Brian Cullinan

Principles of Languages

Assignment 4 - Logic is a fickle master

Bjc76

Project 3 - The objective of this project was to implement a prolog interpreter in scheme.

Part 2:

I accomplished the objective by writing a ridiculous amount of code to make up for not being able to handle the unify function properly. I am pretty sure that if any tricky stuff is done, my code will just die with lots of errors. It is not elegant at all, but it works.

**prove:** works by either finding a predicate right off the bat and evaluating the variables, or recursively searching for each predicate within the function.

**Deep-replace:** searches for variable names and replaces it with the values.

**Output:**

((loves john mary)

(rich katy)

(rich jane)

(near katy mall)

(near jane landfill)

(man tim)

(man hank)

(food quiche)

(hungry hank)

(hungry katy)

((loves ?x3246 ?y3247) :- (man ?x3246) (woman ?y3247))

((woman ?z3248) :- (likes ?z3248 shopping) (eats ?z3248 quiche))

((likes ?x3249 shopping) :- (rich ?x3249) (near ?x3249 mall))

((eats ?x3250 ?y3251) :- (hungry ?x3250) (food ?y3251)))

> (prove '(loves tim ?x))

((?x katy))

**Code:**

**; check a complex list and search for an item in it**

**(define deep-member?**

**(lambda (needle haystack)**

**; if the hackstack is empty we know for sure the answer is false**

**(if (null? haystack)**

**#f**

**(if (list? (car haystack))**

**; check the car list then check the cdr list either one can return true**

**(or (deep-member? needle (car haystack)) (deep-member? needle (cdr haystack)))**

**; convert the symbol to a string and compare it**

**;(if (eq? needle (car haystack))**

**(if (string=? (convert needle) (convert (car haystack)))**

**#t**

**; keep checking the cdr of the list**

**(deep-member? needle (cdr haystack))**

**)**

**)**

**)**

**)**

**)**

**; check a complex list and search for an item in it and replace it**

**(define deep-replace**

**(lambda (replace needle haystack)**

**; if the hackstack is empty we know for sure the answer is false**

**(if (null? haystack)**

**'()**

**(if (list? (car haystack))**

**; check the car list then check the cdr list either one can return true**

**(or (deep-replace replace needle (car haystack)) (deep-replace replace needle (cdr haystack)))**

**; convert the symbol to a string and compare it**

**;(if (eq? needle (car haystack))**

**(if (string=? (convert needle) (convert (car haystack)))**

**(cons replace (deep-replace replace needle (cdr haystack)))**

**; keep checking the cdr of the list**

**(cons (car haystack) (deep-replace replace needle (cdr haystack)))**

**)**

**)**

**)**

**)**

**)**

**;need help converting numbers to string and symbols to strings**

**(define convert**

**(lambda (x)**

**; already a string just return**

**(if (string? x)**

**x**

**;convert symbol**

**(if (symbol? x)**

**(symbol->string x)**

**;convert number**

**(if (number? x)**

**(number->string x)**

**;weird type, just return false**

**'"E"**

**)**

**)**

**)**

**)**

**)**

**;;; UNIFICATION is the heart of the prolog system. The essential task is to find a consistent set of bindings**

**;;; between two clauses, which (for lack of better terms) we call the GOAL and the MATCHTO. Typically, the GOAL**

**;;; the thing you're trying to prove at the moment, and the MATCHTO is the clause in the fact base that you are**

**;; currently attempting to prove the goal with. Of course, MATCHTO can be either a rule or a fact. In the former case,**

**;; if I get a successful unification between the GOAL and the head of the rule (used as MATCHTO), then I need to go on**

**;; and recursively prove the antecedent...with any forward bindings made during unification. If the latter case, life**

**;; is simple: I just return the bindings used to match the fact to my caller (in the appropriate format).**

**;;; PLEASE NOTE: Because #f (false) also equates to "empty list", its not a very reliable return value! So this function**

**;; returns #t to indicate failure. Success, of course, is indicated by return of the proper bindings (if any).**

**;; the UNIFY function is the matcher. It takes a predicate, possibly with variables, to be matched; and a**

**;; fact or head of rule to be matched to. It tries to unify them; if success, it returns a package**

**;; ( (local bindings) (forward bindings)). The former are definite bindings of variables in the goal**

**;; to literals in the matchto. The latter are bindings of variables in the matchto to either literals or**

**;; variables in the goal. The latter need to be propagated to the consequent of the rule (if matchto is**

**;; the head of a rule) before the antecedent is (recursively) proved during the proof process.**

**;; so if the goal is: (loves harry ?x)**

**;; and we matchto (loves harry cindy) we get back (((?x cindy)) ())**

**;; but if we matchto (loves ?z ?q) we get back ( () ((?q ?x) (?z harry))**

**;; if the match fails return #t to indicate failure.**

**(define unify (lambda (goal matchto)**

**; (display matchto)**

**; (display "\n")**

**(unifyhelp goal matchto '( () () ))))**

**;; the meat of unification. Wrapper just passes emtpy bind list to start things off.**

**(define unifyhelp**

**(lambda (goal matchto binds)**

**(cond ((and (null? goal) (null? matchto)) binds ) ;; both empty. We're done!**

**((or (null? goal) (null? matchto)) #t) ;; one but not other empty -> fail**

**((eq? (car goal) (car matchto)) ;; elements match, call with cdrs**

**(unifyhelp (cdr goal) (cdr matchto) binds))**

**;; if variable in target, put checkto see its unbound so far, if so add to forward binds**

**((isvar? (car matchto))**

**(unifyhelp (cdr goal)**

**(deep-replace (car matchto) (car goal) (cdr matchto))**

**(list (car binds) (cons (list (car matchto) (car goal)) (cadr binds)))))**

**;; if var in goal matched to literal in target, check to see that it's unbound, add to**

**;; local bindings.**

**((isvar? (car goal)) ;; var in goal matched to literal in matchto**

**(unifyhelp (deep-replace (car goal) (car matchto) (cdr goal))**

**(cdr matchto)**

**(list (cons (list (car goal) (car matchto)) (car binds)) (cadr binds))))**

**(else #t) ;; if nothing matched, fail!**

**)))**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;; MAKING VARIABLES IN THE FACT BASE UNIQUE**

**;;;;;;;;;;;;**

**;; OVERVIEW: It is a given that variables in the fact base are independent, i.e., when you see ?x in one clause, it's**

**;; different from the ?x that appears in another clause. But the fact that these two variables LOOK the same can be**

**;; a huge pain in the butt in the unification process. To avoid huge headaches, its best to just make all variables**

**;; in the fact base unique right from the start! The UNIQUE function (aided by its helpers) below does this. Simply**

**;; put, it takes every variable it finds in an input clause, and replaces it with a unique, newly-generated variable.**

**;; Of course, appearances of the same variable within the same clause must be replaced by the same new unique variable**

**;; name! This makes writing this function tricky; it has to carry the bindings that it's made so far along with it**

**;; as it goes, so it can do the right thing when it encounters a variable later.**

**;; UNIQUE is a function that just replaces each var in a list with a unique version of the var. To avoid**

**;; name confusion during unification. So (unique '((loves ?x ?y ?x) :- (bites ?x ?y)) yields something like**

**;; (loves ?x323 ?y432 ?x323) :- (bites ?x323 ?y432).**

**;; Several helper functions break this multiple recursion down into nice chunks. Uses gensym to create new names.**

**;; Should be called during Load-Facts to immediately clean up the fact base as it is loaded from the file!!**

**(define unique**

**(lambda (alist)**

**(unique-help alist '())))**

**;; takes a list of preds to work on, plus bindings so far. Returns the the fixed list.**

**(define unique-help**

**(lambda (alist binds)**

**(cond ((null? alist) ())**

**((list? (car alist)) ; we have a list of preds to replace in! Do first one, then pass binds to rest**

**(let\* ((subfirst (unique-one (car alist) binds)))**

**(cons (car subfirst) (unique-help (cdr alist) (cadr subfirst)))))**

**(else ; its the :- symbol. Just cons it on when returning.**

**(cons (car alist) (unique-help (cdr alist) binds))))))**

**;; Uniquifies vars in a single predicate. So (loves ?x ?y ?x) becomes (loves ?x23 ?y56 ?x23. Returns**

**(define unique-one**

**(lambda (apred binds)**

**(cond ((null? apred) (list '() binds))**

**((assoc (car apred) binds) ; its a var that's been bound so far. replace with bound val**

**(let ((therest (unique-one (cdr apred) binds)))**

**(list (cons (cadr (assoc (car apred) binds)) (car therest)) (cadr therest))))**

**((isvar? (car apred)) ; its a new unbound variable**

**(let\* ((newsym (gensym (car apred)))**

**(therest (unique-one (cdr apred) (cons (list (car apred) newsym) binds))))**

**(list (cons newsym (car therest)) (cadr therest))))**

**(else ; its some non-var. Just cons it one unchanged**

**(let ((therest (unique-one (cdr apred) binds)))**

**(list (cons (car apred) (car therest)) (cadr therest)))))))**

**;; isvar? is a helper fn. that returns true if the argument starts with the character '?'**

**(define isvar?**

**(lambda (symbol)**

**(char=? (car (string->list (symbol->string symbol))) #\?)**

**))**

**;; load-file is a simple function that loads the definitions from a file**

**;; It recursives calls itself, reading one line at a time; on the recursive return, it cons'es**

**;; all of the lines together in a giant list, which it returns to caller.**

**( define load-file**

**( lambda ( port )**

**( let ( ( nextrec ( read port ) ) )**

**( cond**

**( ( eof-object? nextrec ) '() ) ;; If I've read off the end, return empty list**

**( else**

**( let\* ( ( nascent-db ( load-file port ) ) ) ;; Recursive call to finish reading file**

**;; Now add the line read at this level to growing list**

**( cons nextrec nascent-db ) ) ) ) ) ) )**

**(define factbase '())**

**(define load-facts**

**(lambda (file)**

**(set! factbase (load-helper (load-file (open-input-file file))))**

**(display "")**

**))**

**(define load-helper**

**(lambda (alist)**

**(if (null? alist)**

**()**

**(cons (unique (car alist)) (load-helper (cdr alist)))**

**)))**

**(define prove**

**(lambda args**

**(let ((result (prove-helper args factbase)))**

**(if (equal? result #t)**

**"fail"**

**(car result)**

**))))**

**(define prove-helper**

**(lambda (args facts)**

**;(display args)**

**;(display "\n")**

**; match with each fact**

**(if (null? facts)**

**#t**

**(if (list? (caar args))**

**(prove-unifier (car args) '(() ()))**

**(if (list? (caar facts))**

**; match car of element**

**(let ((result (unify (car args) (caar facts))))**

**;(display (unifyhelp (caddar facts) result)**

**(if (equal? result #t)**

**(prove-helper args (cdr facts))**

**(let ((result2 (prove-unifier (cddar facts) result)))**

**;(display result2)**

**(if (equal? result2 #t)**

**#t**

**(list (merge-result (car result) (car result2)) (merge-result (cadr result) (cadr result2)))**

**))))**

**(let ((result (unify (car args) (car facts))))**

**(if (equal? result #t)**

**(prove-helper args (cdr facts))**

**result**

**)))))))**

**(define prove-unifier**

**(lambda (args result)**

**;(display result)**

**;(display "\n")**

**(if (null? args)**

**result**

**(letrec ((args2 (deep-replace-all (cadr result) args))**

**(result2 (prove-helper `(,(car args2)) factbase)))**

**;(display combination)**

**;(display "\n")**

**(if (equal? result2 #t)**

**#t**

**(prove-unifier (cdr args2) (list (merge-result (car result) (car result2)) (merge-result (cadr result) (cadr result2))))**

**)))))**

**(define merge-result**

**(lambda (result1 result2)**

**; make sure it isn't already trying to be set**

**(if (null? result1)**

**result2**

**(merge-result (cdr result1) (cons (car result1) (remove-other (caar result1) result2)))**

**)))**

**(define remove-other**

**(lambda (needle alist)**

**(if (null? alist)**

**'()**

**(if (equal? needle (caar alist))**

**(cdr alist)**

**(cons (car alist) (remove-other needle (cdr alist)))**

**))))**

**(define deep-replace-all**

**(lambda (result args)**

**(if (null? args)**

**'()**

**(cons (deep-replace-all-helper result (car args)) (deep-replace-all result (cdr args)))**

**)))**

**(define deep-replace-all-helper**

**(lambda (result arg)**

**(if (null? result)**

**arg**

**(if (null? (cdr result))**

**(deep-replace (cadar result) (caar result) arg)**

**(deep-replace-all-helper (cdr result) (deep-replace (cadar result) (caar result) arg))**

**))))**

**(load-facts "facts.txt")**

**factbase**

**(prove '(loves tim ?x))**

**;(prove '(woman ?x))**

**;(prove '(loves ?x ?t))**

**;(prove '(loves hank ?e))**