

# Associative Database Management

Wilensky Chapter 22

Shakil M. Khan  
adapted from Gunnar Gotshalks

## associative database

---

- an associative database is a collection of facts retrievable by their contents
  - is a poodle a dog?
  - which people does Alice manage?
- as opposed to retrieving facts by their position in the DB
  - give me the 10'th fact
  - what is the 10'th fact ?
- the facts in a database can be stored as patterns
- we can use a pattern matcher to search for facts in a database
  - match a query pattern against the patterns in the database looking for one or more matches

## example database facts

---

- simple facts have no variables
  - ( dog Fido )      Fido is a dog
  - ( loves John Mary )      John loves Mary
- can have more complex facts
  - ( implies ( dog ?x ) ( animal ?x ) )  
                    if x is a dog then x is an animal
  - ( loves ?x ?x )  
                    a person loves himself/herself
- one has to carefully consider how to represent facts; in the Lisp world it is customary to have the first item on a list be the main predicate and the remaining items be the arguments to the predicate

## example queries

---

- queries are patterns themselves – they can be without variables
  - is Fido a dog?      ( dog Fido )
  - does John love Mary?      ( loves John Mary )
- can have more complex queries
  - what is Fido?      ( ?what Fido )
  - who does John love?      ( loves John ?who )
  - who loves whom?      ( loves ?who ?whom )

# implementation

---

- in designing a database we need to consider how the facts will be stored
- in our first implementation, the facts are all stored in a list

```
( dog Fido )  
( loves John Mary )  
( implies ( dog ?x ) ( animal ?x ) )  
( loves ?x ?x )
```

→

```
( ( dog Fido )  
  ( loves john Mary )  
  ( implies (dog (*var* #:var11)) (animal (*var* #:var11)) )  
  ( loves ( *var* #:var12 ) ( *var* #:var12 ) )  
)
```

# add to the database

---

- store the database as the value of a symbol
- want to pass an unevaluated pattern and unevaluated symbol to our add operation
  - 1) use a macro
  - 2) change the value of the symbol to update the DB
  - consider the following:
    - DB: ( (loves ?x Mary) )
    - query: (loves John ?x)
    - try asking the query – it fails!
    - one solution : change the names of the variables in a fact as soon as it is added to the DB
  - 3) thus, need to replace the names of the pattern matching variables to be unique

## add to the database (cont'd)

---

```
(defmacro add-to-data-base ( item d-b-name )  
  `(setq ,d-b-name  
    (cons (replace-variables (quote ,item) )  
      ,d-b-name )  
  )  
)
```

## replace variable names

---

- replace the variables names in item
  - replacing variables names needs to be done consistently
  - create a binding list that keeps track of renaming
  - start off with a nil binding
  - returns the rebuilt item and the bindings of old and new variable names
- (defun replace-variables ( item )  
 ( values ( replace-variables-with-bindings  
 item nil )))

## replace variable names using bindings

- use the current bindings to replace variables consistently

```
(defun replace-variables-with-bindings ( item bindings )  
  - for an atom nothing to replace  
    ( cond ( ( atom item ) ( values item bindings ) )  
  - for a pattern variable return a replacement, if necessary  
    ((pattern-var-p item)  
     (let ((var-binding (get-binding item bindings)))  
       (if var-binding ; if on binding list return the binding  
         (values var-binding bindings)  
         ; else generate a new symbol  
         (let ((newvar (list '*var* (gensym "VAR")))  
             (values newvar (add-binding item newvar  
                                       bindings))))))
```

CSE-3401-June-18-2008

9

## replace variable names using bindings (cont'd)

- item is neither an atom nor a pattern variable
  - use recursion
- have to remember bindings from the "car" recursion for the "cdr" recursion

```
(t ( multiple-value-bind ( newlhs lhsbindings )  
    (replace-variables-with-bindings  
      ( car item ) bindings )  
  ( multiple-value-bind ( newrhs finalbindings )  
    ( replace-variables-with-bindings  
      ( cdr item ) lhsbindings )  
    ( values ( cons newlhs newrhs )  
            finalbindings ))))  
)
```

CSE-3401-June-18-2008

10

## replace variable examples

---

- (replace-variables '(loves John Mary)) →  
(LOVES JOHN MARY)
- (replace-variables '(loves ?x ?x)) →  
(LOVES (\*VAR\* #:VAR20) (\*VAR\* #:VAR20))

## create a database

---

- (setq DB nil)  
→ NIL
- (add-to-data-base (loves John Mary) DB)  
→ ((loves John Mary))
- (add-to-data-base (loves ?x ?x) DB)  
→ ((loves (\*var\* #:var22) (\*var\* #:var22))  
(loves John Mary))
- (add-to-data-base (dog Fido) DB)  
→ ((dog Fido)  
((loves (\*var\* #:var22) (\*var\* #:var22))  
(loves John Mary)))

# query the database

---

- use the matcher program to query the database
    - returns a list of bindings that match
- ```
(defun query ( request data-base )  
  ( mapcan  
    #'( lambda ( item )  
        ( multiple-value-bind ( flag bindings )  
          ( match item request )  
          ( if flag ( list bindings ) ) )  
    data-base ) )
```
- `mapcan` is like `mapcar` except it uses `nconc` in place of `append`
  - `nconc` is a destructive replacement of the `cdr` part of a cell for efficiency
  - `mapcan` also removes `nil` (see chapter 15, page 268-269)

CSE-3401-June-18-2008

13

# example queries

---

- `(query '( Fido dog ) DB ) ; not in database`  
→ `nil`
- `( query '( dog Fido ) DB ) ; in DB - no variables`  
→ `( nil )`
- `(query '(loves John John) DB) ; in DB - hidden variables`  
→ `(( ( ( *var* #:var22 ) John ) ) )`
- `( query '( dog ?name ) DB ) ; variable in query`  
→ `(( ( ( *var* name ) Fido ) ) )`
- `( query '( loves ?x ?y ) DB ) ; multiple matches`  
→ `(( ( (*var* x) (*var* y))(*var* #:var22 ) (*var* x )))  
 ( ( (*var* y ) Mary ) ( (*var* x) John ) )  
 )`

CSE-3401-June-18-2008

14

## implementation (cont'd)

---

- previous implementation becomes slow as the database increases in size
  - search is  $O(n)$  – where  $n$  is the number of facts
- reduce search time by indexing the facts
  - put facts with different predicates on different lists
  - put facts with the same predicate on the same list
  - search significantly shorter lists by only searching lists that match the predicate in the query
- the fact lists are put on the property list of the predicate with the key being the database symbol
  - facts could be in some databases and not in others

## indexing example

---

- enter the following into the indexed database

```
(index '( loves John Mary ) 'DB )
(index '( loves ?x ?x ) 'DB )
(index '( person John ) 'DB )
(index '( poodle Fido ) 'DB )
```
- then look at the property lists for the predicates

```
( symbol-plist 'person ) → ( db ( ( person John ) ) )
( symbol-plist 'poodle ) → ( db ( ( poodle Fido ) ) )
( symbol-plist 'loves ) →
  ( db ( ( loves (*var* #:var13) (*var* #:var13) )
    ( loves John Mary ) ) )
```

## other index lists

---

- the previous examples assumed facts would begin with an atom that could become a symbol with a property list
- what if a fact begins with a list?
  - for example, could represent "if x is a woman then x is mortal" as the following ( --> is a valid symbol in Lisp)  
`( ( ?x woman ) --> ( ?x mortal ) )`
  - have the special atom `*list*` to hold such facts
- what if a fact begins with a variable?
  - "everyone loves Barney" could be encoded as  
`( ?x loves Barney )`
  - have the special atom `*var*` to hold such facts

CSE-3401-June-18-2008

17

## what about searching the entire DB?

---

- if we have a query that begins with a variable, then the variable could match a variable, a list or any atom; hence the entire data base would need to be searched
- how can we do this if the database is scattered across the property lists of many symbols?
- have to keep track of the index symbols with the symbol for the database
  - add to the property list for the database symbol the list of `*keys*` that have been used as indices
  - in the example, several slides back, you could look at the symbol list for DB
- `(symbol-plist 'DB) → ( *keys* ( poodle person loves ) )`

CSE-3401-June-18-2008

18

# index function for a database

---

```
(defun index ( item data-base )  
  - place is where we want to store the item - use the key for the  
    pattern  
  (let ( ( place ( cond ( ( atom ( car item ) ) ( car item ) )  
                        ( ( pattern-var-p ( car item ) ) '*var* )  
                        ( t '*list* ) ) ) )  
    - store the item itself  
    (setf ( get place data-base )  
          ( cons ( replace-variables item ) ; rename variables  
                ( get place data-base ) ) )  
    - store the key for the item - adjoin adds only if not there  
    (setf ( get data-base '*keys* )  
          ( adjoin place ( get data-base '*keys* ) ) ) ) )
```

CSE-3401-June-18-2008

19

# fast query

---

```
(defun fast-query (request data-base)  
  (if (pattern-var-p (car request))  
      (mapcan #'(lambda (key) ; search entire DB  
                (query request (get key data-base)))  
              (get data-base '*keys*))  
      (nconc  
        ; else search under "atom" or *list*  
        (query request (get (if (atom (car request))  
                                (car request) '*list*)  
                            data-base))  
        )  
      ; add in search under *var* if "atom" or *list* search  
      (query request (get '*var* data-base))))
```

CSE-3401-June-18-2008

20

# deductive retrieval

---

- query and fast-query can't perform deduction
  - if  $a$  and  $a \rightarrow b$ , then  $b$
- we want to create a retriever that can do this

# deductive retrieval (cont'd)

---

- we use backward chaining
- store implications in the database in the following form  
(  $\leftarrow$  consequent antecedent )
- in addition to querying the database in the normal way, we add the following query  
(  $\leftarrow$  request ?antecedent )
- if this succeeds, we recursively query using the returned antecedent as a new request
- and so on – we proceed backwards from the query to the base facts

# deductive retrieval example

---

- let's add the following to the database

```
(index '( <- ( mammal ?x ) ( dog ?x ) ) 'DB )
(index '( <- ( dog ?x ) ( poodle ?x ) ) 'DB )
(index '( poodle fido ) 'DB )
```
- and make the following query

```
( mammal fido )
```

  - matches fact 1 using the implication search with request if ( dog fido )
- make the recursive query – matches fact 2
  - ( dog fido ) if ( poodle fido )
- make the recursive query - matches fact 3
  - return success ; no further recursion

CSE-3401-June-18-2008

23

# deductive retrieval function

---

```
(defun retrieve ( request data-base )
  • combine a regular search
  ( nconc ( fast-query request data-base )
  • with a recursive search over the implications
  (mapcan
    ... the function to apply to the implication search ...
  • get the next level of implication search – note the
    use of a macro to construct the pattern to use for the
    search
    (fast-query `( <- ,request ?antecedent )
      data-base )
  )))
```

CSE-3401-June-18-2008

24

## deductive retrieval function (cont'd)

---

... the function to apply to the implication search ...

```
#'( lambda ( bindings )  
  - search for each of the bindings of antecedent and add to  
    the list of bindings  
    ( mapcar #'( lambda ( rbindings )  
                  ( append rbindings bindings ) )  
              bindings )  
  - recursive search on an antecedent; need to replace the  
    variables in antecedent with their values, if any (e.g.  
    Fido for ?x in (dog ?x))  
    (retrieve ( substitute-vars  
                ( get-binding '?antecedent bindings )  
                bindings )  
              data-base)  
))
```

CSE-3401-June-18-2008

25

## substituting variables

---

- suppose we have the following binding list  
( (?antecedent (loves John ?y)) (?y ?z) (?z Mary))
- we do not want to search for the more general  
(loves John ?y)  
because we have bindings that restrict the value of ?y
- a first level substitution for ?z --> ?y yields a search  
pattern of  
(loves John ?z)
- but this is still too general as we have a binding for ?z
- need to do a second level, ?Mary --> ?z, **recursive  
substitution** to get the pattern we want to search on  
(loves John Mary)

CSE-3401-June-18-2008

26

# substitute variables for deductive retrieval

---

```
(defun substitute-vars (item bindings)
  - nothing to do if item is an atom
  (cond ((atom item) item)
        - potential substitution if a variable
        ((pattern-var-p item)
         (let ((binding (get-binding item bindings)))
           - substitute only if we have a binding for the item
           (if binding
               (substitute-vars binding bindings)
               item)))
        - have a list, so recursively substitute on first and rest
        (t (cons (substitute-vars (car item) bindings)
                  (substitute-vars (cdr item) bindings)))))
```

# **Prolog Introduction**

## **Clocksin & Mellish Ch 1 & 2**

Shakil M. Khan  
adapted from Gunnar Gotshalks

## **Prolog history**

---

- Prolog invented (1972) by the AI researcher Alan Colmerauer
  - used at York in the Student Information System to check applications for input errors
- widely used to develop expert systems & other AI applications including natural language processing
  - early ideas developed at University of Montreal; then University of Marseilles

## Prolog use & availability

---

- Prolog rumored to be embedded in MS Office
- on all major and many minor platforms
- several free and shareware versions
- standard: 'Edinburgh-style'

## low- and high-level

---

- Prolog is a higher-level language for knowledge-based programming
  - more powerful, not necessarily as efficient
  - more compact
  - more understandable programs.
- 'pure' Prolog:
  - denotational & declarative
  - just 1 state
    - a 'knowledge' base = database for facts

# Lisp vs. Prolog?

---

- which AI language an AI researcher uses often depends on where they studied
  - at Edinburgh, almost all Prolog
  - at MIT and Stanford, almost all Lisp
  - MIT has used a dialect of Lisp called Scheme in their first year programming course for many years
  - we have been more a 'Prolog shop' than a 'Lisp shop' in this department
  - prof. Stachniak teaches a 4th year course on Logic Programming which includes a more advanced look at Prolog

# what is a Prolog program?

---

- used for solving problems that involves *objects* and the *relationship* between objects
- Prolog is a conversational language
- programming in Prolog consists of:
  - declaring some *facts* about objects and their relationship
  - defining some *rules* about objects and their relationships
  - asking *questions* about objects and their relationships

# facts

---

- a program consists of a database containing **one or more** facts
  - a fact is a relationship between a collection of objects
- e.g.
  - **dog ( fido ).**  
Fido is a dog  
it is true that Fido is a dog
  - **mother ( mary, joe ).**  
Mary is the mother of Joe  
it is true that Mary is the mother of Joe
  - **compete ( ali, leila, tennis ).**  
Ali and Leila compete in tennis  
it is true that Ali and Leila compete in tennis

# facts (cont'd)

---

- the full stop character `'.'` must come at the end of a fact
- relationships can have any number of objects
- names are usually chosen to be meaningful
  - within Prolog, names are just arbitrary strings; it is people who give meaning to names
  - could have used **bSpears(sCowell)** rather than **dog(fido)**!

# rules

- and a program consists of a database of **zero or more** rules
  - a rule is an if...then relationship of facts
- e.g.
  - `use ( umbrella ) :- weather ( raining ) .`  
use an umbrella **if** it is raining
  - `use ( umbrella ) :- weather(raining) , own ( umbrella ) .`  
use an umbrella **if** it is raining and you own an umbrella
  - `use ( umbrella ) :- weather ( raining ) ,  
( own ( umbrella ) ; borrow ( umbrella ) ) .`  
use an umbrella **if** it is raining and you either own an umbrella or can borrow an umbrella

## more on rules

- rules have the general structure
  - **head :- body**
    - only one fact can be in the head – the consequent
    - the body is a Boolean combination of predicates
    - use **,** (and) and **;** (or) and **()** (parenthesis) to logically organize the "condition" – the antecedent/body
- rules are written backwards to
  - emphasize the backward chaining for database search
  - be more regular in structure, since the head is only one predicate

# constants

---

- constants are names of that begin with lower case letters
  - e.g. `ali`, `leila`, `tennis`, `dog`, `fido`, `mother`, `mary`, `joe`, `umbrella`, `raining`, `weather`, `own`, `borrow`
  - names of relationships are constants
  - see text (C&M 2.1.1) for a complete set of rules for the syntax of constants


# variables

---

- in place of constants in facts and rules one can have variables
  - variables are names that begin with upper case letters
  - e.g. `X`, `Y`, `Who`, `Whom`, `List`, `Person`
- example of rules with variables
  - `loves ( Everyone, barney ).`  
everyone loves barney  
for all values of Everyone it is the case that  
`loves(Everyone, barney)` is true
  - `sister_of (X, Y) :- female (X),`  
`parents (X, M, F), parents (Y,M,F).`  
for all X and Y, X is a sister of Y if...

## variables (cont'd)

---

- `dwarf ( Person ) :- brother ( Person, Other ) ,  
dwarf ( Other ) .`  
a person is a dwarf, if they the brother of other and  
the other is a dwarf
- variables can also begin with `_` (underscore)
  - `_` (anonymous variable)
  - `_1 _abc` (not anonymous variable )
  - several anonymous variables in the same structure need  
not be given consistent interpretations  
`c1 (X) :- a1 (X, _), a2 (X), a3 (_).`  
  
may represent two different variables

## queries

---

- a query in Prolog is Boolean combination of predicates – like the antecedent of a rule
  - a query is like a rule, except we leave out the consequent true
  - `true :- dwarf ( alberich ) .`  
becomes simply  
`dwarf ( alberich ) .`
- use comma (and), semicolon (or), and parenthesis to form a query expression
- most common is to have a single predicate

## queries (cont'd)

---

- **answer** is a **binding of the variables** that make the query expression **true** – if no variables then the answer is **yes**; if no such binding exists, the answer is **no**
- the database is searched to match the query (similar to the Lisp database program)
- the search
  - uses backward chaining
  - is depth first
  - is sequential through the database from first to last
- try the exercise on ring.pro

## satisfying goals - backtracking

---

- example DB:
  - `/*1*/ female (mary).`
  - `/*2*/ parent (john, ann, fred).`
  - `/*3*/ parent (mary, ann, fred).`
  - `/*4*/ sister_of (X,Y) :-`  
    `female (X),`  
    `parent (X, M, F), parent (Y, M, F).`
- query:  
    `sister_of (mary, X), female (X).`

## satisfying goals – backtracking (cont'd)

---

- flow of satisfaction (summary)
  - try to match `sister_of (mary, X)`
    - if found in DB, try to satisfy `female (X)` with that binding of X;
      - if found, return that binding;
      - else, `backtrack` to previous goal and try to find another X for which `sister_of` holds; if found continue with this new binding as before; else return `no`
    - else return `no`

## flow of satisfaction

---

- `sister_of (mary, X)`
  - matches with the head of rule 4; mark this position and try to satisfy the body of this rule
  - `female (mary)`, `parent (mary, M, F)`,  
`parent (X, M, F)`.
  - `female (mary)` satisfied by fact 1; mark this position and try to satisfy `parent (mary, M, F)`
  - `parent (mary, M, F)` matches with fact 3 with bindings `parent (mary, ann, fred)`; mark this, and try to satisfy `parent (X, ann fred)`

## flow of satisfaction (cont'd)

---

- `parent(X, ann, fred)` matches with `fact 2` with binding `X → john`; mark this position; thus `sister_of(mary, X)` is satisfied with binding `sister_of(mary, john)`
- now try to satisfy `female(john)`
  - it fails!
- backtrack to `sister_of`
  - the last subgoal of `sister_of` was `parent(X, ann, fred)`, which earlier succeeded with `X → john`; ignore this binding and try to resatisfy it starting from after `fact 2`

## flow of satisfaction (cont'd)

---

- `parent(X, ann, fred)` matches with `fact 3` with binding `X → mary`; mark this position; thus `sister_of(mary, X)` is satisfied with binding `sister_of(mary, mary)`
- now try to satisfy `female(mary)`
  - it matches with fact 1; thus the whole query succeeds with binding `X → mary`, and Prolog returns this binding
- note: we can initiate backtracking using `;`

## running a Prolog program

---

- programs are stored in one or more files that are consulted
- on Prism to run SWI Prolog enter  
    % pl
- the following prompt appears  
    | ?-
- consult the appropriate file(s) – add to the database  
    | ?- consult('ring.pro').
  - while it is possible to enter facts and rules interactively using `consult( user )`, it is inconvenient and error prone
  - SWI-prolog does not have a re-consult predicate, only consult is used

CSE-3401-June-18-2008

21

## running a Prolog program (cont'd)

---

- make zero or more queries
- exit prolog  
    | ?- CTRL-d      /\* and for exiting consult ( user )  
                      on Prism \*/
- `consult(user)` enables you to enter facts & rules into the database without storing them in a file; it is not an effective way to work with Prolog

CSE-3401-June-18-2008

22

# info

---

- class test 1 marks are out
  - max : 38/40 A+
  - min : 12/40 F
  - median : 23/40 D+
  - average : 22.24/40 D+
  - use [courseInfo 3401](#) to check your marks
- check course page after Friday for assignment 2
- this Friday's office hours are cancelled