Algernon for Expert Systems[*]

Benjamin Kuipers
Computer Science Department
University of Texas at Austin
Austin, Texas 78712 USA

18 January 1994

Abstract

This is a **DRAFT** document, showing how to express various kinds of useful inference in Algernon.

[*]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 NSF grants IRI-8904454, IRI-9017047, and IRI-9216584, and by NASA contracts NCC 2-760 and NAG 9-665. The original development of Algernon was supported in part by the Texas Advanced Research Program under grant no. 003658-175.

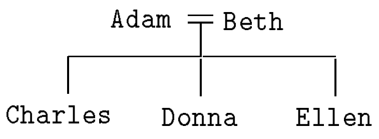
Contents

1 Introduction to Frame Based Knowledge Representation

1.1 Frames

Frames are descriptions of conceptual individuals. Frames can exist for "real" objects such as "The Watergate Hotel", sets of objects such as "Hotels", or more "abstract" objects such as "Cola-Wars" or "Watergate". Frames are essentially defined by their relationships with other frames. Relationships between frames are represented using *slots*. If a frame `f` is in a relationship `r` to a frame `g`, then we put the *value* `g` in the `r` *slot* of `f`.

For example, suppose we are describing the following genealogical tree:



The frame describing Adam might look something like:

```
Adam:
   sex:     Male
   spouse:  Beth
   child:   (Charles Donna Ellen)
```

where `sex`, `spouse`, and `child` are slots. Note that a single slot may hold several values (*e.g.*the children of Adam).

The genealogical tree would then be described by (at least) seven frames, describing the following individuals: Adam, Beth, Charles, Donna, Ellen, Male, and Female.

1.2 Frames and Their Names

Actually, this is not quite right. For initial presentation, we have allowed a certain amount of confusion between the thing *itself*, and its name. Really, the frame for `Adam` would look something like:

```
frame-27:
   name:    ("Adam")
   sex:     (frame-2)
   spouse:  (frame-28)
   child:   (frame-29 frame-30 frame-31)
```

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. In the example above, it is `frame-27`.
- 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. There is also an indexing mechanism that 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 frame is created with 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. However, this should be treated as a happy coincidence when it happens.

1.3 Frames and Predicate Calculus

A frame can be considered just a convenient way to represent a set of predicates applied to constant symbols (*e.g.ground instances* of predicates.). For example, the frame above could be written:

```
sex(Adam,Male)
spouse(Adam,Beth)
child(Adam,Charles)
child(Adam,Donna)
child(Adam,Ellen)
```

More generally, the ground predicate $r(f, g)$ is represented, in a frame based system, by placing the value `g` in the `r` slot of the frame `f` [Hayes, 1979]:

$$r(f,g) \quad \equiv \quad \boxed{\begin{array}{l} \texttt{f:} \\ \quad \texttt{r:} \\ \quad\quad \texttt{values: } \{ \ldots\texttt{g}\ldots \} \end{array}}$$

Throughout this document we use the `teletype font` for Algernon statements and the *italic font* for predicate calculus. We use standard predicate calculus notation except that we find it convenient to use $\rightarrow$ in addition to the usual $\leftarrow$.

1.4 Access Paths

One advantage of a frame based representation is that the (conceptual) objects related to a frame can be easily accessed by looking in a slot of the frame (there is no need, for example, to search the entire knowledge-base). We define an *access path*, in a network of frames, as a sequence of frames each directly accessible from (*i.e.*appearing in a slot of) its predecessor. A sequence of predicates defines an access path iff any variable appearing as the first argument to a predicate has appeared previously in the sequence. For example, "John's parent's sister" can be expressed in Algernon as the path:

```
((parent John ?x) (sister ?x ?y))
```

(variables are represented in Algernon by Lisp atoms whose print names begin with '`?`'). The Algernon path `((parent John ?x) (sister ?x ?y))` is equivalent to the syntactically similar predicate calculus statement:

$$parent(John, ?x) \wedge sister(?x, ?y).$$

In predicate calculus this statement is equivalent to

$$sister(?x, ?y) \wedge parent(John, ?x).$$

However, the corresponding sequence of predicates:

```
((sister ?x ?y) (parent John ?x))
```

is *not* an access path because a query of `(sister ?x ?y)` requires a search of every frame in the entire knowledge-base.

1.5 Rules Are Implications

We represent logical implications as rules. For example, the logical assertion that

$$\forall x, y.[spouse(x, y) \rightarrow spouse(y, x)]$$

can be represented by the *forward-chaining rule*:

```
((spouse ?x ?y) -> (spouse ?y ?x))
```

Intuitively, such a rule says that whenever we learn a relation `(spouse f g)` we should immediately conclude `(spouse g f)`. Such *forward-chaining* ("if-added") rules are generally used to maintain invariants in the knowledge-base.

Algernon also allows *backward-chaining* ("if-needed") rules. For example,

```
((aunt John ?y) <- (parent John ?x) (sister ?x ?y))          (1)
```

Intuitively, this rule says that if you need to find an aunt of `John` then you should look for a parent of `John`, and then a sister of that parent. Notice that the antecedent of this rule is an access path. All rules in Algernon must define access paths. Thus the implication:

```
((aunt John ?y) <- (sister ?x ?y) (parent John ?x))
```

is not allowed since the only way to "fire" this rule would be to search globally for all frames in the knowledge-base that have a `sister` relation with other frames.

The backward-chaining rule (1) represents the content of the logical implication

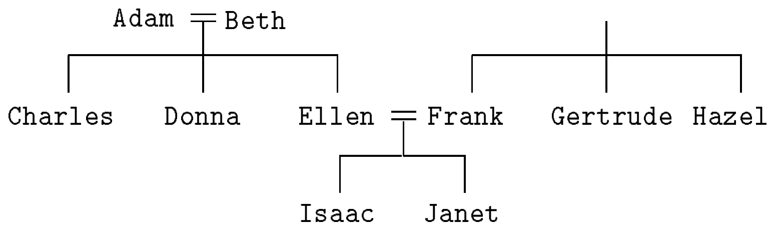
$$\forall x, y.[parent(John, x) \wedge sister(x, y) \rightarrow aunt(John, y)]$$

but with the added knowledge (or restriction) that the implication should only be used to satisfy a query of $aunt(John, y)$.

1.6 Inference in Algernon

There are two basic operations on a knowledge-base — *queries* and *assertions.* Queries retrieve knowledge from the knowledge-base, applying if-needed rules, while assertions add facts to the knowledge-base and apply if-added rules. Of course, a successful if-needed rule will assert its consequent into the knowledge-base, and the antecedent of an if-added rule may have additional formulas to be queried, so the two operations are not disjoint. The rules in an Algernon knowledge-base are organized into small packets by being associated either with frames representing sets, or with slots.

To understand how this works, consider a knowledge-base of frames and rules representing the following genealogy:



There is also a frame `People`, representing the set of all people. We assert the relation `isa` (representing the membership relation between an object and a set containing it) between each frame for a person in the genealogy and the frame `People` (*i.e.*`(isa Adam People)`, `(isa Beth People)`, ...). We may associate the rule:

$$\texttt{((aunt ?z ?y) <- (parent ?z ?x) (sister ?x ?y))} \qquad (2)$$

with the frame *People*. Formally this means:

$$(\forall z, x, y.(isa(z, People) \rightarrow [aunt(z, y) \leftarrow parent(z, x) \wedge sister(x, y)]))$$

Operationally, this means that any query to the `aunt` slot of a frame that has an `isa` relation to the frame `People` will result in the application of this rule. A rule is applied by first querying its antecedent and then, if the antecedent succeeds, asserting its consequent. Querying the antecedent of a rule finds all consistent sets of bindings to the variables in the antecedent. One instance of the consequent is asserted for each binding set, with variables replaced by values in the binding set. This means that retrieval guided by an access path *branches* on the values in a slot that provide possible bindings for a variable.

For example, if we query `(aunt Janet ?x)`, then rule (2) will apply, with `?z` bound to `Janet`. The path will immediately branch on the values in the `parent` slot of `Janet`: bindings of `?x` to `Ellen` or `Frank`. If `?x` is bound to `Ellen`, then Algernon will query `(sister Ellen ?y)`, which binds `?y` to `Donna`, so it will conclude and assert `(aunt Janet Donna)`. However, along the branch where `x` is bound to `Frank`, retrieval of `(sister Frank ?y)` will branch on two possibilities, so Algernon will also conclude and assert that `(aunt Janet Gertrude)` and `(aunt Janet Hazel)`.

Rules may also (less commonly) be associated with slots or sets of slots. If an if-added rule is associated with a slot `r`, or with a set of which `r` is a member, then it is applied whenever a value is added to the `r` slot of *any* frame (if-needed rules associated with slots and sets of slots are similar). For example, one might want to put the slot `less` in the set of partial orders, and thus inherit rules, associated with the set of partial orders, for transitivity, reflexivity, and antisymmetry.

By associating rules with sets of individuals, or with slots representing specific relations, we are imposing a semantically-motivated organization on the knowledge-base, which clarifies the structure of its knowledge. We are also limiting access to those rules, thereby reducing the number of times their antecedents must be tested, and hence gaining efficiency.

2 Example: Frames, Access Paths, and Retrieval by Description

In this and the following sections, many of the features of Algernon will be illustrated by means of simple examples.

2.1 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` assert or query a given access path in the context of 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 `: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.

2.2 The Structure of an Algernon Theory

A theory in Algernon has a conventional format, which is not strictly required, but greatly simplifies creation and understanding of the program. It normally consists of the following five sections.

1. **Taxonomy**: define a containment hierarchy of sets of objects that appear in the theory and certain individual elements of those sets.
2. **Slots**: define the relations that may hold among those objects.
3. **Rules**: define the forward- and backward-chaining inferences that can take place using those relations.
4. **Facts**: assert the specific facts of the situation to be reasoned about. Forward-chaining rules invoked by the assertion of these facts, and backward-chaining rules invoked by the antecedents of those rules, may cause a significant amount of inference to take place.
5. **Queries**: query the knowledge base for desired information. Backward-chaining rules invoked by the query, and forward-chaining rules invoked by assertion of deduced information, may also add additional facts to the knowledge-base.

The taxonomy and slots together constitute the *ontology* of the theory: what objects and relationships are describable within it.

2.3 Taxonomy

The Algernon knowledge base starts with an ontology of fundamental sets and a few individual objects.

```
(tell '((:taxonomy (things
                     (rules)
                     (objects
                      (sets things objects sets slots partitions
                            (partitions main-partition set-partition
                                        slot-info-partition partition-partition))
                      (booleans true false :complete)
                      (contexts *context*))
                     (slots
                      (order-relations
                       (tc-order-relations
                        (equivalence-relations))))))))
```

Additional sets and individuals can be defined using the `:taxonomy` form. Each `:taxonomy` form asserts a tree-structured set of containment and membership relations between sets and elements. A set is described by a list whose first element is its name, whose sublists are descriptions of its subsets, and whose atomic elements are names of its elements. Subsets are not necessarily disjoint. Lattice-structured containment hierarchies can be asserted using multiple `:taxonomy` forms. The set at the root of the taxonomy must already exist, and all names must be uniquely designating. By convention, the names of sets are plural nouns.

The above example describes part of the taxonomy created in the background knowledge base. The set `things` contains at least the subsets `rules`, `objects`, and `slots`. The set `booleans` contains exactly the two elements `true` and `false`. Several of the sets in the hierarchy are also explicitly declared as individual elements of the set `sets`, but those individuals are not completely enumerated.

The following creates, names, and asserts some additional subsets to the set `objects`, including the new set `time-units` and its element `years`.

```
(tell '((:taxonomy (objects
                     (physical-objects
                      (people))
                     (physical-attributes
                      (genders male female))
                     (time-units years)))))
```

2.4 Slots Represent Relations

A slot represents a relation among individuals, and is declared using `:slot`. The domain of an n-ary relation is the cross-product of n sets, which are specified in the second argument to the `:slot` form. The first argument is the name of the relation. By convention, the name P of a relation $P(a, b)$ is chosen to fit the template, "The P of a is b."

The `:cardinality` keyword specifies how many distinct values can consistently be in a slot. For example, `spouse` can only have one value, but `grandparent` would have `:cardinality 4`. (Modern family structure is beyond the scope of this document.) The `friend` slot has no bound.

Binary relations of `:cardinality 1` are functions, defining mapping from individuals in the first domain to individuals in the second domain. Functions support certain inferences that don't apply to more general relations.

The `:backlink` keyword declares an inverse relation, and a forward chaining rule from the current relation to its inverse. The `:inverse` keyword is similar, but creates backlink rules in both directions. The `:comment` keyword is purely for documentation.

```
(tell '((: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.")))
```

Unlike the above binary relations, `age` is a three place relation between a physical object, its age, and the time-unit the age is measured in. Since the age will be represented by a Lisp object (*i.e.*, a number), the keyword `:number` indicates that the second argument is represented by an object belonging to the Lisp datatype `number`, rather than by an Algernon frame.

```
(tell '((:slot age (physical-objects :number time-units)
                   :cardinality 1)))
```

2.5 Rules Specify Inferences

The rules in the Algernon knowledge-base determine which inferences will take place. Here, we give two rules that show how the husband and wife relations determine gender of the participants. Along with the rules implied by the `:backlink` and `:inverse` keywords to the `:slot` declarations, these rules fill out the implications of a single asserted relationship between two people.

```
(tell '((:rules people
         ((wife ?x ?p1) -> (gender ?p1 female))
         ((husband ?x ?p1) -> (gender ?p1 male)))))
```

Rules are associated with sets, so the above rules are only considered in case the variable `?x` is bound to an individual known to belong to the set `people`.

Relatively little is known about controlling mixed forward- and backward-chaining inference, so many systems are restricted to only one direction. This example uses only forward-chaining inference, but later examples will demonstrated mixed inference. One convention that appears to be useful is to use forward-chaining rules to maintain invariants that should be true of the information explicitly appearing in the KB, and backward-chaining rules to do search-based problem-solving. Other conventions are certainly possible.

2.6 Facts Tell the Story

This little story illustrates the use of Algernon to represent the meaning of a simple narrative. It creates frames to represent the individuals involved, asserts and deduces relationships among them.

An important feature is the use of definite descriptions to retrieve the individual about which an assertion should be made. In Algernon, a definite description is an access path that is believed to be adequate, in context, to specify a unique individual.

It is worth noting how easliy the Algernon access paths could be derived from the English sentences they correspond to.

- The function `tell` takes an Algernon access path to assert, and optionally a comment string. There is a corresponding function `ask`.

  Since definite descriptions require context, we create a frame with true-name `ja-story` to represent the current story, and put it into the `current` slot of the global context frame, which is named `*context*`.[1] When the frame bound to `?ja-story` is first created, it belongs to no sets, but the domains associated with the relation `current` allow Algernon to infer that it belongs to the set `contexts`.

```
(tell '((:create ?ja-story ja-story)
        (:clear-slot *context* current)
        (current *context* ?ja-story)))
```

  The effect of `tell` is a change to the structure of the Algernon knowledge-base. With appropriate trace switches set, Algernon will also print a trace message, a description of the set of bindings created by the assertion or query, and a list of newly created frames.

- A first-person pronoun ("I", "me", "my", etc.) is a definite description referring to the current speaker. This is an attribute of the current context. A definite description, specified by the Algernon special form `:the`, will do "find-or-create retrieval", creating a new frame and asserting the description about it if no existing match is found.

  Once the frame representing the speaker has been found, facts such as name and age can be asserted to it.

```
(tell '((:the ?me (speaker (current *context*) ?me))
        (name ?me "John Alden"))
      :comment "My name is John Alden.")

(tell '((:the ?me (speaker (current *context*) ?me))
        (age ?me 25 years))
      :comment "I am 25 years old.")
```

[1] `:create` *is obsolete. Use* `:a` *instead.*

- The definite description "my wife" here causes the creation of a new frame to represent John's wife. The rules we have already seen allow Algernon to infer the appropriate set of `husband`, `spouse`, and `gender` relations.

  Any mentioned frame, other than the speaker, is added to the `recent` slot of the current context, for later retrieval using third person pronouns. This serves as a place-holder for a more serious focus mechanism.

```
  (tell '((:the ?me (speaker (current *context*) ?me))
          (:the ?w (wife ?me ?w))
          (age ?w 23 years)
          (recent (current *context*) ?w))
        :comment "My wife is 23 years old.")
```

  The following example of trace output was produced by this assertion. A single set of bindings resulted from asserting the given access path, and those bindings are shown. The variables `?$x13` and `?$x14` were automatically generated to represent the results of retrieving `(current *context*)`. The variable `?me` is bound to the frame `frame1`, which is the true name of the frame describing John Alden. The `name` slot of `frame1` holds the string `"John Alden"`, so that is shown. The variable `?w` is bound to the frame `frame1-wife`, which was created without a public name, though we will learn it in the next sentence.

```
ASSERTING:   My wife is 23 years old.

 Input preds:          (:the ?me (speaker (current *context*) ?me))
                       (:the ?w (wife ?me ?w)) (age ?w 23 years)
                       (recent (current *context*) ?w)

 Result:
   Bindings:            ?$x14  --- ja-story
                        ?w     --- frame1-wife
                        ?me    --- frame1 "john alden"
                        ?$x13  --- ja-story

Created frame:         frame1-wife

  => T
```

- The definite description "she" or "her" retrieves a recently mentioned female.

```
  (tell '((:the ?her
           (recent (current *context*) ?her)
           (gender ?her female))
          (name ?her "Priscilla"))
        :comment "Her name is Priscilla.")
```

- In the following sentence, we have used `:forc` (find or create) to represent the indefinite description, "a friend named Miles Standish". It would also be possible to use `:a` (create new) to create a new frame representing "a friend", and assert the name and gender of Miles Standish to that frame. (The gender is necessary because Algernon doesn't know that "Miles" is a man's name.)

```
(tell '((:the ?she
          (recent (current *context*) ?she)
          (gender ?she female))
         (:forc ?ms
          (friend ?she ?ms)
          (name ?ms "Miles Standish")
          (gender ?ms male)
          (recent (current *context*) ?ms))
         :comment "She has a friend named Miles Standish.")
```

- The definite description "he" can retrieve the description of Miles from context because he is the only recently mentioned male. This primitive context mechanism can't even give priority to recency, so once a second male is mentioned, it will fail.

```
(tell '((:the ?he
          (recent (current *context*) ?he)
          (gender ?he male))
         (age ?he 40 years))
       :comment "He is 40 years old.")
```

- A name is also a definite description. In simple cases like this one with uniquely-referring single-word names, the Algernon syntactic preprocessor can recognize that `Priscilla` refers to a unique frame and substitute it into the access path. The `:forc` creates a frame describing Cotton Mather.

```
(tell '((:forc ?cm
          (friend Priscilla ?cm)
          (name ?cm "Cotton Mather")
          (gender ?cm male))
         (recent (current *context*) ?cm))
       :comment "Priscilla also has a friend named Cotton Mather.")
```

2.7 Questions: Checking for Understanding

We test whether a human has understood a story by asking questions, first to check on retention of explicitly stated facts, then for the ability to draw reasonable conclusions.

- We return attention to the John Alden story from whatever else we were doing, by explicitly making its context frame, `ja-story`, the current context.

```
(tell '((:clear-slot *context* current)
         (current *context* ja-story))
       :comment "In the John Alden story ...")
```

- This question follows the access path from context, to speaker, to wife, to age. Her name is only retrieved for nicely presenting the output.

```
  (ask '((:the ?me (speaker (current *context*) ?me))
         (:the ?w (wife ?me ?w))
         (age ?w ?a ?units)
         (name ?w ?n))
       :collect '(?n is ?a ?units old)
       :comment "How old is my wife?")
```

  The `:collect` keyword provides restructured output.

```
QUERYING:  How old is my wife?

 Input preds:          (:the ?me (speaker (current *context*) ?me))
                       (:the ?w (wife ?me ?w)) (age ?w ?a ?units) (name ?w ?n)

 Result:
   Bindings:            ?n     --- "priscilla"
                        ?units   --- years
                        ?a     --- 23
                        ?w     --- frame1-wife "priscilla"
                        ?me    --- frame1 "john alden"
                        ?$x20  --- ja-story

  => (("Priscilla" IS 23 YEARS OLD))
```

- This question checks that obvious inferences were made from the given information, by retrieving Priscilla's husband and then his name.

```
  (ask '((husband Priscilla ?h)
         (:eval (pp-frame '?h))
         (name ?h ?n))
       :comment "Who is Priscilla's husband?"
       :collect '?n)
```

  The special `:eval` form applies the lisp function `pp-frame` to the frame bound to the variable `?h`, to display it on the terminal. Then the name is returned.

```
QUERYING:  Who is Priscilla's husband?

 Input preds:          (husband priscilla ?h) (:eval (pp-frame (quote ?h)))
                       (name ?h ?n)

Frame1:
  Isa:        things objects physical-objects people
 Name:       "john alden"
 Age:        (25 years)
 Wife:       frame1-wife
 Gender:     male
 Spouse:     frame1-wife

Result:
  Bindings:            ?n     --- "john alden"
                       ?h     --- frame1 "john alden"

 => ("John Alden")
```

- This question has more than one answer, so both are returned.

```
(ask '((name (friend Priscilla) ?name))
     :comment "What is Priscilla's friend's name ?"
     :collect '?name)
```

Here we collect two names into a list.

```
QUERYING:  What is Priscilla's friend's name ?

 Input pred:           (name (friend priscilla) ?name)

 Result (1 of 2):
   Bindings:            ?name  --- "cotton mather"
                        ?$x21  --- |cotton mather|

 Result (2 of 2):
   Bindings:            ?name  --- "miles standish"
                        ?$x21  --- |miles standish|

  => ("Miles Standish" "Cotton Mather")
```

3 Example: Forward- and Backward-Chaining Inference

This example creates a more extensive theory of family relationships than were necessary for the John Alden story.

3.1 Taxonomy

Most of this taxonomy reiterates what is already declared in the background knowledge base, but it does no harm to repeat it, and it clarifies the assumptions we are depending on.

```
(tell '((:taxonomy (things
                    (objects
                     (physical-objects
                      (people))
                     (physical-attributes
                      (genders male female)))))))
```

3.2 Slots

We define a set of family relationships. Some are gender-specific specializations of others. Some can be deduced from combinations of the others.

```
(tell '((:slot child    (people people))
        (:slot son      (people people))
        (:slot daughter (people people))

        (:slot parent (people people)  :cardinality 2   :inverse child)
        (:slot father (people people)  :cardinality 1)
        (:slot mother (people people)  :cardinality 1)

        (:slot sibling (people people)  :inverse sibling)
        (:slot brother (people people))
        (:slot sister  (people people))

        (:slot grandchild   (people people))
        (:slot grandson     (people people))
        (:slot grandaughter (people people))

        (:slot grandparent (people people)  :cardinality 4  :inverse grandchild)
        (:slot grandfather (people people)  :cardinality 2)
        (:slot grandmother (people people)  :cardinality 2)

        (:slot uncle (people people))
        (:slot aunt  (people people))))
```

3.3 Rules

Gender-specific relations are tightly connected by matched forward- and backward-chaining rules to their gender-neutral counterparts.

```
(tell '((:rules people
         ((father ?x ?f) -> (parent ?x ?f) (gender ?f male))
         ((father ?x ?f) <- (parent ?x ?f) (gender ?f male))

         ((mother ?x ?f) -> (parent ?x ?f) (gender ?f female))
         ((mother ?x ?f) <- (parent ?x ?f) (gender ?f female))

         ((son ?x ?s) -> (child ?x ?s) (gender ?s male))
         ((son ?x ?s) <- (child ?x ?s) (gender ?s male))

         ((daughter ?x ?d) -> (child ?x ?d) (gender ?d female))
         ((daughter ?x ?d) <- (child ?x ?d) (gender ?d female))

         ((brother ?x ?b) -> (sibling ?x ?b) (gender ?b male))
         ((brother ?x ?b) <- (sibling ?x ?b) (gender ?b male))

         ((sister ?x ?b) -> (sibling ?x ?b) (gender ?b female))
         ((sister ?x ?b) <- (sibling ?x ?b) (gender ?b female))

         ((grandfather ?x ?gf) -> (grandparent ?x ?gf) (gender ?gf male))
         ((grandfather ?x ?gf) <- (grandparent ?x ?gf) (gender ?gf male))

         ((grandmother ?x ?gf) -> (grandparent ?x ?gf) (gender ?gf female))
         ((grandmother ?x ?gf) <- (grandparent ?x ?gf) (gender ?gf female))

         ((grandson ?x ?gs) -> (grandchild ?x ?gs) (gender ?gs male))
         ((grandson ?x ?gs) <- (grandchild ?x ?gs) (gender ?gs male))

         ((grandaughter ?x ?gs) -> (grandchild ?x ?gs) (gender ?gs female))
         ((grandaughter ?x ?gs) <- (grandchild ?x ?gs) (gender ?gs female)))))
```

More complex rules show how to infer some relations from others. Note that it isn't correct to say that one relation is *defined* in terms of more primitive relations. Rather, there is a network of inferences that link relations together. For example, the `uncle` relation can be inferred from `parent` and `brother`, or from `aunt` and `husband`, but it can also be asserted without commitment to which type of relation forms the intermediate link.

The `:neq` special form succeeds if two frames are not identical, and is necessary here to define the relation `sibling`. These rules demonstrate the use of embedded terms such as `(father (parent ?a) ?c)` as an abbreviation for `(parent ?a ?b) (father ?b ?c)`.

```
(tell '((:rules people

         ((grandfather ?a ?c)  <- (father (parent ?a) ?c))
         ((grandmother ?a ?c)  <- (mother (parent ?a) ?c))
         ((grandson ?a ?c)     <- (son (child ?a) ?c))
         ((grandaughter ?a ?c) <- (daughter (child ?a) ?c))

         ((sibling ?x ?y) <- (child (parent ?x) ?y) (:neq ?x ?y))

         ;; Aunt and Uncle are a bit different (there is no unisex term):
```

```
((uncle ?x ?u) -> (gender ?u male))
((aunt ?x ?a)  -> (gender ?a female))

((uncle ?x ?u) <- (brother (parent ?x) ?u))
((uncle ?x ?u) <- (husband (aunt ?x) ?u))
((aunt ?x ?a)  <- (sister (parent ?x) ?a))
((aunt ?x ?a)  <- (wife (uncle ?x) ?a))

((husband ?w ?h)
 <-
 (gender ?w female) (spouse ?w ?h) (gender ?h male))
((wife ?h ?w)
 <-
 (gender ?h male) (spouse ?h ?w) (gender ?w female)))))
```

3.4 Facts

Although our simple theory is obviously inadequate to do justice to the complexities of real life, we create a set of frames to represent some of the members of the British royal family and their relationships.

Consider two alternate ways to make these assertions:

1. by creating frames, binding them to variables, and asserting the relationships via the variable bindings;

2. by creating frames with mnemonically-chosen internal names, then relying on the syntactic processor to replace the internal names with references to the frames themselves.

```
(tell '((:create ?ch Charles)             (tell '((:create ?ch Charles)
        (:create ?di Diana)                       (:create ?di Diana)
        (:create ?ha Harry)                       (:create ?ha Harry)
        (:create ?wi William)                     (:create ?wi William)
        (:create ?ph Philip)                      (:create ?ph Philip)
        (:create ?el Elizabeth)                   (:create ?el Elizabeth)
        (:create ?an Andrew)                      (:create ?an Andrew)
        (:create ?sa Sarah)                       (:create ?sa Sarah)))

        (wife ?ch ?di)                    (tell '((wife Charles Diana)
        (son ?ch ?ha)                             (son Charles Harry)
        (son ?ch ?wi)                             (son Charles William)
        (son ?di ?ha)                             (son Diana Harry)
        (son ?di ?wi)                             (son Diana William)
        (father ?ch ?ph)                          (father Charles Philip)
        (mother ?ch ?el)                          (mother Charles Elizabeth)
        (wife ?ph ?el)                            (wife Philip Elizabeth)
        (son ?ph ?an)                             (son Philip Andrew)
        (son ?el ?an)                             (son Elizabeth Andrew)
        (wife ?an ?sa)))                          (wife Sarah Andrew))))
```

3.5 Queries

Here we ask some questions to get new inferences done. Since both questions are in one path, and since there is only one answer to each question, we get one set of bindings back.

```
  (ask '((grandfather William ?gf)
         (grandmother William ?gm))
       :comment "Who are William's grandfathers and grandmothers?")

QUERYING:  Who are William's grandfathers and grandmothers?

 Result:
   Bindings:            ?gm    --- elizabeth "[elizabeth]"
                        ?gf    --- philip "[philip]"

  => T
```

Here's a similar inference.

```
  (ask '((uncle William ?u)
         (aunt William ?v))
       :comment "Who are William's aunts and uncles?")

QUERYING:  Who are William's aunts and uncles?

 Result:
   Bindings:            ?v     --- sarah "[sarah]"
                        ?u     --- andrew "[andrew]"

  => T
```

Here the question has two answers, so we get two bindings back.

```
  (ask '((parent William ?u))
       :comment "Who are William's parents?")

QUERYING:  Who are William's parents?

 Result (1 of 2):
   Binding:             ?u     --- diana "[diana]"

 Result (2 of 2):
   Binding:             ?u     --- charles "[charles]"

  => T
```

In the following, we use a special form, `:all-paths`, which succeeds if all sets of bindings that result from the first path, also satisfy the second access path.

```
  (ask '((:all-paths ((child Charles ?b)) ((gender ?b male))))
       :comment "Are all Charles' children male?")
```

```
QUERYING:  Are all Charles' children male?

  => T

  (ask '((:all-paths ((child Charles ?b)) ((gender ?b female))))
       :comment "Are all Charles' children female?")

QUERYING:  Are all Charles' children female?

 *Query failed.*

  => NIL
```

4 A Simple Expert System: Does John Have The Flu?

This example presents a very simple knowledge-based system using backward-chaining rules to diagnose diseases.

4.1 Taxonomy and Slots

People have symptoms and diseases, and diseases are associated with symptoms. This requires us to define the appropriate sets, with a few illustrative individual diseases and symptoms.

```
(tell '((:taxonomy (Objects
                    (People)
                    (Diseases Flu Plague)
                    (Symptoms Fever High-Fever Nausea Nodules)))))

(tell '((:slot has-disease (people diseases))
        (:slot has-symptom (people symptoms))
        (:slot symptom     (diseases symptoms))
        (:slot temperature (physical-objects :number) :cardinality 1)))
```

The meanings of the relations are:

has-disease(p, d)	$\equiv$	person p has disease d
has-symptom(p, s)	$\equiv$	person p has symptom s
symptom(d, s)	$\equiv$	disease d causes symptom s
temperature(x, t)	$\equiv$	physical object x has temperature t

4.2 Rules – Simple

The simplest approach to diagnosis is to write a set of special-purpose rules for each disease, specifying when we can conclude that a patient has the disease.

```
(tell '((:rules People
         ((has-disease ?x Flu)    <- (has-symptom ?x fever) (has-symptom ?x nausea))
         ((has-disease ?x Plague) <- (has-symptom ?x high-fever) (has-symptom ?x nodules))
         )))
```

These rules will be invoked on a query such as `((has-disease ?p ?d))`, and will chain backward to query the symptoms of `?p`. Therefore, we need rules to conclude symptoms.

```
(tell '((:rules People
         ((has-symptom ?p fever)      <- (temperature ?p ?t) (:test (> ?t 99)))
         ((has-symptom ?p high-fever) <- (temperature ?p ?t) (:test (> ?t 102)))
         ((has-symptom ?p nausea)     <- (:ask (has-symptom ?p nausea)))
         ((has-symptom ?p nodules)    <- (:ask (has-symptom ?p nodules)))
         ((temperature ?p ?t)         <- (:ask (temperature ?p ?t))))))
```

In order to conclude that the patient has the symptom `fever`, we determine the patient's temperature t, and test whether $t > 99$. The `:test` special form allows us to escape to Lisp to evaluate a Lisp expression. We can only determine the patient's temperature by asking the user. Similarly, we can only conclude that the patient has nausea by asking the user.

The `:ask` special form is a trivial user interface, used only for tutorial purposes. We will describe the interfaces with Lisp and with the user in the next section.

4.3 Queries Based on Simple Rules

We can test these rules with a query that finds or creates a frame for a person given a one-symbol name, then asks what diseases that person has.

```
(defun dx-patient (name)
  (ask `((:the ?p (name ?p ,(string name)) (isa ?p People))
         (has-disease ?p ?d))
       :comment "What does the patient have?"
       :collect '(?p has ?d)))
```

Frank, George and Harry all present with fever, but turn out to have different diagnoses. The rules for the two diseases backchain to the rules for fever or high-fever, which in turn backchain to the rule for temperature, which queries the user. In Frank's case, his fever is low enough that the rule for plague fails without asking about nodules.

```
> (dx-patient 'frank)
 Give me a value for ?t in (temperature frank ?t): 100
 Is it true that (has-symptom frank nausea)?  (Yes or No) yes
((FRANK HAS FLU))
```

George is asked about nodules, and indeed has the plague, but at least avoids the flu.

```
> (dx-patient 'george)
 Give me a value for ?t in (temperature george ?t): 103
 Is it true that (has-symptom george nodules)?  (Yes or No) yes
 Is it true that (has-symptom george nausea)?  (Yes or No) no
((GEORGE HAS PLAGUE))
```

Poor Harry has both the plague and the flu.

```
> (dx-patient 'harry)
 Give me a value for ?t in (temperature harry ?t): 104
 Is it true that (has-symptom harry nodules)?  (Yes or No) yes
 Is it true that (has-symptom harry nausea)?  (Yes or No) yes
((HARRY HAS PLAGUE) (HARRY HAS FLU))
```

It can be instructive to follow the "Logic Trace" of the deduction by which Harry was diagnosed.

```
algy> tl
  Trace level = :LOGIC.

algy> (dx-patient 'harry)

  ** Beginning logic Trace **
 Querying: (:the (?j) (name ?j "harry") (isa ?j people)).
   Querying: (name ?j "harry").
   Query failed.
   Creating new frame: harry.
   Asserting: (name harry "harry").
     Inserting new value (isa harry things).
```

```
    Inserting new value (name harry "harry").
  Assert succeeded: (name harry "harry").
  Asserting: (isa harry people).
    Inserting new value (isa harry people).
      Applying: ((isa harry people) -> (isa harry objects)).
        Querying: (isa harry people).
        Query succeeded: (isa harry people).
        Asserting: (isa harry objects).
          Inserting new value (isa harry objects).
            Applying: ((isa harry objects) -> (isa harry things)).
              Querying: (isa harry objects).
              Query succeeded: (isa harry objects).
              Asserting: (isa harry things).
                Value already known (isa harry things).
              Assert succeeded: (isa harry things).
            Rule applied.
        Assert succeeded: (isa harry objects).
      Rule applied.
      Applying: ((isa harry people) -> (isa harry physical-objects)).
        Querying: (isa harry people).
        Query succeeded: (isa harry people).
        Asserting: (isa harry physical-objects).
          Inserting new value (isa harry physical-objects).
            Applying: ((isa harry physical-objects) -> (isa harry objects)).
              Querying: (isa harry physical-objects).
              Query succeeded: (isa harry physical-objects).
              Asserting: (isa harry objects).
                Value already known (isa harry objects).
              Assert succeeded: (isa harry objects).
            Rule applied.
        Assert succeeded: (isa harry physical-objects).
      Rule applied.
  Assert succeeded: (isa harry people).
Query succeeded: (:the (harry) (name harry "harry") (isa harry people)).
(:the (?j) (name ?j "harry") (isa ?j people))
Querying: (has-disease harry ?d).
  Creating new frame: ri-0.
  Creating new frame: ri-1.
  Applying: ((has-disease harry plague) <- (has-symptom harry high-fever)
                                           (has-symptom harry nodules)).
    Querying: (has-symptom harry high-fever).
      Creating new frame: ri-2.
      Applying: ((has-symptom harry high-fever) <- (temperature harry ?t)
                                                   (:test (> ?t 102))).
        Querying: (temperature harry ?t).
          Creating new frame: ri-3.
          Applying: ((temperature harry ?t) <- (:ask (temperature harry ?t))).
            Querying: (:ask (temperature harry ?t)).
Give me a value for ?t in (temperature harry ?t): 104
              Asserting: (temperature harry 104).
                Inserting new value (temperature harry 104).
              Assert succeeded: (temperature harry 104).
            Query succeeded: (:ask (temperature harry 104)).
            Asserting: (temperature harry 104).
              Value already known (temperature harry 104).
            Assert succeeded: (temperature harry 104).
          Rule applied.
        Query succeeded: (temperature harry 104).
        Querying: (:test (> 104 102)).
        Query succeeded: (:test (> 104 102)).
        Asserting: (has-symptom harry high-fever).
          Inserting new value (has-symptom harry high-fever).
        Assert succeeded: (has-symptom harry high-fever).
      Rule applied.
    Query succeeded: (has-symptom harry high-fever).
    Querying: (has-symptom harry nodules).
      Creating new frame: ri-4.
      Applying: ((has-symptom harry nodules) <- (:ask (has-symptom harry
                                                                   nodules))).
        Querying: (:ask (has-symptom harry nodules)).
Is it true that (has-symptom harry nodules)?  (Yes or No) yes
          Asserting: (has-symptom harry nodules).
            Inserting new value (has-symptom harry nodules).
          Assert succeeded: (has-symptom harry nodules).
        Query succeeded: (:ask (has-symptom harry nodules)).
        Asserting: (has-symptom harry nodules).
          Value already known (has-symptom harry nodules).
        Assert succeeded: (has-symptom harry nodules).
      Rule applied.
    Query succeeded: (has-symptom harry nodules).
    Asserting: (has-disease harry plague).
      Inserting new value (has-disease harry plague).
    Assert succeeded: (has-disease harry plague).
  Rule applied.
  Applying: ((has-disease harry flu) <- (has-symptom harry fever)
                                        (has-symptom harry nausea)).
    Querying: (has-symptom harry fever).
      Creating new frame: ri-5.
      Applying: ((has-symptom harry fever) <- (temperature harry ?t)
                                              (:test (> ?t 99))).
        Querying: (temperature harry ?t).
        Query succeeded: (temperature harry 104).
        Querying: (:test (> 104 99)).
        Query succeeded: (:test (> 104 99)).
        Asserting: (has-symptom harry fever).
          Inserting new value (has-symptom harry fever).
        Assert succeeded: (has-symptom harry fever).
      Rule applied.
    Query succeeded: (has-symptom harry fever).
    Querying: (has-symptom harry nausea).
      Creating new frame: ri-6.
      Applying: ((has-symptom harry nausea) <- (:ask (has-symptom harry nausea))).
```

```
        Querying: (:ask (has-symptom harry nausea)).
 Is it true that (has-symptom harry nausea)?  (Yes or No) yes
          Asserting: (has-symptom harry nausea).
            Inserting new value (has-symptom harry nausea).
          Assert succeeded: (has-symptom harry nausea).
        Query succeeded: (:ask (has-symptom harry nausea)).
        Asserting: (has-symptom harry nausea).
          Value already known (has-symptom harry nausea).
        Assert succeeded: (has-symptom harry nausea).
      Rule applied.
    Query succeeded: (has-symptom harry nausea).
    Asserting: (has-disease harry flu).
      Inserting new value (has-disease harry flu).
    Assert succeeded: (has-disease harry flu).
  Rule applied.
 Query succeeded: (has-disease harry flu) (has-disease harry plague).
 (has-disease ?j ?d)
  ** End logic Trace **
  ((HARRY HAS PLAGUE) (HARRY HAS FLU))
```

4.4 Rules – Slightly Less Simple

It might be desirable to organize the knowledge in the Algernon KB in a more modular way.

We will exploit the backlink from the set Diseases to the individual known diseases. The :taxonomy form asserts explicit links from sets to the specified elements, Flu and Plague. However, in case we learn about more diseases, we provide a rule to assert the backlink from an isa relation.

```
(tell '((:rules Diseases
         ((isa ?d Diseases) -> (member Diseases ?d)))))
```

In general, the "upward" isa relation, pointing from an element to a set it belongs to, is not backlinked to the "downward" member relation. Some sets (e.g. Things or People) could have large numbers of elements. The access-limited philosophy of Algernon discourages inferences that scan over large numbers of elements.

We can then formulate a single general-purpose rule that says that a person has a disease if he or she has every symptom of the disease. (Obviously, neither of these approaches is very realistic.)

```
(tell '((:rules People
         ((has-disease ?x ?d)
          <-
          (member Diseases ?d)
          (:all-paths ((symptom ?d ?s)) ((has-symptom ?x ?s)))))))
```

Within this approach, we can specify the symptoms caused by a disease, not in a rule associated with the set People, but as part of the declarative description of the disease itself.

```
(tell '((symptom Flu Fever)          (symptom Plague High-Fever)
        (symptom Flu Nausea)         (symptom Plague Nodules)))
```

4.5 Queries – Slightly Less Simple Rules

Diagnosing the same three patients, we get the same diagnoses, but the order of questions is somewhat different, determined by the way the symptoms are stored in the `symptom` slot of the disease frames.

```
> (dx-patient 'frank)
 Is it true that (has-symptom frank nodules)?  (Yes or No) no
 Is it true that (has-symptom frank nausea)?  (Yes or No) yes
 Give me a value for ?t in (temperature frank ?t): 100
  ((FRANK HAS FLU))

> (dx-patient 'george)
 Is it true that (has-symptom george nodules)?  (Yes or No) yes
 Give me a value for ?t in (temperature george ?t): 103
 Is it true that (has-symptom george nausea)?  (Yes or No) no
  ((GEORGE HAS PLAGUE))

> (dx-patient 'harry)
 Is it true that (has-symptom harry nodules)?  (Yes or No) yes
 Give me a value for ?t in (temperature harry ?t): 104
 Is it true that (has-symptom harry nausea)?  (Yes or No) yes
  ((HARRY HAS PLAGUE) (HARRY HAS FLU))
```

The rules contained explicit sequencing information that guaranteed that Frank was not asked about nodules after his relatively low fever made that question irrelevant. A different ordering of the disease descriptions gives a better order of questions, and eliminates the irrelevant question, but storage order within a slot is not a semantic property of the knowledge base, and cannot be relied on. Methods for ensuring a desired sequence will be discussed in section 6.

For comparison with the previous trace, here is a trace of Harry's diagnosis, using the slightly less verbose "Trace Interesting" mode.

```
algy> ti
  Trace level = :BASIC.

algy> (dx-patient 'harry)

  ** Beginning basic Trace **
 Creating new frame: harry.
 Inserting new value (isa harry things).
 Inserting new value (name harry "harry").
 Inserting new value (isa harry people).
   Applying: ((isa harry people) -> (isa harry objects)).
     Inserting new value (isa harry objects).
       Applying: ((isa harry objects) -> (isa harry things)).
       Rule applied.
   Rule applied.
   Applying: ((isa harry people) -> (isa harry physical-objects)).
     Inserting new value (isa harry physical-objects).
```

```
      Applying: ((isa harry physical-objects) -> (isa harry objects)).
      Rule applied.
  Rule applied.
Creating new frame: ri-13.
Applying: ((has-disease harry ?d) <- (member diseases ?d)
                                    (:all-paths ((symptom ?d ?s))
                                                ((has-symptom harry ?s)))).
  Inserting new value (query-queue set-partition (member diseases ?d)).
  Creating new frame: ri-14.
  Applying: ((has-symptom harry nodules) <- (:ask (has-symptom harry nodules))).
Is it true that (has-symptom harry nodules)?  (Yes or No) yes
    Inserting new value (has-symptom harry nodules).
  Rule applied.
  Creating new frame: ri-15.
  Applying: ((has-symptom harry high-fever) <- (temperature harry ?t)
                                              (:test (> ?t 102))).
    Creating new frame: ri-16.
    Applying: ((temperature harry ?t) <- (:ask (temperature harry ?t))).
Give me a value for ?t in (temperature harry ?t): 104
      Inserting new value (temperature harry 104).
    Rule applied.
    Inserting new value (has-symptom harry high-fever).
  Rule applied.
  Creating new frame: ri-17.
  Applying: ((has-symptom harry nausea) <- (:ask (has-symptom harry nausea))).
Is it true that (has-symptom harry nausea)?  (Yes or No) yes
    Inserting new value (has-symptom harry nausea).
  Rule applied.
  Creating new frame: ri-18.
  Applying: ((has-symptom harry fever) <- (temperature harry ?t)
                                         (:test (> ?t 99))).
    Inserting new value (has-symptom harry fever).
  Rule applied.
  Inserting new value (has-disease harry plague).
  Inserting new value (has-disease harry flu).
Rule applied.
 ** End basic Trace **
 ((HARRY HAS PLAGUE) (HARRY HAS FLU))
```

5 Calling Lisp and the User from Algernon

Algernon is embedded in Lisp, so it may be convenient to escape to Lisp for some operations. This is especially true for operations on Lisp data objects such as numbers, strings, or list structures; and for interaction with the user.

5.1 Evaluating Lisp Expressions

(`:eval` *expression*)	Substitute current bindings and evaluate *expression* for side-effect.
(`:test` *expression*)	Substitute current bindings and evaluate *expression*. Operation succeeds if value is non-nil, and fails otherwise.

The special forms `:eval` and `:test` take a Lisp expression including Algernon variables, substitute any bindings available for the Algernon variables, and evaluate the expression in Lisp.

We have already seen the `:test` special form used in the rule

```
((has-symptom ?p fever) <- (temperature ?p ?t) (:test (> ?t 99)))
```

to test whether the value bound to the variable `?t` satisfies a numerical relation.

For this and several following examples, we will use a small family tree and a few properties of people.

```
(tell '((:taxonomy (physical-objects
                    (people Adam Beth Charles Donna Ellen)))))

(tell '((:slot child    (people people))
        (:slot happy    (people booleans) :cardinality 1)
        (:slot friendly (people booleans) :cardinality 1)
        (:slot age      (people :number)  :cardinality 1)))

(tell '((child Adam Charles)
        (child Adam Donna)
        (child Adam Ellen)))
```

After asserting this information, we use the Algernon interface to inspect the contents of the frame describing Adam:

```
algy> vf adam
Adam:
  Name:       (adam)
  Isa:        objects physical-objects people things
  Child:      ellen donna charles
```

The following query looks up Adam's children and prints their names.

```
(ask '((child Adam ?kid)
       (:eval (format t "~%Adam has a child named ~a." '?kid)))
     :comment "Print names of all three children")
```

On querying the first predicate, `(child Adam ?kid)`, the path will branch three ways, one binding the variable `?kid` to each child. Along each branch, the `:eval` special form will evaluate the `format` expression to print a line. Notice that `'?kid` is quoted in the `format` expression. This is because the binding for the Algernon variable `?kid` is substituted into the expression before Lisp evaluates it. Since that binding is an Algernon frame, it will give an error if it is evaluated as a Lisp symbol, so it must be quoted.

```
QUERYING:  Print names of all three children

Adam has a child named ELLEN.
Adam has a child named DONNA.
Adam has a child named CHARLES.

 Result (1 of 3):
   Binding:            ?kid   --- ellen "[ellen]"

 Result (2 of 3):
   Binding:            ?kid   --- donna "[donna]"

 Result (3 of 3):
   Binding:            ?kid   --- charles "[charles]"

  => T
```

Now let's query the user (via the Lisp function `y-or-n-p`) to find out whether he or she likes each child, and assert that the child is happy if so.

```
  (tell '((child Adam ?kid)
          (:test (y-or-n-p "Do you like ~a?" '?kid))
          (:eval (format t "  ~a is happy!" '?kid))
          (happy ?kid true))
        :comment "Check whether each kid is liked by user.")
```

This path is asserted rather than queried because we want to assert `(happy ?kid true)` if the previous forms succeed, rather than checking whether it is currently known.

In this case, the user likes Ellen and Charles, but not Donna. Two paths from the three-way branch on `(child Adam ?kid)` succeed, while the other fails at the `:test`. Notice that the order in which the questions are asked and results are printed demonstrates that branches of the same path are followed in parallel.

```
ASSERTING:   Check whether each kid is liked by user.
Do you like ELLEN? (Y or N): y
Do you like DONNA? (Y or N): n
Do you like CHARLES? (Y or N): y
  ELLEN is happy!  CHARLES is happy!

 Result (1 of 2):
   Binding:            ?kid   --- ellen "[ellen]"

 Result (2 of 2):
   Binding:            ?kid   --- charles "[charles]"

  => T
```

After this interaction, we view the three frames and verify that, indeed, Ellen and Charles are now known to be happy. Although we could not infer that Donna is happy, we don't know that she is unhappy.

This illustrates an important principle of user-interface design: the user must always have a way to refuse to answer. In this case, we satisfied this requirement by interpreting a "No" answer to `y-or-n-p` as providing no information.

```
algy> vf ellen
Ellen:
  Name:       (ellen)
  Isa:        objects physical-objects people things
  Happy:      true

algy> vf donna
Donna:
  Name:       (donna)
  Isa:        objects physical-objects people things

algy> vf charles
Charles:
  Name:       (charles)
  Isa:        objects physical-objects people things
  Happy:      true
```

5.2 Returning Values from Lisp to Algernon

When a value is returned for a Lisp expression, we may want to bind that value to an Algernon variable, or even to branch on a list of bindings. These capabilities are provided by the special forms `:bind` and `:branch`.

(`:bind` *vars expression*)	Substitute current bindings into *expression* and evaluate. Unify *vars* with result. Continue if unification succeeds; fail otherwise.
(`:branch` *vars expression*)	Substitute current bindings into *expression* and evaluate. Unify *vars* with each element of resulting list. Continue along any path where unification succeeds; fail otherwise.

Suppose we have a Lisp function that returns a list structure.

```
  (defun lists-of-values ()
    '((1 2 3) (4 5 6) (7 8 9)))
```

We can evaluate `(lists-of-values)` and use `:bind` to capture the resulting list as the binding of an Algernon variable.

```
  (ask '((:bind ?val (lists-of-values)))
       :comment ":BIND elements of a list.")

QUERYING:  :BIND elements of a list.

 Result:
   Binding:             ?val   --- ((1 2 3) (4 5 6) (7 8 9))

  => T
```

In a similar construction, `:branch` will treat the elements of the list as alternate bindings for the variable.

```
  (ask '((:branch ?val (lists-of-values)))
       :comment ":BRANCH on elements of a list.")

QUERYING:  :BRANCH on elements of a list.

 Result (1 of 3):
   Binding:             ?val   --- (1 2 3)

 Result (2 of 3):
   Binding:             ?val   --- (4 5 6)

 Result (3 of 3):
   Binding:             ?val   --- (7 8 9)

  => T
```

Since the *vars* argument to `:bind` and `:branch` is unified against the value of the Lisp expression, it can be used for pattern matching and destructuring of values.

Suppose we are receiving information that we must filter and analyze.

```
  (defun gossip ()
    '((Tom loves Mary)
      (Bill hates Joe)
      (Nancy loves Sam)))
```

We can branch on each line of the information stream, succeed only at lines with verb "loves", and identify subject and object in each line. Here we demonstrate two ways to do this task.

```
  (ask '((:branch ?line (gossip))
         (:bind (?subj loves ?obj) '?line))
       :comment ":BIND to test and destructure")

  (ask '((:branch (?subj loves ?obj) (gossip)))
       :comment ":BRANCH to test and destructure")

QUERYING:  :BIND to test and destructure
```

```
 Result (1 of 2):
   Bindings:            ?obj   --- mary
                        ?subj  --- tom
                        ?line  --- (tom loves mary)

 Result (2 of 2):
   Bindings:            ?obj   --- sam
                        ?subj  --- nancy
                        ?line  --- (nancy loves sam)

  => T

QUERYING:  :BRANCH to test and destructure

 Result (1 of 2):
   Bindings:            ?obj   --- mary
                        ?subj  --- tom

 Result (2 of 2):
   Bindings:            ?obj   --- sam
                        ?subj  --- nancy

  => T
```

5.3 The List of Values in a Slot

Algernon normally treats a value b in the slot p of a frame a as an abbreviation for the logical predicate $p(a, b)$. However, there are times when it is useful to have the list of values (b_1 b_2 $\ldots b_n$) in a given Algernon slot.

(`:values` *frame slot*)	Return the list of values stored in *slot* of *frame*. Can serve as *expression* argument to previous forms.
(`:non-values` *frame slot*)	Return the list of non-values stored in *slot* of *frame*. Can serve as *expression* argument to previous forms.
(`:funcall` *function expression*$^+$)	Apply the Lisp function *function* to the arguments *expression*$^+$. Can serve as *expression* argument to previous forms.

In the following, we obtain the list of Adam's children in order to print it out.

```
  (ask '((:bind ?kids (:values Adam child))
         (:eval (format t "~%Adam's children are ~a." '?kids)))
       :comment ":BIND the list of values in a slot.")

QUERYING:  :BIND the list of values in a slot.

Adam's children are (ELLEN DONNA CHARLES).

 Result:
   Binding:            ?kids  --- (ellen "[ellen]" donna "[donna]" charles "[charles]")

  => T
```

We can also obtain the list of values in a slot to pass it to a Lisp function to compute some other value, in this case, the number of elements.

```
  (ask '((:bind ?n (:funcall #'length (:values Adam child)))
         (:eval (format t "~%Adam has ~a children." '?n)))
       :collect '(Adam has ?n children)
       :comment ":BIND and :FUNCALL evaluate a Lisp function")

QUERYING:  :BIND and :FUNCALL evaluate a Lisp function

Adam has 3 children.

 Result:
   Binding:            ?n     --- 3

  => ((ADAM HAS 3 CHILDREN))
```

5.4 Requesting Information from the User

The Lisp interface special forms make it easy to define functions for interacting with the user to request information for the Algernon knowledge base.

This section presents a simple user interface, intended to illustrate the basic method and encourage you to implement your own more sophisticated interface, tailored to the needs of your application.

This interface defines sets of slots associated with rules that invoke a Lisp function appropriate to that type of slot, to interact with the user.

- `User-ask-slots` are slots that automatically request information from the user if it is not already known. We have two categories at the moment: those that take Boolean values and those that take numerical values.

```
    (tell '((:taxonomy (Slots
                        (User-Ask-Slot
                         (Boolean-Ask-Slots)
                         (Numerical-Ask-Slots))))))
```

- Boolean ask-slots have rules (declared with `:srules` since they are accessed from the slot in a query or assertion) that call the CommonLisp function `y-or-n-p` to determine whether the user knows the truth of a given predicate.

```
    (tell '((:srules Boolean-Ask-Slots
             ((?p ?x true)
              <-
              (:test (y-or-n-p "Is it known that ~((~a ~a true)~)?" '?p '?x))))))
```

- Boolean ask-slots also have a set of forward-chaining rules to fill out the various equivalent ways a truth value can be known. (This is clearly redundant. Is it unacceptable? Should truth values have a canonical representation? Hard to say.)

```
(tell '((:srules Boolean-Ask-Slots
         ((?p ?x true)        -> (not (?p ?x false)))
         ((?p ?x false)       -> (not (?p ?x true)))
         ((not (?p ?x true))  -> (?p ?x false))
         ((not (?p ?x false)) -> (?p ?x true)))))
```

- Numerical-valued ask-slots get their values by calling a function named `ask-numerical-value-for`. Its code is provided in the Algernon example file `lisp-interface.lisp`. The user is requested to provide a value. By using the `:branch` special form, the rule can fail if the user refuses to provide a value.

```
(tell '((:srules Numerical-Ask-Slots
         ((?p ?f ?v)
          <-
          (:branch ?v (ask-numerical-value-for '?v '(?p ?f ?v)))))))
```

- There is also a function named `select-option` in the Algernon example file `lisp-interface.lisp`. It allows Algernon to present the user with a menu of options.

We demonstrate these features with some more interaction regarding Adam's family. We assert that the slots `friendly` and `age` belong to sets that will allow them to inherit the interaction rules above.

```
(tell '((isa (:slot friendly) Boolean-Ask-Slots)
        (isa (:slot age)      Numerical-Ask-Slots)))
```

For each of Adam's children, we query whether he or she is friendly. If this is known, we report the result. This is similar to a previous interaction, but specified on a rule inherited from a set.

```
(ask '((child Adam ?kid)
       (friendly ?kid ?tv)
       (:eval (format t "~%Is Adam's child ~a friendly?  ~a." '?kid '?tv)))
     :comment "Ask the user to provide a truth value.")

QUERYING:  Ask the user to provide a truth value.
Is it known that (friendly ellen true)? (Y or N): y
Is it known that (friendly donna true)? (Y or N): n
Is it known that (friendly charles true)? (Y or N): y

Is Adam's child ELLEN friendly?  TRUE.
Is Adam's child CHARLES friendly?  TRUE.

 Result (1 of 2):
   Bindings:           ?tv    --- true "[true]"
```

```
                        ?kid   --- ellen "[ellen]"

 Result (2 of 2):
   Bindings:            ?tv    --- true "[true]"
                        ?kid   --- charles "[charles]"

  => T
```

Next, for each child, ask the user for the child's age. Any non-numerical value counts as refusal to answer, so the rule and the attempt to query `(age ?kid ?n)` fail.

```
  (ask '((child Adam ?kid)
         (age ?kid ?n)
         (:eval (format t "~%~a's age is ~a." '?kid '?n)))
       :comment "Ask the user to provide a numerical value.")

QUERYING:  Ask the user to provide a numerical value.

Please give me a numerical value for ?n in (age ellen ?n):  12

Please give me a numerical value for ?n in (age donna ?n):  no

Please give me a numerical value for ?n in (age charles ?n):  16

ELLEN's age is 12.
CHARLES's age is 16.

 Result (1 of 2):
   Bindings:            ?n     --- 12
                        ?kid   --- ellen "[ellen]"

 Result (2 of 2):
   Bindings:            ?n     --- 16
                        ?kid   --- charles "[charles]"

  => T
```

Finally, we ask the user to select one of a list of options.

```
  (ask '((:eval (format t "~%Which kid is your favorite?"))
         (:bind ?options (:values Adam child))
         (:branch ?choice (select-option '?options ))
         (:eval (format t "~%You picked ~a." '?choice)))
       :comment "Ask the user to pick one of a given set of options.")

QUERYING:  Ask the user to pick one of a given set of options.

Which kid is your favorite?

+------------------------------
| Select one ('0' for none)
```

```
+------------------------------
|   1:   ELLEN
|   2:   DONNA
|   3:   CHARLES
+------------------------------
| Select item number (1 to 3):  1

You picked ELLEN.

 Result:
   Bindings:            ?choice   --- ellen "[ellen]"
                        ?options   --- (ellen "[ellen]" donna "[donna]" charles "[charles]")

  => T
```

6 Sequencing Methods

In a logical inference language such as Algernon, explicit attention is required to enforce or predict the sequence of operations.

6.1 Access Path Sequence

The access-path structure of individual rules enforces one type of ordering constraint.

Consider the following rule, taken from an expert system for the diagnosis of headache. It is invoked for a given patient, bound to the variable `?p`, on the assertion `(isa ?p Block6)`. Once the rule is invoked, the access path structure of the antecedent specifies the order in which the four predicates are queried. Each predicate backward-chains to an `:ask` form to query the user, and the questions are asked in the proper sequence. Forward-chaining rules are invoked to take an appropriate action in case any of the questions are answered `true`.

```
(:rules Block6

  ((isa ?p Block6)
   (flashes/spots-before-eyes-at-onset ?p ?tv1)
   (one-eye-red/tearful ?p ?tv2)
   (weakness-of-hand/arm/leg ?p ?tv3)
   (difficulty-speaking ?p ?tv4)
   ->
   (isa ?p Block7))

  ((flashes/spots-before-eyes-at-onset ?p True) -> (Dx ?p vascular-HA))
  ((one-eye-red/tearful ?p True)                -> (count-blues ?p True))
  ((weakness-of-hand/arm/leg ?p True)           -> (reds ?p True))
  ((difficulty-speaking ?p True)                -> (reds ?p True)))

(:rules Patients

((flashes/spots-before-eyes-at-onset ?p ?tv) <- (:ask (flashes/spots-before-eyes-at-onset ?p ?tv)))
((one-eye-red/tearful ?p ?tv)                <- (:ask (one-eye-red/tearful ?p ?tv)))
((weakness-of-hand/arm/leg ?p ?tv)           <- (:ask (weakness-of-hand/arm/leg ?p ?tv)))
((difficulty-speaking ?p ?tv)                <- (:ask (difficulty-speaking ?p ?tv))))
```

6.2 Block-to-Block Sequence

The overall structure of the Headache Expert System is enforced by the logical ordering on a set of blocks, each of which is associated with a set of questions. The above example shows the questions associated with `Block6`.

The blocks are represented as subsets of `Patients`. When the patient is asserted to be in a given block, the rules associated with that block are fired, as if-added rules triggered by `(isa ?p Blockn)`. When the block is complete, `(isa ?p Blockn+1)` is asserted. Thus, the patient accumulates membership in an increasing set of blocks. Membership is never retracted.

```
(:taxonomy (Patients
               . . .
```

```
              (Block6)
              (Block7)
               . . . ))
```

This control structure works provided that the blocks are organized in a loop-free transition net, rather a general finite-state machine. The strategy is consistent with the monotonic nature of Algernon, where information is only accumulated.

6.3 Counters

Counting is a non-monotonic operation, since the value of the counter is replaced by a new value. This makes it slightly awkward in Algernon, which is primarily oriented toward monotonic inference. However, by carefully using the non-monotonic `:clear-slot` form, we can implement a counter.

```
  (tell '((:taxonomy (Things
                       (Counters)))))

  (tell '((:slot increment     (Counters Booleans) :cardinality 1)
          (:slot current-count (Counters :number)  :cardinality 1)))

  (tell '((:rules Counters
           ((increment ?ctr true)
            (current-count ?ctr ?n)
            (:bind ?m (+ ?n 1))
            ->
            (:clear-slot ?ctr current-count)
            (:clear-slot ?ctr increment)
            (current-count ?ctr ?m)))))
```

The forward-chaining rule to increment a counter *c* is invoked by asserting `(increment` *c* `true)`. It binds a variable to the current count and computes the next value. In the consequent of the rule, it clears the count and asserts the next value. It also clears the predicate `(increment` *c* `true)` so that the next assertion of that value will invoke the forward-chaining rule again.

Testing it with the following assertion,

```
  (tell '((:a ?c (isa ?c Counters))
          (current-count ?c 0)
          (current-count ?c ?n1)
          (:eval (format t "~%Current counter value is ~a." ?n1))
          (increment ?c true)
          (current-count ?c ?n2)
          (:eval (format t "~%Current counter value is ~a." ?n2))
          (increment ?c true)
          (current-count ?c ?n)
          (:eval (format t "~%Current counter value is ~a." ?n)))
```

gives the following output.

```
ASSERTING:   Create and test a counter
```

```
Current counter value is 0.
Current counter value is 1.
Current counter value is 2.

 Result:
   Bindings:           ?n     --- 2
                       ?n2    --- 1
                       ?n1    --- 0
                       ?c     --- frame1 "[nil]"

Created frame:         frame1

  => T
  T

algy> vf frame1
Frame1:
  Isa:        things counters
  Current-count: 2
```