```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; LOGIC PROGRAMMING INTERPRETER
;;;
;;;  from George Lugar and William Stubblefield
;;;    Artificial Intelligence and the Design of Expert Systems
;;;    Benjamin/Cummings, 1989
;;;  Additional examples by William Bricken
;;;
;;;  Takes a database of horn clauses and unifies variables
;;;    to answer queries
;;;
;;;  Streams simulated by lists, does not implement generators.

(defun first-stream (stream)  (car stream))
(defun rest-stream (stream)  (cdr stream))
(defun cons-stream (exp stream)  (cons exp stream))
(defun combine-streams (s1 s2)  (append s1 s2))
(defun empty-streamp (stream)  (null stream))
(defun make-empty-stream ()  nil)

(defvar *ASSERTIONS* nil)

;;;top level interpreter loop
(defun logic-shell ()
  (print 'logic>)             ;the prompt
  (let* ((goal-string (read-line))
         (goal (read-from-string goal-string)))
    (cond ((member goal '(QUIT EXIT BYE)) '*EXITING-LOGIC-SHELL*)
          (T (let ((soln (solve goal nil)))
;(pprint (list 'SOLUTION soln))
             (print-solutions goal soln)
             (terpri)
             (logic-shell))))))

;;;SOLVE takes a goal and a set of substitutions and
;;; returns a stream of augmented substitutions which satisfy the goal
(defun solve (goal subs)
  (cond ((conjunctive-goalp goal)
         (filter-thru-conj-goals
          (body goal) (cons-stream subs (make-empty-stream))))
        (T (infer goal subs *ASSERTIONS*))))

;;;FILTER-THRU-CONJ-GOALS takes a list of goals and a stream of
;;;  substitutions and filters through the goals, eliminating failures
(defun filter-thru-conj-goals (goals subst-stream)
  (cond ((null goals) subst-stream)
        ((filter-thru-conj-goals
          (rest goals)
          (filter-thru-goal (first goals) subst-stream)))))

(defun filter-thru-goal (goal subst-stream)
  ;(declare (notinline filter-thru-goal))
  (cond ((empty-streamp subst-stream)
```

```
         (make-empty-stream))
        (T (combine-streams
            (solve goal (first-stream subst-stream))
            (filter-thru-goal goal (rest-stream subst-stream))))))

;;;INFER attempts to infer a goal from a knowledge-base
(defun infer (goal subs kb)
  ;(declare (notinline infer))
  (cond ((null kb) (make-empty-stream))
        (T (let ((assertion (rename-variables (car kb))))
             (combine-streams
              (cond ((rulep assertion)
                     (use-rule goal assertion subs))
                    (T (use-fact goal assertion subs)))
              (infer goal subs (rest kb)))))))

;;;USE-RULE attempts to infer a goal from a rule by backward chaining
(defun use-rule (goal rule subs)
  (let ((match (unify goal (conclusion rule) subs)))
    (cond ((equal match '*FAILED*) (make-empty-stream))
          (T (solve (premise rule) match)))))

;;;USE-FACT attempts to match a goal with a fact
(defun use-fact (goal pat subs)
  (let ((match (unify goal pat subs)))
    (cond ((equal match '*FAILED*) (make-empty-stream))
          (T
;(pprint (list 'MATCH= match))
           (cons-stream match (make-empty-stream))))))


;;;print utility
(defun print-solutions (goal sub-stream)
  ;(declare (notinline print-solutions))
  (cond ((empty-streamp sub-stream) (print '*DONE*))
        (T (print (apply-substitutions goal (first-stream sub-stream)))
           (print-solutions goal (rest-stream sub-stream)))))

(defun apply-substitutions (pat slst)
  ;(declare (notinline apply-substitutions))
  (cond ((is-constantp pat) pat)
        ((varp pat)
         (let ((binding (get-binding pat slst)))
           (cond (binding
                  (apply-substitutions (get-binding-value binding) slst))
                 (T pat))))
        (T (cons (apply-substitutions (first pat) slst)
                 (apply-substitutions (rest pat) slst)))))


;;; rule format is
;;;  (RULE IF <premise> THEN <conclusion>)

(defun premise (rule)
```

```
  (nth 2 rule))

(defun conclusion (rule)
  (nth 4 rule))

(defun rulep (pat)
  (and (listp pat) (equal (nth 0 pat) 'RULE)))

;;; conjunctive goals are in the form
;;;  (AND <goal1> <goal2> ... <goalN>)

(defun conjunctive-goalp (goal)
  (and (listp goal) (equal (first goal) 'AND)))

(defun body (goal)
  (rest goal))

;;;rename variables each time a rule is used

(defvar *name-list* nil)

(defun rename-variables (assertion)
  (setq *name-list* nil)
  (rename-recur assertion))

(defun rename-recur (exp)
  (cond ((is-constantp exp) exp)
        ((varp exp) (rename exp))
        (T (cons (rename-recur (first exp))
                 (rename-recur (rest exp))))))

(defun rename (var)
  (list 'VAR
        (or (cdr (assoc (second var) *name-list*))
            (let ((name (gensym)))
              (setq *name-list* (acons (second var) name *name-list*))
              name))))

;;;UNIFY utility
(defun unify (p1 p2 slst)
  (cond ((equal slst '*FAILED*) '*FAILED*)
        ((varp p1) (match-var p1 p2 slst))
        ((varp p2) (match-var p2 p1 slst))
        ((is-constantp p1)
         (if (equal p1 p2) slst '*FAILED*))
        ((is-constantp p2) '*FAILED*)
        (T (unify (rest p1) (rest p2)
                  (unify (first p1) (first p2) slst)))))

(defun match-var (var pat slst)
  (cond ((equal var pat) slst)
        (T (let ((binding (get-binding var slst)))
             (cond (binding
                    (unify (get-binding-value binding) pat slst))
```

```
                  ((occursp var pat) '*FAILED*)
                  (T (add-substitution var pat slst)))))))

(defun occursp (var pat)
  (cond ((equal var pat) T)
        ((or (varp pat) (is-constantp pat)) nil)
        (T (or (occursp var (first pat))
               (occursp var (rest pat))))))

(defun is-constantp (item)
  (atom item))

(defun varp (item)
  (and (listp item)
       (equal (length item) 2)
       (equal (first item) 'VAR)))

(defun get-binding (var slst)
  (assoc var slst :test #'equal))

(defun get-binding-value (binding)
  (cdr binding))

(defun add-substitution (var pat slst)
  (acons var pat slst))



#|
;;; After the code is loaded, select an *ASSERTION* database
;;; and load it into the LISP environment.  **Do not** try to
;;; load or evaluate functions in the LOGIC-SHELL. it is a
;;; continuous loop which hides the LISP context and evaluation
;;; mechanisms.

;;;liking example
;;;  a query would look like (bill likes (var x))
;;;try  (george likes (var x))
;;;try  ((var x) likes kids)
;;;try  ((var x) (var y) everything)
;;;try  (bill friend-of (var x))    and explain the result


(setq *assertions*
      '(
        (george likes beer)
        (george likes kate)
        (george likes kids)
        (tom likes everything)
        (bill likes kids)
        (bill likes music)
        (bill likes pizza)
        (bill likes wine)
```

```
        (rule if
              (and ((var x) likes (var y))
                   ((var z) likes (var y)))
              then
              ((var x) friend-of (var z)))
        ))

;;; new liking example
;;;  facts which declare the uniqueness of names have been added
;;;  and the FRIEND-OF rule now excludes being your own friend
;;;try  (bill friend-of (var x))

(setq *assertions*
      '(
        (george likes beer)
        (george likes kate)
        (george likes kids)
        (tom likes everything)
        (bill likes kids)
        (bill likes music)
        (bill likes pizza)
        (bill likes wine)
        (bill is-not george)
        (tom is-not george)
        (tom is-not bill)

        (rule if
              (and ((var x) likes (var y))
                   ((var z) likes (var y))
                   ((var x) is-not (var z)))
              then
              ((var x) friend-of (var z)))
        ))


;;; backward chaining with zeke the zebra
;;;try  ((var x) zeke)
;;;try  (animal (var x))
;;;try  ((var x) (var y))

(setq *assertions*
      '(
        (zebra zeke)
        (rule if (zebra (var x))
              then
              (herbivore (var x)))
        (rule if (giraffe (var x))
              then
              (herbivore (var x)))
        (rule if (tiger (var x))
              then
              (carnivore (var x)))
        (rule if (leopard (var x))
              then
```

```
              (carnivore (var x)))
        (rule if (herbivore (var x))
              then
              (mammal (var x)))
        (rule if (carnivore (var x))
              then
              (mammal (var x)))
        (rule if (mammal (var x))
              then
              (animal (var x)))
        ))


;;;more backward chaining with zeke
;;; this rulebase is branchier in the backward direction
;;;try  ((var x) zeke)
;;;try  (animal (var x))
;;;try  ((var x) (var y))

(setq *assertions*
      '(
        (zebra zeke)
        (rule if (zebra (var x))
              then
              (striped (var x)))
        (rule if (zebra (var x))
              then
              (height (var x) 10))
        (rule if (zebra (var x))
              then
              (herbivore (var x)))
        (rule if (striped (var x))
              then
              (not (solid (var x))))
        (rule if (striped (var x))
              then
              (not (spotted (var x))))
        (rule if (height (var x) 10)
              then
              (smaller-than-a-house (var x)))
        (rule if (height (var x) 10)
              then
              (bigger-than-a-breadbox (var x)))
        (rule if (herbivore (var x))
              then
              (mammal (var x)))
        (rule if (herbivore (var x))
              then
              (eats-vegetables (var x)))
        (rule if (mammal (var x))
              then
              (warm (var x)))
        (rule if (mammal (var x))
              then
```

```
              (animal (var x)))
        ))


;;;animal rule set with two fact bases for unknown animals
;;;try  (fred is (var x))
;;;try  ((var x) is cheetah)
;;;try  (tony is (var x))
;;;try  ((var x) is tiger)
;;;try  ((var x) is mammal)
;;;try  ((var x) is (var y))
;;;try  ((var x) has (var y))
;;;try  ((var x) has (var y) (var z))   explain the results
;;;try  ((var x) (var y) (var z))

(setq *assertions*
      '(
        (fred has dark spots)
        (fred has tawny color)
        (fred eats meat)
        (fred has hair)
        (tony has hair)
        (tony has pointed teeth)
        (tony has black stripes)
        (tony has claws)
        (tony has forward eyes)
        (tony has tawny color)

        (rule if ((var x) has hair)
              then
              ((var x) is mammal))
        (rule if ((var x) gives milk)
              then
              ((var x) is mammal))
        (rule if ((var x) has feathers)
              then
              ((var x) is bird))
        (rule if (and ((var x) flies)
                      ((var x) lays eggs))
              then
              ((var x) is bird))
        (rule if ((var x) eats meat)
              then
              ((var x) is carnivore))
        (rule if (and ((var x) has pointed teeth)
                      ((var x) has claws)
                      ((var x) has forward eyes))
              then
              ((var x) is carnivore))
        (rule if (and ((var x) is mammal)
                      ((var x) has hoofs))
              then
              ((var x) is ungulate))
        (rule if (and ((var x) is mammal)
```

```
                ((var x) chews cud))
          then
          ((var x) is ungulate))
        (rule if (and ((var x) is mammal)
                ((var x) is carnivore)
                ((var x) has tawny color)
                ((var x) has dark spots))
          then
          ((var x) is cheetah))
        (rule if (and ((var x) is mammal)
                ((var x) is carnivore)
                ((var x) has tawny color)
                ((var x) has black stripes))
          then
          ((var x) is tiger))
        (rule if (and ((var x) is ungulate)
                ((var x) has long neck)
                ((var x) has long legs)
                ((var x) has dark spots))
          then
          ((var x) is giraffe))
        (rule if (and ((var x) is ungulate)
                ((var x) has black stripes))
          then
          ((var x) is zebra))
        (rule if (and ((var x) is bird)
                ((var x) does not fly)
                ((var x) has long neck)
                ((var x) has long legs)
                ((var x) is black and white))
          then
          ((var x) is ostrich))
        (rule if (and ((var x) is bird)
                ((var x) does not fly)
                ((var x) swims)
                ((var x) is black and white))
          then
          ((var x) is penguin))
        (rule if (and ((var x) is bird)
                ((var x) flys well))
          then
          ((var x) is albatross))
        )  )


;;;relatives
;;;try  (mother (var x) Bert)
;;;try  (parent (var x) Bert)
;;;try  (sibling Carl (var z))     expalin the results
;;;try  (cousin (var x) (var y))
;;;try  (ancestor Art George)      explain the results

(setq *assertions*
      '(
```

```
        (mother Abby Bert)
        (mother Abby Betty)
        (mother Betty Claire)
        (mother Betty Carl)
        (mother Julia Karen)
        (father Art Bert)
        (father Art Betty)
        (father Gary Henry)
        (father Gary Hank)
        (father Carl George)
        (rule if (mother (var x) (var y))
              then
              (parent (var x) (var y)))
        (rule if (father (var x) (var y))
              then
              (parent (var x) (var y)))
        (rule if (parent (var x) (var y))
              then
              (ancestor (var x) (var y)))
        (rule if (and (ancestor (var x) (var y))
                      (parent (var y) (var z)))
              then
              (ancestor (var x) (var z)))
        (rule if (and (parent (var x1) (var y1))
                      (parent (var x2) (var y2))
                      (sibling (var x1) (var x2)))
              then
              (cousin (var y1) (var y2)))
        (rule if (and (parent (var x) (var y1))
                      (parent (var x) (var y2)))
              then
              (sibling (var y1) (var y2)))
        ))


;;;doing addition
;;;try  ((var x) + 6 = 1 1)
;;;try  ((var x) + (var y) = 0 6)
;;;try  (4 + (var x) = 1 (var y))
;;;try  ((var x) (var y) 6 = (var z) 0)
;;;try  (4 5 + 5 9 = (var x)(var y)(var z))
;;;try  (4 5 + (var x)(var y) = 1 0 4)
;;;try  ((var x) 7 + 5 (var y) = 0 (var z) 1)
;;;try  ((var x) (var y) + 5 (var z) = 0 (var y) (var x))
;;;try  ((var x) (var y) + 5 (var z) = (var x) (var y) (var x))

(setq *assertions*
      '(
        (1 + 0 = 0 1)
        (0 + 0 = 0 0)
        (0 + 1 = 0 1)
        (0 + 2 = 0 2)
        (0 + 3 = 0 3)
        (0 + 4 = 0 4)
```

(0 + 5 = 0 5)
(0 + 6 = 0 6)
(0 + 7 = 0 7)
(0 + 8 = 0 8)
(0 + 9 = 0 9)
(1 + 1 = 0 2)
(1 + 2 = 0 3)
(1 + 3 = 0 4)
(1 + 4 = 0 5)
(1 + 5 = 0 6)
(1 + 6 = 0 7)
(1 + 7 = 0 8)
(1 + 8 = 0 9)
(1 + 9 = 1 0)
(2 + 0 = 0 2)
(2 + 1 = 0 3)
(2 + 2 = 0 4)
(2 + 3 = 0 5)
(2 + 4 = 0 6)
(2 + 5 = 0 7)
(2 + 6 = 0 8)
(2 + 7 = 0 9)
(2 + 8 = 1 0)
(2 + 9 = 1 1)
(3 + 0 = 0 3)
(3 + 1 = 0 4)
(3 + 2 = 0 5)
(3 + 3 = 0 6)
(3 + 4 = 0 7)
(3 + 5 = 0 8)
(3 + 6 = 0 9)
(3 + 7 = 1 0)
(3 + 8 = 1 1)
(3 + 9 = 1 2)
(4 + 0 = 0 4)
(4 + 1 = 0 5)
(4 + 2 = 0 6)
(4 + 3 = 0 7)
(4 + 4 = 0 8)
(4 + 5 = 0 9)
(4 + 6 = 1 0)
(4 + 7 = 1 1)
(4 + 8 = 1 2)
(4 + 9 = 1 3)
(5 + 0 = 0 5)
(5 + 1 = 0 6)
(5 + 2 = 0 7)
(5 + 3 = 0 8)
(5 + 4 = 0 9)
(5 + 5 = 1 0)
(5 + 6 = 1 1)
(5 + 7 = 1 2)
(5 + 8 = 1 3)
(5 + 9 = 1 4)

```
        (6 + 0 = 0 6)
        (6 + 1 = 0 7)
        (6 + 2 = 0 8)
        (6 + 3 = 0 9)
        (6 + 4 = 1 0)
        (6 + 5 = 1 1)
        (6 + 6 = 1 2)
        (6 + 7 = 1 3)
        (6 + 8 = 1 4)
        (6 + 9 = 1 5)
        (7 + 0 = 0 7)
        (7 + 1 = 0 8)
        (7 + 2 = 0 9)
        (7 + 3 = 1 0)
        (7 + 4 = 1 1)
        (7 + 5 = 1 2)
        (7 + 6 = 1 3)
        (7 + 7 = 1 4)
        (7 + 8 = 1 5)
        (7 + 9 = 1 6)
        (8 + 0 = 0 8)
        (8 + 1 = 0 9)
        (8 + 2 = 1 0)
        (8 + 3 = 1 1)
        (8 + 4 = 1 2)
        (8 + 5 = 1 3)
        (8 + 6 = 1 4)
        (8 + 7 = 1 5)
        (8 + 8 = 1 6)
        (8 + 9 = 1 7)
        (9 + 0 = 0 9)
        (9 + 1 = 1 0)
        (9 + 2 = 1 1)
        (9 + 3 = 1 2)
        (9 + 4 = 1 3)
        (9 + 5 = 1 4)
        (9 + 6 = 1 5)
        (9 + 7 = 1 6)
        (9 + 8 = 1 7)
        (9 + 9 = 1 8)

        (rule if
              (and ((var b) + (var d) = (var x0) (var g))
                   ((var a) + (var c) = (var x1) (var x2))
                   ((var x0) + (var x2) = (var x3) (var f))
                   ((var x1) + (var x3) = 0 (var e)))
                   then
              ((var a) (var b) + (var c) (var d) = (var e) (var f) (var g)))
        ))

;;;built-in carry allows easier rules for adding large numbers
;;;try  (4 5 6 7 8 + 8 7 6 5 4 = (var a)(var b)(var c)(var d)(var e)(var f))
;;;try  (4 (var a) 6 + (var a) 3 (var b) = (var c) 6 (var b) (var a))
;;;try  ((var a)(var a) + (var b)(var b) = (var c)(var d)(var c))
```

```
;;;try  ((var a)(var a) + (var b)(var b) = (var c)(var b)(var c))
;;;don't try  ((var s)(var e)(var n)(var d) + (var m)(var o)(var r)(var e)
;;;            = (var m)(var o)(var n)(var e)(var y))

(setq *assertions*
      '(
        (0 + 0 + 0 = 0 0)
        (0 + 0 + 1 = 0 1)
        (0 + 0 + 2 = 0 2)
        (0 + 0 + 3 = 0 3)
        (0 + 0 + 4 = 0 4)
        (0 + 0 + 5 = 0 5)
        (0 + 0 + 6 = 0 6)
        (0 + 0 + 7 = 0 7)
        (0 + 0 + 8 = 0 8)
        (0 + 0 + 9 = 0 9)
        (0 + 1 + 0 = 0 1)
        (0 + 1 + 1 = 0 2)
        (0 + 1 + 2 = 0 3)
        (0 + 1 + 3 = 0 4)
        (0 + 1 + 4 = 0 5)
        (0 + 1 + 5 = 0 6)
        (0 + 1 + 6 = 0 7)
        (0 + 1 + 7 = 0 8)
        (0 + 1 + 8 = 0 9)
        (0 + 1 + 9 = 1 0)
        (0 + 2 + 0 = 0 2)
        (0 + 2 + 1 = 0 3)
        (0 + 2 + 2 = 0 4)
        (0 + 2 + 3 = 0 5)
        (0 + 2 + 4 = 0 6)
        (0 + 2 + 5 = 0 7)
        (0 + 2 + 6 = 0 8)
        (0 + 2 + 7 = 0 9)
        (0 + 2 + 8 = 1 0)
        (0 + 2 + 9 = 1 1)
        (0 + 3 + 0 = 0 3)
        (0 + 3 + 1 = 0 4)
        (0 + 3 + 2 = 0 5)
        (0 + 3 + 3 = 0 6)
        (0 + 3 + 4 = 0 7)
        (0 + 3 + 5 = 0 8)
        (0 + 3 + 6 = 0 9)
        (0 + 3 + 7 = 1 0)
        (0 + 3 + 8 = 1 1)
        (0 + 3 + 9 = 1 2)
        (0 + 4 + 0 = 0 4)
        (0 + 4 + 1 = 0 5)
        (0 + 4 + 2 = 0 6)
        (0 + 4 + 3 = 0 7)
        (0 + 4 + 4 = 0 8)
        (0 + 4 + 5 = 0 9)
        (0 + 4 + 6 = 1 0)
        (0 + 4 + 7 = 1 1)
```

(0 + 4 + 8 = 1 2)
(0 + 4 + 9 = 1 3)
(0 + 5 + 0 = 0 5)
(0 + 5 + 1 = 0 6)
(0 + 5 + 2 = 0 7)
(0 + 5 + 3 = 0 8)
(0 + 5 + 4 = 0 9)
(0 + 5 + 5 = 1 0)
(0 + 5 + 6 = 1 1)
(0 + 5 + 7 = 1 2)
(0 + 5 + 8 = 1 3)
(0 + 5 + 9 = 1 4)
(0 + 6 + 0 = 0 6)
(0 + 6 + 1 = 0 7)
(0 + 6 + 2 = 0 8)
(0 + 6 + 3 = 0 9)
(0 + 6 + 4 = 1 0)
(0 + 6 + 5 = 1 1)
(0 + 6 + 6 = 1 2)
(0 + 6 + 7 = 1 3)
(0 + 6 + 8 = 1 4)
(0 + 6 + 9 = 1 5)
(0 + 7 + 0 = 0 7)
(0 + 7 + 1 = 0 8)
(0 + 7 + 2 = 0 9)
(0 + 7 + 3 = 1 0)
(0 + 7 + 4 = 1 1)
(0 + 7 + 5 = 1 2)
(0 + 7 + 6 = 1 3)
(0 + 7 + 7 = 1 4)
(0 + 7 + 8 = 1 5)
(0 + 7 + 9 = 1 6)
(0 + 8 + 0 = 0 8)
(0 + 8 + 1 = 0 9)
(0 + 8 + 2 = 1 0)
(0 + 8 + 3 = 1 1)
(0 + 8 + 4 = 1 2)
(0 + 8 + 5 = 1 3)
(0 + 8 + 6 = 1 4)
(0 + 8 + 7 = 1 5)
(0 + 8 + 8 = 1 6)
(0 + 8 + 9 = 1 7)
(0 + 9 + 0 = 0 9)
(0 + 9 + 1 = 1 0)
(0 + 9 + 2 = 1 1)
(0 + 9 + 3 = 1 2)
(0 + 9 + 4 = 1 3)
(0 + 9 + 5 = 1 4)
(0 + 9 + 6 = 1 5)
(0 + 9 + 7 = 1 6)
(0 + 9 + 8 = 1 7)
(0 + 9 + 9 = 1 8)

(1 + 0 + 0 = 0 1)

(1 + 0 + 1 = 0 2)
(1 + 0 + 2 = 0 3)
(1 + 0 + 3 = 0 4)
(1 + 0 + 4 = 0 5)
(1 + 0 + 5 = 0 6)
(1 + 0 + 6 = 0 7)
(1 + 0 + 7 = 0 8)
(1 + 0 + 8 = 0 9)
(1 + 0 + 9 = 1 0)
(1 + 1 + 0 = 0 2)
(1 + 1 + 1 = 0 3)
(1 + 1 + 2 = 0 4)
(1 + 1 + 3 = 0 5)
(1 + 1 + 4 = 0 6)
(1 + 1 + 5 = 0 7)
(1 + 1 + 6 = 0 8)
(1 + 1 + 7 = 0 9)
(1 + 1 + 8 = 1 0)
(1 + 1 + 9 = 1 1)
(1 + 2 + 0 = 0 3)
(1 + 2 + 1 = 0 4)
(1 + 2 + 2 = 0 5)
(1 + 2 + 3 = 0 6)
(1 + 2 + 4 = 0 7)
(1 + 2 + 5 = 0 8)
(1 + 2 + 6 = 0 9)
(1 + 2 + 7 = 1 0)
(1 + 2 + 8 = 1 1)
(1 + 2 + 9 = 1 2)
(1 + 3 + 0 = 0 4)
(1 + 3 + 1 = 0 5)
(1 + 3 + 2 = 0 6)
(1 + 3 + 3 = 0 7)
(1 + 3 + 4 = 0 8)
(1 + 3 + 5 = 0 9)
(1 + 3 + 6 = 1 0)
(1 + 3 + 7 = 1 1)
(1 + 3 + 8 = 1 2)
(1 + 3 + 9 = 1 3)
(1 + 4 + 0 = 0 5)
(1 + 4 + 1 = 0 6)
(1 + 4 + 2 = 0 7)
(1 + 4 + 3 = 0 8)
(1 + 4 + 4 = 0 9)
(1 + 4 + 5 = 1 0)
(1 + 4 + 6 = 1 1)
(1 + 4 + 7 = 1 2)
(1 + 4 + 8 = 1 3)
(1 + 4 + 9 = 1 4)
(1 + 5 + 0 = 0 6)
(1 + 5 + 1 = 0 7)
(1 + 5 + 2 = 0 8)
(1 + 5 + 3 = 0 9)
(1 + 5 + 4 = 1 0)

```
(1 + 5 + 5 = 1 1)
(1 + 5 + 6 = 1 2)
(1 + 5 + 7 = 1 3)
(1 + 5 + 8 = 1 4)
(1 + 5 + 9 = 1 5)
(1 + 6 + 0 = 0 7)
(1 + 6 + 1 = 0 8)
(1 + 6 + 2 = 0 9)
(1 + 6 + 3 = 1 0)
(1 + 6 + 4 = 1 1)
(1 + 6 + 5 = 1 2)
(1 + 6 + 6 = 1 3)
(1 + 6 + 7 = 1 4)
(1 + 6 + 8 = 1 5)
(1 + 6 + 9 = 1 6)
(1 + 7 + 0 = 0 8)
(1 + 7 + 1 = 0 9)
(1 + 7 + 2 = 1 0)
(1 + 7 + 3 = 1 1)
(1 + 7 + 4 = 1 2)
(1 + 7 + 5 = 1 3)
(1 + 7 + 6 = 1 4)
(1 + 7 + 7 = 1 5)
(1 + 7 + 8 = 1 6)
(1 + 7 + 9 = 1 7)
(1 + 8 + 0 = 0 9)
(1 + 8 + 1 = 1 0)
(1 + 8 + 2 = 1 1)
(1 + 8 + 3 = 1 2)
(1 + 8 + 4 = 1 3)
(1 + 8 + 5 = 1 4)
(1 + 8 + 6 = 1 5)
(1 + 8 + 7 = 1 6)
(1 + 8 + 8 = 1 7)
(1 + 8 + 9 = 1 8)
(1 + 9 + 0 = 1 0)
(1 + 9 + 1 = 1 1)
(1 + 9 + 2 = 1 2)
(1 + 9 + 3 = 1 3)
(1 + 9 + 4 = 1 4)
(1 + 9 + 5 = 1 5)
(1 + 9 + 6 = 1 6)
(1 + 9 + 7 = 1 7)
(1 + 9 + 8 = 1 8)
(1 + 9 + 9 = 1 9)

(rule if
      (and (       0 + (var a0) + (var b0) = (var c1) (var r0))
           ((var c1) + (var a1) + (var b1) = (var c2) (var r1)))
           then
      (           (var a1) (var a0)
                +  (var b1) (var b0)
       = (var c2) (var r1) (var r0)))
```

```
        (rule if
              (and (       0 + (var a0) + (var b0) = (var c1) (var r0))
                   ((var c1) + (var a1) + (var b1) = (var c2) (var r1))
                   ((var c2) + (var a2) + (var b2) = (var c3) (var r2)))
                   then
              (           (var a2) (var a1) (var a0)
                        +  (var b2) (var b1) (var b0)
               = (var c3) (var r2) (var r1) (var r0)))

        (rule if
              (and (       0 + (var a0) + (var b0) = (var c1) (var r0))
                   ((var c1) + (var a1) + (var b1) = (var c2) (var r1))
                   ((var c2) + (var a2) + (var b2) = (var c3) (var r2))
                   ((var c3) + (var a3) + (var b3) = (var c4) (var r3)))
                   then
              (           (var a3) (var a2) (var a1) (var a0)
                        +  (var b3) (var b2) (var b1) (var b0)
               = (var c4) (var r3) (var r2) (var r1) (var r0)))

        (rule if
              (and (       0 + (var a0) + (var b0) = (var c1) (var r0))
                   ((var c1) + (var a1) + (var b1) = (var c2) (var r1))
                   ((var c2) + (var a2) + (var b2) = (var c3) (var r2))
                   ((var c3) + (var a3) + (var b3) = (var c4) (var r3))
                   ((var c4) + (var a4) + (var b4) = (var c5) (var r4)))
                   then
              (           (var a4) (var a3) (var a2) (var a1) (var a0)
                        +  (var b4) (var b3) (var b2) (var b1) (var b0)
               = (var c5) (var r4) (var r3) (var r2) (var r1) (var r0)))
        ))

|#
```