the ? mark on the variable is mandatory. In this example, we have not defined the action explicitly; the action is created implicitly with the **build-tpfdd** name. Any other methods with the same name and arguments will be included as methods for that action. The **:title** keyword argument provides a way to name this method, apart from the operation it performs. This is important, since there may be many methods that perform the same operation, either in parallel, or in unique problem contexts. While the title is optional, its use is strongly recommended! When a method is defined, the title becomes part of its name in the knowledge base. If it is re-defined during debugging, the old instance will only be overwritten correctly if the titles, as well as the action names, match. In other words, it is best to use a title and never change it.

The **:situation** keyword provides an opportunity to restrict the firing of this method. This argument is optional, but if used will define conditions under which this method is an appropriate response to a task request. Note that, unlike most message passing systems, Loom methods are not attached to a specific object; they are really pattern-directed functions that operate on objects passed through the argument list.

Finally, the **:response** keyword supplies the functional body of the method. This contains a list of S-expressions that are evaluated within the scope of the bindings defined in the argument list.

4.3 Activating Methods

Methods are triggered through the **perform** function. To trigger the **build-tpfdd** method in the previous section, the call would look like the following:

```
> (perform (build-tpfdd (get-instance 'tf1)))
|I|TF1 is a TPFDD object.
```

Notice that the basic elements of the **perform** call look just like a function call: the generic function (i.e. action) name, followed by its arguments in a form that matches the argument list of the action's definition. Notice also that the method wants the instance object, rather than its name. This is provided by the **get-instance** function.

When this **perform** is executed, observe how the action is performed. Of course, this simple action is only for demonstration purposes. More complex examples will be demonstrated in later sections.

4.4 Defining Actions with Multiple Methods

Much of the power of object oriented programming comes from the ability to specialize operations in a context-sensitive way. This is accomplished by defining multiple methods, and providing a mechanism to activate the appropriate method (or methods) in any situation.

In Loom, we can specialize a generic action by providing multiple methods to perform the operation. Each method represents an alternative means of accomplishing the action's task. These methods are defined having the same name and argument list, but are distinguished by unique **:title** clauses. Their **:situation** clause specifies the context in which each method is applicable. Let's see how this works in practice. Let's define a new **build-tpfdd** method with a new title, with no **:situation** clause:

```
> (defmethod build-tpfdd (?obj)
  :title "This object is not necessarily a TPFDD."
  :response ((format t "Oops!!! Object ~a is not a TPFDD!!~%"
                    ?obj)))
```

This method will react to any invocation, without any situation. Experiment with invoking the two methods we have defined, and see when each fires:

- Invoke the method using the TPFDD instance.
- Invoke the method using one of your city instances.

As you will observe, these methods work quite nicely -- the appropriate method fires depending whether the object is or is not a TPFDD object. This may seem surprising, since the two situations are not disjoint: one essentially says “Fire if I am a TPFDD,” while the other says “Fire all the time.”

This example works because of the default behavior specified within the (implicitly defined) action. Remember, the generic function is represented by an action definition, in this case generated implicitly from our method definitions. One feature of an action is a *filter* function that chooses which method or methods to try to fire (yes, more than one **can** fire). The default filter selects the *most specific* methods to fire. In this example, the method with the “**Build TPFDD if you are a TPFDD object.**” title is tried first, since its **:situation** clause is more specific than the competing method. In cases where a more specific method does **not** fire, the next most specific method is tried.

4.5 Defining Actions with Different Filters

It is possible to change the default filter by explicitly building an action definition. This definition replaces the implicit definition, and allows the opportunity to specify a filter function other than the default:

```
> (defaction build-tpfdd (?obj)
      :filters (:select-all))
```

Experiment with the behavior of this new filter using our example methods. As you can see, it is possible for more than one method to fire under this filter. At this time, the filters supported are **:most-specific**, **:select-one**, **:warning**, **:error**, and **:select-all**, **:overrides**, and **:last-one**. They seem to cover the most useful cases. Other filters may be defined in the future if sufficient utility is identified.

- Re-set the filter for our action to **:most-specific**.

4.6 Knowledge Base Access within Methods

The action clauses within a method definition are basically just LISP code. The functions and commands we have been typing in at the top level, being standard LISP function and macro calls, can be embedded quite naturally within this LISP code. For example, we can do complex knowledge base queries when the method fires:

```
> (defmethod build-tpfdd (?obj)
  :title "Construct TPFDD for high-priority cities."
  :situation (tpfdd ?obj)
  :response ((do-retrieve (?c)
              (priority-1-medical-need ?c)
              (format t "~a needs a tpfdd record.~%" ?c))))
```

This action retrieves all the cities in the knowledge base classified as **priority-1-medical-need**.

- Invoke **build-tpfdd** with **tf1**.

4.7 Actions Calling Actions

Of course, actions are also callable from any embedded LISP code. This allows these generic functions to call each other when appropriate. This gives behavior analogous to message passing in more common object oriented systems. There can be complications, however, in keeping the appropriate bindings straight for object names vs. the objects themselves. As an example, let's extend our previous example to tell any appropriately needy cities to build a TPFDD record describing the transportation requirement to ship them a medical team. Notice that this overwrites the old method definition with the same name, argument list, and title:

```
> (defmethod build-tpfdd (?obj)
  :title "Construct TPFDD for high-priority cities."
  :situation (tpfdd ?obj)
  :response ((do-retrieve (?c)
              (priority-1-medical-need ?c)
              (perform (build-record ?c ?obj))))))
```

The embedded **perform** does not yet have a method defined. We want to define a method that creates a new instance, fills some of its roles, and then tells the TPFDD about the new record by adding a new role filler to it.

One way to do this is to build the entire method in one definition. The following example shows how this is done.

```
> (defmethod build-record (?need-obj ?tpfdd-obj)
  :title "Construct TPFDD for high-priority-cities."
```

```

:situation (priority-1-medical-need ?need-obj)
:response
  ((let ((?new-obj (createm nil 'tpfdd-record)))
    (format t "Creating record for ~a.~%" ?need-obj)
    (tellm (about ?new-obj (cargo-type medical-team)
              (destination ?need-obj)))
    (tellm (transportation-requirement ?tpfdd-obj
              ?new-obj))))))

```

- Test the new **build-record** action by performing **build-tpfdd** with **tf1**, and examine the records **build-record** has created.

Below is an equivalent means for defining our method. This definition illustrates how a method can invoke other methods using **perform**.

```

(defmethod build-record (?need-obj ?tpfdd-obj)
  :title "Create and add a TPFDD record to a TPFDD."
  :situation (priority-1-medical-need ?need-obj)
  :response ((let ((new-obj (create nil 'tpfdd-record)))
    (format t "Creating record for ~a.~%" ?need-obj)
    (set-value new-obj 'destination ?need-obj)
    (set-value new-obj 'cargo-type 'medical-team)
    (tellm)
    (perform (add-record ?tpfdd-obj new-obj))))))

```

There are a few interesting things going on in this method. First, notice the **create** function call to create a new instance of type **tpfdd-record**. Then, notice the use of the built-in function **set-value** to assign a role-filler to an object. Finally, notice the reference to a new method which gives a more specific method to add a new TPFDD record to an existing TPFDD object:

```

(defmethod add-record (?tpfdd ?record)
  :title "Create and add a TPFDD record to a TPFDD."
  :situation (tpfdd ?tpfdd)
  :response ((format t " Adding ~a to ~a.~%" ?record ?tpfdd)
    (tellm (transportation-requirement ?tpfdd
              ?record))))

```

Now, try the whole chain of methods:

```
> (perform (build-tpfdd (get-instance 'tf1)))
```

- Examine the resulting knowledge base entries using the **pi** function.

Chapter 5 — Behavioral Programming: Productions

This chapter will introduce ideas of data-driven programming using the production paradigm in Loom. Mechanisms for defining, modifying, selecting, and executing Loom productions will be covered.

Loom provides powerful data-driven programming mechanisms that are closely integrated with all of the other representational facilities. In the previous chapter, we saw how Loom incorporates the object-oriented programming paradigm for behavioral programming. Now, we will see how data-driven programming is similarly incorporated in Loom through the use of **defproduction** and related constructs.

5.1 Defining a Production

To set up our discussion of productions, we need to extend our ongoing example. Figure 2 shows a more detailed look at Crisis Island. We will need to build on our previously defined notion of a transportation path by adding a **road** concept as a **transportation-path** that crosses a particular section of land.

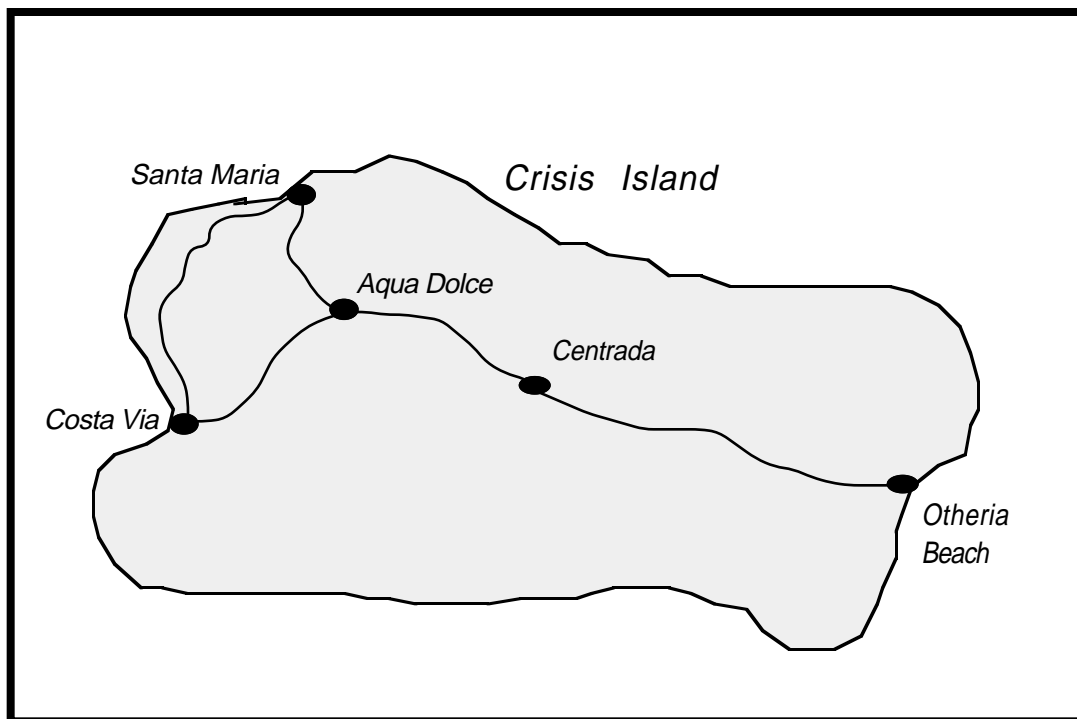


Figure 2: A More Detailed Look at Crisis Island

First we'll update our **transportation-path** concept to include a **path-status** relation and add two types of paths:

```
(defconcept transportation-path :is
  (and path
    (at-least 2 terminus)
    (exactly 1 name)
    (exactly 1 path-status)))
(defrelation path-status
  :range (:one-of "normal" "degraded" "damaged" "unusable")
  :characteristics single-valued)
(defconcept road :is
  (and transportation-path
    (at-least 1 land-crossed)))
(defconcept sea-lane :is
  (and transportation-path
    (at-least 1 ocean-crossed)))
(defconcept damaged-road :is
  (and road
    (the path-status (or "damaged" "unusable"))))
```

- Add the relations **land-crossed** and **ocean-crossed** with domain transportation-path and range string.
- Add town instances **c7** and **c8** named "Santa Maria" and "Aqua Dolce" respectively

Now, lets represent the major roads on the island:

```
(tell (create r1 transportation-path)
  (about r1 (terminus c1)(terminus c7)
    (name "West Coastal Road")
    (land-crossed "Western Crisis Island")))
(tell (create r2 transportation-path)
  (about r2 (terminus c1)(terminus c8)
    (name "Aqua Dolce Road")
    (land-crossed "Western Crisis Island")))
(tell (create r3 transportation-path)
  (about r3 (terminus c7)(terminus c8)
    (name "Hill Road")
    (land-crossed "Western Crisis Island")))
```

```
(tell (create r4 transportation-path)
      (about r4 (terminus c8)(terminus c2)
              (name "Ridge Road West")
              (land-crossed "Central Crisis Island")))
(tellm (create r5 transportation-path)
       (about r5 (terminus c2)(terminus c3)
               (name "Ridge Road East")
               (land-crossed "Eastern Crisis Island")))
```

We now have a fairly good set of models describing the major components of our area of operation and the domain in general. Let's add a few concepts describing various types of reports and messages.

```
(defconcept report)
(defconcept message :is-primitive
  (and report
    (exactly 1 title)
    (at-most 1 sender)
    (at-least 1 recipient)
    (exactly 1 contents)))
(defrelation title
  :range string)
(defrelation sender
  :range string)
(defrelation recipient
  :range string)
(defrelation contents
  :range string)
```

Okay! All of the above definitions should be familiar to you by now. Let's consider an example of the type of data-driven behavior we would like to represent. In general, when a transportation path has been degraded, the crisis planner needs to know about the change of status. We will write this rule as follows: when a road is known to be damaged or unusable, send a message to the "planning team".

```
> (defproduction P1-Report-Bad-Road
   :when (detects (damaged-road ?road))
   :perform (send-status-message ?road "Planning Team"
                                "is seriously damaged"))
```

5.2 Executing a Production

The **:when** clause is the trigger condition for the production, and the **:perform** clause specifies the action to be taken when the production fires. This is similar to the **:situation/:response** pairing we saw in methods, but there are some significant differences. The **:when** clause must include a “transition test” using one of the operators **detects**, **undetected**, or **changes**. The rule will fire when an object becomes recognizable as a **damaged-road**. We can also test for an update that causes the object to *no longer* be a **damaged-road** by making the **:when** clause to be: **(undetected (damaged-road ?road))**. We have not yet defined the methods and action executed by this production, so we'll do that now.

```
(defaction send-status-message (?obj ?to ?contents)
  :filters (:most-specific))
(defmethod send-status-message (?obj ?to ?contents)
  :title "Send a Message about an Object"
  :response
  ((format t "To: ~a Re: ~a ~a.~%" ?to ?obj ?contents)))
```

- Give the road **r1** a **path-status** of **normal**. The instance should look something like:

```
> (pi r1)
(TELL
 (:ABOUT R1
  ROAD
  (PATH-STATUS "normal")
  (LAND-CROSSED "Western Crisis Island")
  (NAME "West Coastal Road")
  (TERMINUS C1)
  (TERMINUS C7)))
```

- Check for the existence of any damaged roads using a **retrieve**.

You should notice that the production has not yet fired because no damaged roads exist. Now tell the planner that the **path-status** for **r1** has become “unusable”.

```
> (tellm (about r1 (path-status "unusable")))
Rete match changes at agent time 38:
> entry: |I|R1 |C|DAMAGED-ROAD
To: Planning Team Re: |I|R1 is seriously damaged.
```

38

We can modify the production in another important way: instead of executing immediately, we can **:schedule** a production's task for execution later according to a **:priority** which can be either **:low** or **:high**. Since sending this type of alert is not as critical as some other messages, we'll give it low priority, and the redefined production will look like this:

```
> (defproduction P2-Report-Bad-Road
  :when (damaged-road ?road)
  :schedule (send-status-message ?road "Planning Team"
    "is seriously damaged(2nd rule)")
  :priority :low)
```

Tasks that are scheduled in this way will be executed at the end of the current match cycle after all other productions are fired.

- Reset the status of **r1** to normal.
- Execute the following set of statements (or similar **tells**) and observe when the productions fire.

```
> (tell (about r1 (path-status "unusable")))
|I|R1
> (tell (about c7 (damage-estimate structural-damage)))
|I|C7
> (tell (about r2 (path-status "normal")))
|I|R2
> (tell (about r3 (path-status "normal")))
|I|R3
> (tell (about r4 (path-status "degraded")))
|I|R4
> (tellm (about r5 (path-status "normal")))
Rete match changes at agent time 40:
> entry: |I|R5 |C|TRANSPORTATION-PATH
> entry: |I|R5 |C|ROAD
> entry: |I|R4 |C|TRANSPORTATION-PATH
> entry: |I|R4 |C|ROAD
> entry: |I|R3 |C|TRANSPORTATION-PATH
> entry: |I|R3 |C|ROAD
> entry: |I|R2 |C|TRANSPORTATION-PATH
> entry: |I|R2 |C|ROAD
```

```
> entry: |I|R1 |C|DAMAGED-ROAD
To: Planning Team Re: |I|R1 The road is seriously damaged.
To: Planning Team Re: |I|R1 The road is seriously damaged(2nd rule).
40

> (retrieve ?bad-road (damaged-road ?bad-road))
(|I|R1)
```

Notice that nothing happens until **tellm** is encountered. After Loom re-classifies the instances, productions are matched (see the lines marked “rete match changes”). The low priority rule fires last.

⌘ ⌘ ⌘