Short Algernon Reference Manual

For Algernon Version 1.3.0 *

B. J. Kuipers†
J. M. Crawford‡

January 18, 1994

Contents

*This work has taken place in the Qualitative Reasoning Group at the Artificial Intelligence Laboratory, The University of Texas at Austin. Research of the Qualitative Reasoning Group is supported in part by the Texas Advanced Research Program under grant no. 003658-175, NSF grants IRI-8905494 and IRI-8904454, and by NASA grant NAG 2-507.

†Department of Computer Sciences, The University of Texas At Austin, Austin, Texas 78712 (kuipers@cs.utexas.edu).

‡Computational Intelligence Research Laboratory (CIRL), Computer and Information Sciences Department, University of Oregon, Eugene, OR 97403 (jc@cs.uoregon.edu).

1 Knowledge Representation in Algernon

1.1 Facts, Formulas, and Frames

The set of facts in an Algernon knowledge base can be viewed in two different ways.

- Viewed as a logic, the KB contains a collection of ground atomic formulas, representing predicates applied to constant symbols.
- Viewed as a semantic network, the KB contains a collection of frames describing individual objects, linked by known relations.

An advantage of the logic view is the vast body of relevant theory, useful for providing guarantees of expressive and inferential power. An advantage of the semantic network view is that the organization of knowledge into frames provides direct access to relevant information, and supports tractable reasoning. The (conceptual) objects related to a frame can be easily accessed by looking in a slot of the frame without searching the entire knowledge-base.

In Algernon, these two views are equivalent, since the ground atomic formula $r(f,g)$ is represented by placing the value `g` in the `values` facet of the `r` slot of the frame `f` [Hayes, 1979]. Similarly, if we know $\neg r(f,h)$, the value `h` is placed in the `non-values` facet of the `r` slot of the frame `f`.

$r(f,g) \wedge \neg r(f,h) \quad \equiv$

```
f:
    r:
        values: { ...g... }
        non-values: { ...h... }
```

1.2 True Names and Public Names

A frame has two different types of name.

- Every frame has a single *true name* (tname), which is a symbol referring uniquely to that frame. In the Lisp implementation, the tname is the Lisp symbol on whose property list the frame structure is stored.
- A frame can have any number of *public names* (pnames), which are strings, and need not be uniquely referring. Public names are stored as values in the `name` slot of the frame. An indexing mechanism (case insensitive) allows frames to be retrieved given a public name.

True names are the pointers from one frame to another that actually represent the structure of the knowledge base. Public names are for communication with other agents.

For ease of debugging and interaction, when a new frame has a simple public name, Algernon attempts to generate a true name that has the same or similar printed representation. Similarly, the user interface attempts to retrieve the desired frame, whether you type its true name or its public name.

1.3 Access Paths

Facts are retrieved from an Algernon knowledge-base by following *access paths* of alternating frames and slots, in which each frame is only accessed if it appears in a known slot of a known frame. Syntactically, a sequence of atomic formulae defines an access path iff any variable appearing as a predicate-symbol or as the first argument to a predicate has appeared previously in the sequence.

An Algernon knowledge base contains rules as well as facts. Rules are of two types:

- forward-chaining (*if-added*) rules, applied when a new fact is asserted into the knowledge-base;
- backward-chaining (*if-needed*) rules, applied when an unknown fact is queried.

In almost all cases, a rule is associated with the slot of the frame specified in its *key*: the left-most atomic formula in the rule. (The exceptions are *generic rules*, associated with sets of slots, where the specific slot is determined by the actual assertion or query.) When a rule is invoked, the asserted or queried formula is unified against the key. Starting with these bindings, the antecedent of the rule must be an access path. The antecedent is queried, and the consequent is asserted for every set of bindings resulting from the query of the antecent. (See *Algernon for Expert Systems* for examples.)

2 The Syntax and Semantics of Algernon

2.1 The Formal Syntax of Algernon

Notation: The *italic font* is used for non-terminals and the `teletype font` is used for terminals, (*i.e.*, strings that actually occur in Algernon code).

[**x**] denotes zero or one appearance of x.

x* denotes zero or more repetitions of x.

x$^+$ denotes one or more repetitions of x.

atomic-formula = (*sterm fterm vterm*$^+$) | (`not` (*sterm fterm vterm*$^+$))
fterm = *variable* | *frame* | (*sterm fterm*) | (`:slot` *slot*)
sterm = *variable* | *slot*
vterm = *fterm* | *number* | *string* | (`:quote` *expression*) | (`quote` *expression*)

domain = *frame* | `:number` | `:string` | `:symbol` | `:list`

path = (*form*$^+$) plus access path restriction on variables.

rule = (*form*$^+$ `<-` *form**) |
(*form*$^+$ `->` *form**)

vars = *variable* | (*variable*$^+$)

form = *atomic-formula* |

(`:taxonomy` *set-descriptor*)

(`:slot` *symbol* (*domain*$^+$) *slot-descriptor**) |

(`:rules` *fterm rule*$^+$) |
(`:srules` *fterm rule*$^+$) |

(`:eval` *expression*) |
(`:test` *expression*) |
(`:boundp` *variable*) |
(`:unboundp` *variable*) |
(`:bind` *variable expression*) |
(`:branch` *variable expression*) |
(`:funcall` *function expression**) |

(`:a` *vars* . *path*) |
(`:forc` *vars* . *path*) |
(`:the` *vars* . *path*) |
(`:any` . *path*) |
(`:cut` . *path*) |
(`:db` *atomic-formula*) |
(`:retrieve` *atomic-formula*) |

(`:unp` . *path*) |
(`:fail` . *path*) |
(`:assume` *atomic-formula*) |
(`:neq` *fterm fterm*) |

(`:or` *path*$^+$) |
(`:all-paths` *path path*) |

(`:delete` *atomic-formula*) |
(`:clear-slot` *fterm slot*) |
(`:del-rule` *fterm rule*) |
(`:del-srule` *fterm rule*) |
(`:del-rules` *fterm slot*) |
(`:del-srules` *fterm slot*) |

(`:ask` *atomic-formula*) |
(`:show` *fterm*) |

(`:w-contra-positive` . *path*) |
(`:wo-contra-positive` . *path*) |
(`:no-completion` *form*)

slot-descriptor = `:cardinality` *number* |
`:backlink` *slot* |
`:inverse` *slot* |
`:comment` *string*

set-descriptor = (*fterm set-descriptor** *fterm**)

expression = Any Lisp expression |
(`:values` *frame slot*) |
(`:non-values` *frame slot*)

variable = a Lisp symbol whose print name begins with "`?`".

frame = a Lisp symbol that represents a frame in the knowledge-base.
slot = a Lisp symbol declared as a slot in the knowledge-base.
number = a Lisp number.
string = a Lisp string.
symbol = a Lisp symbol.
function= a Lisp function (in a form suitable to be passed to the Lisp function `funcall`).

2.2 Special Forms

Algernon supports a number of "special forms". This section provides a description of these forms and examples of their use.

Most forms have similar behavior in assertions and queries. The form `:all-paths` is the only serious exception to this rule. All other forms have exactly the same behavior in assertions and queries.

2.2.1 Declaring a Taxonomy of Frames

(`:taxonomy` *set-descriptor*)

Adds to the basic taxonomic structure of the knowledge-base. Sets are described by lists whose atomic elements are (names of) the members of the sets and whose sublists are subsets of the set. The set forming the root of the taxonomy (`set1` in the example below) must exist in the knowledge base when the `:taxonomy` form is asserted.

The sets and elements in a `:taxonomy` form must have uniquely designating names.

The form:

```
(:taxonomy (set1
            (set2 frame1 frame2)
            (set3 frame3)))
```

requires that `set1` already exists in the knowledge base; asserts that `set2` and `set3` are subsets of `set1`, not necessarily disjoint; that `frame1` and `frame2` are elements of `set2`; and that `frame3` is an element of `set3`.

2.2.2 Declaring Slots and their Domains

(`:slot` *atom* (*domain*$^+$) *descriptor**)

Declares a new slot and types it using *domain*$^+$. Typing is enforced using an if-added rule. For example:

```
(:slot has-disease (people diseases))
```

declares the slot `has-disease` to be a relation between people and diseases. If one then asserts:

```
(has-disease p1 d1)
```

Algernon concludes that `p1` isa `people`, and `d1` isa `diseases`.

In general, domains must be Algernon sets, defined by the time the slot is accessed. A warning is given if a domain is undefined when the `:slot` is asserted. However, a domain may also be declared to be one of the Lisp datatypes, `:number`, `:string`, `:symbol` or `:list`. The variable `*lisp-type-domains*` holds the current complete list of permissible Lisp datatypes. The domain specification `nil` is completely unconstrained.

`:slot` takes the following "keyword" arguments:

`:cardinality` *n* Asserts that the slot holds at most *n* values. Defaults to `nil` (no restriction). In the current implementation, cardinality information is primarily used to limit the application of if-needed rules (if a slot is queried but is full then the rules are not applied).

`:backlink` *slot2* Backlinks the slot to *slot2*. This means that if Algernon learns `(slot f1 f2)` it will conclude (via an if-added rule) (*slot2* `f2 f1`). *slot2* must already be defined.

`:inverse` *slot2* Asserts that the inverse of the slot is *slot2*. This is equivalent to bi-directional backlinks.

`:comment` *string* Adds a comment for the slot.

(`:declare-slot` *atom* (*domain*$^+$) [`:cardinality` *number*])

This is a limited-purpose version of `:slot`, allowing only the `:cardinality` keyword, used in the early stages of setting up the knowledge-base.

See section 4 for examples of the use of `:taxonomy` and `:slot`.

2.2.3 Declaring Rules

(`:rules` *fterm* *rule*$^+$)

(`:srules` *fterm* *rule*$^+$)

These two forms add rules, associated with the frame referred to by *fterm*, to the knowledge-base. The only difference between them is the way in which the rules are indexed (and therefore in the way they will later be accessed). Consider a query of `(r frame1 ?x)` (assertions are similar). Such a query will access rules from several sources:

1. It will access *slot* rules (rules added using `:srules`) associated with the slot `r`.
2. It will access 'normal' rules (rules added using `:rules`) associated with any set which `frame1` is a member of (*i.e.*, rules associated with any frame `f` such that `(isa frame1 f)` is a fact in the knowledge-base).
3. Finally, it will access slot rules associated with any set that the slot `r` is known to be a member of.

As an example of a set of slots, consider the set `transitive-relations`. For this set one might want a rule like:

```
(:srules transitive-relations
    ((?r ?x ?z) <- (?r ?x ?y) (?r ?y ?z)))
```

One could then assert that `(isa (:slot less) transitive-relations)` and the rule above would enable Algernon to conclude `(less frame1 frame3)` from `(less frame1 frame2)` and `(less frame2 frame3)`.

Note that this rule for transitive relations has a variable in the *slot* position. Such rules are called *generic* rules and can be associated with sets of frames or slots (though obviously not with individual slots). Recall that at the time of a query or assertion, both the frame and the slot being accessed are known.

2.2.4 Calling Algernon from Lisp

An Algernon knowledge-base can be regarded as the knowledge about the world possessed by an individual agent. Algernon interacts with the world through a *tell/ask* (or equivalently, *assert/query*) interface by which we tell things to Algernon and ask questions of the knowledge-base.

```
tell path &key :retrieve :eval :collect :comment
ask path &key :retrieve :eval :collect :comment
```

The functions `tell` and `ask` take an access path and assert or query it to the Algernon knowledge base. These operations may branch on multiple bindings while following the path. If the keyword `:retrieve` is `t`, only facts explicitly stored in the KB are retrieved, and no backward-chaining rules are invoked. If no sets of bindings are found, the operation fails and `nil` is returned.

If the operation succeeds, then the Lisp forms provided for the `:eval` and `:collect` keywords are instantiated with values substituted for Algernon variables in each binding found. The `:eval`

forms are evaluated, and the values of the `:collect` forms are collected and returned. If no `:collect` form is provided, `t` is returned after a successful operation. The `:comment` string may be printed as trace output.

For example, if the brothers of Tom are Bob and Mike then:

```
(ask '((brother Tom ?x)) :collect '?x)
```

will return: `(Bob Mike)`. If further, Bob drives a Honda and Mike drives a Ford then:

```
(ask '((brother Tom ?x) (drives ?x ?c))
     :collect '(?x drives ?c))
```

will return:

```
((Bob drives Honda)
 (Mike drives Ford))
```

The `a-assert` and `a-query` functions are obsolete, but still supported. They call `tell` and `ask`, respectively, with trace output turned on.

`a-assert` *string path*
`a-query` *string path*

where `string` is a comment string, and `path` is an Algernon path (defined formally in section 2.1 above).

2.2.5 Calling Lisp from Algernon

The special forms in this section allow a Lisp expression to be evaluated, and the value used in Algernon in various ways. Before an expression is evaluated in Lisp, any Algernon variables appearing in the expression are replaced by their bindings.

(`:eval` *expression*)

Evaluates *expression* as a Lisp expression for side effects. Returned value is ignored. Thus one can print out the children of `Tom` using the path:

```
((child Tom ?x) (:eval (format t "A child of Tom is ~a.~%" '?x))).
```

(`:test` *expression*)

Evaluates *expression* as a Lisp expression and succeeds iff it does not evaluate to `nil`.

(`:boundp` *variable*)

Succeeds if *variable* is bound to a value (not another variable), and fails otherwise.

(`:unboundp` *variable*)

Succeeds if *variable* is unbound, and fails otherwise.

(`:funcall` *function* $exp_1 \ldots exp_n$)

Applies *function* to the values obtained by evaluating $exp_1 \ldots exp_n$. Each of the exp_i may be of one of four forms:

`:assumptions` — returns the list of lists of assumptions currently in force, in case the Lisp function needs them for something.

(`:values` *frame slot*) — evaluates to a list of the values stored in *slot* of *frame*, stripped of the associated assumptions.

(`:non-values` *frame slot*) — evaluates to a list of the explicit non-values in *slot* of *frame*, stripped of their associated assumptions.

otherwise — evaluated as a Lisp expression.

When `:funcall` appears as a top-level special form in a path, the value returned is ignored by Algernon. However, `:funcall` can also be embedded in `:bind` or `:branch` forms, in which case the value returned can be bound to Algernon variables.

(`:bind` *variable expression*)

(`:bind` (*variable*$^+$) *expression*)

After substituting in the values of any Algernon variables, evaluate the second argument and unify the result with the first argument. The *expression* may be a Lisp expression or an embedded `:funcall` expression. For example, to determine the number of values in a slot:

```
(:bind ?n (:funcall #'length (:values frame slot)))
```

Because the *variable* argument is *unified* with the result of evaluating *expression*, `:bind` can be used to destructure and test properties of the value returned. For example, suppose the Lisp function `(gossip)` returns a list of the form `(Tom loves Mary)` or `(Bill hates Joe)` or some such. We could select for a particular case, and bind variables to the names of the protagonists with:

```
(:bind (?subj loves ?obj) (gossip))
```

The value returned from the Lisp function is captured using `multiple-value-bind`, with the second argument being the set of assumptions. This means that a Lisp function could store information in an external datastructure, indexed under the appropriate assumptions, using rules something like the following:

```
((R ?x ?y)  ->  (:funcall #'Fout '?x '?y :assumptions))
((R ?x ?y)  <-  (:bind ?y (:funcall #'Fin '?x :assumptions)))
```

(This has not been tested, and the returned values will be cached in the slot, which may not be what we want.)

(`:branch` *variable expression*)

(`:branch` (*variable*$^+$) *expression*)

`:branch` works just like `:bind`, except that the result of evaluating *expression* must be a list of values, and *variable* is unified against each of those values along a separate branch.

Since parts of these forms are evaluated by Lisp after Algernon variables are replaced by their values, and since Algernon frames are represented by Lisp symbols that are often unbound, it is important to quote variables whose bindings shouldn't be evaluated. For example, if `foo` is a function taking a single numerical argument, then

`((:bind ?f 'foo) (:bind ?n (:funcall '?f 2)))` or even
`((:bind ?f 'foo) (:bind ?n (?f 2)))` will work as expected, but
`((:bind ?f 'foo) (:bind ?n (:funcall ?f 2)))` will not.

2.2.6 Controlled Retrieval

There are six main controlled-retrieval forms, summarized as follows:

term	number of values returned	frames created?
(`:a` *variable(s) form**)	1	always
(`:the` *variable(s) form*$^+$)	1	on failure
(`:forc` *variable(s) form*$^+$)	1 to N	on failure
(`:any` *form*$^+$)	0 or 1	no
(`:cut` *form*$^+$)	0 or 1	no
(`:retrieve` *form*)	0 to N	no

(`:a` *variable* [*. path*])

(`:a` (*variable*$^+$) [*. path*])

`:a` creates new frames to bind to the specified variable(s), and asserts *path* about those frames. (`:a` generalizes and replaces `:create`.) For example, `(:a ?x (name ?x "Tom"))` will create a new frame, bind `?x` to it, and assert the value `"Tom"` into its `name` slot. Algernon's naming heuristics may also be able to use the Lisp atom `Tom` to hold the frame, but this is not guaranteed.

(`:forc` *variable . path*)

(`:forc` (*variable*$^+$) *. path*)

`:forc` implements "find-or-create" retrieval. `:forc` first queries *path*. If this query succeeds then the `:forc` succeeds (binding *variable*). If the path fails then a new frame is created, *variable* is bound to it, and the path is asserted. (`:forc` (*variable*$^+$) *. path*) is similar, but allows multiple variables.

One use of `:forc` is to guarantee that certain slots always hold some value. For example, one might express "Every car has a steering wheel." as:

```
(:rules cars
 ((steering-wheel ?c ?w) <- (:forc ?w (steering-wheel ?c ?w))))
```

Care must be taken with such rules, however, as they can easily cause infinite chains (*e.g.*, "Every man has a father") — see [Crawford, 90].

(`:the` *variable* . *path*)

(`:the` (*variable*$^+$) . *path*)

`:the` is just like `:forc` except that it fails if the query of its path returns multiple bindings for its variables.

It is used for definite descriptions, when a referring phrase is presumed to designate a unique entity.

(`:any` . *path*)

If there are multiple bindings found when following the path, an arbitrary one is selected and returned. Used to force a unique element, or failure.

(`:cut` . *path*)

Like `:any`, returns a single binding on success, but does depth-first search through the path to find a single binding, unlike most Algernon retrieval which does breadth-first retrieval of all bindings. The Algernon rule `(p <- (:cut a b) c d)` should be roughly equivalent to `p :- a,b,!,c,d` in Prolog.

(`:retrieve` *atomic-formula*)

(`:db` *atomic-formula*)

`:retrieve` (and its synonym `:db`) suppress the usual application of deduction rules while querying its atomic-formula. It should be used only in queries. `:retrieve` distributes over functional forms: `((:retrieve (p (f ?x) ?y)))` expands to `((:retrieve (f ?x ?$X2)) (:retrieve (p ?$X2 ?y)))`.

2.2.7 Negation and Non-Monotonic Reasoning

Non-monotonic reasoning is still a research topic in Algernon. To use non-monotonic forms like `:unp` and `:assume` successfully, you will probably have to understand something of the internals of how Algernon does inference (see section 2.3).

(`:neq` *fterm1 fterm2*)

Succeeds iff *fterm1* $\neq$ *fterm2*. (`:neq` *fterm1 fterm2*) is equivalent to `(:eval (not (eql '`*fterm1* `'`*fterm2*`)))`

(`:fail` . *path*)

(`:unp` . *path*)

Unprovable. Succeeds exactly when a query of *path* fails. `:unp` is used primarily in default rules. For example:

```
(:rules Birds
 ((flies ?x True) <- (:unp (not (flies ?x True)))
                     (:assume (normal ?x Birds flies))))
```

(`:assume` is discussed below).

The idiom (`:fail (:retrieve` *atomic-formula*`)`) succeeds if *atomic-formula* is not explicitly stored in the knowledge base, without applying any rules.

(`:assume` *atomic-formula*)

Adds *atomic-formula* to the knowledge-base as an assumption. Assumptions differ from facts in two ways. First, an attempt is made to prove the negation of the atomic-formula, and if this attempt succeeds, the `:assume` fails. Second, any future conclusion which depends on the assumption is tagged with the assumption, so that if the assumption is later withdrawn the conclusion is also withdrawn.

2.2.8 Control Structure

(`:or` $path_1 \ldots path_n$)

`:or` queries its paths in order, returning the stream of bindings produced by the first one that succeeds. The following paths are not queried at all.

(`:all-paths` *path1 path2*)

`:all-paths` first queries *path1*. If it appears within a query, it then queries *path2* under every substitution generated by the first query and succeeds iff all of these queries succeed. If it appears within an assertion it asserts *path2* under every substitution generated by the first query and succeeds iff all of these assertions succeed. For example, one could query "Are all Adam's children male?" as:

```
((:all-paths ((child Adam ?x)) ((gender ?x male))))
```

Note that, like `:unp`, `:all-paths` and `:or` are non-monotonic.

2.2.9 Deleting Things

All of these operations are non-monotonic, and should only be used with great care.

(`:delete` *atomic-formula*)
Removes *atomic-formula* from the knowledge-base.

(`:clear-slot` *fterm slot*)
Removes *all* values from slot *slot* of the frame referred to by *fterm*.

(`:del-rule` *fterm rule*)

(`:del-srule` *fterm rule*)
Used to delete a rule (or slot rule) from a frame.

(`:del-rules` *fterm slot*)

(`:del-srules` *fterm slot*)

Used to delete all rules (or slot rules) from a slot of a frame.

2.2.10 Interacting with the User

These primitive forms are included only to make it easy to create self-contained demonstration examples. The correct way to interact with the user is to write a user interface in Lisp and call it using `:test`, `:bind`, or `:branch`.

(`:show` *fterm*)

Print the contents of the frame referred to by *fterm*. Any slots on the list `*dont-print-slots*` are not printed by `:show` or the interface command `visit-frame`.

(`:ask` *atomic-formula*)

Asks user for a value for *atomic-formula*. If *atomic-formula* is ground then Algernon simply asks the user if *atomic-formula* is true. If the user answers `yes` then the atomic-formula is asserted and the `:ask` succeeds. If the user answers `no` then Algernon concludes the negation of the atomic-formula and the `:ask` fails.

If the atomic-formula is not ground then Algernon asks for a value for the variable in the atomic-formula. If the slot of the atomic-formula is typed to hold values from a set, and the members of the set are known, then Algernon requires the user to enter a value in the set.

2.2.11 Controlling Style of Inference

(`:retrieve` *atomic-formula*)

(`:db` *atomic-formula*)

`:retrieve` (and its synonym `:db`) succeeds if the atomic-formula is already in the knowledge base. It does not apply any further rules to attempt to deduce the atomic-formula. `:retrieve` should be used only in queries. It distributes over functional forms: `((:retrieve (p (f ?x) ?y)))` expands to `((:retrieve (f ?x ?$X2)) (:retrieve (p ?$X2 ?y)))`.

(`:w-contra-positive` . *path*)

(`:wo-contra-positive` . *path*)

These forms are used to enable and disable the addition to the knowledge-base of the contra-positives of rules. The default is *not* to add contra-positives. Contra-positives are used to minimize proofs by contradiction when reasoning with disjunctions. (An example of the use of disjunction is given in [Crawford, 90].)

(`:no-completion` . *path*)

Suppresses rule completion (see section 2.3.1).

2.2.12 Obsolete Forms

The following slots are obsolete or for internal use only. Any use should be replaced by the indicated form.

Obsolete form	Current form	Section
`:decl-slots`	`:slot`	section 2.2.2
`:lisp`	`:eval`	section 2.2.5
`:bind-to-values`	`:bind`	section 2.2.5
`:branch-on-values`	`:branch`	section 2.2.5
`:apply`	`:funcall`	section 2.2.5
`:create`	`:a`	section 2.2.6
`:in-own-partition`	none	none

2.3 Notes on Inference in Algernon

2.3.1 Rule Completion

In ALL, the application of an if-added rule is triggered by the assertion of a fact (into a slot from which the rule can be accessed) which matches the first atomic-formula of the antecedent of the rule. This contrasts with some expert system shells in which every rule is applied whenever a new fact is asserted. One potential problem with this approach is that, if one is not careful, it can be the case that accessible rules, whose antecedents are entailed by the knowledge-base, may never fire.

For example, consider a knowledge-base including the rule:

```
((r1 ?x ?y) (r2 ?y ?z) -> (r3 ?x ?z))
```

Suppose we assert the fact `(r1 frame1 frame2)`, but the rule fails because no fact in the knowledge-base matches `(r2 frame2 ?z)`. Later, if we assert `(r2 frame2 frame3)`, the rule will not be triggered. We refer to this problem as *if-added incompleteness*. Algernon's solution to this problem is to *complete* the rule (when `(r1 frame1 frame2)` is added), by adding the shortened rule:

```
((r2 frame2 ?z) -> (r3 frame1 ?z))
```

This rule is added to the `selfset` of `frame2` (the selfset of a frame is the set consisting exactly of the frame). One might worry that rule completion would add a large number of rules to the knowledge-base and thus slow reasoning. However, since they are associated with selfsets, such rules are inaccessible to almost all operations.

One problem with rule completion is that it requires one to be careful when removing facts from the knowledge-base (always a very dangerous operation). In the case above, if you later delete the fact `(r1 frame1 frame2)`, using `(:clear-slot frame1 r1)` for example, the rule completion is *not* undone and the shortened rule remains. This is a symptom of the larger problem that Algernon expects to reason monotonically and is not prepared to 'roll back' the assertion of a fact. If you want to be able to later retract a fact then you must enter it as an assumption (see section 2.2 above).

3 Interactive Interface Commands

A full list of commands for the Algernon interactive interface is given below. The abbreviation for each command is given in parentheses.

For each Algernon command *com* listed here (except `exit`), there is an equivalent Lisp function called `acom-`*com* which takes the arguments shown for the command.

3.1 General Commands

`query (q) P`	Query path P in knowledge base. (Essentially equivalent to `(a-query string P)`.)
`assert (a) P`	Assert path P in knowledge base. (Essentially equivalent to `(a-assert string P)`.)
`help (?) Topic`	Prints help message on topic Topic (help help lists topics).
`reset`	Reset Algernon.
`clear-window (cw)`	Clear window.
`version (v)`	Prints out the current version number.
`profile-kb (pkb)`	Prints some statistics about the knowledge base.
`exit (e x quit)`	Exit Algernon.

3.2 Saving and Restoring the Knowledge Base

`dump-kb (dkb) name`	Record state of knowledge-base to a file.
`kb-snapshot (kbs) name`	Record the current state of the knowledge-base in memory. Equivalent to the Lisp function `(kb-snapshot string)`.
`write-snapshot (ws) name`	Write a snapshot to disk.
`list-snapshots (ls)`	List current snapshots.
`delete-snapshot (ds) name`	Delete a snapshot.
`delete-all-snapshots (das)`	Delete current snapshots.
`load-kb (lkb) name`	Load a saved knowledge-base (from disk or a snapshot).
`kb-directory (kbd)`	check/change directory for storing snapshots.

3.3 Browsing the Knowledge Base

`current-frame (cf)`	Show current frame.
`who (w) n`	Visit frames with name n (n a symbol or list).
`visit-frame (vf) f`	Visit frame f (f a symbol).
`visit-slot (vs) s`	Visit frames in slot s of the current frame
`visit-slot-nonv (vsn) s`	Visit frames in non-value facet of slot s of the current frame.
`pop (p)`	Pop back to last visited frame.
`next-frame (nf)`	Go to next frame in current frame list.
`previous-frame (pf)`	Go to previous frame in current frame list.
`rules (r) s`	Show all rules which apply to slots of current frame.

3.4 Tracing Algernon Inference

There are many trace switches in Algernon. The following commands turn on selections of them, providing decreasing amounts of information. `trace-logic` is perhaps the most useful for debugging.

`trace-all (ta)`	Trace everything (very verbose).
`trace-control (tc)`	Trace logic plus control queues.
`trace-logic (tl)`	Trace queries, assertions, and rule applications.
`trace-interest (ti)`	Trace interesting events (new facts and contradictions).
`trace-off (to)`	Turn off all tracing.
`default-trace (dt)`	Default trace (currently: trace off).

It can be useful to turn on tracing during an operation, using the Lisp versions of the trace commands. For example, to start the trace after it is learned that the color of `block` is red,

```
(:rules blocks
  ((color ?b red) -> (:eval (acom-trace-logic))))
```

3.5 Output Control

Algernon is also willing to describe the results of each operation.

`normal-output (no)`	Bindings and kb changes of operations shown.
`minimal-output (mo)`	Bindings resulting from operations shown.
`silent-output (so)`	Output silent unless operation fails.
`verbose-output (vo)`	Results, kb changes, and predicates of operations shown.
`last-op (lo)`	Show results of more recent operations.

3.6 Running Programs and Examples

`lisp (l) exp`	Evaluate the exp as a Lisp expression.
`load f`	Load file f (f a string).
`compile-load (cl) f`	Compile and load file f (f a string).
`load-ex (le) f`	Load example file f.
`run-ex (re) e`	Run example e.

3.7 Debugging

`backtrace (bt)` Print Algy operation stack.

You can also evaluate `(algy-backtrace)` in a Lisp error-break.

3.8 Useful Global Variables

`*dont-print-slots* (nil)`	Suppress printing of these slots.
`*cerror-on-failed-assert* (t)`	Continuable error on failure.
`*cerror-on-failed-query* (nil)`	Continuable error on failure.
`*search-strategy* (depth-first)`	Strategy for rule chaining.
`*special-forms*`	List of all legal special forms.

4 The Built-In Knowledge-Base

The built-in knowledge-base sets up the top level of the taxonomy as well as several basic slots. The source code for the knowledge base is in the file `akbase.lisp`.

4.1 Naming Conventions

The following conventions have been used in naming frames and slots:

- Names for *sets* are generally plural (*e.g.*, `people`) to distinguish them from individuals.
- Names for slots are generally chosen so that `(r f1 f2)` can be read as "f1 r f2" or "An r of f1 is f2" (depending on whether `r` is a verb or a noun).

4.2 Fundamental Objects and Relations

The basic structure of the knowledge base is organized around sets, their elements, and their subsets.

`(isa x S)`	$\equiv x \in S$	"x is an element of S."
`(member S x)`	$\equiv x \in S$	"An element of S is x."
`(subset A B)`	$\equiv A \supseteq B$	"A subset of A is B.
`(superset B A)`	$\equiv B \subseteq A$	"A superset of B is A.

Knowledge representation in Algernon exploits access limitations, so an individual will normally be linked explicitly to some of its containing sets, but most large sets will *not* be explicitly linked to their members.

The taxonomy of fundamental objects is set up as follows:

```
(:taxonomy (things
            (rules)
            (objects
             (sets things objects sets slots)
             (booleans true false :complete)
             (contexts global-context))
            (slots
             (order-relations
              (tc-order-relations
               (equivalence-relations))))))
```

Every frame in the knowledge-base is a member of the set `things`, which breaks down into the sets `objects` and `slots`, and so on. In the knowledge-base, the taxonomy is represented using `isa` and `superset` relations (and their inverses `member` and `subset`). `isa` links an object to a set of which it is a member, and `superset` links a set to a superset of it (thus `member` links a set to one of its members, and `subset` links a set to one of its subsets).

A basic design decision in building large taxonomies is whether to link an object in the taxonomy to every set it is a member of, to link it to just the 'lowest' set in the taxonomy it is a member of (and then link this set up to the sets higher up in the taxonomy using `superset` links), or to link it to some of the sets it is a member of. We have chosen to take the third course, and link objects to just the "important" sets they are a member of. This distinction is represented using the relations `superset` and `imp-superset`. Intuitively, `imp-superset` links a set to an 'important' superset. Operationally this means that if Algernon learns that an object, `x` is a member of a set `s`, and `s` has an important superset `S`, then a forward-chaining rule immediately adds an `isa` link between `x` and `S`.

```
(:rules things
  ((isa ?x ?s1) (imp-superset ?s1 ?s2) -> (isa ?x ?s2)))
```

The slot declarations for some of the important slots defined in the built-in knowledge-base are given below:

```
(:slot isa (things sets)
       :comment "(isa ?x ?s) = ?x is a member of the set ?s.")

(:slot member (sets things)
       :backlink isa
       :comment "(member ?s ?x) = A member of ?s is ?x.")

(:slot subset (sets sets)
       :comment "(subset ?s1 ?s2) = A subset of ?s1 is ?s2.")
```

```
(:slot superset (sets sets)
       :inverse subset
       :comment "(superset ?s1 ?s2) = A superset of ?s1 is ?s2.")

(:slot selfset (things sets)
       :cardinality 1
       :backlink member
       :comment "The selfset of x is the set {x}.")

(:slot less (objects objects)
       :comment "(less ?x ?y) = ?x less than ?y.")

(:slot greater (objects objects)
       :inverse less
       :comment "(greater ?x ?y) = ?x greater than ?y.")

(:slot equal (objects objects)
       :inverse equal
       :comment "(equal ?x ?y) = ?x is equal to ?y.")

(:slot least (objects sets)
       :comment "(least ?x ?s) = ?x is the least member of ?s.")

(:slot greatest (objects sets)
       :comment "(greatest ?x ?s) = ?x is the greatest member of ?s.")
```

4.3 Knowledge of Common Things

Distinct from the fundamental ontology of the knowledge base, Algernon has knowledge of a few common everyday objects and relations, mostly added to make our examples work.

Notice that there is no problem with declaring the same objects or relationship twice, if the information is compatible and name references are unambiguous.

```
(:taxonomy (things
            (objects
             (physical-attributes
              (colors black white red blue green
                      yellow brown orange purple)
              (genders male female :complete))
             (physical-objects
              (people)))))
```

The slots relating these objects are the following.

```
(:slot color (physical-objects colors)
       :cardinality 1
       :comment "(color x c) = The color of x is c.")

(:slot gender (physical-objects genders)
```

```
         :cardinality 1
         :comment "(gender x g) = The gender of x is g.")

(:slot spouse (people people)
         :cardinality 1
         :backlink spouse
         :comment "(spouse a b) = The spouse of a is b.")

(:slot wife (people people)
         :cardinality 1
         :backlink spouse
         :comment "(wife a b) = The wife of a is b.")

(:slot husband (people people)
         :cardinality 1
         :inverse wife
         :comment "(husband a b) = The husband of a is b.")

(:slot friend (people people)
         :comment "(friend a b) = A friend of a is b.")

;; Husband and wife also imply genders:
(:rules people
 ((wife ?x ?p1) -> (gender ?p1 female)))
(:rules people
 ((husband ?x ?p1) -> (gender ?p1 male)))

(:slot super-context (contexts contexts)
         :comment "(super-context c1 c2) = A super-context of c1 is c2.")

(:slot sub-context  (contexts contexts)
         :inverse super-context
         :comment "(sub-context c1 c2) = A sub-context of c1 is c2.")

(:slot current (contexts contexts)
         :cardinality 1
         :backlink super-context
         :comment "(current c1 c2) = The current sub-context of c1 is c2.")

(:slot speaker (contexts people)
         :cardinality 1
         :comment "(speaker c s) = The speaker in c is s.")

(:slot recent (contexts objects)
         :comment "(recent c r) = A recently mentioned thing in c is r.")
```

5 Helpful Hints

Reader contributions are solicited to expand this section.

- **Algernon is not Prolog.** Despite the superficial similarities, there are several important differences in the inference algorithms used. See section 2.3 for a brief description of the inference algorithms used in Algernon.
- Avoid using the non-monotonic special forms. They are essential in some cases, but we currently do not have a good theory of non-monotonic reasoning so it is difficult to predict their behavior without knowledge of the Algernon inference algorithms.
- It is a good idea to create a taxonomy and link your frames and slots into it. Setting up a large taxonomy in Algernon can be a slow process so it is often a good idea to set up the taxonomy and then take a "snapshot" of the knowledge-base (see section 3.2).
- In general, rules should be associated with frames (instead of slots). There are two important exceptions to this rule. First, some rules are associated with slots because they allow Algernon to conclude `isa` relations (and thus access rules). For example, it would do little good to associate the rule:

  ```
  ((father ?x ?y) -> (isa ?x people))
  ```

  with the set `people`. The second exception are rules associated with sets of slots (*e.g.*, rules for partial orders).
- Don't try to use `:taxonomy` to assert that a slot belongs to a set of slots. You can define sets of slots using `:taxonomy`, but all slots should be created and defined using `:slot`, and then linked to the taxonomy by asserting explicit `isa` relations.
- There is a simple theory of coreference (reasoning about whether two frames describe the same object) that is no longer loaded by default into the knowledge-base. Old examples that use the `coreference` link may fail.

5.1 Bugs and Problems

If you discover a bug in Algernon, please send a report, with enough information for us to reproduce the bug, to `algernon@cs.utexas.edu`.

References

[Brachman & Levesque, 1985] Brachman, Ronald J. and Levesque, Hector, J. (1985). *Readings in Knowledge Representation*, Morgan Kaufmann, Los Altos, Cal.

[Crawford & Kuipers, 1989] J. M. Crawford and B. J. Kuipers. 1989. Toward a theory of access-limited logic for knowledge representation. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*. Los Altos, CA: Morgan Kaufmann.

[Crawford, 90] J.M. Crawford. (1990). Access-Limited Logic — A language for knowledge representation. Doctoral dissertation, Department of Computer Sciences, The University of Texas at Austin. Published as Technical Report AI90-141, Artificial Intelligence Laboratory, The University of Texas at Austin.

[Crawford, Farquhar & Kuipers, 1990] J. M. Crawford, A. Farquhar, B. J. Kuipers. 1990. QPC: a compiler from physical models into qualitative differential equations. *Proceedings of the National Conference on Artificial Intelligence (AAAI-90)*, AAAI/MIT Press, 1990. Revised version in Boi Faltings and Peter Struss (Eds.), *Recent Advances in Qualitative Physics*, MIT Press, 1992.

[Crawford & Kuipers, 1991a] J. M. Crawford & B. J. Kuipers. 1991a. ALL: formalizing access-limited reasoning. In John Sowa (Ed.), *Principles of Semantic Networks*, pp. 299-330. San Mateo, CA: Morgan Kaufmann.

[Crawford & Kuipers, 1991b] J. M. Crawford & B. J. Kuipers. 1991b. Algernon – a tractable system for knowledge representation. *SIGART Bulletin* **2**(3): 35-44, June 1991.

[Crawford & Kuipers, 1991c] Crawford, J.M. and Kuipers, B.J. (1991c). Negation and Proof by Contradiction in Access-Limited Logic. *Proceedings of the National Conference on Artificial Intelligence (AAAI-91)*.

[Hayes, 79] Hayes, Patrick J. (1979). The logic of frames. In *Frame Conceptions and Text Understanding*, ed. D. Metzing, Walter de Gruyter and Co., Berlin, pp. 46-61. (Reprinted in [Brachman & Levesque, 1985], pp. 288-295.)

Index