# SRL+

## Kernel Language Definition and User Manual

## Version 1.0

J. Mark Wright, Mark S. Fox, David Adam[1]

Carnegie Group Inc
Station Square at Commerce Court
Pittsburgh, PA 15219

21 September 1984

[1]Earlier versions of this manual have appeared as internal reports of the Intelligent Systems Laboratory, Robotics Institute, Carnegie-Mellon University, Pittsburgh Pennsylvania 15213.

# Table of Contents

**1. Introduction** 1

**2. Language Overview** 7

**3. Schema Manipulation** 15

3.1. Schemata: How to create, delete, and test their existence 16

3.2. Creating and manipulating meta-schemata 17

**4. Slots and Values** 21

4.1. Slot Definition 22

4.1.1. Meta-slots: Creating and accessing them 24

4.2. Value Definition 26

4.2.1. Units: Value/meta-value pairs 30

4.3. Facets: Slot meta-information 33

4.3.1. Domain and Range Facets 33

4.3.2. Cardinality Facet 37

4.3.3. Demon Facet 37

4.4. Schemata as Values: Auto-linking 40

**5. Schema Relations** 43

5.1. User Defined Relations 45

5.1.1. Inheritance-specs 45

5.1.2. Inheritance-specs 47

5.1.3. Inclusion-spec 48

5.1.4. Exclusion-spec 49

5.1.5. Elaboration-spec 51

5.1.6. Map-spec 54

5.1.7. Introduction-spec 55

5.2. Creating and Using Relations 57

5.2.1. Creating Relations 57

5.2.2. The Relation Compiler 58

5.2.3. Using Relations 59

5.3. Caching and Dependency Links 59

5.3.1. Local Specialization of Relations 61

**6. Paths and Inheritance** 65

6.1. Paths 66

6.2. Path Selection 67

6.3. Schema Typing and Relational Transitivity 68

6.4. Slot Accessibility 69

6.5. Inheritance Algorithm 70

**7. Contexts** 73

7.1. Using contexts 74

**8. Altering The SRL Environment** 77

8.1. Switches: allow the user to modify the SRL environment 78

**9. Error Handling** 81

9.1. The error-handling system 82

| | |
|---|---|
| **10. SRL$_o$: Object Programming in SRL+** | **85** |
| 10.1. Messages and Message Sending | 85 |
| **11. SRL Data Base System** | **87** |
| 11.1. Introduction | 88 |
| 11.2. DB manipulation | 88 |
| 11.3. DB Functions | 89 |
| 11.4. Utility Functions | 89 |
| 11.5. Global Parameters | 90 |
| **Appendix I. Using SRL2** | **91** |
| I.1. Initializing SRL2 | 91 |
| I.2. System Limitations | 91 |
| **Appendix II. User Commands** | **93** |
| **Appendix III. Database Backup System** | **95** |
| **Appendix IV. Error Messages** | **97** |
| **Appendix V. System Relations** | **99** |
| **Appendix VI. History** | **103** |
| **Index** | **107** |

# List of Figures

**Figure 4-1:** The restriction grammar 34
**Figure 4-2:** The cardinality grammar 37
**Figure 6-1:** The path grammar 66

# List of Schemata

Schema 2-1: The **mammal** schema 9

Schema 2-2: The **mammal** schema with NURSING-METHOD slot filled with the value "breast." 9

Schema 2-3: In the **mammal** schema, the value "breast" has a meta-value that contains the CREATOR slot, which has the value "M.Fox." 10

Schema 2-4: The NURSING-METHOD slot has a facet called range. 10

Schema 4-1: The **value** schema 31

Schema 4-2: The **slot** schema 33

Schema 4-3: The **dog** schema with a cardinality restriction 37

Schema 4-4: The **demon** schema 38

Schema 4-5: The **fido** schema with "mood-demon" and the **mood-demon** schema 39

Schema 4-6: How inverse relations connect schemata 40

Schema 4-7: The inverse of the HAS-WINGS slot 41

Schema 4-8: The inverse of the HAS-COLOR slot 41

Schema 5-1: A system-generated inverse relation 44

Schema 5-2: The **relation** schema 45

Schema 5-3: The **inheritance-spec** schema 47

Schema 5-4: The **inclusion-spec** schema 48

Schema 5-5: The **exclusion-spec** schema 50

Schema 5-6: The **walls-ex-spec** 51

Schema 5-7: The **elaboration-spec** schema 52

Schema 5-8: The **dog-elab-spec** 52

Schema 5-9: The **dog** schema with elaborated slots 53

Schema 5-10: The meta-slot of the LEFT-FRONT-LEG slot 53

Schema 5-11: The **map-spec** schema 54

Schema 5-12: The **toy-map-spec** 55

Schema 5-13: The **introduction-spec** schema 56

Schema 5-14: The inherited value "4" is copied into the **dog** schema 60

Schema 9-1: The **SRL-error** schema 82

Schema 9-2: The **error-spec** schema 83

Schema 10-1: The message schema 85

# 1. Introduction

The introduction is designed to acquaint the user with SRL+ package. The introduction briefly explains all the elements of the package, their capabilities, and possible applications.

## SRL+: AN INTEGRATED KNOWLEDGE ENGINEERING ENVIRONMENT

SRL+ is an integrated knowledge representation and problem solving environment for constructing knowledge-based systems. SRL+ functions as a high performance productivity tool for knowledge engineers and Artificial Intelligence systems developers. The system drastically reduces the effort required to build knowledge bases and customize problem solving strategies for specific domain applications. SRL+ extends the user's capabilities by combining a feature-rich knowledge representation language with a variety of powerful problem solving techniques. SRL+ offers:

- A uniform knowledge representation language with user-definable inheritance relations.
- A logic programming language.
- A rule-based programming language.
- An object-based programming langauge.
- An agenda mechanism.
- Discrete simulation language.
- Window/canvas interface.
- An embedded database for large applications.
- 2D and business color graphics.
- A natural language interface.

SRL+ is written in Common Lisp, the recognized industry standard. SRL+ is based on a proven experimental prototype, which has been used to solve diverse "real world" problems in a variety of production environments.

---

## SRL+'s Knowledge Representation Facilities

SRL+ provides a frame-based language, which is efficient, easy to use, and suitable for both small and large applications. The schema is the basic representation unit in SRL+. The schema is a symbolic representation of a concept such as an object, process, or control strategy. The schema has slots and values that store attributive, structural, and relational information about the concept. The relations in a schema network allow one schema to inherit information from other schemata.

**User defined relations and inheritance semantics:** SRL+ provides a set of pre-determined relations to perform inheritance, but also permits users to create their own relations and inheritance semantics. By defining their own relations, users can construct schemata and relations that closely match the domain (e.g., "revision-of", "son-of", etc.). Consequently, the complexity of mapping domain knowledge into a knowledge base is reduced.

**Meta-knowledge representation:** Meta-information can be associated with any part of a schema -- the schema itself, its slots, or values. Meta-knowledge provides information about the creation of the schema, slot, or value (e.g, when, where, how, and why it was created).

**User-defined dependency relations:** The user can provide SRL+ with meta-knowledge that defines how knowledge has been derived or inherited by a schema.

**Procedural attachment:** Programs can be associated with slots in the form of demons.

**User-controlled search:** Paths enable the user to control the way inheritance search is performed. Paths allow the user to specify which schemata and relations may be searched during inheritance.

**Error-handling:** The integrated schema-based error handling facility permits the user to define how the system reacts to errors.

## Inference Strategies

Knowledge-based systems have employed a variety of techniques to solve some real world problems. Yet no single technique has proven adequate. SRL+ offers the user a powerful set of problem solving techniques that may be combined in one application. Each technique is defined by schemata and integrated with the SRL+ representation language.

**Integrated production rule interpretor:** Production rules are represented as schemata, and are matched against schemata in the knowledge base. Both forward and backward chaining control strategies may be used. The rule-based programming of SRL+ is fully explained in the $SRL_P$ document.

**Integrated logic programming environment:** The system combines the modus ponens inferencing used in logic programming with representation power SRL+. The inheritance mechanism provides default reasoning not available in other logic programming environments. Logic programming is further described in the $SRL_H$ document.

**Integrated object programming language:** This problem solving tool supports the message-sending paradigm for invoking procedures. Objects are represented as schemata that use the SRL inheritance mechanism to retrieve procedures. Procedures executed in reaction to messages may be logic programs, rules, or Lisp functions (see chapter 10.)

**Multi-queue event manager:** A mutli-queue event manager enables the user to schedule events to occur in a simulated or normal operating mode. Symbolic event-based simulations of complex real-world processes may be implemented using this mechanism. The multi-queue event manager is explained in the $SRL_E$ documentation.

## System Building Tools

**Multi-window/canvas interface:** Schemata for windows, displays, and canvases are instantiated to build interfaces with mousc input. This wide bandwidth interface enables the user to view and edit schemata and schema networks simultaneously. The window/canvas interface is described in the $SRL_W$ document.

**Schema-driven command system:** Command interfaces are easy to build using hierarchical command system that includes spelling correction, built-in help facilities, and a standard command library.

**2D and business color graphics:** A CORE-based graphics package lets the user construct 2D graphic displays. Business graphics are included in the package. SRL+'s graphic cpabilities are described in the $SRL_G$ documentation.

**Natural language interface:** An interface based on the PLUME natural language parser allows the user to query the knowledge base, edit, and update factual and procedural knowledge represented as schemata. The interface can also be used to issue commands to the application system. The natural language interface is explained in the $SRL_{NL}$ document.

## Other Features

**Integrated database system:** A multi-user database system is provided to store schemata. Schemata that are used frequently are cached in memory. The database system is detailed in the $SRL_{DB}$ documentation.

**Version Management:** Using the context mechanism, users can create different versions of the same model.

## Operating Environments

SRL+ is implemented in Common Lisp, and runs on the Carnegie Group Knowledge Engineering Workstation, Digital Equipment Corporation VAX/VMS computers, and the Symbolics 3600. SRL+ is also implemented in Franz Lisp, and runs under the UNIX operating system on the VAX. In the future, SRL+ will be available on a wide spectrum of machines and operating systems.

## Training

Training and support services are provided to each SRL+ purchaser. An intensive, two-week, hands-on tutorial help users become proficient in developing knowledge-based programs using the SRL+ package.

## Applications

The SRL+ prototype has been employed in a range of applications including:

1. **Callisto**: a project management system which focuses on the semantic representation of activities and production configurations (Fox, Greenberg, & Sathi, 1984).
2. **INET**$^{TM}$: A corporate distribution analysis system which models and simulates a corporation's manufacturing, distribution, and sales organization (Reddy & Fox, 1983).
3. **ISIS**: A production management system which models, schedules,and monitors activities (Fox, 1983; Fox & Smith, 1984).
4. **Rome**: A quantitiative reasoning system for long range planning (Kosy et al., 1983; Kosy & Wise, 1984).
5. **PDS**: A rule-based architecture for the sensor-based diagnosis of physical processes (Fox et al., 1983).

# 2. Language Overview

This chapter provides a basic overview of SRL's components with a few simple examples and analogies to help understand the concepts involved in the language. Each component is described more fully in the appropriate chapter. It is recommended that the user read this chapter for comprehension and definition of terms before proceeding.

## The Schema: A way of representing information

In SRL, the schema is a way of representing information, just as words are a way of representing information. Words represent information by symbolizing the properties defining an object, process, or concept. The word "photograph" stands for the properties that define a photograph, e.g. paper, rectangular, image, glossy, etc. When a single word is used, all these properties (information) are communicated. Instead of words, SRL uses the schema to symbolize an object, process, or concept. The schema symbolizing the concept of a person might be called the **person** schema.

## Slots and Values: Describe and structure information contained in a concept

A schema represents a whole concept, which is composed of one or more properties or attributes. The schema has slots and values that represent the attributes comprising a concept. For example, the concept of a person might include attributes like hair color and height. Thus, the **person** schema would have a HAIR-COLOR and HEIGHT slot.

Slots and values are a means of representing the attributes, structural, and relational information embodied in the concept the schema represents. For example the concept of a person might include the attributes hair-color and height. Thus, the **person** schema would have a HAIR-COLOR and a HEIGHT slot. Slots are represented by strings like "hair-color" and "height."

Slots are filled with values that describe the attribute the slot stands for. For instance, the hair-color slot could have the value "brown." Slots can have more than one value. If the person schema has a PETS slot, it could be filled with the values "cat," "dog," and "fish."

While words and schemata are alike because both symbolize concepts, the schema surpasses the word by structuring the information it represents. Slots and values order a concept's attributes; a slot represents an attribute belonging to a concept, and the value describes the attribute. Words symbolize concepts without organizing their properties.

## Printing a schema: What does it look like?

Physically, a schema is composed of a schema name (printed in the bold font), and a set of slots (printed in small caps). A schema is always enclosed by double braces, with its name appearing at the top of the display. The following example depicts the **mammal** schema which has two slots, NURSING-METHOD and BIRTH-PROCESS.

```
{{ mammal
    NURSING-METHOD:
    BIRTH-PROCESS:          }}
```

Schema 2-1: The **mammal** schema

Slots can have values that are any lisp expression. Values are placed next to the slot they fill and encased in quotation marks. In the **mammal** schema below, the NURSING-METHOD slot has a value "breast."

```
{{ mammal
    NURSING-METHOD:   "breast"
    BIRTH-PROCESS:  }}
```

Schema 2-2: The **mammal** schema with NURSING-METHOD slot filled with the value "breast."

## Meta-information: Provides additional information about a schema, slot, or value

In a paper, book, or article, footnotes provide additional information about a point being made. The notes may expand on the assertion, or tell a reader where to look for further explanantion. Similarly, meta-information provides information about schemata, slots, and values.

Meta-information provides information about the creation of schema, slot, or value, such as how, when, where, or why it was created. Any part of a schema may have meta-information associated with it -- the schema itself, slots within the schema, and values of a slot.

Meta-information is represented as a schema, and may be "attached" to the part of the schema it has information about, just as numbers attach footnotes to the paragraph they explain. When the schema representing meta-information is "attached" to the original schema, it is referred to as a meta-schema. When attached to the slot, the schema representing meta-information is called a meta-slot. If "attached" to a value, the meta-information schema is known as a meta-value.

The slots of a meta-schema, meta-slot, or meta-value are printed in italics, and indented beneath the schema, slot, or value they are attached to. In the example below, the CREATOR slot and the value "M.

Fox" are indented beneath the value "breast." This means the CREATOR slot with the value "M. Fox" are in the meta-value (meta-information schema attached to a value) of "breast."

```
{{ mammal:
   NURSING-METHOD:   "breast"
             creator:  "M.Fox"    }}
```

**Schema 2-3:** In the **mammal** schema, the value "breast" has a meta-value that contains the CREATOR slot, which has the value "M.Fox."

A schema representing meta-information can have slots just like a schema that symbolizes a concept. The slots of a meta-slot (a meta-information schema attached to a slot) are called facets. The term facet is introduced due to the frequent use of the slots of meta-slots. (see section 4.3). In the schema below, the NURSING-METHOD slot has a facet called *range*. In other words, the NURSING-METHOD slot has a meta-slot which contains a *range* slot.

```
{{ mammal
   NURSING-METHOD:   "breast"
      range:
   BIRTH-PROCESS: }}
```

**Schema 2-4:** The NURSING-METHOD slot has a facet called range.

SRL has four pre-determined facets: domain, range, cardinality, and has-demon. The domain facet is filled with information about the slots a schema can hold. The range facet contains information about the values that may fill a slot, and the cardinality facet holds the number of values a slot can have. The has-demon facet is filled with functions or programs that execute after certain SRL commands are called.

## Inheritance: A means for a schema to get information from other schemata

A schema is a way of representing information, meta-knowledge supplies additional information about a schema and its parts, and inheritance is a means for a schema to receive information from other schemata.

A schema inherits information via its relations. A relation is a special kind of slot that connects two schemata, and, at the same time, allows a schema to get information from other schemata.

Inheritance is always uni-directional; meaning when two schemata are joined by a relation, only one can inherit information. In this sense, SRL's method of inheritance mimics actual human genetic inheritance. Information flows from schema 1 to schema 2, but not from schema 2 to schema 1, just as children receive genetic traits from their parents, but parents do not inherit traits from their children. As a result, the network of relations connecting schemata is hierarchical, descending from top to bottom.

SRL has two pre-detemined relations for connecting schemata: IS-A and INSTANCE. The IS-A relation indicates the schema is prototypical of a class of objects, concept, processes, etc. The **fido** schema is prototypical of the class of dogs; the **fido** schema would have an IS-A slot filled with "dog" reflecting this relationship. The INSTANCE relation means the schema is physical instantiation of a class or concept. The **fido** schema is an INSTANCE of a Scottish Terrier. The important distinction to remember between IS-A and INSTANCE is that the INSTANCE relation refers to a physical representation of a class or concept.

IS-A and INSTANCE relations can be used to form taxonomic hierarchies of schemata. In other words, the two relations can be used to order classes of concepts, objects, or processes. For example, **mammal** IS-A warm-blooded, vertebrate, with a live birth-process. **dog** IS-A mammal, and **fido** is an INSTANCE of a dog. As demonstrated, a series of IS-A and INSTANCE relations order a class (in this case mammals), showing how one member of the class descends from another.

What kind of information does a schema inherit from other schemata? Information stored in slots and values. For instance, if a user accesses the HANDS slot in the **person** schema, and the slot does not currently exist, then SRL checks the **person** schema's relations to other schemata. If the person schema is related to a schema that has a HANDS slot, the value is inherited by the **person** schema. Thus, by way of inheritance, the **person** schema is able to receive information (in the form of a slot) from another schema. Values are inherited in the same way.

## User-defined Relations: Enable control of inheritance

In addition to IS-A and INSTANCE, SRL has a powerful facility allowing users to define their own relations. Since relations enable inheritance, the user can directly control what slots and values a schema inherits by defining his own relations. Relations to control inheritance are constructed with inheritance specs that specify the information a relation can pass to a schema. At present, SRL has five inheritance-specs: inclusion-spec, exclusion-spec, elaboration-spec, map-spec, and an introduction-spec.

The inclusion-spec determines what information a relation may be passed unchanged by the relation. The exclusion-spec specifies the information that may not be passed, therefore a user can construct relations to prevent a schema from inheriting certain information. The elaboration-spec permits a slot in one schema to be elaborated into several slots in another schema; an ECONOMY slot in the **country** schema might be elaborated into the INFLATION, RISE-IN-PRICES, and REAL-WAGE-INCREASE slots in the **Guatemala** schema. Using a relation, a slot or value in one schema can be mapped onto a slot or value in another schema. For instance, the FRONT-LEGS slot of the **mammal** schema might be mapped onto the ARMS slot in the **person** schema. The introduction spec allows new slots to be added to a schema when a relation is attached.

## Paths: Control inheritance

Paths are an alternative to relations for controlling what slots and values a schema can inherit. We can think of paths as roads. The roads we take dictate our direction, likewise paths determine the course of the inheritance search. When a slot is accessed for a value, and the value needs to be inherited, the search for the value can be controlled by specifying the path at the time the value is accessed. The path tells the system exactly what schemata and relations can be searched to inherit the value.

A relation's transitivity may be specified using the path grammar. Transitivity refers to the paths that are acceptable for two schemata to be considered related by a particular relation. Continuing to think of paths as roads, with regard to transitivity, paths are the roads necessary to consider point A connected to point B by a specific route.

Paths have one more important function. They can be used to show how two *types* of schemata can be combined to form a more complex *type*. Schema types are determined according to a certain rule (see section 5.3).

## Contexts: Permit version management and alternate worlds reasoning

SRL is equipped with a context facility permitting version management and alternate worlds reasoning with SRL models. Each context acts as a virtual copy of a database in which schemata are stored. In the copy, schemata can be created, modified, and destroyed without altering the original context. Thus, a user can test simulations, or create a model without affecting the orginal context or database.

The context mechanism also saves time and space. Contexts are structured as trees, where each context may inherit the schemata present in its parent context. Because a context can inherit from its parent context, only schemata used in the child context are represented there. This avoids copying schemata that will never be used in the child context, which saves space and time. These savings can be important when dealing with large databases.

## Error Handling: The system's response

SRL's error-handling mechanism is schema-based. When an error occurs, an instance of the **SRL-error** schema is generated. The **error** schema contains slots that hold information about the error, such as what schema, slot, or value the error is associated with. The schema also has a slot that corresponds to each type of error. These slot are filled with **error-specs** that define the system's response to the error.

# 3. Schema Manipulation

This chapter explains the functions used to create and manipulate a schema. Examples demonstrating the functions follow a set format. The command being illustrated is written on the left side of the example box. The text explaining the command and what it returns is written on the right side of the example box.

## 3.1. Schemata: How to create, delete, and test their existence

A word symbolizes a concept and a schema symbolizes a concept. The word representing a concept is often used to name a schema symbolizing the same idea. For instance, the word 'dictionary' stands for the concept of a dictionary, and is also used to name the schema representing the notion of a dictionary, e.g. the **dictionary** schema. One symbol is used to name another. In SRL, the schema's name is known as a *string*. Every schema in the same context must have a unique string. In other words, one context cannot have two **dictionary** schemata.[2] A string referring to a schema is called the *pointer to the schema.*

Eventhough a context cannot have two schemata with the same name, there is no limit on the number of schemata that can be created in a context. Once created, a schema can be deleted. A user can also test a schema to see if it exists in a particular context.

The schemac command (pronounced "schema cee") creates a schema. In SRL, commands names are generally composed of the part of the schema they affect, and a letter abbreviating the action they perform, eg. schema<u>c</u> <u>c</u>reates a schema. In the example below, the schemac command is used to create the **dog** schema. Next, the ps function is used to pretty print the **dog** schema.

---

```
1. (schemac "dog")
 "dog"
```
*The schemac command creates the* **dog** *schema. SRL + returns "dog."*

```
2. (ps "dog")
          {{ dog      }}
```
*The ps command prints the* **dog** *schema.*

---

Below, the schema-p command is used to test whether the **dog** schema exists, and the schemad command is called to delete the schema.

### Function Summary: schema commands

The words in angle brackets refer to the arguments the function expects. It is not necessary to give the function all the arguments it can requires. For instance, the schemac command executes without specifying the context the schema is to be created in.

**(schemac** <schema> **[** <context> **[** <database> **] ])**
  RETURNS: a pointer to <schema> in <context>.
  SIDE-EFFECT: A schema is created with the name <schema> If the user wants to specify a

[2]The length of the string is restricted by the database system.

| | |
|---|---|
| 1. (schema-p "dog") | *The schema-p predicate is used to see if the* **dog** *schema exists.* |
| t | *The function returns true because the schema does exist.* |
| 2. (schemad "dog") | *The schemad command deletes the* **dog** *schema.* |
| t | *The function returns t, meaning the schema existed, and has been deleted.* |

database, he must also specify a context.

(ps <schema> [<context>]))
RETURNS: t
NOTE: Pretty-prints <schema> as found in <context>. All printing is done to $outport$

**(schemad** <schema> [ <context> ] )
RETURNS: t if <schema> existed in <context>, otherwise nil.
SIDE-EFFECT: <schema> is deleted from <context>. All slots, values, and any attached meta-schemata, including those attached to slots and values are also deleted.

**(schema-p** <schema> [ <context> ] )
RETURNS: t if <schema> is a pointer to an object of type schema in <context>, otherwise nil.
NOTE: If <schema> exists in a ancestor context of <context>, then t is also returned.

## 3.2. Creating and manipulating meta-schemata

Meta-information provides information about the creation of a schema, slot, or value, e.g. how, when, where, or why it was created. Meta-information is represented as a schema, and is attached to the part of the schema it provides information about. When the meta-information schema is attached to a schema, it is known as a meta-schema. If the meta-information schema is attached to a slot, it called a meta-slot. If attached to a value, the meta-information schema is known as a meta-value.

A meta-information schema is created and manipulated like any other schema. A tip to the user when creating a meta-schema: mschemac command does not create a meta-schema; it merely attaches a meta-schema that has already been created (with schemac) to a schema. Therefore, make sure to create the meta-schema with schemac before calling mschemac. The following example illustrates the procedure for creating a meta-schema with the mschemac command. Below, the **dog-meta** schema is created and attached to the **dog** schema.

---

| | |
|---|---|
| 1. (schemac "dog-meta")<br>"dog-meta" | *The* **dog-meta** *schema is created.* |
| 2. (mschemac "dog" "dog-meta")<br>"dog-meta" | *The* **dog-meta** *schema is attached to the* **dog** *schema.* |

---

Instead of creating a meta-schema, the user can let the system generate a meta-schema. This is done with the mschemag command -- g stands for "get." Provided no meta-schema already exists, when the user types:

```
(mschemag "car")
```

SRL will create a meta-schema for the **car** schema. If a meta-schema already exists, the name of the meta-schema is returned.

System generation of meta-schemata is controlled by the switch **$meta-schema**. The **$meta-schema** switch is a variable whose value is the name of a schema specified by the user. There are two ways to specify the value of **$meta-schema**: the srl-set function, or the user may include the value of **$meta-schema** as an argument in the mschemag function. Srl-set is very similar to the lisp function, SETQ. To specify the value of the switch using the srl-set function, the user should type:

```
(srl-set "$meta-schema" "auto-meta"),
```

where "auto-meta" is the value of **$meta-schema**. To specify the value when using the mschemag function the user would type:

```
(mschemag "car" "auto-meta")
```

The third argument, "auto-meta," specifies the value of the **$meta-schema** switch.

As long as the switch has a non-nil value (in other words, the switch has a schema name as its value), the system automatically generates a meta-schema when mschemag is called. The switch may be set to **nil**, in which case, the system will not generate meta-schemata automatically when mschemag is used. If the user does not set the value of **$meta-schema**, the switch is set to the default, **schema**.

After setting the value of the switch, and calling the mschemag command, the system creates a meta-schema. The system generated meta-schema is linked to the value of the **$meta-schema** switch by an INSTANCE relation. For example, if the value of **$meta-schema** is "auto-meta," then the meta-schema generated from the previous example would be linked to "auto-meta" by an instance relation.

The following example demonstrates how a meta-schema is created by SRL using the mschemag command. The example also shows how a system-generated meta-schema is made an instance of the **$meta-schema** switch, whose value is specified in the mschemag function call.

| | |
|---|---|
| 1. (schemac "apple")<br>"apple" | *The* **apple** *schema is created.* |
| 2. (mschemag "apple")<br>"1 + 73621434500020784" | *The* **apple** *schema's system-generated meta-schema is returned.*[3] |
| 3. (ps "apple")<br>{{ **apple**<br>        ⟨*instance*: schema⟩ }} | *The ps command prints the* **apple** *schema. The* INSTANCE *slot with the value "schema" means* **apple's** *meta-schema is an instance of* **schema,** *which is the value of the* **$meta-schema** *switch.* |

As with other schemata, meta-schemata can be deleted and the user can test their existence. The mschema-p command determines if a schema has a meta-schema attached to it. The next example illustrates the mschema-p command

| | |
|---|---|
| 1. (mschema-p "dog")<br>"dog-meta" | *Mschema-p tests whether a schema has a meta-schema. The name of the meta-schema is returned* |
| 2. (mschemag "dog")<br>"dog-meta" | *The mschemag command retrieves the meta-schema of* **dog** *schema.* |

## Function Summary: meta-schema commands

The ⟨generate-switch⟩ argument in the mschemag function refers to the value of the **$meta-schema** switch.

(**mschemag** ⟨schema⟩ [⟨generate-switch⟩ [⟨context⟩ ] ] )

RETURNS: the pointer to the meta-schema attached to ⟨schema⟩ if one exists, otherwise **nil**.

SIDE-EFFECT: If either ⟨generate-switch⟩ or the switch $meta-schema is non-nil and a meta-schema does not already exist, then one is created and its pointer returned. If ⟨generate-switch⟩ is the name of a schema, then the meta-schema is made an INSTANCE of that schema. Otherwise, if $meta-schema is non-nil, the meta-schema is made an INSTANCE

[3] When the meta-schema is generated by the system, the user does not need to create the meta-schema using the schemac command before calling a meta-schema command.

of the schema held as the value of the $meta-schema switch.

(**mschemac** <schema> <meta-schema> [ <context> ])
RETURNS: <meta-schema>
SIDE-EFFECT: <meta-schema> is attached to <schema>. The previous meta-schema, if not the same as <meta-schema>, is deleted.

(**mschemad** <schema> [ <context> ])
RETURNS: t if a meta-schema exists for <schema>, otherwise nil.
SIDE-EFFECT: Deletes the meta-schema associated with <schema>.

(**mschema-p** <schema> [ <context> ] )
RETURNS: the meta-schema associated with <schema> if it exists, otherwise nil.
NOTE: Useful for determining if a meta-schema already exists for schema without automatically generating one via mschemag.

## Relevant switches:

**$meta-schema**: if the setting is non-nil, a meta-schema is generated automatically, and linked to the schema name held by the switch. The default setting for the switch is **schema.** If the switch is set to nil, no meta-schemata are generated by the system.

# 4. Slots and Values

This chapter details the creation of slots, values, meta-slots, and meta-values, and presents commands to manipulate them. The chapter also introduces four special facets: domain, range, cardinality, and has-demon, and explains their functions.

## 4.1. Slot Definition

A schema symbolizes a concept, and slots represent and structure the attributes embodied in the concept. For example, the **telephone** schema stands for the concept of a telephone, and the slots CORD and METHOD-OF-DIALING represent the attributes that comprise the notion of a telephone.

Using the slotc command, a slot can be created for any attribute. There is no limit on the number of slots a schema can have. However, like a schema, a slot must have a unique name; that is, the telephone schema cannot hold two CORD slots. A slot name is a string such as "method-of-dial."

Below, the slotc function is used to make a HAS-COLOR slot in the **dog** schema.

---

```
1. (slotc "dog" "has-color")
"has-color"
```

*The slotc command creates the* HAS-COLOR *slot in the* **dog** *schema.*

```
2. (ps "dog")

      {{ dog
            HAS-COLOR:        }}
```

*The* **dog** *schema is printed.*

---

After being created, slots may be deleted using the slotd command. But, when a slot is deleted, so is its value. In other words, if the slotd command is used to delete the CORD slot, which has a value of 50 feet (meaning the cord is 50 feet long), both CORD and its value are deleted once the command is called.

With the slote function, a user can determine if a slot currently exists in a schema, or if it may be inherited from another schema. Let's say the slote command is called to see if the **dog** schema contains a BIRTH-PROCESS slot. If **dog** schema itself does not have the slot, but **dog** is related to the **mammal** schema which has the slot, then slote returns "t" because the slot may be inherited from the **mammal** schema.

Slot-all returns a list of slots accessible along a path, in other words, slots that may be inherited through a particular set of relations. The slot-elab function returns the slots that are an elaboration of a particular slot. The slot-elab command is explained and illustrated later in the manual.

This example demonstrates the slote and slotd functions. Slote tests whether the STEM slot exists in the **pear** schema, and slotd is called to delete the STEM slot from **pear**.

---

```
1. (schemac "pear")
```

*The* **pear** *schema is created.*

| | |
|---|---|
| "pear" | |
| 2. (schemac "stem")<br>"stem" | *The* **stem** *schema is created.* |
| 3. (slotc "pear" "stem")<br>"stem" | *Stem is made a slot in* **pear.** |
| 4. (slote "pear" "stem")<br>**t** | *Slote tests whether* STEM *exists or can be inherited by the* **pear** *schema.*<br>*The function returns "pear" because the slot exists in* **pear.** |
| 5. (slotd "pear" "stem")<br>"stem" | *The* STEM *slot is removed from* **pear.**<br>*"stem" is returned because the slot existed in the* **pear** *schema.* |

---

## Function Summary: slot commands

While it may not be necessary to give a function all the arguments it can take, if the user wants to skip over one argument, and specify the next, he must type nil in place of the missing argument. For instance, when using the slote command, the user may not want to specify the context argument. In this case, he would type:

```
(slote "dog" "favorite-food" nil (repeat (step "is-a" t) 0 inf))
```

The argument following nil, (repeat (step "is-a" t) 0 inf), is a path stating that only schemata related to the **dog** schema by an "is-a" relation are to be searched for the FAVORITE-FOOD slot. For explanation of paths see chapter(6).

Other slot manipulation functions include:

(**slot** <schema> [ <context> ])
RETURNS: the names of the slots directly defined in <schema>.
NOTE: It does not return the names of slots that are accessible by inheritance.

(**slote** <schema> <slot> [ <context> [ <path> ] ])
RETURNS: t if the slot exists in <schema> or is accessible by inheritance, otherwise nil.

(**slot-all** <schema> <path> [ <context> ])
RETURNS: a list of slots accessible along the path specification (see section 6.1 for a description of paths).

(**slotc** <schema> <slot> [ <context> ])
RETURNS: <slot>

SIDE-EFFECT: Creates a <slot> slot in <schema>. If <slot> is a **relation**, then it is noted by the system for use for inheritance. (See section 4.4)

NOTE: If the **$restrict** switch is t, then <schema> is tested to see if it satisfies the *domain* facet restriction of <slot>. (See section 4.3.1)

(**slotd** <schema> <slot> [ <context> ])

RETURNS: <slot> if <slot> existed in <schema>, otherwise nil.

SIDE-EFFECT: Deletes the <slot> slot from <schema> and the attached meta-slot. In addition, the values of <slot> and their meta-values are deleted.

(**slot-elab** <schema> <slot> [ <context> [ <path> ] ])

RETURNS: the slots in <schema> that are elaborations of <slot>. (See section 5.1.5).

(**slot-unlab** <schema> <slot> [ <context> [ <path> ] ])

RETURNS: the slot of which <slot> is an elaboration.

## 4.1.1. Meta-slots: Creating and accessing them

Meta-information about a slot is represented as a schema. The meta-information schema attached to a slot is called a meta-slot. The mslotc command attaches a meta-slot to a slot. However, just like the mschemac command, the user must first create the meta-slot with the schemac command before calling mslotc. How can a meta-slot be created with schemac? Because the meta-slot is actually a schema representing meta-information about a slot. The schemac command creates the schema representing meta-information; mslotc attaches it to a slot.

The example below illustrates the procedure for creating a meta-slot with the mslotc command.

| | |
|---|---|
| 1.(schemac "ears") "ears" | *The* **ears** *schema is created.* |
| 2.(slotc "dog" "ears") "ears" | *The* **ears** *schema is made a slot of the* **dog** *schema.* |
| 3.(schemac "lopsided")~r "lopsided" | *The* **lopsided** *schema is created.* |
| 4.(mslotc "dog" "ears" "lopsided") "lopsided" | *The* **lopsided** *schema is made a meta-slot of the* EARS *slot.* |

As with the meta-schema, the user can let the system generate a meta-slot instead of creating one himself. To do this, the user calls the mslotg command. System creation of meta-slots is regulated by the **meta-slot** switch. **$meta-slot** operates identically to the **$meta-schema** switch; **$meta-slot** is a variable whose value is a schema name which may be specified by the user. The value of the **$meta-slot** switch is set the same way the **$meta-schema** switch is set, with the srl-set function or when the mslotg command is used (see p.18.)

If the switch has a non-nil value, the system will generate a meta-slot whenever mslotg is called. If the user does not give the variable a value, the switch is set to hold the **slot** schema as a default. The **slot** schema is described later in this chapter.

When a meta-slot is generated by SRL, it is linked to the value of **$meta-slot** switch by an INSTANCE relation. If the user gives the **meta-slot** switch a value of **height-meta**, the meta-slot generated by the system will be an instance of this schema.

The next example demonstrates how the system generates a meta-slot, and how it is connected to the value of **$meta-slot** switch.

---

| | |
|---|---|
| 1. (schemac "person")<br>"person" | *The* **person** *schema is created.* |
| 2. (shemac "height") | *The* **height** *schema is created.* |
| 3. (slotc "person" "height")<br>"height" | *The* **height** *schema is made a slot in the* **person** *schema.* |
| 4. (mslotg "person" "height" "height-meta")<br>"3 + 2670269573" | *The mslotg command is used to generate a meta-slot for the* HEIGHT *slot. The value of the* **$meta-slot** *switch is specified by the argument "height-meta".* |
| 5. (ps "person") | *The* **person** *schema is pretty-printed.* |

```
{{ person
      HEIGHT:
            instance: "height"]
            [domain: "person"]
            slot: "height"]       }}
```

*The* HEIGHT *slot's meta-slot is an instance of "height," which is the value of* **$meta-slot** *switch.*

---

## Function Summary: meta-slot commands

(**mslotg** <schema> <slot> [ <generate-switch> [ <context> ] ])

RETURNS: the pointer to the meta-slot associated with the <slot> slot of <schema> if one exists, otherwise nil.

SIDE-EFFECT: If <generate-switch> or the switch **$meta-slot** is non-nil, and a meta-slot does not already exist, one is created. The meta-slot is linked to the schema that corresponds to the name of the slot the meta-slot holds information about. If there is no schema that corresponds to the name of the slot, one is created. If <generate-switch> holds the name of a schema, then the schema representing the slot to which the meta-slot is attached is made an instance of the schema held in <generate switch>. Otherwise, if **$meta-slot** is non-nil, the newly created schema is linked to the switch by an IS-A relation.

**(mslotc** <schema> <slot> <meta-slot> [ <context> ] )
RETURNS: <meta-slot>
SIDE-EFFECT: Attaches the <meta-slot> to <slot>. If a meta-slot is already attached, and is not equal to <meta-slot>, the previous meta-slot is deleted.

**(mslotd** <schema> <slot> [ <context> ])
RETURNS: Returns the meta-slot associated with <slot> in <schema> if one exists, otherwise nil.
SIDE-EFFECT: Deletes the meta-slot associated with <slot> of <schema>, if one exists.

**(mslot-p** <schema> <slot> [ <context> ])
RETURNS: Returns the meta-slot associated with <slot> of <schema> if it exists, otherwise nil.
NOTE: Useful for determining if a meta-slot already exists for slot without automatically generating one via mslotg.

### Relevant switches:

**$meta-slot:** if the setting is non-nil, a meta-slot is generated automatically, and linked to the schema name held by the switch. If the switch is set to nil, no meta-slots are generated by the system.

## 4.2. Value Definition

Slots define attributes of a schema. Recalling the last example, the HEIGHT slot represents the height attribute associated with the **person** schema. Slots can be filled with values which describe an attribute. For example, the HEIGHT slot might be filled with the value five feet. The value five feet describes the height attribute.

Slots may contain a list of values. A value may be any legal lisp object that is not circular. Using circular constructs can cause unpredictable behavior. Values that are strings have special meaning in SRL. If a value is a string that corresponds to schema name (in other words, the value itself is a schema), SRL may perform different actions. Values that are not strings, e.g. numbers, do not receive special treatment.

A variety of functions are provided for filling and retrieving values in a slot. The valuec1 function creates the first value in a slot. Valuec creates any value for a slot; the value does not necessarily have to be placed in the first position of the slot, as is the case with valuec1. When a valuec1 or valuec command is used to create a new value for a slot, all previous values are removed from that slot.

Below, the valuec1 command is used to fill HAS-COLOR slot in **dog** with the single value "brown," then the same slot is filled with a new value.

---

| | |
|---|---|
| 7. (valuec1 "dog" "has-color" "brown")<br>"brown" | *"Brown" is made the first value in the* **dog** *schema.* |
| 8. (ps "dog")<br>{{ **dog**<br>HAS-COLOR: "brown" }} | *The* **dog** *schema is printed with the new value.* |
| 9. (valuec1 "dog" "has-color" "tan")<br>"tan" | *Another value is created for the* HAS-COLOR *slot.* |
| 10.(ps "dog")<br>{{ **dog**<br>HAS-COLOR: "tan" }} | *The* **dog** *schema is printed with the new value "tan The original value "brown" has been destroyed and replaced with "tan."* |

---

SRL has a valuecn command allowing user to specify where a value should be placed in a slot. The "n" in the valuecn command corresponds to the value's position in the slot. Say the user wants to insert a value in the third position of a slot, he would simply type:

```
(valuecn "dog" "has-color" "spotted" 3)
```

The 3 at the end of the function call indicates the value "spotted" should be placed in the slot's third position.

The valuecn function may also be used to insert a value bewteen to values. If a slot has three values, and the user wants to insert a value between the values in the first and second position, he would type:

```
(valuecn "dog" "has-color" "white" 2),
```

and "white" would be placed in the second position accordingly. Contrary to valuec1 and valuec, valuecn <u>does</u> <u>not</u> delete previous values. Instead, the value already in the position to be filled by the new value is moved over one position. For example, the values in the second, before "white" was inserted, is moved over to the third position.

In addition to valuec commands (valuec1, valuec, valuecn), the valuea commands can be used to values in slots. The valuea commands are similar to valuecn because they do not delete previous values when adding a new one. Valuea1 adds a value to the values already present. Valuea adds a list of values to the values in the slot. A list of values is denoted: '(a b c d), where a, b, c, and d are elements of a list. The valuea commands are demonstrated below.

| | |
|---|---|
| 1. (schemac "coat") | *The* **coat** *schema is created.* |
| "coat" | |
| 2. (slotc "dog" "coat") | *The* **coat** *schema is made a* |
| "coat" | *slot in the* **dog** *schema.* |
| 3. (valuea1 "dog" "coat" "short-haired") | *The value "short-haired" is* |
| "short-haired" | *added to the* COAT *slot.* |
| 4. (valuea "dog" "coat" '(spotted black white)) | *A list of values is* |
| (spotted black white) | *added to the value "short-haired" in the* |
| | COAT *slot.* |
| 5. (ps "dog") | *The* **dog** *schema is pretty-printed.* |

```
{{   dog
        COAT: "short-haired" spotted black white  }}
```

SRL provides an assortment of other functions for manipulating values. To delete a value, the user calls the valued command. Valueg retrieves all the values a slot contains. Values may be deleted and retrieved selectively; the user can specify a certain value to delete or retrieve, e.g. the value in the second position of the slot. The value-sort function orders values according to a predicate. Here, the notion of a predicate is identical to a LISP predicate, which is a "question-answering function," that returns **t** or **nil**.[4] If the predicate returns **for value 1, value 1 can precede value 2, or vice-versa.**

**Generally, slots may not be filled if they do not exist in the schema.**[5] **Because the** FAVORITE-FOOD **slot does not exist in the dog schema, it cannot be filled with the value mailmen. However, slots and their values may be inherited from other schemata. (This is discussed at length in chapter 5.** If the slot cannot be inherited, the filling command fails and an error is generated. See chapter 9 for information on error handling.

[4]Touretsky, p.15

[5]See section 9 on error-handling for how to by-pass this restriction.

## Function Summary: value commands

The following functions define and access slot values. They do not copy lisp objects which are the values. The lisp objects passed to these functions are the actual values stored in the schemata. Similarly, the lisp objects retrieved by these functions are the values stored in the schemata. If performing destructive operations on the objects, it is advisable to work with copies of the values returned by these functions.

(**valuec** <schema> <slot> <value-list> [ <context> [ <path> [ <db-switch> ] ] ])
RETURNS: a list of pointers to the units created for the elements of <value-list>.
SIDE-EFFECT: Fills the value of <slot> with <value-list> and removes the previous values of the slot, and deletes their meta-values.

(**valuec1** <schema> <slot> <value> [ <context> [ <path> [ <db-switch> ] ] ])
RETURNS: a pointer to the unit created for <value>.
SIDE-EFFECT: Fills the value of <slot> with the single <value>, and removes the previous values of the slot, including their meta-values.
NOTE: like valuec, but only takes a single value.

(**valuecn** <schema> <slot> <value> <position> [ <context> [ <path> [ <db-switch> ] ] ])
RETURNS: Inserts <value> in <position> in the slot.

(**valuea** <schema> <slot> <value-list> [ <context> [ <path> [ <db-switch> ] ] ])
RETURNS: a list of pointers to the units created for the elements of <value-list>.
SIDE-EFFECT: Adds the list of values to the end of the existing values of <slot>. If the slot is empty, inheritance is performed before adding.

(**valuea1** <schema> <slot> <value> [ <context> [ <path> [ <db-switch> ] ] ])
RETURNS: the unit created for <value>.
NOTE: Like valuea, but takes a single value as an argument.

(**valuez** <schema> <slot> [ <context> [ <db-switch> ] ])
RETURNS: t if the command deleted values from the slot, otherwise nil.
SIDE-EFFECT: Removes all values in the slot, and deletes any attached meta-values.

(**valued** <schema> <slot> <value> [ <context> [ <path> [ <db-switch> ] ] ])
RETURNS: t if the command deleted a value from the slot, otherwise nil.
SIDE-EFFECT: Removes all values equal to <value> in the <slot> slot, and deletes any attached meta-values.

(**valuedq** <schema> <slot> <value> [ <context> [ <path> [ <db-switch> ] ] ])
RETURNS: t if the command deleted a value from the slot, otherwise nil.
SIDE-EFFECT: Removes all values equal to <value> in the <slot> slot, and deletes any attached

meta-values.
NOTE: like valued but uses *eq* instead of *equal*.

(**valuedn** <schema> <slot> <position> [ <context> [ <path> [ <db-switch> ] ] ])
RETURNS: Deletes the value in <position> from the slot.

(**valued-pred** <schema> <slot> <predicate> [ <context> ] [ <path> [ <db-switch> ] ] ])
RETURNS: t if any values were deleted because of the command, otherwise nil
NOTE: Deletes values from the slot that cause <predicate> to return t. The predicate is passed the <schema> <slot> <context> and a value.

(**valueg** <schema> <slot> [ <context> [ <path> [ <db-switch> ] ] ])
RETURNS: the list of values contained in <slot> in the given <context>.

(**valueg1** <schema> <slot> [ <context> [ <path> [ <db-switch> ] ] ])
RETURNS: the first value contained in <slot>.
NOTE: Like valueg, but returns only the first value of the <slot>.

(**valuegn** <schema> <slot> <position> [ <context> [ <path> [ <db-switch> ] ] ])
RETURNS: Retrieves the value in <position> from the slot.

(**value-sort** <schema> <slot> <predicate> [ <context> ])
RETURNS: The list of units associated with the the slot, after sorting.
NOTE: Sorts the values in the <slot> of <schema> based on predicate. The predicate is passed to values *val1* and *val2*. It should return a non-nil value if val1 can precede val2. If the predicate returns nil, then val2 will be made to precede val1.

## 4.2.1. Units: Value/meta-value pairs

Values are literally only half of what fills a slot. The actual fillers of slot are called units. A unit is a value/meta-value pair. The uvalue commands manipulate units. The difference between uvalue commands and the value functions (eg valuec, valuea) introduced earlier is that value commands only return values, whereas uvalue commands return the whole unit, the value/meta-value pair. After a value command is called, a null meta-value is stripped away from the unit, and only the value is returned to the user.[6]

Meta-values may be created by the system or the user. To generate meta-values with SRL, the user

[6]The null meta-value that is stripped away after a value command is called, is created when the value function is first executed.

calls the munitg command. Like **$meta-schema** and **$meta-slot**, **$meta-value** is a variable whose value is a schema name specified by the user. The value of **$meta-value** is set with the srl-set function or when the munitg command is called (see p.18.)

If the switch has a non-nil setting, a meta-value is automatically generated by the system when the user calls munitg. The default setting is the **value** schema. The **value** schema and its slots are described below.

---

```
{{ value
     CREATOR:
     CREATION-TIME: }}
```

**Schema 4-1:** The **value** schema

---

*creator*: Person or process that created the value.

*creation-time*: The time when the value was placed in the slot.

When the meta-value is created, it is linked to the **$meta-value** switch by an INSTANCE relation. If the value of **$meta-value** is **value** schema, a system generated meta-value would be linked to the **value** schema by an INSTANCE relation. This INSTANCE relation makes the slots in the **value** schema accessible in the meta-value generated by the system.

Below, several of the commands that access units and meta-values are demonstrated.

## Function Summary: unit commands

The following commands enable the user to access units. A meta-value is manipulated just like any other schema.

(**uvalued-pred** <schema> <slot> <predicate> [ <context> ])
  RETURNS: t if any values were deleted because of the command, otherwise nil
  NOTE: Deletes values from the slot that cause <predicate> to return t. The predicate is passed the <schema> <slot> <context> and a unit.

(**uvalueg** <schema> <slot> [ <context> [ <path> [ <db-switch> ] ] ])

(**uvalueg1** <schema> <slot> [ <context> [ <path> [ <db-switch> ] ] ])
  RETURNS: The first value-unit in <slot> in the given <context>.
  NOTE: Like valueg, but returns only the first value unit in the <slot>.

(**uvaluegn** <schema> <slot> <position> [ <context> [ <path> [ <db-switch> ] ] ])
  RETURNS: Retrieves the <position> unit from the slot.

---

```
1.(uvalueg "dog" "has-color")
({|1value| "brown" nil "1 + 935214639221906845" "dog" "has-color"
"$root-context" nil} {|1value| "spotted" nil "2 + 93521463922190684"
"dog" "has-color" "$root-context" nil})

2.(uvaluegn "dog" "has-color" 2)
({|1value| "spotted" nil "2 + 93521463922190684" "dog" "has-color"
"$root-context" nil})

3.(setq x (uvaluegn "dog" "has-color" 2))
{|1value| "spotted" nil "2 + 93521463922190684" "dog" "has-color"
"$root-context" nil}
```

*The unit of the second position of the* HAS-COLOR *slot is returned. The unit contains both the value and the meta-value of the position. The variable X is set to hold the unit as its value. When the unitg command is used, which takes the unit as an argument, the entire unit need not be typed to call the command. X is typed in place of the unit.*

```
4.(unitg x)
"spotted"
```

*The value of the unit is returned.*

---

**(unitg** <value-unit>)

**(munitg** <value-unit> **[** <generate-switch> **[** <context **[** <database> **] ] ]**)

NOTE: If either <generate-switch> or the switch $meta-value is non-nil, and a meta-value does not already exist, one is created, and its pointer returned. If <generate-switch> is the name of a schema, then the meta-value is made an INSTANCE of that schema. Otherwise, if $meta-value is non-nil, the meta-value is made an INSTANCE of the schema held as the value of the $meta-value switch.

**(munitc** <value-unit> <meta-value>)

RETURNS: <meta-value>

**(munitd** <value-unit> **[** <context> **]**)

RETURNS: **t**

**(munit-p** <value-unit>)

NOTE: Useful for determining if a meta-value already exists for a value-unit without automatically generating one via munitg.

### Relevant switches:

**$meta-value:** if the setting is non-nil, a meta-slot is generated automatically, and linked to the schema name held by the switch. If the switch is set to nil, no meta-values are generated by the system.

## 4.3. Facets: Slot meta-information

When a meta-slot is created, it is linked to the **$meta-slot** switch by an INSTANCE relation. If the user does not specify the value of the switch, it is set to hold the **slot** schema. Because the meta-slot is related to the **slot** schema by an INSTANCE relation, the meta-slot can inherit the slots in the **slot** schema. The **slot** schema is displayed below.

```
{{ slot
     RANGE: t
          range: <value-op>
     DOMAIN: t
          range: <schema-op>
     CARDINALITY: (0 inf)
     HAS-DEMON:
          range: (type "instance" "demon")
     INVERSE:
     HAS-META-VALUE: }}
```

**Schema 4-2:** The **slot** schema

The slots of a meta-slot are called facets. Thus, if a meta-slot is created that is an instance of the **slot** schema, that meta-slot's facets would be the slots in the **slot** schema.

The first four slots in the **slot** schema represent the standard facets of SRL. Each facet may have zero or more values. INVERSE holds the inverse of the slot, eg. the inverse of the BLACK slot might be the WHITE slot. A slot's inverse is used in auto-linking schemata (see section 4.4). While users may define their own facets, SRL gives the first four facets special meaning when interpreting slots and values.

### 4.3.1. Domain and Range Facets

The *domain* facet is used to specify restrictions on the form and type of schemata where slot may reside. The *range* facet is used to restrict the values a slot may hold. The restrictions that are the values of the domain and range facets are like tests. For example, restrictions can be used to test whether a slot can be created or inherited by a certain schema. If the schema satisfies the restriction, the slot may be inherited or created . Elements of the restriction grammar can be combined to test many things about a schema, slot, or value.

The domain and range facets may have only one value, and the value must follow the grammar given below. Examples using the restriction grammar follow an explanation of the grammar's terms.

```
<schema-op> ::= (meta <schema-op>) | (slot <slot> <slot-op>) |
          (all <slot-op>) | (some <slot-op>) |
          (or <schema-op>+) | (and <schema-op>+) |
          (type <relation> <schema>) | (pred <function>)
          (not <schema-op>) | <lit> | t[7]

<slot-op> ::= (meta <schema-op>) | (schema <schema-op>) |
          (all <value-op>) | (some <value-op>) |
          (or <slot-op>+) | (and <slot-op>+) |
          (not <slot-op>) | <lit> | t

<value-op> ::= (schema <schema-op>) | (map <value-op>+) |
          (all <value-op>) | (some <value-op>) |
          (meta <schema-op>) | (pred <function>)
          (or <value-op>+) | (and <value-op>+) |
          (not <value-op>) | <lit> | t

<range-restriction> ::= <value-op>

<domain-restriction> ::= <schema-op>
```

**Figure 4-1:** The restriction grammar

- **meta**: Takes the meta schema associated with the schema, slot, or value, and performs a test on it. For instance, the meta-schema of a schema might be tested to see if is type "instance" "dog." The test fails if there is no meta information (It is automatically created if the appropriate switch is on).

- **slot**: Performs the requested test on the slot specified as the <slot> argument. The <slot> is inherited if necessary. The test fails if <slot> does not exist.

[7] In the restriction grammar, "lit" stands for literal.

- **all**: Succeeds if every element succeeds. From a schema it tests every slot. From a slot, it tests every value. From a value, it tests every element of the list that is a value. Note that the value must be a list.

- **some**: The test succeeds if at least one element passes test. Determines elements as **all** does.

- **or**: Succeeds if one test succeeds.

- **and**: Succeeds if all tests succeed.

- **not**: Succeeds if the test fails.

- **type**: Succeeds if (r-test <current-schema> <relation> <schema>) succeeds. Fails if element is not a schema.

- **<lit>**: Succeeds if the element being tested is equal to <lit>.

- **t**: Always succeeds.

- **schema**: The current element is interpreted as a schema, and the test succeeds when the test on the schema succeeds. The test fails if the element is not a schema.

- **map**: Map interprets the value as a list of values. The list is mapped onto the tests in the restriction. The map succeeds if all of the tests succeed. If the map contains more tests than the value has elements, nil is used in place of the missing value elements. If the value contains more elements than the map contains tests, the extra value elements are ignored.

- **pred**: The predicate provides a means for the user to perform tests that the restriction grammar does not cover. For instance, the value of the slot STREET-NUMBER must be a positive integer. Each **<function>** takes an element and a context as parameters, and returns t if the element succeeds the test within the context.

The **$restrict** switch controls whether the system checks for domain and range restrictions. If the switch is set to **t**, the system checks for restrictions. The switch default setting is **nil**.

Domain and range restrictions may be placed in the domain and range facets respectively, or they can be inherited. For example, say **has-color** were defined as a relation. Relations can have *domain* and *range* restrictions. The meta-slot on the HAS-COLOR slot of **german-shepherd** would be linked, via an INSTANCE relation, to the schema for the **has-color** relation. *domain* and *range* restrictions from the **has-color** relation are inherited by this link, and used in the context of the HAS-COLOR slot of **german-shepherd**. If the inherited *domain* restriction does not allow the HAS-COLOR slot to be placed in the **german-shepherd** schema, adding the slot will result in an error.

Inherited range restrictions behave similarly to inherited domain restrictions. Further discussion of

*The following examples illustrate how the restriction grammar's terms can be combined to test different things about a schema, slot, or value.*

```
(some "foo")
```

*This is an example of a domain restriction. The restriction returns true if the* FOO *slot exists in the schema being tested.*

```
(meta (slot "creator"
 (some (or "peter" "dave"))))
```

*This restriction is used to test a meta-information schema. The test succeeds if the meta-information schema has a* CREATOR *slot, which is filled with some value, either "peter" or "dave."*

```
(map (pred (lambda (val context)
        (<5 val)))
(pred (lambda (val context)
        (>10 val))))
```

*This restriction tests a value. The value satistfies the restriction if the value is a list whose first element is less than 5, and second element is greater than 10.*

```
{{pet-of
   domain: (or (type "is-a" "cat")
        (and (type "is-a" "dog")
        (not (some "guards"))))
                                   }}
```

*This is a domain restriction for the* PET-OF *slot. The slot may only be placed in a schema which is type "is-a" "cat" or "is-a" "dog,"*

*and does not have a* GUARDS *slot.*

*domain* and *range* restriction of relations can be found in chapter 5.

In addition to placing restrictions in domain and range facets or inheriting them, the user can call the restriction-test function to check for restrictions themselves. The restriction-test function is displayed below.

**(restriction-test** <restriction> <schema> **[** <slot> **[**<value-list> **[** <context> **] ] ])**

RETURNS: t if the restriction test succeeds, nil otherwise.

NOTE: This function determines the type of restriction test based on the number of parameters given. If no <slot> or <value-list> is provided, then the restriction is performed on the schema. If no <value-list> is provided, then the restriction-test is performed on the slot. If there is a <value-list>, then the restriction-test is performed on each element of the list of values. It succeeds if each value passes the restriction-test. **nil** may be used for any parameter that is to be skipped.

## 4.3.2. Cardinality Facet

The *cardinality* facet is used to restrict the number of values a slot may contain. The value of the *cardinality* facet describes the minimum and maximum number of values allowed. **inf** means a slot may contain any number of values. Cardinality checking is controlled by the **$cardinality** switch.

```
<cardinality>::= (<min> <max>)
<min>::= < >0 number>
<max>::= <number> | inf
```

**Figure 4-2:** The cardinality grammar

In the example below, the NAME of **dog** has a *cardinality* restriction of (0 1). This means that the slot may have zero or one value, but no more. An attempt to add more than one value to the slot will result in an error.

```
{{ dog
    NAME:
        cardinality: (0 1)  }}
```

**Schema 4-3:** The **dog** schema with a cardinality restriction

## 4.3.3. Demon Facet

Reactive processing is provided by the specification and attachment of a demon schema to any slot in a schema. The purpose of the demon schema is to specify a function, and the conditions under which it is executed. The **demon** schema is defined as follows:

```
{{ demon
     ACCESS: <access>+
          range: (type "IS-A" "SRL access function")
     ACCESSOR:
     ACCESS-VALUE:
     CURRENT-VALUE:
     WHEN:
          range: (or before after)
     ACTION:
          range: <Must be a function definition>
     EFFECT:
          range: (or alter-value block side-effect)  }}
```

**Schema 4-4:** The **demon** schema

A demon may execute before or after access is made, as defined by the WHEN slot. A demon's action is evaluated if the type of access matches the contents of the ACCESS slot. SRL places the name of the function performing the access, in the ACCESSOR slot. If the function is called with a value, the value is placed in the ACCESS-VALUE slot. The current value of the schema and slot is placed in CURRENT-VALUE (note if the value is inherited and not copied it is still the current value for after demons). Note: Because the units placed in the ACCESS-VALUE, CURRENT-VALUE, and ACCESSOR are not copied, the user should not modify them. The contents of the ACTION slot are one or more lambda expressions or functions whose parameters are:

1. The name of the schema in which the demon resides.

2. The name of the slot in which the demon resides.

3. The name of the demon schema.

A demon ACTION may affect the function access in one of three ways, as specified in the EFFECT slot:

1. **side-effect**: the execution of the action has no direct affect on the access.

2. **alter-value**: The value used or returned by the function is altered depending on the type of demon. If the demon is a before demon, the value used with the function is altered before the function is executed. If the demon is an after demon, the value returned to the user is altered.

3. **block**: A block demon prevents the function called by the user, e.g. valuec, from being evaluated. For instance, if the user wants a value placed in memory, they must perform the function themselves, or else the block demon will prevent the value from being stored in memory. The demon is undefined after the function has executed.

The ACTION function returns either a list of units or a list of values. SRL will differentiate between them. Because units are meaningless for valued, valuedq, valuedn, and valued-pred, only a list of a single value will be expected.

Execution of demons is controlled by the **$demon** switch. A value of *t* enables the checking and evaluation of demons attached to slots during slot access. Default is *nil*.

In the example below, a *has-demon* facet, filled with "mood-demon," has been added to the MOOD slot of **fido**. The **mood-demon** schema is also shown. This demon is only interpreted after a valuec or valuec1 command has been used on the MOOD slot in **fido**. The demon has an effect type of side-effect, meaning the demon will not affect the value put into the MOOD slot of **fido**, but may cause some side-effect to take place in the model. When a **valuec** command is used, the function change-tail-fn is evaluated, given the parameters "**fido**," "MOOD," and "mood-demon." The evaluation causes the TAIL slot of **fido** to change to "wagging," if **fido**'s MOOD slot has been changed to hold "happy".

```
{{ fido
    MOOD:
      has-demon: "mood-demon"
    TAIL:             }}

{{ mood-demon
    INSTANCE: "demon"
    ACCESS: "valuec" "valuec1"
    WHEN: after
    ACTION: change-tail-fn
    EFFECT: side-effect   }}

 (defun change-tail-fn (schema slot demon)
        (cond ((equal (valueg1 demon "current-value") "happy")
                (valuec1 schema "tail" "wagging"))))
```

**Schema 4-5:** The **fido** schema with "mood-demon" and the **mood-demon** schema

## Function Summary: commands used with facets

**(restriction-test** <restriction> <schema> [ <slot> [<value-list> [ <context> ] ] ])

RETURNS: t if the restriction test succeeds, nil otherwise.

NOTE: This function determines the type of restriction test based on the number of parameters given. If no <slot> or <value-list> is provided, then the restriction is performed on the schema. If no <value-list> is provided, then the restriction-test is performed on the slot. If there is a <value-list>, then the restriction-test is performed on each element of the list of values. It succeeds if each value passes the restriction-test. **nil** may be used for any parameter that is to be skipped.

**restriction grammar:** tests whether a slot or value satisfies a schema's domain and range restrictions.

### Relevant switches:

**$meta-slot:** if non-nil, a meta-slot is generated automatically, and linked to the schema name held by the switch.

**$restrict:** if the switch is set to t, then domain and range restrictions are checked. If set to nil, they are ignored.

**$demon:** If the switch is set t, then demons are evaluated and executed, otherwise demons are ignored.

## 4.4. Schemata as Values: Auto-linking

When a slot is given a value that is a schema, an inverse relation is put in place. The inverse relation connects the slot and the value filling it by creating the reverse of that slot in the value schema. For example, suppose the **has-color** slot's inverse is **color-of**, and **fido** contains a HAS-COLOR slot. Suppose the HAS-COLOR slot is filled with "brown," and that "brown" is also a schema. When the value "brown" is added to the HAS-COLOR slot of **fido**, the COLOR-OF slot of **brown** is also filled with the value "fido". The name of the inverse relation is taken from the inverse slot of the meta-schema attached to the slot being filled.

```
{{ fido
     INSTANCE: "dog"
     HAS-COLOR: "brown"  }}

{{ brown
     COLOR-OF: "fido"   }}

{{ has-color
    INVERSE: "color-of"   }}
```

**Schema 4-6:** How inverse relations connect schemata

Here is another example of inverse relations. The HAS-WINGS slot, whose inverse is WINGS-OF, is filled with "feathers," which is a schema. When "feathers" is added to the slot, the **feathers** schema is given a WINGS-OF slot that is filled with the value "bird."

```
{{ bird
    INSTANCE: "mammal"
    HAS-WINGS: "feathers" }}

{{ feathers
    WINGS-OF: "bird"    }}

{{ has-wings:
    INVERSE: "wings-of"  }}
```

**Schema 4-7:** The inverse of the HAS-WINGS slot

A slot's inverse is determined when it is first needed. If the information necessary to determine the inverse of a slot is absent, SRL creates the information. Whenever an inverse link needs to be put in place, the system takes the inverse from the meta-slot of the slot being filled (with a value that is a schema), and checks to see if a schema with that name exists. If none exists, one is created. If the inverse is not specified, it too is created. SRL generates inverses by appending the suffix *+inv* to the slot name. The schema associated with the inverse is created, and given an inverse of the original slot. For instance, if HAS-COLOR had no inverse, its inverse would be HAS-COLOR+INV. The **has-color** and **has-color+inv** schemata would reflect this relationship.

The following would be created if there was no **has-color** schema.

```
{{ has-color
    INVERSE: "has-color+inv"  }}

{{ has-color+inv
    INVERSE: "has-color"  }}

{{ brown
    HAS-COLOR+INV: "fido"     }}
```

**Schema 4-8:** The inverse of the HAS-COLOR slot

The system switch **$inverse** controls whether inverse links are always created, never created, or created only if the slot is defined as a relation. The switch **$inverse** has three legal settings: **all**, **relations**, and **none**. If the switch is set to **all**, auto-linking always takes place. If set to **none**, no auto-linking takes place. If set to **relations**, auto-linking is only performed for relations. **relations** is the default setting for **$inverse**.

## Relevant switches:

**$inverse:** controls the creation of inverse relations. The switch has threee legal settings: all, none, relations.

# 5. Schema Relations

This chapter explains the workings of inheritance -- how relations connect schemata and how relations control inheritance.

A relation is a special kind of slot which connects two schemata. The relation is a passageway that allows two schemata connected by the relation to inherit information from each other. Two relations are already defined in SRL: IS-A and INSTANCE. An **is-a** relation indicates one schema "is-a" typical member of a class. The instance relation suggests one schema "is-a" typical member of a class, but also implies there is a tangible object.

Two schemata are related if one schema appears as the value of a slot in the other schema. The slot where this value appears is the relation. For example, if the slot, INSTANCE, is added to the **fido** schema, and given a value of **dog**, then **fido** is related to **dog** by the INSTANCE relation. The INSTANCE relation indicates that **fido** is an instance, or a particular example of a **dog**. Any slot may be used to form a relation, but only certain relations allow slots and their values to be inherited between schemata. The instance relation has been defined to pass all slots and values from the schema that is the value of the INSTANCE slot to the schema containing the instance slot, e.g. all slots and values pass from **dog** to **fido**.

Inheritance occurs automatically whenever information is needed. If the value of **fido**'s HAS-COLOR slot is accessed (the actual command is *(valueg1 "fido" "has-color")*), first the schema **fido** is examined for the value. If no value is found, SRL retrieves relations between **fido** and other schemata so the value may be inherited. (In this case there is only one relation, INSTANCE). Next slots of the related schema (in this example **dog**) are examined, and the value "brown" is found. The access function will return "brown" as the result of the query.

```
{{ fido
    INSTANCE: "dog"   }}

{{ dog
    IS-A: "mammal"
    HAS-COLOR: "brown"
    INSTANCE+INV: "fido"  }}
```

**Schema 5-1:** A system-generated inverse relation

This is just one example of a relation. There are many relations, varying in which slots and values can be inherited between schemata. Any slot may be a relation, although the information passed may be different from another relation. Unless otherwise specified, a relation passes no information. SRL gives the user the power to specify the semantics of each relation, determining which slots and values are inherited. A number of relations are already defined for the user by SRL. These relations are discussed in Appendix V. Section 5.1 describes how users can create their own relations. It is also possible to specialize a relation in each particular use (see section 5.3.1).

## 5.1. User Defined Relations

One of the most powerful facilities offered by SRL is user definable inheritance relations. Most knowledge representation languages offer the user a set of relations to meet most needs. However, in SRL the user can define relations specifically tailored to the domain currently under study. Each relation is defined by a schema, and is composed of *inheritance-specs* specified in the schema. Inheritance-specs are schemata specifying exactly what information can be passed by the relation. Five kinds of inheritance-specs are currently defined in SRL2: inclusion, exclusion, elaboration, map, and introduction. The construction of relation schemata and the use of inheritance-specs are discussed below.

### 5.1.1. Inheritance-specs

For every relation in SRL, a schema with the same name must exist. The schema for each relation should be linked to the **relation** schema by a network of IS-A relations. The **relation** schema is a prototype for structuring a schema's relations. Because the **relation** is a refinement of the **slot** definition, the **relation** schema is linked to the **slot** schema by an IS-A relation. A **relation** has the semantics of a **slot**, but can also inherit slots and values. The **relation** schema is given below, and the contents of each of its slots defined.

```
{{ relation
    IS-A: "slot"
    TRANSITIVITY: nil
        range: "must follow path grammar"
    INCLUSION:
    EXCLUSION:
    ELABORATION:
    MAP:
    INTRODUCTION:   }}
```

Schema 5-2: The **relation** schema

INVERSE: (Inherited from the slot schema) Every relation has an inverse relation. When a relation is defined, an inverse relation is simultaneously defined, relating schemata in the opposite direction. In the **fido** example, the **dog** schema would automatically acquire an INSTANCE+INV slot, filled with fido (see section 4.4). If no inverse slot name is given by the user, the system automatically generates a name, and creates the corresponding schema. TRANSITIVITY:

Defines the transitivity of the relation (see section 6.3).

| | |
|---|---|
| INCLUSION: | Instantiations of **inclusion-specs** involved in the relation should be placed here. |
| EXCLUSION: | Instantiations of **exclusion-specs** involved in the relation should be placed here. |
| ELABORATION: | Instantiations of **elaboration-specs** involved in the relation should be placed here. |
| MAP: | Instantiations of **map-specs** involved in the relation should be placed here. . |
| INTRODUCTION: | Instantiations of **introduction-specs** involved in the relation. |

Below is an example of a relation schema **pet-of**. Its domain is restricted so schemata must be of *type* IS-A **dog** to have a PET-OF slot. The PET-OF slot's value must be of *type* IS-A **person**. The relation has one **inclusion-spec** named **pet-of-incl-spec**, which is discussed later. **pet-of** has an inverse relation called **has-pet**. Whenever a PET-OF slot is placed in a schema and given a value, a corresponding HAS-PET slot is put in the schema indicated by the value, and filled with the original schema.

```
{{ pet-of
     IS-A: "relation"
     DOMAIN: (type "is-a" "dog")
     RANGE: (type "is-a" "person")
     INVERSE: "has-pet"
     INCLUSION: "pet-of-incl-spec"     }}
```

Notice a simple hierarchy of relations has been defined. Each relation being is linked to the **relation** schema by the IS-A relation. The user may build a hierarchy of any size. It might be interesting to define a **dog-of** relation that is related to the **pet-of** relation by an IS-A slot. The relation would have the same relational properties of the **pet-of** relation, but could be refined in the case of pets that are dogs. The inherited relational properties are represented by inheritance-specs, discussed later in this section.

Users should be careful on two points, however, when making a hierarchy of relations. First, make sure that it is not necessary to be able to interpret a certain relation in order to interpret that relation. For instance, in our **dog-of** example it was necessary to inherit from the **pet-of** relation along an IS-A link. What would happen if instead that was another **dog-of** link? Quickly the system would into endless recursion trying to interpret the **dog-of** relation in order to interpret the **dog-of** relation!

```
{{dog-of
    DOG-OF: pet-of }}
```

The second point of caution is understanding the importance of compiling relations (section 5.2.1). Using relations interpretively is slow, and is especially true when working with a complex hierarchy relations. Interpreting should only be done during the debugging phase of constructing a relation. Relations should be compiled as soon possible if reasonable performance is desired.

## 5.1.2. Inheritance-specs

Inheritance-specs specify what information may be passed between schemata. Five kinds of inheritance-specs are currently defined in SRL2: inclusion-specs, exclusion-specs, elaboration-specs, map-specs, and introduction-specs. Each spec is a schema. To attach an inclusion-spec to a relation, the pointer to the spec must be stored in the relational schema. In the example above, an inclusion-spec, *pet-of-incl-spec*, was stored in the INCLUSION slot of the **pet-of** relation. If this spec had been stored in the EXCLUSION slot of the schema instead, it would be interpreted as an exclusion-spec. A description of the **inheritance-spec** schema is given below followed by a description of each kind of **inheritance-spec.**

```
{{ inheritance-spec
    IS-A + INV: "introduction-spec" "map-spec" "elaboration-spec"
                "exclusion-spec" "inclusion-spec"
    RANGE: t
        range: "must follow <value-op> grammar"
    DOMAIN: t
        range: "must follow <schema-op> grammar"
    CONDITION: t
        range: (or t <predicate>)
    SLOT-DEPENDENCY:
        range: (schema (type "is-a" "slot"))
    VALUE-DEPENDENCY:
        range: (schema (type "is-a" "slot"))  }}
```

**Schema 5-3:** The **inheritance-spec** schema

The DOMAIN, RANGE and CONDITION slots are used in each **inheritance-spec.** The SLOT-DEPENDENCY, and VALUE-DEPENDENCY specs are included in most specs. A description of how the specs are used is given below.

The DOMAIN, RANGE and CONDITION slots are used by each **inheritance-spec**. The SLOT-DEPENDENCY, and VALUE-DEPENDENCY specs are used by most specs. A description of how they are used is given below.

DOMAIN and RANGE: A relation may use many inheritance-specs. Each spec applies only in some instances of the relation. For the inheritance-spec to apply, two criteria must be met. The <domain> schema of the relation must pass the test in the DOMAIN slot. The schema containing to be inherited from (the value in the <relation> slot)(range) must pass the test specified in the RANGE slot. If these two criteria are met, the spec applies in this particular instance of the relation.

CONDITION: The condition specification applies to the whole spec. If the condition is true, then the spec can be used. The contents of this slot are a lambda expression (or schema that can be interpreted as a procedure) with the following parameters: <domain schema of the relation>, <range schema of the relation>, and <inheritance spec schema>.

SLOT-DEPENDENCY: Used when slots are inherited across relations. The meta-slot of the newly inherited slot is linked to the meta-slot of the original slot by the relation stored in the SLOT-DEPENDENCY slot.

VALUE-DEPENDENCY: Used when values are inherited across relations. The inherited meta-values are linked to the original meta-values by the relation held in the VALUE-DEPENDENCY slot.

## 5.1.3. Inclusion-spec

An inclusion-spec specifies what information can be passed unchanged by the relation. The **inclusion-spec** is the most simple and easy to use. The **inclusion-spec** restricts the inheritance of values and slots. A spec may allow the slot to be inherited, but not the value. If the slot may not be inherited across the spec, the restriction on the value will be ignored. A template of an inclusion-spec appears below, followed by a description of each of its slots.

```
{{ inclusion-spec
    IS-A: "inheritance-spec"
    SLOT-DEPENDENCY: "included-from"
    VALUE-DEPENDENCY: "included-from"
    SLOT-RESTRICTION: t
        range: "must follow <schema-op> grammar"
    VALUE-RESTRICTION: t
        range: "must follow <value-op> grammar"  }}
```

**Schema 5-4:** The **inclusion-spec** schema

The definition of the slots not inherited from the **inheritance-spec** schema are:

SLOT-RESTRICTION: Specifies which slots may be inherited across the spec. This uses the restriction grammar that starts with the schema, since the slot may not exist when the spec is evaluated.

VALUE-RESTRICTION: Specifies the values that may be included. The restriction specification for values is used.

The SLOT-DEPENDENCY and VALUE-DEPENDENCY slots are inherited from inheritance-specs, and used to specify the relations for linking meta information. The restrictions (see figure 4-1) allow specification of some class of slots or values for inheritance. Generally these restrictions determine whether a slot or value is in the set specified by the restriction.

Since the domain and range slots are empty, the default value of *all* is inherited from the **inheritance-spec** schema from both slots. This means any schema taking part in the **pet-of** relation (according to the DOMAIN and RANGE slots in the **pet-of** schema) may use this inclusion-spec for performing inheritance. For example, if the RANGE slot had been filled with (or "jones" "smith"), only those schemata related by the **pet-of** relation with the RANGE "jones" or "smith" could use this **inclusion-spec** to perform inheritance. There might be another **inclusion-spec** with range (type "is-a" "person") that would apply in another instance, etc.

---

The **pet-of** inclusion-spec is created by typing:

```
1.(mk-schema "pet-of-incl-spec" ("instance" "inclusion-spec")
   ("slot-restriction" (type "is-a" "address") ))
"pet-of-incl-spec"

 {{ pet-of-incl-spec
        INSTANCE: "inclusion-spec"
        SLOT-RESTRICTION: (type "is-a" "address")  }}
```

---

## 5.1.4. Exclusion-spec

The exclusion-spec is analogous to the inclusion-spec but specifies what slots and/or values *cannot* be inherited. If the inclusion-spec from the revious example, pet-of-incl-spec, had been stored in the EXCLUSION slot of **pet-of** instead of the INCLUSION slot, then exactly those things that were passed by the relation before would explicitly not be passed by the new relation. Any slots or values that pass their corresponding restrictions may not be inherited across the exclusion-spec. The value restriction only applies if the slot restriciton fails.

Exclusion-specs are evaluated before inclusion-specs. If an exclusion-spec denies inheritance of some information, but an inclusion-spec allows inheritance of the same information, the information cannot be inherited. Exclusion specs are convenient for blocking inheritance of a slot or value, when a relation has an elaborate structure of inheritance specs.

Building exclusion-specs follows directly from inclusion-specs.

```
{{ exclusion-spec
     IS-A: "inheritance-spec"
     TYPE:
          range: (or slot value)
     SLOT-RESTRICTION: (not t)
          range: "must follow <schema-op> grammar"
     VALUE-RESTRICTION: (not t)
          range: "must follow <value-op> grammar"  }}
```

**Schema 5-5:** The **exclusion-spec** schema

The following is an example of an exclusion that could be used to restrict the **house** schema's inheritance of a particular value. The house schema is related to the **building** schema by an "is-a" relation. However, this particular **house** belongs to an architect, and is round. Therefore it is necessary to restrict the relation between **house** and **building** so information about a building's walls is not inherited by the house schema.

```
{{ building
   HAS-WINDOWS:
   HAS-DOORS:
   HAS-FLOORS:
   HAS-WALLS: "four"  }}

{{ house
   IS-A: "building:
   INSTANCE: "is-a"
       exclusion: "walls-ex-spec"  }}
```

To create the **walls-ex-spec**, type:

```
1.(mk-schema "walls-ex-spec" ("is-a" "inheritance-spec")
   ("type" "value") ("value" "four"))
"walls-ex-spec"

{{ walls-ex-spec
   IS-A: "inheritance-spec"
   TYPE: "value"
   VALUE: "four"        }}
```

Schema 5-6: The **walls-ex-spec**

## 5.1.5. Elaboration-spec

Elaboration-specs are used to refine the definition of a slot. The elaboration-spec provides slots defining both the slot is to be elaborated, and the slots resulting from the elaboration. Like other inheritance-specs, the elaboration-spec contain DOMAIN and RANGE slots, specify and the *condition* when the spec applies. The **elaboration-spec** schema is given below.

```
{{ elaboration-spec
     IS-A: "inheritance-spec"
     SLOT-DEPENDENCY: "elaborates"
     SLOT-RESTRICTION:
          range: (schema (type "is-a" "slot"))
     NEW-SLOTS:
          range: (all (schema (type "is-a" "slot")))  }}
```

**Schema 5-7:** The **elaboration-spec** schema

The fields of the elaboration-spec that differ from or, did not appear in previous inheritance-specs, are described below.

SLOT-RESTRICTION: The slot restriction is the slot in the range schema to be elaborated into new-slots in the domain schema. This only allows one slot.

NEW-SLOTS: Any slot matching the SLOT-RESTRICTION is elaborated into the new slots specified by the NEW-SLOT restriction. The value of the NEW-SLOTS slot must be a list of slot names.

Each slot specified in NEW-SLOTS will be created (i.e. slotc) inherited using this spec, providing the slot specified in SLOT-RESTRICTION either exists in the range schema, or may be inherited by it. Values may not be inherited with elaboration specs. The SLOT-DEPENDENCY relation will specify the relation used to link the meta-schemata of the new slots to the meta-schema slot being elaborated. An example:

```
{{ mammal
     BODY: }}

{{ dog
     IS-A "mammal"
          Elaboration: "dog-elab"  }}

{{ dog-elab
     INSTANCE "Elaboration-spec"
     SLOT-RESTRICTION: "body"
     NEW-SLOTS: "left-front-leg" "right-front-leg"
                "left-back-leg" "right-back-leg"
                "torso" "neck" "head"  }}
```

**Schema 5-8:** The **dog-elab-spec**

The slots LEFT-FRONT-LEG, RIGHT-FRONT-LEG, LEFT-BACK-LEG, RIGHT-BACK-LEG, TORSO, NECK, and HEAD elaborate the BODY slot in **mammal**. All of the new-slots are created in the domain schema (i.e. dog) at the time of the first access of a new-slot. The elaborated schema has the following form:

```
{{ dog
    IS-A: "mammal"
    LEFT-FRONT-LEG:
    RIGHT-FRONT-LEG:
    LEFT-BACK-LEG:
    RIGHT-BACK-LEG:
    TORSO:
    HEAD:
    NECK:      }}
```

**Schema 5-9:** The **dog** schema with elaborated slots

If the switch **slot-dependencies** is non-nil, SRL creates an elaborates relation linking each of the new slots to the slot being elaborated (in this case, BODY). The elaborates relation is attached to the meta-slot of each of the slots involved in elaboration. The **elaborates** relation is used, because the above spec does not specify a SLOT-DEPENDENCY, and **elaborates** is the default relation. The following is an example of the meta-slot for the LEFT-FRONT-LEG slot:

```
{{ dog.left-front-leg
    INSTANCE: relation
    DOMAIN: "dog"
    SLOT: "left-front-leg"
    ELABORATES: "mammal.body"   }}
```

**Schema 5-10:** The meta-slot of the LEFT-FRONT-LEG slot

This example shows how the LEFT-FRONT-LEG slot of the **dog** is related to the BODY of the **mammal** by an **elaborates** link.

SRL provides functions for returning the elaborations of slots in specified schema.

(**slot-elab** <schema> <slot> [ <context> [ <path> ] ])
    RETURNS: the slots in <schema> that are elaborations of <slot>.

(**slot-unlab** <schema> <slot> [ <context> [ <path> ] ])
    RETURNS: the slot that <slot> is an elaboration of.

## 5.1.6. Map-spec

The Map primitive allows the representation of one to one mappings between slots, and transformations on their values. For instance, a mammal's front legs could be mapped onto man's arms. Like all other specs, the **map-spec** uses the DOMAIN, RANGE, and CONDITION slots. The map primitive uses the relation **mapped-from** for slot and value dependencies.

```
{{ map-spec
    IS-A: "inheritance-spec"
    SLOT-DEPENDENCY: "mapped-from"
    VALUE-DEPENDENCY: "mapped-from"
    DOMAIN-MAP:
        range: "must follow <schema-op> grammar"
        cardinality: (1 1)
    RANGE-MAP:
        range: (schema (type "is-a" "slot"))
        cardinality: (0 1)
    MAP-FUNCTION: }}
```

Schema 5-11: The **map-spec** schema

The **map-spec** has two independent functions. The first is mapping one slot name onto another. The second is mapping one value onto another. One slot is mapped onto another only when one slot is specified in the DOMAIN-MAP, and a different slot is specied in the RANGE-MAP. If more than one slot is specified in the DOMAIN-MAP, or no slot is specified in the range-map, then the RANGE-MAP is ignored, and the same slot is used for the range.

If inheritance is to be performed, and the DOMAIN-MAP matches the slot being searched for, then the range schema is searched for a value in the RANGE-MAP. If a value is found, it is altered by applying the MAP-FUNCTION to it. A map function specified by the user must accept the following parameters:

```
    (<map-function> <domain> <relation> <range> <uvalue> <context>)
```

The domain argument in the example above is the schema in the domain of the relation. The relation argument is the relation along which inheritance is proceeding. The range argument is the schema in the range of the relation, and uvalue is the value to be inherited, in unit form. The function returns the new value either in unit or value form. If no map-function is specified, then the identity function is assumed.

An example of the map's use is the creation of a relation called **toy**. The **toy** relation transforms the dimensions of the range-map when inherited by the domain of the relation. Because the DOMAIN-MAP may match more than one slot, the RANGE-MAP is not specified. SRL assumes that the name of the slot in the range will be the same as the slot in the domain, and only the value is transformed.

```
{{ toy-dog
     TOY: "dog" }}


{{ toy
     IS-A: "relation"
     MAP: "toy-map-spec"  }}

{{ "toy-map-spec"
     INSTANCE: "map-spec"
     DOMAIN-MAP: (type "is-a" "dimension")
     MAP-FUNCTION: scale  }}

(defun scale (domain relation urange uvalue context)
   (times .1 (unitg uvalue)))
```

**Schema 5-12:** The **toy-map-spec**

## 5.1.7. Introduction-spec

Sometimes, it is desirable to add new slots to a schema when a relation is instantiated. The **introduction-spec** allows new slots to be added to the schema when a relation is attached. For example, if fido is to guard a house, (**fido** GUARDS **house**) implies that fido has some new attributes that are not defined in the house schema, such as "control-commands." The **introduction-spec** enables the new attributes to be attached to **fido.**

```
{{ introduction-spec
    IS-A: "inheritance-spec"
    NEW-SLOT:
        range: (type "is-a" "slot")
        cardinality: (1 1)
    META-SLOT:
        range: (set (type "INSTANCE" "introduction-spec"))
        cardinality: (0 inf)
    NEW-VALUE:
    META-VALUE:
        range: (set (type "INSTANCE" "introduction-spec"))
        cardinality: (0 inf)      }}
```

**Schema 5-13:** The **introduction-spec** schema

An **introduction-spec** specifies a slot to be created in the NEW-SLOT slot. A list of introduction specs may be specified for the meta-slot of the slot, which was created. This list is the value of META-SLOT. Optionally, the spec may specify values stored in the NEW-VALUE slot, to fill the slot created by the intro-spec. Introduction specs are specified for each value in the corresponding value of the META-VALUE slot of the spec. The META-SLOT and META-VALUE defines what can be introduced for the slot and value respectively. They contain **introduction-specs.**

Introduction specs can be expensive to use because checking for intro-specs must be done whenever a slot is created. Intro-specs are controlled by the **$introduction** switch. If the switch is set to "t," the intro-specs are checked when a slot is created. Intro-specs are not checked when the switch is set to **nil**. The default setting for **$introduction** is **nil.**

In the example below, the **guards** relation is defined to introduce the CONTROL-COMMANDS slot, with the four values specified, into any schema given a GUARDS slot. When the GUARDS slot is created in **fido**, the slot CONTROL-COMMANDS is also created. The values, "kill," "maim," "stop," and "stay" are added to the CONTROL-COMMANDS slot. The spec for each value's meta-value is then evaluated.Introduction specs are evaluated when a slot is created. Notice the DOMAIN and RANGE restrictions have been used. The GUARDS slot may only be placed in schemata that are of (type "is-a" "mammal") (which in this example **Fido** IS-A), and may only be filled with a "person" or "place" (the **Jones house** IS-A "place").

```
{{ guards
    IS-A: "relation"
    DOMAIN: (type "is-a" "mammal")
    RANGE: (schema (or (type "is-a" "person")
                (type "is-a" "place")))
    INVERSE: "guarded-by"
    INTRODUCTION: "guard-intro"

{{ guard-intro
    INSTANCE: introduction-spec
    NEW-SLOT: "control-commands"
    NEW-VALUE: "kill" "maim" "stop" "stay"
    META-VALUE: ("kill-intro") ("maim-intro") ("stop-intro") ("stay-intro") }}

{{ kill-intro
    IS-A: "introduction-spec"
    NEW-SLOT: "instance"
    NEW-VALUE: "kill-command" }}

{{ fido
    IS-A: "dog"
    GUARDS: "Jones house"
    CONTROL-COMMANDS:
            "kill"
             instance: "value" "kill-command"
            "maim"
             instance: "value" "maim-command"
            "stop"
             instance: "value" "stop-command"
            "stay"
             instance: "value" "stay-command" }}

{{ Jones house
    IS-A: "place" }}
```

## 5.2. Creating and Using Relations

### 5.2.1. Creating Relations

The following is a recipe for building a new relation:

1. Create a schema with the name the relation is to have (e.g. create a **pet-of** schema).

2. Link the new relation to the relation hierarchy. The simplest way to link the new relation is to create an IS-A slot using slotc, and fill it with the value "relation". It is important that

the new relation be linked to the **relation** schema, directly or indirectly. The actual relations used, are not important (with the qualifications given in section 5.1). The new relation must be able to inherit all the **relation** schema's slots.

3. Fill the DOMAIN, RANGE, TYPE, CONDITION and INVERSE slots as desired. SRL will attempt to inherit all values not directly specified in the new relation.

4. Build any inheritance specs needed to specify the relation desired. Existing inheritance specs may be re-used even if they are already used in other relations.

5. Fill inheritance-spec slots with the names of the spec instantiations.

6. [optional] Create a transitivity grammar for the relation (see section 6.1) and store it in the PATH slot of the relational schema.

7. Declare the relation as compiled or interpreted. This is discussed below.

SRL2 determines whether a slot and/or its value can be inherited along a relation by interpreting the inheritance specs defined in the relation. Inheritance specs are examined in the following order:

1. exclusion If an exclusion spec matches the slot under consideration, the inheritance cannot take place along the relation.
2. elaboration
3. map
4. inclusion

If a relation has an introduction spec, it is evaluated when the relation is created in the schema, resulting in the introduction of the information in the designated schema at that time.

The following relations are used to declare and test relations:

(**relation** <relation-list> [ <context> ] )
    RETURNS: t
    SIDE-EFFECT: Declares <relation> to be an interpreted relation.

(**relation-p** <relation> [ <context> ] )
    RETURNS: t if <relations> has been declared to be a relation

(**relationp** [ <context> ] )
    RETURNS: A list of all the relations that have been declared to the system.

## 5.2.2. The Relation Compiler

Inheritance is faster when relations are compiled instead of interpreted. The most efficient compiled relations only employ inclusion specs with simple DOMAIN and SLOT restrictions. The powerful *type* facility is expensive, and slows down inheritance. Using a range restriction means that each value in the relation must be checked. If all ranges are acceptable, only the relation's slot and domain must be checked. Restricting the values will not slow inheritance significantly.

Inheritance performed by interpreting a relation is very slow. Yet one advantage to using interpreted relations is that their definition is dynamic, whereas the definition of a compiled relation is fixed at the time of compilation. The interpreted relation's definition will change accordingly when alterations are made to other relations in the relational hierarchy. Changes made to other relations are inherited by the interpreted relation each time it is used.

(**relationc** <relation-list> [ <context> ] )
  RETURNS: t
  SIDE-EFFECT: Declares <relation> to be a relation and compiles it.

(**relationd** <relation> <relation-list>) )
  RETURNS: t
  SIDE-EFFECT: Deletes the relation from the relation list.

## 5.2.3. Using Relations

Once a relation has been defined (by SRL or by the user), it must still be put into a schema to be of any use. Placing a relation in a schema involves two steps. In order to declare <schema1> to be related to <schema2> by the <relation> relation:

1. Create a <relation> slot in <schema1> (a slotc command)

2. Fill the <relation> slot of <schema1> with the value <schema2> (a valuec1 command).

Inheritance may now take place along this relation if inheritance properties for <relation> have been defined by the user. Note, however, that adding and filling an instance of the relation are subject to the DOMAIN and RANGE restrictions of the relation (see section 4.3). Auto-linking (section 4.4) may take place when the slot is filled.

Any time a slot is created that represents a declared relation, SRL notes that the slot is a relation. A relaiton cannot be removed, just as a slot's designation cannot be changed. If an attribute slot is created, it remains an attribute slot even if the attribute slot is later declared to be a relation.

## 5.3. Caching and Dependency Links

Slots and values are inherited by SRL when they are needed. For example, **fido** needs four legs to run, so he inherits NUMBER-OF-LEGS slot and its values from **dog** schema. If the slot is accessed, (i.e.valueg1("fido" "number-of-legs" "four")), then four is inherited down from dog. Slots and values are not stored in the schema where they are inherited to, unless they have been modified. If there is any attempt to alter the contents of a slot within a schema, the slot is copied down into that schema. If the values are to be used in the new slot then, they are copied down. A **valuec** only copies the slot since the values have been supplied by the user. A **valued** command copies down the new values to delete the appropriate value(s).

```
{{ fido
     INSTANCE: "dog"    }}

{{ dog
     IS-A: "mammal"
     INSTANCE+INV: "fido"
     HAS-COLOR:
     NUMBER-OF-LEGS: "4" }}
```

**Schema 5-14:** The inherited value "4" is copied into the **dog** schema

The user can force slots and values to be cached whenever they are inherited. Caching values is governed by the $cache switch. If **$cache** is set to t, whenever a value or slot is inherited into a schema, a copy is stored in the schema. For example, if **$cache** is set to t when inheriting the NUMBER-OF-LEGS slot and its value, then a copy of the slot and value would be stored in the **Fido** schema. If **$cache** is set to nil, then the value "4" would be inherited and returned to the user, but neither the NUMBER-OF-LEGS slot or the value would be copied into the **fido** schema. It is recommended to operate with the **$cache** switch on if the value will be retrieved frequently.

Often, it is important to know where a slot or value was inherited from. A dependency mechanism puts links in place to represent data dependencies resulting from inheritance. Simply put, the links indicate where a slot or value was inherited from. The **$slot-dependency** and **$value-dependency** switches control whether dependency links are put in place for slots and values respectively. The links are put into place when the value of the switch is non-nil.

Dependency links for inherited slot stake the form of relations. The relations link the the meta-slots of the slot inherited from and the newly inherited slot. The relation used as a dependency link is determined by the inheritance spec used during inheritance. The value of SLOT-DEPENDENCY in the appropriate inheritance spec becomes the relation used to link the two meta-slots.

```
{{ fido
      IS-A: "dog"
      COLOR: "black"
          included-from: "dog-color-meta"
          schema: "fido"
          slot: "color"  }}

{{ dog
      IS-A + INV: "fido"
      COLOR:
          included-to:"fido-color-meta"
          schema: "dog"
          slot: "color"  }}
```

In the example above, the IS-A relation is used to the link the meta-slots of **fido** and **dog**. The IS-A relation is used because he SLOT-DEPENDENCY slot of the inheritance-spec was filled with the value IS-A. The meta-slots of **fido** and **dog** contain *included-to* or *included-from* facets, a *schema* facet, and a *slot* facet. The meta-slots provide information about the schema inherited from (in this case, **dog**), or the schema inherited to (in this case, **fido**), and the slot being inherited (in this case, COLOR).

Dependencies between inherited values is noted in the same way, except they are denoted by relations linking the meta-values of the values involved in inheritance. The relation used is found in the VALUE-DEPENDENCY slot of the spec.

There is an interaction between the **$meta-slot** and **$slot-dependency** switches, and between the **$meta-value** and **$value-dependency** switches. If $meta-slot and $meta-value hold the name of a schema, then meta-slots and meta-values are automatically created. Meta-slots are automatically linked to the schema representing the slot (which the meta-slot has information about) by an instance relation. For example, the meta-slot on the HAS-COLOR slot would be made an instance of **has-color** schema. Meta-values are automatically linked to the value of the **$meta-value** switch by an INSTANCE relation. However, if a dependency was noted for an inherited slot or value, the INSTANCE relation is not put in. Turning on dependencies when the appropriate meta switch is not on, is an error.

## 5.3.1. Local Specialization of Relations

Relations may be specialized with each instance of their use. A relation's inheritance semantics can be altered by specifying the desired change in the relation's inheritance specification facets. For example, it may be said a *platypus is-a mammal*, however, the platypus birth process is not mammalian. Thus, it is necessary to restrict the relation so information about a mammal's birth-process is not inherited by the platypus schema. First, an exclusion-spec denying inheritance of the birth-process values, but not the slot slot itself (the platypus still has a birth process, even if not mammalian), must be constructed. The relation is specialized by filling the exclusion facet of the IS-A slot in **platypus** with the birth-exclusion-spec.

The following schemata define the BIRTH-PROCESS slot's exclusion from inheritance along the IS-A relation linking **platypus** to **mammal**. The exclusion specification **birth-ex-spec** is placed in the EXCLUSION facet of the IS-A slot in **platypus.**

```
{{ platypus
    IS-A: "mammal"
    INSTANCE: "is-a"
        exclusion: "birth-ex-spec"  }}

{{ birth-ex-spec
    INSTANCE: "exclusion-spec"
    TYPE: slot
    SLOT: "birth-process"    }}
```

A recipe for specializing relational schemata is given below:

1. Create any inheritance-specs needed to modify the relation (see section on user defined relations).

2. Get the meta-slot attached to the relation in question. For the platypus example, this would be the meta-slot attached to the IS-A slot. If a meta-slot does not exist, then create one.

3. Create slots in the meta-slot for each inheritance-spec. Slot names correspond to inheritance-spec type as with user defined relations. It is usually the case that the meta-slot is related to the **relation** schema, hence inherits the standard inheritance specification slots.

4. Fill the slots with the names of the new inheritance-specs.

To undo a specialization of a relation, repeat the same process but delete the slot and value instead.

Local specialization of a relation is controlled by the **$local** switch. If it is set to t, then local specializations of relations will be interpreted (default is set to nil), otherwise they will be ignored.

## Function summary: relation commands

**mk-schema:** is a utility function that enables an entire schema to be specified in one command. However, meta-information associated with the schema cannot be specified with the command. mk-schema should be used to build relations.

(**relationc** <relation-list> [ <context> ] )
RETURNS: t
SIDE-EFFECT: Declares <relation> to be a relation and compiles it.

(**relationd** <relation> <relation-list>) )
RETURNS: t
SIDE-EFFECT: Deletes the relation from the relation list.

(**relation** <relation-list> [ <context> ] )
RETURNS: t
SIDE-EFFECT: Declares <relation> to be an interpreted relation.

(**relation-p** <relation> [ <context> ] )
RETURNS: t if <relations> has been declared to be a relation

(**relationp** [ <context> ] )
RETURNS: A list of all the relations that have been declared to the system.

## Relevant switches:

**$cache:** if the switch is set to t, then a copy of an inherited slot or value is stored in the schema that inherits the slot or value.

**$value-dependency:** if set to t, then dependency links for values are put in place.

**$slot-dependency:** if set to t, dependency links are put in place for slots.

**$introduction:** if the switch is set to t, introduction-specs are checked when a schema is created.

**$local:** if set to t, then local specializations of relations are interpreted.

# 6. Paths and Inheritance

This chapter shows how paths can be used to control the inheritance search, define a relation's transitivity, and form complex schema types.

## 6.1. Paths

The construction and interpretation of relations was discussed in the previous section previous section. However, the discussion only addressed individual relations. Most models contain many relations which form complex, tangled hierarchies. It is important, particularly for inheritance, to understand and control how relations interact. Paths are one tool for controlling search through a network of relations. Paths can be used to both guide inheritance and define the transitivity properties of relations. The transitivity property of a relation refers to how the relation connects two schemata, but does not necessarily indicate whether the schemata can inherit information from eachother.

A path is like a map that defines each step (or alternative to a step) of an inheritance, thus guiding the search. Paths are composed of specific slots, meta-slots, values, and meta-values. Paths can also be composed of specific slot/value patterns, and boolean combinations of these patterns.

The grammar for specifying paths in SRL is presented below. How each construct is interpreted is also discussed. The applications of paths is addressed subsequent sections.

```
<path> ::=
        (or <path>+) |
        (list <path>+ ) |
        (repeat <path> <min> <max>) |
        (step <slot-op> <value-op>) |
        (path <relation>) |
        t | nil

<min> ::= <max> ::= [0, inf]
```

**Figure 6-1:** The path grammar

A path is specified as a regular expression. A path is interpreted as a "treasure map," defining each step (and alternatives) a search may take. The constructs are discussed below.

- **or** - Any of the sub-paths of the or may be taken.
- **list** - Each of the sub-paths of the list must be followed in order. The list is only satisfied when all of the sub-paths are satisfied in the order given.
- **repeat** - <path> must be satisfied at least <min> times, and no more than <max> times. This must happen consecutively.

- **step** - The <slot-op> is performed on the relation being traversed, and the <value-op> is performed on the range of the relation. If both succeed then the step may be made. The range of the relation becomes the domain of any subsequent steps that are taken. <slot-op> and <value-op> must be valid restrictions (see section 4.3.1).

- **path** - The value of the TRANSITIVITY slot of the <relation> is substituted here (see section 6.3). must be fulfilled for the path to be fulfilled. Grammars that are left recursive are always a problem when using paths. Right recursive grammers are sometimes a problem.

- **t** - Specifies that all paths are acceptable.

- **nil** - Specifies that no paths are acceptable. nil indicates that the current schema is an endpoint of the path being searched. For instance,
  ```
  (or nil <path>)
  ```
  means the search either stops immediately, or may explore <path>. nil may also be used to signify that no inheritance should be performed.

An example of a path restricting the search to only IS-A relations is:

```
(repeat (step "is-a" t) 0 inf)
```

If only alternating IS-A and PART-OFs are desired:

```
(repeat (list (step "is-a" t) (step "part-of" t)) 0 inf)
```

If the path is restricted to IS-AS and the range of the IS-A must be a mammal:

```
(repeat (step "is-a" (schema (type "is-a" "mammal"))) 0 inf)
```

To search only along relations that have a "search" facet filled with the value "primary", the path would be:

```
(repeat (step (meta (type "search" "primary")) t) 0 inf)
```

## 6.2. Path Selection

Multiple roles introduces problems when determining which slot values to inherit if the source of the slots is ambiguous. Consider the example of "fido":

---

{{ **fido**
  IS-A: "guard" "pet" }}

{{ **guard**
  TEMPERAMENT: "hostile" }}

{{ **pet**
  TEMPERAMENT: "docile" }}

---

When requesting the value of **fido**'s TEMPERAMENT slot, what value should be returned? example { (valueg "fido" "temperament") = = >"hostile" ? "docile" ? } Hence, a slot's value can be inherited from more than one place. SRL allows the user to inherit values selectively. By specifying a path expression in the access, only the specified relations will be searched. For example, to restrict inheritance along only the "is-a" relation to "pet", the access would have the following form:

```
        (valueg "fido" "temperament" nil '(step "is-a" "pet"))

    ==> (docile)
```

Note only a single step along the "is-a" relation to "pet" will be made as defined by the path expression. If "pet" did not contain a value for the TEMPERAMENT slot, relations from "pet" to other schemata would not be searched for a value. The following path expression could be used to search any schema related to pet:

example { (valueg "fido" "temperament" nil '(list (step "is-a" "pet") t)) } 3: Unterminated literal 4: Unbalanced parens

Placing at the end of the list permits the access to search along any relation leading out of "pet".

## 6.3. Schema Typing and Relational Transitivity

Paths and other constructs in SRL talk about *types* of schemata. Schemata are typed in the following way: A schema **X** is said to be of *type Y Z* if the schema **X** contains a slot Y that has *Z* as its value. With regard to the dog schema, **dog** can be called type IS-A **mammal** since the **dog** schema contains an IS-A slot that has the value **mammal**. Schema typing, as described here, is referred to as *atomic typing*.

Atomic typing can be used to specify more complex types by combining two or more atomic types. For example, **fido** cannot be said to be type IS-A **mammal** because the schema does not have an IS-A slot filled with the value **mammal**. But, we can note fido is of type INSTANCE **dog**, and that **dog** is of type IS-A **mammal**. Thus two atomic types can be combined to define a more complex type, in this instance, type IS-A **mammal**.

Path expressions are used to specify how types combine within the knowledge hierarchy. For example, type **is-a** can be defined as:

```
              (list (repeat (step instance t) 0 1)
                    (repeat (step is-a t) 0 inf))
```

This path specification states that a schema is of type IS-A <schema> if it is linked by an optional

INSTANCE relation and an arbitrary number of IS-A relations to <schema>. Thus, a number of atomic types have been combined to define a more complex type. Using the type definition created by the path specification, **fido** is of type IS-A **mammal**, since **fido** is linked to **dog** by an INSTANCE relation, and **dog** in turn is linked to **mammal** by an IS-A relation.

```
{{ dog
     IS-A: "mammal"
     HAS-COLOR: "brown"  }}

{{ fido
     INSTANCE: "dog"    }}
```

For path such expressions to be recognized for schema typing, the expression must be stored in the TRANSITIVITY slot of the relation in question. In this example, it is stored in the TRANSITIVITY slot of the **is-a** relation. The TRANSITIVITY slot stores a path describing how two schemata must be related to satisfy transitivity.

The following functions use path expressions to test a relationship, and generate all schemata entering into the relation.

(**r-test** <schema1> <relation> <schema2>)
RETURNS: t if <schema1> satisfies the transitivity grammar for the relation.
NOTE: Takes the path found in the TRANSITIVITY slot of the <relation> and, using that path expression, tries to find a path between the <schema1> and <schema2>.

(**r-find** <relation> <schema>)
RETURNS: a list of all schemata that are of type <relation> <schema>
NOTE: Takes the path found in the TRANSITIVITY slot of the <relation> and finds all schemata that are linked to <schema> by paths admissible by the TRANSITIVITY.

## 6.4. Slot Accessibility

A relation linking two or more schemata allows properties to be transferred (i.e., slots) among schemata. Understanding when a slot in a related is schema is accessible is important for the user. The following defines the accessibility of slots in schema B from schema A when a relation R links schema A to B.

- If the slot already exists in A (by previous inheritance, or introduction), then it is accessible.
- If the slot is the domain of a map spec, then the mapped version is accessible.
- If the slot is the range of a map spec, then it is not accessible.

- If the slot is excluded by an exclusion spec in R, then it is not accessible.
- If the slot is either in the domain (slot) or the range (new-slots) of an elaboration spec, then it is accessible.
- If the slot is specified by an inclusion-spec, then it is accessible.

The following functions provide the user with information about what slots exist and/or are accessible:

(**slot** <schema> [ <context> ])
RETURNS: the names of the slots directly defined in <schema>.
NOTE: It does not return the names of slots that are accessible by inheritance.

(**slote** <schema> <slot> [ <context> [ <path> ] ])
RETURNS: t if the slot exists in <schema> or is accessible by inheritance, otherwise nil.

(**slot-all** <schema> <path> [ <context> ])
RETURNS: a list of slots accessible along the path specification.

## 6.5. Inheritance Algorithm

SRL uses a breadth first search when inheriting slots. Values are inherited by checking to see if inherited slots contain values. When inheriting a slot, SRL first checks all the schemata that are separated from the original schema by only one relation. Then the schemata two relations away are searched and so on. At each level, relations are searched in the order they were created in the schema.[8] Inheritance proceeds until a slot is found. For each relation, the values are searched in order they appear in the slot. Value inheritance proceeds until a slot is found which has values. If **$inherit-all** is non-nil inheritance for values proceeds until all values have been found.

Only slots that were created as relations are used for inheritance. The user may further narrow the search by using the path argument to the function being called, which specifies acceptable paths for inheritance to follow. If the path is not provided a path of **t** is assumed, and all relations are searched.

SRL2 determines whether a slot and/or its value can be inherited along a relation by interpreting the inheritance specs defined in the relation. The order is given in section 5.2.1.

When the definition of a slot is inherited from schema A to B, the following is performed:

1. A slot is created in schema B.
2. If **$slot-dependency** is non-nil, a meta-slot is created for the new slot in schema B, and it is linked to the meta-slot of the inherited slot in schema A by the relation specified in the

[8] Reading the relations top to bottom as they appear in the pretty-printed schema gives the order in which they were created.

spec that allows the inheritance.

When a value is inherited from schema A to B, the following is performed:

1. A value unit is created and added to the slot in schema B.
2. If **$value-dependency** is non-nil, then a meta-value is created for the value in schema B, and is linked to the meta-value of the value in schema A by the relation specified in the spec.

## Function Summary: commands to test transitvity

**(r-test** <schema1> <relation> <schema2>)
RETURNS: t if <schema1> satisfies the transitivity grammar for the relation.
NOTE: Takes the path found in the TRANSITIVITY slot of the <relation> and, using that path expression, tries to find a path between the <schema1> and <schema2>.

**(r-find** <relation> <schema>)
RETURNS: a list of all schemata that are of type <relation> <schema>
NOTE: Takes the path found in the TRANSITIVITY slot of the <relation> and finds all schemata that are linked to <schema> by paths admissible by the TRANSITIVITY.

## Relevant switches:

**$inherit-all:** if set to t, then inheritance continues unitl all possible values to inherit have been found.

# 7. Contexts

This chapter illustrates the context facility and the way it extends user capabilities by enabling version management of different scenarios and alternate worlds reasoning. Contexts are also structured to save time and space when dealing with the database.

## 7.1. Using contexts

SRL2 provides the user with a context mechanism. A context serves as a virtual copy of a collection of schemata. Contexts allow the user to extrapolate from a model without destroying the original model. This can be useful for conducting simulations with the same starting point. Schemata can be created, altered, and deleted without changing the original model. While making an actual copy of all the schemata would achieve the same effect as contexts, the context facility only copies those schemata used within the new context, which saves time and space. When dealing with very large databases, saving space and time can be important.

When SRL2 is initialized, the user is automatically put into the context **"$root-context"**. Any new context created by the user will be a subcontext of the **"$root-context"**. Contexts are arranged in trees, so that every context (other than the *"$root-context"*) has a single parent context, and may have an indefinite number of children contexts. Any context may assume/access/inherit information found in any of its ancestor contexts, but may not assume information found in descendant contexts. When accessing information in SRL, first the specified context is checked for the information. If the required information is not found, then the parent of this context is checked for the information. The process continues until the information is found, or the root of the context tree is reached. When a schema containing the desired information is found within a parent context, the schema is copied down to the child context where the search for the information began.

Creating or accessing a schema always occurs within a context. A context may be specified in which to look for the schema. If no context is specified, the search begins in the context name held in variable **$context**. A schema name must be unique within a context. However, schemata with the same name may exist within different contexts, and contain conflicting information. For example, a user might have two contexts, "Pluto" and "Mars", each containing a **horse** schema. The blood-color slot of the dog schema might have a value of "red" in the context of "Pluto", but have a value of "green" in the context of "Mars".

Contexts can be used in two or more ways. The context facility can be used to support version management of models. Each time a revision of the SRL model takes place, a new context can be created. Changes can be made in the new context and merged with the original context. The context facility can also be used for alternative worlds reasoning. A context can be created for each world being simulated. The results of each simulation scenario, or possible world, can be contrasted with other simulations.

The following example depicts **dog** as two different schemata. The first example shows the schema as it appears in the "$root-context". The second shows the schema as it appears in another context, "Mars", which is a child of the "$root-context".

---

```
; dog schema as it appears in the "$root-context"
 {{ dog
      BLOOD-COLOR: "red"
      BIRTH-PROCESS: "live"  }}

; dog schema as it appears in the context "Mars"
 {{ dog
      BLOOD-COLOR: "green"
      BIRTH-PROCESS: "hatched"   }}
```

---

Assume a third context, "Earth," also exists, and is a child of the "$root-context" (thus a sibling of "Mars"). Accessing the value of the BIRTH-PROCESS slot in the **dog** schema in the context "Mars" would return "hatched," while accessing the same value in the "Earth" context will return "live." Accessing the BIRTH-PROCESS value in the "Mars" context was completed when the value "hatched" was found and returned. The same value was accessed in the "Earth" context, but the **dog** schema does not exist in the "Earth" context. So the parent context of "Earth" **"$root-context"**) was accessed, and value "live" obtained and returned. Here is the dog schema as it appears in the "earth" context:

---

```
 {{ dog
      BLOOD-COLOR: "red"  }}
```

---

A default mechanism for contexts has been incorporated into SRL so the context need not be specified every time an access is made. The name of the current default context is stored in the variable **$context**. Choosing a new default context is called *asserting* a context.

## Function Summary: context commands

The following comands manipulate contexts. Contexts can be created, deleted, and tested for a specific relationship to another context. For instance, the **context-parent** command returns the parent context of a child context.

**(contextc** <child-context> **[** <parent-context> **])**
RETURNS: <child-context>
SIDE-EFFECT: Creates a new context <child-context> as a child of context <parent-context>. If <parent-context> is not specified, the context held in **$context** becomes the parent.
NOTE: <parent-context> must already exist, and <child-context> does not exist.

**(contextd [** <context> **])**
RETURNS: <context>

SIDE-EFFECT: Removes context <context> and its descendants from the context tree. If a context is not specified the context held in **$context** is removed.

NOTE: When attempting to remove a context that has children, an error will be generated requesting the user to delete the children contexts first.

(**contexta** <context>)

RETURNS: <context>

SIDE-EFFECT: Sets the system default context to <context>.

(**contextm** { <child-context> ])

RETURNS: <child-context>

SIDE-EFFECT: Merges the <child-context> with its parent, overwriting old (parent context) information. If child-context is the current value of **$context**, then **$context** is set to the parent.

(**context-parent** [ <context> ])

RETURNS: the name of the parent context of <context> in the context tree.

(**context-children** [ <context> ])

RETURNS: the names of all contexts that are children of <context> from the context tree.

(**in-context-p** <schema> [ <context> ])

RETURNS: t if <schema> exists in context <context>.

NOTE: this function differs from schema-p in that only <context> is examined for <schema>, and not the ancestors of <context>.

(**context-p** <context>)

RETURNS: t if <context> is a context. Otherwise nil is returned.

## Relevant switches:

**"$root-context"**: This is the context the user is put into when the system is first initialized.

# 8. Altering The SRL Environment

This chapter enumerates the system switches, which enable the user to modify the SRL environment.

## 8.1. Switches: allow the user to modify the SRL environment

Switches, which have been mentioned throughout the manual, are variables that can be set to have different values. Changing the switch settings allows the user to alter the SRL environment. Switch settings are changed by giving the switch a different value. Some switches take schema names as values. For example, the switch **$meta-slot** holds the name of a schema. The switch is on when it holds the name of a schema. All meta-slots generated when the switch is on are linked to the schema whose name is held by the switch. When the switch is empty (in other words, it does not hold a schema name), it is off, and no meta-slots are generated.

Other switches take different values. For example, **$inverse** can have the values all, none, or relations. If the switch has the value all, then inverse links are put inplace for all slots. The switch is off when it is set to none. The user can set switches according to his needs, or maintain the default settings provided by SRL.

Here is the mechanism that controls SRL switches, and allows the user to set and retrieve their values.

(**srl-get** <srl-switch>)
 RETURNS: The value current of the srl-switch.

(**srl-set** <srl-switch> <switch-setting>)
 RETURNS: <switch-setting>
 SIDE-EFFECT: The value of srl-switch is set to switch-setting.

These two functions provide the means to set and get (retrieve) SRL switches. To provide further ease for the user, **srl-let** environment allows the user to lambda bind the values of their choice any number of switches. These settings are removed when the user exits srl-let environment. srl-let is a macro.

The basic database for SRL provides another means of altering switch settings. SRL's basic database contains a schema called **SRL**. Each slot in **SRL** corresponds to a system variable. The value of the slot contains SRL's default settings for switches. The user may alter the performance of SRL in one of two ways. He may either reset the values in **SRL**, and re-initialize SRL using the SRLinit function described below, or he can set the variable's value directly using a LISP setq function.

There are two functions for initializing and viewing the current settings of system variables.

(**SRLpp** )
 NOTE: prints out the current settings of all system variables defined in the rest of this chapter.

(**SRLinit** [ <schema> ])
 NOTE: initializes all system variables to those specified in <schema>. If no schema is specified, then the **SRL** schema is used. If a schema is specified, it should be an instance of the

SRL schema.

**"$root-context"** This is the name of the root context which all context trees stem from. This context is created automatically with an SRL environment, and is used as the default context when the system is initialized.

**$access** The name of the current srl function call being made is maintained in this variable for error handling.

**$current-error** contains the name of the latest instance of the **SRL-error** schema that was created.

**$db-cache-unlimited**: If this is non-nil, then schemata cannot be swapped-out.

**$db-cache-max**: The maximum number of schemata that can be resident in core memory.

**$db-cache-keep**: The number of schemata that can remain resident in core memory after excess schemata are swapped-out.

**$default-context** Holds the name of the current default context used by the system, and is initialized to "$root-context".

**$queuelength**: Holds the number of schemata currently resident in the cache.

**$cache** Specifies whether or not inherited information is to be cached at the point where it was inherited, or recomputed each time it is needed. A non-nil value will cause the information to be cached. The meta-values of the cached value and the original value are linked by an includes, elaborates, or maps relation depending on the particular inheritance specification used. The default value is nil.

**$demon** A value of *t* will enable the checking and evaluation of demons attached to slots during slot access. Default is *nil*.

**$inherit-all** If t, then all relations are searched for the existence of all values for a slot, and the results added to the slot. If nil, then search stops at the first value found. Default: nil.

**$inverse** When a slot <slot> in schema <schema> is filled with a value that is also a schema (<schema-2>), a slot <inverse-slot> is constructed in <schema-2> and is filled with the value <schema>. Thus a backward link has been created. Setting **$inverse** to **all** enables this facility for all slots. A setting of **relations** enables it for relations. With a setting of **none** no inverse linking is performed. The default value is **relations.**

**$local**: Inheritance relations may be specialized at each instance of their use. The specializing inheritance specsare interpreted if the switch is set to *t*. If the switch is set to *nil*, local specializations are ignored. The default value is *nil*.

**$meta-schema**: If non-nil, then a meta-schema is created for a schema at schemac time. The meta-schema is linked to the value of **$meta-schema** by an INSTANCE relation.

Default is **schema**.

**$meta-slot**: If non-nil, then a meta-slot is created for a slot at slotc time. The meta-slot is linked to the schema of the same name as the slot by an INSTANCE relation. If the schema with the name of the slot doesn't exist one is created, and linked to the value of **$meta-slot** by an IS-A relation. Default is **slot**.

**$meta-value**: If non-nil, then a meta-value is created for a value at value creation time. The meta-value is linked to the value of **$meta-value** by an INSTANCE relation. Default is **value**.

**$restrict** Specifies whether or not restrictions on slots should be applied to new values of value facets. A switch value of *t* will cause restriction to take place, and *nil* will disable it. The default value is *nil*.

**$slot-dependency**: If this variable is non-nil, then dependency information concerning the inheritance of slots is maintained.

**$value-dependency**: If this variable is non-nil then dependency information concerning the inheritance of values is maintained.

## Function summary: commands to set and retrieve switch values

(**srl-get** <srl-switch>)
RETURNS: The value current of the srl-switch.

(**srl-set** <srl-switch> <switch-setting>)
RETURNS: <switch-setting>
SIDE-EFFECT: The value of srl-switch is set to switch-setting.

(**SRLpp** )
NOTE: prints out the current settings of all system variables defined in the rest of this chapter.

(**SRLinit** [ <schema> ])
NOTE: initializes all system variables to those specified in <schema>. If no schema is specified, then the SRL schema is used. If a schema is specified, it should be an instance of the SRL schema.

# 9. Error Handling

This chapter provides information about the system's error-handling facility and its response to errors.

## 9.1. The error-handling system

SRL2 offers a simple error handling system with capabilities similar to ON conditions in PL/1. The system for handling errors contains an **SRL-error** schema with slots enumerating the types of errors SRL detects and signals.

```
{{ SRL-error
    ERROR: "name of error"
    SCHEMA: "schema for which error took place"
    SLOT: "slot for which error took place"
    VALUE: "value for which error took place"
    CONTEXT: "the context in which the error took place"
    AUXILIARY: "holds extraneous information relevant to the error"
    MESSAGE: "error message generated by system"
    ACCESS: "SRL access being performed at time of error"
    ACCESS-ARGUMENTS: "The arguments SRL was called with."

    NO-SCHEMA:
        range: (type "instance" "error-spec")
    ILLEGAL-SCHEMA-NAME:
        range: (type "instance" "error-spec")
    NO-SLOT:
        range: (type "instance" "error-spec")
    NO-VALUE:
        range: (type "instance" "error-spec")
    DOMAIN-ERROR:
        range: (type "instance" "error-spec")
    RANGE:
        range: (type "instance" "error-spec")
    ILLEGAL-PATH:
        range: (type "instance" "error-spec")
    BAD-ELAB-REL-SPEC:
        range: (type "instance" "error-spec")
    NO-MAP-DOMAIN:
        range: (type "instance" "error-spec")
    ILLEGAL-PREDICATE:
        range: (type "instance" "error-spec")
    INVALID-CONTEXT:
        range: (type "instance" "error-spec")
    INVALID-DATABASE:
        range: (type "instance" "error-spec")    }}
```

**Schema 9-1:** The **SRL-error** schema

When an error occurs, an instance of the **SRL-error** schema is generated. The SCHEMA, SLOT, VALUE, and CONTEXT slots are filled with the name of the schema, slot, value, and context of the access causing the error. The name of the latest instance of the **SRL-error** schema is stored in the variable $current-error. This schema can be referenced at any time to determine the success or failure of an access. Next, the **SRL-error** schema is interrogated for a slot that corresponds to the error. The contents of the slot should be a single **error-spec** (figure 9-2), which defines the error system's response:

**error**: A system error is forced.

**top-level**: a throw is executed from the point of the error in SRL to the user level SRL access function that caused the error. The SRL access function returns the value specified in the VALUE slot of the **error-spec**.

SRL2 interprets an **error-spec** by first executing the contents of the ACTION slot and either forcing a lisp error if the type is error or evaluating and returning the value of the VALUE slot. It is assumed the all functions in the ACTION and VALUE slot take the single parameter: <current-system-error>.

```
{{ error-spec
     TYPE:
          range: (or error top-level value)
     SIGNAL:
          range: (or t nil)
     ACTION: <function>
          range: "a function"
     VALUE:
          range: "a function" }}
```

**Schema 9-2:** The **error-spec** schema

All system errors are generated by calling the following function:

(SRL-error <error> <access> <access-args> <schema> <slot> <value> <context> <auxiliary> <message>)

Each of the parameters is stored in the corresponding slot in the new instance of the **SRL-error** schema. <error> is the name of the error and should correspond to a slot in **SRL-error**. <access> is the type of access being performed, i.e., valueg, valuec, etc. <access-args> is the arguments that the access function was called with. <schema> is the schema being accessed (if applicable), <slot> the slot being accessed, and <value> the value being placed (if applicable). <context> is the context where the access occurred causing the error. <auxiliary> is a wild card parameter to pass any other information relevant to the error. <message> is the message to be printed to the user. If the error-spec for the error is of type value, then the SRL-error function returns the value. Otherwise a lisp throw is executed to return to the user level function in lisp error mode.

A list of the errors types already defined for SRL are found in appendix IV.

# 10. $SRL_o$: Object Programming in SRL+

It is often appropriate to model intelligent programming problems as heirarchical structures of entities containing both data and program code. These entities are most commonly refered to as *objects* and the programming paradigm in general is known as **object oriented programming** (first appearing in the programming language **Smalltalk**.)

This is of course a very natural thing to do in SRL since its purpose is to represent heirachies of information. SRL+ therefore presents $SRL_o$, an object oriented extension to SRL. The only new concept introduced here is a mechanism by which the contents of a slot are interpreted as program code.

## 10.1. Messages and Message Sending

An object is an ordinary schema which may have slots with program code as values or names of schemata which represent $SRL_h$ programs or $SRL_p$ systems. The result of interpreting the code is the response to sending a *message* to an object schema. This is done with the function **(send-message <message>)** where message is an instance of the following schema:

```
{{ message
     SCHEMA:
     SLOT:
     CONTEXT:
     PARAMETERS:
     DELETE:  }}
```

**Schema 10-1:** The message schema

For each member or the list found in the (valueg1 <message> "slot") slot of the (valueg1 <message> "schema"), if it is lambda expression or function name it is executed with the following list of parameters: (append (valueg <message> "schema") (valueg <message> "slot") (valueg <message> "parameters")), if it is a $SRL_h$ program schema it is loaded into the $SRL_h$ interpreter, and if it is a $SRL_p$ system schema it is loaded into the $SRL_p$ interpreter. The result of the last evaluation is returned.

There are two forms of the **send-message** function:

**(send-message** <message>)
    RETURNS: the result of the code evaluation
    NOTE: *send-message* is used to procedurally evaluate the the (valueg1 <message> slot) slot in the (valueg1 <message> schema).

(**send-message** <schema> <slot> [ <parameter-list> [<context>]])
  RETURNS: the result of the code evaluation
  NOTE: *send-message* is used to procedurally evaluate (valueg <schema> <slot>).

The second form is provided to eliminate the overhead of creating a message schema. If the first parameter to send-message satisfies (r-test <first-param> "is-a" "message"), the first form is assumed, otherwise the second is assumed.

# 11. SRL Data Base System

This chapter explains SRL's multi-user database system.

## 11.1. Introduction

DB is a mutli-process data base system especially designed for SRL2. DB creates files to store and buffer schemata, permitting programs which treat more schemata than can fit in islisp's address system. The system is designed to be invisible at the user level; the user only deals with DB when starting or terminating an islisp process. The database has the extension *db*.

A user gains access to a database by *opening* a *connection* to the database. A database connection may be read-only or write-enabled. A process makes personal changes to a database with a read-only connection; such changes are stored in the process's buffer and are invisible to other processes. When the process attempts to read the changed data, the process's buffer will be accessed instead of the original database. Database changes made during a read-only connection are only written into the original database after using the dbupdate function. A write-enabled connection writes changes directly to the database instead of placing the alterations in a buffer.

Multiple processes may have connections to the same database simultaneously, but only if all of the connections are read-only. A connection can always be changed from write-enabled to read-only, but a connection can only be changed from read-only to write-enabled if no other connection to the database is open. When changing a connection from read-only to write-enabled, a process may merge changes stored in the process's buffer with the original database.

## 11.2. DB manipulation

As soon as the number of schemata in memory exceeds the value of variable **$db-cache-max**, which is set by the user (initially 500), excess schemata will be automatically *swapped-out*. Specifically, those schemata that have been least recently accessed will be written to either the temporary buffer or the database itself, depending on the write-mode of the connection, until the number of associated schemata in memory equals the value of the variable **$db-cache-keep**, which is set by the user (initially 250). As soon as the user tries to access a swapped-out schema, it will automatically be *swapped-in* from the file.

*Currently, database connections are always read-only, except during a* ***db-update*** *call.* **Db-update** switches/changes/toggles the connection's write-mode, and can be used to merge buffered changes into the database file periodically.

An existing database can be accessed by calling **db-connect**. **db-connect** allows the user to access any schemata that were associated with the database when it was last updated. These schemata will be swapped-in automatically as the user tries to access them.

The user should not attempt to save a lisp's state, once there is an open database.

## 11.3. DB Functions

The following functions that comprise the DB system. Unless otherwise noted, the functions return **t** on success and **nil** on failure. User errors generally invoke an error break.

**(db-connect** '<db> ['<write-mode>['<share-mode>['<new-db>] ] ])
NOTE: Connects to the existing database <db>, unless <new-db> is non-nil, then the routine will create a database. The routine will fail if it tries to connect to a nonexistent database, or to create an existing one. The default is nil. <write-mode> must be *db-write-enabled* or *db-read-only*. The default is *db-read-only*. <share-mode> indicates whether the database is to be shared with another process concurrently. <share-mode> must be *db-shared* or *db-exclusive*. All databases end in the extension .db. If the <db> argument ends in .db, then the argument is used as is. If the argument doesn't end in <db>, the extension <db> is added. Therefore if <db> were *basic* the system would use *basic.db*. There might be some system specific restrictions (set by the user) in addition to those listed.

**(db-update )**
NOTE: Merges buffered changes for database <db>.

**(db-close )**
NOTE: Closes the database, and flushes all schemata from memory. After a **db-close** it is possible to save the lisp.

## 11.4. Utility Functions

Here are utility functions, which might be of interest to users working with DB in particular applications.

**(db-get-db )**
RETURNS: the name of the database SRL is connected to, or **nil** if there is no database.

**(db-check-db** '<db>)
RETURNS: **t** if islisp is connected to <db>, and **nil** otherwise.

**(db-get-contexts )**
RETURNS: all contexts associated with the current data base.

**(db-get-schemata** [<context>])
RETURNS: a list of all schemata in <context> associated with the database.

**(db-update-context** <context>)
SIDE-EFFECT: Writes all changes to schemata in <context> to the database.

**(db-change-write-mode** <db> <write-mode>)
SIDE-EFFECT: Sets the writemode of <db> to be <write-mode>.

## 11.5. Global Parameters

These are global parameters to DB, which the user can set.

**$db-cache-unlimited**: If this is non-nil, then the swapping-out of schemata will be temporarily disabled.

**$db-cache-max**: The maximum number of schemata that can be resident in a cache.

**$db-cache-keep**: The number of schemata that should remain resident after excess schemata are swapped-out.

**$queuelength**: Holds the number of schemata currently resident in the cache.

# Appendix I Using SRL2

This appendix describes how to start up an SRL system for your personal use. Defined here is how the system is used on machines in the Carnegie-Mellon environment. Use outside the environment may vary greatly.

## I.1. Initializing SRL2

SRL2 is implemented in Franzlisp (Foderaro, 80) running on a VAX running under the UNIX operating system. The following steps should be followed in starting SRL2 for the first time:

1. Find the account where the SRL system is maintained on your machine. At CMU it is [isl1]/usr/islisp/islisp6.

2. Find the database directory in that account that contains the basic database file. At CMU it is [isl1]/usr/islisp/islisp6/db/basic.db.

3. Copy basic.db onto your account. This will be the kernel database system that you will use.

4. Run SRL2 by typing the name of the SRL lisp system to the unix shell. At CMU it is called islisp6 ([isl1]/usr/islisp/bin/islisp6).

5. Connect to your basic database system by typing to your SRL lisp system:

   ```
   (db-connect 'basic)
   ```

   You are now ready to use SRL2

Upon starting SRL2, a root context will have been created called "$root-context". This will be used as the default context until otherwise changed by the user.

## I.2. System Limitations

Due to database requirements, all schema and context names in srl must be strings of at most 30 characters. I addition the character " + " has been reserved and should not be used in the names of schemata. File identifiers for the data base system must by lisp symbols of at most 99 characters.

# Appendix II
# User Commands

# Appendix III
# Database Backup System

It is sometimes useful to backup a number of schemata residing in lisp or a database into a file of executable SRL commands. A backup of this sort makes it possible to recreate the schemata from a session that were otherwise lost due to an abortive lisp process or a loss of database integrity. The following describes a system to accomplish this. Note, however, that the form of the SRL commands created by this process bare little resemblance to the SRL commands available to the user. To create a backup file, follow the steps below:

1. Start islisp. If schemata are to be backed up from a database, connect to the database(s).
2. Declare a file to lisp using the lisp *file* command.
3. For each schema to be stored in the file, use the *file-func* command described below.
4. When all desired schemata have been associated with a file, use the *save-file* command described below to write the file.

The command *find-file* is provided so the user may determine which file a schema is associated with. To restore the schemata to lisp, use the lisp *dskin* command.

**(file-func** <schema> '<file>)
SIDE-EFFECT: Associates <schema> with <file> so that when <file> is updated, <schema> will be saved as part of that update

**(save-file** '<file>)
SIDE-EFFECT: All schemata associated with <file> are stored in file.

**(find-file** <schema>)
RETURNS: the name of the file that <schema> is associated with.

# Appendix IV
# Error Messages

The following are the defined error types in SRL:

**no-schema:**

DEFINITION: Schema specified in the range of a relation does not exist.
SOURCE: slot and value inheritance.
CONTINUATION EFFECT: search continues, but further errors may occur.

**illegal-schema-name:**

DEFINITION: Schema name specified was not of the proper data type (lisp string).
SOURCE: Schema creation or access.
CONTINUATION EFFECT:

**no-slot:**

DEFINITION: A slot referred to during access does not exist, nor is inheritable by the schema.
SOURCE: value creation/appending.
CONTINUATION EFFECT: execution continues with further errors possible.

**no-value:**

DEFINITION:
SOURCE:
CONTINUATION EFFECT:

**range:**

DEFINITION: A range check on the new value of slot failed.
SOURCE: value creation/appending.
CONTINUATION EFFECT: execution continues with illegal value placed in slot.

**domain-error:**

DEFINITION: Illegal domain for created relation (slot).
SOURCE: slot creation (slotc).
CONTINUATION EFFECT: slot is created.

**illegal-path**

DEFINITION: search spec for a slot access is incorrectly defined.
SOURCE: any slot access function (internal: find-slot and find-value).
CONTINUATION EFFECT: search is pruned at point of error.

**bad-elab-rel-spec:**

DEFINITION: contents of relation slot in elaboration spec are in error.
SOURCE: interpretation of an elaboration spec.
CONTINUATION EFFECT: relation is not created.

**no-map-domain:**

DEFINITION: during the interpretation of a map specification, the domain slot to be mapped is not defined.
SOURCE: map specification evaluation.
CONTINUATION EFFECT: the map spec is ignored.

**illegal-predicate:**

DEFINITION: illegal/undefined predicate in the condition of slot of a relation.
SOURCE: during relation inheritance interpretation.
CONTINUATION EFFECT: value returned by SRL-error is used as the condition's value.

**invalid-context:**

DEFINITION: Context does not exist, or name specified is of wrong data type (must be a lisp string).
SOURCE: Context creation and access
CONTINUATION EFFECT:

**invalid-database:**

DEFINITION: Database is not currently active or name specified is not a lisp symbol.
SOURCE:
CONTINUATION EFFECT:

# Appendix V System Relations

```
{{ mapped-to
        IS-A: "relation" "metaslot-relation"
        INVERSE: "mapped-from"
        FUNCTION: (function no-inheritance)}}
```

```
{{ mapped-from
        IS-A: "relation" "metaslot-relation"
        INSTANCE-MAP:
        INSTANCE-CONTEXT:
        INVERSE: "mapped-to"
        FUNCTION: (function no-inheritance)}}
```

```
{{ elaborated-to
        IS-A: "relation" "metaslot-relation"
        INVERSE: "elaborated-from"
        FUNCTION: (function no-inheritance)}}
```

```
{{ elaborated-from
        IS-A: "relation" "metaslot-relation"
        INSTANCE-MAP:
        INSTANCE-CONTEXT:
        INVERSE: "elaborated-to"
        FUNCTION: (function no-inheritance)}}
```

```
{{ included-to
        IS-A: "relation" "metaslot-relation"
        INVERSE: "included-from"
        FUNCTION: (function no-inheritance)}}
```

```
{{ included-from
      IS-A: "relation" "metaslot-relation"
      INSTANCE-MAP:
      INSTANCE-CONTEXT:
      INVERSE: "included-to"
      FUNCTION: is-a-fn
      INCLUSION: "is-a-inclusion-spec"}}
```

```
{{ metaslot-relation
      IS-A: "relation"
      IS-A+INV: "mapped-to" "mapped-from" "elaborated-to"
                "elaborated-from" "included-to" "included-from"
      FUNCTION: (function do-not-use-message)}}
```

```
{{ basic-relation
      IS-A: "relation"
      IS-A+INV: "instance+inv" "instance" "is-a+inv" "is-a"
      FUNCTION: (function do-not-use-message)}}
```

```
{{ instance
      IS-A: "relation" "basic-relation"
      TRANSITIVITY: (list (step instance t) (repeat (step is-a t) 0 inf))
      INCLUSION: "is-a-inclusion-spec"
      INVERSE: "instance+inv"}}
```

```
{{ instance+inv
      IS-A: "relation" "basic-relation"
      INVERSE: "instance"}}
```

```
{{ is-a
      IS-A: "relation" "basic-relation"
      TRANSITIVITY: (list (repeat (step instance t) 0 1) (repeat (step is-a t) 0 inf))
      INCLUSION: "is-a-inclusion-spec"
      INVERSE: "is-a+inv"}}
```

```
{{ is-a+inv
        IS-A: "basic-relation"
        INVERSE: "is-a"}}
```

```
{{ is-a-inclusion-spec
        IS-A: "inclusion-spec"
        SLOT-RESTRICTION: (not (or "is-a" "instance" "is-a+inv"
                              "instance+inv"))
        VALUE-RESTRICTION: t
        CONDITION: t}}
```

# Appendix VI History

Schema Representation Language (SRL) is a family of artificial intelligence knowledge representation languages, which have evolved over the past several years at CMU. The initial reason for creating another knowledge representation was to explore issues in inheritance (Fox, 1979). As the first version of SRL became widely used in the Intelligent Systems Laboratory, other reasons for developing another version arose. A language was needed that was efficient, adaptable, simple to use, and capable of supporting large applications requiring requiring some form of database management. SRL has also been expanded to incorporate many of the ideas found in current knowledge representation systems (Brachman, 1977; Fahlman, 1977; Levesque & Mylopolous, 1977).

Much of the power of knowledge representation systems is derived from their inheritance mechanisms, and from the formalization of their semantics. This work contributes primarily to the definition of inheritance mechanisms, but also to the formalization of semantics.

SRL approaches inheritance and semantics by:

- allowing users to define new relations and their inheritance semantics declaratively.
- enabling and differentiating multi-path inheritance.
- allowing schemata to inherit from their parts.
- allowing the user to define new slots, facets, and meta-information.
- allowing selective search specifications by the user.

Early versions of SRL offered an inheritance mechanism with power similar to that of SRL2, but the cost of interpreting relations with this mechanism proved prohibitive. Subsequent versions of the language restricted the definition of the facility to help speed up the system. However, the development of a relation compiler made it possible to resurrect the inheritance mechanism in its full generality. Changes in the restriction mechanism increased the regularity of the facility's grammar, and its generality. SRL2 incorporates these changes in the inheritance mechanism and the restriction facility.

Solving other problems also resulted in SRL's improvement and the user's benefit. For example, our applications required the definition of new relations having inheritance semantics peculiar to the domain. Rather than search for a few universal inheritance relations to solve this problem, a set of primitives was created for defining these relations and their inheritance semantics. This type of feature offers the user maximum flexibility in defining the knowledge representation system.

A third problem, encountered in the definition of SRL, was the basic difference between relations and attributes. It was hard to explain why the possession of an attribute was any different than a structural relation or a sub-category relation (e.g., is-a). In decribing a "person," is having a "head" considered an attribute or a relation? Does having a parameter only imply that there is a relation "to have" or "possess" between the object and the attribute? SRL solves these problems with slots, which are relations where information can be transferred bi-directionally. Slots give the user flexibility in constructing a knowledge representation system by allowing him to define a relation's semantics as he chooses.

SRL2 serves as the hub of a large body of software developed in the Intelligent Systems Laboratory and is the result of many people's efforts. In particular, we would like to thank Joe Mattis and Drew Mendler for their work on the Database system. We would also like to thank Brad Allen, Mike Greenberg, Mike Rychener and Gary Strohm for their help, comments and ideas.

# References

Bartlett, F.C., (1932), *Remembering*, Cambridge: Cambridge University Press.

Bobrow D., and Winograd, T., (1977), "KRL: Knowledge Representation Language," *Cognitive Science*. Vol 1, No. 1, 1977.

Brachman, R.J., (1977), "A Structural Paradigm for Representing Knowledge," (Ph.D. Thesis), Harvard University, May 1977.

Fahlman, S.E., (1977), "A System for Representing and Using Real-World Knowledge," (Ph.D. Thesis), Artificial Intelligence Laboratory, MIT, AI-TR-450.

Foderaro, J. K., (1980), The FRANZ LISP manual, University of California at Berkeley.

Fox, M.S., (1979), "On Inheritance in Knowledge Representation", *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, Tokyo, Japan.

Fox, M.S., (1981), "The Intelligent Management System: An Overview", Technical Report CMU-RI-TR-81-4, Robotics Institute, Carnegie-Mellon University, Pittsburgh, PA, July 1981.

Kowalski, R.A., (1979), "Logic for Problem Solving", North-Holland, New York.

Lenat, D., (1976), "AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search," (Ph.D. Thesis) Computer Science Dept., Stanford University.

Levesque, H., and Mylopoulos, J., (1978), "A Procedural Semantics for Semantic Networks," AI MEMO 78-1, Dept. of Computer Science, University of Toronto.

Minsky M., (1975), "A Framework for Representing Knowledge", In *The Psychology of Computer Vision*", P. Winston (Ed.), New York: McGraw-Hill.

# Index

$access 79
$cache 60, 79
$cardinality 37
$context 74, 75
$current-error 79, 83
$db-cache-keep 79, 88, 90
$db-cache-max 79, 88, 90
$db-cache-unlimited 79, 90
$default-context 79
$demon 79
$inherit-all 79
$inverse 41, 79
$local 62, 79
$meta-schema 79
$meta-slot 80
$meta-value 80
$queuelength 79, 90
$restrict 24, 80
$root-context 74, 75, 79, 91
$slot-dependency 60, 70, 80
$value-dependency 60, 71, 80

Cardinality facet 37
Context-children function 76
Context-p function 76
Context-parent function 76
Contexta function 76
Contextc function 75
Contextd function 75
Contextm function 76

Db-change-write-mode function 89
Db-check-db function 89
Db-close function 89
Db-connect function 89
Db-get-contexts function 89
Db-get-db function 89
Db-get-schemata function 89
Db-update function 89
Db-update-context function 89
Demon facet 37
Domain facet 24, 33, 46, 48

File-func function 95
Find-file function 95

In-context-p function 76

Mschema-p function 20
Mschemac function 20
Mschemad function 20
Mschemag function 19
Mslot-p function 26
Mslotc function 26
Mslotd function 26
Mslotg function 25
Munit-p function 32
Munitc function 32
Munitd function 32
Munitg function 32

Ps function 17

R-find function 69, 71
R-test function 69, 71
Range facet 33, 48
Relation function 58, 63
Relation-p function 58, 63
Relationc function 59, 63
Relationd function 59, 63
Relationp function 58, 63
Restriction-test function 36, 39

Save-file function 95
Schema-p function 17
Schemac function 16
Schemad function 17
Send-message function 85, 86
Slot function 23, 70
Slot-all function 23, 70
Slot-elab function 24, 53
Slot-unlab function 24, 53
Slotc function 23
Slotd function 24
Slote function 23, 70
Srl-get function 78, 80
Srl-set function 78, 80
SRLinit function 78, 80
SRLpp function 78, 80

Unitg function 32
Uvalued-pred function 31
Uvalueg function 31
Uvalueg1 function 31
Uvaluegn function 31

Value-sort function 30
Valuea function 29
Valuea1 function 29
Valuec function 29
Valuec1 function 29
Valuecn function 29
Valued function 29
Valued-pred function 30
Valuedn function 30
Valuedq function 29
Valueg function 30
Valueg1 function 30
Valuegn function 30
Valuez function 29