Logic Programming in Scheme

Nils M Holm

Print and distribution: Lulu Press, Inc
http://lulu.com/nmh/

Preface

This text discusses Logic Programming in terms of purely functional and symbolic Scheme.

It is the "Scheme version" of "*Logic Programming in Symbolic LISP*". It covers exactly the same topics, but uses Scheme as its host language.

Chapter 1 introduces logic programming basics by means of the AMK logic programming system.

Chapter 2 outlines the application of the techniques from Chapter 1 to a well-known logic puzzle.

Chapter 3 describes the implementation of the AMK logic programming system. *Complete code included!*

The AMK source code can be found in the SketchyLISP section at `http://t3x.org`.

Before reading this text, you should be familiar with the basics of the Scheme programming language, including concepts like lists, functions, closures, and recursion.

This work has been greatly influenced by the book "The Reasoned Schemer" by Daniel P. Friedman, et al.

The AMK (Another Micro Kanren) logic programming system is losely based upon Oleg Kiselyov's "Sokuza Mini Kanren".

Welcome to the world of Logic Programming.
Go ahead, explore, and enjoy!

Nils M Holm, 2007

Contents

1

Introduction

1.1 Functions vs Goals

In Functional Programming, functions are combined to form programs. For example, the `append` function of Scheme concatenates some lists:

```
(append '(orange) '(juice)) => (orange juice)
```

The basic building stones of Logic Programming are called *goals*. A goal is a function that maps knowledge to knowledge:

```
(run* () (appendo '(orange) '(juice) '(orange juice))) => (())
```

An application of `run*` is called a *query*. `Run*` is the interface between Scheme and the logic programming subsystem. The result of `run*` is called the *answer* to the corresponding query.

The goal used in the above query is **append**^{0} [page 17].

An answer of the form `(())` means "yes". In the above example, this means that `(orange juice)` is indeed equal to the concatenation of `(orange)` and `(juice)`.

A goal returning a positive answer is said to *succeed*.

The goal

```
(run* () (appendo '(orange) '(juice) '(fruit salad))) => ()
```

does *not* succeed, because `(fruit salad)` cannot be constructed by appending `(orange)` and `(juice)`.

A goal that does not succeed is said to *fail.* Failure is represented by `()`.

When one or more arguments of a goal are replaced with variables, the goal attempts to *infer* the values of these variables.

Logic variables are created by the `var` function:

```
(define vq (var 'q))
```

Any argument of a goal can be a variable:

```
(run* (vq) (appendo '(orange) '(juice) vq)) => ((orange juice))
(run* (vq) (appendo '(orange) vq '(orange juice))) => ((juice))
(run* (vq) (appendo vq '(juice) '(orange juice))) => ((orange))
```

In this example, `run*` is told that we are interested in the value of *vq.* It runs the given goal and then returns the value or values of *vq* rather than just success or failure.

Goals are non-deterministic, so a query may return more than a single outcome:

```
(run* (vq) (fresh (dont-care)
             (appendo dont-care vq '(a b c d))))
=> ((a b c d) (b c d) (c d) (d) ())
```

This query returns all values which give `(a b c d)` when appended to something that is not interesting. In other words, it returns all suffixes of `(a b c d)`.

The part we do not care about is bound to the *fresh* variable *dont-care.* The `fresh` form

```
(fresh (x1 x2 ...) goal)
```

is just syntactic sugar for

```
(let ((x1 (var 'x1)) (x2 (var 'x2)) ...) goal)
```

When querying the second position of **append**^{0}, the query returns all prefixes of the given list:

```
(run* (vq) (fresh (dont-care)
             (appendo vq dont-care '(a b c d))))
=> (() (a) (a b) (a b c) (a b c d))
```

What do you think is the answer to the following query?

```
(run* (vq) (fresh (x y) (appendo x y vq)))
```

Does it have an answer at all?

Answer: The query has no answer, because there is an indefinite number of combinations that can be used to form a concatenation with an unspecified prefix and suffix. So **append**^{0} never stops generating combinations of values for x and y.

1.2 Unification

Unification is an algorithm that forms the heart of every logic programming system. The "unify" goal is written $==$. The query

```
(run* () (== x y))
```

means "unify x with y". The answer of this query depends on the values of x and y:

```
(run* (vq) (== 'pizza 'pizza))  => (())
(run* (vq) (== 'cheese 'pizza)) => ()
(run* (vq) (== vq vq))          => (())
(run* (vq) (== 'cheese vq))     => (cheese)
```

When two atoms are passed to $==$, it succeeds if the atoms are equal.

When a variable is passed to $==$, the variable is *bound* to the other argument:

```
(run* (vq) (== vq 'cheese)) => (cheese)
(run* (vq) (== 'cheese vq)) => (cheese)
```

The order of arguments does not matter.

When two variables are unified, these two variables are guaranteed to *always* bind to the same value:

```
(run* (vq) (fresh (x) (== vq x)))
```

makes *vq* and *x* bind to the same value. Binding a value to one of them at a later time automatically binds that value to both of them.

Non-atomic arguments are unified recursively by first unifying the car parts of the arguments and then unifying their cdr parts:

```
(run* (vq) (== '(x (y) z) '(x (y) z))) => (())
(run* (vq) (== '(x (y) z) '(x (X) z))) => ()
(run* (vq) (== vq '(x (y) z)))         => ((x (y) z))
```

Inference works even if variables are buried inside of lists:

```
(run* (vq) (== (list 'x vq 'z)  '(x (y) z))) => ((y))
```

Because `(y)` is the only value for *vq* that makes the goal succeed, that value is bound to *vq*.

How does this work?

- `x` is unified with `x`;
- *vq* is unified with `(y)` (binding *vq* to `(y)`);
- `z` is unified with `z`.

Each unification may expand the "knowledge" of the system by binding a variable to a value or unifying two variables.

When unifying lists, the cdr parts of the lists are unified in the context of the knowledge gained during the unification of their car parts:

```
(run* (vq) (== '(pizza fruit-salad) (list vq vq))) => ()
```

The unification cannot succeed, because first *vq* is bound to `pizza` and then the same variable is bound to `fruit-salad`.

When *vq* is unified with `pizza`, *vq* is *fresh*. A variable is fresh if it is not (yet) bound to any value.

Only fresh variables can be bound to values.

When a form is unified with a bound variable, it is unified with the *value* of that variable. Hence

```
(run* (vq) (== '(pizza fruit-salad) (list vq vq))) => ()
```

is equivalent to

```
(run* (vq) (== '(pizza fruit-salad) (list vq 'pizza))) => ()
```

The following query succeeds because no contradiction is introduced:

```
(run* (vq) (== '(pizza pizza) (list vq vq))) => (pizza)
```

First *vq* is unified with `pizza` and then *the value of vq* (which is `pizza` at this point) is unified with `pizza`.

Bound variables can still be unified with fresh variables:

```
(run* (vq) (fresh (vx)
             (== (list 'pizza vq)
                 (list  vx    vx)))) => (pizza)
```

Here *vx* is unified with `pizza` and then the fresh variable *vq* is unified with *vx*, binding *vq* and *vx* to the same value.

Again, the order of unification does not matter:

```
(run* (vq) (fresh (vx)
             (== (list vq 'pizza)
                 (list vx  vx)))) => (pizza)
```

1.3 Logic Operators

The **any** goal succeeds, if at least one of its *subgoals* succeeds:

```
(run* (vq) (any (== vq 'pizza)
                (== 'orange 'juice)
                (== 'yes 'no)))
=> (pizza)
```

In this example, one of the three subgoals succeeds and contributes to the anwer.

Because **any** succeeds only if at least one of its subgoals succeeds, it fails if no subgoals are given:

```
(run* () (any)) => ()
```

Multiple subgoals of **any** may unify the same variable with different forms, giving a non-determistic answer:

```
(run* (vq) (any (== vq 'apple)
                (== vq 'orange)
                (== vq 'banana)))
=> (apple orange banana)
```

No contradiction is introduced. *Vq* is bound to each of the three values.

The **any** goal implements the *union* of the knowledge gained by running its subgoals:

```
(run* (vq) (any fail
                (== vq 'fruit-salad)
                fail))
=> (fruit-salad)
```

It succeeds even if some of its subgoals fail. Therefore it is equivalent to the *logical or*.

Fail is a goal that always fails.

The **all** goal is a cousin of **any** that implements the *logical and*:

```
(run* (vq) (all (== vq 'apple)
                (== 'orange 'orange)
                succeed))
=> (apple)
```

Succeed is a goal that always succeeds.

All succeeds only if all of its subgoals succeed, but it does more than this.

All forms the intersection of the knowledge gathered by running its subgoals by removing any contradictions from their answers:

```
(run* (vq) (all (== vq 'apple)
                (== vq 'orange)
                (== vq 'banana))) => ()
```

This goal fails because *vq* cannot be bound to `apple`, `orange`, and `banana` at the same time.

This effect of **all** is best illustrated in combination with **any**:

```
(run* (vq) (all (any (== vq 'orange)
                     (== vq 'pizza))
                (any (== vq 'apple)
                     (== vq 'orange))))
=> (orange)
```

The first **any** binds *vq* to `orange` or `pizza` and the second one binds it to `apple` or `orange`.

All forms the intersection of this knowledge by removing the contradictions *vq*=`pizza` and *vq*=`apple`. *Vq*=`orange` is no contradiction because it occurs in both subgoals of **all**.

All fails if at least one of its subgoals fails. Therefore, it succeeds, if no goals are passed to it:

```
(run* () (all)) => (())
```

1.4 Parameterized Goals

A parameterized goal is a function returning a goal:

```
(define (conso a d p) (== (cons a d) p))
```

Applications of `conso` evaluate to a goal, so **cons**^{0} can be used to form goals in queries:

```
(run* (vq) (conso 'heads 'tails vq)) => ((heads . tails))
```

In the prose, `conso` is written **cons**^{0}. The trailing "0" of goal names is pronounced separately (e.g. "cons-oh").

Obviously, **cons**^{0} implements something that is similar to the `cons` function.

However, **cons**^{0} can do more:

```
(run* (vq) (conso 'heads vq '(heads . tails))) => (tails)
(run* (vq) (conso vq 'tails '(heads . tails))) => (heads)
```

So **conso**^{0} can be used to define two other useful goals:

```
(define (caro p a)
  (fresh (_)
    (conso a _ p)))
```

Car^0 is similar to the `car` function of Scheme and **cdr**^{0} is similar to its `cdr` function:

```
(define (cdro p d)
  (fresh (_)
    (conso _ d p)))
```

Like in PROLOG, the name _ indicates that the value bound to that variable is of no interest.

Unlike in PROLOG, however, _ is an ordinary variable without any special properties.

When the second argument of **car**^{0} and **cdr**^{0} is a variable, they resemble `car` and `cdr`:

```
(run* (vq) (caro '(x . y) vq)) => (x)
(run* (vq) (cdro '(x . y) vq)) => (y)
```

Like **cons**^{0}, **car**^{0} and **cdr**^{0} can do more than their Scheme counterparts, though:

```
(run* (vq) (caro vq 'x)) => ((x . _.0))
(run* (vq) (cdro vq 'y)) => ((_.0 . y))
```

The query

```
(run* (vq) (caro vq 'x))
```

asks: "what has a car part of `x`?" and the answer is "any pair that has a car part of `x` and a cdr part that does not matter."

Clever, isn't it?

1.5 Reification

Atoms of the form `_.n`, where n is a unique number, occur whenever an answer would otherwise contain fresh variables:

```
(run* (vq) (fresh (x y z)
             (== vq (list x y z))))
=> ((_.0 _.1 _.2))
```

In the remainder of this text, `_.n` may be spelled $\_{n}$.

$\_{0}$, $\_{1}$, etc are called *reified variables.*

The replacement of fresh variables with reified names is called *reification.* It replaces each fresh variable with a unique "item" (*res* being the latin word for "item").

1.6 Recursion

Here is a recursive Scheme predicate:

```
(define (mem? x l)
  (cond ((null? l) #f)
    ((eq? x (car l)) #t)
    (else (mem? x (cdr l)))))
```

`Mem?` tests whether l contains x:

```
(mem? 'c '(a b c d e f)) => #t
(mem? 'x '(a b c d e f)) => #f
```

In logic programming, there is no function composition. So you cannot write code like `(eq? x (car l))`.

Each argument of a goal *must* be either a datum or a variable. Only **any** and **all** have subgoals:

```
(define (memo x l)
  (fresh (a d)
    (any (all (caro l a)
              (eqo x a))
         (all (cdro l d)
              (memo x d)))))
```

Here are some observations:

- One **any** containing one or multiple **all** goals is the logic programming equivalent of `cond`.
- Each time **mem**^{0} is entered, a fresh a and d is created.
- **Mem**^{0} does not seem to check whether l is `()`.

Does **mem**^{0} work? Yes:

```
(run* () (memo 'c '(a b c d e f))) => (())
(run* () (memo 'x '(a b c d e f))) => ()
```

How does it work?

The first **all** unifies the car part of l with a. In case l=`()`, **all** fails.

Eq^0 is a synonym for `==`.

If a (which is now an alias of `(car l)`) can be unified with x, this branch of **any** succeeds.

If l is `()`, both of these goals fail:

```
(caro l a) => ()
(cdro l d) => ()
```

and so the entire **mem**^{0} fails. There is no need to test for l=`()` explicitly.

The second **all** of **mem**^{0} unifies d with the cdr part of l and then recurses.

1.7 Converting Predicates to Goals

A predicate is a function returning a truth value.

Each goal is a predicate in the sense that it either fails or succeeds.

There are four steps involved in the conversion of a predicate to a goal:

c1. Decompose function compositions.

c2. Replace functions by parameterized goals.

c3. Replace `cond` with **any** and its clauses with **all**.

c4. Remove subgoals that make the predicate fail.

`Mem` [page 11] is converted this way:

```
((eq? x (car l)) #t)
```

becomes (by c1, c2, c3)

```
(fresh (a)
  (all (caro l a)
       (eqo x a)))
```

and

```
(else (mem? x (cdr l)))
```

becomes (again by c1, c2, c3)

```
(fresh (d)
  (all (cdro l d)
       (memo x d)))
```

Finally

```
((null? l) #f)
```

is removed (by c4) and `cond` is replaced with **any** (by c3).

In the original definition of **mem**^{0} [page 11], `(fresh (a) ...)` and `(fresh (d) ...)` are combined and placed before **any**.

1.8 Converting Functions to Goals

`Member` is similar to `mem?` (and identical to the Scheme standard procedure with the same name):

```
(define (member x l)
  (cond ((null? l) #f)
    ((eq? x (car l)) l)
    (else (member x (cdr l)))))
```

Instead of returning just `#t` in case of success, it returns the first sublist of l whose car part is x:

```
(member 'orange '(apple orange banana)) => (orange banana)
```

Functions are converted to goals in the same way as predicates, but there is one additional rule:

c5. Add an additional argument to unify with the result.

Member^0 is similar to **mem**^{0}, but it has an additonal argument r for the result, and an additional goal which unifies the answer with r:

```
(define (membero x l r)
  (fresh (a d)
    (any (all (caro l a)
              (eqo x a)
              (== r l))
         (all (cdro l d)
              (membero x d r)))))
```

Like `member`, **member**^{0} can be queried to deliver the first sublist of l whose head is x:

```
(run* (vq) (membero 'orange '(apple orange banana) vq))
=> ((orange banana))
```

Member^0 even delivers *all* the sublists of l beginning with x:

```
(run* (vq) (membero 'b '(a b a b a b c) vq))
=> ((b a b a b c) (b a b c) (b c))
```

If you are only interested in the first one, take the car part of the anwer.

Member^0 can also be used to implement the identity function:

```
(run* (vq) (fresh (_)
             (membero vq '(orange juice) _)))
=> (orange juice)
```

How does this work?

The question asked here is “what should vq be unified with to make `(membero vq '(orange juice) _)` succeed?”

The **eq**^{0} in **member**^{0} unifies vq with `orange` and because vq is fresh, it succeeds.

The second case also succeeds. It binds l to `(juice)` and re-tries the goal. In this branch, vq is still fresh.

The **eq**^{0} in **member**^{0} unifies vq with `juice` and because vq is fresh, it succeeds.

The second case also succeeds. It binds l to `()` and re-tries the goal. In this branch, vq is still fresh.

`(Membero vq () _)` fails, because neither **car**^{0} nor **cdr**^{0} can succeed with l=`()`.

Any forms the union of vq=`orange` and vq=`juice`, which is the answer to the query.

1.9 COND vs ANY

Scheme's `cond` syntax tests the predicates of its clauses sequentially and returns the normal form of the expression associated with the first true predicate:

```
(cond (#t 'bread)
      (#f 'with)
      (#t 'butter)) => bread
```

Even though the clause `(#t 'butter)` also has a true predicate, the above `cond` will *never* return `butter`.

A combination of **any** and **all** can be used to form a logic programming equivalent of `cond`:

```
(run* (vq) (any (all succeed (== vq 'bread))
                (all fail    (== vq 'with))
                (all succeed (== vq 'butter))))
=> (bread butter)
```

Any replaces `cond` and **all** introduces each individual case.

Unlike `cond`, though, this construct returns the values of *all* cases that succeed.

While `cond` ignores the remaining clauses in case of success, **any** keeps trying until it runs out of subgoals.

This is the reason why **member**^{0} [page 14] returns all sublists starting with a given form:

```
(run* (vq) (membero 'b '(a b a b a b c) vq))
=> ((b a b a b c) (b a b c) (b c))
```

It works this way:

When the head of l is *not* equal to `b`, the first subgoal of **any** in **member**^{0} fails, so nothing is added to the answer.

When the head of l is equal to `b`, the first subgoal of **any** succeeds, so l is added to the answer.

In either case, the second goal is tried. It succeeds as long as l can be decomposed. It fails when the end of the list l has been reached.

When the second goal succeeds, the whole **any** is tried on the cdr part of l, which may add more sublists to the answer.

What happens when the order of cases is reversed in member^0?

```
(define (r-membero x l r)
  (fresh (a d)
    (any (all (cdro l d)
              (r-membero x d r))
         (all (caro l a)
              (eqo x a)
              (== r l)))))
```

Because **any** keeps trying until it runs out of goals, **r-member**^{0} does indeed return all matching sublists, just like **member**^{0}. However ...

```
  (run* (vq) (membero 'b '(a b a b c) vq)) => ((b a b c) (b c))
(run* (vq) (r-membero 'b '(a b a b c) vq)) => ((b c) (b a b c))
```

Because **r-member**^{0} first recurses and then checks for a matching sublist, its answer lists the last matching sublist first.

Reversing the goals of **member**^{0} makes it return its results in reverse order.

While **member**^{0} implements the identity function, **r-member**^{0} implements a function that reverses a list:

```
(run* (vq) (fresh (_)
             (r-membero vq '(ice water) _)))
=> (water ice)
```

1.10 First Class Variables

Logic variables are first class values.

When a bound logic variable is used as an argument to a goal, the value of that variable is passed to the goal:

```
(run* (vq) (fresh (x)
             (all (== x 'piece-of-cake)
                  (== vq x))))
=> (piece-of-cake)
```

When a fresh variable is used as an argument to a goal, the *variable itself* is passed to that goal:

```
(run* (vq) (fresh (x)
             (== vq x)))
=> (_.0)
```

(Because the variable x is fresh, it is reified by the interpreter *after* running the query, giving $\_{0}$.)

Variables can even be part of compound data structures:

```
(run* (vq) (fresh (x)
             (conso 'heads x vq)))
=> ((heads . _.0))
```

Unifying a variable that is part of a data structure at a later time causes the variable part of the data structure to be "filled in" belatedly:

```
(run* (vq) (fresh (x)
           (all (conso 'heads x vq)
                (== x 'tails))))
=> ((heads . tails))
```

The **append**^{0} goal makes use of this fact:

```
(define (appendo x y r)
  (any (all (== x '()) (== y r))
       (fresh (h t tr)
         (all (conso h t  x)
              (conso h tr r)
              (appendo t y tr)))))
```

Given this definition, how is the following query processed?

```
(run* (vq) (appendo '(a b) '(c d) vq)) => ((a b c d))
```

In its recursive case, **append**^{0} first decomposes x=`(a b)` into its head h=`a` and tail t=`(b)`:

```
(conso h t x)
```

The next subgoal states that the head h consed to *tr* (the **t**ail of the **r**esult) gives the result of **append**^{0}:

```
(conso h tr r)
```

Because *tr* is fresh at this point, r is bound to a structure containing a variable:

r_0 = (cons `a` tr_0)

Tr and r are called tr_0 and r_0 here, because they are the first instances of these variables.

When the goal recurses, tr_0 is passed to **append**^{0} in the place of r:

`(appendo (b) (c d) tr`$_0$`)`

Append^0 creates fresh instances of *tr* and r (called tr_1 and r_1).

At this point r_1 and tr_0 may be considered *the same variable*, so

`(conso h` tr_1 r_1`)`

results in

r_1 = tr_0 = (cons `b` tr_1)

and

r_0 = (cons `a` tr_0) = (cons `a` (cons `b` tr_1))

When **append**^{0} recurses one final time, tr_1 is passed in the place of r and the instance r_2 is created:

`(appendo () (c d) tr`$_1$`)`

Because $x=$`()`, the subgoal handling the trivial case is run, resulting in:

```
(== y r2)
```

and because r_2 and tr_1 are the same,

$r_2 = tr_1 =$ `(c d)`
$r_1 = tr_0 =$ (cons `b` tr_1) $=$ (cons `b` `(c d)`)
$r_0 =$ (cons `a` tr_0) $=$ (cons `a` (cons `b` tr_1)) $=$ (cons `a` (cons `b` `(c d)`))

1.11 First Class Goals

Like Scheme functions, goals are first class values.

The **filter**^{0} goal makes use of this fact:

```
(define (filtero p l r)
  (fresh (a d)
    (any (all (caro l a)
              (p a)
              (== a r))
         (all (cdro l d)
              (filtero p d r)))))
```

Filter^0 extracts all members with a given property from a list.

The property is described by the goal **p** which is passed as an argument to **filter**^{0}:

```
(run* (vq) (filtero pairo '(a b (c . d) e (f . g)) vq))
=> ((c . d) (f . g))
```

where **pair**^{0} is defined this way:

```
(define (pairo x)
  (fresh (_1 _2)
    (conso _1 _2 x)))
```

Because all goals are in fact parameterized goals, there is no real need to invent a new function name, though. `Lambda` works fine:

```
(run* (vq) (fresh (_1 _2)
             (filtero (lambda (x) (conso _1 _2 x))
                      '(a b (c . d) e (f . g)) vq)))
=> ((c . d) (f . g))
```

Because each application of `var` is guaranteed to yield a fresh variable and *_1* and *_2* are never referred to anyway, the anonymous goal can be simplied further:

```
(lambda (x) (conso (var '_) (var '_) x))
```

This property of `var` is so handy that `(_)` is introduced as a synonym for `(var '_)`, allowing to write

```
(lambda (x) (conso (_) (_) x))
```

1.12 Negation

The **neg** goal succeeds if its subgoal fails, and fails if its subgoal succeeds:

```
(run* () (neg fail)) => (())
(run* () (neg succeed)) => ()
```

Neg *never* contributes any knowledge:

When its subgoal succeeds, **neg** itself fails, thereby deleting all knowledge gathered so far.

When its subgoal fails, there is no knowledge to add.

However, **neg** is not as straight-forward as it seems:

```
(define (nullo x) (eqo '() x))
```

The **null**^{0} goal tests whether its argument is `()`.

What should be the answer to the question "what is not equal to `()`?"

```
(run* (vq) (neg (nullo vq)))
```

Neg^0 answers this question using a principle called the "closed world assumption", which says "what cannot be proven true must be false".

So the answer to above question is "nothing". Because the value of *vq* is not known, **neg**^{0} cannot prove that it is not equal to `()` and fails:

```
(run* (vq) (neg (nullo vq))) => ()
```

Technically, it works like this:

Vq is fresh, so **null**^{0} unifies it with `()` and succeeds. Because **null**^{0} succeeds, **neg** must fail.

This approach has its consequences:

```
(run* (vq)
  (all (any (== vq 'orange)
            (== vq 'pizza)
            (== vq 'ice-cream))
       (neg (== vq 'pizza))))
       => (orange ice-cream)
```

```
(run* (vq)
  (all (neg (== vq 'pizza))
       (any (== vq 'orange)
            (== vq 'pizza)
            (== vq 'ice-cream))))
            => ()
```

Depending on its context, **neg** has different functions.

In the righthand example, it makes the entire query fail, because the fresh variable *vq* can be unified with `pizza`.

In the lefthand example, where *vq* already has some values, it removes the association of *vq* and `pizza`.

Therefore

Negation should be used with great care.

1.13 Cutting

The **member**^{0} goal [page 14] returned all sublists whose heads matched a given form:

```
(run* (vq) (membero 'b '(a b a b c) vq)) => ((b a b c) (b c))
```

For the case that you are *really, really* only interested in the first match, there is a technique called *cutting*.

It is implemented by the **one** goal:

```
(run* (vq) (one fail
                (== vq 'apple)
                (== vq 'pie)))
=> (apple)
```

As soon as one subgoal of **one** succeeds, **one** itself succeeds immediately and "cuts off" the remaining subgoals.

The name **one** indicates that at most "**one** of its subgoals" can succeed.

Using **one**, a variant of **member**^{0} can be implemented which succeeds with the first match:

```
(define (firsto x l r)
  (fresh (a d)
    (one (all (caro l a)
              (eqo x a)
              (== r l))
         (all (cdro l d)
              (firsto x d r)))))
```

The only difference between **member**^{0} and **first**^{0} is that **first**^{0} uses **one** in the place of **any**.

First^0 cuts off the recursive case as soon as the first case succeeds:

```
(run* (vq) (firsto 'b '(a b a b c) vq)) => ((b a b c))
```

One is much more like `cond` than **any**.

However **one** suppresses *backtracking*, which is one of the most interesting properties of logic programming systems.

Here is another predicate:

```
(define (juiceo x)
  (fresh (tail next)
    (all (cdro x tail)
         (caro tail next)
         (eqo next 'juice))))
```

Juice^0 succeeds, if its argument is a list whose second element is equal to `juice`, e.g.:

```
(run* () (juiceo '(orange juice))) => (())
(run* () (juiceo '(cherry juice))) => (())
(run* () (juiceo '(apply  pie  ))) => ()
```

Given the **juice**^{0} predicate, **member**^{0} can be used to locate your favorite juice on a menu:

```
(define menu '(apple pie   orange pie   cherry pie
               apple juice orange juice cherry juice))

(run* (vq) (all (membero 'orange menu vq)
                (juiceo vq)))
=> ((orange juice cherry juice))
```

When **member**^{0} finds the sublist starting with the `orange` right before `pie`, **juice**^{0} fails and backtracking is initiated.

Member^0 then locates the next occurence of `orange` and this time **juice**^{0} succeeds.

Using **first**^{0} suppresses backtracking and so your favorite juice is never found:

```
(run* (vq) (all (firsto 'orange menu vq)
                (juiceo vq)))
=> ()
```

Therefore

Cutting should be used with great care.

2

Application

The *Zebra Puzzle* is a well-known logic puzzle.

It is defined as follows:

1. Five persons of different nationality live in five houses in a row. The houses are painted in different colors. The persons enjoy different drinks and brands of cigarettes. All persons own different pets.
2. The Englishman lives in the red house.
3. The Spaniard owns a dog.
4. Coffee is drunk in the green house.
5. The Ukrainian drinks tea.
6. The green house is directly to the right of the ivory house.
7. The Old Gold smoker owns snails.
8. Kools are being smoked in the yellow house.
9. Milk is drunk in the middle house.
10. The Norwegian lives in the first house on the left.
11. The Chesterfield smoker lives next to the fox owner.
12. Kools are smoked in the house next to the house where the horse is kept.
13. The Lucky Strike smoker drinks orange juice.
14. The Japanese smokes Parliaments.
15. The Norwegian lives next to the blue house.

Who owns the zebra?

To solve the puzzle, two questions must be answered:

1. how to represent the data;
2. how to add facts.

There are five houses and five attributes are linked to each house, so the row of houses can be represented by a five-element list of records. Each record is a 5-tuple holding the given attributes:

(*nation cigarette drink pet color*)

Known facts are represented by symbols and unknown ones by variables.

The fact "the Spaniard owns the dog" would look like this:

```
(list 'spaniard (var 'cigarette) (var 'drink) 'dog (var 'color))
```

Of course, inventing new variable names for each unknown attribute is awkward, so we make use of the anonymous variable (_):

```
(list 'spaniard (_) (_) 'dog (_))
```

The application of additional facts is explained by means of a simpler variant of the puzzle with only two attributes and two houses:

```
(list (list (var 'person1) (var 'drink1))
      (list (var 'person2) (var 'drink2)))
```

1. In one house lives a Swede.
2. In one house lives a beer drinker.
3. In one house lives a Japanese who drinks wine.
4. The beer drinker lives in the left house.

Applying the first fact yields the following options (variables are in lower case, known facts in upper case):

```
( ((Swede drink1)   (person1 drink2))
  ((person1 drink1) (Swede drink2)  ) )
```

which means that the Swede (whose drink is unknown) can live in the first or in the second house.

Adding the second fact multiplies the options:

```
( ((Swede Beer)     (person1 drink1))
  ((Swede drink1)   (person1 Beer)  )
  ((person1 Beer)   (Swede drink1)  )
  ((person1 drink1) (Swede Beer)    ) )
```

The key to the application of facts is unification. The fact

```
(list 'Swede (_))
```

can be unified with any fresh variable like *h1* or *h2*. The variables *h1* and *h2* represent the two houses.

```
(fresh (h1)
  (run* (h1) (== (list 'Swede (_)) h1)))
=> ((swede _.0))
```

To create *all* possible outcomes, each fact must be applied to *each* house:

```
(fresh (h)
  (run* (h) (fresh (h1 h2)
              (all (== h (list h1 h2))
                   (any (== h1 (list 'Swede (_)))
                        (== h2 (list 'Swede (_))))))))
=> (((swede _.0) _.1)
    (_.0 (swede _.1)))
```

Remember: reified variables like $\_{0}$ and $\_{1}$ denote something that is not known and/or of no interest.

In the above result, $\_{0}$ represents an unknown drink in the first outcome and an unknown house in the second one. $\_{1}$ represents an unknown house in the first outcome and an unknown drink in the second.

Each fact has be unified with all outcomes produced by the applications of the previous facts.

A goal which automatically unifies a fact with all outcomes found so far would be helpful. In fact such a goal has been defined earlier in this text.

Mem^0 [page 11] tries to unify a given form with each member of a list. Replace "form" with "fact" and "list" with "outcomes", and here we go:

```
(fresh (h)
  (run* (h) (all (== h (list (_) (_)))
                 (memo (list 'Swede (_)) h)
                 (memo (list (_) 'Beer) h))))
=> (((swede beer) _.0)
    ((swede _.0) (_.1 beer))
    ((_.0 beer) (swede _.1))
    (_.0 (swede beer)))
```

At this point the query is still *underspecified*; the known facts are not sufficient to tell where the Swede lives or whether he drinks beer or not.

By adding the third fact, some outcomes are eliminated:

```
(fresh (h)
  (run* (h) (all (== h (list (_) (_)))
                 (memo (list 'Swede (_)) h)
                 (memo (list (_) 'Beer) h)
                 (memo (list 'Japanese 'Wine) h))))
=> (((swede beer) (japanese wine))
    ((japanese wine) (swede beer)))
```

The query is still underspecified, but because the third fact contradicts the assumption that the other person drinks beer, we now know that the Swede drinks beer.

To add the final fact, another goal is needed. **Left**^{0} checks whether x is directly on the left of y in the list l:

```
(define (lefto x y l)
  (fresh (h t ht)
    (any (all (caro l h)
              (cdro l t)
              (caro t ht)  ; ht = head of tail
              (== h x)
              (== ht y))
         (all (cdro l t)
              (lefto x y t)))))
```

Using **left**^{0}, we can add the fact that the beer drinker lives in the lefthand house. The puzzle is thereby solved:

```
(fresh (h)
  (run* (h) (all (== h (list (_) (_)))
                 (memo (list 'Swede (_)) h)
                 (memo (list (_) 'Beer) h)
                 (memo (list 'Japanese 'Wine) h)
                 (lefto (list (_) 'Beer) (_) h))))
=> (((swede beer) (japanese wine)))
```

To solve the Zebra Puzzle, only one additional predicate is needed. It has to express that x is *next to* y.

X is next to y, if x is on the left of y or y is on the left of x, so:

```
(define (nexto x y l)
  (any (lefto x y l)
       (lefto y x l)))
```

Predicates which state that a house is at a specific position in the row are not required, because houses can be placed directly in the initial record:

```
(list (list 'norwegian (_) (_) (_) (_))
      (_)
      (list (_) (_) 'milk (_) (_))
      (_)
      (_))
```

The solution to the Zebra Puzzle follows on the next page.

Note that the Zebra puzzle is in fact underspecified. The drink of the Norwegian is not known. In case you prefer a fully specified query, you may comment out the following goal in the `zebra` program:

```
(memo (list (_) (_) 'water (_) (_)) h)
```

It adds the additional information that one of the persons drinks water. This information is not revealed in the original puzzle, though.

```
(define (zebra)
  (fresh (h)
    (run* (h)
      (all
        (== h (list (list 'norwegian (_) (_) (_) (_))
                    (_)
                    (list (_) (_) 'milk (_) (_))
                    (_)
                    (_)))
        (memo (list 'englishman (_) (_) (_) 'red) h)
        (lefto (list (_) (_) (_) (_) 'green)
               (list (_) (_) (_) (_) 'ivory) h)
        (nexto (list 'norwegian (_) (_) (_) (_))
               (list (_) (_) (_) (_) 'blue) h)
        (memo (list (_) 'kools (_) (_) 'yellow) h)
        (memo (list 'spaniard (_) (_) 'dog (_)) h)
        (memo (list (_) (_) 'coffee (_) 'green) h)
        (memo (list 'ukrainian (_) 'tea (_) (_)) h)
        (memo (list (_) 'luckystrikes 'orangejuice (_) (_)) h)
        (memo (list 'japanese 'parliaments (_) (_) (_)) h)
        (memo (list (_) 'oldgolds (_) 'snails (_)) h)
        (nexto (list (_) (_) (_) 'horse (_))
               (list (_) 'kools (_) (_) (_)) h)
        (nexto (list (_) (_) (_) 'fox (_))
               (list (_) 'chesterfields (_) (_) (_)) h)
;        (memo (list (_) (_) 'water (_) (_)) h)
        (memo (list (_) (_) (_) 'zebra (_)) h)))))
```

Here is the result of the program:

```
(zebra)
=> (((norwegian kools _.0 fox yellow)
    (ukrainian chesterfields tea horse blue)
    (englishman oldgolds milk snails red)
    (japanese parliaments coffee zebra green)
    (spaniard luckystrikes orangejuice dog ivory)))
```

3

Implementation

The complete implementation is written in purely functional Scheme.

3.1 Basics

These are the **fail** and **succeed** goals:

```
(define (fail x)
  '())

(define (succeed x)
  (list x))
```

`Var` creates a logic variable and `var?` checks whether an object is a logic variable. Logic variables are represented by forms like `(? . x)` where x is the name of the variable.

```
(define (var x)
  (cons '? x))

(define (var? x)
  (and (pair? x)
       (eq? (car x) '?)))
```

Empty-s represents ignorance:

```
(define empty-s '())
```

Knowledge is represented by *substitutions*. Substitutions are implemented using association lists. *Empty-s* is an empty substitution.

`Ext-s` adds the association of the variable x with the value v to the substitution s.

```
(define (ext-s x v s)
  (cons (cons x v) s))
```

`Walk` looks up the value of x in the substitution s:

```
(walk vx '((vx . bread))) => bread
```

(*Vx* denotes the logic variable x, i.e. the form `(? . x)`.)

`Walk` may look like `assoc` or `assq`, but it does more:

```
(define (walk x s)
  (cond ((not (var? x)) x)
    (else (let ((v (assq x s)))
            (if v (walk (cdr v) s)
                  x)))))
```

When the value associated with a variable is another variable, `walk` looks up the other variable, thereby tracking chains of variables:

```
(walk vx '((vx . vy) (vz . sushi) (vy . vz))) => sushi
```

This is how the unification of variables is implemented.

When the variable passed to `walk` is fresh or a fresh variable is found while tracking a chain of variables, the fresh variable is returned:

```
(walk vx empty-s) => vx
```

This is why fresh variables are first-class objects.

Everything that is not a pair is an atom:

```
(define (atom? x)
  (not (pair? x)))
```

`Unify` is the heart of AMK. It unifies x with y, looking up values of variables in s.

```
(define (unify x y s)
  (let ((x (walk x s))
        (y (walk y s)))
    (cond
      ((eqv? x y) s)
      ((var? x) (ext-s x y s))
      ((var? y) (ext-s y x s))
      ((or (atom? x) (atom? y)) #f)
      (else (let ((s (unify (car x) (car y) s)))
              (and s (unify (cdr x) (cdr y) s)))))))
```

Upon success `unify` returns s or an extension of s with new substitutions added.

In case of failure `unify` returns `#f`.

3.2 Goals

This is the **==** goal. **==** is like `unify`, but it succeeds or fails rather than returning a substitution or error flag.

```
(define (== x y)
  (lambda (s)
    (let ((s2 (unify x y s)))
      (if s2 (succeed s2)
             (fail s)))))
```

Note that **==** returns a procedure that must be applied to a substitution to let the unification take place:

```
(== vq 'orange-juice) => #<procedure (s)>
((== vq 'orange-juice) empty-s) => (((vq . orange-juice)))
```

Also note that when **==** succeeds, it adds another list around the resulting substitution.

Here is a helper function of the **any** goal:

```
(define (any* . g*)
  (lambda (s)
    (letrec
```

```
    ((try
       (lambda g*
         (cond ((null? g*) (fail s))
           (else (append ((car g*) s)
                         (apply try (cdr g*))))))))
    (apply try g*))))
```

It forms a list of substitutions by applying each member of the list of goals g^* to the given knowledge s and appending the results:

```
((any* (== vq 'ice) (== vq 'cream)) empty-s)
=> (((vq . ice)) ((vq . cream)))
```

Any* creates a *list of substitutions*. Each individual substitution is free of conflicting associations.

Any is the only goal that may produce multiple substitutions.

Any itself has to be implemented as syntax:

```
(define-syntax any
  (syntax-rules ()
    ((_) fail)
    ((_ g ...)
       (any* (lambda (s) (g s)) ...))))
```

The reason is that Scheme evaluates expressions eagerly. When a recursive goal (like **mem**^{0} on page 11) is evaluated, recursion could occur too early, resulting in indefinite evaluation.

If **any** was a function, recursion would occur *before* **any** was applied. To postpone the application of its subgoals **any**, eta expands them, so that

```
(any (g1) (g2) ...)
```

becomes

```
(any* (lambda (s) ((g1) s))
      (lambda (s) ((g2) s))
      ...)
```

Here is **all**:

```
(define (all . g*)
  (lambda (s)
    (letrec
      ((try
         (lambda (g* s*)
           (cond ((null? g*) s*)
             (else (try (cdr g*)
                        (apply append
                          (map (car g*) s*))))))))
      (try g* (succeed s)))))
```

All applies all its subgoals to the knowledge $s*$.

Because some of its subgoals may be applications of **any**, **all** uses `map` to map over multiple substitutions.

Each application of a subgoal eliminates all contradictions with that goal and the remaining substitutions are appended to form a new one.

The application of the first subgoal to $s*$ results in a new substitution $s*_1$. The next subgoal of **all** is mapped over $s*_1$, giving $s*_2$, etc.

Each subgoal is applied to the conjunction of the subgoals applied to far.

The subgoals of **all** need no protection by eta expansion, because all of them have to be evaluated anyway.

One is similar to **any**, but instead of appending substitutions, it returns s as soon as one of its subgoals succeeds:

```
(define failed? null?)

(define (one* . g*)
  (lambda (s)
    (letrec
      ((try
         (lambda g*
           (cond ((null? g*) (fail s))
             (else (let ((out ((car g*) s)))
                     (cond ((failed? out)
                             (apply try (cdr g*)))
                           (else out))))))))
      (apply try g*))))
```

For the same reasons as **any**, **one** has to protect its goals using eta expansion:

```
(define-syntax one
  (syntax-rules ()
    ((_) fail)
    ((_ g ...)
       (one* (lambda (s) (g s)) ...))))
```

Here is the **neg** goal. Its implementation is more straight-forward than its application:

```
(define (neg g)
  (lambda (s)
    (let ((out (g s)))
      (cond ((failed? out) (succeed s))
        (else (fail s))))))
```

`Fresh` is used to create fresh logic variables:

```
(fresh (a b c) (list a b c)) => ((? . a) (? . b) (? . c))
```

It is merely some syntactic sugar for `let`:

```
(define-syntax fresh
  (syntax-rules ()
    ((_ () g)
       (let () g))
    ((_ (v ...) g)
       (let ((v (var 'v)) ...) g))))
```

3.3 Interface

`Occurs?` and `circular?` are helper functions that will be used by `walk*`, which is explained right after them.

`Occurs?` checks whether the symbol or variable x occurs in the form y. Like `walk`, `occurs?` tracks variables. Values of variables are looked up in s.

```
(define (occurs? x y s)
  (let ((v (walk y s)))
    (cond
      ((var? y) (eq? x y))
      ((var? v) (eq? x v))
      ((atom? v) #f)
      (else (or (occurs? x (car v) s)
                (occurs? x (cdr v) s))))))
```

A value of a variable that contains references to that variable is called *circular*:

```
((== vq (list vq)) empty-s) => '(((vq . (vq))))
```

A circular answer is not valid, because it is self-referential.

`Circular?` checks whether the value of a variable is circular:

```
(define (circular? x s)
  (let ((v (walk x s)))
    (cond ((eq? x v) #f)
      (else (occurs? x (walk x s) s)))))
```

`Walk*` is like `walk`: it turns a variable x into a value v. In addition it replaces all variables found in v with their values.

`Walk*` brings the answers generated by the logic extensions into a comprehensible form:

```
((all (== vq (list vx)) (== vx 'foo)) empty-s)
=> (((vx . foo) (vq . (vx))))
(walk* vq '((vx . foo)  (vq . (vx)))
=> (foo)
```

When `walk*` encounters a fresh variable, it leaves it in the result.

When the variable to be `walk*`ed is bound to a circular value, `walk*` returns `_bottom_`.

```
(define _BOTTOM_ (var 'bottom))

(define (walk* x s)
  (letrec
```

```
    ((w* (lambda (x s)
           (let ((x (walk x s)))
             (cond
               ((var? x) x)
               ((atom? x) x)
               (else (cons (w* (car x) s)
                           (w* (cdr x) s))))))))
    (cond ((circular? x s) _BOTTOM_)
      ((eq? x (walk x s)) empty-s)
      (else (w* x s)))))
```

`Reify-name` generates a reified name.

```
(define (reify-name n)
  (string->symbol
    (string-append "_." (number->string n))))
```

`Reify` creates a substitution in which each fresh variable contained in the form v is associated with a unique reified name:

`(reify (list` *vx vy vz*`)) => ((`*vz* `. _.2) (`*vy* `. _.1) (`*vx* `. _.0))`

The value v that is passed to `reify` must have been `walk*`ed before.

```
(define (reify v)
  (letrec
    ((reify-s
       (lambda (v s)
         (let ((v (walk v s)))
           (cond ((var? v)
               (ext-s v (reify-name (length s)) s))
             ((atom? v) s)
             (else (reify-s (cdr v)
                     (reify-s (car v) s))))))))
    (reify-s v empty-s)))
```

`Preserve-bottom` implements bottom preservation. No surprise here.

Explanation: The term *bottom* is used in mathematics to denote an undefined result, like a diverging function. *Bottom preservation* is a principle that says that any form that contains a bottom element is itself equal to bottom.

When an answer contains the symbol `_bottom_`, the answer has a circular structure and the query that resulted in that answer should fail.

```
(define (preserve-bottom s)
  (if (occurs? _BOTTOM_ s s) '() s))
```

`Run` is a helper function of `run*`, which forms the primary interface for submitting queries to AMK.

```
(define (run x g)
  (preserve-bottom
    (map (lambda (s)
           (walk* x (append s (reify (walk* x s)))))
         (g empty-s))))
```

X may be a logic variable or `()`.

When x is a variable, `run` returns the value or values of that variable. When x=`()`, `run` returns `(())` or `()`.

When a query fails, `run` returns `()`.

`Run` runs the goal g and then `walk*`s each substitution of the answer.

It also reifies the fresh variables contained in each substitution.

`Run*` is just some syntactic sugar on top of `run`:

```
(define-syntax run*
  (syntax-rules ()
    ((_ () goal) (run #f goal))
    ((_ (v) goal) (run v goal))))
```

3.4 Anonymous Variables

Here is the _ function:

```
(define (_) (var '_))
```

It exploits the fact that logic variables are created by `cons` and compared using `eq?`:

```
(eq? (var '_) (var '_)) => #f
```

Because `cons` always creates a fresh pair, two logic variables cannot be identical, even if they share the same name.

Index