Logic Programming

A language that doesn't affect the way you think about programming is not worth knowing.

— Alan Perlis
Logic Programming

→ In logic programming, the programmer specifies constraints on the solution to a problem, but finding the solution is left to the system

→ Approximation to this is Prolog (= programming in logic)

→ Relies on three ideas:

1) A single uniform database

2) logic variables (bound by unification, not assignment)

3) Automatic backtracking
Idea 1: A Uniform Data Base

→ In logic programming, the programmer specifies constraints on the solution to a problem as constraints or clauses that are added to a uniform database

Example 1: “the population of San Francisco is 750,000”

Functional programming: function “population”

> (defun population (C) …)
> (population SF)
=> 750,000

Logic programming: relation “population”

> (<- (population SF 750,000))
> (?- (population SF ?pop))
=> ?pop = 750,000

Relations are more general. Why?
In logic programming, the programmer specifies constraints on the solution to a problem as constraints or **clauses that are added to a uniform database**

Example 2: “likes” relations

\[
\begin{align*}
& (\text{likes} \text{ Kim} \text{ Robin}) \\
& (\text{likes} \text{ Sandy} \text{ Lee}) \\
& (\text{likes} \text{ Sandy} \text{ Kim}) \\
& (\text{likes} \text{ Robin} \text{ cats})
\end{align*}
\]

These represent the facts that Kim likes Robin, Sandy likes Lee, etc.,

**Question:** do these representations also have these respective meanings? (cf McDermott's “AI meets NS” paper)
Some “meaning” or knowledge may be added by adding rules to the database

For example, the knowledge that Sandy likes anyone who likes cats is added as follows:

\[(\leftarrow \text{likes Sandy } ?x \text{ (likes } ?x \text{ cats)})\]

This is a logical assertion, a declaration that for any \(?x\), the fact \((\text{likes Sandy } ?x)\) is true if \((\text{likes } ?x \text{ cats})\) is true.

As a piece of prolog code, it has a procedural backward-chaining interpretation, saying that “If you ever need to find an \(?x\) for which \((\text{likes Sandy } ?x)\) is true, find one for which \((\text{likes } ?x \text{ cats})\) is true.

What would be a procedural forward-chaining interpretation?
Idea 1: A Uniform Data Base

→ Some “meaning” or knowledge may be added by adding rules to the database

Example 2, the following assertion

\[ (\leftarrow \text{likes Kim } ?x \text{ ) (likes } ?x \text{ Lee) (likes } ?x \text{ Kim)} \]

declares that Kim likes anyone who likes both Lee and Kim.

What are the backward and forward-chaining interpretations?
A clause '(← head body)' is represented as a cons cell:

```
(defun clause-head (clause) (first clause))
(defun clause-body (clause) (rest clause))
```

How should clauses be indexed in Prolog?
Idea 1: A Uniform Data Base

Implementation: representing and storing clauses

A clause '(* head body)' is represented as a cons cell:

```lisp
(defun clause-head (clause) (first clause))
(defun clause-body (clause) (rest clause))
```

How should clauses be indexed in Prolog?

backward-chaining interpretation: “when you want to prove the head, try to prove the body” => on the head

```lisp
(defun get-clauses (pred) (get pred 'clauses))
(defun predicate (relation) (first relation))
(defun *db-predicates* nil
    "A list of all predicates stored in the database.")
```
Idea 1: A Uniform Data Base

Implementation: **adding clauses**

Defining prolog in lisp is like defining a new language, with its own syntax and semantics, so we need a macro

```lisp
(defmacro <- (&rest clause)
  "Add a clause to the data base."
  '(add-clause ,clause))

(defun add-clause (clause)
  "Add a clause to the data base, indexed by head's predicate."
  ;; The predicate must be a non-variable symbol.
  (let ((pred (predicate (clause-head clause))))
    (assert (and (symbolp pred) (not (variable-p pred))))
    (pushnew pred *db-predicates*)
    (setf (get pred 'clauses)
      (nconc (get-clauses pred) (list clause))
      pred))
```
Idea 1: A Uniform Data Base

→ Implementation: removing clauses

(defun clear-db ()
  "Remove all clauses (for all predicates) from the database."
  (mapc #'clear-predicate *db-predicates*))

(defun clear-predicate (predicate)
  "Remove the clauses for a single predicate."
  (setf (get predicate 'clauses) nil))

→ What about retrieving clauses?
Idea 2: Unification of Logic Variables

→ Unification is like pattern matching of patterns (where a pattern may contain variables)

\[
>\text{(pat-match '(?x + ?y) '(2 + 1)) } \Rightarrow (\text{('?Y . 1) ('?X . 2))}
\]

\[
>\text{(unify '(?x + 1) '}(2 + ?y)) \Rightarrow (\text{('?Y . 1) ('?X . 2))}
\]

→ Within the unification framework, variables are logic variables (internal parameters of quantifications)

Once assigned a value, a logic variable cannot be altered

But they can be identified or equated

\[
>\text{(unify '}(f ?x) '}(f ?y)) \Rightarrow (\text{('?X . ?Y))}
\]
Idea 2: Unification of Logic Variables

→ “Sophisticated reasoning” is possible with unification:

> (unify '(?a + ?a = 0)' '(?x + ?y = ?y)) ⇒
  ((?Y . 0) (?X . ?Y) (?A . ?X))

> (unifier '(?a + ?a = 0)' '(?x + ?y = ?y)) ⇒ (0 + 0 = 0)
Idea 2: Unification of Logic Variables

→ “Sophisticated reasoning” is possible with unification:

> (unify '(?a + ?a = 0) '(?x + ?y = ?y)) ⇒ 
((?Y . 0) (?X . ?Y) (?A . ?X))

> (unifier '(?a + ?a = 0) '(?x + ?y = ?y)) ⇒ (0 + 0 = 0)

→ However:

Unification does provide a way to identify variables (i.e., constrain their value domain)

But it does not provide a way to automatically solve equations or apply other constraints other than equality

> (unifier '(?a + ?a = 2) '(?x + ?y = ?y)) ⇒ (2 + 2 = 2)
Idea 2: Unification of Logic Variables

→ The main difference between pattern matching and unification is that the both inputs may contain variables

(defun unify (x y &optional (bindings no-bindings))
  "See if x and y match with given bindings."
  (cond ((eq bindings fail) fail)
        ((variable-p x) (unify-variable x y bindings))
        ((variable-p y) (unify-variable y x bindings))
        ((eql x y) bindings)
        ((and (consp x) (consp y))
         (unify (rest x) (rest y)
                (unify (first x) (first y) bindings)))
        (t fail)))
Idea 2: Unification of Logic Variables

→ The main difference between pattern matching and unification is that the both inputs may contain variables

→ This “extended power” has some implications

(defun unify-variable (var x bindings)
   "Unify var with x, using (and maybe extending) bindings."
   ;; Warning - buggy version
   (if (get-binding var bindings)
       (unify (lookup var bindings) x bindings)
       (extend-bindings var x bindings)))

> (unify '(?x + 1) '(2 + ?y)) ⇒ ((?Y . 1) (?X . 2))
> (unify '?x '?y) ⇒ ((?X . ?Y))
> (unify '(?x ?x) '(?y ?y)) ⇒ ((?Y . ?Y) (?X . ?Y))
> (unify '(?x ?x ?x) '(?y ?y ?y))
>>Trap #o43622 (PDL-OVERFLOW REGULAR)
Idea 2: Unification of Logic Variables

→ Solution, part 1: test for "equality of variables" early

```
(defun unify (x y &optional (bindings no-bindings))
  "See if x and y match with given bindings."
  (cond ((eq bindings fail) fail)
        ((eql x y) bindings) ;*** moved this line
        ((variable-p x) (unify-variable x y bindings))
        ((variable-p y) (unify-variable y x bindings))
        ((and (consp x) (consp y))
         (unify (rest x) (rest y)
              (unify (first x) (first y) bindings)))
        (t fail)))
```

> (unify '(?x ?x) '(?y ?y)) ⇒ ((?X . ?Y))
> (unify '(?x ?x ?x) '(?y ?y ?y)) ⇒ ((?X . ?Y))
> (unify '(?x ?y) '(?y ?x)) ⇒ ((?Y . ?X) (?X . ?Y))
> (unify '(?x ?y a) '(?y ?x ?x))
>>Trap #043622 (PDL-OVERFLOW REGULAR)
Idea 2: Unification of Logic Variables

→ Solution, part 2: deal with the value of variables for comparison

(funcunify-variable (var x bindings)
  "Unify var with x, using (and maybe extending) bindings."
  (cond ((get-binding var bindings)
          (unify (lookup var bindings) x bindings))
         ((and (variable-p x) (get-binding x bindings)) ;***
          (unify var (lookup x bindings) bindings)) ;***
         (t (extend-bindings var x bindings))))

> (unify '(?x ?y) '(?y ?x)) ⇒ ((?X . ?Y))
> (unify '(?x ?y a) '(?y ?x ?x)) ⇒ ((?Y . A) (?X . ?Y))

→ What about the following?

> (unify '?x '(f ?x)) ⇒ ((?X F ?X))
Idea 2: Unification of Logic Variables

Solution, part 3: test for the occurrence of variables in values

(defun unify-variable (var x bindings)
  "Unify var with x, using (and maybe extending) bindings."
  (cond ((get-binding var bindings)
       (unify (lookup var bindings) x bindings))
       ((and (variable-p x) (get-binding x bindings))
       (unify var (lookup x bindings) bindings))
       ((and *occurs-check* (occurs-check var x bindings))
        fail)
       (t (extend-bindings var x bindings))))

(defun occurs-check (var x bindings)
  "Does var occur anywhere 'inside x?"
  (cond ((eq var x) t)
       ((and (variable-p x) (get-binding x bindings))
        (occurs-check var (lookup x bindings) bindings))
       ((consp x) (or (occurs-check var (first x) bindings)
                        (occurs-check var (rest x) bindings))
        (t nil)))
Idea 2: Unification of Logic Variables

Examples

> (unify '(*x a) (*y x)) ⇒ ((?Y . A) (?X . ?Y))

> (unify '?x '(f ?x)) ⇒ NIL

> (unify '(*x y) '((f y) (f ?x))) ⇒ NIL

> (unify '(*x y z) '((y z) (x z) (x y))) ⇒ NIL

> (unify 'a 'a) ⇒ ((T . T))

With *occurs-check* = NIL

> (unify '?x '(f ?x)) ⇒ ((?X F ?X))

> (unify '*x y) '((f y) (f ?x)) ⇒
((?Y F ?X) (?X F ?Y))

> (unify '*x y z) '((y z) (x z) (x y)) ⇒
Idea 2: Unification of Logic Variables

Note, because the value in a binding can now be a variable, The lisp `sublis` function is no longer sufficient.

```lisp
(defun subst-bindings (bindings x)
    "Substitute the value of variables in bindings into x, taking recursively bound variables into account."
    (cond ((eq bindings fail) fail)
          ((eq bindings no-bindings) x)
          ((and (variable-p x) (get-binding x bindings))
             (subst-bindings bindings (lookup x bindings)))
          ((atom x) x)
          (t (reuse-cons (subst-bindings bindings (car x))
                         (subst-bindings bindings (cdr x))
                         x))))
```
Idea 2: Unification of Logic Variables

→ Note, because the value in a binding can now be a variable, The lisp `sublis` function is no longer sufficient

```lisp
(defun unifier (x y)
  "Return something that unifies with both x and y (or fail)."
  (subst-bindings (unify x y) x))
```

```lisp
> (unifier '(?x ?y a) '(?y ?x ?x)) ⇒ (A A A)
```

```lisp
> (unifier '((?a * ?x ^ 2) + (?b * ?x) + ?c)
  '(?z + (4 * 5) + 3)) ⇒
  ((?A * 5 ^ 2) + (4 * 5) + 3)
```
Idea 2: Unification of Logic Variables

→ We can now implement the prolog database “lookup” functionality.

The function prove checks if a given goal unifies with a fact in the database or can be derived by applying the rules on facts.

To derive a fact, we can use a rule that entails it (i.e. of which the head unifies with the goal), if we can prove all “goals” in the body of the rule.

(defun prove (goal bindings)
   "Return a list of possible solutions to goal."
   (mapcan #'(lambda (clause)
      (let ((new-clause (rename-variables clause)))
         (prove-all (clause-body new-clause)
                    (unify goal (clause-head new-clause) bindings))))
      (get-clauses (predicate goal))))
Idea 2: Unification of Logic Variables

(defun prove (goal bindings)
  "Return a list of possible solutions to goal."
  (mapcan #'(lambda (clause)
      (let ((new-clause (rename-variables clause)))
        (prove-all (clause-body new-clause)
          (unify goal (clause-head new-clause) bindings)))))
  (get-clauses (predicate goal)))

(defun prove-all (goals bindings)
  "Return a list of solutions to the conjunction of goals."
  (cond ((eq bindings fail) fail)
        ((null goals) (list bindings))
        (t (mapcan #'(lambda (goal1-solution)
            (prove-all (rest goals) goal1-solution)
            (prove (first goals) bindings))))))

(defun rename-variables (x)
  "Replace all variables in x with new ones."
  (sublis (mapcar #'(lambda (var) (cons var (gensym (string var)))))
    (variables-in x))
  x))
Idea 2: Unification of Logic Variables

(defun unification)

(defmacro ?- (&rest goals) '(prove-all ',goals no-bindings))

(\ - (likes Kim Robin))
(\ - (likes Sandy Lee))
(\ - (likes Sandy Kim))
(\ - (likes Robin cats))
(\ - (likes Sandy ?x) (\ ikes ?x cats))
(\ - (likes Kim ?x) (\ ikes ?x Lee) (\ ikes ?x Kim))
(\ - (likes ?x ?x))

> (?- (likes Sandy ?who))
Idea 2: Unification of Logic Variables

(defmacro ?- (&rest goals) '(prove-all ',goals no-bindings))

(<- (likes Kim Robin))
(<- (likes Sandy Lee))
(<- (likes Sandy Kim))
(<- (likes Robin cats))
(<- (likes Sandy ?x) (likes ?x cats))
(<- (likes Kim ?x) (likes ?x Lee) (likes ?x Kim))
(<- (likes ?x ?x))

> (?- (likes Sandy ?who))

((?WHO . LEE))
((?WHO . KIM))
((?X2856 . ROBIN) (?WHO . ?X2856))
((?X2860 . CATS) (?X2857 . CATS) (?X2856 . SANDY) (?WHO . ?X2856))
((?X2865 . CATS) (?X2856 . ?X2865) (?WHO . ?X2856))
((?WHO . SANDY) (?X2867 . SANDY)))
Idea 2: Unification of Logic Variables

The power or “relational” or declarative programming revisited:

```prolog
> (?- (likes Sandy ?who))
?WHO = LEE;
?WHO = KIM;
?WHO = ROBIN;
?WHO = SANDY;
?WHO = CATS;
?WHO = SANDY;
```

```prolog
> (?- (likes ?who Sandy))
?WHO = SANDY;
?WHO = KIM;
?WHO = SANDY;
```

```prolog
> (?- (likes Robin Lee))
No.
```
Idea 2: Unification of Logic Variables

The power or “relational” or declarative programming revisited:

```
> (?- (likes ?x ?y) (likes ?y ?x))
?Y = KIM
?X = SANDY;
?Y = SANDY
?X = SANDY;
?Y = SANDY
?X = SANDY;
?Y = SANDY
?X = SANDY;
?Y = SANDY
?X = SANDY;
?Y = SANDY
?X = SANDY;
?Y = ?X3251
?X = ?X3251;
```
Idea 2: Unification of Logic Variables

- Note that there can be different ways to prove the same clause

- All of them are reported in the implementation, which therefore includes and exhaustive search

- The search strategy is “top down” (first the top goal, then the clauses that entail it etc.) and “left to right” (first the first clause, then the second etc.)

- Prolog in lisp through “pretty printing”, macros, and lisp's “universal syntax”:
The power or “relational” or declarative programming with unification

Prolog clauses allow to express relations that we normally think of as “program”, not “data”.

For instance, we can define the “member” relation:

\[
(- (\text{member} \ ?\text{item} \ (?\text{item} \ . \ ?\text{rest}))) \\
(- (\text{member} \ ?\text{item} \ (?x \ . \ ?\text{rest})) (\text{member} \ ?\text{item} \ ?\text{rest}))
\]

In lisp, this would be

\[
(\text{defun lisp-member} \ (\text{item} \ \text{list}) \hspace{2cm} \\
(\text{and} \ (\text{consp} \ \text{list}) \hspace{2cm} \\
 (\text{or} \ (\text{eql} \ \text{item} \ (\text{first} \ \text{list})) \\
 \hspace{2cm} (\text{lisp-member} \ \text{item} \ (\text{rest} \ \text{list}))))))
\]
The power or “relational” or declarative programming with unification

Prolog clauses allow to express relations that we normally think of as “program”, not “data”.

For instance, we can define the “member” relation:

\[
\text{(<- (member ?item (?item . ?rest)))} \\
\text{(<- (member ?item (?x . ?rest)) (member ?item ?rest))}
\]

For example:

\[
\text{> (?- (member 2 (1 2 3)))} \\
\text{Yes;} \\
\text{> (?- (member ?x (1 2 3)))} \\
\text{?X = 1;} \\
\text{?X = 2;} \\
\text{?X = 3;}
\]

\[
\text{> (?- (member 2 (1 2 3 2 1)))} \\
\text{Yes;} \\
\text{Yes;} \\
\text{Yes;}
\]
The power or “relational” or declarative programming with unification

Prolog clauses allow to express relations that we normally think of as “program”, not “data”.

For instance, we can define the “member” relation:

\[
\text{member} \ ?\text{item} \ ?\text{item} (\ ?\text{rest} \ . \ ?\text{rest})
\]

\[
\text{member} \ ?\text{item} \ ?\text{x} (\ ?\text{rest})\text{member} \ ?\text{item} \ ?\text{rest}
\]

Exercises:

- Define the set of all natural numbers in Prolog
- What relation is involved?
- what are the sizes of its domain and the codomain?
The power or “relational” or declarative programming with unification

In fact, there is too much power for our current implementation to handle

E.g., how to make sense with queries like the following?

?- (member 2 ?list))
?- (member ?item ?list))

Or how to deal with the fact that the set of natural numbers is infinitely large?

← (natural-number 0))
← (natural number (+ ?n 1)) (natural-number ?n))

?- (natural number ?x))
?X = 0
?X = (+ 0 1)
?X = (+ (+ 0 1) 1)
?X = (+ (+ (+ 0 1) 1) 1)
?X = (+ (+ (+ (+ 0 1) 1) 1) 1)
...

Idea 3: Automatic Backtracking

→ The general idea is to compute only a single solution, and delay the computation of the other solutions until another one is requested

Approach 1: use “delayed” or “lazy” evaluation, that is, use “pipes” rather than lists:

(Charniak et al. 1987)

(defun prove (goal bindings)
  "Return a list of possible solutions to goal."
  (mappend-pipe #'(lambda (clause)
                    (let ((new-clause (rename-variables clause)))
                        (prove-all (clause-body new-clause)
                                   (unify goal (clause-head new-clause)
                                              bindings))))
                    (get-clauses (predicate goal))))

(defun prove-all (goals bindings)
  "Return a list of solutions to the conjunction of goals."
  (cond ((eq bindings fail) fail)
        ((null goals) (list bindings))
        (t (mappend-pipe #'(lambda (goall-solution)
                             (prove-all (rest goals) goall-solution))
                         (prove (first goals) bindings)))))
Idea 3: Automatic Backtracking

Approach 1: use “delayed” or “lazy” evaluation with “pipes” rather than lists:

(defmacro make-pipe (head tail)
  "Create a pipe by evaluating head and delaying tail."
  '(cons .head #'(lambda () .tail))
)

(defconstant empty-pipe nil)

(defun head (pipe) (first pipe))

(defun tail (pipe)
  "Return tail of pipe or list, and destructively update the tail if it is a function."
  (if (functionp (rest pipe))
      (setf (rest pipe) (funcall (rest pipe)))
      (rest pipe)))

(defun pipe-elt (pipe i)
  "The i-th element of a pipe, 0-based"
  (if (= i 0)
      (head pipe)
      (pipe-elt (tail pipe) (- i 1))))
Idea 3: Automatic Backtracking

Approach 1: use "delayed" or "lazy" evaluation with "pipes" rather than lists:

```lisp
(defun integers (&optional (start 0) end)
  "A pipe of integers from START to END. If END is nil, this is an infinite pipe."
  (if (or (null end) (<= start end))
    (make-pipe start (integers (+ start 1) end)
     nil))

> (setf c (integers 0)) ⇒ (0 . #<CLOSURE 77350123>)
> (pipe-elt c 0) ⇒ 0
> (pipe-elt c 5) ⇒ 5
> c ⇒ (0 1 2 3 4 5 . #<CLOSURE 77351636>)
> (setf i (integers 0 10)) ⇒ (0 . #<CLOSURE 77375357>)
> (pipe-elt i 10) ⇒ 10
> (pipe-elt i 11) ⇒ NIL
> i ⇒ (0 1 2 3 4 5 6 7 8 9 10)
```
Idea 3: Automatic Backtracking

Approach 2: keeping track of alternatives and making use of “primitives”:

(1) Change prove and prove-all so that they compute only a single solution

(2) Change prove so that it can handle “primitives”

(3) Add a primitive to show a solution and ask if more solutions are required

(4) Define a top-level-prove that automatically adds the defined primitive to the end of the list of goals
Idea 3: Automatic Backtracking

Approach 2: keeping track of alternatives and making use of “primitives”

(1) Change prove and prove-all so that they compute only a single solution

(defun prove-all (goals bindings)
  "Find a solution to the conjunction of goals."
  (cond ((eq bindings fail) fail)
       ((null goals) bindings)
       (t (prove (first goals) bindings (rest goals))))))

(defun prove (goal bindings other-goals)
  "Return a list of possible solutions to goal."
  (some #'(lambda (clause)
    (let ((new-clause (rename-variables clause)))
      (prove-all
        (append (clause-body new-clause) other-goals)
        (unify goal (clause-head new-clause) bindings)))
      (get-clauses (predicate goal)))))
Idea 3: Automatic Backtracking

Approach 2: keeping track of alternatives and making use of “primitives”

(2) Change `prove` so that it can handle “primitives”

```lisp
(defun prove (goal bindings other-goals)
  "Return a list of possible solutions to goal."
  (let ((clauses (get-clauses (predicate goal))))
    (if (listp clauses)
        (some
         #'(lambda (clause)
             (let ((new-clause (rename-variables clause)))
               (prove-all
                (append (clause-body new-clause) other-goals)
                (unify goal (clause-head new-clause) bindings)))
            clauses)
        ;; The predicate's "clauses" can be an atom:
        ;; a primitive function to call
        (funcall clauses (rest goal) bindings
                  other-goals)))
```
Idea 3: Automatic Backtracking

Approach 2: keeping track of alternatives and making use of “primitives”

(3) Add a primitive to show a solution and ask if more solutions are required

```
(setf (get 'show-prolog-vars 'clauses) 'show-prolog-vars)

(defun show-prolog-vars (vars bindings other-goals)
  "Print each variable with its binding. Then ask the user if more solutions are desired."
  (if (null vars)
    (format t "~&Yes")
    (dolist (var vars)
      (format t "~&~a = "a" var
                   (subst-bindings bindings var))))
  (if (continue-p)
    fail
    (prove-all other-goals bindings)))
```

Three arguments

Returns either fail or calls prove-all to continue
Idea 3: Automatic Backtracking

Approach 2: keeping track of alternatives and making use of “primitives”

(3) Add a primitive to show a solution and ask if more solutions are required

(defun continue-p ()
  "Ask user if we should continue looking for solutions."
  (case (read-char)
    (#\; t)
    (#\. nil)
    (#\newline (continue-p))
    (otherwise
      (format t " Type ; to see more or . to stop")
      (continue-p))))
Idea 3: Automatic Backtracking

Approach 2: keeping track of alternatives and making use of "primitives"

(4) Define a top-level-prove that automatically adds the defined primitive to the end of the list of goals

(defun top-level-prove (goals)
  (prove-all ‘(,@goals (show-prolog-vars ,@(variables-ingoals))
               no-bindings)
  (format t "~&No.") (values))
The power or "relational" or declarative programming with unification and automatic backtracking

→ What is the set of lists that contain 2?

> (?- (member 2 ?list))
?LIST = (2 . ?REST3302);
?LIST = (?X3303 2 . ?REST3307);
?LIST = (?X3303 ?X3308 2 . ?REST3312);
No.

→ When is an item a member of a list?

> (?- (member ?item ?list))
?ITEM = ?ITEM3318
?LIST = (?ITEM3318 . ?REST3319);
?ITEM = ?ITEM3323
?LIST = (?X3320 ?ITEM3323 . ?REST3324);
?ITEM = ?ITEM3328
?LIST = (?X3320 ?X3325 ?ITEM3328 . ?REST3329);
?ITEM = ?ITEM3333
No.
The power or "relational" or declarative programming with unification and automatic backtracking

The "length" relation is a relation between lists (the membership relation) and the natural numbers relation

\[
\begin{align*}
\left( \leftarrow \text{length } (\ ) \ 0 \right) \\
\left( \leftarrow \text{length } (?x . \ ?y) \ (1+ \ ?n) \ (\text{length } ?y \ ?n) \right) \\
> \ (\leftarrow \text{length } (a \ b \ c \ d) \ ?n) \\
?N = (1+ (1+ (1+ (1+ 0))))\); \\
\text{No}.
\end{align*}
\]

\[
\begin{align*}
> \ (\leftarrow \text{length } ?\text{list } (1+ (1+ 0))) \\
?\text{LIST} = (?X3869 \ ?X3872)\); \\
\text{No}.
\end{align*}
\]

\[
\begin{align*}
> \ (\leftarrow \text{length } ?\text{list } ?n) \\
?\text{LIST} = \text{NIL} \\
?N = 0; \\
?\text{LIST} = (?X3918) \\
?N = (1+ 0); \\
?\text{LIST} = (?X3918 \ ?X3921) \\
?N = (1+ (1+ 0)). \\
\text{No}.
\end{align*}
\]
The Zebra Puzzle

1. There are five houses in a line, each with an owner, a pet, a cigarette, a drink, and a color.

2. The Englishman lives in the red house.

3. The Spaniard owns the dog.

4. Coffee is drunk in the green house.

5. The Ukrainian drinks tea.

6. The green house is immediately to the right of the ivory house.

7. The Winston smoker owns snails.

8. Kools are smoked in the yellow house.

9. Milk is drunk in the middle house.
The Zebra Puzzle

10. The Norwegian lives in the first house on the left.
11. The man who smokes Chesterfields lives next to the man with the fox.
12. Kools are smoked in the house next to the house with the horse.
13. The Lucky Strike smoker drinks orange juice.
15. The Norwegian lives next to the blue house.

=> Who drinks water and who owns zebras?
The Zebra Puzzle

We make use of the following basic relations

\[
(- \text{ (member ?item (?item . ?rest)))}
\]

\[
(- \text{ (member ?item (?x . ?rest)) \ (member ?item ?rest))}
\]

\[
(- \text{ (nextto ?x ?y ?list) \ (iright ?x ?y ?list))}
\]

\[
(- \text{ (nextto ?x ?y ?list) \ (iright ?y ?x ?list))}
\]

\[
(- \text{ (iright ?left ?right (?left ?right . ?rest)))}
\]

\[
(- \text{ (iright ?left ?right (?x . ?rest))}
\]

\[
\text{ (iright ?left ?right ?rest))}
\]

\[
(- \text{ (= ?x ?x))}
\]
The Zebra Puzzle

We make use of the following basic relations

\[
\begin{align*}
&(\leftarrow (\text{member } ?\text{item} (\text{?item} . \text{?rest}))) \\
&(\leftarrow (\text{member } ?\text{item} (?x . \text{?rest})) (\text{member } ?\text{item} \text{?rest})) \\
&(\leftarrow (\text{nextto } ?\text{x} \text{?y } \text{?list}) (\text{iright } ?\text{x} \text{?y } \text{?list})) \\
&(\leftarrow (\text{nextto } ?\text{x} \text{?y } \text{?list}) (\text{iright } ?\text{y} \text{?x } \text{?list})) \\
&(\leftarrow (\text{iright } ?\text{left } \text{right} (?\text{left} \text{?right} . \text{?rest}))) \\
&(\leftarrow (\text{iright } ?\text{left } \text{right} (?\text{x} . \text{?rest}))
\quad (\text{iright } ?\text{left } \text{right } \text{?rest})) \\
&(\leftarrow (= ?\text{x} ?\text{x}))
\end{align*}
\]

And define a list of houses where each house is a list of the form

\[
\text{(house nationality pet cigarette drink house-color)}
\]

The constraint “The Englishmen lives in the red house” then becomes the clause

\[
(\text{member } (\text{house Englishmen } ? \text{ ? ? red}) \text{ ?list-of-houses})
\]
(← (zebra ?h ?w ?z))

;; Each house is of the form:
;; (house nationality pet cigarette drink house-color)
(= ?h ((house norwegian ? ? ? ?) 1,10

  (?)
(member (house englishman ? ? ? red) ?h) 2
(member (house spaniard dog ? ? ?) ?h) 3
(member (house ? ? ? coffee green) ?h) 4
(member (house ukrainian ? ? tea ?) ?h) 5

(member (house ? snails winston ? ?) ?h) 7
(member (house ? ? kools ? yellow) ?h) 8
(nextto (house ? ? chesterfield ? ?)

  (house ? fox ? ? ?) ?h)
(nextto (house ? ? kools ? ?)? 12

  (house ? horse ? ? ?) ?h)
(member (house ? ? luckystrike orange-juice ?) ?h) 13
(member (house japanese ? ? parliaments ? ?) ?h) 14
(nextto (house norwegian ? ? ? ?)

  (house ? ? ? blue) ?h)

;; Now for the questions:
(member (house ?w ? ? water ?) ?h) 01
(member (house ?z zebra ? ? ?) ?h)) 02
The power or “relational” or declarative programming with unification and automatic backtracking

... and some limitations again

```prolog
> (?- (length ?L (1+ (1+ 0))) (member a ?L))
?L = (A ?X4057);
?L = (?Y4061 A);
No.

> (?- (member a ?L) (length ?L (1+ (1+ 0))))
?L = (A ?X4081);
?L = (?Y4085 A);[Abort]
```

Again, there is no “understanding” of the concept of length so that, after all, the logic programmer does have to worry about the flow of control...
The power or “relational” or declarative programming with unification and automatic backtracking

Note 1: in the zebra puzzle we used an identity relation as follows:

(← (= $?x $?x))

What does “=” mean now? (e.g., is it “eq” or “eql” or “equal” or …)?
The power or “relational” or declarative programming with unification and automatic backtracking

Note 1: in the zebra puzzle we used an identity relation as follows:

\[ \leftarrow (\ = \ ?x \ ?x) \]

What does “=” mean now? (e.g., is it “eq” or “eql” or “equal” or …)?

It means unification!

Prolog really is a synergy of three components:

→ a uniform database
→ a query or “testing for equal” mechanism with the power of unification
→ exhaustive search with automatic backtracking
The power or “relational” or declarative programming with unification and automatic backtracking

Note 1: in the zebra puzzle we used an identity relation as follows:

`(<- (= ?x ?x))`

What does “=” mean now? (e.g., is it “eq” or “eql” or “equal” or …)?

It means unification!

Note 2: in the zebra puzzle, there were 5 attributes for each of the five houses, so there are over 24 billion candidate solutions, far too many to test one at a time

It is the concept of unification and logic variables that makes backward chaining feasible, by making it possible to specify partial solutions.
Extending the power or “relational” or declarative beyond standard unification

By further extending the available unification power (and hence further specifying how more and more complex structures can be considered equal), the need for backtracking can be eliminated completely!

```
(?- (length ?l 4)
    (member d ?l) (member a ?l) (member c ?l) (member b ?l)
    (= ?l (a b c d)))
```

If unification would know about lists, the first line could bind ?l to some structure like \#S(LIST :length 4).

The second line could extend it to \#S(LIST :length 4  :members (d a c b))

The third line could then specify the order as in \#S(LIST :length 4 :members (d a b c) :order (a b c d))