Loom Reference Manual
Version 2.0

David Brill

December 28, 1993

Contents

LOOM 2.0

1 Introduction

This Manual is written as a reference document for users familiar with the basic principles of the Loom language, or of other classification-based knowledge representation languages. It contains a quick reference quide to the Loom system, detailed descriptions of each function in the Loom application programmer interface, and a complete grammar for the Loom definition and assertion languages.

The Loom Reference Manual is a companion volume to the Loom Users Guide. Many functions documented here, particularly those used to define concepts and relations, are described at greater length in that guide. Also, the Users Guide discusses the overall Loom architecture and the factors that motivated various design decisions—issues that are beyond the scope of this document.

2 Loom Quick Guide

2.1 Fundamental Operations

2.1.1 Definitions

`defaction` *name parameters* `&key filters missing-methods` [*Macro*] (p. 46)

`defconcept` *name* `&optional` *documentation* `&key is is-primitive implies defaults partitions exhaustive-partitions in-partition predicate function roles indices keys mixin-classes mixin-slots annotations identifier kb characteristics` [*Macro*] (p. 49)

`define-concept &key name is is-primitive implies defaults partitions exhaustive-partitions in-partition predicate function mixin-classes mixin-slots annotations identifier kb characteristics` [*Function*] (p. 57)

`define-relation &key name is is-primitive implies domain domains range arity predicate function inheritance-link inheritance-method annotations identifier kb characteristics` [*Function*] (p. 58)

`defkb` *name parentKbs* `&key path-name package export-names-p monotonic-p` [*Macro*] (p. 59)

`defmethod` *name parameters* `&key title situation with overrides response` [*Macro*] (p. 61)

`defproduction` *name* `&key when perform schedule do priority` [*Macro*] (p. 64)

`defrelation` *name* `&optional` *documentation* `&key is is-primitive implies domain domains range arity predicate function inheritance-link inheritance-method annotations identifier kb characteristics` [*Macro*] (p. 66)

`defset` *name* `&optional` *documentation* `&key is annotations identifier kb` [*Macro*] (p. 72)

2.1.2 Implications

`default` *antecedent consequent* [*Macro*] (p. 48)

`implies` *antecedent consequent* [*Macro*] (p. 148)

`term-implies` *concept1 concept2* [*Relation*] (p. 278)

2.1.3 Assertions and Retractions

`add-type` *instance concept* `&key kb no-error-p` [*Function*] (p. 17)

`add-value` *instance role value* `&key kb no-error-p` [*Function*] (p. 18)

`create` *identifier concept* `&key kb add-suffix-p clos-instance-p` [*Function*] (p. 42)

`destroy` *instance* `&key dont-unintern-p` [*Method*] (p. 78)

`forget &body` *propositions* [*Macro*] (p. 109)

`remove-type` *instance concept* `&key kb no-error-p` [*Function*] (p. 235)

`remove-value` *instance role value* `&key kb no-error-p`	[*Function*]	(p. 236)
`set-value` *instance role value* `&key kb no-error-p`	[*Function*]	(p. 261)
`set-values` *instance role values* `&key kb no-error-p`	[*Function*]	(p. 262)
`tell &body` *propositions*	[*Macro*]	(p. 275)

2.1.4 Retrieval

`ask` *query* `&key kb 3-valued-p`	[*Macro*]	(p. 20)
`concept--instances` *concept instance*	[*Relation*]	(p. 35)
`do-retrieve` *variables query* `&body` *body*	[*Macro*]	(p. 84)
`get-indices` *concept* `&key direct-p`	[*Function*]	(p. 117)
`get-instances` *concept* `&key direct-p asserted-p`	[*Function*]	(p. 119)
`get-inverse-values` *instance role* `&key kb no-error-p`	[*Function*]	(p. 120)
`get-keys` *concept* `&key direct-p`	[*Function*]	(p. 121)
`get-matching-instances` *concepts roleFillers*	[*Function*]	(p. 122)
`get-types` *instance* `&key asserted-p direct-p`	[*Function*]	(p. 141)
`get-value` *instance role* `&key direct-p kb no-error-p`	[*Function*]	(p. 142)
`get-values` *instance role* `&key direct-p kb no-error-p`	[*Function*]	(p. 144)
`list-inverse-role-names&values` *instance*	[*Function*]	(p. 171)
`list-role-names&values` *instance*	[*Function*]	(p. 178)
`list-tuples` *relation*	[*Function*]	(p. 181)
`query` *variables expression*	[*Function*]	(p. 227)
`retrieve` *variables query* `&key kb generators`	[*Macro*]	(p. 246)
`role-members` *instance relation value*	[*Relation*]	(p. 250)

2.1.5 Finding an Object

`fb` *behaviorOrTitle* `&optional` *title*	[*Macro*]	(p. 87)
`fc` *datum* `&key kb`	[*Macro*]	(p. 89)
`fi` *datum* `&key kb`	[*Macro*]	(p. 92)
`find-action` *action* `&key no-warning-p`	[*Function*]	(p. 95)
`find-concept` *concept* `&key no-warning-p ignore-package-p kb`	[*Function*]	(p. 96)
`find-instance` *instance* `&key no-warning-p ignore-package-p kb`	[*Function*]	(p. 99)
`find-kb` *kb* `&key no-warning-p`	[*Function*]	(p. 101)
`find-method` *actionOrTitle* `&optional` *title* `&rest` *no-error-p*	[*Macro*]	(p. 103)
`find-production` *production* `&key no-warning-p`	[*Function*]	(p. 105)
`find-relation` *relation* `&key no-warning-p ignore-package-p kb`	[*Function*]	(p. 106)
`fr` *datum* `&key kb`	[*Macro*]	(p. 111)
`get-action` *action* `&key no-error-p`	[*Function*]	(p. 115)
`get-concept` *concept* `&key no-error-p kb`	[*Function*]	(p. 116)
`get-instance` *instance* `&key error-p kb`	[*Function*]	(p. 118)
`get-method` *action title* `&key no-error-p`	[*Function*]	(p. 124)
`get-production` *production* `&key no-error-p`	[*Function*]	(p. 126)

`get-relation` *relation* `&key no-error-p kb`	[*Function*]	(p. 127)
`identifier--instance` *symbol instance*	[*Relation*]	(p. 147)
`name--concept` *symbol concept*	[*Relation*]	(p. 203)
`name--relation` *symbol relation*	[*Relation*]	(p. 204)
`where-is-it` *object*	[*Function*]	(p. 295)

2.1.6 Subsumption and Instantiation

`concept--instances` *concept instance*	[*Relation*]	(p. 35)
`direct-subconcepts` *concept1 concept2*	[*Relation*]	(p. 80)
`direct-superconcepts` *concept1 concept2*	[*Relation*]	(p. 81)
`find-subsumers&subsumees` *expression* `&optional` *kb*	[*Function*]	(p. 108)
`get-instances` *concept* `&key direct-p asserted-p`	[*Function*]	(p. 119)
`get-subconcepts` *concept* `&key direct-p`	[*Function*]	(p. 137)
`get-subrelations` *relationOrConcept* `&key direct-p`	[*Function*]	(p. 138)
`get-superconcepts` *concept* `&key direct-p`	[*Function*]	(p. 139)
`get-superrelations` *relationOrConcept* `&key direct-p`	[*Function*]	(p. 140)
`instance--asserted-concepts` *instance concept*	[*Relation*]	(p. 156)
`instance--cached-concepts` *instance concept*	[*Relation*]	(p. 157)
`instance--concepts` *instance concept*	[*Relation*]	(p. 158)
`instance--type` *instance concept*	[*Relation*]	(p. 160)
`isa-p` *object concept*	[*Relation*]	(p. 162)
`most-general-concepts` *conceptList*	[*Function*]	(p. 200)
`most-specific-concepts` *conceptList*	[*Function*]	(p. 201)
`subconcept-p` *concept1 concept2* `&key cant-be-equal-p`	[*Macro*]	(p. 268)
`subconcepts` *concept1 concept2*	[*Relation*]	(p. 269)
`superconcepts` *concept1 concept2*	[*Relation*]	(p. 273)
`test-not-type-p` *instance concept* `&key kb no-error-p`	[*Function*]	(p. 279)
`test-type-p` *instance concept* `&key kb no-error-p`	[*Function*]	(p. 280)

2.2 Programming

2.2.1 Control

`call-use-loom` *packageName* `&key dont-create-knowledge-base-p knowledge-base-name path-name loom-imports`	[*Function*]	(p. 24)
`createm` *identifier concept* `&key kb add-suffix-p clos-instance-p`	[*Macro*]	(p. 42)
`destroym` *instance* `&key dont-unintern-p`	[*Macro*]	(p. 78)
`finalize-definitions`	[*Function*]	(p. 94)
`forgetm &body` *propositions*	[*Macro*]	(p. 109)
`initialize-instances`	[*Function*]	(p. 152)
`initialize-network &key destroy-kbs-p`	[*Function*]	(p. 153)
`initialize-tasks`	[*Function*]	(p. 154)

2.2.2 Debugging

2.3 Loom Objects

2.3.1 Concepts and Relations

2.3.2 Roles

`Min-Restriction-P` *object*	[*Concept*]	(p. 199)
`relation--restrictions` *relation restriction*	[*Relation*]	(p. 234)
`Restriction-P` *object*	[*Concept*]	(p. 240)
`restriction--concepts` *restriction concept*	[*Relation*]	(p. 241)
`restriction--max` *restriction max*	[*Relation*]	(p. 242)
`restriction--min` *restriction min*	[*Relation*]	(p. 243)
`restriction--relation` *restriction relation*	[*Relation*]	(p. 244)
`restriction--value-restriction` *restriction vr*	[*Relation*]	(p. 245)
`Value-Restriction-P` *object*	[*Concept*]	(p. 294)

2.3.3 Instances

`change-object` *instance class*	[*Function*]	(p. 27)
`copy-instance` *instance* `&key identifier kb add-suffix-p`	[*Function*]	(p. 40)
`create` *identifier concept* `&key kb add-suffix-p clos-instance-p`	[*Function*]	(p. 42)
`destroy` *instance* `&key dont-unintern-p`	[*Method*]	(p. 78)
`do-instances` *variable* `&body` *body*	[*Macro*]	(p. 83)
`find-knowledge-base-of-instance` *instance*	[*Function*]	(p. 102)
`find-or-create-instance` *instance concept*	[*Function*]	(p. 104)
`forget &body` *propositions*	[*Macro*]	(p. 109)
`Incoherent` *object*	[*Concept*]	(p. 150)
`instance-p` *object*	[*Function*]	(p. 155)
`instance--identifier` *instance symbol*	[*Relation*]	(p. 159)
`list-instances`	[*Function*]	(p. 170)
`loom-concept` *instance*	[*Slot*]	(p. 186)
`make-object` *class* `&body` *initargs* `&key identifier kb`	[*Macro*]	(p. 189)
`object-name` *object*	[*Method*]	(p. 209)
`tell &body` *propositions*	[*Macro*]	(p. 275)
`unmake-object` *instance*	[*Function*]	(p. 285)

2.3.4 Frames

`add-type` *instance concept* `&key kb no-error-p`	[*Function*]	(p. 17)
`add-value` *instance role value* `&key kb no-error-p`	[*Function*]	(p. 18)
`fadd-value` *instance role value*	[*Function*]	(p. 85)
`fget-value` *instance role*	[*Macro*]	(p. 91)
`fremove-value` *instance role value*	[*Function*]	(p. 113)
`fset-value` *instance role value*	[*Function*]	(p. 114)
`get-instances` *concept* `&key direct-p asserted-p`	[*Function*]	(p. 119)
`get-inverse-values` *instance role* `&key kb no-error-p`	[*Function*]	(p. 120)
`get-name` *object*	[*Function*]	(p. 125)
`get-subconcepts` *concept* `&key direct-p`	[*Function*]	(p. 137)
`get-subrelations` *relationOrConcept* `&key direct-p`	[*Function*]	(p. 138)

`get-superconcepts` *concept* `&key direct-p`	[*Function*]	(p. 139)
`get-superrelations` *relationOrConcept* `&key direct-p`	[*Function*]	(p. 140)
`get-types` *instance* `&key asserted-p direct-p`	[*Function*]	(p. 141)
`get-value` *instance role* `&key direct-p kb no-error-p`	[*Function*]	(p. 142)
`get-values` *instance role* `&key direct-p kb no-error-p`	[*Function*]	(p. 144)
`remove-type` *instance concept* `&key kb no-error-p`	[*Function*]	(p. 235)
`remove-value` *instance role value* `&key kb no-error-p`	[*Function*]	(p. 236)
`set-value` *instance role value* `&key kb no-error-p`	[*Function*]	(p. 261)
`set-values` *instance role values* `&key kb no-error-p`	[*Function*]	(p. 262)
`test-not-type-p` *instance concept* `&key kb no-error-p`	[*Function*]	(p. 279)
`test-type-p` *instance concept* `&key kb no-error-p`	[*Function*]	(p. 280)

2.3.5 Knowledge Bases

`change-kb` *knowledgeBase* `&key no-checking-p`	[*Function*]	(p. 26)
`clear-kb &optional` *knowledgeBase* `&key partitions`	[*Macro*]	(p. 29)
`current-kb`	[*Function*]	(p. 45)
`defkb` *name parentKbs* `&key path-name package export-names-p monotonic-p`	[*Macro*]	(p. 59)
`find-kb` *knowledgeBase* `&key no-warning-p`	[*Function*]	(p. 101)
`find-knowledge-base-of-instance` *instance*	[*Function*]	(p. 102)
`in-kb` *knowledgeBase*	[*Macro*]	(p. 149)
`Knowledge-Base` *object*	[*Concept*]	(p. 164)
`list-kb &optional` *knowledgeBase* `&key partitions sort-p`	[*Macro*]	(p. 172)
`list-knowledge-bases`	[*Function*]	(p. 174)
`load-kb` *knowledgeBase* `&key path-name`	[*Function*]	(p. 184)
`save-kb &optional` *knowledgeBase* `&key partitions path-name`	[*Function*]	(p. 252)
`use-loom` *packageName* `&key dont-create-knowledge-base-p knowledge-base-name path-name loom-imports`	[*Macro*]	(p. 291)

2.3.6 Behaviors

`defaction` *name parameters* `&key filters missing-methods`	[*Macro*]	(p. 46)
`defmethod` *name parameters* `&key title situation with overrides response`	[*Macro*]	(p. 61)
`defproduction` *name* `&key when perform schedule do priority`	[*Macro*]	(p. 64)
`delete-all-methods` *action*	[*Function*]	(p. 74)
`delete-method &key action title method error-p`	[*Function*]	(p. 76)
`delete-production` *production* `&key error-p`	[*Function*]	(p. 77)
`fail &optional` *result*	[*Macro*]	(p. 86)
`fb` *behaviorOrTitle* `&optional` *title*	[*Macro*]	(p. 87)
`find-action` *action* `&key no-warning-p`	[*Function*]	(p. 95)
`find-method` *actionOrTitle* `&optional` *title* `&rest` *no-error-p*	[*Macro*]	(p. 103)

`find-production` *production* `&key no-warning-p`	[*Function*]	(p. 105)
`get-action` *action* `&key no-error-p`	[*Function*]	(p. 115)
`get-method` *action title* `&key no-error-p`	[*Function*]	(p. 124)
`get-production` *production* `&key no-error-p`	[*Function*]	(p. 126)
`initialize-tasks`	[*Function*]	(p. 154)
`kill-task` *task*	[*Function*]	(p. 163)
`list-methods` *action*	[*Function*]	(p. 176)
`list-productions &key kb`	[*Function*]	(p. 177)
`list-tasks` *list-tasks* `&key priorities`	[*Function*]	(p. 180)
`pb` *behavior* `&optional` *title*	[*Macro*]	(p. 210)
`perform` *(actionName* `&rest` *arguments)* `&key returnOption`	[*Macro*]	(p. 214)
`perform-task` *task returnOption*	[*Function*]	(p. 216)
`print-methods` *action*	[*Function*]	(p. 226)
`schedule` *(actionName* `&rest` *arguments)* `&key priority`	[*Macro*]	(p. 254)
`schedule-task` *task*	[*Function*]	(p. 256)
`scheduled-p` *task*	[*Function*]	(p. 257)

2.4 Built-In Predicates

2.4.1 Built-In Concepts

`Character` *object*	[*Concept*]	(p. 28)
`Concept` *object*	[*Concept*]	(p. 32)
`Cons-Or-Null` *object*	[*Concept*]	(p. 38)
`Constant` *object*	[*Concept*]	(p. 39)
`Incoherent` *object*	[*Concept*]	(p. 150)
`Incoherent-Concept` *object*	[*Concept*]	(p. 151)
`Integer` *object*	[*Concept*]	(p. 161)
`Knowledge-Base` *object*	[*Concept*]	(p. 164)
`Loom-Thing` *object*	[*Concept*]	(p. 188)
`Max-Restriction-P` *object*	[*Concept*]	(p. 193)
`Meta-Concept` *object*	[*Concept*]	(p. 196)
`Min-Restriction-P` *object*	[*Concept*]	(p. 199)
`Non-Loom-Thing` *object*	[*Concept*]	(p. 207)
`Number` *object*	[*Concept*]	(p. 208)
`Primitive-P` *object*	[*Concept*]	(p. 225)
`Relation` *object*	[*Concept*]	(p. 229)
`Restriction-P` *object*	[*Concept*]	(p. 240)
`Single-Valued-P` *object*	[*Concept*]	(p. 266)
`String` *object*	[*Concept*]	(p. 267)
`Symbol` *object*	[*Concept*]	(p. 274)
`Thing` *object*	[*Concept*]	(p. 281)
`User-Defined-P` *object*	[*Concept*]	(p. 293)

`Value-Restriction-P` *object* [*Concept*] (p. 294)

2.4.2 Built-In Relations

`concept--characteristics` *concept characteristic* [*Relation*] (p. 34)
`concept--instances` *concept instance* [*Relation*] (p. 35)
`concept--name` *concept symbol* [*Relation*] (p. 36)
`concept--restrictions` *concept restriction* [*Relation*] (p. 37)
`count` *setOfObjects number* [*Relation*] (p. 41)
`direct-subconcepts` *concept1 concept2* [*Relation*] (p. 80)
`direct-superconcepts` *concept1 concept2* [*Relation*] (p. 81)
`identifier--instance` *symbol instance* [*Relation*] (p. 147)
`instance--asserted-concepts` *instance concept* [*Relation*] (p. 156)
`instance--cached-concepts` *instance concept* [*Relation*] (p. 157)
`instance--concepts` *instance concept* [*Relation*] (p. 158)
`instance--identifier` *instance symbol* [*Relation*] (p. 159)
`instance--type` *instance concept* [*Relation*] (p. 160)
`isa-p` *object concept* [*Relation*] (p. 162)
`max` *setOfNumbers number* [*Relation*] (p. 192)
`member-of` *object set* [*Relation*] (p. 194)
`members` *set object* [*Relation*] (p. 195)
`min` *setOfNumbers number* [*Relation*] (p. 198)
`name--concept` *symbol concept* [*Relation*] (p. 203)
`name--relation` *symbol relation* [*Relation*] (p. 204)
`predecessor` *set member1 member2* [*Relation*] (p. 224)
`relation--domain` *relation concept* [*Relation*] (p. 231)
`relation--name` *relation symbol* [*Relation*] (p. 232)
`relation--range` *relation concept* [*Relation*] (p. 233)
`relation--restrictions` *relation restriction* [*Relation*] (p. 234)
`restriction--concepts` *restriction concept* [*Relation*] (p. 241)
`restriction--max` *restriction max* [*Relation*] (p. 242)
`restriction--min` *restriction min* [*Relation*] (p. 243)
`restriction--relation` *restriction relation* [*Relation*] (p. 244)
`restriction--value-restriction` *restriction vr* [*Relation*] (p. 245)
`role-members` *instance relation value* [*Relation*] (p. 250)
`same-as` *set1 set2* [*Relation*] (p. 251)
`subconcepts` *concept1 concept2* [*Relation*] (p. 269)
`subset` *set1 set2* [*Relation*] (p. 270)
`successor` *set member1 member2* [*Relation*] (p. 271)
`sum` *setOfNumbers number* [*Relation*] (p. 272)
`superconcepts` *concept1 concept2* [*Relation*] (p. 273)
`term-implies` *concept1 concept2* [*Relation*] (p. 278)
+ *number1 number2 number3* [*Relation*] (p. 15)

− *number1 number2 number3*	[*Relation*]	(p. 15)
> *number1 number2*	[*Relation*]	(p. 16)
< *number1 number2*	[*Relation*]	(p. 16)
= *number1 number2*	[*Relation*]	(p. 16)
>= *number1 number2*	[*Relation*]	(p. 16)
<= *number1 number2*	[*Relation*]	(p. 16)
/ = *number1 number2*	[*Relation*]	(p. 16)

3 Loom Function Library

This section summarizes the functions which constitute the application programmer interface (API) to Loom. Each entry describes a particular Lisp function or macro in the interface, or a Loom concept or relation that can be used in conjunction with the Loom query facility to obtain meta-level information. The function descriptions contain the following information:

- a brief statement of the function's purpose;
- a calling syntax giving the function's name and the names of its arguments, including **&optional**, **&key**, and **&rest** arguments;
- a type specification for each argument, along with default values for optional arguments;
- the value or values returned by the function, and possible error conditions;
- general remarks about the semantics and use of the function, including information about side-effects;
- examples showing calls to the function, and indicating the type of reasoning that the function implements;
- cross-references to related functions.

+ *Relation*
− *Relation*

Purpose:

The + and − relations are functions used for performing arithmetic operations.

Syntax:

+ *number1 number2 number3*
− *number1 number2 number3*

Domains:

The *number1* and *number2* arguments are the integer or real numbers to be added or subtracted.

Range:

The *number3* argument is the integer or real result.

Examples:

```
(tellm (:about Joe Man (age 40) (wife Sue)) (age Sue 30))
(ask (- (age Joe) (age Sue) 10)) ⇒ T
(retrieve ?x (:and (Man ?x) (> (age ?x) (+ (age (wife ?x)) 5)))) ⇒
    (|I|JOE)
```

See Also:

count (p. 41)
sum (p. 272)
/ = (p. 16)

/ = *Relation*
< *Relation*
<= *Relation*
= *Relation*
> *Relation*
>= *Relation*

Purpose:

These relations are predicates used for comparing numbers.

Syntax:

/ = *number1 number2*
< *number1 number2*
<= *number1 number2*
= *number1 number2*
> *number1 number2*
>= *number1 number2*

Domain:

The *number1* argument is the integer or real number that is to be compared with *number2*.

Range:

The *number2* argument is an integer or real number.

Examples:

```
(tellm (:about Joe Man (age 65) (wife Sue)) (age Sue 66))
(ask (>= (age Joe) 65)) ⇒ T
(retrieve ?x (:and (Man ?x) (< (age ?x) (age (wife ?x))))) ⇒ (|I|JOE)
```

See Also:

max (p. 192)
min (p. 198)
+ (p. 15)

add-type *Function*

Purpose:

The **add-type** function adds a concept to the type of a given instance.

Syntax:

add-type *instance concept* `&key` *kb no-error-p*

Arguments:

The *instance* argument is an instance, or an instance identifier.

The *concept* argument is the concept, or the name of the concept, being added.

The *kb* argument is the knowledge base, or the symbol or string that names the knowledge base, where the lookup of *instance* and *concept* begins. This defaults to the current knowledge base.

If the *no-error-p* argument is **t**, then error messages are suppressed.

Value:

If the update is successful, the newly-added concept is returned. If *no-error-p* is set and *instance* (or *concept*) cannot be found, the two values **nil** and **:missing-instance** (or **:missing-concept**) are returned.

Remarks:

If *instance* is classifiable, **add-type** places it on the reclassify queue. However, the type of *instance* is not recomputed until the instance classifier is called (by **tellm** or **new-time-point**, for example).

Examples:

```
(add-type 'Joe 'Man) ⇒ |C|MAN
(add-type (fi Joe) (fc Married) :kb "USER-KB") ⇒ |C|MARRIED
(add-type 'Joe 'xxx :no-error-p t) ⇒ NIL :MISSING-CONCEPT
(new-time-point)
(pi Joe) ⟶ (TELL (:ABOUT JOE MARRIED MAN))
```

See Also:

get-types (p. 141)
new-time-point (p. 205)
remove-type (p. 235)
tell (p. 275)

add-value *Function*

Purpose:

The **add-value** function adds a value to a given role on a given instance.

Syntax:

add-value *instance role value* `&key` *kb no-error-p*

Arguments:

The *instance* argument is an instance or an instance identifier.

The *role* argument is a relation, or the name of a relation.

The *value* argument is a value to be added to the fillers of *role*. If *value* is a list, the list is interpreted as a single role filler. If *value* is a symbol, the value is coerced to an instance unless the the range of *role* indicates that it should be interpreted as a constant.

The *kb* argument is the knowledge base, or the symbol or string that names the knowledge base, where the lookup of *instance* and *role* begins. This defaults to the current knowledge base.

If the *no-error-p* argument is **t**, then error messages are suppressed.

Value:

If the update is successful, the newly added value is returned. If it fails and *no-error-p* is set, then: (1) if *role* is single-valued and already has a filler, the two values **nil** and **:too-many-fillers** are returned, (2) if the type of *value* conflicts with the range of *role*, **nil** and **:type-clash** are returned, (3) if *instance* (or *role*) cannot be found, **nil** and **:missing-instance** (or **:missing-role**) are returned.

Examples:

```
(defrelation rr)
(add-value 'Joe 'rr 'Fred) ⇒ |I|FRED
(add-value (fi Joe) (fr rr) (fi Sue) :kb "USER-KB") ⇒ |I|SUE
(add-value 'Joe 'rr '(3 4 5)) ⇒ (3 4 5)
(get-values 'Joe 'rr) ⇒ (|I|FRED |I|SUE (3 4 5))
(defrelation ss :range Symbol)
(add-value 'Joe 'ss 'Fred) ⇒ FRED
(defrelation r :characteristics :single-valued)
(add-value 'Joe 'r 'Fred) ⇒ |I|FRED
```

```
(add-value 'Joe 'r 'Sue :no-error-p t) ⇒ NIL :TOO-MANY-FILLERS
(get-values 'Joe 'r) ⇒ (|I|FRED)
```

See Also:

add-type (p. 17)
remove-value (p. 236)
set-values (p. 262)
tell (p. 275)
get-values (p. 144)

ask *Macro*

Purpose:

The **ask** macro is used to determine whether a proposition is true with respect to the current state of the knowledge base.

Syntax:

`ask` *query* `&key` *kb 3-valued-p*

Arguments:

The *query* argument is an arbitrary expression in the Loom query language (see **Remarks** below). This language has the expressive power of the first-order predicate calculus.

The *kb* argument is the name of the knowledge base in which *query* is to be compiled or evaluated.

If the *3-valued-p* argument is **t**, the query returns **:true**, **:false**, or **:unknown**.

Value:

Normally, the **ask** macro returns **t** if the proposition is provably true, and **nil** otherwise. If the *3-valued-p* option is selected, **ask** returns **:true** if the proposition is provably true, **:false** if it is provably false, and **:unknown** otherwise. Whether a given proposition is provably true (or false) may depend on whether open-world or closed-world semantics are currently assumed.

Remarks:

The query expressions in the *query* argument above have the following syntax:

```
query-expr ::=
    ( {:AND | :OR} query-expr+ ) |
    ( {:NOT | :FAIL} query-expr ) |
    ( :IMPLIES query-expr query-expr ) |
    ( {:FOR-SOME | :FOR-ALL} ( ?Var+ ) query-expr ) |
    ( :COLLECT ( ?Var ) query-expr ) |
    ( concept instance ) |
    ( relation instance+ value ) |
    ( :SAME-AS instance instance ) |
    ( :SUBSET instance instance ) |
    ( :PREDCALL LispPredicate value+ ) |
    ( :ABOUT instance about-clause* ) ;
```

Each *about-term* in a query **:about** clause has the form:

```
about-clause ::=
    concept |
    ( concept ) |
    ( relation value ) |
    ( :FILLED-BY relation value+ ) |
    ( {:AT-LEAST | :AT-MOST | :EXACTLY} Integer relation ) |
    ( {:ALL | :SOME | :THE} relation concept ) ;
```

The query-expression operators have the following semantics:

- If Q = (**:and** Q1...Qn), Q is true if Q1...Qn are all true.
- If Q = (**:or** Q1...Qn), Q is true if at least one of Q1...Qn is true.
- If Q = (**:not** Q1), Q is true if Loom can prove that Q1 cannot be true.
- If Q = (**:fail** Q1), Q is true if Loom cannot prove that Q1 is true.
- If Q = (**:for-some** (?X1...?Xn) Q1), Q is true if there are bindings for ?X1...?Xn that satisfy Q1.
- If Q = (**:for-all** (?X1...?Xn) (**:implies** Q1 Q2)), Q is true if all bindings of ?X1...?Xn that satisfy Q1 necessarily satisfy Q2.
- If Q = (C I), Q is true if I is an instance of concept C.
- If Q = (R I V), Q is true if V is a filler of role R on instance I.
- If Q = (R I1...In), Q is true if the tuple <I1...In> satisfies n-ary relation R.
- If Q = (**:same-as** V1 V2), Q is true if values V1 and V2 are equivalent.
- If Q = (**:subset** V1 V2), Q is true if V1 and V2 evaluate to sets of values, and V1 is a subset of V2.
- If Q = (**:predcall** P V), Q is true if application of Lisp predicate P to value V returns a value other than **nil**.
- If Q = (**:about** I C) or (**:about** I (C)), Q is true if I is an instance of concept C.
- If Q = (**:about** I (R V)), Q is true if the filler of role R on instance I is V.
- If Q = (**:about** I (**:filled-by** R V1...Vn)), Q is true if every value Vi is a filler of role R on instance I.
- If Q = (**:about** I (**:at-least** K R)), Q is true if instance I has at least K fillers of role R. The **:at-most** and **:exactly** operators have corresponding semantics.
- If Q = (**:about** I (**:all** R C)), Q is true if all fillers of role R on instance I are instances of concept C. The **:some** and **:the** operators have corresponding semantics.

A *concept*, *relation*, or *instance* may be either a symbol that names a Loom object, or a variable beginning with the character **?**. An *instance* may also be a constant or a formula, where a formula is a list of the form (*relation instance*).

Any **?**-variable not bound in a **:for-some** or **:for-all** clause is assumed to be bound externally. The value of an external **?**-variable should be a Loom object, i.e., a concept, relation, or instance, rather than the name of such an object.

The **:filled-by-list** operator cannot be used in the **:about** clause of a query. The Loom Grammar at the end of this manual is too general in this respect.

Concepts and relations referenced by name in a query must be defined at the time the query is compiled. Instances referenced by identifier must exist at the time the query is executed.

Examples:

```
(ask (Artist Joe)) ⇒ T
(ask (age Joe 40) :kb user-kb) ⇒ NIL
(ask (child Joe Fred) :3-valued-p t) ⇒ :UNKNOWN
(ask (max (age (child Joe)) 13))
(ask (distance Paris Rome 1000))
(ask (:and (Woman Boxer) (Senator Boxer)))
(ask (:not (Dog Duke)))
(ask (:fail (:or (French Jean) (resides Jean France))))
(ask (:for-some ?y (:and (author Joe ?y) (Best-Seller ?y))))
(ask (:for-some ?y (:and (age (child Joe) ?y) (:predcall #'oddp ?y))))
(ask (:for-all ?y (:implies (author Joe ?y) (Best-Seller ?y))))
(ask (:same-as (home Joe) (office Joe)))
(ask (:subset (customer Fred) (customer Joe)))
(ask (:about Joe (:at-least 2 child) (:exactly 0 son)))
(ask (:about Joe (:all child Female) (:some child Teen-Ager)))
(ask (:about Joe (:filled-by daughter Mary Sue)))
(setq ?C (fc Artist) ?R (fr friend) ?V (fi Fred) ?I (fi Joe))
(ask (:and (?C ?I) (?R ?I ?V) (:about ?I (:all ?R ?C))))
```

See Also:

defmethod (p. 61)
defproduction (p. 64)
query (p. 227)
retrieve (p. 246)
tell (p. 275)
why (p. 296)

binary-tuple *Relation*

Remarks:

The relation **binary-tuple** is at the top of the binary relation hierarchy, that is, it subsumes all other relations having arity two.

Examples:

```
(defrelation age)
(ask (subconcepts binary-tuple age)) ⇒ T
```

See Also:

n-ary-tuple (p. 202)
Relation (p. 229)
subconcepts (p. 269)

call-use-loom *Function*

Purpose:

The **call-use-loom** function imports Loom symbols into a specified package and optionally creates a new knowledge base that is associated with that package.

Syntax:

call-use-loom *packageName* `&key` *dont-create-knowledge-base-p knowledge-base-name path-name loom-imports*

Arguments:

The *packageName* argument is a symbol or string that identifies the Lisp package which will use Loom.

If the *dont-create-knowledge-base-p* argument is **t**, **call-use-loom** does not create a new knowledge base. Otherwise, it defines a knowledge base named *packageName*-KB.

The *knowledge-base-name* argument is a symbol or string to be used as the name of the new knowledge base. It overrides the default name, *packageName*-KB.

The *path-name* argument is a string specifying the file to be used for saving and restoring the knowledge base. The pathname may be supplied here, or it may be passed directly to **save-kb**.

The *loom-imports* argument is a string, or a list of strings, identifying Loom symbols which are to be shadowing-imported into *packageName*. These symbols override their counterparts in the **common-lisp** package, or in other packages inherited by *packageName*.

Value:

This function return **nil**.

Remarks:

In contrast to **use-loom**, which should only be called at top-level, the **call-use-loom** function can be invoked from within other functions.

The variable "LOOM::***shadow-import-from-loom***" contains symbols (such as **defmethod**, **trace**, and **pi**) which are automatically shadowing-imported into *packageName* in addition to any symbols specified by *loom-imports*.

Examples:

```
(call-use-loom "USER") ⇒ NIL
(call-use-loom 'test :dont-create-knowledge-base-p t
                     :path-name "test-kb.lisp") ⇒ NIL
(call-use-loom "FOO" :knowledge-base-name "FOO-KB"
                     :loom-imports '("ASK" "TELL")) ⇒ NIL
```

See Also:

use-loom (p. 291)

change-kb *Function*

Purpose:

The **change-kb** function resets the current knowledge base.

Syntax:

change-kb *knowledgeBase* `&key` *no-checking-p*

Arguments:

The *knowledgeBase* argument is a knowledge base, or a symbol or string that names a knowledge base. This function operates in a package-independent manner, so it is not necessary to specify the package in which a knowledge base name resides.

If the *no-checking-p* argument is **t**, *knowledgeBase* is assumed to be an object and no error checking is performed.

Value:

The **change-kb** function returns the knowledge base corresponding to *knowledgeBase*.

Remarks:

In contrast to **in-kb**, which should only be called at top-level, the **change-kb** function can be invoked from within other functions.

Examples:

```
(change-kb "USER-KB") ⇒ |K|USER-KB
(change-kb 'bottom-kb) ⇒ |K|BOTTOM-KB
(change-kb (find-kb 'user-kb) :no-checking-p t) ⇒ |K|USER-KB
```

See Also:

current-kb (p. 45)
defkb (p. 59)
in-kb (p. 149)
list-knowledge-bases (p. 174)

change-object *Function*

Purpose:

The **change-object** function changes a CLOS instance from one class to another and revises the indexing on the corresponding concepts.

Syntax:

change-object *instance class*

Arguments:

The *instance* argument is a CLOS instance.

The *class* argument is a CLOS class, or the name of a CLOS class, which shadows a Loom concept.

Value:

The **change-object** function returns *instance*.

Examples:

```
(setq ?i (make-object 'Single)) ⇒ #<Single #X20FA286>
(retrieve ?x (Single ?x)) ⇒ (#<Single #X20FA286>)
(change-object ?i 'Married) ⇒ (#<Married #X20FA286>)
(retrieve ?x (Single ?x)) ⇒ NIL
(retrieve ?x (Married ?x)) ⇒ (#<Married #X20FA286>)
(change-object ?i (find-class 'Single)) ⇒ (#<Single #X20FA286>)
(ask (Single ?i)) ⇒ T
```

See Also:

make-object (p. 189)

Character *Concept*

Purpose:

The **Character** predicate tests whether a given object is a Lisp character.

Syntax:

Character *object*

Argument:

The *object* argument is any object.

Value:

The **Character** predicate returns **t** if *object* is a character.

Examples:

```
(ask (Character #\a)) ⇒ T
```

See Also:

Non-Loom-Thing (p. 207)
Thing (p. 281)

clear-kb *Macro*

Purpose:

The **clear-kb** macro deletes all objects in specified partitions of a knowledge base and its descendants.

Syntax:

clear-kb `&optional` *knowledgeBase* `&key` *partitions*

Arguments:

The *knowledgeBase* argument is the knowledge base to be cleared, or a symbol or string which names that knowledge base. It defaults to the current knowledge base.

The *partitions* argument is a keyword or list of keywords specifying the types of objects to be deleted. Available partitions are **:instances**, **:concepts**, **:relations**, and **:behaviors**. If this argument is not supplied, all partitions are used.

Value:

The cleared knowledge base is returned.

Examples:

```
(clear-kb) ⇒ |K|USER-KB
(clear-kb "USER-KB" :partitions (:concepts :relations)) ⇒ |K|USER-KB
```

See Also:

defkb (p. 59)
delete-concept (p. 75)
destroy (p. 78)
initialize-instances (p. 152)
initialize-network (p. 153)
save-kb (p. 252)

compute-conjunction-concept *Function*

Purpose:

The **compute-conjunction-concept** function creates and classifies a concept that is the conjunction of a specified set of currently-existing classified concepts or relations.

Syntax:

compute-conjunction-concept *concepts*

Arguments:

The *concepts* argument is a list of classified concepts or relations.

Value:

This function returns a new, classified concept that represents the conjunction of *concepts*. The new concept has a system-generated name.

Examples:

`(compute-conjunction-concept (list (fc A) (fc B) (fc C)))` ⇒ `|C|A&B&C`

See Also:

define-concept (p. 57)
define-relation (p. 58)

compute-value-restriction *Function*

Purpose:

The **compute-value-restriction** function computes the value restriction of a given concept on a given relation.

Syntax:

compute-value-restriction *concept relation* `&key` *return-a-list-p*

Arguments:

The *concept* argument is a concept.

The *relation* argument is a relation.

If the *return-a-list-p* argument is `t`, then a list of the concepts which constitute the value restriction will be returned, rather than a single concept that represents the conjunction of those concepts.

Value:

The value restriction of *concept* is the type of all fillers of the *relation* role in instances of *concept*. Depending on *return-a-list-p*, **compute-value-restriction** returns either a list of the concepts which make up this type, or a concept which is the unification of all the concepts in the type.

Examples:

```
(defconcept A :implies (:and (:all r B) (:all r C)))
(defconcept C :implies D)
(defrelation r :range E)
(compute-value-restriction (fc A) (fr r)) ⇒ |C|B&C&D&E
(compute-value-restriction (fc A) (fr r) :return-a-list-p t) ⇒
    (|C|B |C|C |C|D |C|E)
```

See Also:

restriction--value-restriction (p. 245)
Value-Restriction-P (p. 294)

Concept *Concept*

Purpose:

The **Concept** predicate tests whether a given object is a Loom concept.

Syntax:

Concept *object*

Argument:

The *object* argument is any object.

Value:

The **Concept** predicate returns **t** if *object* is a unary concept.

Examples:

```
(ask (Concept Person)) ⇒ T
(retrieve ?x (Concept ?x)) ⇒ (|C|PERSON |C|THING |C|CONCEPT ...)
```

See Also:

concept-p (p. 33)
defconcept (p. 49)
Loom-Thing (p. 188)
Relation (p. 229)
Thing (p. 281)

concept-p *Function*

Purpose:

The **concept-p** function tests whether a given object is a Loom concept.

Syntax:

concept-p *object*

Arguments:

The *object* argument is any object.

Value:

The **concept-p** function returns **t** if *object* is a unary concept.

Examples:

```
(concept-p (fc Person)) ⇒ T
(concept-p (fc Knowledge-Base)) ⇒ T
```

See Also:

Concept (p. 32)
defconcept (p. 49)
find-concept (p. 96)
get-concept (p. 116)
instance-p (p. 155)
Loom-Thing (p. 188)
meta-concept-p (p. 197)
relation-p (p. 230)

concept--characteristics *Relation*

Purpose:

The **concept--characteristics** relation returns the characteristics which mark a particular concept.

Syntax:

concept--characteristics *concept characteristic*

Domain:

The **concept** argument is the concept or relation whose characteristics are to be found.

Range:

The **characteristic** argument is a keyword.

Examples:

```
(retrieve ?x (concept--characteristics Thing ?x)) ⇒ (:CLOS-CLASS)
(retrieve ?x (concept--characteristics binary-tuple ?x)) ⇒
    (:CLOSED-WORLD :SYMMETRIC)
(defrelation spouse :characteristics (:single-valued :closed-world))
(ask (concept--characteristics spouse :single-valued)) ⇒ T
```

See Also:

defconcept (p. 49)
defrelation (p. 66)

concept--instances *Relation*

Purpose:

The **concept--instances** relation finds all instances of a given concept, or all members of a given set.

Syntax:

concept--instances *concept instance*

Domain:

The *concept* argument is the concept whose instances are to be found, or the set concept whose members are to be found.

Range:

The *instance* argument is an instance or a symbolic set member.

Remarks:

The instances of *concept* are computed from its local instances and the local instances of all concepts below it.

Examples:

```
(tellm (Man Joe) (Man Fred))
(retrieve ?x (concept--instances Man ?x)) ⇒ (|I|JOE |I|FRED)
(defset Primary-Color :is (:one-of 'Red 'Green 'Blue))
(ask (concept--instances Primary-Color 'Red)) ⇒ T
```

See Also:

defset (p. 72)
get-instances (p. 119)
instance--concepts (p. 158)
isa-p (p. 162)
test-type-p (p. 280)

concept--name *Relation*

Purpose:

The **concept--name** relation finds the name of a given concept.

Syntax:

concept--name *concept symbol*

Domain:

The *concept* argument is the unary concept whose name is to be found.

Range:

The **symbol** argument is the symbol which names *concept*.

Examples:

```
(setq ?c (define-concept nil))
(retrieve ?x (concept--name ?c ?x)) ⇒ (THING_3)
(ask (concept--name Thing 'Thing)) ⇒ T
```

See Also:

defconcept (p. 49)
define-concept (p. 57)
name--concept (p. 203)
object-name (p. 209)

concept--restrictions *Relation*

Purpose:

The **concept--restrictions** relation finds all the local and inherited restriction features associated with a particular concept. (See **defconcept** Remarks for a summary of the restriction constructs available in Loom.)

Syntax:

concept--restrictions *concept restriction*

Domain:

The *concept* argument is the concept whose restrictions are to be returned.

Range:

The *restriction* argument is a Loom restriction feature. This may be a strict or implied feature, that is, it may have been declared in the **:is** or the **:implies** clause of a concept definition.

Examples:

```
(defconcept B :is (:and (:at-least 1 r) (:all r A)))
(defconcept C :is (:and B (:filled-by r Joe)))
(retrieve ?x (concept--restrictions C ?x)) ⇒
    (|FTR|(:all R A) |FTR|(:at-least 1 R A) |FTR|(:filled-by R (|I|JOE)))
```

See Also:

get-role (p. 128)
get-roles (p. 135)
relation--restrictions (p. 234)
restriction--concepts (p. 241)
restriction--relation (p. 244)
Restriction-P (p. 240)

Cons-Or-Null *Concept*

Purpose:

The **Cons-or-Null** predicate tests whether a given object is a Lisp cons or **nil**.

Syntax:

Cons-Or-Null *object*

Argument:

The *object* argument is any object.

Value:

The **Cons-Or-Null** predicate returns **t** if *object* is a cons or **nil**.

Examples:

```
(ask (Cons-Or-Null '(1 2 3))) ⇒ T
(ask (Cons-Or-Null nil)) ⇒ T
```

See Also:

Constant (p. 39)
Non-Loom-Thing (p. 207)

Constant *Concept*

Purpose:

The **Constant** predicate tests whether a given object is a Loom constant.

Syntax:

Constant *object*

Argument:

The *object* argument is any object.

Value:

The **Constant** predicate returns **t** if *object* is a number, character, string, symbol, or list.

Examples:

```
(ask (Constant 98.6)) ⇒ T
(ask (Constant #) ⇒ T
(ask (Constant "abc")) ⇒ T
(ask (Constant 'foo)) ⇒ T
(ask (Constant '(a b c))) ⇒ T
(ask (Constant ())) ⇒ T
(ask (Constant Thing)) ⇒ NIL
```

See Also:

Character (p. 28)
Cons-Or-Null (p. 38)
Integer (p. 161)
Non-Loom-Thing (p. 207)
Number (p. 208)
String (p. 267)
Symbol (p. 274)

copy-instance *Function*

Purpose:

The **copy-instance** function creates a new (unclassified) Loom instance that contains assertions equivalent to those in an existing Loom instance.

Syntax:

copy-instance *instance* `&key` *identifier kb add-suffix-p*

Arguments:

The *instance* argument is the instance to be copied.

The *identifier* argument is a symbol or string which identifies the newly-created copy. If this argument is **nil**, the copy is given a unique system-generated identifier.

If the *add-suffix-p* argument is **t**, *identifier* is suffixed with a unique integer. If *identifier* is **nil**, this is a no-op.

The *kb* argument is the knowledge base in which the copy will be interned. This defaults to the current knowledge base.

Value:

The **copy-instance** function returns the newly-created copy of *instance*.

Remarks:

The copy does not actually assume the type of the original instance until the instance classifier is run (by **tellm** or **new-time-point**, for example).

If *identifier* is the same as the identifier of an existing instance, and *add-suffix-p* is **nil**, then Loom asks whether the user really wants to overwrite the existing instance, causing the older instance to be partially uninterned.

Examples:

```
(copy-instance (fi Joe)) ⇒ |I|PERSON-34
(copy-instance (fi Joe) :identifier "Moe" :kb kb) ⇒ |I|MOE
(copy-instance (fi Moe) :identifier 'Moe :add-suffix-p t) ⇒ |I|MOE-35
```

See Also:

create (p. 42)
new-time-point (p. 205)
tell (p. 275)

count *Relation*

Purpose:

The **count** relation counts the number of individuals in a collection.

Syntax:

count *setOfObjects number*

Domain:

The *setOfObjects* argument is a list.

Range:

The *number* argument is the result of the arithmetic operation.

Examples:

```
(tellm (:about Joe (:filled-by child Mary Sue Pat)))
(retrieve ?x (count (child Joe) ?x)) ⇒ (3)
(ask (> (count (child Joe)) 2)) ⇒ T
```

See Also:

max (p. 192)
min (p. 198)
sum (p. 272)

create *Function*
createm *Macro*

Purpose:

The **create** function creates a new Loom or CLOS instance of a specified type.

The **createm** macro creates a new instance, and then calls the Loom instance classifier to compute the type of the instance (and of any other instances currently on the classify queues).

Syntax:

create *identifier concept* `&key` *kb add-suffix-p clos-instance-p*
createm *identifier concept* `&key` *kb add-suffix-p clos-instance-p*

Arguments:

The *identifier* argument is a symbol or string which identifies the new instance. If the instance is a Loom instance, or inherits the mixin **INSTANCE-IN-KNOWLEDGE-BASE**, and its *identifier* is **nil**, it is given a unique system-generated identifier.

The *concept* argument is the concept, or the name of the concept, to be used as the creation type of the new instance. When no *concept* is supplied, the creation type is **Thing**.

The *kb* argument is the knowledge base in which the new instance will be interned. This defaults to the current knowledge base.

If the *add-suffix-p* argument is **t**, *identifier* is suffixed with a unique integer. If *identifier* is **nil**, this is a no-op.

If the *clos-instance-p* argument is **t**, or if the global feature **:prefer-clos-instances** is set, a CLOS instance (see **make-object**) is created instead of a Loom instance. If there is no existing CLOS class corresponding to *concept*, Loom attempts to create the needed class. If this attempt fails, or if *concept* is **nil**, a standard Loom instance is created, and a warning is issued unless the value of *clos-instance-p* is **:no-warning**.

Value:

These functions return the newly-created instance.

Remarks:

The new instance does not actually assume its creation type until a **createm**, or some other function such as **tellm** or **new-time-point**, causes the instance classifier to be run.

If *identifier* is the same as the identifier of an existing instance, Loom asks whether the user wants to clobber the old instance (and possibly compromise the integrity of the database) or ignore the request for a new instance.

Examples:

```
(createm 'Joe 'A) ⇒ |I|JOE
(create 'Fred nil) ⇒ |I|FRED
(createm nil (fc A) :kb (find-kb "USER-KB")) ⇒ |I|A-13
(create 'Sue 'A :clos-instance-p t) ⇒ |i|SUE
(create nil 'A :clos-instance-p t) ⇒ #<A #X20FDDFE>
(createm 'Fred nil :add-suffix-p t) ⇒ |I|FRED-14
(create nil nil :clos-instance-p t) ⇒ |I|THING-15
```

See Also:

copy-instance (p. 40)
destroy (p. 78)
find-or-create-instance (p. 104)
make-object (p. 189)
new-time-point (p. 205)
tell (p. 275)

create-concept *Function*

Purpose:

The **create-concept** function creates a new concept or relation.

Syntax:

create-concept *name type kb*

Arguments:

The *name* argument is the symbol which names the new concept. If this is null, the concept is not interned in any knowledge base.

The *type* argument is either **:concept** or **:relation**.

The *kb* argument is the knowledge base in which the new concept is to be interned. This defaults to the current knowledge base.

Value:

The **create-concept** function returns the new concept.

Remarks:

The newly-created concept or relation is empty. It is not classified or queued for classification.

Examples:

```
(create-concept 'Box :concept (find-kb "USER-KB")) ⇒ |C|BOX
(create-concept 'weight :relation nil) ⇒ |R|WEIGHT
(create-concept nil :concept nil) ⇒ |C|:UNNAMED
```

See Also:

defconcept (p. 49)
define-concept (p. 57)
define-relation (p. 58)
defrelation (p. 66)

current-kb *Function*

Purpose:

The **current-kb** function gets the current knowledge-base.

Syntax:

current-kb

Arguments:

This function takes no arguments.

Value:

The **current-kb** function returns the current knowledge base.

Examples:

```
(current-kb) ⇒ |K|USER-KB
```

See Also:

change-kb (p. 26)
defkb (p. 59)
list-kb (p. 172)
list-knowledge-bases (p. 174)

defaction *Macro*

Purpose:

The **defaction** macro defines or redefines an action. An action specifies a generic operation (analagous to a CLOS generic function) that is implemented by one or more Loom methods.

Syntax:

defaction *name parameters* `&key` *filters missing-methods*

Arguments:

The *name* argument is a non-**nil** symbol. The action has the same name as the methods that implement it. It cannot have the same name as an existing production.

The *parameters* argument contains zero or more formal parameters. Keyword parameters are supported, but optional and rest parameters are not. The action must have the same number of required and keyword parameters as the methods that implement it.

The *filters* argument is a keyword, or list of keywords, where each keyword identifies a filter. This series of filters specifies the strategy to be used for choosing among methods for the action. Candidate methods pass through each filter in the sequence, so that the number of surviving candidates is monotonically non-increasing. The filtering process continues as long as more than one method remains. Actions may use the following filters:

- The **:overrides** filter eliminates methods which have been declared to be less preferable than some other candidate method (see **defmethod**).
- The **:most-specific** filter eliminates any method whose situation pattern is specialized by some other candidate method's pattern.
- The **:last-one** filter selects the most-recently defined of the surviving candidate methods.
- The **:select-one** filter performs a pseudo-random selection of one of the candidate methods.
- The **:select-all** filter passes all of the surviving methods.
- The **:warning** filter passes a single surviving candidate method, or lists the surviving methods and warns that it cannot choose among them.
- The **:error** filter passes a single surviving candidate method, or lists the surviving methods and breaks.

If *filters* is not supplied, it defaults to **(:most-specific :last-one)**.

The *missing-method* argument may be either **:no-op**, **:warning**, or **:error**. This argument specifies the action to be taken if no applicable method can be found. It defaults to **:warning**.

Value:

The defined or redefined action is returned.

Remarks:

If a method is defined before the corresponding action, Loom automatically creates that action.

Examples:

```
(defaction test (x)) ⇒ |ACTION|TEST
(defaction pack (?obj &key ?fragile-p)) ⇒ |ACTION|PACK
(defaction move (?obj ?to)
    :filters (:overrides :most-specific :last-one)
    :missing-method :no-op) ⇒ |ACTION|MOVE
```

See Also:

defmethod (p. 61)
defproduction (p. 64)
delete-all-methods (p. 74)
list-methods (p. 176)
print-methods (p. 226)

default *Macro*

Purpose:

The **default** macro specifies that a default implication holds between two concepts. This macro allows new defaults to be incrementally added to previously-defined concepts.

Syntax:

default *antecedent consequent*

Arguments:

The *antecedent* and *consequent* arguments are concept-forming expressions (see **defconcept** Remarks).

Value:

The antecedent concept is returned.

Examples:

```
(default Quaker Pacifist) ⇒ |C|QUAKER
(default (:satisfies (?x) (friend Joe ?x))
         Famous) ⇒ |C|THING_1
(default (:filled-by speaks 'Portuguese)
         (:or Portuguese Brazilian)) ⇒ |C|THING_2
```

See Also:

defconcept (p. 49)
get-role-default-values (p. 130)
implies (p. 148)

defconcept *Macro*

Purpose:

The **defconcept** macro defines or redefines a concept.

Syntax:

defconcept *name* `&optional` *documentation*
`&key` *is is-primitive implies defaults partitions exhaustive-partitions in-partition predicate function roles indices keys mixin-classes mixin-slots annotations identifier kb characteristics*

Arguments:

The *name* argument is a symbol. If *name* is null, the concept is given a system-generated name.

The *documentation* argument is a string which attaches a comment to the concept being defined. This string can be retrieved by calling the Lisp **documentation** function with **type** as the doc-type argument.

The *is* and *is-primitive* arguments are concept-forming expressions (see Remarks below) that define the concept—that is, they specify a condition which is both necessary and sufficient for membership in the concept. These arguments are identical except that *is-primitive* introduces primitiveness into the definition. If neither argument is provided, then the concept is assumed to be a primitive specialization of the built-in concept **Thing**.

The *implies* and *defaults* arguments are concept-forming expressions (see Remarks below) that attach a rule to the concept—that is, they specify a condition which is implied by membership in the concept. The *implies* and *defaults* arguments introduce strict and default implications, respectively. Strict implications are always applied, while default implications are only applied when their consequents are consistent with the current state of the knowledge base.

The *partitions* and *exhaustive-partitions* arguments contain a symbol or list of symbols representing the partitions or exhaustive partitions owned by the concept being defined. Each partition P owned by concept C contains a set of subconcepts of C. If P is non-exhaustive, then an instance of C may not be an instance of more than one member of P. If P is exhaustive, then an instance of C must be an instance of exactly one member of P.

The *in-partition* argument states that the concept being defined is a member of the specified partition. The concept should directly specialize the owner of the partition.

The *predicate* argument consists of a one-variable lambda list followed by a body of Lisp expressions. With the variable bound to an object, the Lisp expressions can be evaluated to determine whether the object is a member of the concept being defined. This membership test supercedes the definition given in the *is* or *is-primitive* argument.

The *function* argument consists of an empty lambda list followed by a body of Lisp expressions that can be evaluated to generate the members of the concept being defined. This function supercedes the definition given in the *is* or *is-primitive* argument.

The *roles* argument is a relation name, or a list of relation names. Loom uses *roles* to generate slots on the CLOS class corresponding to the concept being defined. If any member of *roles* has not been defined, a multiple-valued binary relation is automatically created. (Typically, this relation will subsequently be redefined by the user.)

The *indices* argument is a list, each member of which is either a relation name or a list of relation names. For each entry in *indices*, Loom creates a hash index that accelerates retrieval based on the values of the roles specified by that entry. The hash index maps from a role filler to one or more instances. All roles in *indices* must be single-valued. The **get-matching-instances** function takes advatage of the indices, as does **query** for simple queries involving only types and role fillers. Currently, however, **retrieve** does not use the indices.

The *keys* argument has the same form, and generally the same function, as the *indices* argument (see above). The principal difference is that the set of values of roles in a key must be unique for each instance in the corresponding concept. The specification of *keys* without corresponding *indices* does not provide accelerated access. Warnings are issued if (1) multiple instances are asserted to have the same value for a key, or (2) values are not supplied for all members of a key.

The *class-name* argument is the name to be given to the defined concept's shadowing CLOS class, if and when such a class is created. If this argument is not supplied, the new CLOS class will have the same name as the concept.

The *existing-class-name* argument is the name of an existing CLOS class which is to shadow the concept being defined, if such a class is needed for the creation of CLOS instances. Normally, the shadowing class is a newly-created CLOS class having the same name as the concept.

The *mixin-classes* argument is the name of a CLOS class, or a list of such names. A concept having mixin classes is automatically assigned the **:clos-class** characteristic. CLOS instances of such a concept inherit properties of the specified mixin classes as follows:

- The **INSTANCE-WITH-CONCEPTS** mixin allows new concepts to be added to the type of a CLOS instance.
- The **INSTANCE-WITH-DYNAMIC-SLOTS** mixin allows role values to be attached to a CLOS instance for roles other than those declared in the CLOS

class for the instance.

- The **INSTANCE-IN-MATCH-NETWORK** mixin causes CLOS instances to be entered into production rule memory so that applicable productions will fire when the state of the database changes. This mixin is a subclass of **INSTANCE-WITH-CONCEPTS**.
- The **INSTANCE-IN-KNOWLEDGE-BASE** mixin causes a CLOS instance to be interned into the instances partition of a Loom knowledge base.
- The **INSTANCE-WITH-INVERSES** mixin causes inverse links to be created in response to assertions of role fillers. Inverse links assist Loom in avoiding dangling references.
- The **INSTANCE-WITH-EVERYTHING** mixin adds the functionality of all the mixins listed above.

Users may provide their own mixins in addition to the above built-in classes.

The *mixin-slots* argument is a CLOS slot specifier, or a list of such specifiers. A concept having mixin slots is automatically assigned the **:clos-class** characteristic. The mixin-slots information is not used by Loom but instead is passed directly to the CLOS class definition facility. Thus, these slots don't necessarily correspond to any Loom relations. Loom assumes that the mixin-slots information has precedence over its own default behavior for constructing CLOS classes.

The *annotations* argument is a list of facts to be asserted about the concept being defined. Each element of the list is either a concept-name or an expression of the form (*RelationName instance*), where *instance* is either a constant or a symbol that identifies a Loom instance.

The *identifier* argument is the name under which the concept will be interned in the **:instances** partition of its knowledge base. This argument defaults to *name*.

The *kb* argument is a string or symbol that names the knowledge base in which the concept is to be interned. This defaults to the current knowledge base.

The *characteristics* argument is a keyword or list of keywords which specify various attributes of the concept. Concepts may have the following user-settable characteristics:

- The **:backward-chaining** characteristic indicates that instances do not classify under the concept during instance classification, and caching of subsumption information is not performed. Instead, subsumption relations are deduced in response to queries from either the user or Loom itself. Usually the system sets this characteristic.
- The **:clos-class** characteristic directs Loom to create a CLOS class corresponding to the concept being defined. The class is created the first time that **make-object**

is called to make an instance of the class, or when a method is defined for the class, or when **finalize-definitions** is called.

- The **:non-exported** characteristic prevents the name of the concept from being exported, thus overriding the **:export-names-p** mark on the concept's knowledge base.
- The **:closed-world** characteristic indicates that closed-world semantics are to apply to the concept (rather than the default open-world semantics). This allows Loom to non-monotonically conclude that an instance is not of a particular type.
- The **:monotonic** characteristic indicates that instances asserted to belong to the concept will always belong to that concept. Instances of monotonic concepts are easier for Loom to truth-maintain. Also, the **:monotonic** characteristic can cause derived inferences to have the force of assertions.
- The **:perfect** characteristic indicates that instances of the concept always have and always will belong to that concept. These instances must be asserted to belong to the concept at the time they are created. Perfect concepts are not truth-maintained, and the Loom production facility does not monitor them, except when they are first created.

Concepts may also have the following Loom-assigned characteristics:

- The **:system-defined** characteristic indicates that the concept has been created by Loom.
- The **:undefined** characteristic indicates that the concept has been referenced in a definition, but has not yet been defined itself.
- The **:incoherent** characteristic indicates that a conflict exists in the definition of the concept. For example, a concept that specializes two disjoint concepts would be marked **:incoherent**.

Value:

The defined or redefined concept is returned.

Remarks:

Negation is not yet supported in concept definitions.

There should only be one *is* or *is-primitive* keyword argument.

Concepts referenced in the *is* or *is-primitive* clause should not refer (directly or indirectly) to the concept being defined, since cycles are not permitted in definitions.

Currently, Loom automatically makes a concept backward-chaining if any of the following conditions hold:

- it is defined with the **:backward-chaining** characteristic;
- it has no primitiveness;
- it defines a set;
- it specializes any of the following built-in concepts: **Constant**, **Symbol**, **String**, **Character**, **Number**, **Integer**, **Cons-or-Null**;
- it is defined with a **:satisfies**, **:predicate**, or **:function**;
- recursively, its definition references a backward-chaining concept.

Currently, Loom automatically makes a concept closed-world if any of the following conditions hold:

- it is defined with the **:closed-world** or **:perfect** characteristic;
- it is primitive and the global **:closed-world** feature is set;
- it defines a symbolic set;
- recursively, all of its definitional referents are closed.

Currently, concepts with no primitiveness, and concepts defined with the **:predicate** option, are automatically marked **:backward-chaining**.

If a concept definition refers to a concept or relation that has not yet been defined, that concept or relation is immediately created by Loom.

In normal operation, a newly-defined concept is immediately classified, provided that all the concepts and relations it references have been classified.

The concept-forming expressions that appear in the *is*, *is-primitive*, *implies*, and *defaults* arguments above have the following syntax:

```
concept-expr ::=
    ConceptName |
    ( {:AND | :OR} concept-expr+ ) |
    ( :ONE-OF {Number+ | InstanceId+} ) |
    ( :THROUGH Number Number ) |
    ( {:AT-LEAST | :AT-MOST | :EXACTLY} Integer relation-expr ) |
    ( {:ALL | :SOME | :THE} relation-expr ConceptName ) |
    ( {:FILLED-BY | :NOT-FILLED-BY} relation-expr
                    {InstanceId | Constant}+ ) |
    ( {:SAME-AS | :SUBSET} relation-expr relation-expr ) |
    ( {< | > | <= | >= | = | /=} relation-expr
                    {relation-expr | Number} ) |
    ( :RELATES relation-expr relation-expr
                    {relation-expr | Constant} ) |
    ( :SATISFIES ( ?Var ) query-expr ) |
    set-expr ;
```

The concept-forming operators have the following semantics:

- If C = (**:and** C1...Cn), an instance of C is an instance of each of the concepts C1...Cn.
- If C = (**:or** C1...Cn), an instance of C is an instance of at least one of the concepts C1...Cn.
- If C = (**:at-least** K R), an instance of C has at least K fillers of the role R.
- If C = (**:at-most** K R), an instance of C has at most K fillers of the role R.
- If C = (**:exactly** K R), an instance of C has exactly K fillers of the role R.
- If C = (**:all** R C1), and I is an instance of C, all fillers of the role R on I are instances of concept C1.
- If C = (**:some** R C1), and I is an instance of C, at least one filler of the role R on I is an instance of concept C1.
- If C = (**:the** R C1), and I is an instance of C, the role R on I has exactly one filler, and that filler is an instance of concept C1.
- If C = (**:filled-by** R V1...Vn), and I is an instance of C, the role R on I contains all of the fillers V1...Vn.
- If C = (**:not-filled-by** R V1...Vn), and I is an instance of C, the role R on I does not contain any of the fillers V1...Vn.
- If C = (**:same-as** R1 R2), an instance of C has the same set of fillers for role R1 as for role R2.
- If C = (**:subset** R1 R2), and I is an instance of C, the fillers of role R1 on I are a subset of the fillers of role R2.
- If C = (< R1 R2), and I is an instance of C having single-valued roles R1 and R2, the filler of R1 is numerically less than the filler of R2. The other arithmetic operators have corresponding semantics.
- If C = (**:relates** R R1 R2), and I is an instance of C having single-valued roles R1 and R2, the relation R holds between the filler of R1 and the filler of R2.
- If C = (**:satisfies** (?X) Q), and I is an instance of C, the the Loom query Q is satisfied when ?X is bound to I.

The auxiliary-definition keyword arguments having the following semantics:

- If C = **:predicate** ((X) F1...Fn), and I is an instance of C, the Lisp forms F1...Fn return a non-nil value when they are evaluated with X bound to I.
- If C = **:function** (() F1...Fn), and I is an instance of C, I is a member of the values returned when the Lisp forms F1...Fn are evaluated.

Examples:

```
(defconcept nil "Loom assigns name") ⇒ |C|THING_3
(defconcept A) ⇒ |C|A
(defconcept B :is-primitive A)
(defconcept C :is (:and Adult Male Human))
(defconcept D :is (:or Dog Cat))
(defconcept E :is (:through 0 9))
(defconcept F :is (:one-of 7 11 13))
(defconcept G :is (:one-of Larry Moe Curly))
(defconcept H :is (:and A (:at-least 3 child) (:at-most 2 son)
                          (:exactly 1 pet)))
(defconcept I :is (:and A (:all child Happy) (:some son Teen-Ager)
                          (:the pet Dog)))
(defconcept J :is (:and A (:filled-by friend Bill Hillary)))
(defconcept K :is (:and A (:not-filled-by blood-type 'A-Pos)))
(defconcept L :is (:and A (:same-as wife best-friend)))
(defconcept M :is (:and A (:subset customer friend)))
(defconcept N :is (:and A (= length width) (> weight 100)))
(defconcept O :is (:and A (:relates divorced father mother)))
(defconcept P :is (:satisfies (?x) (:and (Dog ?x) (:fail (Purebred ?x)))))
(defconcept AA :implies (:and Expensive (:the chef French)))
(defconcept BB :defaults (:or (:filled-by class 'Senior) Grad-Student))
(defconcept CC :partitions (($PET$ (Dog Cat Bird Fish))))
(defconcept DD :exhaustive-partitions $PRIMATE$)
(defconcept EE :in-partition $PRIMATE$)
(defconcept FF :predicate ((n) (oddp n)))
(defconcept GG :function (() (loop for n from 0 to 9 collect n)))
(defconcept HH :roles (owner name breed age))
(defconcept II :indices (owner name breed) :keys (owner name))
(defconcept JJ :class-name ANIMAL)
(defconcept KK :existing-class-name INTEGER)
(defconcept LL :mixin-classes (INSTANCE-IN-KNOWLEDGE-BASE
                               INSTANCE-WITH-CONCEPTS))
(defconcept MM :annotations (Noun (plural "-es")))
(defconcept NN :identifier Age-as-Concept :kb user-kb)
(defconcept OO :characteristics (:closed-world clos-class))
```

See Also:

Concept (p. 32)
concept--characteristics (p. 34)
concept--name (p. 36)
concept--restrictions (p. 37)

concept-p (p. 33)
create-concept (p. 44)
default (p. 48)
define-concept (p. 57)
defrelation (p. 66)
defset (p. 72)
disjoint-concepts-p (p. 82)
finalize-definitions (p. 94)
implies (p. 148)
meta-concept-p (p. 197)
Primitive-P (p. 225)
User-Defined-P (p. 293)

define-concept *Function*

Purpose:

The **define-concept** function allows the user to procedurally define or redefine a Loom concept.

Syntax:

define-concept `&key` *name is is-primitive implies defaults partitions exhaustive-partitions in-partition predicate function mixin-classes mixin-slots annotations identifier kb characteristics*

Arguments:

The *name* argument is a symbol or string. If *name* is null, the concept is given a system-generated name.

All of the remaining **define-concept** arguments are Lisp expressions which evaluate to atoms or lists that are legal arguments to **defconcept**. Variables appearing in these expressions may evaluate to Loom objects or to the names of such objects.

Value:

This function returns the newly-defined (or redefined) concept.

Examples:

```
(define-concept) ⇒ |C|THING_1
(define-concept :name 'Person) ⇒ |C|PERSON
(let ((supers (list 'Man (fc Rich))))
  (define-concept :is `(:and ,@supers))) ⇒ |C|RICH&MAN
```

See Also:

compute-conjunction-concept (p. 30)
create-concept (p. 44)
defconcept (p. 49)
rename-concept (p. 238)
define-relation (p. 58)

define-relation *Function*

Purpose:

The **define-relation** function allows the user to procedurally define or redefine a Loom relation.

Syntax:

define-relation `&key` *name is is-primitive implies domain domains range arity predicate function inheritance-link inheritance-method annotations identifier kb characteristics*

Arguments:

The *name* argument is a symbol or string. If *name* is null, the relation is given a system-generated name.

All of the remaining **define-relation** arguments are Lisp expressions which evaluate to atoms or lists that are legal arguments to **defrelation**. Variables appearing in these expressions may evaluate to Loom objects or to the names of such objects.

Value:

These functions return the newly-defined (or redefined) relation.

Examples:

```
(define-relation) ⇒ |R|BINARY-TUPLE_2
(define-relation :name "son" :is '(:and child (:range Male))) ⇒ |R|SON
(let ((arity 3))
  (define-relation :name 'distance :arity arity)) ⇒ |R|DISTANCE
```

See Also:

compute-conjunction-concept (p. 30)
define-concept (p. 57)
defrelation (p. 66)
rename-concept (p. 238)

defkb *Macro*

Purpose:

The **defkb** macro creates a new knowledge base.

Syntax:

defkb *name parentKbs* `&key` *path-name package export-names-p monotonic-p*

Arguments:

The *name* argument is a symbol or string that names the new knowledge base.

The *parentKbs* argument is a symbol or string, or a list of symbols or strings, which name the immediate parents of the new knowledge base. If this argument is **nil**, it defaults to **upper-structure-kb**.

The *path-name* argument is a string which specifies the file to be used for saving and restoring the new knowledge base.

The *package* argument is a symbol or string that specifies the package into which a saved version of the knowledge base should be loaded. This package name is written into the file created by **save-kb**. It defaults to the current package.

If the *export-names-p* argument is **t**, Loom exports the names of all objects created in the new knowledge base. Names are exported from the package in which they reside, which may be different from the *package* argument or from the current package. By default, names are not exported.

If the *monotonic-p* argument is **t**, Loom assumes that assertions made about instances in this knowledge base will not be retracted. Operations that Loom cannot undo, such as the generation and merging of skolems, should only be performed in a monotonic knowledge base.

Value:

The newly created knowledge base is returned.

Remarks:

If an old knowledge base with the same name exists, **defkb** destroys that knowledge base, and its children recursively, before creating the new knowledge base.

The **upper-structure-kb** knowledge base is created by Loom and populated with built-in concepts and relations. All user knowledge bases are descendants of **upper-structure-kb**.

The **bottom-kb** knowledge base is created by Loom and is dynamically relinked so that it is always lower than any other knowledge base. Loom objects in all knowledge bases are visible from **bottom-kb**.

Unlike Lisp packages, inheritance between knowledge bases is transitive. From a given knowledge base, references can be made to objects in that knowledge base, in its parents, and in all of its ancestors.

Examples:

```
(defkb foo-kb nil) ⇒ |K|FOO-KB
(defkb foobar-kb (foo-kb bar-kb) :pathname "foobar-kb.lisp"
    :package USER :export-names-p t) ⇒ |K|FOOBAR-KB
```

See Also:

change-kb (p. 26)
clear-kb (p. 29)
current-kb (p. 45)
find-kb (p. 101)
find-knowledge-base-of-instance (p. 102)
in-kb (p. 149)
Knowledge-Base (p. 164)
list-kb (p. 172)
list-knowledge-bases (p. 174)
load-kb (p. 184)
save-kb (p. 252)
use-loom (p. 291)

defmethod *Macro*

Purpose:

The **defmethod** macro defines or redefines a Loom method. A Loom method is one of a set of operations that implements a Loom action.

Syntax:

defmethod *name parameters* `&key` *title situation with overrides response*

Arguments:

The *name* argument is a non-**nil** symbol. All methods which implement a given action have the same name as that action. Methods cannot have the same name as an existing production.

The *parameters* argument contains zero or more formal parameters. Keyword parameters are supported, but optional and rest parameters are not. Methods must have the same number of required and keyword parameters as the action that they implement.

The *title* argument is a string which titles the method.

The *situation* argument is the guard condition which is used to select candidate methods. This argument is a query expression (see **retrieve** Remarks), i.e., it is syntactically identical to the body of a **retrieve** statement. Any variables which appear in the *situation* but are not *parameters* must be introduced by a **:for-some** or **:for-all** operator. If the method's action uses the **:most-specific** filter, and there is more than one method whose *situation* pattern is satisfied when the action is invoked, then the method with the most specific *situation* pattern is selected.

The *with* argument specifies a condition which must be satisfied before *response* is performed. This argument is a query expression (see **retrieve** Remarks). If the expression contains free variables—that is, variables which are not *parameters* and are not bound by a **:for-some** or **:for-all**—then the *response* is performed once for each binding of these variables. This argument defaults to a condition which is always **t**.

The *overrides* argument is a title or list of titles of the methods which are overridden (see **defaction**) by the method being defined.

The *response* argument is a list of Lisp forms which are to be evaluated if the *with* test is satisfied when the method is invoked. Variables bound within the *situation* and *with* clauses are accessible from the *response*.

Value:

The defined or redefined Loom method is returned.

Remarks:

If the **defmethod** macro has no *response* argument and no *situation* argument, then it expands to **CLOS:defmethod** instead of **LOOM:defmethod**.

A method definition is taken to redefine an existing method having the same name if (1) both methods have the same *title*, or (2) neither method has a *title* but they have the same *situation*.

If a method is defined before the corresponding action, Loom automatically creates that action.

An existing method cannot be redefined to have a different number of parameters unless that method's action is explicitly redefined first.

Overridden methods must be defined before the method that overrides them.

Currently, concepts and relations referenced in the *situation* and *with* clauses of a method must be defined before the method itself is defined.

The **use-loom** and **call-use-loom** functions perform a shadowing-import of the symbol **LOOM:defmethod** into the using package.

Examples:

```
(defmethod test (x)
    :response ((run-test x))) ⇒ |METHOD|TEST-:UNTITLED
(defmethod move (?obj ?loc)
    :title "move box"
    :situation (Box ?obj)
    :response ((format t "Move box ~S to ~S~%" ?obj ?loc)
                   (move-box ?obj ?loc))) ⇒ |METHOD|MOVE-"move box"
(defmethod foo (?x &key ?z)
    :title "this method"
    :overrides ("other method")
    :situation (:and (A ?x) (:for-some (?i) (:and (r ?x ?i) (A ?i))))
    :with (s ?x ?y)
    :response ((bar ?x ?y ?z))) ⇒ |METHOD|FOO-"this method"
```

See Also:

ask (p. 20)
defaction (p. 46)
defproduction (p. 64)
delete-all-methods (p. 74)
delete-method (p. 76)
fail (p. 86)

defproduction *Macro*

Purpose:

The **defproduction** macro defines or redefines a production. A Loom production is a data-directed rule that causes an action to be performed whenever a particular kind of event is detected.

Syntax:

defproduction *name* `&key` *when perform schedule do priority*

Arguments:

The *name* argument is a non-nil symbol. Productions cannot have the same name as an existing action.

The *when* argument is the trigger condition for the production. It consists of either a production expression, or a conjunction of production expressions and query expressions (see **retrieve** Remarks). A production expression describes a transition (see Remarks below). It is a list containing the keyword **:detects**, **:undetects**, or **:changes**, followed by a literal or conjunction of literals, where a literal has the form (*ConceptName ?Var*) or (*RelationName ?Var ?Var*).

The *perform* and *schedule* arguments specify a task to be performed or scheduled when the production fires. The *perform* option causes the task to be executed as soon as the production fires, while the **schedule** option places it on a queue for later execution. The *perform* and *schedule* arguments are action expressions consisting of a Loom action name followed by one or more variables or constants. Any **?**-variables must have been bound in the *when* expression.

The *do* argument specifies a Lisp program to be run when the production fires. It consists of a list of Lisp forms containing **?**-variables bound in the *when* expression.

The *priority* argument indicates the queue on which a scheduled task is to be placed. It has the values **:high** or **:low**—the default being **:high**. This is a no-op unless a *schedule* argument is supplied.

Value:

The defined or redefined production is returned.

Remarks:

The **defproduction** macro must have a *when* argument. It must also have exactly one *perform*, *schedule*, or *do* argument.

A **:detects** production expression is true if its query became satisfied during the most recent state change. An **:undetects** expression is true if its query ceased to be satisfied during the most recent state change.. A **:changes** expression is true if the fillers of the specified role changed during the most recent state change.

Production rules cannot trigger on concepts or relation that have no primitiveness, or that are marked **:monotonic** or **:perfect**.

Currently, concepts and relations referenced in the *when* clause of a production must be defined before the production itself is defined.

Production firing results from an explicit or implicit call to **new-time-point**. All productions fire before any scheduled tasks are performed.

Examples:

```
(defproduction almost-full-container
    :when (:detects (:and (Container ?x)
                    (>= (pct-full ?x) 90)))
    :perform (move ?x)) ⇒ |PRODUCTION|ALMOST-FULL-CONTAINER
(defproduction P1
    :when (:and (:undetects (A ?x))
                (:for-some (?y) (:and (r ?x ?y) (A ?y))))
    :schedule (foo ?x 3)
    :priority :high) ⇒ |PRODUCTION|P1
(defproduction P2
    :when (:changes (r ?x))
    :do ((print (get-universal-time))
         (format t "The r of ~S is now ~S~%"
                          ?x (get-value ?x 'r)))) ⇒ |PRODUCTION|P2
```

See Also:

ask (p. 20)
defaction (p. 46)
defmethod (p. 61)
delete-production (p. 77)
find-production (p. 105)
get-production (p. 126)
list-productions (p. 177)
new-time-point (p. 205)
perform (p. 214)
schedule (p. 254)
trace-all (p. 284)
untrace-all (p. 290)

defrelation *Macro*

Purpose:

The **defrelation** macro defines or redefines a binary or n-ary relation.

Syntax:

defrelation *name* `&optional` *documentation*
`&key` *is is-primitive implies domain domains range arity predicate function inheritance-link inheritance-method annotations identifier kb characteristics*

Arguments:

The *name* argument is a symbol. If *name* is null, the relation is given a system-generated name.

The *documentation* argument is a string which attaches a comment to the relation being defined. This string can be retrieved by calling the Lisp **documentation** function with **variable** as the doc-type argument.

The *is* and *is-primitive* arguments are relation-forming expressions (see Remarks below) that define the relation—that is, they specify a condition that is both necessary and sufficient for a tuple to satisfy the relation. These arguments are identical except that *is-primitive* introduces primitiveness into the definition. If neither argument is provided, then the relation is assumed to be a primitive specialization of the built-in relation **Binary-Tuple** or **N-Ary-Tuple**.

The *implies* argument is a relation-forming expression (see Remarks below) that attaches a rule to the relation—that is, it specifies a condition which is necessary for a tuple to satisfy the relation.

The *domain* and *range* arguments are concept-forming expressions (see **defconcept** Remarks). Supplying these arguments is equivalent to using **:domain** and **:range** expressions in the *implies* argument.

The *domains* argument is only used in the definition of n-ary relations. It is a list of concept-forming expressions whose length is equal to one less than the arity of the relation being defined. Supplying this argument is equivalent to using a **:domains** expression in the *implies* argument.

The *arity* argument is an integer which specifies the arity of an n-ary relation. A relation with an arity of n has $n-1$ domains and one range.

The *predicate* argument consists of a two-variable lambda list followed by a body of Lisp expressions (more variables are required for n-ary relations). When the variables are bound to objects, the Lisp expressions can be evaluated to determine whether the relation being defined holds between those objects. This test supercedes the definition given in the *is* or *is-primitive* argument.

The *function* argument consists of a one-variable lambda list followed by a body of Lisp expressions (more variables are required for n-ary relations). When the variable is bound to an object I, the Lisp expressions can be evaluated to generate other objects which are linked to I by the relation being defined. This function supercedes the definition given in the *is* or *is-primitive* argument.

The *inheritance-link* argument is the name of an inter-concept link across which annotations are inherited. Typically, this link is the metarelation **direct-subconcepts**. For example, if relation R is declared to have the **direct-subconcepts** inheritance link, and concept B is a subconcept of A, then when (R A x) is asserted, Loom will infer (R B x).

The *inheritance-method* argument is either **:nearest-neighbor** or **:union**. The **:nearest-neighbor** option indicates that annotations are inherited across *inheritance-link* only from the nearest concept (e.g., the most-specific superconcept) which has such annotations. The **:union** option indicates that annotations are inherited transitively across the inheritance link.

The *annotations* argument is a list of facts to be asserted about the relation being defined. Each element of the list is either a concept-name or an expression of the form (*RelationName instance*), where *instance* is either a constant or a symbol that identifies a Loom instance.

The *identifier* argument is the name under which the relation will be interned in the **:instances** partition of its knowledge base. This argument defaults to *name*.

The *kb* argument is a string or symbol that names the knowledge base in which the relation is to be interned. This defaults to the current knowledge base.

The *characteristics* argument is a keyword or list of keywords which specify various characteristics of the relation. Relations may have the following user-settable characteristics:

- The **:single-valued** characteristic indicates that the relation, when used as a role, may have only one value. If two (or more) different instances are asserted to be fillers of a single-valued role, Loom automatically retracts all but the last assertion. This clipping behavior can be globally overridden by unsetting the **:clip-roles** feature (see **set-features**), and it can then be locally restored by setting the **:clip-roles** characteristic.
- The **:multiple-valued** characteristic is used with relations whose definition contains the **:function** option. It declares that the relation, when used as a role, will return more than one value. By default, **:function** relations are **:single-valued**, while ordinary relations are **:multiple-valued**.

- The **:clip-roles** characteristic is used with **:single-valued** relations. It causes clipping to be performed on such relations even though the **:clip-roles** feature has been unset globally.
- The **:symmetric** characteristic indicates that the relation is its own inverse. For example, if R is **:symmetric**, then $(R\ x\ y)$ implies $(R\ y\ x)$.
- The **:commutative** characteristic indicates that a relation of arity n holds regardless of the ordering of its first $n-1$ arguments. For example, if R is **:commutative**, then $(R\ x\ y\ z)$ implies $(R\ y\ x\ z)$.
- The **:read-only** characteristic indicates that the relation, when used as a role, cannot be asserted to. Such roles are not monitored by productions, and they cannot be used to generate (or filter) instances unless they are defined with the **:function** (or **:predicate**) option.
- The **:read-write** characteristic overrides the **:read-only** characteristic in roles that would otherwise inherit **:read-only** status from their supers or their inverses.
- The **:cache-computation** characteristic is used with binary relations whose definition contains the **:function** option. It causes computed values to be cached as ordinary role fillers, so that subsequent accesses do not require recomputation. The cached values are not truth-maintained. Retracting with **forget** *, or calling **set-value** with a null filler argument, clears the cache.
- The **:hash-on-domains** characteristic is used with n-ary relations. It indicates that the values satisfying the relation are to be stored as hashed tuples.
- The **:non-exported** characteristic prevents the name of the relation from being exported, thus overriding the **:export-names-p** mark on the relation's knowledge base.
- The **:open-world** characteristic indicates that open-world semantics are to apply to the relation, even if closed-world semantics prevail globally because the **:closed-world-roles** feature has been set. In Loom, relations have open-world semantics by default.
- The **:closed-world** characteristic indicates that closed-world semantics are to apply to the relation, even if open-world semantics are assumed globally (which is the default).
- The **:monotonic** characteristic indicates that when the relation, used as a role, assumes a certain value, it will always have that value. Monotonic relations are easier for Loom to truth-maintain. Also, the **:monotonic** characteristic can cause derived inferences to have the force of assertions.
- The **:perfect** characteristic indicates that if the relation, used as a role, is filled by certain instances, it always has and always will be filled by those instances. The instances must be asserted to fill the role at the time they are created. Perfect roles are not truth-maintained, and the Loom production facility does not monitor them.

Relations may also have the following Loom-assigned characteristics:

- The **:system-defined** characteristic indicates that the relation has been created by Loom.
- The **:undefined** characteristic indicates that the relation has been referenced in a definition, but has not yet been defined itself.

Value:

The defined or redefined relation is returned.

Remarks:

Negation and disjunction are not yet supported in relation definitions.

There should only be one *is* or *is-primitive* keyword argument.

Relations referenced in the *is* or *is-primitive* clause should not refer (directly or indirectly) to the relation being defined, since cycles are not permitted in definitions.

Currently, Loom automatically makes a relation closed-world if any of the following conditions hold:

- it is defined with the **:closed-world** characteristic;
- it is primitive (or defined as an inverse or composition), and it does not have the **:open-world** characteristic, and the global **:closed-world** feature is set;
- recursively, all of its definitional referents are closed.

If a relation definition refers to a concept or relation that has not yet been defined, that concept or relation is immediately created by Loom.

In normal operation, a newly-defined relation is immediately classified, provided that all the concepts and relations it references have been classified.

The relation-forming expressions that appear in the *is*, *is-primitive*, and *implies* arguments above have the following syntax:

```
relation-expr ::=
    RelationName |
    ( :AND relation-expr+ ) |
    ( {:DOMAIN | :RANGE} concept-expr ) |
    ( :DOMAINS concept-expr concept-expr+ ) |
    ( :INVERSE relation-expr ) |
    ( :COMPOSE relation-expr+ ) |
    ( :SATISFIES ( ?Var ?Var+ ) query-expr ) ;
```

The relation-forming operators have the following semantics:

- If R = (**:and** R1…Rn), and tuple t is an instance of R, then t is an instance of each of the relations R1…Rn.
- If R = (**:domain** C), and <I1,I2> is an instance of R, then I1 is an instance of C.
- If R = (**:range** C), and <I1,I2> is an instance of R, then I2 is an instance of C.
- If R = (**:domains** C1…Cn), and <I1,…,In> is an instance of R, then for i from 1 to $n-1$, Ii is an instance of Ci.
- If R = (**:inverse** RI), and <I1,I2> is an instance of R, then <I2,I1> is an instance of RI.
- If R = (**:compose** R1…Rn), and <I1,In> is an instance of R, then for i from 1 to $n-1$, there is an <Ii,Ii+1> that is an instance of Ri.
- If R = (**:satisfies** (?X1…?Xn) Q), and <I1,…,In> is an instance of R, then the Loom query Q is satisfied when the variables ?X1…?Xn are bound (in order) to I1…In.

The auxiliary-definition keyword arguments have the following semantics:

- If R = **:predicate** ((X1…Xn) F1…Fj), and <I1,…,In> is an instance of R, then the Lisp forms F1…Fj return a non-nil value when they are evaluated with X1…Xn bound (in order) to I1…In.
- If R = **:function** ((X1…Xn) F1…Fj), and <I1,…,In,In+1> is an instance of R, then In+1 is a member of the values returned when the Lisp forms F1…Fn are evaluated.

Examples:

```
(defrelation nil "Loom assigns name") ⇒ |R|BINARY-TUPLE_4
(defrelation r0) ⇒ |R|R0
(defrelation r1 :is-primitive r0)
(defrelation r2 :is (:and relative friend))
(defrelation r3 :is (:and president (:domain Corporation)))
(defrelation daughter :is (:and child (:range Female)))
(defrelation husband :is (:inverse wife))
(defrelation grandparent :is (:compose parent parent))
(defrelation r* :is (:satisfies (?x ?z)
                          (:or (r ?x ?z)
                               (:for-some ?y (:and (r ?x ?y)
                                                   (r* ?y ?z))))))
(defrelation s0 :implies (:and super (:range Concept)))
(defrelation s1 :domain Person)
```

```
(defrelation spj :arity 3 :domains (Supplier Part))
(defrelation s2 :range Number)
(defrelation s3 :predicate ((n1 n2) (> n1 n2)))
(defrelation s4 :function ((n1 n2) (+ n1 n2)))
(defrelation s5 :inheritance-link direct-subconcepts
                :inheritance-method nearest-neighbor)
(defrelation s6 :annotations (Verb (past "-ed")))
(defrelation s7 :identifier age-as-relation :kb user-kb)
(defrelation s8 :characteristics (:single-valued :closed-world))
```

See Also:

concept--characteristics (p. 34)
create-concept (p. 44)
defconcept (p. 49)
define-relation (p. 58)
defset (p. 72)
direct-subconcepts (p. 80)
implies (p. 148)
meta-concept-p (p. 197)
Primitive-P (p. 225)
Relation (p. 229)
relation--domain (p. 231)
relation--name (p. 232)
relation--range (p. 233)
relation--restrictions (p. 234)
relation-p (p. 230)
set-features (p. 258)
Single-Valued-P (p. 266)
User-Defined-P (p. 293)

defset *Macro*

Purpose:

The **defset** macro defines or redefines a symbolic set.

Syntax:

defset *name* `&optional` *documentation* `&key` *is annotations identifier kb*

Arguments:

The *name* argument is a symbol. If *name* is null, the set is given a system-generated name.

The *documentation* argument is a string which attaches a comment to the set being defined. This string can be retrieved by calling the Lisp **documentation** function with **type** as the doc-type argument.

The *is* argument is a set-forming expression (see Remarks below) that defines the set as a group of ordered or unordered symbolic constants.

The *annotations* argument is a list of facts to be asserted about the set being defined. Each element of the list is either a concept name or an expression having the form (*RelationName instance*), where *instance* is either a constant or a symbol that identifies a Loom instance.

The *identifier* argument is the name under which the set will be interned in the **:instances** partition of its knowledge base. This argument defaults to *name*.

The *kb* argument is a string or symbol that names the knowledge base in which the set is to be interned. This defaults to the current knowledge base.

Value:

The defined or redefined set is returned.

Remarks:

All symbolic set members appearing in a **defset** should be quoted.

The **defconcept** macro should be used for defining non-symbolic sets such as numeric intervals or sets of Loom objects.

The set-forming expressions that appear in the *is* argument above have the following syntax:

```
set-expr ::=
    SetName |
    ( {:AND | :OR} set-expr+ ) |
    ( {:ONE-OF | :THE-ORDERED-SET} Symbol+ ) |
    ( :THROUGH Symbol Symbol ) ;
```

The set-forming operators have the following semantics:

- If C = (**:and** C1...Cn), a member of C is a member of the intersection of the sets C1...Cn.
- If C = (**:or** C1...Cn), a member of C is a member of the union of the sets C1...Cn.
- If C = (**:one-of** S1...Sn), a member of C is one of the symbols S1...Sn.
- If C = (**:the-ordered-set** S1...Sn), a member of C is one of the symbols S1...Sn, where for i from 1 to $n-1$, $Si < Si + 1$.
- If C = (**:through** Si Sj), where Si and Sj are members of an ordered set, a member of C is a symbol in the closed interval [Si,Sj].

Examples:

```
(defset Color :is (:one-of 'Red 'Green 'Blue ...)) ⇒ |C|COLOR
(defset Day :is (:the-ordered-set 'Monday 'Tuesday ...'Sunday))
(defset Weekday :is (:and Day (:through 'Monday 'Friday)))
```

See Also:

concept--instances (p. 35)
defconcept (p. 49)
defrelation (p. 66)
predecessor (p. 224)
successor (p. 271)

delete-all-methods *Function*

Purpose:

The **delete-all-methods** function deletes all methods that belong to a given action.

Syntax:

delete-all-methods *action*

Arguments:

The *action* argument is the action from which all methods are to be deleted.

Value:

The **delete-all-methods** function returns **nil**.

Examples:

`(delete-all-methods (fb move))` ⇒ `NIL`

See Also:

defaction (p. 46)
defmethod (p. 61)
delete-method (p. 76)
list-methods (p. 176)

delete-concept *Function*

Purpose:

The **delete-concept** function deletes a concept or relation.

Syntax:

delete-concept *concept* `&key` *type delete-merged-concepts-p*

Arguments:

The *concept* argument is a concept or relation, or the name of a concept or relation.

The *type* argument is the keyword **:concept** or **:relation**. It is needed when *concept* is a symbol that names both a concept and relation.

If *concept* has merged into some other concept, then if the *delete-merged-concepts-p* argument is **t**, both concepts will be deleted. Otherwise, the concepts will be unmerged and only *concept* will be deleted.

Value:

The **delete-concept** function normally returns *concept*. If merged concepts were deleted, then a list of these concepts is returned.

Remarks:

For each dependent of *concept*—that is, each concept whose definition directly or indirectly references *concept*—**delete-concept** asks if that dependent may be deleted. The *concept* itself is only deleted after the user has approved deletion of all dependents.

Examples:

```
(delete-concept 'A) ⇒ |C|A
(delete-concept 'r :type :relation) ⇒ |R|R
(defconcept C :is B)
(delete-concept (fc C) :delete-merged-concepts-p t) ⇒ (|C|B |C|C)
```

See Also:

clear-kb (p. 29)
destroy-unclassified-concepts (p. 79)
initialize-network (p. 153)

delete-method *Function*

Purpose:

The **delete-method** function deletes a method from a specified action.

Syntax:

delete-method `&key` *action title method error-p*

Arguments:

The *action* argument is the action from which the method is to be deleted.

The *title* argument is a string which titles the method.

The *method* argument is a method.

If the *error-p* argument is **t**, an error is generated if the specified method cannot be found.

Value:

The **delete-method** function returns the deleted method, or **nil** if the method cannot be found and *error-p* is not set.

Remarks:

Calls to **delete-method** must supply either (1) *action* and *title* arguments, or (2) a *method* argument.

Examples:

```
(delete-method :method (first (list-methods (fb move)))) ⇒
    |METHOD|MOVE-:UNTITLED
(delete-method :action (fb move) :title "move box") ⇒
    |METHOD|MOVE-"move box"
```

See Also:

defaction (p. 46)
defmethod (p. 61)
delete-all-methods (p. 74)
find-method (p. 103)

delete-production *Function*

Purpose:

The **delete-production** function deletes a specified production.

Syntax:

delete-production *production* `&key` *error-p*

Arguments:

The *production* argument is the production, or the name of the production, which is to be deleted.

If the *error-p* argument is **t**, an error is generated if the specified production cannot be found.

Value:

The **delete-production** function returns the specified production, or **nil** if the production cannot be found and *error-p* is not set.

Remarks:

This function renders productions inoperative but does not actually destroy them.

Examples:

```
(delete-production 'almost-full-container :error-p t) ⇒
    |PRODUCTION|ALMOST-FULL-CONTAINER
```

See Also:

defproduction (p. 64)
find-production (p. 105)
list-productions (p. 177)
trace-all (p. 284)

destroy *Method*
destroym *Macro*

Purpose:

The **destroy** function retracts all propositions relating to a given instance, and removes the instance from the extensions of all concepts in its type. The **destroym** macro does the same thing and then advances to a new knowledge base state.

Syntax:

destroy *instance* `&key` *dont-unintern-p*
destroym *instance* `&key` *dont-unintern-p*

Arguments:

The *instance* argument is an instance or instance identifier.

If the *dont-unintern-p* argument is `t`, *instance* remains accessible (as a **Thing** with no role fillers or features). Otherwise, *instance* is deleted from the knowledge base.

Value:

The **destroy** function returns `t`.

The **destroym** macro returns the current agent time.

Remarks:

Caution: Pointers to a deleted CLOS instance I are deleted from adjacent instances only if I inherits the mixin **INSTANCE-WITH-INVERSES**. Otherwise it is the user's responsibility to avoid creating dangling references.

Examples:

```
(destroy 'Joe) ⇒ T
(destroy (fi Sue) :dont-unintern-p t) ⇒ NIL
(destroym 'Sue) ⇒ 34
```

See Also:

clear-kb (p. 29)
create (p. 42)
forget (p. 109)
initialize-instances (p. 152)
initialize-network (p. 153)
unmake-object (p. 285)

destroy-unclassified-concepts *Function*

Purpose:

The **destroy-unclassified-concepts** function destroys all concepts that are currently unclassified.

Syntax:

destroy-unclassified-concepts

Arguments:

The *destroy-unclassified-concepts* function takes no arguments.

Value:

The **destroy-unclassified-concepts** function returns a list of the names of the concepts that were destroyed.

Remarks:

All dependents – that is, all concepts whose definitions directly or indirectly reference the unclassified concepts – are recursively deleted.

This function is often employed to delete concepts created "by accident", e.g., when a concept reference within a definition is misspelled.

Examples:

```
(defconcept A :is-primitive B :implies C)
(destroy-unclassified-concepts) ⇒ (A B C)
```

See Also:

delete-concept (p. 75)
find-definitional-cycles (p. 98)
list-unclassified-concepts (p. 182)

direct-subconcepts *Relation*

Purpose:

The **direct-subconcepts** relation relates a concept or relation to its immediate subconcepts (subrelations).

Syntax:

direct-subconcepts *concept1 concept2*

Domain:

The *concept1* argument is the immediate superconcept or superrelation.

Range:

The *concept2* argument is the immediate subconcept or subrelation.

Examples:

```
(defconcept Man :is-primitive Person)
(defconcept Bachelor :is (:and Man (:exactly 0 wife)))
(ask (direct-subconcepts Man Bachelor)) ⇒ T
(retrieve ?x (direct-subconcepts Person ?x)) ⇒ (|C|MAN)
```

See Also:

direct-superconcepts (p. 81)
get-subconcepts (p. 137)
get-subrelations (p. 138)
isa-p (p. 162)
subconcept-p (p. 268)
subconcepts (p. 269)

direct-superconcepts *Relation*

Purpose:

The **direct-superconcepts** relation relates a concept or relation to its immediate superconcepts (superrelations).

Syntax:

direct-superconcepts *concept1 concept2*

Domain:

The *concept1* argument is the immediate subconcept or subrelation.

Range:

The *concept2* argument is the immediate superconcept or superrelation.

Examples:

```
(defrelation child :is-primitive relative)
(defrelation daughter :is (:and child (:range Female)))
(ask (direct-superconcepts child relative)) ⇒ T
(retrieve ?x (direct-superconcepts daughter ?x)) ⇒ (|R|CHILD)
```

See Also:

direct-subconcepts (p. 80)
get-superconcepts (p. 139)
get-superrelations (p. 140)
isa-p (p. 162)
subconcept-p (p. 268)
superconcepts (p. 273)

disjoint-concepts-p *Method*

Purpose:

The **disjoint-concepts-p** method is used to determine whether two concepts are disjoint, i.e., cannot have members in common.

Syntax:

disjoint-concepts-p *concept1 concept2*

Arguments:

The *concept1* and *concept2* arguments are concepts or sets.

Value:

The **disjoint-concepts-p** function returns non-**nil** if *concept1* is provably disjoint from *concept2*.

Remarks:

The disjointness test is not yet implemented for relations.

Examples:

```
(disjoint-concepts-p (fc Woman) (fc Bachelor)) ⇒ T
(disjoint-concepts-p (fc Weekday) (fc Weekend-Day) ⇒ T
```

See Also:

defconcept (p. 49)
Incoherent (p. 150)
Incoherent-Concept (p. 151)

do-instances *Macro*

Purpose:

The **do-instances** macro executes a body of Lisp code once for each instance in each knowledge base of a Loom network.

Syntax:

do-instances *variable* `&body` *body*

Arguments:

The *variable* argument is a list containing a single iteration variable.

The *body* argument is the Lisp code to be executed for each binding of the iteration variable.

Value:

This function returns **nil**.

Remarks:

This function iterates over all instances, including concepts and relations.

Examples:

```
(do-instances (x)
    (foo x)
    (unless (meta-concept-p x) (bar x)))
```

See Also:

initialize-instances (p. 152)
list-instances (p. 170)
list-kb (p. 172)

do-retrieve *Macro*

Purpose:

The **do-retrieve** macro executes a body of Lisp code once for each instance or tuple retrieved by a given query.

Syntax:

do-retrieve *variables query* `&body` *body*

Arguments:

The *variables* argument is a symbol, or list of symbols, each beginning with the character **?**.

The *query* argument is an arbitrary Loom query expression (see **retrieve** Remarks).

The *body* argument is the Lisp code to be executed for each binding of *variables*.

Value:

This function returns **nil**.

Examples:

```
(do-retrieve (?p ?a)
    (:and (Person ?p) (age ?p ?a))
    (tellm (age ?p (+ ?a 1))))
```

See Also:

retrieve (p. 246)

fadd-value *Function*

Purpose:

The **fadd-value** function adds a value to a given role on a given instance. It is faster than **add-value** because it does no error checking or argument coercion.

Syntax:

fadd-value *instance role value*

Arguments:

The *instance* argument is the instance which is to receive an added role filler.

The *role* argument is a relation, or the name of a relation.

The *value* argument is a value to be added to the fillers of *role*. If *value* is a list, the list is interpreted as a single role filler. If *role* is single-valued, and already has a filler, the old filler is replaced by *value*.

Value:

The **fadd-value** function returns *value*.

Examples:

```
(defrelation rr)
(fadd-value (fi Joe) (fr rr) (fi Mary)) ⇒ |I|MARY
(fadd-value (fi Joe) (fr rr) (fi Sue)) ⇒ |I|SUE
(fadd-value (fi Joe) 'rr '(3 4 5)) ⇒ (3 4 5)
(get-values 'Joe 'rr) ⇒ (I|MARY |I|SUE (3 4 5))
(defrelation r :characteristics :single-valued)
(fadd-value (fi Joe) (fr r) (fi Mary)) ⇒ |I|MARY
(fadd-value (fi Joe) (fr r) (fi Sue)) ⇒ |I|SUE
(get-values 'Joe 'r) ⇒ (|I|SUE)
```

See Also:

add-value (p. 18)
fremove-value (p. 113)

fail *Macro*

Purpose:

The **fail** macro causes the method in which it is invoked to fail immediately.

Syntax:

fail `&optional` *result*

Arguments:

The *result* argument is returned as the value of the failed method. If no *result* is supplied, the method returns **nil**.

Examples:

```
(defmethod move (?x)
    :title "move box"
    :situation (Box ?x)
    :response ((if *condition*
                   (fail)
                   (move-box ?x))))
(setq *condition* t)
(perform (move (fi Box3)) :return :success-p) ⇒ NIL
```

See Also:

defmethod (p. 61)
perform (p. 214)

fb *Macro*

Purpose:

The **fb** macro performs an exhaustive search for the action, method, or production having a specified name. It should be used interactively rather than in application code because it is much slower than **find-action**, **find-method**, or **find-production**.

Syntax:

fb *behaviorOrTitle* `&optional` *title*

Arguments:

The *behaviorOrTitle* argument is a Lisp form which either names a behavior (an action, method, or production), or evaluates to a behavior or behavior name. If it is a symbol, special characters (e.g., wildcards) are interpreted as in **fc**, and package prefixes are optional because the search extends across packages. If it is a string, it is taken to be a method title.

The *title* argument is a string which titles a method.

Value:

The **fb** function returns the action, method, or production with the specified name, or **nil** if none is found.

Remarks:

If *behaviorOrTitle* and *title* are supplied, **fb** searches for a method of *behaviorOrTitle* having the specified title. If *behaviorOrTitle* is a title, then **fb** searches for any method at all having the title *title*. Otherwise, **fb** searches for an action or production named *behaviorOrTitle*.

Examples:

```
(fb move) ⇒ |ACTION|MOVE
(fb bottom-kb^move "move box") ⇒ |METHOD|MOVE-"move box"
(fb "move box") ⇒ |METHOD|MOVE-"move box"
(fb *-container) ⇒ |PRODUCTION|ALMOST-FULL-CONTAINER
```

See Also:

fc (p. 89)
find-action (p. 95)
find-method (p. 103)

find-production (p. 105)
get-action (p. 115)
get-method (p. 124)
get-production (p. 126)
pb (p. 210)
where-is-it (p. 295)

fc *Macro*

Purpose:

The **fc** macro performs an exhaustive search for the concept having a specified name. It should be used interactively rather than in application code because it is much slower than **get-concept** or **find-concept**.

Syntax:

fc *datum* `&key` *kb*

Arguments:

The *datum* argument is a Lisp expression. It may be an extended identifier having the form *kb*^*name*, where *kb* is the knowledge base from which the search starts. It may also be an identifier beginning and/or ending with the wildcard character *, in which case the search proceeds until the first matching concept is found. It is not necessary to specify the package in which *datum* resides since the search is package-independent. If *datum* is a symbol, Loom first tries to locate the concept identified by that symbol. If this fails, or if *datum* is a list, Loom evaluates *datum* and searches for the concept corresponding to the resulting value.

The *kb* argument is the name of a knowledge base, or a form which evaluates to a knowledge base or knowledge base name. This argument defaults to the current knowledge base.

Value:

Two values are returned: (1) the concept having the specified name, or **nil** if none was found, (2) the knowledge base in which the concept was found, or **nil** if no concept was found.

Remarks:

This is the most thorough and flexible of the functions for locating Loom concepts, but it is also the slowest. Because *datum* may be evaluated at compile time, **fc** cannot be reliably used in application code.

Examples:

```
(fc Thing) ⇒ |C|THING |K|UPPER-STRUCTURE-KB
(fc user-kb^binary-search) ⇒ |C|BINARY-SEARCH |K|USER-KB
(fc *-search :kb bottom-kb) ⇒ |C|BINARY-SEARCH |K|USER-KB
(setq concepts '(Man Woman))
(fc (first concepts)) ⇒ |C|MAN |K|USER-KB
```

See Also:

fb (p. 87)
fi (p. 92)
find-concept (p. 96)
fr (p. 111)
get-concept (p. 116)
pc (p. 212)
where-is-it (p. 295)

fget-value *Macro*

Purpose:

The **fget-value** macro gets the value of a given role on a given instance. It is faster than **get-value** because it does no error checking or argument coercion.

Syntax:

fget-value *instance role*

Arguments:

The *instance* argument is the instance whose role value is to be retrieved.

The *role* argument is a relation, or the name of a relation. If this argument is a quoted symbol, better performance is obtained because query checks can be compiled out.

Value:

If *role* is single-valued, **fget-value** returns a single filler (which may be a list). If *role* is multiple-valued, a list of fillers is returned.

Remarks:

The **fget-value** function does not ensure that updates to the current state have been incorporated. If assertions are made by functions (like **fget-value**) that don't advance to a new state, it is necessary to manually update the database by calling **new-time-point** before calling **fget-value**.

Examples:

```
(defrelation r :characteristics :single-valued)
(defrelation rr)
(set-value 'Joe 'r 'Sue)
(set-values 'Joe 'rr '(Mary Sue (3 4 5)))
(fget-value (fi Joe) 'r) ⇒ NIL
(new-time-point)
(fget-value (fi Joe) 'r) ⇒ |I|SUE
(fget-value (fi Joe) 'rr) ⇒ (|I|MARY |I|SUE (3 4 5))
```

See Also:

fset-value (p. 114)
get-value (p. 142)
new-time-point (p. 205)

fi *Macro*

Purpose:

The **fi** macro performs an exhaustive search for the instance having a specified identifier. It should be used interactively rather than in application code because it is much slower than **get-instance** or **find-instance**.

Syntax:

fi *datum* `&key` *kb*

Arguments:

The *datum* argument is a Lisp expression. It may be an extended identifier having the form *kb^identifier*, where *kb* is the knowledge base from which the search starts. It may also be an identifier beginning and/or ending with the wildcard character *, in which case the search proceeds until the first matching instance is found. It is not necessary to specify the package in which *datum* resides since the search is package-independent. If *datum* is a symbol, Loom first tries to locate the instance identified by that symbol. If this fails, or if *datum* is a list, Loom evaluates *datum* and searches for the instance corresponding to the resulting value.

The *kb* argument is the name of a knowledge base, or a form which evaluates to a knowledge base or knowledge base name. This argument defaults to the current knowledge base.

Value:

Two values are returned: (1) the instance having the specified identifier, or **nil** if none was found, (2) the knowledge base in which the instance was found, or **nil** if no instance was found.

Remarks:

This is the most thorough and flexible of the functions for accessing Loom instances, but it is also the slowest. Because *datum* may be evaluated at compile time, **fi** cannot be reliably used in application code.

Examples:

```
(fi Joe) ⇒ |I|JOE |K|USER-KB
(fi Thing) ⇒ |C|THING |K|UPPER-STRUCTURE-KB
(fi user-kb^Los-Angeles) ⇒ |I|LOS-ANGELES |K|USER-KB
(fi Los* :kb bottom-kb) ⇒ |R|LOS-ANGELES |K|USER-KB
(setq instances '(Joe Fred))
(fi (second instances)) ⇒ |I|FRED |K|USER-KB
```

See Also:

fc (p. 89)
find-instance (p. 99)
get-instance (p. 118)
pi (p. 217)
where-is-it (p. 295)

finalize-definitions *Function*

Purpose:

The **finalize-definitions** function causes Loom to perform a number of internal actions that finalize the state of the definitions in the network before data manipulation operations are performed. In particular, this function can be used to force the early creation of CLOS classes for all concepts having the **:clos-class** characteristic. (Normally these classes aren't created until the concepts are first instantiated.)

Syntax:

finalize-definitions

Arguments:

This function takes no arguments.

Value:

The **finalize-definitions** function returns **nil**.

Remarks:

Simply calling **tellm** with no arguments has the same effect as calling **finalize-definitions**.

Examples:

```
(finalize-definitions) ⇒ NIL
```

See Also:

make-object (p. 189)
tellm (p. 275)

find-action *Function*

Purpose:

The **find-action** function, a robust variant of **get-action**, searches for the action having a specified name.

Syntax:

find-action *action* `&key` *no-warning-p*

Arguments:

The *action* argument is an action, or the name of an action.

If the *no-warning-p* argument is **t**, then no warning is generated if Loom cannot find an action having the specified name.

Value:

The **find-action** function returns the action with the specified name, or **nil** if none is found.

Remarks:

The **find-** functions tend to be more flexible, slower, and do more error-checking than the **get-** functions (such as **get-action**).

Examples:

```
(find-action 'move) ⇒ |ACTION|MOVE
(find-action 'open :no-warning-p t) ⇒ |ACTION|OPEN
```

See Also:

defaction (p. 46)
fb (p. 87)
get-action (p. 115)

find-concept *Function*

Purpose:

The **find-concept** function, a robust variant of **get-concept**, searches for the concept having a specified name.

Syntax:

find-concept *concept* `&key` *no-warning-p ignore-package-p kb*

Arguments:

The *concept* argument is a concept, or a symbol or string which names a concept. If it is a symbol, it may be an extended identifier having the form *kb^name*, where *kb* is the knowledge base from which the search is to proceed.

If the *no-warning-p* argument is **t**, then no warning is generated if Loom cannot find a concept having the specified name.

If the *ignore-package-p* argument is **t**, then Loom searches across packages for the specified name.

The *kb* argument is the knowledge base, or the name of the knowledge base, from which the search is to start. The concept will be found if it is interned in *kb* or any knowledge base above *kb*. This argument defaults to the current knowledge base.

Value:

The **find-concept** function returns the concept with the specified name, or **nil** if none is found.

Remarks:

The **find-** functions tend to be more flexible, slower, and do more error-checking than the **get-** functions (such as **get-concept**). Package-independent searches may be especially slow.

Examples:

```
(find-concept 'Thing) ⇒ |C|THING
(find-concept 'user-kb^Person :no-warning-p t) ⇒ |C|PERSON
(find-concept "Computer" :kb 'bottom-kb) ⇒ |C|COMPUTER
```

See Also:

fc (p. 89)
find-instance (p. 99)
find-relation (p. 106)
get-concept (p. 116)
name--concept (p. 203)
pc (p. 212)

find-definitional-cycles *Function*

Purpose:

The **find-definitional-cycles** function finds sets of concepts whose definitions depend on one another.

Syntax:

find-definitional-cycles

Arguments:

This function has no arguments.

Value:

The **find-definitional-cycles** function returns a list of cycles. Each cycle is a list of concepts and/or relations which describe a circular path through the terminological network.

Remarks:

All concepts must be classified before any data manipulation operations can be performed. Two reasons why unclassified concepts sometimes remain after evaluating a set of definitions are: (1) some concept has been referenced but not defined, and (2) cycles exist in the definitions. The first situation can be addressed by making sure that all concepts returned by **list-undefined-concepts** have been defined. The second can be addressed by using **find-definitional-cycles** to locate circularly defined concepts, and then moving restrictions out of definitions and into implications (that is, out of the **:is** clause and into the **:implies** clause).

Examples:

```
(defconcept A :is (:all r A))
(defconcept B :is (:all r C))
(defconcept C :is-primitive B)
(find-definitional-cycles) ⇒ ((|C|A |C|A) (|C|B |C|C |C|B))
```

See Also:

destroy-unclassified-concepts (p. 79)
list-kb (p. 172)
list-unclassified-concepts (p. 182)

find-instance *Function*

Purpose:

The **find-instance** function, a robust variant of **get-instance**, searches for the istance having a specified identifier.

Syntax:

find-instance *instance* `&key` *no-warning-p ignore-package-p kb*

Arguments:

The *instance* argument is an instance, or a symbol or string which identifies an instance. If it is a symbol, it may be an extended identifier having the form *kb^identifier*, where *kb* is the knowledge base from which the search is to proceed.

If the *no-warning-p* argument is **t**, then no warning is generated if Loom cannot find an instance having the specified identifier.

If the *ignore-package-p* argument is **t**, then Loom searches across packages for the specified name.

The *kb* argument is the knowledge base, or the name of the knowledge base, from which the search is to start. The instance will be found if it is interned in *kb* or any knowledge base above *kb*. This argument defaults to the current knowledge base.

Value:

The **find-instance** function returns the instance with the specified identifier, or **nil** if none is found.

Remarks:

The **find-** functions tend to be more flexible, slower, and do more error-checking than the **get-** functions (such as **get-instance**). Package-independent searches may be especially slow.

Examples:

```
(find-instance 'Joe) ⇒ |I|JOE
(find-instance 'user-kb^Mary :no-warning-p t) ⇒ |I|MARY
(find-instance 'New-York :ignore-package-p t) ⇒ |I|NEW-YORK
(find-instance "Joe" :kb 'bottom-kb) ⇒ |I|JOE
```

See Also:

do-instances (p. 83)
fi (p. 92)
find-concept (p. 96)
find-knowledge-base-of-instance (p. 102)
find-or-create-instance (p. 104)
find-relation (p. 106)
get-instance (p. 118)
identifier--instance (p. 147)
list-instances (p. 170)
pi (p. 217)

find-kb *Function*

Purpose:

The **find-kb** function finds the knowledge base having a specified name.

Syntax:

find-kb *knowledgeBase* `&key` *no-warning-p*

Arguments:

The *knowledgeBase* argument is a knowledge base, or a symbol or string that names a knowledge base. This function operates in a package-independent manner, so it is not necessary to specify the package in which a knowledge base name resides.

If the *no-warning-p* argument is **t**, then no warning is issued if *knowledgeBase* is not a knowledge base, symbol, or string.

Value:

The **find-kb** function returns the knowledge base, or **nil** if no knowledge base with the specified name can be found.

Examples:

```
(find-kb 'upper-structure-kb) ⇒ |K|UPPER-STRUCTURE-KB
(find-kb "USER-KB" :no-warning-p t) ⇒ |K|USER-KB
```

See Also:

defkb (p. 59)
find-knowledge-base-of-instance (p. 102)
list-knowledge-bases (p. 174)

find-knowledge-base-of-instance *Function*

Purpose:

The **find-knowledge-base-of-instance** function finds the knowledge base in which a given instance resides.

Syntax:

find-knowledge-base-of-instance *instance*

Arguments:

The *instance* argument is an instance.

Value:

The **find-knowledge-base-of-instance** function returns the knowledge base in which *instance* is interned.

Examples:

`(find-knowledge-base-of-instance (fi Joe))` ⇒ `|K|USER-KB`

See Also:

defkb (p. 59)
find-instance (p. 99)

find-method *Macro*

Purpose:

The **find-method** function, a robust variant of **get-method**, searches for the method having a specified name or title.

Syntax:

find-method *actionOrTitle* `&optional` *title* `&rest` *no-error-p*

Arguments:

The *actionOrTitle* argument is either an action, the name of an action, or a string which titles a method.

The *title* argument is a string which titles a method.

If the *no-error-p* argument is `t`, then no error is generated if Loom cannot find a method having the specified name or title.

Value:

The **find-method** macro returns the method with the specified name and/or title. If there is none, **nil** is returned if *no-error-p* is set, and otherwise an error is generated.

Remarks:

If an action is supplied as the *actionOrTitle* argument of **find-method**, then a *title* argument must also be supplied. If *actionOrTitle* is a title, then **find-method** searches for any method at all having a matching title.

The **find-** functions tend to be more flexible, slower, and do more error-checking than the **get-** functions. In particular, **find-method** is much slower than **get-method**.

Examples:

```
(find-method 'move "move box") ⇒ |METHOD|MOVE-"move box"
(find-method "open box" :no-error-p t) ⇒ |METHOD|OPEN-"open box"
```

See Also:

defaction (p. 46)
defmethod (p. 61)
fb (p. 87)
get-method (p. 124)
list-methods (p. 176)

find-or-create-instance *Function*

Purpose:

The **find-or-create-instance** function either finds an instance having a given identifier, or creates a new instance with that identifier.

Syntax:

find-or-create-instance *instance concept*

Arguments:

The *instance* argument is an instance, or an instance identifier.

The *concept* argument is a concept, or the name of a concept, that is used as the creation type of the new instance. If *concept* is **nil**, the creation type is **Thing**.

Value:

The found or newly-created instance is returned.

Remarks:

The new instance does not actually assume its creation type until **tellm**, **new-time-point**, or some other function, causes the knowledge base state to be advanced.

Examples:

```
(find-or-create-instance 'Joe nil) ⇒ |I|JOE
(find-or-create-instance 'Fred 'Man) ⇒ |I|FRED
(find-or-create-instance 'Mary (fc Woman)) ⇒ |I|MARY
(pi Mary) ⇒ (TELL (:ABOUT MARY THING))
(new-time-point)
(pi Mary) ⇒ (TELL (:ABOUT MARY WOMAN))
```

See Also:

create (p. 42)
find-instance (p. 99)
new-time-point (p. 205)
tell (p. 275)

find-production *Function*

Purpose:

The **find-production** function, a robust variant of **get-production**, searches for the production having a specified name.

Syntax:

find-production *production* `&key` *no-warning-p*

Arguments:

The *production* argument is a production rule, or the name of a production rule.

If the *no-warning-p* argument is **t**, then no warning is generated if Loom cannot find a production having the specified name.

Value:

The **find-production** function returns the production with the specified name, or **nil** if none is found.

Remarks:

The **find-** functions tend to be more flexible, slower, and do more error-checking than the **get-** functions (such as **get-production**).

Examples:

```
(find-production 'P2 ⇒ |PRODUCTION|P2
(find-production 'almost-full-container :no-warning-p t) ⇒
    |PRODUCTION|ALMOST-FULL-CONTAINER
```

See Also:

defproduction (p. 64)
fb (p. 87)
get-production (p. 126)
list-productions (p. 177)

find-relation *Function*

Purpose:

The **find-relation** function, a robust variant of **get-relation**, searches for the relation having a specified name.

Syntax:

find-relation *relation* `&key` *no-warning-p ignore-package-p kb*

Arguments:

The *relation* argument is a relation, or a symbol or string which names a relation. If it is a symbol, it may be an extended identifier having the form *kb^name*, where *kb* is the knowledge base from which the search is to proceed.

If the *no-warning-p* argument is **t**, then no warning is generated if Loom cannot find a relation having the specified name.

If the *ignore-package-p* argument is **t**, then Loom searches across packages for the specified name.

The *kb* argument is the knowledge base, or the name of the knowledge base, from which the search is to start. The relation will be found if it is interned in *kb* or any knowledge base above *kb*. This argument defaults to the current knowledge base.

Value:

The **find-relation** function returns the relation with the specified name, or **nil** if none is found.

Remarks:

The **find-** functions tend to be more flexible, slower, and do more error-checking than the **get-** functions (such as **get-relation**). Package-independent searches may be especially slow.

Examples:

```
(find-relation '+) ⇒ |R|+
(find-relation 'user-kb^age :no-warning-p t) ⇒ |R|AGE
(find-relation 'r :ignore-package-p t) ⇒ |R|R
(find-relation "child" :kb 'bottom-kb) ⇒ |R|CHILD
```

See Also:

find-concept (p. 96)
find-instance (p. 99)
fr (p. 111)
get-relation (p. 127)
name--relation (p. 204)
pr (p. 222)

find-subsumers&subsumees *Function*

Purpose:

The **find-subsumers&subsumees** function determines where a concept would classify in the inheritance network, without actually defining and classifying that concept.

Syntax:

find-subsumers&subsumees *expression* `&optional` *kb*

Arguments:

The *expression* argument is a concept-forming expression (see **defconcept** Remarks). It is syntactically equivalent to the **:is** clause of a concept definition.

The *kb* argument is the knowledge base in which *expression* is to be evaluated. It defaults to the current knowledge base.

Value:

The **find-subsumers&subsumees** function returns three values: (1) a list of concepts that subsume *expression*, (2) a list of concepts that are subsumed by *expression*, (3) a concept equivalent to *expression*, or **nil** if none exists.

Examples:

```
(find-subsumers&subsumees '(and Man (:exactly 0 wife))) ⇒
    (|C|MAN) (|C|CONFIRMED-BACHELOR) |C|BACHELOR
(find-subsumers&subsumees '(:exactly 0 wife)) (find-kb "USER-KB") ⇒
    (|C|THING) (|C|BACHELOR) NIL
```

See Also:

direct-subconcepts (p. 80)
direct-superconcepts (p. 81)
subconcept-p (p. 268)

forget *Macro*
forgetm *Macro*

Purpose:

The **forget** macro is used to retract propositions from the Loom database. These propositions may state that an instance belongs to a given concept or has a particular feature, or that a certain relationship holds between two or more instances.

The **forgetm** macro retracts one or more propositions, and then advances to a new knowledge base state.

Syntax:

forget `&body` *propositions*
forgetm `&body` *propositions*

Arguments:

The *propositions* argument consists of zero or more propositions (see **Remarks** below) which are to be retracted.

Value:

The **forget** macro normally returns the symbol **ok**. However, if *propositions* contains one or more **:about** clauses, the subject of the last such clause is returned.

The **forgetm** macro returns the newly-advanced state time.

Remarks:

Each proposition in the *propositions* argument above has the following syntax:

```
proposition ::=
    ( concept instance ) |
    ( relation instance+ value ) |
    ( :CREATE ?Var concept [:CLOS] ) |
    ( :ABOUT instance about-clause* )
```

Each *about-clause* in a proposition has the form:

```
about-clause ::=
    concept | ( concept ) | ( relation value ) |
    ( :FILLED-BY relation value+ ) |
    ( {:AT-LEAST | :AT-MOST | :EXACTLY} Integer relation ) |
    ( {:ALL | :SOME | :THE} relation concept )
```

A *concept*, *relation*, or *instance* may be either a symbol that names a Loom object, or a variable beginning with the character **?**. An *instance* may also be a constant or a formula, where a formula is a list of the form (*relation instance*).

All **?**-variables in a retraction are assumed to be bound externally. The value of an external **?**-variable should be a Loom object, i.e., a concept, relation, or instance, rather than the name of such an object.

If the symbol * is used in place of a role value, *all* fillers of that role are retracted.

The Loom Grammar at the end of this manual is too general in the following respects: (1) Loom does not currently support the retraction of **:same-as** propositions, and (2) **:filled-by-list** cannot be used in the **:about** clause of a retraction.

Concepts and relations referenced by name in a retraction must be defined at the time the retraction is compiled. References to an instance, say A, bind to a previously declared concept or relation with name A, if such exists, and otherwise they bind to an existing instance with identifier A.

Examples:

```
(forget (Person Joe)) ⇒ OK
(forget (age Joe 30) (wife Joe Sue) (Teacher Sue)) ⇒ OK
(forget (age (wife Joe) (+ (age Joe) 1))) ⇒ OK
(forgetm (child Joe *)) ⇒ 3
(setq ?C (fc Person) ?R (fr age) ?I (fi Joe) ?V 30)
(forgetm (?C ?I) (?R ?I ?V)) ⇒ 4
(forget (:about Joe Person (wife Sue))) ⇒ |I|JOE
(forgetm (:about Joe (child Fred) (child Mary))) ⇒ 5
(forgetm (:about Joe (:filled-by child Fred Mary Don))) ⇒ 6
(forgetm (:about Joe (:at-least 3 child) (:at-most 1 daughter)
                     (:exactly 2 son))) ⇒ 7
(forgetm (:about Joe (:all child Adult) (:some son Bachelor))) ⇒ 8
(setq ?I (fi Joe) ?R (fr child) ?C (fc Adult))
(forgetm (:about ?I (:all ?R ?C))) ⇒ 9
```

See Also:

destroy (p. 78)
initialize-instances (p. 152)
new-time-point (p. 205)
remove-type (p. 235)
remove-value (p. 236)
tell (p. 275)

fr *Macro*

Purpose:

The **fr** macro performs an exhaustive search for the relation having a specified name. It should be used interactively rather than in application code because it is much slower than **get-relation** or **find-relation**.

Syntax:

fr *datum* `&key` *kb*

Arguments:

The *datum* argument is a Lisp expression. It may be an extended identifier having the form *kb^name*, where *kb* is the knowledge base from which the search starts. It may also be an identifier beginning and/or ending with the wildcard character *, in which case the search proceeds until the first matching relation is found. It is not necessary to specify the package in which *datum* resides since the search is package-independent. If *datum* is a symbol, Loom first tries to locate the relation named by that symbol. If this fails, or if *datum* is a list, Loom evaluates *datum* and searches for the relation corresponding to the resulting value.

The *kb* argument is the name of a knowledge base, or a form which evaluates to a knowledge base or knowledge base name. This argument defaults to the current knowledge base.

Value:

Two values are returned: (1) the relation having the specified name, or **nil** if none was found, (2) the knowledge base in which the relation was found, or **nil** if no relation was found.

Remarks:

This is the most thorough and flexible of the functions for accessing Loom relations, but it is also the slowest. Because *datum* may be evaluated at compile time, **fr** cannot be reliably used in application code.

Examples:

```
(fr binary-tuple) ⇒ |R|BINARY-TUPLE |K|UPPER-STRUCTURE-KB
(fr user-kb^roundtrip-fare) ⇒ |R|ROUNDTRIP-FARE |K|USER-KB
(fr *trip* :kb bottom-kb) ⇒ |R|ROUNDTRIP-FARE |K|USER-KB
(setq relations '(height weight))
(fr (first relations)) ⇒ |R|HEIGHT |K|USER-KB
```

See Also:

fc (p. 89)
find-relation (p. 106)
get-relation (p. 127)
pr (p. 222)
where-is-it (p. 295)

fremove-value *Function*

Purpose:

The **fremove-value** function removes a value from a given role on a given instance. It is faster than **remove-value** because it does no error checking.

Syntax:

fremove-value *instance role value*

Arguments:

The *instance* argument is the instance from which a role filler is to be removed.

The *role* argument is a relation, or the name of a relation.

The *value* argument is a value to be removed from the fillers of *role*. If *value* is a list, it is interpreted as a single role filler.

Value:

The **fremove-value** function returns *value*.

Examples:

```
(defrelation rr)
(set-values 'Joe 'rr '(Mary Sue (3 4 5))) ⇒ (|I|MARY |I|SUE (3 4 5))
(fremove-value (fi Joe) (fr rr) (fi Mary)) ⇒ |I|MARY
(fremove-value (fi Joe) 'rr '(3 4 5)) ⇒ (3 4 5)
(get-values 'Joe 'rr) ⇒ (|I|SUE)
```

See Also:

fadd-value (p. 85)
remove-value (p. 236)

fset-value *Function*

Purpose:

The **fset-value** function replaces the current value of a given role on a given instance. It is faster than **set-value** because it does no error checking or argument coercion.

Syntax:

fset-value *instance role value*

Arguments:

The *instance* argument is the instance whose role value is to be changed.

The *role* argument is a relation, or the name of a relation.

The *value* argument is the new value of *role*. If *role* is single-valued and *value* is a list, the list is interpreted as a single role filler. If *role* is multiple-valued, then *value* must be a list, and each member of the list is interpreted as a separate filler.

Value:

The **fset-value** function returns *value*.

Examples:

```
(defrelation r :characteristics :single-valued)
(fset-value (fi Joe) (fr r) '(3 4 5)) ⇒ (3 4 5)
(fset-value (fi Joe) (fr r) (fi Sue)) ⇒ |I|SUE
(defrelation rr)
(fset-value (fi Joe) 'rr (list (fi Mary) (fi Sue))) ⇒ (|I|MARY |I|SUE)
```

See Also:

fget-value (p. 91)
set-value (p. 261)

get-action *Function*

Purpose:

The **get-action** function returns the action having a specified name.

Syntax:

get-action *action* `&key` *no-error-p*

Arguments:

The *action* argument is an action, or the name of an action.

If the *no-error-p* argument is **t**, then no error is generated if Loom cannot find an action having the specified name.

Value:

The **get-action** function returns the specified action, or **nil** if it cannot be found.

Remarks:

The **get-** functions are faster and do less error-checking than the **find-** functions. Applications should use the **get-** functions for best performance.

Examples:

```
(get-action 'move) ⇒ |ACTION|MOVE
(get-action 'open :no-error-p t) ⇒ |ACTION|OPEN
```

See Also:

defaction (p. 46)
find-action (p. 95)

get-concept *Function*

Purpose:

The **get-concept** function returns the concept having a specified name.

Syntax:

get-concept *concept* `&key` *no-error-p kb*

Arguments:

The *concept* argument is a concept, or a symbol which names a concept. If it is a symbol, it may be an extended identifier having the form *kb^name*, where *kb* is the knowledge base from which the search is to proceed.

If the *no-error-p* argument is **t**, then no error is generated if Loom cannot find a concept having the specified name.

The *kb* argument is the knowledge base, or the name of the knowledge base, from which the lookup is to be performed. The concept will be found if it is interned in *kb* or any knowledge base above *kb*. This argument defaults to the current knowledge base.

Value:

The **get-concept** function returns the concept with the specified name. If there is none, **nil** is returned if *no-error-p* is set, and otherwise a continuable error is generated.

Remarks:

The **get-** functions are faster and do less error-checking than the **find-** functions (such as **find-concept**). Applications should use the **get-** functions for best performance.

Examples:

```
(get-concept 'Thing) ⇒ |C|THING
(get-concept 'user-kb^Person :no-error-p t) ⇒ |C|PERSON
(get-concept 'Computer :kb 'bottom-kb) ⇒ |C|COMPUTER
```

See Also:

find-concept (p. 96)
get-instance (p. 118)
get-relation (p. 127)

get-indices *Function*

Purpose:

The **get-indices** function returns the indices attached to a given concept. These indices accelerate retrieval based on the values of specified roles of the concept.

Syntax:

get-indices *concept* `&key` *direct-p*

Arguments:

The *concept* argument is concept, or the name of a concept.

If the *direct-p* argument is **t**, only indices directly defined on *concept* are returned. Otherwise, inherited indices are also returned.

Value:

The **get-indices** function returns a list of indices, where each index is identified by either a role name or a list of role names.

Examples:

```
(defconcept A :indices (r))
(defconcept B :is-primitive A :indices ((s1 s2)))
(get-indices 'A) ⇒ ((|R|R))
(get-indices 'B) ⇒ ((|R|R) (|R|S1 |R|S2))
(get-indices (fc B) :direct-p t) ⇒ ((|R|S1 |R|S2))
```

See Also:

defconcept (p. 49)
get-keys (p. 121)
get-roles (p. 135)

get-instance *Function*

Purpose:

The **get-instance** function returns the instance having a specified identifier.

Syntax:

get-instance *instance* `&key` *error-p kb*

Arguments:

The *instance* argument is an instance, or a symbol which identifies an instance. If it is a symbol, it may be an extended identifier having the form *kb^identifier*, where *kb* is the knowledge base from which the search is to proceed.

If the *error-p* argument is **t**, then an error is generated if Loom cannot find an instance having the specified identifier.

The *kb* argument is the knowledge base, or the name of the knowledge base, from which the lookup is to be performed. The object will be found if it is interned in *kb* or any knowledge base above *kb*. This argument defaults to the current knowledge base.

Value:

The **get-instance** function returns the instance with the specified identifier. If there is none, **nil** is returned unless *error-p* is set, in which case a continuable error is generated.

Remarks:

The **get-** functions are faster and do less error-checking than the **find-** functions (such as **find-instance**). Applications should use the **get-** functions for best performance.

Examples:

```
(get-instance 'Joe) ⇒ |I|JOE
(get-instance 'user-kb^Mary :error-p t) ⇒ |I|MARY
(get-instance 'New-York :kb 'bottom-kb) ⇒ |I|NEW-YORK
```

See Also:

find-instance (p. 99)
get-concept (p. 116)
get-relation (p. 127)

get-instances *Relation*

Purpose:

The **get-instances** relation finds all instances that satisfy a given concept.

Syntax:

get-instances *concept* `&key` *direct-p asserted-p*

Arguments:

The *concept* argument is the concept, or the name of the concept, whose instances are to be found.

If the *direct-p* argument is **t**, then Loom returns those instances that belong to *concept*, but not to any of its subconcepts.

If the *asserted-p* argument is **t**, then only instances which have been explicitly asserted to belong to *concept* are returned.

Value:

The **get-instances** function returns a list of instances that satisfy *concept*.

Examples:

```
(defconcept Woman :is (:and Person Female))
(tellm (Person Joe) (Woman Sue) (:about Mary Person Female))
(get-instances 'Person) ⇒ (|I|JOE |I|SUE |I|MARY)
(get-instances (fc Person) :direct-p t) ⇒ (|I|JOE)
(get-instances 'Woman) ⇒ (|I|SUE |I|MARY)
(get-instances (fc Woman) :asserted-p t) ⇒ (|I|SUE)
```

See Also:

concept--instances (p. 35)
get-types (p. 141)
test-type-p (p. 280)

get-inverse-values *Function*

Purpose:

The **get-inverse-values** function returns a list of the instances which have a specified role filled by a specified instance.

Syntax:

get-inverse-values *instance role* `&key` *kb no-error-p*

Arguments:

The *instance* argument is an instance, or an instance identifier.

The *role* argument is a relation, or the name of a relation.

The *kb* argument is the knowledge base, or the symbol or string that names the knowledge base, where the lookup of *instance* and *role* begins. This defaults to the current knowledge base.

If the *no-error-p* argument is **t**, then error messages are suppressed.

Value:

If the retrieval is successful, this function returns a list containing the inverse filler or fillers of *role*. Otherwise, if *no-error-p* is set and *instance* (or *role*) cannot be found, the two values **nil** and **:missing-instance** (or **:missing-role**) are returned.

Examples:

```
(defrelation son :is (:and child (:range Male)))
(defrelation parent :is (:inverse child))
(defrelation wife :characteristics :single-valued)
(tellm (son Fred Joe) (parent Joe Mary) (wife Joe Sue))
(get-inverse-values 'Joe 'child) ⇒ (|I|FRED |I|MARY)
(get-inverse-values (fi Sue) (fr wife) :kb "USER-KB") ⇒ (|I|JOE)
(get-inverse-values 'Joe 'xxx :no-error-p t) ⇒ NIL :MISSING-ROLE
```

See Also:

get-values (p. 144)
retrieve (p. 246)

get-keys *Function*

Purpose:

The **get-keys** function returns the keys attached to a given concept. Each set of key/value pairs must be unique for each instance of the concept. If the keys are indexed, then Loom will flag occurrences of duplicate keys.

Syntax:

get-keys *concept* `&key` *direct-p*

Arguments:

The *concept* argument is concept, or the name of a concept.

If the *direct-p* argument is **t**, only keys directly defined on *concept* are returned. Otherwise, inherited keys are also returned.

Value:

The **get-keys** function returns a list of keys, where each key is identified by either a role name or a list of role names.

Examples:

```
(defconcept A :indices (r) :keys (r))
(defconcept B :is-primitive A :indices ((s1 s2)) :keys ((s1 s2)))
(get-keys 'A) ⇒ ((|R|R))
(get-keys 'B) ⇒ ((|R|R) (|R|S1 |R|S2))
(get-keys (fc B) :direct-p t) ⇒ ((|R|S1 |R|S2))
```

See Also:

defconcept (p. 49)
get-indices (p. 117)
get-roles (p. 135)

get-matching-instances *Function*

Purpose:

The **get-matching-instances** function returns all instances that (1) satisfy each member of a specified list of concepts, and (2) have role fillers matching each member of a specified list of role/fillers pairs.

Syntax:

get-matching-instances *concepts roleFillers*

Arguments:

The *concepts* argument is a (possibly empty) list of concepts, or names of concepts, which must be satisfied.

The *roleFillers* argument is a list of role/fillers pairs which must be matched. Each pair is a list containing (1) a relation or relation name, and (2) one or more instances, instance identifiers, or constants.

Value:

This function returns a list of matching instances.

Remarks:

Loom may use the first member of *concepts* as an instance generator, so the ordering of *concepts* is significant. For best performance, the first concept should be the one with the lowest estimated number of instances.

If no appropriate role indices exist, and if *concepts* is null, Loom issues a warning that it must scan the entire knowledge base to generate instances that might have the specified role fillers.

Examples:

```
(tellm (:about Joe Famous Writer
                    (wife Sue) (:filled-by kids-ages 3 4 5)))
(get-matching-instances
    '(Famous Writer) '((wife Sue) (kids-ages 3 4 5))) ⇒ (|I|JOE)
(get-matching-instances
    '(Famous) '((wife Sue) (kids-ages 3) (kids-ages 5))) ⇒ (|I|JOE)
(get-matching-instances
    `(,(fc Writer)) `((,(fr wife) ,(fi Sue)))) ⇒ (|I|JOE)
```

See Also:

get-indices (p. 117)
get-types (p. 141)
get-values (p. 144)
query (p. 227)
retrieve (p. 246)

get-method *Function*

Purpose:

The **get-method** function returns the method having a specified name and title.

Syntax:

get-method *action title* `&key` *no-error-p*

Arguments:

The *action* argument is an action, or the name of an action.

The *title* argument is a string which titles a method.

If the *no-error-p* argument is `t`, then no error is generated if Loom cannot find a method having the specified name and title.

Value:

The **get-method** function returns the method with the specified name and title. If there is none, **nil** is returned if *no-error-p* is set, and otherwise an error is generated.

Remarks:

The **get-** functions are faster and do less error-checking than the **find-** functions. In particular, **get-method** is much faster than **find-method**.

Examples:

```
(get-method 'move "move box") ⇒ |METHOD|MOVE-"move box"
(get-method 'open "open box" :no-error-p t) ⇒ |METHOD|OPEN-"open box"
```

See Also:

defaction (p. 46)
defmethod (p. 61)
find-method (p. 103)

get-name *Function*

Purpose:

The **get-name** function returns a name or identifier for any named Loom object. Named objects include concepts, sets, relations, Loom instances, CLOS instances of type **INSTANCE-IN-KNOWLEDGE-BASE**, actions, productions, and knowledge bases.

Syntax:

get-name *object*

Arguments:

The *object* argument is any object.

Value:

The name or identifier of *object* is returned.

Examples:

```
(get-name (fc Box)) ⇒ BOX
(get-name (fr on)) ⇒ ON
(get-name (fi Box3)) ⇒ BOX3
(get-name (find-action 'move)) ⇒ MOVE
(get-name (find-production 'full-box-rule)) ⇒ FULL-BOX-RULE
(get-name (find-kb "USER-KB")) ⇒ USER-KB
```

See Also:

concept--name (p. 36)
instance--identifier (p. 159)
object-name (p. 209)
relation--name (p. 232)

get-production *Function*

Purpose:

The **get-production** function returns the production having a specified name.

Syntax:

get-production *production* `&key` *no-error-p*

Arguments:

The *production* argument is a production rule, or the name of a production rule.

If the *no-error-p* argument is **t**, then no error is generated if Loom cannot find a production having the specified name.

Value:

The **get-production** function returns the production with the specified name. If there is none, **nil** is returned if *no-error-p* is set, and otherwise an error is generated.

Remarks:

The **get-** functions are faster and do less error-checking than the **find-** functions (such as **find-production**). Applications should use the **get-** functions for best performance.

Examples:

```
(get-production 'P2 ⇒ |PRODUCTION|P2
(get-production 'almost-full-container :no-error-p t) ⇒
    |PRODUCTION|ALMOST-FULL-CONTAINER
```

See Also:

defproduction (p. 64)
find-production (p. 105)

get-relation *Function*

Purpose:

The **get-relation** function returns the relation having a specified name.

Syntax:

get-relation *relation* `&key` *no-error-p kb*

Arguments:

The *relation* argument is a relation, or a symbol which names a relation. If it is a symbol, it may be an extended identifier having the form *kb^name*, where *kb* is the knowledge base from which the search is to proceed.

If the *no-error-p* argument is **t**, then no error is generated if Loom cannot find a relation having the specified name.

The *kb* argument is the knowledge base, or the name of the knowledge base, from which the lookup is to be performed. The relation will be found if it is interned in *kb* or any knowledge base above *kb*. This argument defaults to the current knowledge base.

Value:

The **get-relation** function returns the relation with the specified name. If there is none, **nil** is returned if *no-error-p* is set, and otherwise a continuable error is generated.

Remarks:

The **get-** functions are faster and do less error-checking than the **find-** functions (such as **find-relation**). Applications should use the **get-** functions for best performance.

Examples:

```
(get-relation '+) ⇒ |R|+
(get-relation 'user-kb^age :no-error-p t) ⇒ |R|AGE
(get-relation 'child :kb 'bottom-kb) ⇒ |R|CHILD
```

See Also:

find-relation (p. 106)
get-concept (p. 116)
get-instance (p. 118)

get-role *Function*

Purpose:

The **get-role** function gets a role for a specified relation on a specified concept, or **nil** if no such role can exist.

Syntax:

get-role *concept relation*

Arguments:

The *concept* argument is a concept, or the name of a concept.

The *relation* argument is a relation, or the name of a relation.

Value:

The **get-role** function returns a role object for the relation *relation* on the concept *concept*.

Examples:

```
(defconcept B
    :is (:and (:all r A) (:exactly 3 r) (:filled-by r Joe Fred)))
(defconcept C
    :is-primitive B
    :implies (:and (:some s A) (:at-most 2 s))
    :defaults (:filled-by s Sue))
(get-role 'C 'r) ⇒ (|ROLE|(R 3 3 A))
(get-role (fc C) (fr s)) ⇒ (|ROLE|(S 1 2 NIL)))
```

See Also:

concept--restrictions (p. 37)
get-role-cardinality (p. 129)
get-role-default-values (p. 130)
get-role-max-cardinality (p. 131)
get-role-min-cardinality (p. 132)
get-role-types (p. 133)
get-role-values (p. 134)
get-roles (p. 135)
restriction--relation (p. 244)

get-role-cardinality *Function*

Purpose:

The **get-role-cardinality** function returns the number of fillers of a given role on a given concept or instance, providing that the maximum and minimum number of such fillers is the same.

Syntax:

get-role-cardinality *conceptOrInstance relation*

Arguments:

The *conceptOrInstance* argument is a Loom concept or instance.

The *relation* argument is the relation whose upper and lower bounds are to be compared.

Value:

The **get-role-min-cardinality** function returns the minimum number retriction that applies to *conceptOrInstance* for *relation*, if that minimum is the same as the maximum. Otherwise, it returns **nil**.

Examples:

```
(defconcept B
    :is (:and (:all r A) (:exactly 3 r) (:filled-by r Joe Fred)))
(defconcept C
    :is-primitive B
    :implies (:and (:some s A) (:at-most 2 s))
    :defaults (:filled-by s Sue))
(tellm (C Mary))
(get-role-cardinality (fc C) (fr r)) ⇒ 3
(get-role-cardinality (fc C) (fr s)) ⇒ NIL
(get-role-cardinality (fi Mary) (fr r)) ⇒ 3
(get-role-cardinality (fi Mary) (fr s)) ⇒ NIL
```

See Also:

get-role (p. 128)
get-role-max-cardinality (p. 131)
get-role-min-cardinality (p. 132)

get-role-default-values *Function*

Purpose:

The **get-role-default-values** method finds the default fillers of a given role on a given concept.

Syntax:

get-role-default-values *concept relation*

Arguments:

The *concept* argument is a Loom concept.

The *relation* argument is the relation whose default fillers are to be found.

Value:

The **get-role-default-values** function returns the default fillers of the role on *concept* and *relation*.

Examples:

```
(defconcept B
    :is (:and (:all r A) (:exactly 3 r) (:filled-by r Joe Fred)))
(defconcept C
    :is-primitive B
    :implies (:and (:some s A) (:at-most 2 s))
    :defaults (:filled-by s Sue))
(get-role-default-values (fc C) (fr r)) ⇒ NIL
(get-role-default-values (fc C) (fr s)) ⇒ (|I|SUE)
```

See Also:

default (p. 48)
defconcept (p. 49)
get-role (p. 128)
get-role-values (p. 134)

get-role-max-cardinality *Method*

Purpose:

The **get-role-max-cardinality** method finds the maximum number of fillers of a given role on a given concept or instance.

Syntax:

get-role-max-cardinality *conceptOrInstance relation*

Arguments:

The *conceptOrInstance* argument is a Loom concept or instance.

The *relation* argument is the relation for which an upper bound is to be found.

Value:

The **get-role-max-cardinality** function returns the maximum number retriction that applies to *conceptOrInstance* for *relation*.

Examples:

```
(defconcept B
    :is (:and (:all r A) (:exactly 3 r) (:filled-by r Joe Fred)))
(defconcept C
    :is-primitive B
    :implies (:and (:some s A) (:at-most 2 s))
    :defaults (:filled-by s Sue))
(tellm (C Mary))
(get-role-max-cardinality (fc C) (fr r)) ⇒ 3
(get-role-max-cardinality (fc C) (fr s)) ⇒ 2
(get-role-max-cardinality (fi Mary) (fr r)) ⇒ 3
(get-role-max-cardinality (fi Mary) (fr s)) ⇒ 2
```

See Also:

get-role (p. 128)
get-role-min-cardinality (p. 132)
restriction--max (p. 242)

get-role-min-cardinality *Method*

Purpose:

The **get-role-min-cardinality** method finds the minimum number of fillers of a given role on a given concept or instance.

Syntax:

get-role-min-cardinality *conceptOrInstance relation*

Arguments:

The *conceptOrInstance* argument is a Loom concept or instance.

The *relation* argument is the relation for which a lower bound is to be found.

Value:

The **get-role-min-cardinality** function returns the minimum number retriction that applies to *conceptOrInstance* for *relation*.

Examples:

```
(defconcept B
    :is (:and (:all r A) (:exactly 3 r) (:filled-by r Joe Fred)))
(defconcept C
    :is-primitive B
    :implies (:and (:some s A) (:at-most 2 s))
    :defaults (:filled-by s Sue))
(tellm (C Mary))
(get-role-min-cardinality (fc C) (fr r)) ⇒ 3
(get-role-min-cardinality (fc C) (fr s)) ⇒ 1
(get-role-min-cardinality (fi Mary) (fr r)) ⇒ 3
(get-role-min-cardinality (fi Mary) (fr s)) ⇒ 1
```

See Also:

get-role (p. 128)
get-role-max-cardinality (p. 131)
restriction--min (p. 243)

get-role-types *Method*

Purpose:

The **get-role-types** method finds the type of the fillers of a given role on a given concept or instance. The type is a list of concepts.

Syntax:

get-role-types *conceptOrInstance relation*

Arguments:

The *conceptOrInstance* argument is a Loom concept or instance.

The *relation* argument is the relation whose fillers' type is to be found.

Value:

The **get-role-types** method returns the type (value-restriction) that applies to *conceptOrInstance* for *relation*.

Examples:

```
(defconcept B
    :is (:and (:all r A) (:exactly 3 r) (:filled-by r Joe Fred)))
(defconcept C
    :is-primitive B
    :implies (:and (:some s A) (:at-most 2 s))
    :defaults (:filled-by s Sue))
(tellm (C Mary))
(get-role-types (fc C) (fr r)) ⇒ (|C|A)
(get-role-types (fc C) (fr s)) ⇒ NIL
(get-role-types (fi Mary) (fr r)) ⇒ (|C|A)
(get-role-types (fi Mary) (fr s)) ⇒ (|C|THING)
```

See Also:

compute-value-restriction (p. 31)
get-role (p. 128)
restriction--value-restriction (p. 245)

get-role-values *Method*

Purpose:

For a given concept, the **get-role-values** method finds the strict fillers of a given role on that concept. For a given instance, it finds both the strict and default fillers of a given role on that instance.

Syntax:

get-role-values *conceptOrInstance relation*

Arguments:

The *conceptOrInstance* argument is a Loom concept or instance.

The *relation* argument is the relation whose fillers are to be found.

Value:

If *conceptOrInstance* is a concept, **get-role-values** returns the strict fillers of the role on *conceptOrInstance* and *relation*. If *conceptOrInstance* is an instance, **get-role-values** returns both the strict and default features.

Examples:

```
(defconcept B
    :is (:and (:all r A) (:exactly 3 r) (:filled-by r Joe Fred)))
(defconcept C
    :is-primitive B
    :implies (:and (:some s A) (:at-most 2 s))
    :defaults (:filled-by s Sue))
(tellm (C Mary))
(get-role-values (fc C) (fr r)) ⇒ (|I|JOE |I|FRED)
(get-role-values (fc C) (fr s)) ⇒ NIL
(get-role-values (fi Mary) (fr r)) ⇒ (|I|JOE |I|FRED)
(get-role-values (fi Mary) (fr s)) ⇒ (|I|SUE)
```

See Also:

get-role (p. 128)
get-role-default-values (p. 130)

get-roles *Function*

Purpose:

The **get-roles** function gets the local and inherited roles of a specified concept.

Syntax:

get-roles *concept*

Arguments:

The *concept* argument is a concept, or the name of a concept.

Value:

The **get-roles** function returns a list of the role objects attached to *concept*.

Remarks:

The printed representation of a role object is a list containing the role's relation, its minimum and maximum number of fillers, and the type (value-restriction) of the fillers.

The accessor **relation** can be used to get the relation which a retrieved role restricts.

The accessor **concept** can be used to get the concept to which a retrieved role is attached.

Examples:

```
(defconcept B
    :is (:and (:all r A) (:exactly 3 r) (:filled-by r Joe Fred)))
(defconcept C
    :is-primitive B
    :implies (:and (:some s A) (:at-most 2 s))
    :defaults (:filled-by s Sue))
(get-roles 'B) ⇒ (|ROLE|(R 3 3 A))
(get-roles (fc C)) ⇒ (|ROLE|(R 3 3 A) |ROLE|(S 1 2 NIL))
(setq role (first (get-roles (fc C))))
(relation role) ⇒ |R|R
(concept role) ⇒ |C|C
```

See Also:

concept--restrictions (p. 37)
get-indices (p. 117)
get-keys (p. 121)

get-role (p. 128)
list-role-names&values (p. 178)
relation--restrictions (p. 234)
restriction--concepts (p. 241)
Restriction-P (p. 240)

get-subconcepts *Function*

Purpose:

The **get-subconcepts** function finds all subconcepts, or only the direct subconcepts, of a given concept.

Syntax:

get-subconcepts *concept* `&key` *direct-p*

Arguments:

The *concept* argument is the concept, or the name of the concept, whose subconcepts are to be found.

If the *direct-p* argument is **t**, only the most general subconcepts are returned. Otherwise, all subconcepts, including *concept* itself, are returned.

Value:

This function returns a list of concepts.

Examples:

```
(defconcept Person :is-primitive Animal)
(defconcept Man :is-primitive Person)
(get-subconcepts 'Animal) ⇒ (|C|ANIMAL |C|PERSON |C|MAN)
(get-subconcepts (fc Animal) :direct-p t) ⇒ (|C|PERSON)
```

See Also:

direct-subconcepts (p. 80)
get-subrelations (p. 138)
get-superconcepts (p. 139)
subconcept-p (p. 268)
subconcepts (p. 269)

get-subrelations *Function*

Purpose:

The **get-subrelations** function finds all subrelations, or only the direct subrelations, of a given relation. If the argument to **get-subrelations** is a symbol, and no matching relation can be found, then Loom looks for a matching concept and returns the concept's subconcepts.

Syntax:

get-subrelations *relationOrConcept* `&key` *direct-p*

Arguments:

The *relationOrConcept* argument is the relation (or concept), or the name of the relation (or concept), whose subrelations (or subconcepts) are to be found.

If the *direct-p* argument is **t**, only the most specific subrelations (or subconcepts) are returned. Otherwise, all subrelations (or subconcepts), including *relationOrConcept* itself, are returned.

Value:

This function returns a list of relations, or a list of concepts.

Examples:

```
(defconcept Child :is-primitive Person)
(defrelation child :is-primitive relative)
(defrelation daughter :is-primitive child)
(get-subrelations 'relative) ⇒ (|R|RELATIVE |R|CHILD |R|DAUGHTER)
(get-subrelations (fr relative) :direct-p t) ⇒ (|R|CHILD)
(get-subrelations 'Person) ⇒ (|C|PERSON |C|CHILD)
```

See Also:

direct-subconcepts (p. 80)
get-subconcepts (p. 137)
get-superrelations (p. 140)
subconcept-p (p. 268)
subconcepts (p. 269)

get-superconcepts *Function*

Purpose:

The **get-superconcepts** function finds all superconcepts, or only the direct superconcepts, of a given concept.

Syntax:

get-superconcepts *concept* `&key` *direct-p*

Arguments:

The *concept* argument is the concept, or the name of the concept, whose superconcepts are to be found.

If the *direct-p* argument is **t**, only the most specific superconcepts are returned. Otherwise, all superconcepts, including *concept* itself, are returned.

Value:

This function returns a list of concepts.

Examples:

```
(defconcept Person :is-primitive Animal)
(defconcept Man :is-primitive Person)
(get-superconcepts 'Man) ⇒ (|C|MAN |C|PERSON |C|ANIMAL |C|THING)
(get-superconcepts (fc Man) :direct-p t) ⇒ (|C|PERSON)
```

See Also:

direct-superconcepts (p. 81)
get-subconcepts (p. 137)
get-superrelations (p. 140)
subconcept-p (p. 268)
superconcepts (p. 273)

get-superrelations *Function*

Purpose:

The **get-superrelations** function finds all superrelations, or only the direct superrelations, of a given relation. If the argument to **get-superrelations** is a symbol, and no matching relation can be found, then Loom looks for a matching concept and returns the concept's supers.

Syntax:

get-superrelations *relationOrConcept* `&key` *direct-p*

Arguments:

The *relationOrConcept* argument is the relation (or concept), or the name of the relation (or concept), whose superrelations (or superconcepts) are to be found.

If the *direct-p* argument is **t**, only the most specific superrelations (or superconcepts) are returned. Otherwise, all superrelations (or superconcepts), including *relationOrConcept* itself, are returned.

Value:

This function returns a list of relations, or a list of concepts.

Examples:

```
(defconcept Child :is-primitive Person)
(defrelation child :is-primitive relative)
(get-superrelations 'child) ⇒ (|R|CHILD |R|RELATIVE |R|BINARY-TUPLE)
(get-superrelations (fr child) :direct-p t) ⇒ (|R|RELATIVE)
(get-superrelations 'Person) ⇒ (|C|PERSON |C|THING)
```

See Also:

direct-superconcepts (p. 81)
get-subrelations (p. 138)
get-superconcepts (p. 139)
subconcept-p (p. 268)
superconcepts (p. 273)

get-types *Function*

Purpose:

The **get-types** function returns the asserted and inferred concepts in the type of a specified instance.

Syntax:

get-types *instance* `&key` *asserted-p direct-p*

Arguments:

The *instance* argument is an instance or instance identifier.

If the *asserted-p* argument is **t**, only concepts explicitly asserted by the user are returned.

If the *direct-p* argument is **t**, only the most specific asserted and inferred concepts are returned.

Value:

This function returns a list of concepts.

Examples:

```
(defconcept C :is (:and B (:at-least 1 r)))
(tellm (:about Joe A B (r 3)))
(get-types 'Joe) ⇒ (|C|A |C|B |C|C |C|THING)
(get-types (fi Joe) :asserted-p t) ⇒ (|C|A |C|B)
(get-types (fi Joe) :direct-p t) ⇒ (|C|A |C|C)
```

See Also:

add-type (p. 17)
get-instances (p. 119)
get-matching-instances (p. 122)
get-values (p. 144)
instance--asserted-concepts (p. 156)
instance--cached-concepts (p. 157)
instance--concepts (p. 158)
instance--type (p. 160)
retrieve (p. 246)

get-value *Function*

Purpose:

The **get-value** function gets the filler of a given single-valued role on a given instance.

Syntax:

get-value *instance role* `&key` *direct-p kb no-error-p*

Arguments:

The *instance* argument is an instance or an instance identifier.

The *role* argument is a single-valued relation, or the name of such a relation.

If the *direct-p* argument is **t**, then the filler of *role* is returned only if it was directly asserted to fill that role. A default filler, inferred filler, or filler of a subrole of *role* is not returned.

The *kb* argument is the knowledge base, or the symbol or string that names the knowledge base, where the lookup of *instance* and *role* begins. This defaults to the current knowledge base.

If the *no-error-p* argument is **t**, then error messages are suppressed.

Value:

If the retrieval is successful, the filler of *role* is returned. Otherwise, if *no-error-p* is set, then: (1) if *role* has multiple fillers, the first filler is returned, (2) if *instance* (or *role*) cannot be found, the two values **nil** and **:missing-instance** (or **:missing-role**) are returned.

Examples:

```
(defrelation r :characteristics :single-valued)
(defrelation s :is-primitive r :characteristics :single-valued)
(set-value 'Joe 's 'Sue)
(get-value 'Joe 'r) ⇒ |I|SUE
(get-value 'Joe 'r :direct-p t) ⇒ NIL
(set-value 'Joe 's '(3 4 5))
(get-value (fi Joe) (fr s) :kb "USER-KB") ⇒ (3 4 5)
(defrelation ss)
(set-values 'Joe 'ss '(3 4 5))
(get-value 'Joe 'ss :no-error-p t) ⇒ 3
(get-value 'Joe 'xx :no-error-p t) ⇒ NIL :MISSING-ROLE
```

See Also:

get-values (p. 144)
set-value (p. 261)

get-values *Function*

Purpose:

The **get-values** function gets the fillers of a given multiple-valued or single-valued role on a given instance.

Syntax:

get-values *instance role* `&key` *direct-p kb no-error-p*

Arguments:

The *instance* argument is an instance or an instance identifier.

The *role* argument is a relation, or the name of a relation.

If the *direct-p* argument is **t**, then the fillers of *role* are returned only if they were directly asserted to fill that role. Default fillers, inferred fillers, and fillers of subroles of *role* are not returned.

The *kb* argument is the knowledge base, or the symbol or string that names the knowledge base, where the lookup of *instance* and *role* begins. This defaults to the current knowledge base.

If the *no-error-p* argument is **t**, then error messages are suppressed.

Value:

If the retrieval is successful, this function returns a list containing the filler or fillers of *role*. Otherwise, if *no-error-p* is set and *instance* (or *role*) cannot be found, the two values **nil** and **:missing-instance** (or **:missing-role**) are returned.

Examples:

```
(clear-kb) (defrelation rr)
(defrelation ss :is (:and rr (:range A)))
(tellm (A Sue) (:about Joe (rr Sue) (:filled-by ss Mary '(3 4 5))))
(get-values 'Joe 'rr) ⇒ (|I|SUE |I|MARY (3 4 5))
(get-values 'Joe 'rr :direct-p t) ⇒ (|I|SUE)
(defrelation r :characteristics :single-valued)
(set-value 'Joe 'r 'Sue)
(get-values (fi Joe) (fr r)) ⇒ (|I|SUE)
(set-value 'Joe 'r '(3 4 5))
(get-values 'Joe 'r :kb "CL-USER-KB") ⇒ ((3 4 5))
(get-values 'Joe 'x :no-error-p t) ⇒ NIL :MISSING-ROLE
```

See Also:

add-value (p. 18)
get-inverse-values (p. 120)
get-types (p. 141)
list-role-names&values (p. 178)
remove-value (p. 236)
retrieve (p. 246)
set-values (p. 262)

get-version-string *Function*

Purpose:

The **get-version-string** function gives the version number of the Loom in which it is evaluated.

Syntax:

get-version-string

Arguments:

This function has no arguments.

Value:

The **get-version-string** function returns a string containing version numbers and dates of the Loom and Lisp currently running.

Examples:

```
(get-version-string) ⇒
    "LOOM version 2.0 (7/2/93), patch level 35 (10/26/93)
     Lucid Common Lisp Lucid Common Lisp,
     Development Environment Version 4.1, 12 October 1992"
```

See Also:

load-loom-patches (p. 185)

identifier--instance *Relation*

Purpose:

The **identifier--instance** relation finds the instance having a given identifier.

Syntax:

identifier--instance *symbol instance*

Domain:

The *symbol* argument is a symbol which identifies an instance.

Range:

The *instance* argument is the instance to be found from its identifier.

Examples:

```
(ask (identifier--instance 'Joe Joe)) ⇒ T
(retrieve ?x (identifier--instance 'Joe ?x) ⇒ (|I|JOE)
```

See Also:

fi (p. 92)
find-instance (p. 99)
instance--identifier (p. 159)
object-name (p. 209)

implies *Macro*

Purpose:

The **implies** macro specifies that a strict implication holds between two concepts, or between two relations. This macro allows new implications to be incrementally added to previously-defined concepts and relations.

Syntax:

implies *antecedent consequent*

Arguments:

The *antecedent* and *consequent* arguments are concept-forming expressions (see **defconcept** Remarks) or relation-forming expressions (see **defrelation** Remarks). The *antecedent* and *consequent* cannot both be relations defined with the **:satisfies** operator.

Value:

The antecedent concept or relation is returned.

Examples:

```
(implies Surfer Tan) ⇒ |C|SURFER
(implies Person (:exactly 2 has-arm)) ⇒ |C|PERSON
(implies husband (:and love honor cherish)) ⇒ |R|HUSBAND
(implies (:satisfies (?x ?y) (:and (Positive ?x) (Negative ?y)))
         attract) ⇒ |R|ATTRACT_1
(implies (:and Stress-Test (:some applied-to Ischemic))
         (:filled-by risk 'HIGH))) ⇒ |C|STRESS-TEST_1
```

See Also:

default (p. 48)
defconcept (p. 49)
defrelation (p. 66)
term-implies (p. 278)

in-kb *Macro*

Purpose:

The **in-kb** macro resets the current knowledge base at both compile-time and run-time.

Syntax:

in-kb *knowledgeBase*

Arguments:

The *knowledgeBase* argument is a symbol or string that names a knowledge base. The **in-kb** macro operates in a package-independent manner, so it is not necessary to specify the package in which a knowledge base name resides.

Value:

This macro returns **nil**.

Remarks:

Unlike **change-kb**, the **in-kb** macro can only appear at the top level of a Lisp program.

Examples:

```
(in-kb "USER-KB") ⇒ NIL
(in-kb 'bottom-kb) ⇒ NIL
(in-kb upper-structure-kb) ⇒ NIL
```

See Also:

change-kb (p. 26)
defkb (p. 59)

Incoherent *Concept*

Purpose:

The **Incoherent** predicate tests an instance for incoherence.

Syntax:

Incoherent *object*

Argument:

The *object* argument is any object.

Value:

The **Incoherent** predicate returns **t** if *object* is an incoherent instance.

Remarks:

There are many possible sources of incoherence. An instance may be incoherent because it instantiates an incoherent concept, or its type contains disjoint concepts, or it has asserted restrictions that conflict with concepts in its type, or it has role fillers whose number or type clash with restrictions that derive from the instance's type.

Examples:

```
(defconcept Bachelor :is (:and Man (:exactly 0 wife)))
(tellm (:about Joe Bachelor (wife Sue)))
(ask (Incoherent Joe)) ⇒ T
```

See Also:

disjoint-concepts-p (p. 82)
Incoherent-Concept (p. 151)

Incoherent-Concept *Concept*

Purpose:

The **Incoherent-Concept** predicate tests a concept for incoherence.

Syntax:

Incoherent-Concept *object*

Argument:

The *object* argument is any object.

Value:

The **Incoherent-Concept** predicate returns **t** if *object* is an incoherent concept.

Remarks:

There are many possible sources of incoherence. Most often, a concept is incoherent because it specializes two or more disjoint concepts, or it contains conflicting number or value restrictions.

Examples:

```
(defconcept Pet :partitions (($PET$ (Cat Dog)))) ...
(defconcept Cat-Dog :is (:and Cat Dog))
(retrieve ?x (Incoherent-Concept ?x)) ⇒ (|C|CAT-DOG)
```

See Also:

Concept (p. 32)
disjoint-concepts-p (p. 82)
Incoherent (p. 150)

initialize-instances *Function*

Purpose:

The **initialize-instances** function destroys all instances in all knowledge bases.

Syntax:

initialize-instances

Arguments:

This function takes no arguments.

Value:

The initialize-instances function returns the current knowledge-base.

Examples:

`(initialize-instances)` ⇒ `|K|USER-KB`

See Also:

clear-kb (p. 29)
destroy (p. 78)
initialize-network (p. 153)

initialize-network *Function*

Purpose:

The **initialize-network** function clears all knowledge bases by destroying the concepts, relations, instances, and behaviors that they contain. It also rebuilds the top-level knowledge base, **upper-structure-kb**, populating it with built-in concepts such as **Thing**, **Constant**, etc. Previously declared knowledge bases are preserved, but they are now empty.

Syntax:

initialize-network `&key` *destroy-kbs-p*

Arguments:

If the *destroy-kbs-p* argument is **t**, Loom clears and rebuilds **upper-structure-kb**, clears **bottom-kb**, and destroys all other knowledge bases.

Value:

The initialize-network function returns the current knowledge-base. If *destroy-kbs-p* causes knowledge bases to be destroyed (rather than just cleared), **upper-structure-kb** becomes the current knowledge base.

Examples:

```
(initialize-network) ⇒ |K|USER-KB
(initialize-network :destroy-kbs-p t) ⇒ |K|UPPER-STRUCTURE-KB
```

See Also:

clear-kb (p. 29)
delete-concept (p. 75)
destroy (p. 78)
initialize-instances (p. 152)

initialize-tasks *Function*

Purpose:

The **initialize-tasks** function clears all of the task queues, i.e., all scheduled tasks are deleted.

Syntax:

initialize-tasks

Arguments:

This function takes no arguments.

Value:

The **initialize-tasks** function returns **nil**.

Examples:

`(initialize-tasks)` ⇒ `NIL`

See Also:

list-tasks (p. 180)
perform (p. 214)
perform-task (p. 216)
schedule (p. 254)
schedule-task (p. 256)

instance-p *Function*

Purpose:

The **instance-p** function tests whether a given object is an instance of a Loom concept.

Syntax:

instance-p *object*

Arguments:

The *object* argument is any object.

Value:

The **instance-p** function returns **t** if *object* is a concept, relation, Loom instance, CLOS instance, or knowledge-base.

Examples:

```
(instance-p (fc Person)) ⇒ T
(instance-p (fr age)) ⇒ T
(instance-p (fi Joe)) ⇒ T
(instance-p (find-kb "USER-KB")) ⇒ T
(instance-p 3) ⇒ NIL
(instance-p 'Joe) ⇒ NIL
```

See Also:

concept-p (p. 33)
create (p. 42)
loom-instance-p (p. 187)
Loom-Thing (p. 188)
relation-p (p. 230)
tell (p. 275)
Thing (p. 281)

instance--asserted-concepts *Relation*

Purpose:

The **instance--asserted-concepts** relation finds all of the concepts that have been explicitly asserted about a given instance.

Syntax:

instance--asserted-concepts *instance concept*

Domain:

The *instance* argument is an instance.

Range:

The *concept* argument is a Loom concept.

Examples:

```
(tellm (:about Joe A B (r 3)))
(ask (instance--asserted-concepts Joe B)) ⇒ T
(retrieve ?x (instance--asserted-concepts Joe ?x)) ⇒ (|C|A |C|B)
```

See Also:

get-types (p. 141)
instance--cached-concepts (p. 157)
instance--concepts (p. 158)
instance--type (p. 160)

instance--cached-concepts *Relation*

Purpose:

The **instance--cached-concepts** relation finds the most specific concepts that have been asserted or inferred about a given instance.

Syntax:

instance--cached-concepts *instance concept*

Domain:

The *instance* argument is an instance.

Range:

The *concept* argument is a Loom concept.

Remarks:

The inferred concepts returned by this relation are those which have been recognized and cached by the forward-chainer, not those derivable by backward-chaining.

Examples:

```
(defconcept C :is (:and B (:at-least 1 r)))
(defconcept D :is (:some r 3)) ; backward-chaining (no primitiveness)
(tellm (:about Joe A B (r 3)))
(ask (instance--cached-concepts Joe C) ⇒ T
(retrieve ?x (instance--cached-concepts Joe ?x)) ⇒ (|C|A |C|C)
```

See Also:

get-types (p. 141)
instance--asserted-concepts (p. 156)
instance--concepts (p. 158)
instance--type (p. 160)

instance--concepts *Relation*

Purpose:

The **instance--concepts** relation finds the most specific concepts that have been asserted about a given instance, or have been inferred and cached by the forward-chainer, or are derivable by backward-chaining.

Syntax:

instance--concepts *instance concept*

Domain:

The *instance* argument is an instance.

Range:

The *concept* argument is a Loom concept.

Examples:

```
(defconcept C :is (:and B (:at-least 1 r)))
(defconcept D :is (:some r 3)) ; backward-chaining (no primitiveness)
(tellm (:about Joe A B (r 3)))
(ask (instance--concepts Joe D)) ⇒ T
(retrieve ?x (instance--concepts Joe ?x)) ⇒
    (|C|A |C|C |C|D |C|LOOM-THING)
```

See Also:

concept--instances (p. 35)
get-types (p. 141)
instance--asserted-concepts (p. 156)
instance--cached-concepts (p. 157)
instance--type (p. 160)
test-type-p (p. 280)

instance--identifier *Relation*

Purpose:

The **instance--identifier** relation finds the identifier of a given instance.

Syntax:

instance--identifier *instance symbol*

Domain:

The *instance* argument is any instance.

Range:

The **symbol** argument is the identifier of the specified instance.

Examples:

```
(ask (instance--identifier Joe 'Joe)) ⇒ T
(setq ?i (create nil nil))
(retrieve ?x (instance--identifier ?i ?x)) ⇒ (THING-1)
```

See Also:

create (p. 42)
identifier--instance (p. 147)
object-name (p. 209)
tell (p. 275)

instance--type *Relation*

Purpose:

The **instance--type** relation returns a single concept which is the unification of the most specific concepts that have been asserted or inferred about a given instance.

Syntax:

instance--type *instance concept*

Domain:

The *instance* argument is an instance.

Range:

The *concept* argument is a Loom concept.

Remarks:

The inferred concepts returned by this relation are those which have been recognized and cached by the forward-chainer, not those derivable by backward-chaining.

Examples:

```
(defconcept C :is (:and B (:at-least 1 r)))
(defconcept D :is (:some r 3)) ; backward-chaining (no primitiveness)
(tellm (:about Joe A B (r 3)))
(retrieve ?x (instance--type Joe ?x)) ⇒ (|C|A&B&C)
```

See Also:

get-types (p. 141)
instance--asserted-concepts (p. 156)
instance--cached-concepts (p. 157)
instance--concepts (p. 158)

Integer *Concept*

Purpose:

The **Integer** predicate tests whether a given object is an integer.

Syntax:

Integer *object*

Argument:

The *object* argument is any object.

Value:

The **Integer** predicate returns **t** if *object* is an integer.

Examples:

```
(ask (Integer 3)) ⇒ T
(ask (Integer 3.5)) ⇒ NIL
```

See Also:

Constant (p. 39)
Non-Loom-Thing (p. 207)
Number (p. 208)

isa-p *Relation*

Purpose:

The **isa-p** relation is a generalized subsumption predicate. It can be used to determine whether an object instantiates a given concept, or whether a concept or relation specializes some other concept or relation.

Syntax:

isa-p *object concept*

Domain:

The *object* argument is an instance, concept, or relation which may instantiate or specialize *concept*.

Range:

The *concept* argument is a concept or relation.

Examples:

```
(ask (isa-p Man Person)) ⇒ T
(ask (isa-p Joe Man)) ⇒ T
(ask (isa-p Joe Person)) ⇒ T
```

See Also:

concept--instances (p. 35)
get-types (p. 141)
instance--concepts (p. 158) **subconcept-p** (p. 268)
subconcepts (p. 269)
superconcepts (p. 273)
test-type-p (p. 280)

kill-task *Function*

Purpose:

The **kill-task** function removes a given task from its queue.

Syntax:

kill-task *task*

Arguments:

The *task* argument is the task which is to be dequeued. The task was originally created when a production fired, or when there was an explicit call to **schedule**.

Value:

This function returns *task* if the task was scheduled, and **nil** otherwise.

Remarks:

The **kill-task** function dequeues, but does not destroy, tasks. These tasks may be subsequently performed or rescheduled.

Examples:

```
(setq t1 (schedule (move (fi box1))))
(kill-task t1) ⇒ |TASK|(MOVE |I|BOX1)
```

See Also:

list-tasks (p. 180)
perform (p. 214)
schedule (p. 254)
schedule-task (p. 256)

Knowledge-Base *Concept*

Purpose:

The **Knowledge-Base** predicate tests whether a given object is a Loom knowledge base.

Syntax:

Knowledge-Base *object*

Arguments:

The *object* argument is any object.

Value:

This predicate returns **t** if *object* is a Loom knowledge base.

Examples:

```
(setq ?x (find-kb "USER-KB"))
(ask (Knowledge-base ?x)) ⇒ T
```

See Also:

defkb (p. 59)
instance-p (p. 155)
list-knowledge-bases (p. 174)
Loom-Thing (p. 188)

list-depend-ons *Function*

Purpose:

The **list-depend-ons** function lists all concepts that are directly or indirectly referenced by the definition of a given concept.

Syntax:

list-depend-ons *concept* `&key` *all-p*

Arguments:

The *concept* argument is the concept or relation from which a chain of dependencies is to be computed.

If the *all-p* argument is **t**, then dependencies are followed through implications and defaults, as well as through definitions.

Value:

The **list-depend-ons** function returns a list of the concepts and relations on which *concept* directly or indirectly depends.

Examples:

```
(defconcept A :is (:all r B))
(defconcept B :implies (:at-least 1 s))
(defrelation r :range C)
(list-depend-ons (fc A)) ⇒ (|C|B |R|R)
(list-depend-ons (fc A) :all-p t) ⇒ (|C|B |C|C |R|R |R|S)
```

See Also:

find-definitional-cycles (p. 98)
list-dependents (p. 166)

list-dependents *Function*

Purpose:

The **list-dependents** function lists all concepts whose definitions directly or indirectly reference a given concept.

Syntax:

list-dependents *concept* `&key` *all-p*

Arguments:

The *concept* argument is the concept or relation from which a chain of dependencies is to be computed.

If the *all-p* argument is **t**, then dependencies are followed through implications and defaults, as well as through definitions.

Value:

The **list-dependents** function returns a list of the concepts and relations which directly or indirectly depend on *concept*.

Examples:

```
(defconcept A)
(defconcept B :is-primitive A)
(defconcept C :is (:all r B))
(defconcept D :defaults (:at-least 1 r))
(defrelation r :range A)
(list-dependents (fc A)) ⇒ (|C|C |C|B)
(list-dependents (fc A) :all-p t) ⇒ (|R|R |C|D |C|C |C|B)
```

See Also:

find-definitional-cycles (p. 98)
list-depend-ons (p. 165)

list-features *Function*

Purpose:

The **list-features** function lists (1) all available Loom features, and (2) all features that are currently set. These features control the overall behavior of Loom, including the amount and type of dialogue that Loom has with the user.

Syntax:

list-features `&key` *dont-display-p*

Arguments:

If the *dont-display-p* argument is `t`, then the lists of available features and currently-set features are not displayed.

Value:

The **list-features** function returns a list of the features which are currently set.

Remarks:

The following Loom features are available:

- If the **:ask-about-redefinitions** feature is set, Loom checks with the user before redefining a concept. This feature is set by default.
- If the **:ask-for-concept-names** feature is set, Loom prompts the user for concept names whenever system-defined concepts are generated. The prompting is done after the new concepts have been completed. If the user declines to supply a name, the concept is assigned a system-generated name.
- If the **:ask-for-names-immediately** feature is set, Loom asks the user to name system-defined conjunction concepts as soon as they are created and classified.
- If the **:automatic-clos-classes** feature is set, a CLOS class is automatically created whenever **make-object** (or **tell** or **create** with the **:clos** flag set) is called to instantiate a concept that does not already have an associated CLOS class. If this feature is disabled, a CLOS class is not created unless the concept explicitly contains the **:clos-class** characteristic. The **:automatic-clos-classes** feature is set by default.
- If the **:clip-roles** feature is set, clipping is performed on all single-valued roles. That is, if two (or more) different instances are asserted to be fillers of a single-valued role, Loom automatically retracts all but the last assertion. This feature is set by default.

- If the **:clip-types** feature is set, clipping of instance types is enabled. That is, if an instance's type is asserted to contain two (or more) concepts which are disjoint, or would otherwise cause the instance to become incoherent, then Loom automatically retracts all but the last assertion.
- If the **:closed-world** feature is set, closed-world semantics are assumed for all concepts and relations.
- If the **:display-match-changes** feature is set, whenever a production memory match or instance classification occurs, Loom indicates which concepts have entered or exited the type of each affected instance, and which rete nodes the instance has entered or exited. This feature is set by default.
- If the **:domain-implies-role** feature is set when Loom creates a shadowing CLOS class for a concept C, then the new class will have slots corresponding to every relation whose domain is C.
- If the **:dont-name-system-defined-concepts** feature is set, Loom does not generate a name for system-defined concepts and relations (provided **:ask-for-names-immediately** is also enabled).
- If the **:emit-classify-dots** feature is set, a dot, minus, or plus is printed when a concept is classified, unclassified, or completed, respectively, and an exclamation point is printed when a concept is sealed (i.e., ready to participate in data manipulation operations). Also, the character **M** is printed when concepts merge during classification, **U** is printed when they unmerge, and **C** is printed when a CLOS class is created. This feature is set by default.
- If the **:emit-match-stars** feature is set, an asterisk is printed whenever the type of an instance changes. Also, Loom prints a **T** whenever a new type is created as a result of unifying a newly-matched concept with the concepts already in an instance's type. This feature is set by default.
- If the **:expand-system-defined-print-names** feature is set, the printed representation of an instance has the form |**I**|*identifier*(*concepts*), where *concepts* contains the names of the most-specific user-named concepts which the instance matches. If this feature is not set, type information is omitted from the printed representation of the instance.
- If the **:open-world** feature is set, open-world semantics are assumed for all concepts and relations. This feature is set by default.
- If the **:prefer-clos-instances** feature is set, Loom automatically creates a CLOS instance when **tell** or **create** is called. If there is no CLOS class corresponding to the concept being instantiated, Loom attempts to create one. If this effort fails, a standard Loom instance is created, and a warning is issued (unless the *clos-instance-p* argument to **create** was **:no-warning**).
- If the **:suppress-diehard-output** feature is set, all Loom informational and warning output is suppressed. For example, **save-kb** typeout, classify dots, type-

change messages, and incoherence warnings are all inhibited. It is advisable to set this feature when loading very large knowledge bases because of the overhead associated with lengthy typeout (especially when running under Emacs).

- If the **:warn-about-syntax-corrections** feature is set, a warning is issued whenever a syntax error is detected in a concept definition or in a query. This feature is set by default.
- If the **:warn-on-nonprimitive-clos-classes** feature is set, a warning is issued whenever a CLOS class is created to shadow a non-primitive concept. This feature is set by default.

Examples:

```
(list-features) ⟶
    Possible features:
      (:OPEN-WORLD :CLOSED-WORLD :PREFER-CLOS-INSTANCES ...)
    Current features:
      (:OPEN-WORLD :EMIT-MATCH-STARS ...)
(list-features :dont-display-p t) ⇒
    (:OPEN-WORLD :EMIT-MATCH-STARS ...)
```

See Also:

reset-features (p. 239)
set-features (p. 258)
unset-features (p. 286)
with-default-features (p. 297)
with-feature-changes (p. 298)
with-features (p. 301)

list-instances *Function*

Purpose:

The **list-instances** function lists all instances in all knowledge bases.

Syntax:

list-instances

Arguments:

The **list-instances** function has no arguments.

Value:

This function returns a list of Loom and/or CLOS instances.

Remarks:

Concepts and relations are filtered out of the returned instances.

Examples:

`(list-instances)` ⇒ `(|I|JOE #<Person #X17A3CAE> ...)`

See Also:

do-instances (p. 83)
find-instance (p. 99)
initialize-instances (p. 152)
list-kb (p. 172)

list-inverse-role-names&values *Function*

Purpose:

The **list-inverse-role-names&values** function lists each instance and role in which a specified Loom instance appears as a value.

Syntax:

list-inverse-role-names&values *instance*

Arguments:

The *instance* argument is an instance.

Value:

The **list-inverse-role-names&values** function returns an assoc list whose members each consist of a role name and a list of instances which fill the inverse of that role.

Examples:

```
(tellm (child Fred Joe) (husband Sue Joe) (child Mary Joe))
(list-inverse-role-names&values (fi Joe)) ⇒
    ((CHILD |I|FRED |I|MARY) (HUSBAND |I|SUE))
```

See Also:

concept--restrictions (p. 37)
get-roles (p. 135)
get-values (p. 144)
list-role-names&values (p. 178)
show (p. 264)

list-kb *Macro*

Purpose:

The **list-kb** macro lists all objects in specified partitions of a knowledge base.

Syntax:

list-kb `&optional` *knowledgeBase* `&key` *partitions sort-p*

Arguments:

The *knowledgeBase* argument is the knowledge base to be listed, or a symbol or string which names that knowledge base. It defaults to the current knowledge base.

The *partitions* argument is a keyword or list of keywords specifying the types of objects to be retrieved. Available partitions are **:instances**, **:concepts**, **:relations**, and **:behaviors**. If this argument is not supplied, all partitions are used.

The *sort-p* argument is **t** or **:sort-each-partition**. If it is **t**, all objects in the specified partitions are sorted together. If it is **:sort-each-partition**, the specified partitions are sorted separately.

Value:

A list of knowledge base objects is returned.

Remarks:

Though concepts and relations are interned in the **:instances** partition (as well as their own partitions), these objects are not included when the **:instances** partition is listed.

Examples:

```
(list-kb)
(list-kb "USER-KB") ⇒ (|I|JOE |C|Z |C|A |R|R |ACTION|MOVE)
(list-kb :sort-p t) ⇒ (|C|A |I|JOE |ACTION|MOVE |R|R |C|Z)
(list-kb :partitions (:concepts :relations)
         :sort-p :sort-each-partition) ⇒ (|C|A |C|Z |R|R)
```

See Also:

defkb (p. 59)
do-instances (p. 83)
find-definitional-cycles (p. 98)
list-instances (p. 170)

list-knowledge-bases (p. 174)
list-merged-concepts (p. 175)
list-system-defined-concepts (p. 179)
list-unclassified-concepts (p. 182)
list-undefined-concepts (p. 183)

list-knowledge-bases *Function*

Purpose:

The **list-knowledge-bases** function lists all Loom knowledge bases, including the built-in knowledge bases **upper-structure-kb** and **bottom-kb**.

Syntax:

list-knowledge-bases

Value:

This function returns a list of knowledge bases.

Examples:

`(list-knowledge-bases)` ⇒ `(|K|BOTTOM-KB |K|USER-KB |K|UPPER-STRUCTURE-KB)`

See Also:

change-kb (p. 26)
current-kb (p. 45)
defkb (p. 59)
list-kb (p. 172)
use-loom (p. 291)

list-merged-concepts *Function*

Purpose:

The **list-merged-concepts** function lists all concepts into which other concepts have merged.

Syntax:

list-merged-concepts `&optional` *kb*

Arguments:

The *kb* argument specifies which knowledge base to search. If this argument is not supplied, all knowledge bases are searched.

Value:

The **list-merged-concepts** function returns a list of merged concepts and/or relations.

Examples:

```
(defconcept A :is B)
(defconcept C :is (:exactly 1 r))
(defconcept D :is (:and (:at-least 1 r) (:at-most 1 r)))
(list-merged-concepts) ⇒ (|C|B |C|C)
(list-merged-concepts (find-kb "USER-KB")) ⇒ (|C|B |C|C)
```

See Also:

list-kb (p. 172)

list-methods *Function*

Purpose:

The **list-methods** function lists all the methods which implement a given action.

Syntax:

list-methods *action*

Arguments:

The *action* argument is the action whose methods are to be listed.

Value:

The **list-methods** function returns a list of methods.

Examples:

```
(list-methods (fb move)) ⇒
    (|METHOD|MOVE-:UNTITLED |METHOD|MOVE-"move box")
```

See Also:

defaction (p. 46)
defmethod (p. 61)
find-method (p. 103)
print-methods (p. 226)

list-productions *Function*

Purpose:

The **list-productions** function lists the productions in a given knowledge base, or in all knowledge bases.

Syntax:

list-productions `&key` *kb*

Arguments:

The *kb* argument is the knowledge base, or the string or symbol that names the knowledge base, whose productions are to be listed. If this argument is not supplied, all knowledge bases are searched.

Value:

The **list-productions** function returns a list of productions.

Examples:

```
(list-productions) ⇒ (|PRODUCTION|P1 |PRODUCTION|P2)
(list-productions :kb "USER-KB") ⇒ (|PRODUCTION|P3)
```

See Also:

defproduction (p. 64)
find-production (p. 105)

list-role-names&values *Function*

Purpose:

The **list-role-names&values** function lists the name and value of each role of a specified Loom or CLOS instance.

Syntax:

list-role-names&values *instance*

Arguments:

The *instance* argument is an instance.

Value:

The **list-role-names&values** function returns an assoc list whose members each consist of a role name and a list of instances which fill that role.

Examples:

```
(tellm (parent Joe Fred) (wife Joe Sue) (parent Joe Mary))
(list-role-names&values (fi Joe)) ⇒
    ((PARENT |I|FRED |I|MARY) (WIFE |I|SUE))
```

See Also:

concept--restrictions (p. 37)
get-roles (p. 135)
get-values (p. 144)
list-inverse-role-names&values (p. 171)
show (p. 264)

list-system-defined-concepts *Function*

Purpose:

The **list-system-defined-concepts** function lists all concepts that have been created by Loom in the course of classification and matching.

Syntax:

list-system-defined-concepts `&optional` *kb*

Arguments:

The *kb* argument specifies which knowledge base to search. If this argument is not supplied, all knowledge bases are searched.

Value:

This function returns a list of concepts and/or relations which have been defined by Loom.

Examples:

```
(defconcept A :implies B)
(tellm (:about Joe C D))
(list-system-defined-concepts) ⇒ (|C|A&B |C|C&D)
(list-system-defined-concepts (find-kb "USER-KB")) ⇒ (|C|A&B |C|C&D)
```

See Also:

list-kb (p. 172)
list-undefined-concepts (p. 183)

list-tasks *Function*

Purpose:

The **list-tasks** function returns the tasks in a specified queue, or in all queues.

Syntax:

list-tasks `&key` *priorities*

Arguments:

The *priorities* argument specifies the queue(s) to be listed. This argument is either a priority keyword or a list of such keywords. The currently available priorities are **:high** and **:low**.

Value:

This function returns a list of queues, where each queue contains a priority keyword followed by the tasks scheduled at that priority.

Examples:

```
(list-tasks) ⇒
    ((:HIGH |TASK|(T1 |I|JOE) |TASK|(T2 |I|SUE))
     (:LOW |TASK|(T3 |I|JOE)))
(list-tasks :priorities :low) ⇒
    ((:LOW |TASK|(T3 |I|JOE)))
(list-tasks :priorities '(:high :low)) ⇒
    ((:HIGH |TASK|(T1 |I|JOE) |TASK|(T2 |I|SUE))
     (:LOW |TASK|(T3 |I|JOE)))
```

See Also:

initialize-tasks (p. 154)
kill-task (p. 163)
schedule (p. 254)
schedule-task (p. 256)

list-tuples *Function*

Purpose:

The **list-tuples** function returns the tuples asserted to a given n-ary relation.

Syntax:

list-tuples *relation*

Arguments:

The *relation* argument is a relation.

Value:

This function returns a list of tuples. The length of each tuple is equal to the arity of *relation*.

Examples:

```
(list-tuples (fr distance)) ⇒ ((|I|LA |I|SFO 400) (|I|LA |I|NYC 3000))
```

See Also:

ask (p. 20)
retrieve (p. 246)
tell (p. 275)

list-unclassified-concepts *Function*

Purpose:

The **list-unclassified-concepts** function lists all concepts that have been defined or referenced, but have not yet been classified.

Syntax:

list-unclassified-concepts

Arguments:

The **list-unclassified-concepts** function takes no arguments.

Value:

This function returns a list of unclassified concepts and/or relations.

Remarks:

The Loom classifier does not classify a concept until it is defined and all the concepts referenced in its definition are classified.

Examples:

```
(defconcept A :is-primitive B :implies C)
(list-unclassified-concepts) ⇒ (|C|A |C|B |C|C)
```

See Also:

destroy-unclassified-concepts (p. 79)
find-definitional-cycles (p. 98)
list-kb (p. 172)
list-undefined-concepts (p. 183)

list-undefined-concepts *Function*

Purpose:

The **list-undefined-concepts** function lists all concepts that have been referenced in definitions but have not themselves been defined.

Syntax:

list-undefined-concepts `&optional` *kb*

Arguments:

The *kb* argument specifies which knowledge base to search. If this argument is not supplied, all knowledge bases are searched.

Value:

This function returns a list of undefined concepts and/or relations.

Examples:

```
(defconcept A :is-primitive B :implies C)
(list-undefined-concepts) ⇒ (|C|B |C|C)
(list-undefined-concepts (find-kb "USER-KB")) ⇒ (|C|B |C|C)
```

See Also:

destroy-unclassified-concepts (p. 79)
find-definitional-cycles (p. 98)
list-kb (p. 172)
list-system-defined-concepts (p. 179)
list-unclassified-concepts (p. 182)

load-kb *Function*

Purpose:

The **load-kb** function restores a knowledge base from Loom source definitions saved in a file. This file may contain definitions of concepts, relations, instances, and behaviors. Normally, the file will have been created by **save-kb**.

Syntax:

load-kb *knowledgeBase* `&key` *path-name*

Arguments:

The *knowledgeBase* argument is the knowledge base to be restored, or a symbol or string which names that knowledge base.

The *path-name* argument is a string that specifies the file to be used for restoring *knowledgeBase*. This defaults to the file that was associated with *knowledgeBase* when that knowledge base was defined. If no file was associated with *knowledgeBase*, a warning is issued.

Value:

The pathname of the loaded knowledge base is returned.

Examples:

```
(defkb test-kb nil :path-name "test-kb.lisp")
(load-kb 'test-kb) ⇒ "test-kb.lisp"
(load-kb "TEST-KB" :path-name "dump.lisp") ⇒ "dump.lisp"
```

See Also:

defkb (p. 59)
list-kb (p. 172)
save-kb (p. 252)

load-loom-patches *Function*

Purpose:

The **load-loom-patches** function loads the latest patches for the currently running Loom.

Syntax:

load-loom-patches `&key` *source-if-newer-p*

Arguments:

If the *source-if-newer-p* argument is **t**, then the source patch file will be loaded if it is newer than the binary. Otherwise, the compiled patches will be loaded. This argument defaults to **t**.

Value:

This function returns the pathname of the patch file that was loaded.

Examples:

`(load-loom-patches)` ⇒ `#P"/auto/loom/BIN-LCL4.1/patch/loom-patches.sbin"`

See Also:

get-version-string (p. 146)

loom-concept *Slot*

Purpose:

The **loom-concept** slot of a CLOS instance contains a pointer to the instance's concept—that is, it points back to the Loom concept that is shadowed by the instance's CLOS class.

Syntax:

loom-concept *instance*

Arguments:

The *instance* argument is a CLOS instance.

Value:

This function returns a shadowed Loom concept.

Examples:

```
(defconcept Adult)
(setq ?i (make-object 'Adult))
(loom-concept ?i) ⇒ |C|ADULT
```

See Also:

make-object (p. 189)

loom-instance-p *Function*

Purpose:

The **loom-instance-p** function tests whether a given object is a Loom instance.

Syntax:

loom-instance-p *object*

Arguments:

The *object* argument is any object.

Value:

The **loom-instance-p** function returns **t** if *object* is a concept, relation, or other Loom instance.

Examples:

```
(loom-instance-p (fc Person)) ⇒ T
(loom-instance-p (fr age)) ⇒ T
(loom-instance-p (fi Joe)) ⇒ T
```

See Also:

concept-p (p. 33)
instance-p (p. 155)
relation-p (p. 230)

Loom-Thing *Concept*

Purpose:

The **Loom-Thing** predicate succeeds if its argument is an instance of a Loom-defined class.

Syntax:

Loom-Thing *object*

Arguments:

The *object* argument is any object.

Value:

The **Loom-Thing** predicate returns **t** if *object* is a concept, relation, Loom instance, CLOS instance, or knowledge base.

Examples:

```
(ask (Loom-Thing Thing)) ⇒ T
(create 'Joe nil)
(ask (Loom-Thing Joe)) ⇒ T
(setq ?ci (create 'Mary (fc Thing) :clos-instance-p t))
(ask (Loom-Thing ?ci)) ⇒ T
(setq ?kb (find-kb "USER-KB"))
(ask (Loom-Thing ?kb)) ⇒ T
(ask (Loom-Thing 3)) ⇒ NIL
```

See Also:

Concept (p. 32)
concept-p (p. 33)
instance-p (p. 155)
Knowledge-Base (p. 164)
loom-instance-p (p. 187)
Non-Loom-Thing (p. 207)
Relation (p. 229)
relation-p (p. 230)
Thing (p. 281)

make-object *Macro*

Purpose:

The **make-object** macro makes a CLOS instance of a concept—that is, it creates an instance of a CLOS class which shadows a Loom concept, and it establishes an index link between that concept and the instance.

Syntax:

make-object *class* `&body` *initargs* `&key` *identifier kb*

Arguments:

The *class* argument is a CLOS class, or the name of CLOS class, which shadows a Loom concept.

The *initargs* argument consists of zero or more slot initialization arguments which are passed unevaluated to the CLOS **make-instance** function.

The *identifier* argument is the symbol which identifies the new CLOS instance. This argument is ignored unless the instance is of type **INSTANCE-IN-KNOWLEDGE-BASE** (see Remarks below).

The *kb* argument is the knowledge base in which the instance's identifier is to be interned. This argument is ignored if the instance is not of type **INSTANCE-IN-KNOWLEDGE-BASE** (see Remarks below).

Value:

The **make-object** macro returns a newly-created CLOS instance.

Remarks:

In applications which do not require full forward-chaining inference, the use of CLOS instances rather than Loom instances may increase performance significantly.

CLOS instances can be created by **tell** and **create**, as well as by **make-object**. The behavior of **tell** and **create** depends on the setting of the **:prefer-clos-instances** feature.

If the **:automatic-clos-classes** feature is set, a shadowing CLOS class is created for a given concept the first time that **make-object** is called to instantiate that concept. (Calls to **tell** and **create** may also result in class creation.) If **:automatic-clos-classes** is not set, a class will not be created unless the concept has the **:clos-class** characteristic. If the **:warn-on-nonprimitive-clos-classes** feature is set, a warning is issued when a class is created for a non-primitive concept.

The shadowing CLOS class normally has the same name as the concept being instantiated, but this can be overridden by the **:class-name** option in the concept's definition. Similarly, the **:existing-class-name** option can be used to shadow the concept with an existing CLOS class rather than a new one.

Loom supports automatic revision of CLOS classes. When a Loom concept is redefined, its shadowing class (if any) is likewise redefined the next time that the concept is instantiated.

If a type specifier for a non-existent CLOS class occurs in the parameter list of a **defmethod**, Loom will dynamically create the class if there is a concept with the same name as the type specifier and it is possible to create a shadowing class for that concept.

When a shadowing class is created for concept C, the superclasses of the new class are computed from the most-specific shadowed superconcepts of C. A slot C.R is allocated on C if R is neither **:read-only** nor inherited from a superclass of C, and one of the following is true:

- relation R appears in the **:roles** option of C's definition;
- concept C restricts relation R, i.e., C has a restriction such as **(:at-least 1 R)** or **(:all R B)**;
- the **:domain-implies-role** feature is set, and the domain of R is C (except when C is **Thing**);
- the above rules specify a slot D.R, and D specializes C, and there is no CLOS class on D or between D and C.

If role R of concept C is defined to have a default filler, then slot C.R on the new class is initialized to that filler.

When a concept is defined to have mixin classes (see **defconcept**), it is automatically shadowed, and its instances inherit properties of the specified mixins as follows:

- The **INSTANCE-WITH-CONCEPTS** mixin allows new concepts to be added to the type of a CLOS instance.
- The **INSTANCE-WITH-DYNAMIC-SLOTS** mixin allows new roles and role values to be attached to a CLOS instance.
- The **INSTANCE-IN-MATCH-NETWORK** mixin lets instances be matched in production rule memory, so that applicable productions will fire when the state of the database changes. A subclass of **INSTANCE-WITH-CONCEPTS**.
- The **INSTANCE-IN-KNOWLEDGE-BASE** mixin causes a CLOS instance to be interned into the instances partition of a Loom knowledge base.
- The **INSTANCE-WITH-INVERSES** mixin supports inference based on knowledge about the inverses of the roles of a CLOS instance.

- The **INSTANCE-WITH-EVERYTHING** mixin adds the functionality of all the mixins listed above.

CLOS instances may be asserted to and queried with the standard **tell** and **retrieve** operators.

Examples:

```
(defconcept Woman :mixin-classes INSTANCE-WITH-CONCEPTS)
(setq ?i (make-object 'Woman)) ⇒ #<Woman #X1777546>
(tellm (American ?i))
(retrieve ?x (American ?x)) ⇒ (#<Woman #X1777546>)

(defconcept Person :mixin-classes INSTANCE-WITH-DYNAMIC-SLOTS)
(setq ?i (make-object (find-class 'Person))) ⇒ #<Person #X177FFDE>
(tellm (age ?i 33))
(retrieve ?x (:and (Person ?x) (age ?x 33))) ⇒ #<Person #X177FFDE>

(defconcept Doctor :mixin-classes INSTANCE-IN-MATCH-NETWORK
                   :implies Rich)
(make-object 'Doctor) ⇒ #<Doctor #X178B8EE>
(retrieve ?x (Rich ?x)) ⇒ (#<Doctor #X178B8EE>)

(defconcept Dog :mixin-classes INSTANCE-IN-KNOWLEDGE-BASE)
(make-object 'Dog :identifier 'Fido) ⇒ |i|FIDO
(find-instance' Fido) ⇒ |i|FIDO

(defconcept Man :roles wife :mixin-classes INSTANCE-WITH-INVERSES)
(defrelation husband :is (:inverse wife))
(setq ?i (make-object 'Man)) ⇒ #<Man #X179ABCE>
(tellm (wife ?i Mary))
(retrieve ?x (husband Mary ?x)) ⇒ (#<Man #X179ABCE>)
```

See Also:

change-object (p. 27)
create (p. 42)
finalize-definitions (p. 94)
loom-concept (p. 186)
tell (p. 275)
unmake-object (p. 285)

max *Relation*

Purpose:

The **max** relation is an aggregation operator that finds the largest of a set of numbers.

Syntax:

max *setOfNumbers number*

Domain:

The *setOfNumbers* argument is a list of integer and/or real numbers.

Range:

The *number* argument is the largest member of *setOfNumbers*.

Examples:

```
(ask (max '(3 5 4) 5)) ⇒ T
(tellm (:about Joe (:filled-by friend Fred Sue))
       (age Fred 30) (age Sue 40))
(retrieve ?x (max (age (friend Joe)) ?x)) ⇒ (40)
```

See Also:

count (p. 41)
min (p. 198)
sum (p. 272)
> (p. 16)

Max-Restriction-P *Concept*

Purpose:

The **Max-Restriction-P** predicate tests whether a given object is a Loom restriction feature that specifies the maximum number of fillers of a role.

Syntax:

Max-Restriction-P *object*

Argument:

The *object* argument is any object.

Value:

The **Max-Restriction-P** predicate returns **t** if *object* is a number restriction specifying an upper bound.

Examples:

```
(defconcept B :is (:and A (:at-most 2 r)))
(setq ?res (first (retrieve ?x (concept--restrictions B ?x))))
(ask (Max-Restriction-P ?res)) ⇒ T
(retrieve ?max
    (:for-some (?res)
               (:and (concept--restrictions B ?res)
                     (Max-Restriction-P ?res)
                     (restriction--max ?res ?max)))) ⇒ (2)
```

See Also:

concept--restrictions (p. 37)
get-role-max-cardinality (p. 131)
get-roles (p. 135)
restriction--max (p. 242)
Restriction-P (p. 240)

member-of *Relation*

Purpose:

The **member-of** relation tests whether an object is a member of a set.

Syntax:

member-of *object set*

Domain:

The *object* argument is any object.

Range:

The *set* argument is a list of objects.

Examples:

```
(ask (member-of 5 '(3 5 4))) ⇒ T
(tellm (:about Fred (wife Pam)
       (:about Mary (:filled-by friends Joe Pam Sue))
(ask (member-of (wife Fred) (friends Mary))) ⇒ T
```

See Also:

members (p. 195)
same-as (p. 251)
subset (p. 270)

members *Relation*

Purpose:

The **members** relation is a function that returns the members of a set.

Syntax:

members *set object*

Domain:

The *set* argument is a list of objects.

Range:

The *object* argument is any object.

Examples:

```
(ask (members '(3 5 4) 5)) ⇒ T
(tellm (:about Mary (:filled-by friends Joe Pam Sue)))
(retrieve ?x (friends Mary ?x)) ⇒ (|I|JOE |I|PAM |I|SUE)
(retrieve ?x (members (friends Mary) ?x)) ⇒ (|I|JOE |I|PAM |I|SUE)
```

See Also:

member-of (p. 194)
same-as (p. 251)
subset (p. 270)

Meta-Concept *Concept*

Purpose:

The **Meta-Concept** predicate tests whether a given object is a Loom concept or relation.

Syntax:

Meta-Concept *object*

Argument:

The *object* argument is any object.

Value:

The **Meta-Concept** predicate returns **t** if *object* is either a concept or a relation.

Examples:

```
(ask (Meta-Concept Meta-Concept)) ⇒ T
(retrieve ?x (Meta-Concept ?x)) ⇒ (|C|PERSON |R|AGE ...)
```

See Also:

Concept (p. 32)
Loom-Thing (p. 188)
meta-concept-p (p. 197)
Relation (p. 229)
Thing (p. 281)

meta-concept-p *Function*

Purpose:

The **meta-concept-p** function tests whether a given object is a Loom concept or relation.

Syntax:

meta-concept-p *object*

Arguments:

The *object* argument is any object.

Value:

The **meta-concept-p** function returns **t** if *object* is either a concept or relation.

Examples:

```
(meta-concept-p (fc Person)) ⇒ T
(meta-concept-p (fr age)) ⇒ T
```

See Also:

concept-p (p. 33)
defconcept (p. 49)
defrelation (p. 66)
Meta-Concept (p. 196)
relation-p (p. 230)

min *Relation*

Purpose:

The **min** relation is an aggregation operator that finds the smallest of a set of numbers.

Syntax:

min *setOfNumbers number*

Domain:

The *setOfNumbers* argument is a list of integer and/or real numbers.

Range:

The *number* argument is the smallest member of *setOfNumbers*.

Examples:

```
(ask (max '(4 3 5) 3)) ⇒ T
(tellm (:about Joe (:filled-by friend Fred Sue))
       (age Fred 30) (age Sue 40))
(retrieve ?x (min (age (friend Joe)) ?x)) ⇒ (30)
```

See Also:

count (p. 41)
max (p. 192)
sum (p. 272)
< (p. 16)

Min-Restriction-P *Concept*

Purpose:

The **Min-Restriction-P** predicate tests whether a given object is a Loom restriction feature that specifies the minimum number of fillers of a role.

Syntax:

Min-Restriction-P *object*

Argument:

The *object* argument is any object.

Value:

The **Min-Restriction-P** predicate returns **t** if *object* is a number restriction specifying a lower bound.

Examples:

```
(defconcept B :is (:and A (:at-least 1 r)))
(setq ?res (first (retrieve ?x (concept--restrictions B ?x))))
(ask (Min-Restriction-P ?res)) ⇒ T
(retrieve ?min
    (:for-some (?res)
               (:and (concept--restrictions B ?res)
                     (Min-Restriction-P ?res)
                     (restriction--min ?res ?min)))) ⇒ (1)
```

See Also:

concept--restrictions (p. 37)
get-role-min-cardinality (p. 132)
get-roles (p. 135)
restriction--min (p. 243)
Restriction-P (p. 240)

most-general-concepts *Function*

Purpose:

The **most-general-concepts** function finds the most general concepts in a given list of concepts.

Syntax:

most-general-concepts *conceptList*

Arguments:

The *conceptList* argument is a list of concepts or relations.

Value:

This function returns a list of concepts or relations.

Examples:

```
(setq concepts (list (fr sister) (fr relative) (fr son) (fr child)))
(most-general-concepts concepts) ⇒ (|R|RELATIVE)
```

See Also:

most-specific-concepts (p. 201)
subconcept-p (p. 268)

most-specific-concepts *Function*

Purpose:

The **most-specific-concepts** function finds the most specific concepts in a given list of concepts.

Syntax:

most-specific-concepts *conceptList*

Arguments:

The *conceptList* argument is a list of concepts or relations.

Value:

This function returns a list of concepts or relations.

Examples:

```
(setq concepts (list (fr sister) (fr relative) (fr son) (fr child)))
(most-specific-concepts concepts) ⇒ (|R|SISTER |R|SON)
```

See Also:

most-general-concepts (p. 200)
subconcept-p (p. 268)

n-ary-tuple *Relation*

Remarks:

The relation **n-ary-tuple** is at the top of the n-ary relation hierarchy, that is, it subsumes all other relations having arity three or more.

Examples:

```
(defrelation distance :arity 3)
(ask (subconcepts n-ary-tuple distance)) ⇒ T
```

See Also:

binary-tuple (p. 23)
Relation (p. 229)
subconcepts (p. 269)
Thing (p. 281)

name--concept *Relation*

Purpose:

The **name--concept** relation finds the concept having a given name.

Syntax:

name--concept *symbol concept*

Domain:

The **symbol** argument is a symbol which names a concept.

Range:

The *concept* argument is the concept to be found from its name.

Examples:

```
(ask (name--concept 'Person Person)) ⇒ T
(retrieve ?x (name--concept 'Person ?x)) ⇒ (|C|PERSON)
```

See Also:

concept--name (p. 36)
fc (p. 89)
find-concept (p. 96)
object-name (p. 209)

name--relation *Relation*

Purpose:

The **name--relation** relation finds the relation having a given name.

Syntax:

name--relation *symbol relation*

Domain:

The **symbol** argument is a symbol which names a relation.

Range:

The *concept* argument is the concept to be found from its name.

Examples:

```
(ask (name--relation 'age age)) ⇒ T
(retrieve ?x (name--relation 'age ?x)) ⇒ (|R|AGE)
```

See Also:

find-relation (p. 106)
fr (p. 111)
object-name (p. 209)
relation--name (p. 232)

new-time-point *Function*

Purpose:

The **new-time-point** function causes Loom to advance to a new knowledge base state. It runs the instance classifier to recompute the type of each instance which may have been affected by updates that occurred since the last time point. It also fires productions and performs any tasks that have been scheduled.

Syntax:

new-time-point

Arguments:

This function takes no arguments.

Value:

The value of this function is the number of the new state. Due to production side-affects, the number system may have advanced more than one state before the call returns. The last state is returned.

Remarks:

The instance classifier cannot be run until all concepts have been classified.

The instance-classification cycle has the highest priority, followed by the production-firing cycle, and then by the task-performing cycle.

The functions **createm**, **tellm**, **forgetm**, and **destroym** each call **new-time-point** just before they return.

If the **:emit-match-stars** feature is set, an asterisk is printed whenever the type of an instance changes. If the **:display-match-changes** feature is set when an update occurs, Loom indicates which concepts have entered or exited the type of each affected instance, and which instances have entered or exited production memory (rete nodes).

Examples:

`(new-time-point)` ⇒ `34`

See Also:

add-type (p. 17)
copy-instance (p. 40)
createm (p. 42)
defmethod (p. 61)

defproduction (p. 64)
destroym (p. 78)
find-or-create-instance (p. 104)
forgetm (p. 109)
perform (p. 214)
remove-type (p. 235)
schedule (p. 254)
tellm (p. 275)

Non-Loom-Thing *Concept*

Purpose:

The **Non-Loom-Thing** predicate succeeds if its argument is not an instance of a Loom-defined class.

Syntax:

Non-Loom-Thing *object*

Arguments:

The *object* argument is any object.

Value:

The **Non-Loom-Thing** predicate returns **t** if *object* is not a concept, relation, Loom instance, CLOS instance, or knowledge base.

Examples:

```
(ask (Non-Loom-Thing 3)) ⇒ T
(ask (Non-loom-Thing '(cons a b))) ⇒ T
(ask (Non-loom-Thing #'cons)) ⇒ T
(ask (Non-loom-Thing Thing)) ⇒ NIL
```

See Also:

Loom-Thing (p. 188)
Thing (p. 281)

Number *Concept*

Purpose:

The **Number** predicate tests whether a given object is a number.

Syntax:

Number *object*

Argument:

The *object* argument is any object.

Value:

The **Number** predicate returns **t** if *object* is a number.

Examples:

```
(ask (Number 3)) ⇒ T
(ask (Number 3.5)) ⇒ T
```

See Also:

Constant (p. 39)
Integer (p. 161)
Non-Loom-Thing (p. 207)

object-name *Method*

Purpose:

The **object-name** method returns a name or identifier for any named Loom object. Named objects include concepts, sets, relations, Loom instances, CLOS instances of type **INSTANCE-IN-KNOWLEDGE-BASE**, actions, productions, and knowledge bases.

Syntax:

object-name *object*

Arguments:

The *object* argument is any object.

Value:

The name or identifier of *object* is returned.

Examples:

```
(object-name (fc Box)) ⇒ BOX
(object-name (fr on)) ⇒ ON
(object-name (fi Box3)) ⇒ BOX3
(object-name (find-action 'move)) ⇒ MOVE
(object-name (find-production 'full-box-rule)) ⇒ FULL-BOX-RULE
(object-name (find-kb "USER-KB")) ⇒ USER-KB
```

See Also:

concept--name (p. 36)
get-name (p. 125)
identifier--instance (p. 147)
instance--identifier (p. 159)
name--concept (p. 203)
name--relation (p. 204)
relation--name (p. 232)
rename-concept (p. 238)

pb *Macro*

Purpose:

The **pb** macro prints the definition of a given action, method, or production. It should be used interactively rather than in application code.

Syntax:

pb *behavior* `&optional` *title*

Arguments:

The *behavior* argument is a Lisp form which either names a behavior (an action, method, or production), or evaluates to a behavior or behavior name. If it is a symbol, special characters are interpreted as in **pc**, and package prefixes are optional because the search extends across packages.

The *title* argument is a string which titles a method.

Value:

The return value is **nil**.

Remarks:

If *behavior* and *title* are supplied, **pb** searches for a method of *behavior* having the specified title. Otherwise, it searches for an action or production named *behavior*.

Examples:

```
(pb move) ⟶
    (DEFACTION MOVE (?X)
        :FILTERS (:MOST-SPECIFIC)
        :MISSING-METHOD :WARNING)
(pb bottom-kb^move "move box") ⟶
    (DEFMETHOD MOVE (?X)
        :TITLE "move box"
        :SITUATION (BOX ?X)
        :RESPONSE ((MOVE-BOX ?X)))
(pb *-container) ⟶
    (DEFPRODUCTION ALMOST-FULL-CONTAINER
        :WHEN (:DETECTS (:AND (CONTAINER ?X) (>= (PCT-FULL ?X) 90)))
        :PERFORM (MOVE ?X))
```

See Also:

fb (p. 87)
pc (p. 212)
show (p. 264)

pc *Macro*

Purpose:

The **pc** macro prints the definition of a given concept. It should be used interactively rather than in application code.

Syntax:

pc *datum*

Arguments:

The *datum* argument is a Lisp form. It may be an extended identifier having the form *kb^name*, where *kb* is the knowledge base from which the search starts. It may also be an identifier beginning and/or ending with the wildcard character *, in which case the search proceeds until the first matching object is found. It is not necessary to specify the package in which *datum* resides since the search is package-independent. If *datum* is a symbol, Loom first tries to locate the object named by that symbol. If this fails, or if *datum* is a list, Loom evaluates *datum* and searches for the object corresponding to the resulting value.

Value:

The return value is **nil**.

Remarks:

The definitions printed by **pc** reflect inferences made by Loom during classification and matching. Unlike the output of **save-kb**, these definitions may differ considerably from the original source definitions supplied by the user.

Loom provides the **pc** macro as a development tool. Because *datum* may be evaluated at compile time, **pc** cannot be reliably used in application code.

Examples:

```
(pc Thing) ⟶
    (DEFCONCEPT THING
        :CHARACTERISTICS :CLOS-CLASS
        :KB UPPER-STRUCTURE-KB)
(pc *cycle) ⟶
    (DEFCONCEPT UNICYCLE :IS (:AND CYCLE (:EXACTLY 1 HAS-WHEEL)))
(pc (first concepts)) ⟶
    (DEFCONCEPT PERSON :IS-PRIMITIVE ANIMAL)
```

See Also:

fc (p. 89)
find-concept (p. 96)
pb (p. 210)
pi (p. 217)
po (p. 219)
pr (p. 222)
show (p. 264)

perform *Macro*

Purpose:

The **perform** macro is used to create tasks. A task is the application of a specific Loom action (implemented as a set of methods) to one or more objects. The **perform** macro executes the task immediately, rather than queuing it for later execution (as **schedule** does).

Syntax:

perform (*actionName* `&rest` *arguments*) `&key` *returnOption*

Arguments:

The *actionName* argument is the name of the action which is to be applied to *arguments*. The procedure actually performed depends on which of the methods attached to *actionName* have their situation patterns satisfied by *arguments*.

The *arguments* argument contains zero or more variables or constants. Keyword arguments are supported, but optional and rest arguments are not.

The *returnOption* argument specifies what kind of value the action should return. The options are:

- The **:value** option indicates that **perform** should return the value of the last method that was applied, or **nil** if there were no applicable methods.
- The **:success-p** option indicates that **perform** should return **t** if (1) one or more methods were applied and none of them called **fail**, or (2) there were no applicable methods and the **:no-op** missing-method option was supplied in the definition of *actionName*. Otherwise, **nil** should be returned.
- The **:task** option indicates that a task should be created and immediately performed, and that **perform** should return the completed task object.

If *returnOption* is not supplied, it defaults to **:value**.

Value:

The return value of **perform** is determined by the *returnOption* argument (see above).

Remarks:

Tasks may be performed as a result of production firings, as well as explicit calls to **perform**.

Examples:

```
(perform (weigh-box box)) ⇒ 44
(perform (inspect-box box shipper) :return :success-p) ⇒ T
(perform (notify-shipper (fi Acme)) :return :task) ⇒
    |TASK|(NOTIFY-SHIPPER |I|ACME)
```

See Also:

defmethod (p. 61)
defproduction (p. 64)
fail (p. 86)
new-time-point (p. 205)
perform-task (p. 216)
schedule (p. 254)
schedule-task (p. 256)

perform-task *Function*

Purpose:

The **perform-task** function performs a specified Loom task.

Syntax:

perform-task *task returnOption*

Arguments:

The *task* argument is the task to be performed. The task was originally created when a production fired, or when there was an explicit call to **perform** or **schedule**.

The *returnOption* argument specifies what kind of value the task should return. The options are **:value**, **:success-p**, or **:task**. See **perform** for a description of these options.

Value:

The return value of **perform-task** is determined by the *returnOption* argument (see above).

Remarks:

Tasks may be performed as a result of a call to **perform-task** or, more commonly, as a result of an explicit or implicit call to **new-time-point**. After instance classification and production firing, **new-time-point** performs all queued tasks in order of their priority. In the current version of Loom, the highest-priority task is the one at the front of the highest-priority queue.

Examples:

```
(setq t1 (schedule (move (fi box3))))
(perform-task t1 :value) ⇒ |I|BOX3
(perform-task t1 :success-p) ⇒ T
(perform-task t1 :task) ⇒ |TASK|(MOVE |I|BOX3)
```

See Also:

perform (p. 214)
schedule (p. 254)
schedule-task (p. 256)

pi *Macro*

Purpose:

The **pi** macro prints an assertion that contains all the facts currently known about a particular instance. This macro should be used interactively rather than in application code.

Syntax:

pi *datum* `&key` *assertions-only-p*

Arguments:

The *datum* argument is a Lisp form. It may be an extended identifier having the form *kb^name*, where *kb* is the knowledge base from which the search starts. It may also be an identifier beginning and/or ending with the wildcard character *, in which case the search proceeds until the first matching object is found. It is not necessary to specify the package in which *datum* resides since the search is package-independent. If *datum* is a symbol, Loom first tries to locate the object named by that symbol. If this fails, or if *datum* is a list, Loom evaluates *datum* and searches for the object corresponding to the resulting value.

If the *assertions-only-p* argument is **t**, **pi** behaves like **save-kb** and only prints assertions explicitly made by the user. Otherwise, the output reflects inferences made by Loom, and it may differ considerably from the user's original input.

Value:

The return value is **nil**.

Remarks:

Loom provides the **pi** macro as a development tool. Because *datum* may be evaluated at compile time, **pi** cannot be reliably used in application code.

Examples:

```
(defconcept Man :is-primitive Person)
(defrelation wife :domain Man)
(tellm (:about Joe Person (wife Sue)) (:about Joe-Bob (age 39)))
(setq inst (tell (:about Sue (:exactly 2 child))))
(pi Joe) ⟶ (TELL (:ABOUT JOE MAN (WIFE SUE)))
(pi Joe :assertions-only-p t) ⟶ (TELL (:ABOUT JOE PERSON (WIFE SUE)))
(pi Joe-*) ⟶ (TELL (:ABOUT JOE-BOB THING (AGE 39)))
(pi inst) ⟶ (TELL (:ABOUT SUE THING (:EXACTLY 2 CHILD)))
```

See Also:

fi (p. 92)
find-instance (p. 99)
pc (p. 212)
show (p. 264)

po *Function*

Purpose:

The **po** function prints the definition of an arbitrary Loom object.

Syntax:

po *object*

Arguments:

The *object* argument is a concept, relation, instance, action, method, production, or knowledge base.

Value:

The **po** function returns **nil**.

Remarks:

Like **pc** and related functions, the definitions generated by **po** reflect inferences made by Loom during classification and recognition.

Examples:

```
(po (find-concept 'Box)) ⟶
    (DEFCONCEPT BOX
        :IS-PRIMITIVE CONTAINER)
(po (find-kb 'user-kb)) ⟶
    (DEFKB USER-KB UPPER-STRUCTURE-KB
        :PACKAGE "USER")
```

See Also:

pb (p. 210)
pc (p. 212)
pi (p. 217)
pprint-object (p. 221)
pr (p. 222)
show (p. 264)

power-level *Function*

Purpose:

The **power-level** function allows the user to turn off (or down) certain expensive types of inferencing in order to gain a possibly substantial increase in classifier performance.

Syntax:

power-level `&optional` *level*

Arguments:

The *level* argument is either **:low**, **:medium**, or **:high**. The default level is **:high**, which maximizes inferencing at the expense of speed. Setting the level to **:low** causes some inferences to be missed, but these are relatively seldom-used and losing them should not affect most applications.

Value:

This function returns the *level* argument if one was supplied, and otherwise it returns the current power level.

Remarks:

It is difficult to precisely characterize the types of inferencing affected by the **power-level**. In practice, the user can compare the results of different power settings and choose a lower level if it affords a reasonable speedup without sacrificing required deductions.

Examples:

```
(power-level) ⇒ :HIGH
(power-level :low) ⇒ :LOW
```

See Also:

new-time-point (p. 205)

pprint-object *Function*

Purpose:

The **pprint-object** function pretty-prints the definition of an object, or returns a form representing the object's definition.

Syntax:

pprint-object *object* `&optional` *stream*

Arguments:

The *object* argument is any object.

The *stream* argument is a Lisp stream. This defaults to ***standard-output***.

Value:

This function pretty-prints a definition for *object* to *stream*, and returns **nil**. If *stream* is explicitly set to **nil**, then nothing is printed, and an s-expression representing the object's definition is returned.

Examples:

```
(pprint-object (fc Box)) ⟶
    (defconcept Box
      :is-primitive Container)
(pprint-object (fc Box) nil) ⇒
    (DEFCONCEPT BOX :IS-PRIMITIVE CONTAINER)
```

See Also:

po (p. 219)
show (p. 264)

pr *Macro*

Purpose:

The **pr** macro prints the definition of a given relation. It should be used interactively rather than in application code.

Syntax:

pr *datum*

Arguments:

The *datum* argument is a Lisp form. It may be an extended identifier having the form *kb*^*name*, where *kb* is the knowledge base from which the search starts. It may also be an identifier beginning and/or ending with the wildcard character *, in which case the search proceeds until the first matching object is found. It is not necessary to specify the package in which *datum* resides since the search is package-independent. If *datum* is a symbol, Loom first tries to locate the object named by that symbol. If this fails, or if *datum* is a list, Loom evaluates *datum* and searches for the object corresponding to the resulting value.

Value:

The return value is **nil**.

Remarks:

The definitions printed by **pr** reflect inferences made by Loom during classification and matching. Unlike the output of **save-kb**, these definitions may differ considerably from the original source definitions supplied by the user.

Loom provides the **pr** macro as a development tool. Because *datum* may be evaluated at compile time, **pc** cannot be reliably used in application code.

Examples:

```
(pr binary-tuple) —→
    (DEFRELATION BINARY-TUPLE
        :CHARACTERISTICS (:CLOSED-WORLD :SYMMETRIC))
(pr weight*) —→
    (DEFRELATION WEIGHT-IN-KILOS
        :IS (:AND WEIGHT (:RANGE KILOS)))
(pr (first relations)) —→
    (DEFRELATION SIBLING
        :IS-PRIMITIVE RELATIVE)
```

See Also:

find-relation (p. 106)
fr (p. 111)
pc (p. 212)
show (p. 264)

predecessor *Relation*

Purpose:

The **predecessor** relation returns the element which precedes a specified member of an ordered set.

Syntax:

predecessor *set member1 member2*

Domains:

The *set* argument is a concept defined as an ordered set.

The *member1* argument is a numeric or symbolic member of *set*.

Range:

The *member2* argument is the numeric or symbolic member which immediately precedes *member1* in *set*.

Examples:

```
(defset Level :is (:the-ordered-set 'LOW 'MEDIUM 'HIGH))
(ask (predecessor Level 'HIGH 'MEDIUM)) ⇒ T
(retrieve ?x (predecessor Level 'MEDIUM ?x)) ⇒ (LOW)
```

See Also:

concept--instances (p. 35)
defset (p. 72)
successor (p. 271)
< (p. 16)

Primitive-P *Concept*

Purpose:

The **Primitive-P** predicate tests whether a given concept or relation is marked primitive.

Syntax:

Primitive-P *concept*

Argument:

The *concept* argument is a concept or relation.

Value:

The **Primitive-P** predicate returns **t** if *concept* is a primitive concept or relation.

Examples:

```
(defconcept Person)
(ask (Primitive-P Person)) ⇒ T
(retrieve ?x (:and (Concept ?x) (Primitive-P ?x))) ⇒ (|C|PERSON ...)
```

See Also:

defconcept (p. 49)
defrelation (p. 66)
Single-Valued-P (p. 266)
User-Defined-P (p. 293)

print-methods *Function*

Purpose:

The **print-methods** function prints the definitions of all the methods which implement a given action.

Syntax:

print-methods *action*

Arguments:

The *action* argument is the action whose method definitions are to be printed.

Value:

This function returns **nil**.

Examples:

```
(print-methods (fb move)) ⟶
    (DEFMETHOD MOVE (?OBJ ?LOC)
        :RESPONSE ((MOVE-OBJECT ?OBJ ?LOC)))
    (DEFMETHOD MOVE (?OBJ ?LOC)
        :TITLE "move box"
        :SITUATION (BOX ?OBJ)
        :RESPONSE ((MOVE-BOX ?OBJ ?LOC)))
```

See Also:

defaction (p. 46)
defmethod (p. 61)
find-method (p. 103)
list-methods (p. 176)

query *Function*

Purpose:

The **query** function provides a functional interface for running arbitrary queries. It is identical to **retrieve** except that its arguments are evaluated and free variables must be declared special.

Syntax:

query *variables expression*

Arguments:

The *variables* argument is either **nil**, a single query variable, or a list of query variables (where a query variable is a symbol beginning with the character '?'). If *variables* is **nil**, **query** has the effect of an **ask** rather than a **retrieve**.

The *expression* variable is an arbitrary expression in the Loom query language. (See **retrieve** Remarks for a summary of the query syntax). Any free variables, (i.e., externally bound query variables) that appear in *expression* must be declared special.

Value:

If *variables* is null, **t** is returned if *expression* is provably true, and **nil** otherwise. If *variables* is a symbol, **query** returns a list of values satisfying *expression*. If *variables* is a list, a list of tuples is returned, where each tuple contains as many values as there are variables.

Remarks:

It is preferable to use **retrieve** whenever possible because **retrieve** is generally much faster than **query**.

Examples:

```
(tellm (:about Joe Writer (wife Sue) (age 37)))
(query nil '(Writer Joe)) ⇒ T
(query '?x '(Writer ?x)) ⇒ (|I|JOE)
(query '(?x ?y) '(:and (Writer ?x) (wife ?x ?y))) ⇒ ((|I|JOE |I|SUE))
(let ((?V 37))
  (declare (special ?V))
  (query '(?x ?y) '(:and (Writer ?x) (wife ?x ?y) (age ?x ?V)))) ⇒
      ((|I|JOE |I|SUE))
```

See Also:

get-matching-instances (p. 122)
get-types (p. 141)
get-values (p. 144)
retrieve (p. 246)

Relation *Concept*

Purpose:

The **Relation** predicate tests whether a given object is a Loom relation.

Syntax:

Relation *object*

Argument:

The *object* argument is any object.

Value:

The **Relation** predicate returns **t** if *object* is a binary or n-ary relation.

Examples:

```
(ask (Relation age)) ⇒ T
(retrieve ?x (Relation ?x)) ⇒ (|R|AGE |R|BINARY-TUPLE |R|= ...)
```

See Also:

binary-tuple (p. 23)
Concept (p. 32)
defrelation (p. 66)
Loom-Thing (p. 188)
n-ary-tuple (p. 202)
relation-p (p. 230)
Thing (p. 281)

relation-p *Function*

Purpose:

The **relation-p** function tests whether a given object is a Loom relation.

Syntax:

relation-p *object*

Arguments:

The *object* argument is any object.

Value:

The **relation-p** function returns **t** if *object* is a binary or n-ary relation.

Examples:

```
(relation-p (fr age)) ⇒ T
(relation-p (fr =)) ⇒ T
```

See Also:

concept-p (p. 33)
defrelation (p. 66)
instance-p (p. 155)
Loom-Thing (p. 188)
meta-concept-p (p. 197)
Relation (p. 229)

relation--domain *Relation*

Purpose:

This relation finds the domain of a given relation.

Syntax:

relation--domain *relation concept*

Domain:

The *relation* argument is a binary or n-ary relation.

Range:

The *concept* argument is the concept which is the domain of *relation*. In the case of an n-ary *relation*, *concept* is a list of concepts. The list has length one less than the arity of *relation*.

Examples:

```
(retrieve ?x (relation--domain wife ?x)) ⇒ (|C|MAN)
(retrieve ?x (relation--domain + ?x)) ⇒ ((|C|NUMBER |C|NUMBER))
(ask (relation--domain relation--domain Relation)) ⇒ T
```

See Also:

defrelation (p. 66)
relation--range (p. 233)

relation--name *Relation*

Purpose:

The **relation--name** relation finds the name of a given relation.

Syntax:

relation--name *relation symbol*

Domain:

The *relation* argument is the relation whose name is to be found.

Range:

The **symbol** argument is the symbol which names *relation*.

Examples:

```
(retrieve ?x (relation--name age ?x)) ⇒ (AGE)
(ask (relation--name binary-tuple 'binary-tuple)) ⇒ T
```

See Also:

define-relation (p. 58)
defrelation (p. 66)
name--relation (p. 204)

relation--range *Relation*

Purpose:

This relation finds the range of a given relation.

Syntax:

relation--range *relation concept*

Domain:

The *relation* argument is a binary or n-ary relation.

Range:

The *concept* argument is the concept which is the range of *relation*.

Examples:

```
(retrieve ?x (relation--range wife ?x)) ⇒ (|C|WOMAN)
(retrieve ?x (relation--range + ?x)) ⇒ (|C|NUMBER)
(ask (relation--range relation--range Concept)) ⇒ T
```

See Also:

defrelation (p. 66)
relation--domain (p. 231)

relation--restrictions *Relation*

Purpose:

The **relation--restrictions** relation finds all the restriction objects on a given relation. (See **defconcept** Remarks for a summary of the restriction constructs available in Loom.)

Syntax:

relation--restrictions *relation restriction*

Domain:

The *relation* argument is the restricted relation in the returned features.

Range:

The *restriction* argument is a Loom restriction feature. This may be a strict or implied feature, that is, it may have been declared in the **:is** or the **:implies** clause of a concept definition.

Examples:

```
(defconcept B :is (:and A (:at-least 1 r)) :implies (:all r B))
(defconcept C :implies (:filled-by r 3))
(retrieve ?x (relation--restrictions r ?x)) ⇒
    (|FTR|(:filled-by R (3)) |FTR|(:all R B) |FTR|(:at-least 1 R B))
```

See Also:

concept--restrictions (p. 37)
get-roles (p. 135)
restriction--relation (p. 244)

remove-type *Function*

Purpose:

The **remove-type** function removes a concept from the type of a given instance.

Syntax:

remove-type *instance concept* `&key` *kb no-error-p*

Arguments:

The *instance* argument is an instance, or an instance identifier.

The *concept* argument is the concept, or the name of the concept, being removed.

The *kb* argument is the knowledge base, or the symbol or string that names the knowledge base, where the lookup of *instance* and *concept* begins. This defaults to the current knowledge base.

If the *no-error-p* argument is **t**, then error messages are suppressed.

Value:

If the update is successful, the newly-removed concept is returned. If *no-error-p* is set and *instance* (or *concept*) cannot be found, the two values **nil** and **:missing-instance** (or **:missing-concept**) are returned.

Remarks:

This function does not cause *instance* to be reclassified.

Examples:

```
(tellm (:about Joe Rich Married))
(remove-type 'Joe 'Rich) ⇒ |C|RICH
(remove-type (fi Joe) (fc Married) :kb "USER-KB") ⇒ |C|MARRIED
(remove-type 'Joe 'xxx :no-error-p t) ⇒ NIL :MISSING-CONCEPT
(new-time-point)
(pi Joe) ⟶ (TELL (:ABOUT JOE THING))
```

See Also:

add-type (p. 17)
forget (p. 109)
new-time-point (p. 205)
remove-value (p. 236)

remove-value *Function*

Purpose:

The **remove-value** function removes a value from a given role on a given instance.

Syntax:

remove-value *instance role value* `&key` *kb no-error-p*

Arguments:

The *instance* argument is an instance or an instance identifier.

The *role* argument is a relation, or the name of a relation.

The *value* argument is the value to be removed from the fillers of *role*. If *value* is a list, the list is interpreted as a single role filler. If *value* is a symbol, the value is coerced to an instance unless the the range of *role* indicates that it should be interpreted as a constant.

The *kb* argument is the knowledge base, or the symbol or string that names the knowledge base, where the lookup of *instance* and *role* begins. This defaults to the current knowledge base.

If the *no-error-p* argument is **t**, then error messages are suppressed.

Value:

If the update is successful, the newly removed value is returned. If *no-error-p* is set and *instance* (or *role*) cannot be found, the two values **nil** and **:missing-instance** (or **:missing-role**) are returned.

Examples:

```
(defrelation rr)
(set-values 'Joe 'rr '(Mary Sue (3 4 5))) ⇒ (|I|MARY |I|SUE (3 4 5))
(remove-value 'Joe 'rr 'Mary) ⇒ |I|MARY
(remove-value 'Joe 'rr '(3 4 5)) ⇒ (3 4 5)
(get-values 'Joe 'rr) ⇒ (|I|SUE)
(defrelation ss :range Symbol)
(add-value 'Joe 'ss 'Mary) ⇒ MARY
(remove-value 'Joe 'ss 'Mary) ⇒ MARY
(remove-value 'Joe 'xx 'Mary :no-error-p t) ⇒ NIL :MISSING-ROLE
```

See Also:

add-value (p. 18)
forget (p. 109)
remove-type (p. 235)

rename-concept *Function*

Purpose:

The **rename-concept** function renames an existing concept. All references to the old name are replaced by references to the new name.

Syntax:

rename-concept *newName concept* `&key` *type*

Arguments:

The *newName* argument is a symbol or string.

The *concept* argument is the concept or relation to be renamed, or the current name of that concept or relation.

The *type* argument is the keyword **:concept** or **:relation**. It is needed when *concept* is a symbol which names both a concept and relation.

Value:

The **rename-concept** function returns the renamed concept.

Examples:

```
(rename-concept "WEIGHT-IN-KILOS" (fr weight)) ⇒ |R|WEIGHT-IN-KILOS
(rename-concept 'Hue 'Tone :type :concept) ⇒ |C|HUE
```

See Also:

define-concept (p. 57)
define-relation (p. 58)
object-name (p. 209)

reset-features *Function*

Purpose:

The **reset-features** function resets all of the Loom features to their default values.

Syntax:

reset-features

Arguments:

This function takes no arguments.

Value:

The **reset-features** function returns a list containing the the Loom features that are set by default.

Remarks:

The default features are: **:ask-about-redefinitions**, **:automatic-clos-classes**, **:clip-roles**, **:display-match-changes**, **:emit-classify-dots**, **:emit-match-stars**, **:open-world**, **:warn-about-syntax-corrections**, **:warn-on-nonprimitive-clos-classes**

Examples:

`(reset-features)` ⇒ `(:OPEN-WORLD :EMIT-MATCH-STARS ...)`

See Also:

list-features (p. 167)
set-features (p. 258)
unset-features (p. 286)
with-default-features (p. 297)

Restriction-P *Concept*

Purpose:

The **Restriction-P** predicate tests whether a given object is a Loom restriction object.

Syntax:

Restriction-P *object*

Argument:

The *object* argument is any object.

Value:

The **Restriction-P** predicate returns **t** if *object* is a min or max restriction, a filled-by or not-filled-by restriction, or a value restriction.

Examples:

```
(defconcept B :is (:and A (:at-least 1 r)))
(setq ?res (first (retrieve ?x (concept--restrictions B ?x))))
(ask (Restriction-P ?res)) ⇒ T
```

See Also:

concept--restrictions (p. 37)
get-roles (p. 135)
Min-Restriction-P (p. 199)
Max-Restriction-P (p. 193)
Value-Restriction-P (p. 294)

restriction--concepts *Relation*

Purpose:

The **restriction--concepts** relation is the inverse of **concept--restrictions**. It finds all of the concepts whose definitions include a particular restriction.

Syntax:

restriction--concepts *restriction concept*

Domain:

The *restriction* argument is a local or inherited restriction feature of a concept.

Range:

The *concept* argument is a concept which contains *restriction*.

Examples:

```
(defconcept B :is (:and A (:at-least 2 r)) :implies (:all s A))
(defconcept C :implies (:all s A))
(setq ?res (second (retrieve ?x (concept--restrictions B ?x))))
(ask (restriction--concepts ?res C)) ⇒ T
(retrieve (?res ?c) (:and (concept--restrictions B ?res)
                          (restriction--concepts ?res ?c))) ⇒
    ((|FTR|(:all S A) (|C|B |C|C))
     (|FTR|(:at-least 2 R) (|C|B)))
```

See Also:

concept--restrictions (p. 37)
get-roles (p. 135)

restriction--max *Relation*

Purpose:

The **restriction--max** relation extracts the upper bound from from a given max restriction.

Syntax:

restriction--max *restriction max*

Domain:

The *restriction* argument is a local or inherited restriction feature of a concept.

Range:

The *max* argument is the upper bound of *restriction*.

Examples:

```
(defconcept B :is (:and A (:at-most 2 r)) :implies (:all s A))
(retrieve ?max
         (:for-some (?res)
                    (:and (concept--restrictions B ?res)
                          (Max-Restriction-P ?res)
                          (restriction--max ?res ?max)))) ⇒ (2)
```

See Also:

concept--restrictions (p. 37)
get-role-cardinality (p. 129)
get-role-max-cardinality (p. 131)
Max-Restriction-P (p. 193)

restriction--min *Relation*

Purpose:

The **restriction--min** relation extracts the lower bound from from a given min restriction.

Syntax:

restriction--min *restriction min*

Domain:

The *restriction* argument is a local or inherited restriction feature of a concept.

Range:

The *min* argument is the lower bound of *restriction*.

Examples:

```
(defconcept B :is (:and A (:at-least 1 r)) :implies (:all s A))
(retrieve ?min
          (:for-some (?res)
                     (:and (concept--restrictions B ?res)
                           (Min-Restriction-P ?res)
                           (restriction--min ?res ?min)))) ⇒ (1)
```

See Also:

concept--restrictions (p. 37)
get-role-cardinality (p. 129)
get-role-min-cardinality (p. 132)
Min-Restriction-P (p. 199)

restriction--relation *Relation*

Purpose:

This relation takes a Loom restriction and returns the relation that the restriction constrains.

Syntax:

restriction--relation *restriction relation*

Domain:

The *restriction* argument is a local or inherited restriction feature of a concept.

Range:

The *relation* argument is the relation component of *restriction*.

Examples:

```
(defconcept B :is (:and A (:exactly 2 r)) :implies (:all s A))
(setq ?res (first (retrieve ?x (concept--restrictions B ?x))))
(ask (restriction--relation ?res r)) ⇒ T
(retrieve ?rel
          (:for-some (?res)
                     (:and (concept--restrictions B ?res)
                           (restriction--relation ?res ?rel)))) ⇒
    (|R|R |R|S)
```

See Also:

concept--restrictions (p. 37)
get-role (p. 128)
get-roles (p. 135)
relation--restrictions (p. 234)
restriction--max (p. 242)
restriction--min (p. 243)
restriction--value-restriction (p. 245)

restriction--value-restriction *Relation*

Purpose:

This relation takes a Loom restriction and returns the concept that serves as its value restriction.

Syntax:

restriction--value-restriction *restriction vr*

Domain:

The *restriction* argument is a local or inherited restriction feature of a concept.

Range:

The *vr* argument is the concept which acts as the value restriction of *restriction*.

Examples:

```
(defconcept B :is (:and A (:at-most 2 r)) :implies (:all s A))
(retrieve ?vr
        (:for-some (?res)
                 (:and (concept--restrictions B ?res)
                      (Value-Restriction-P ?res)
                      (restriction--value-restriction ?res ?vr)))) ⇒
   (|C|A)
```

See Also:

compute-value-restriction (p. 31)
concept--restrictions (p. 37)
get-role-types (p. 133)
Value-Restriction-P (p. 294)

retrieve *Macro*

Purpose:

The **retrieve** macro is used for retrieving facts from a knowledge base.

Syntax:

retrieve *variables query* `&key` *kb generators*

Arguments:

The *variables* argument is a symbol or list of symbols each beginning with the character ?.

The *query* argument is an expression in the Loom query language (see **Remarks** below). This language has the expressive power of the first-order predicate calculus.

The *kb* argument is the name of the knowledge base in which *query* is to be evaluated.

The *generators* argument is a list containing one or more of the terms in *query*. The Loom query optimizer will prefer these terms when selecting generators for the query variables. Normally, *generators* are terms that are expected to generate few instances.

Value:

If *variables* is a symbol, the **retrieve** macro returns a list of values satisfying *query*. If *variables* is a list, a list of tuples is returned, where each tuple contains as many values as there are output variables.

Remarks:

The query expressions appearing in the *query* argument above have the following syntax:

```
query-expr ::=
    ( {:AND | :OR} query-expr+ ) |
    ( {:NOT | :FAIL} query-expr ) |
    ( :IMPLIES query-expr query-expr ) |
    ( {:FOR-SOME | :FOR-ALL} ( ?Var+ ) query-expr ) |
    ( :COLLECT ( ?Var ) query-expr ) |
    ( concept instance ) |
    ( relation instance+ value ) |
    ( :SAME-AS instance instance ) |
    ( :SUBSET instance instance ) |
    ( :PREDCALL LispPredicate value+ ) |
    ( :ABOUT instance about-clause* )
```

Each *about-term* in a query **:about** clause has the form:

```
about-clause ::=
    concept |
    ( concept ) |
    ( relation value ) |
    ( :FILLED-BY relation value+ ) |
    ( {:AT-LEAST | :AT-MOST | :EXACTLY} Integer relation ) |
    ( {:ALL | :SOME | :THE} relation concept )
```

The query-expression operators have the following semantics:

- If Q = (**:and** Q1…Qn), Q is true if Q1…Qn are all true.
- If Q = (**:or** Q1…Qn), Q is true if at least one of Q1…Qn is true.
- If Q = (**:not** Q1), Q is true if Q1 is not true.
- If Q = (**:fail** Q1), Q is true if Loom cannot prove that Q1 is true.
- If Q = (**:for-some** (?X1…?Xn) Q1), Q is true if there are bindings for ?X1…?Xn that cause Q1 to be satisfied.
- If Q = (**:for-all** (?X1…?Xn) (**:implies** Q1 Q2)), Q is true if all bindings of ?X1…?Xn that satisfy Q1 also satisfy Q2.
- If Q = (**:collect** ?X Q1), Q returns the values V1…Vn such that Q1 is satisfied when ?X is bound to each Vi.
- If Q = (C I), Q is true if I is an instance of concept C.
- If Q = (R I V), Q is true if V is a filler of role R on instance I.
- If Q = (R I1…In), Q is true if the tuple <I1…In> is in the n-ary relation R
- If Q = (**:same-as** V1 V2), Q is true if values V1 and V2 are equivalent.
- If Q = (**:subset** V1 V2), Q is true if V1 and V2 evaluate to sets of values, and V1 is a subset of V2.
- If Q = (**:predcall** P V), Q is true if application of Lisp predicate P to value V returns true.
- If Q = (**:about** I C) or (**:about** I (C)), Q is true if I is an instance of concept C.
- If Q = (**:about** I (R V)), Q is true if the filler of role R on instance I is V.
- If Q = (**:about** I (**:filled-by** R V1…Vn)), Q is true if every value Vi is a filler of role R on instance I.
- If Q = (**:about** I (**:at-least** K R)), Q is true if instance I has at least K fillers of role R. The **:at-most** and **:exactly** operators have corresponding semantics.
- If Q = (**:about** I (**:all** R C)), Q is true if all fillers of role R on instance I are instances of concept C. The **:some** and **:the** operators have corresponding semantics.

A *concept*, *relation*, or *instance* may be either a symbol that names a Loom object, or a variable beginning with the character **?**. An *instance* may also be a constant or a formula, where a formula is a list of the form (*relation instance*).

Any **?**-variable not bound in *variables*, or in the variables of a **:for-some**, **:for-all**, or **:collect** clause, is assumed to be bound externally. The value of an external **?**-variable should be a Loom object, i.e., a concept, relation, or instance, rather than the name of such an object.

The **:filled-by-list** operator cannot be used in the **:about** clause of a query. The Loom Grammar at the end of this manual is too general in this respect.

Concepts and relations referenced by name in a query must be defined at the time the query is compiled. Instances referenced by identifier must exist at the time the query is executed.

Examples:

```
(retrieve ?x (Artist ?x)) ⇒ (|I|JOE ...)
(retrieve ?x (age Joe ?x)) ⇒ (39)
(retrieve (?x) (age Joe ?x) :kb user-kb) ⇒ ((39))
(retrieve ?x (max (age (child Joe)) ?x))
(retrieve ?x (distance Paris Rome ?x))
(retrieve (?x ?y) (wife ?x ?y)) ⇒ ((|I|JOE |I|SUE) (|I|FRED |I|MARY))
(retrieve ?x (:and (Woman ?x) (Senator ?x)) :generators ((Senator ?x)))
(retrieve ?x (:and (pet Joe ?x) (:not (Dog ?x))))
(retrieve ?x (:and (Artist ?x) (:fail (:or (French ?x)
                                           (resides ?x France)))))
(retrieve ?x (:and (Writer ?x) (:for-some ?y (:and (author ?y ?x)
                                                   (Best-Seller ?y)))))
(retrieve ?x (:and (Writer ?x) (:for-all ?y (:implies (author ?y ?x)
                                                      (Best-Seller ?y)))))
(retrieve ?x (:and (Salesman ?x) (:same-as (home ?x) (office ?x))))
(retrieve ?x (:and (Salesman ?x) (:subset (customer ?x) (customer Joe))))
(retrieve ?x (:and (age (child Joe) ?x) (:predcall #'oddp ?x)))
(retrieve ?x (:about ?x Man (:at-least 2 child) (:exactly 0 son)))
(retrieve ?x (:about ?x Man (:all child Female) (:some child Teen-Ager)))
(retrieve ?x (:about ?x Man (:filled-by daughter Mary Sue)))
(setq ?C (fc Artist) ?R (fr friend) ?V (fi Joe))
(retrieve ?x (:and (?C ?x) (?R ?x ?V) (:about ?x (:all ?R ?C))))
```

See Also:

ask (p. 20)
do-retrieve (p. 84)
get-inverse-values (p. 120)

role-members *Relation*

Purpose:

The **role-members** relation returns all the fillers of a specified role on a specified instance.

Syntax:

role-members *instance relation value*

Domains:

The *instance* argument is a role-bearing Loom instance.

The *relation* argument is the role of *instance* whose fillers are to be found.

Range:

The *value* argument is an asserted or inferred filler of the role *relation* on *instance*.

Examples:

```
(retrieve ?x (role-members Joe age ?x)) ⇒ (30)
(retrieve ?x (role-members Joe child ?x)) ⇒ (|I|FRED |I|MARY)
(ask (role-members Joe car 'Ford)) ⇒ T
```

See Also:

get-values (p. 144)
retrieve (p. 246)

same-as *Relation*

Purpose:

The **same-as** relation tests whether two sets are equivalent.

Syntax:

same-as *set1 set2*

Domain:

The *set1* argument is a list of objects.

Range:

The *set2* argument is a list of objects.

Remarks:

This relation uses Lisp **equal** to test equality.

The **same-as** relation is semantically related to, but syntactically distinct from, the **:same-as** concept-forming operator (see **defconcept** Remarks).

Examples:

```
(tellm (:about Fred (:filled-by friends Joe Pam Sue))
       (:about Mary (:filled-by friends Sue Joe Pam)))
(ask (same-as (friends Fred) (friends Mary))) ⇒ T
```

See Also:

member-of (p. 194)
members (p. 195)
subset (p. 270)

save-kb *Function*

Purpose:

The **save-kb** function writes out source definitions for the concepts, relations, instances, and behaviors in a specified knowledge base.

Syntax:

save-kb `&optional` *knowledgeBase* `&key` *partitions path-name*

Arguments:

The *knowledgeBase* argument is the knowledge base to be saved, or a symbol or string which names that knowledge base. If this argument is not supplied to **save-kb**, it defaults to the current knowledge base.

The *partitions* argument is a keyword or list of keywords specifying the types of objects to be saved. Available partitions are **:concepts**, **:relations**, **:instances**, and **:behaviors**. If this argument is not supplied, all partitions are saved.

The *path-name* argument is a string that specifies the file to be used for saving *knowledgeBase*. This defaults to the file that was associated with *knowledgeBase* when that knowledge base was defined. If no file was associated with *knowledgeBase*, a warning is issued.

Value:

The pathname of the saved knowledge base is returned.

Remarks:

The output of **save-kb** reflects the definitions originally supplied by the user. Unlike **pc**, these definitions are not augmented by inferences made during classification and recognition.

Unless the **:suppress-diehard-output** feature is set, **save-kb** prints the name of each saved object on standard output.

Examples:

```
(defkb test-kb nil :path-name "test-kb.lisp")
(save-kb) ⇒ "test-kb.lisp"
(save-kb "TEST-KB" :path-name "dump.lisp") ⇒ "dump.lisp"
(save-kb :partitions '(:concepts :relations)) ⇒ "test-kb.lisp"
```

See Also:

clear-kb (p. 29)
defkb (p. 59)
list-kb (p. 172)
load-kb (p. 184)

schedule *Macro*

Purpose:

The **schedule** macro is used to create tasks. A task is the application of a specific Loom action (implemented as a set of methods) to one or more objects. The **schedule** macro places the task on a queue for later execution, rather than executing it immediately (as **perform** does).

Syntax:

schedule (*actionName* `&rest` *arguments*) `&key` *priority*

Arguments:

The *actionName* argument is the name of the action which is to be applied to *arguments*. The procedure actually performed depends on which of the methods attached to *actionName* have their situation patterns satisfied by *arguments*.

The *arguments* argument contains zero or more variables or constants. Keyword arguments are supported, but optional and rest arguments are not.

The *priority* argument is currently either **:high** or **:low**. This argument specifies the queue on which the task is to be placed. The default value is **:high**.

Value:

The return value of **schedule** is the newly-created task.

Remarks:

Tasks may be scheduled as a result of production firings, as well as explicit calls to **perform**.

Scheduled tasks are executed when **perform-task** is called explicitly, or when the matcher is run (as a result of calling **new-time-point**, **tellm**, etc.).

Examples:

```
(schedule (inspect-box box shipper)) ⇒
    |TASK|(INSPECT-BOX |I|BOX3 |I|ACME)
(schedule (notify-shipper (fi Acme)) :priority :low) ⇒
    |TASK|(NOTIFY-SHIPPER |I|ACME)
```

See Also:

defmethod (p. 61)
defproduction (p. 64)
kill-task (p. 163)
list-tasks (p. 180)
new-time-point (p. 205)
perform (p. 214)
schedule-task (p. 256)

schedule-task *Function*

Purpose:

The **schedule-task** function places a given task on the priority queue (currently, **:high** or **:low**) that was specified when the task was created.

Syntax:

schedule-task *task*

Arguments:

The *task* argument is the task which is to be enqueued. The task was originally created when a production fired, or when there was an explicit call to **perform** or **schedule**.

Value:

The **schedule-task** function returns *task*.

Remarks:

Tasks may be performed as a result of a call to **schedule-task** or, more commonly, as a result of an explicit or implicit call to **new-time-point**. After matching and production firing, **new-time-point** performs all queued tasks in order of their priority. In the current version of Loom, the highest-priority task is the one at the front of the highest-priority queue.

Examples:

```
(setq t1 (perform (move (fi box3))))
(schedule-task t1) ⇒ |TASK|(MOVE |I|BOX3)
```

See Also:

initialize-tasks (p. 154)
kill-task (p. 163)
list-tasks (p. 180)
perform (p. 214)
perform-task (p. 216)
schedule (p. 254)
scheduled-p (p. 257)

scheduled-p *Function*

Purpose:

The **scheduled-p** function tests whether a given task is currently scheduled.

Syntax:

scheduled-p *task*

Arguments:

The *task* argument is the task which is to be checked. The task was originally created when a production fired, or when there was an explicit call to **perform** or **schedule**.

Value:

The **scheduled-p** function returns the queue containing *task*, or **nil** if *task* is not currently scheduled.

Examples:

```
(setq t1 (schedule (move (fi box3))))
(scheduled-p t1) ⇒ (:HIGH |TASK|(MOVE |I|BOX3))
```

See Also:

schedule (p. 254)
schedule-task (p. 256)
perform (p. 214)

set-features *Function*

Purpose:

The **set-features** function enables selected Loom features. These features control the overall behavior of Loom, including the amount and type of dialogue that Loom has with the user.

Syntax:

set-features `&rest` *features*

Arguments:

The *features* argument is one or more keywords which specify the features to be set:

- If the **:ask-about-redefinitions** feature is set, Loom checks with the user before redefining a concept. This feature is set by default.
- If the **:ask-for-concept-names** feature is set, Loom prompts the user for concept names whenever system-defined concepts are generated. The prompting is done after the new concepts have been completed. If the user declines to supply a name, the concept is assigned a system-generated name.
- If the **:ask-for-names-immediately** feature is set, Loom asks the user to name system-defined conjunction concepts as soon as they are created and classified.
- If the **:automatic-clos-classes** feature is set, a CLOS class is automatically created whenever **make-object** (or **tell** or **create** with the **:clos** flag set) is called to instantiate a concept that does not already have an associated CLOS class. If this feature is disabled, a CLOS class is not created unless the concept explicitly contains the **:clos-class** characteristic. The **:automatic-clos-classes** feature is set by default.
- If the **:clip-roles** feature is set, clipping is performed on all single-valued roles. That is, if two (or more) different instances are asserted to be fillers of a single-valued role, Loom automatically retracts all but the last assertion. This feature is set by default.
- If the **:clip-types** feature is set, clipping of instance types is enabled. That is, if an instance's type is asserted to contain two (or more) concepts which are disjoint, or would otherwise cause the instance to become incoherent, then Loom automatically retracts all but the last assertion.
- If the **:closed-world** feature is set, closed-world semantics are assumed for all concepts and relations.

- If the **:display-match-changes** feature is set, whenever a production memory match or instance classification occurs, Loom indicates which concepts have entered or exited the type of each affected instance, and which rete nodes the instance has entered or exited. This feature is set by default.
- If the **:domain-implies-role** feature is set when Loom creates a shadowing CLOS class for a concept C, then the new class will have slots corresponding to every relation whose domain is C.
- If the **:dont-name-system-defined-concepts** feature is set, Loom does not generate a name for system-defined concepts and relations (provided **:ask-for-names-immediately** is also enabled).
- If the **:emit-classify-dots** feature is set, a dot, minus, or plus is printed when a concept is classified, unclassified, or completed, respectively, and an exclamation point is printed when a concept is sealed (i.e., ready to participate in data manipulation operations). Also, the character **M** is printed when concepts merge during classification, **U** is printed when they unmerge, and **C** is printed when a CLOS class is created. This feature is set by default.
- If the **:emit-match-stars** feature is set, an asterisk is printed whenever the type of an instance changes. Also, Loom prints a **T** whenever a new type is created as a result of unifying a newly-matched concept with the concepts already in an instance's type. This feature is set by default.
- If the **:expand-system-defined-print-names** feature is set, the printed representation of an instance has the form |**I**|*identifier*(*concepts*), where *concepts* contains the names of the most-specific user-named concepts which the instance matches. If this feature is not set, type information is omitted from the printed representation of the instance.
- If the **:open-world** feature is set, open-world semantics are assumed for all concepts and relations. This feature is set by default.
- If the **:prefer-clos-instances** feature is set, Loom automatically creates a CLOS instance when **tell** or **create** is called. If there is no CLOS class corresponding to the concept being instantiated, Loom attempts to create one. If this effort fails, a standard Loom instance is created, and a warning is issued (unless the *clos-instance-p* argument to **create** was **:no-warning**).
- If the **:suppress-diehard-output** feature is set, all Loom informational and warning output is suppressed. For example, **save-kb** typeout, classify dots, type-change messages, and incoherence warnings are all inhibited. It is advisable to set this feature when loading very large knowledge bases because of the overhead associated with lengthy typeout (especially when running under Emacs).
- If the **:warn-about-syntax-corrections** feature is set, a warning is issued whenever a syntax error is detected in a concept definition or in a query. This feature is set by default.

- If the **:warn-on-nonprimitive-clos-classes** feature is set, a warning is issued whenever a CLOS class is created to shadow a non-primitive concept. This feature is set by default.

Value:

The **set-features** function returns a list of the Loom features which are currently set.

Examples:

```
(set-features :closed-world) ⇒
    (:CLOSED-WORLD :EMIT-MATCH-STARS ...)
(set-features :prefer-clos-instances :suppress-diehard-output) ⇒
    (:PREFER-CLOS-INSTANCES :SUPPRESS-DIEHARD-OUTPUT :CLOSED-WORLD ...)
```

See Also:

list-features (p. 167)
reset-features (p. 239)
unset-features (p. 286)
with-feature-changes (p. 298)
with-features (p. 301)

set-value *Function*

Purpose:

The **set-value** function replaces the current value of a given single-valued role on a given instance.

Syntax:

set-value *instance role value* `&key` *kb no-error-p*

Arguments:

The *instance* argument is an instance or an instance identifier.

The *role* argument is a single-valued relation, or the name of such a relation.

The *value* argument is the new value of *role*. If *value* is a list, the list is interpreted as a single role filler. If *value* is a symbol, the value is coerced to an instance unless the the range of *role* indicates that it should be interpreted as a constant.

The *kb* argument is the knowledge base, or the name of the knowledge base, where lookup of *instance* and *role* begins. This defaults to the current knowledge base.

If the *no-error-p* argument is `t`, then error messages are suppressed.

Value:

If the update is successful, the new value of *role* is returned. If it fails and *no-error-p* is set, then: (1) if the type of *value* conflicts with the range of *role*, the two values **nil** and **:type-clash** are returned, (2) if *instance* (or *role*) cannot be found, **nil** and **:missing-instance** (or **:missing-role**) are returned.

Examples:

```
(defrelation r :characteristics :single-valued)
(set-value 'Joe 'r 3) ⇒ 3
(set-value 'Joe 'r '(3 4 5)) ⇒ (3 4 5)
(set-value (fi Joe) (fr r) nil :kb "USER-KB") ⇒ NIL
(set-value 'Joe 'r 'Mary) ⇒ |I|MARY
(defrelation s :characteristics :single-valued :range Symbol)
(set-value 'Joe 's 'Mary) ⇒ MARY
(set-value 'Joe 's 3 :no-error-p t) ⇒ NIL :TYPE-CLASH
```

See Also:

get-value (p. 142)
set-values (p. 262)

set-values *Function*

Purpose:

The **set-values** function replaces the current value(s) of a given multiple-valued or single-valued role on a given instance.

Syntax:

set-values *instance role values* `&key` *kb no-error-p*

Arguments:

The *instance* argument is an instance or an instance identifier.

The *role* argument is a relation, or the name of a relation.

The *values* argument is a list of objects which will be the new fillers of *role*. If a member of *values* is a symbol, the member is coerced to an instance unless the the range of *role* indicates that it should be interpreted as a constant. If *role* is single-valued, the value of *role* is set to the first member of *values*.

The *kb* argument is the knowledge base, or the symbol or string that names the knowledge base, where the lookup of *instance* and *role* begins. This defaults to the current knowledge base.

If the *no-error-p* argument is **t**, then error messages are suppressed.

Value:

If the update is successful, the new value of *role* is returned. If it fails and *no-error-p* is set, then: (1) if the type of any member of *value* conflicts with the range of *role*, the two values **nil** and **:type-clash** are returned, (2) if *instance* (or *role*) cannot be found, **nil** and **:missing-instance** (or **:missing-role**) are returned.

Examples:

```
(defrelation rr) (set-values 'Joe 'rr '(3 4 5)) ⇒ (3 4 5)
(set-values (fi Joe) (fr rr) nil :kb "USER-KB") ⇒ NIL
(set-values 'Joe 'rr '(Mary)) ⇒ (|I|MARY)
(defrelation ss :range Symbol)
(set-values 'Joe 'ss '(Mary Sue)) ⇒ (MARY SUE)
(set-values 'Joe 'ss '(3) :no-error-p t) ⇒ NIL :TYPE-CLASH
(defrelation r :characteristics :single-valued)
(set-values 'Joe 'r '(3 4 5)) ⇒ (3 4 5)
(get-values 'Joe 'r ⇒ (3)
```

See Also:

add-value (p. 18)
get-values (p. 144)
remove-value (p. 236)
set-value (p. 261)
tell (p. 275)

show *Method*
show-all *Method*

Purpose:

These methods print the slot names and values of an arbitrary Loom object. The **show** method displays selected slots of the object, while **show-all** displays all slots (in the manner of Lisp **describe**).

Syntax:

show *object* `&key` *stream*
show-all *object* `&key` *stream*

Arguments:

The *object* argument is a Loom object, or a symbol which names a concept, relation, or instance.

The *stream* argument is a Lisp output character stream. This argument defaults to **t**.

Value:

The **show** and **show-all** methods return **nil**.

Examples:

```
(show 'Thing) ⟶ ...
(show-all (fr +)) ⟶ ...
(show (first (list-methods (fb move))))⟶ ...
```

See Also:

list-role-names&values (p. 178)
pc (p. 212)
pi (p. 217)
pprint-object (p. 221)
pr (p. 222)

show-progress *Function*

Purpose:

The **show-progress** function is a hook for user interfaces. If the **:emit-classify-dots** feature is set, it is called with argument **:classify** whenever a concept classifies, completes, or merges. If the **:emit-match-stars** feature is set, it is called with argument **:match** whenever the type of an instance changes. The user may redefine **show-progress** to give custom progress indications, in addition to the standard dots, stars, etc. issued by Loom.

Syntax:

show-progress *activity*

Arguments:

The *activity* argument indicates the type of Loom activity currently in progress. This may be either **:classify** or **:match**.

Value:

This function returns **t**.

Examples:

```
(defun show-progress (activity)
  (case activity
    (:classify (print-something))
    (:match (print-something-else)))
  t)
```

See Also:

set-features (p. 258)

Single-Valued-P *Concept*

Purpose:

The Single-Valued-P predicate tests whether a given relation is marked single-valued.

Syntax:

Single-Valued-P *relation*

Argument:

The *relation* argument is a relation.

Value:

The **Single-Valued-P** predicate returns **t** if *relation*, when used as a role, can only have a single filler.

Examples:

```
(defrelation age :characteristics :single-valued)
(ask (Single-Valued-P age)) ⇒ T
(retrieve ?x (:and (Relation ?x) (Single-Valued-P ?x))) ⇒ (|R|AGE ...)
```

See Also:

Primitive-P (p. 225)
User-Defined-P (p. 293)

String *Concept*

Purpose:

The **String** predicate tests whether a given object is a Lisp string.

Syntax:

String *object*

Argument:

The *object* argument is any object.

Value:

The **String** predicate returns **t** if *object* is a string.

Examples:

```
(ask (String "abc")) ⇒ T
```

See Also:

Constant (p. 39)
Non-Loom-Thing (p. 207)

subconcept-p *Macro*

Purpose:

The **subconcept-p** macro is the basic Loom subsumption test for classified concepts.

Syntax:

subconcept-p *concept1 concept2* `&key` *cant-be-equal-p*

Arguments:

The *concept1* and *concept2* arguments are classified concepts or relations.

If the *cant-be-equal-p* argument is **t**, Loom does not perform an equality test on the two concepts when determining whether one specializes the other.

Value:

The **subconcept-p** macro returns **t** if *concept1* and *concept2* are identical, or if *concept1* lies directly or indirectly below *concept2* in the subsumption hierarchy.

Examples:

```
(subconcept-p (fc Bachelor) (fc Bachelor)) ⇒ T
(subconcept-p (fc Bachelor) (fc Man)) ⇒ T
(subconcept-p (fc Bachelor) (fc Thing) :cant-be-equal-p t) ⇒ T
(subconcept-p (fr daughter) (fr child)) ⇒ T
(subconcept-p (fc Integer) (fc Non-Loom-Thing)) ⇒ T
(subconcept-p (fc Dog-Cat) (fc Incoherent)) ⇒ T
```

See Also:

direct-subconcepts (p. 80)
direct-superconcepts (p. 81)
find-subsumers&subsumees (p. 108)
isa-p (p. 162)
most-general-concepts (p. 200)
most-specific-concepts (p. 201)
subconcepts (p. 269)
superconcepts (p. 273)

subconcepts *Relation*

Purpose:

The **subconcepts** relation is the transitive closure of **direct-subconcepts**. It finds all subconcepts of a given concept or relation.

Syntax:

subconcepts *concept1 concept2*

Domain:

The *concept1* argument, a concept or relation, is a superconcept or superrelation of *concept2*.

Range:

The *concept2* argument, a concept or relation, is a subconcept or subrelation of *concept1*.

Examples:

```
(defconcept Man :is-primitive Person)
(defconcept Bachelor :is (:and Man (:exactly 0 wife)))
(ask (subconcepts Man Bachelor)) ⇒ T
(retrieve ?x (subconcepts Person ?x)) ⇒ (|C|MAN |C|BACHELOR)
```

See Also:

direct-subconcepts (p. 80)
get-subconcepts (p. 137)
get-subrelations (p. 138)
isa-p (p. 162)
subconcept-p (p. 268)
superconcepts (p. 273)

subset *Relation*

Purpose:

The **subset** relation tests whether a set is a subset of another set.

Syntax:

subset *set1 set2*

Domain:

The *set1* argument is a list of objects.

Range:

The *set2* argument is a list of objects.

Examples:

```
(ask (subset '(3 4) '(3 5 4))) ⇒ T
(tellm (:about Fred (:filled-by friends Joe Sue))
       (:about Mary (:filled-by friends Sue Joe Pam)))
(ask (subset (friends Fred) (friends Mary))) ⇒ T
```

See Also:

member-of (p. 194)
members (p. 195)
same-as (p. 251)

successor *Relation*

Purpose:

The **successor** relation returns the element which follows a specified member of an ordered set.

Syntax:

successor *set member1 member2*

Domains:

The *set* argument is a concept defined as an ordered set.

The *member1* argument is a numeric or symbolic member of *set*.

Range:

The *member2* argument is the numeric or symbolic member which immediately follows *member1* in *set*.

Examples:

```
(defset Level :is (:the-ordered-set 'LOW 'MEDIUM 'HIGH))
(ask (successor Level 'LOW 'MEDIUM)) ⇒ T
(retrieve ?x (successor Level 'MEDIUM ?x)) ⇒ (HIGH)
```

See Also:

concept--instances (p. 35)
defset (p. 72)
predecessor (p. 224)
> (p. 16)

sum *Relation*

Purpose:

The **sum** relation is an aggregation operator that returns the sum of a set of numbers.

Syntax:

sum *setOfNumbers number*

Domain:

The *setOfNumbers* argument is a list of integer and/or real numbers.

Range:

The *number* argument is the sum of *setOfNumbers*.

Examples:

```
(ask (sum '(3 4 5) 12)) ⇒ T
(tellm (:about Box3 (:filled-by contents Item1 Item2))
       (weight Item1 30) (weight Item2 40))
(retrieve ?x (sum (weight (contents Box3)) ?x)) ⇒ (70)
```

See Also:

count (p. 41)
min (p. 198)
max (p. 192)
+ (p. 15)

superconcepts *Relation*

Purpose:

The **superconcepts** relation is the transitive closure of **direct-superconcepts**. It finds all superconcepts of a given concept or relation.

Syntax:

superconcepts *concept1 concept2*

Domain:

The *concept1* argument, a concept or relation, is a subconcept or subrelation of *concept2*.

Range:

The *concept2* argument, a concept or relation, is a superconcept or superrelation of *concept1*.

Examples:

```
(defrelation child :is-primitive relative)
(defrelation daughter :is (:and child (:range Female)))
(ask (superconcepts child relative)) ⇒ T
(retrieve ?x (superconcepts daughter ?x)) ⇒
    (|R|CHILD |R|RELATIVE |R|BINARY-TUPLE)
```

See Also:

direct-superconcepts (p. 81)
get-superconcepts (p. 139)
get-superrelations (p. 140)
isa-p (p. 162)
subconcept-p (p. 268)
subconcepts (p. 269)

Symbol *Concept*

Purpose:

The **Symbol** predicate tests whether a given object is a Lisp symbol.

Syntax:

Symbol *object*

Argument:

The *object* argument is any object.

Value:

The **Symbol** predicate returns **t** if *object* is a symbol.

Examples:

```
(ask (Symbol 'foo)) ⇒ T
```

See Also:

Constant (p. 39)
Non-Loom-Thing (p. 207)

tell *Macro*
tellm *Macro*

Purpose:

The **tell** macro is used to assert propositions to the Loom database. A proposition states that an instance belongs to a given concept or has a particular feature, or that a relationship holds between two or more instances, or that two instances are equivalent.

The **tellm** macro asserts one or more propositions, and then calls the Loom matcher to recompute the types of the affected instances.

Syntax:

tell `&body` *propositions*
tellm `&body` *propositions*

Arguments:

The *propositions* argument consists of zero or more propositions (see **Remarks** below) which are to be asserted.

Value:

The **tell** macro normally returns the symbol **ok**. However, if *propositions* contains one or more **:about** clauses, the subject of the last such clause is returned.

The **tellm** macro returns the number of the newly-advanced knowledge base state.

Remarks:

Each proposition in the *propositions* argument above has the following syntax:

```
proposition ::=
    ( concept instance ) |
    ( relation instance+ value ) |
    ( :CREATE ?Var concept [:CLOS] ) |
    ( :SAME-AS instance instance ) |
    ( :ABOUT instance about-clause* )
```

Each *about-clause* in a proposition has the form:

```
about-clause ::=
    concept |
    ( concept ) |
    ( relation value ) |
    ( :FILLED-BY relation value+ ) |
    ( :FILLED-BY-LIST relation List ) |
    ( {:AT-LEAST | :AT-MOST | :EXACTLY} Integer relation ) |
    ( {:ALL | :SOME | :THE} relation concept )
```

A *concept*, *relation*, or *instance* may be either a symbol that names a Loom object, or a variable beginning with the character **?**. An *instance* may also be a constant or a formula, where a formula is a list of the form (*relation instance*).

The scope of the **?**-variable in a **:create** proposition extends to all of the propositions following that **:create**. In general, any **?**-variable not bound in a preceding **:create** proposition is assumed to be bound externally. The value of an external **?**-variable should be a Loom object, i.e., a concept, relation, or instance, rather than the name of such an object.

A **:create** proposition may optionally contain the **:clos** keyword. If this keyword is supplied, or if the global feature **:prefer-clos-instances** is set, a CLOS instance (see **make-object**) is created instead of a Loom instance. If there is no existing CLOS class corresponding to *concept*, Loom attempts to create the needed class. If this attempt fails, a standard Loom instance is created and a warning is issued.

Concepts and relations referenced by name in an assertion must be defined at the time the assertion is compiled. References to an instance, say A, bind to a previously declared concept or relation with name A, if such exists. Otherwise, they bind to an existing instance with identifier A. If no such instance can be found, then a new instance with identifier A is created. Subsequent references to A bind to the same object as the first one.

A **:same-as** proposition asserts that two instances are equivalent. It causes them to be merged into a single instance that combines the assertions made about each of them individually.

If two or more values are asserted to be fillers of a single-valued role, Loom automatically clips, i.e., retracts, all but the last assertion (providing that the **:auto-clip** feature is set). Similarly, a **:filled-by-list** assertion has the effect of retracting all previously-asserted fillers of a role.

Examples:

```
(tell (Person Joe)) ⇒ OK
(tell (age Joe 30) (wife Joe Sue) (Teacher Sue)) ⇒ OK
(tell (age (wife Joe) (+ (age Joe) 1))) ⇒ OK
(tellm (Person Honey) (:same-as (wife Joe) Honey)) ⇒ 3
(setq ?C (fc Person) ?R (fr age) ?I (fi Joe) ?V 30)
(tellm (?C ?I) (?R ?I ?V)) ⇒ 4
(tellm (:create ?X Dog) (pet Joe ?X)) ⇒ 5
(tell (:about Joe Person (wife Sue))) ⇒ |I|JOE
(tellm (:about Joe (child Fred) (child Mary))) ⇒ 6
(tellm (:about Joe (:filled-by child Fred Mary Don))) ⇒ 7
(setq ?L (list (fi Fred) (fi Mary) (fi Don)))
(tellm (:about Joe (:filled-by-list child ?L))) ⇒ 8
(tellm (:about Joe (:at-least 3 child) (:at-most 1 daughter)
                   (:exactly 2 son))) ⇒ 9
(tellm (:about Joe (:all child Adult) (:some son Bachelor))) ⇒ 10
(setq ?I (fi Joe) ?R (fr child) ?C (fc Adult))
(tellm (:about ?I (:all ?R ?C))) ⇒ 11
```

See Also:

add-type (p. 17)
add-value (p. 18)
ask (p. 20)
copy-instance (p. 40)
create (p. 42)
destroy (p. 78)
find-or-create-instance (p. 104)
forget (p. 109)
make-object (p. 189)
new-time-point (p. 205)
retrieve (p. 246)
set-values (p. 262)

term-implies *Relation*

Purpose:

The **term-implies** relation returns all of the concepts (or relations) that are implied by a given concept (or relation).

Syntax:

term-implies *concept1 concept2*

Domain:

The *concept1* argument is a concept or relation that implies *concept2*.

Range:

The *concept2* argument is a concept or relation implied by *concept1*.

Remarks:

The implicands of *concept1* include all of (1) its superconcepts, (2) its consequents, that is, concepts which were introduced in its definition's **:implies** clause, or in an **implies** statement, and (3) the superconcepts of its consequents, and the consequents of its superconcepts, recursively.

Examples:

```
(defconcept AA)
(defconcept A :is-primitive AA :implies B)
(defconcept BB)
(defconcept B :is-primitive BB)
(retrieve ?x (term-implies A ?x)) ⇒ (|C|A |C|AA |C|B |C|BB |C|THING)
(defrelation r)
(defrelation s)
(implies r s)
(retrieve ?x (term-implies r ?x)) ⇒ (|R|R |R|S |R|BINARY-TUPLE)
```

See Also:

defconcept (p. 49)
defrelation (p. 66)
implies (p. 148)
superconcepts (p. 273)

test-not-type-p *Function*

Purpose:

The **test-not-type-p** function looks for a proof that a given instance is not a member of a given concept.

Syntax:

test-type-p *instance concept* `&key` *kb no-error-p*

Arguments:

The *instance* argument is the instance, or the identifier of the instance, to be tested.

The *concept* argument is a concept, or the name of a concept.

The *kb* argument is the knowledge base, or the symbol or string that names the knowledge base, where the lookup of *instance* and *concept* begins. This defaults to the current knowledge base.

If the *no-error-p* argument is **t**, then error messages are suppressed.

Value:

The **test-not-type-p** function returns **t** if it can prove that *instance* is not a member of *concept*. If *no-error-p* is set and *instance* (or *concept*) cannot be found, the two values **nil** and **:missing-instance** (or **:missing-concept**) are returned.

Examples:

```
(defconcept Pet :partitions (($PET$ (Dog Cat Bird))))
(tellm (Dog Fido))
(test-not-type-p 'Fido 'Cat) ⇒ T
(test-not-type-p (fi Fido) (fc Bird) :kb "CL-USER-KB") ⇒ T
(test-not-type-p 'Fido 'xxx :no-error-p t) ⇒ NIL :MISSING-CONCEPT
(defset Primary-Color :is (:one-of 'Red 'Green 'Blue))
(test-not-type-p ''Orange 'Primary-Color) ⇒ T
(test-not-type-p 3 'String) ⇒ T
```

See Also:

get-types (p. 141)
instance--concepts (p. 158)
isa-p (p. 162)
test-type-p (p. 280)

test-type-p *Function*

Purpose:

The **test-type-p** function looks for a proof that a given instance is a member of a given concept.

Syntax:

test-type-p *instance concept* `&key` *kb no-error-p*

Arguments:

The *instance* argument is the instance, or the identifier of the instance, to be tested.

The *concept* argument is a concept, or the name of a concept.

The *kb* argument is the knowledge base, or the symbol or string that names the knowledge base, where the lookup of *instance* and *concept* begins. This defaults to the current knowledge base.

If the *no-error-p* argument is **t**, then error messages are suppressed.

Value:

The **test-type-p** function returns **t** if it can prove that *instance* is a member of *concept*. If *no-error-p* is set and *instance* (or *concept*) cannot be found, the two values **nil** and **:missing-instance** (or **:missing-concept**) are returned.

Examples:

```
(tellm (:about Fido Male Dog))
(test-type-p 'Fido 'Dog) ⇒ T
(test-type-p (fi Fido) (fc Male) :kb "USER-KB") ⇒ T
(test-type-p 'Fido 'xxx :no-error-p t) ⇒ NIL :MISSING-CONCEPT
(defset Primary-Color :is (:one-of 'Red 'Green 'Blue))
(test-type-p 'Red 'Primary-Color) ⇒ T
(test-type-p 3 'Number) ⇒ T
(test-type-p 'Thing 'Concept) ⇒ T
(test-type-p (fc Concept) (fc Thing)) ⇒ T
```

See Also:

get-types (p. 141)
instance--concepts (p. 158)
isa-p (p. 162)
test-not-type-p (p. 279)

Thing *Concept*

Purpose:

The concept **Thing** is at the top of the concept hierarchy, that is, it subsumes all other concepts. Also, all Loom objects and non-Loom objects are instances of Thing.

Syntax:

Thing *object*

Arguments:

The *object* argument is any object.

Value:

The **Thing** predicate always returns **t**.

Examples:

```
(ask (subconcepts Thing Person)) ⇒ T
(ask (Thing Person)) ⇒ T
(ask (Thing Joe)) ⇒ T
(ask (Thing 3)) ⇒ T
```

See Also:

binary-tuple (p. 23)
Concept (p. 32)
Loom-Thing (p. 188)
n-ary-tuple (p. 202)
Non-Loom-Thing (p. 207)
subconcepts (p. 269)

trace *Macro*

Purpose:

The Loom **trace** function is identical to Lisp's **trace**, except that it also traces Loom methods and relations.

Syntax:

trace &body *functionNames*

Arguments:

The *functionNames* argument supplies the names of zero or more functions, methods, and sealed relations which are to be traced.

Value:

If *functionNames* are supplied, **trace** simply returns these names. Otherwise, a list of the names of all currently-traced functions, methods, and relations is returned.

Remarks:

On entry to a traced Loom method, Loom prints the method's action name, title, and arguments. If the method is untitled, its situation pattern is used instead of the title. On exit, the method value is printed.

When the predicate of a traced relation is invoked, Loom prints the names of the relation and its arguments. When the predicate returns, its value is printed. Relation predicates are invoked during both recognition and query processing.

The **use-loom** amd **call-use-loom** functions perform a shadowing-import of the symbol "LOOM:**trace**" into the using package.

Examples:

```
(defun foo (x) x)
(defmethod M (?x) :title "test" :action ((foo ?x)))
(trace foo M) ⇒ (FOO M)
(perform (M 3)) ⟶
    1 ENTER M-"test":  3
    | 2 Enter FOO 3
    | 2 Exit FOO 3
    1 EXIT M-"test":  3
```

See Also:

defmethod (p. 61)
perform (p. 214)
untrace (p. 289)
use-loom (p. 291)

trace-all *Function*

Purpose:

The **trace-all** function enables global tracing of productions.

Syntax:

trace-all

Arguments:

This function takes no arguments.

Value:

The **trace-all** function returns **t**.

Remarks:

When a traced production fires, Loom prints the name of the production and the triggering instance.

Examples:

```
(defproduction P :when (:detects (A ?x)) :do ((print ?x)))
(trace-all) ⇒ T
(tellm (A Joe)) ⟶
    Fire:  P (|I|JOE)
    |I|JOE
```

See Also:

defproduction (p. 64)
list-productions (p. 177)
untrace-all (p. 290)

unmake-object *Function*

Purpose:

The **unmake-object** function undoes the indexing between a CLOS instance and its Loom concept.

Syntax:

unmake-object *instance*

Arguments:

The *instance* argument is a CLOS instance, i.e., an instance of a CLOS class that shadows a Loom concept.

Value:

The **unmake-object** function returns **nil**.

Examples:

```
(setq i (make-object 'Lawyer)) ⇒ #<Lawyer #X213628E>
(retrieve ?x (Lawyer ?x)) ⇒ (#<Lawyer #X213628E>)
(unmake-object i) ⇒ NIL
(retrieve ?x (Lawyer ?x)) ⇒ NIL
```

See Also:

change-object (p. 27)
destroy (p. 78)
make-object (p. 189)

unset-features *Function*

Purpose:

The **unset-features** function disables selected Loom features. These features control the overall behavior of Loom, including the amount and type of dialogue that Loom has with the user.

Syntax:

unset-features `&rest` *features*

Arguments:

The *features* argument is one or more keywords which specify the features to be unset:

- If the **:ask-about-redefinitions** feature is set, Loom checks with the user before redefining a concept. This feature is set by default.
- If the **:ask-for-concept-names** feature is set, Loom prompts the user for concept names whenever system-defined concepts are generated. The prompting is done after the new concepts have been completed. If the user declines to supply a name, the concept is assigned a system-generated name.
- If the **:ask-for-names-immediately** feature is set, Loom asks the user to name system-defined conjunction concepts as soon as they are created and classified.
- If the **:automatic-clos-classes** feature is set, a CLOS class is automatically created whenever **make-object** (or **tell** or **create** with the **:clos** flag set) is called to instantiate a concept that does not already have an associated CLOS class. If this feature is disabled, a CLOS class is not created unless the concept explicitly contains the **:clos-class** characteristic. The **:automatic-clos-classes** feature is set by default.
- If the **:clip-roles** feature is set, clipping is performed on all single-valued roles. That is, if two (or more) different instances are asserted to be fillers of a single-valued role, Loom automatically retracts all but the last assertion. This feature is set by default.
- If the **:clip-types** feature is set, clipping of instance types is enabled. That is, if an instance's type is asserted to contain two (or more) concepts which are disjoint, or would otherwise cause the instance to become incoherent, then Loom automatically retracts all but the last assertion.
- If the **:closed-world** feature is set, closed-world semantics are assumed for all concepts and relations.

- If the **:display-match-changes** feature is set, whenever a production memory match or instance classification occurs, Loom indicates which concepts have entered or exited the type of each affected instance, and which rete nodes the instance has entered or exited. This feature is set by default.
- If the **:domain-implies-role** feature is set when Loom creates a shadowing CLOS class for a concept C, then the new class will have slots corresponding to every relation whose domain is C.
- If the **:dont-name-system-defined-concepts** feature is set, Loom does not generate a name for system-defined concepts and relations (provided **:ask-for-names-immediately** is also enabled).
- If the **:emit-classify-dots** feature is set, a dot, minus, or plus is printed when a concept is classified, unclassified, or completed, respectively, and an exclamation point is printed when a concept is sealed (i.e., ready to participate in data manipulation operations). Also, the character **M** is printed when concepts merge during classification, **U** is printed when they unmerge, and **C** is printed when a CLOS class is created. This feature is set by default.
- If the **:emit-match-stars** feature is set, an asterisk is printed whenever the type of an instance changes. Also, Loom prints a **T** whenever a new type is created as a result of unifying a newly-matched concept with the concepts already in an instance's type. This feature is set by default.
- If the **:expand-system-defined-print-names** feature is set, the printed representation of an instance has the form |**I**|*identifier*(*concepts*), where *concepts* contains the names of the most-specific user-named concepts which the instance matches. If this feature is not set, type information is omitted from the printed representation of the instance.
- If the **:open-world** feature is set, open-world semantics are assumed for all concepts and relations. This feature is set by default.
- If the **:prefer-clos-instances** feature is set, Loom automatically creates a CLOS instance when **tell** or **create** is called. If there is no CLOS class corresponding to the concept being instantiated, Loom attempts to create one. If this effort fails, a standard Loom instance is created, and a warning is issued (unless the *clos-instance-p* argument to **create** was **:no-warning**).
- If the **:suppress-diehard-output** feature is set, all Loom informational and warning output is suppressed. For example, **save-kb** typeout, classify dots, type-change messages, and incoherence warnings are all inhibited. It is advisable to set this feature when loading very large knowledge bases because of the overhead associated with lengthy typeout (especially when running under Emacs).
- If the **:warn-about-syntax-corrections** feature is set, a warning is issued whenever a syntax error is detected in a concept definition or in a query. This feature is set by default.

- If the **:warn-on-nonprimitive-clos-classes** feature is set, a warning is issued whenever a CLOS class is created to shadow a non-primitive concept. This feature is set by default.

Value:

The **unset-features** function returns a list of the Loom features which are currently set.

Examples:

```
(unset-features :closed-world) ⇒
    (:OPEN-WORLD :EMIT-MATCH-STARS ...)
(unset-features :emit-match-stars :prefer-clos-instances) ⇒
    (:OPEN-WORLD :WARN-ON-NONPRIMITIVE-CLOS-CLASSES ...)
```

See Also:

list-features (p. 167)
reset-features (p. 239)
set-features (p. 258)
with-feature-changes (p. 298)
with-features (p. 301)

untrace *Macro*

Purpose:

The Loom **untrace** function is identical to Lisp's **untrace**, except that it also untraces Loom methods and relations.

Syntax:

untrace `&body` *functionNames*

Arguments:

The *functionNames* argument supplies the names of zero or more functions, methods, and sealed relations which are to be untraced.

Value:

If *functionNames* are supplied, **untrace** simply returns these names. Otherwise, it returns a list of the functions, methods, and relations for which tracing is being stopped.

Remarks:

The **use-loom** amd **call-use-loom** functions perform a shadowing-import of the symbol "LOOM:**untrace**" into the using package.

Examples:

```
(defun foo (x) x)
(defmethod M (?x) :title "test" :action ((foo ?x)))
(untrace foo M) ⇒ (FOO M)
```

See Also:

defmethod (p. 61)
perform (p. 214)
trace (p. 282)

untrace-all *Function*

Purpose:

The **untrace-all** function disables global tracing of productions.

Syntax:

untrace-all

Arguments:

This function takes no arguments.

Value:

The **untrace-all** function returns **nil**.

Examples:

`(untrace-all)` ⇒ `NIL`

See Also:

defproduction (p. 64)
list-productions (p. 177)
trace-all (p. 284)

use-loom *Macro*

Purpose:

The **use-loom** macro imports Loom symbols into a specified package and optionally creates a new knowledge base that is associated with that package. In contrast to **call-use-loom**, which can be invoked from within other functions, **use-loom** should only be called at top-level.

Syntax:

use-loom *packageName* `&key` *dont-create-knowledge-base-p knowledge-base-name path-name loom-imports*

Arguments:

The *packageName* argument is a symbol or string that identifies the Lisp package which will use Loom.

If the *dont-create-knowledge-base-p* argument is **t**, **use-loom** does not create a new knowledge base. Otherwise, it defines a knowledge base named *packageName*-KB.

The *knowledge-base-name* argument is a symbol or string to be used as the name of the new knowledge base. It overrides the default name, *packageName*-KB.

The *path-name* argument is a string specifying the file to be used for saving and restoring the knowledge base. The pathname may be supplied here, or it may be passed directly to **save-kb**.

The *loom-imports* argument is a string, or a list of strings, identifying Loom symbols which are to be shadowing-imported into *packageName*. These symbols override their counterparts in the **common-lisp** package, or in other packages inherited by *packageName*.

Value:

This function returns **nil**.

Remarks:

The variable "LOOM::***shadow-import-from-loom***" contains symbols (such as **defmethod**, **trace**, and **pi**) that are automatically shadowing-imported into *packageName* in addition to any symbols specified by *loom-imports*.

The **compile-loom** and **load-loom** utilities are used to build the Loom system from source files or binaries. After loading the required files, these utilities ask the user to call **use-loom** to set up an initial knowledge base.

Examples:

```
(use-loom "USER") ⇒ NIL
(use-loom test :dont-create-knowledge-base-p t
               :path-name "test-kb.lisp") ⇒ NIL
(use-loom 'foo :knowledge-base-name "FOO-KB"
               :loom-imports '("ASK" "TELL")) ⇒ NIL
```

See Also:

call-use-loom (p. 24)
defkb (p. 59)
defmethod (p. 61)
list-knowledge-bases (p. 174)
save-kb (p. 252)
trace (p. 282)

User-Defined-P *Concept*

Purpose:

The User-Defined-P predicate tests whether a given concept or relation was defined by the user rather than by Loom.

Syntax:

User-Defined-P *object*

Argument:

The *object* argument is a concept or relation.

Value:

The **User-Defined-P** predicate returns **t** if *object* was defined by the user.

Remarks:

Concepts and relations in the Loom upper structure are marked user-defined.

Examples:

```
(defconcept A :implies B)
(ask (User-Defined-P A)) ⇒ T
(ask (User-Defined-P A&B)) ⇒ NIL
(retrieve ?x (User-Defined-P ?x)) ⇒ (|C|A |C|B |R|> |C|CONCEPT ...)
```

See Also:

defconcept (p. 49)
Primitive-P (p. 225)
Single-Valued-P (p. 266)

Value-Restriction-P *Concept*

Purpose:

The **Value-Restriction-P** predicate tests whether a given object is a Loom restriction object that specifies the type of the fillers of a role.

Syntax:

Value-Restriction-P *object*

Argument:

The *object* argument is any object.

Value:

The **Value-Restriction-P** predicate returns **t** if *object* is a value restriction.

Examples:

```
(defconcept B :is (:and A (:all r A)))
(setq ?res (first (retrieve ?x (concept--restrictions B ?x))))
(ask (Value-Restriction-P ?res)) ⇒ T
(retrieve ?vr
    (:for-some (?res)
               (:and (concept--restrictions B ?res)
                     (Value-Restriction-P ?res)
                     (restriction--value-restriction ?res ?vr)))) ⇒
    (|C|A)
```

See Also:

compute-value-restriction (p. 31)
concept--restrictions (p. 37)
get-role-types (p. 133)
get-roles (p. 135)
restriction--value-restriction (p. 245)
Restriction-P (p. 240)

where-is-it *Function*

Purpose:

The **where-is-it** function returns the knowledge base and the knowledge base partition of all Loom objects having a given name.

Syntax:

where-is-it *object*

Arguments:

The *object* argument is a Loom object, or the name of a Loom object. If it is a name, special characters are interpreted as in **fc**, and package prefixes are optional because the search extends across package boundaries.

Value:

The **where-is-it** function returns a list of triples. Each triple contains an object having the specified name, followed by the knowledge base and knowledge base partition in which the object was found. The knowledge base partitions are **:concepts**, **:relations**, and **:instances**.

Remarks:

Though concepts and relations are interned in the **:instances** partition (as well as their own partitions), **where-is-it** does not produce **:instances** entries for these objects.

Examples:

```
(where-is-it (fc Thing)) ⇒
    ((|C|THING UPPER-STRUCTURE-KB :CONCEPTS))
(where-is-it 'child) ⇒
    ((|C|CHILD USER-KB :CONCEPTS) (|R|CHILD USER-KB :RELATIONS))
(where-is-it 'Joe) ⇒
    ((|I|JOE USER-KB :INSTANCES) (|I|JOE TEST-KB :INSTANCES))
```

See Also:

fb (p. 87)
fc (p. 89)
fi (p. 92)
fr (p. 111)
list-kb (p. 172)

why *Macro*

Purpose:

The **why** macro generates a top-level explanation (in English) of the support for the truth of a given premise.

Syntax:

why *premise*

Arguments:

The **why** macro is syntactically identical to **ask**. Its *premise* argument is a restricted query-forming expression (see **ask** Remarks), which currently must have the form (*conceptName instanceId*).

Value:

The **why** macro returns **nil**.

Remarks:

The Loom explanation facility will be substantially expanded in the future. It is currently limited to queries about the type of an instance. It prints "Sorry, I should know but I don't" if the query goes beyond the implemented explanation capability.

Examples:

```
(why (Woman Sue)) —→
    SUE fills the role WIFE attached to the instance FRED; and
    The range of relation WIFE is WOMAN.
```

See Also:

ask (p. 20)

with-default-features *Macro*

Purpose:

The **with-default-features** macro effectively sets all Loom features to their default values, executes a body of code, and then resets the features to their original values.

Syntax:

with-default-features `&body` *body*

Arguments:

The *body* argument is one or more Lisp forms to be executed with the default feature settings.

Value:

The **with-default-features** macro returns the value of the last form in *body*.

Remarks:

The default features are: **:ask-about-redefinitions**, **:automatic-clos-classes**, **:clip-roles**, **:display-match-changes**, **:emit-classify-dots**, **:emit-match-stars**, **:open-world**, **:warn-about-syntax-corrections**, **:warn-on-nonprimitive-clos-classes**

Examples:

```
(with-default-features
    (do-this)
    (do-that))
```

See Also:

list-features (p. 167)
reset-features (p. 239)
with-feature-changes (p. 298)
with-features (p. 301)

with-feature-changes *Macro*

Purpose:

The **with-feature-changes** macro sets or unsets specified Loom features, executes a body of code, and then resets all features to their original values.

Syntax:

with-feature-changes (`&key` *set unset*) `&body` *body*

Arguments:

The *set* argument is a Loom feature, or a list of Loom features, to be temporarily set. The following features are available:

- If the **:ask-about-redefinitions** feature is set, Loom checks with the user before redefining a concept. This feature is set by default.
- If the **:ask-for-concept-names** feature is set, Loom prompts the user for concept names whenever system-defined concepts are generated. The prompting is done after the new concepts have been completed. If the user declines to supply a name, the concept is assigned a system-generated name.
- If the **:ask-for-names-immediately** feature is set, Loom asks the user to name system-defined conjunction concepts as soon as they are created and classified.
- If the **:automatic-clos-classes** feature is set, a CLOS class is automatically created whenever **make-object** (or **tell** or **create** with the **:clos** flag set) is called to instantiate a concept that does not already have an associated CLOS class. If this feature is disabled, a CLOS class is not created unless the concept explicitly contains the **:clos-class** characteristic. The **:automatic-clos-classes** feature is set by default.
- If the **:clip-roles** feature is set, clipping is performed on all single-valued roles. That is, if two (or more) different instances are asserted to be fillers of a single-valued role, Loom automatically retracts all but the last assertion. This feature is set by default.
- If the **:clip-types** feature is set, clipping of instance types is enabled. That is, if an instance's type is asserted to contain two (or more) concepts which are disjoint, or would otherwise cause the instance to become incoherent, then Loom automatically retracts all but the last assertion.

- If the **:display-match-changes** feature is set, whenever a production memory match or instance classification occurs, Loom indicates which concepts have entered or exited the type of each affected instance, and which rete nodes the instance has entered or exited. This feature is set by default.
- If the **:domain-implies-role** feature is set when Loom creates a shadowing CLOS class for a concept C, then the new class will have slots corresponding to every relation whose domain is C.
- If the **:dont-name-system-defined-concepts** feature is set, Loom does not generate a name for system-defined concepts and relations (provided **:ask-for-names-immediately** is also enabled).
- If the **:emit-classify-dots** feature is set, a dot, minus, or plus is printed when a concept is classified, unclassified, or completed, respectively, and an exclamation point is printed when a concept is sealed (i.e., ready to participate in data manipulation operations). Also, the character **M** is printed when concepts merge during classification, **U** is printed when they unmerge, and **C** is printed when a CLOS class is created. This feature is set by default.
- If the **:emit-match-stars** feature is set, an asterisk is printed whenever the type of an instance changes. Also, Loom prints a **T** whenever a new type is created as a result of unifying a newly-matched concept with the concepts already in an instance's type. This feature is set by default.
- If the **:expand-system-defined-print-names** feature is set, the printed representation of an instance has the form |**I**|*identifier*(*concepts*), where *concepts* contains the names of the most-specific user-named concepts which the instance matches. If this feature is not set, type information is omitted from the printed representation of the instance.
- If the **:prefer-clos-instances** feature is set, Loom automatically creates a CLOS instance when **tell** or **create** is called. If there is no CLOS class corresponding to the concept being instantiated, Loom attempts to create one. If this effort fails, a standard Loom instance is created, and a warning is issued (unless the *clos-instance-p* argument to **create** was **:no-warning**).
- If the **:suppress-diehard-output** feature is set, all Loom informational and warning output is suppressed. For example, **save-kb** typeout, classify dots, type-change messages, and incoherence warnings are all inhibited. It is advisable to set this feature when loading very large knowledge bases because of the overhead associated with lengthy typeout (especially when running under Emacs).
- If the **:warn-about-syntax-corrections** feature is set, a warning is issued whenever a syntax error is detected in a concept definition or in a query. This feature is set by default.
- If the **:warn-on-nonprimitive-clos-classes** feature is set, a warning is issued whenever a CLOS class is created to shadow a non-primitive concept. This feature

is set by default.

The *unset* argument is a Loom feature, or a list of Loom features, to be temporarily unset. Available features are listed above.

The *body* argument is one or more Lisp forms to be executed with the indicated feature settings.

Value:

The **with-feature-changes** macro returns the value of the last form in *body*.

Remarks:

The **:open-world** and **:closed-world** features cannot be temporarily set or unset.

Examples:

```
(with-feature-changes (:set :ask-for-concept-names
                       :unset '(:emit-match-stars :display-match-changes))
    (do-this)
    (do-that))
```

See Also:

list-features (p. 167)
set-features (p. 258)
unset-features (p. 286)
with-default-features (p. 297)
with-features (p. 301)

with-features *Macro*

Purpose:

The **with-features** macro effectively unsets all Loom features, sets only a specified group of features, executes a body of code, and then resets all features to their original values.

Syntax:

with-features (`&rest` *features*) `&body` *body*

Arguments:

The *features* argument is a Loom feature, or a list of Loom features, to be temporarily set. The following features are available:

- If the **:ask-about-redefinitions** feature is set, Loom checks with the user before redefining a concept. This feature is set by default.
- If the **:ask-for-concept-names** feature is set, Loom prompts the user for concept names whenever system-defined concepts are generated. The prompting is done after the new concepts have been completed. If the user declines to supply a name, the concept is assigned a system-generated name.
- If the **:ask-for-names-immediately** feature is set, Loom asks the user to name system-defined conjunction concepts as soon as they are created and classified.
- If the **:automatic-clos-classes** feature is set, a CLOS class is automatically created whenever **make-object** (or **tell** or **create** with the **:clos** flag set) is called to instantiate a concept that does not already have an associated CLOS class. If this feature is disabled, a CLOS class is not created unless the concept explicitly contains the **:clos-class** characteristic. The **:automatic-clos-classes** feature is set by default.
- If the **:clip-roles** feature is set, clipping is performed on all single-valued roles. That is, if two (or more) different instances are asserted to be fillers of a single-valued role, Loom automatically retracts all but the last assertion. This feature is set by default.
- If the **:clip-types** feature is set, clipping of instance types is enabled. That is, if an instance's type is asserted to contain two (or more) concepts which are disjoint, or would otherwise cause the instance to become incoherent, then Loom automatically retracts all but the last assertion.

- If the **:display-match-changes** feature is set, whenever a production memory match or instance classification occurs, Loom indicates which concepts have entered or exited the type of each affected instance, and which rete nodes the instance has entered or exited. This feature is set by default.
- If the **:domain-implies-role** feature is set when Loom creates a shadowing CLOS class for a concept C, then the new class will have slots corresponding to every relation whose domain is C.
- If the **:dont-name-system-defined-concepts** feature is set, Loom does not generate a name for system-defined concepts and relations (provided **:ask-for-names-immediately** is also enabled).
- If the **:emit-classify-dots** feature is set, a dot, minus, or plus is printed when a concept is classified, unclassified, or completed, respectively, and an exclamation point is printed when a concept is sealed (i.e., ready to participate in data manipulation operations). Also, the character **M** is printed when concepts merge during classification, **U** is printed when they unmerge, and **C** is printed when a CLOS class is created. This feature is set by default.
- If the **:emit-match-stars** feature is set, an asterisk is printed whenever the type of an instance changes. Also, Loom prints a **T** whenever a new type is created as a result of unifying a newly-matched concept with the concepts already in an instance's type. This feature is set by default.
- If the **:expand-system-defined-print-names** feature is set, the printed representation of an instance has the form |**I**|*identifier*(*concepts*), where *concepts* contains the names of the most-specific user-named concepts which the instance matches. If this feature is not set, type information is omitted from the printed representation of the instance.
- If the **:prefer-clos-instances** feature is set, Loom automatically creates a CLOS instance when **tell** or **create** is called. If there is no CLOS class corresponding to the concept being instantiated, Loom attempts to create one. If this effort fails, a standard Loom instance is created, and a warning is issued (unless the *clos-instance-p* argument to **create** was **:no-warning**).
- If the **:suppress-diehard-output** feature is set, all Loom informational and warning output is suppressed. For example, **save-kb** typeout, classify dots, type-change messages, and incoherence warnings are all inhibited. It is advisable to set this feature when loading very large knowledge bases because of the overhead associated with lengthy typeout (especially when running under Emacs).
- If the **:warn-about-syntax-corrections** feature is set, a warning is issued whenever a syntax error is detected in a concept definition or in a query. This feature is set by default.
- If the **:warn-on-nonprimitive-clos-classes** feature is set, a warning is issued whenever a CLOS class is created to shadow a non-primitive concept. This feature

is set by default.

The *body* argument is one or more Lisp forms to be executed with the indicated feature settings.

Value:

The **with-features** macro returns the value of the last form in *body*.

Remarks:

The **:open-world** and **:closed-world** features cannot be temporarily set.

Examples:

```
(with-features (:emit-match-stars :display-match-changes)
    (do-this)
    (do-that))
```

See Also:

list-features (p. 167)
set-features (p. 258)
unset-features (p. 286)
with-default-features (p. 297)
with-feature-changes (p. 298)

4 Loom Grammar

loom-expr ::=
definition |
statement ;
definition ::=
(DEFCONCEPT ConceptName concept-clause*) |
(DEFSET SetName set-clause*) |
(DEFRELATION RelationName relation-clause*) |
(IMPLIES {concept-expr concept-expr | relation-expr relation-expr}) |
(DEFAULT concept-expr concept-expr) |
(DEFACTION ActionName (Var*) action-clause*) |
(DEFMETHOD ActionName (Var*) method-clause+) |
(DEFPRODUCTION ProductionName production-clause+) |
(DEFKB KbName (KbName*) kb-clause*) ;
concept-clause ::=
{:IS | :IS-PRIMITIVE | :CONSTRAINTS | :DEFAULTS} concept-expr |
{:PARTITIONS | :EXHAUSTIVE-PARTITIONS} {partition-expr | (partition-expr+)} |
:IN-PARTITION PartitionName |
:PREDICATE ((Var) LispForm+) |
:FUNCTION (( ) LispForm+) |
:ROLES {RelationName | (RelationName+)} |
{:INDICES | :KEYS} ({RelationName | (RelationName+)}+) |
{:CLASS-NAME | :EXISTING-CLASS-NAME} ClassName |
:MIXIN-CLASSES {mixin-class | (mixin-class+)} |
:MIXIN-SLOTS {SlotName | (SlotName+)} |
:ANNOTATIONS ({ConceptName | (RelationName {InstanceId | Constant})}+) |
:IDENTIFIER Symbol |
:KB KbName |
:CHARACTERISTICS {concept-characteristic | (concept-characteristic+)} ;
concept-expr ::=
ConceptName |
({:AND | :OR} concept-expr+) |
(:ONE-OF {Number+ | InstanceId+}) |
(:THROUGH Number Number) |
({:AT-LEAST | :AT-MOST | :EXACTLY} Integer relation-expr) |
({:ALL | :SOME | :THE} relation-expr ConceptName) |
({:FILLED-BY | :NOT-FILLED-BY} relation-expr {InstanceId | Constant}+) |
({:SAME-AS | :SUBSET} relation-expr relation-expr) |
({< | > | <= | >= | = | / =} relation-expr {relation-expr | Number}) |
(:RELATES relation-expr relation-expr {relation-expr | Constant}) |
(:SATISFIES (?Var) query-expr) |
set-expr ;
partition-expr ::=
PartitionName |
(PartitionName ({ConceptName+ | RelationName+})) ;
mixin-class ::=
INSTANCE-WITH-CONCEPTS | INSTANCE-WITH-DYNAMIC-SLOTS |
INSTANCE-IN-MATCH-NETWORK | INSTANCE-IN-KNOWLEDGE-BASE |
INSTANCE-WITH-INVERSES | INSTANCE-WITH-EVERYTHING ;

```
concept-characteristic ::=
    :BACKWARD-CHAINING | :CLOS-CLASS | :NON-EXPORTED | :CLOSED-WORLD |
    :MONOTONIC | :PERFECT ;
set-clause ::=
    :IS set-expr |
    :ANNOTATIONS ( {ConceptName | ( RelationName {InstanceId | Constant} )}+ ) |
    :IDENTIFIER Symbol |
    :KB KbName ;
set-expr ::=
    SetName |
    ( {:AND | :OR} set-expr+ ) |
    ( {:ONE-OF | :THE-ORDERED-SET} Symbol+ ) |
    ( :THROUGH Symbol Symbol ) ;
relation-clause ::=
    {:IS | :IS-PRIMITIVE | :CONSTRAINTS} relation-expr |
    {:DOMAIN | :RANGE} concept-expr |
    :DOMAINS ( concept-expr concept-expr+ ) |
    :ARITY Integer |
    :PREDICATE ( ( Var Var+ ) LispForm+ ) |
    :FUNCTION ( ( Var+ ) LispForm+ ) |
    :INHERITANCE-LINK RelationName |
    :INHERITANCE-METHOD {:NEAREST-NEIGHBOR | :UNION} |
    :ANNOTATIONS ( {ConceptName | ( RelationName {InstanceId | Constant} )}+ ) |
    :IDENTIFIER Symbol |
    :KB KbName |
    :CHARACTERISTICS {relation-characteristic | ( relation-characteristic+ )} ;
relation-expr ::=
    RelationName |
    ( :AND relation-expr+ ) |
    ( {:DOMAIN | :RANGE} concept-expr ) |
    ( :DOMAINS concept-expr concept-expr+ ) |
    ( :INVERSE relation-expr ) |
    ( :COMPOSE relation-expr+ ) |
    ( :SATISFIES ( ?Var ?Var+ ) query-expr ) ;
relation-characteristic ::=
    :SINGLE-VALUED | :MULTIPLE-VALUED | :CLIP-ROLES | :SYMMETRIC | :COMMUTATIVE |
    :READ-ONLY | :READ-WRITE | :CACHE-COMPUTATION | :HASH-ON-DOMAINS |
    :NON-EXPORTED | :OPEN-WORLD | :CLOSED-WORLD | :MONOTONIC | :PERFECT ;
action-clause ::=
    :FILTERS {filter | ( filter+ )} |
    :MISSING-METHOD {:NO-OP | :WARNING | :ERROR} ;
filter ::=
    :MOST-SPECIFIC | :LAST-ONE | :SELECT-ONE | :SELECT-ALL | :OVERRIDES |
    :WARNING | :ERROR ;
method-clause ::=
    :TITLE Title |
    :SITUATION query-expr |
    :WITH query-expr |
    :OVERRIDES {Title | ( Title+ )} |
    :RESPONSE ( LispForm+ ) ;
```

production-clause ::=
    :WHEN { production-expr | (:AND production-expr+ query-expr*)} |
    {:PERFORM | :SCHEDULE} (ActionName Arg+) |
    :DO (LispForm+) |
    :PRIORITY {:LOW | :HIGH} ;
production-expr ::=
    ({:DETECTS | :UNDETECTS} {production-pattern | (:AND production-pattern+)}) |
    (:CHANGES (RelationName ?Var)) ;
production-pattern ::=
    (ConceptName ?Var) |
    (RelationName ?Var ?Var) ;
kb-clause ::=
    :PATH-NAME PathName |
    :PACKAGE PackageName |
    :EXPORT-NAMES-P Boolean |
    :MONOTONIC-P Boolean ;
statement ::=
    ({TELL | TELLM | FORGET | FORGETM} proposition*) |
    (ASK query-expr) |
    (RETRIEVE {?Var | (?Var+)} query-expr) |
    ({PERFORM | SCHEDULE} (ActionName Arg+)) ;
proposition ::=
    (concept instance) |
    (relation instance+ value) |
    (:CREATE ?Var concept [:CLOS]) |
    (:SAME-AS instance instance) |
    (:ABOUT instance about-clause*) ;
query-expr ::=
    ({:AND | :OR} query-expr+) |
    ({:NOT | :FAIL} query-expr) |
    (:IMPLIES query-expr query-expr) |
    ({:FOR-SOME | :FOR-ALL} (?Var+) query-expr) |
    (:COLLECT (?Var) query-expr) |
    (concept instance) |
    (relation instance+ value) |
    (:SAME-AS instance instance) |
    (:SUBSET instance instance) |
    (:PREDCALL LispPredicate value+) |
    (:ABOUT instance about-clause*) ;
about-clause ::=
    concept |
    (concept) |
    (relation value) |
    (:FILLED-BY relation value+) |
    (:FILLED-BY-LIST relation List) |
    ({:AT-LEAST | :AT-MOST | :EXACTLY} Integer relation) |
    ({:ALL | :SOME | :THE} relation concept) ;

concept ::=
    ConceptName |
    ?Var ;
relation ::=
    RelationName |
    ?Var ;
instance ::=
    InstanceId |
    (relation instance) |
    ?Var ;
value ::=
    instance |
    Constant |
    (:ONE-OF Constant+) ;

Index