```
;;;;          Least-General Generalization (LGG)
;;;   This version was written by GJS on 9 March 2020.

#|
              The LGG is the dual to the Unifier,
           which is the Least Special Specialization


   The LGG of two terms was introduced by Plotkin in
   1970.  (See Gordon D. Plotkin, "A Note on Inductive
   Generalization" in Machine Intelligence 5, Meltzer and
   Michie editors, pp.153--163.
|#
```

For example

```
(define p1
  '(define (expt x n)
     (if (= n 0)
         1
         (* x (expt x (- n 1))))))

(define p2
  '(define (repeated f n)
     (if (= n 0)
         identity
         (compose f (repeated f (- n 1))))))

(pp (lgg p1 p2))
((define ((@ |G229|) (@ |G230|) n)
   (if (= n 0)
       (@ |G231|)
       ((@ |G232|) (@ |G230|)
                   ((@ |G229|) (@ |G230|) (- n 1)))))
 (((* compose) (@ |G232|))
  ((1 identity) (@ |G231|))
  ((x f) (@ |G230|))
  ((expt repeated) (@ |G229|))))
```

```
;;; Here we see the power!


(pp (lgg-compare p1 p2))
(define ((: (repeated expt)) (: (f x)) n)
  (if (= n 0)
      (: (identity 1))
      ((: (compose *)) (: (f x))
                       ((: (repeated expt)) (: (f x))
                                            (- n 1)))))
```

```
;;;; The LGG program

(define (lgg-compare pat1 pat2)
  (apply lgg-simplify (lgg pat1 pat2)))

(define (lgg pat1 pat2)
  (lgg-internal pat1 pat2
                (new-bindings)
                (lambda (pat* bindings*)
                  (list pat* bindings*))))

(define (lgg-internal pat1 pat2 bindings cont)
  (cond ((equal? pat1 pat2)
         (cont pat1 bindings))
        ((and (not (var? pat1)) (not (var? pat2))
              (pair? pat1) (pair? pat2))
         (lgg-pairs pat1 pat2 bindings cont))
        (else          ; not equal and not pairs
         (make-generalization pat1 pat2 bindings cont))))
```

```
(define (lgg-pairs pat1 pat2 bindings cont)
  (lgg-internal (car pat1)
                (car pat2)
                bindings
                (lambda (car-pat bindings*)
                  (lgg-internal (cdr pat1)
                                (cdr pat2)
                                bindings*
                                (lambda (cdr-pat bindings**)
                                  (cont (cons car-pat cdr-pat)
                                        bindings**))))))

(define (make-generalization pat1 pat2 bindings cont)
  (let* ((key (list pat1 pat2))
         (seen? (lookup-binding key bindings)))
    (if seen?
        (cont (binding-variable seen?) bindings)
        (let ((var (new-var)))
          (cont var
                (extend-bindings (list pat1 pat2)
                                 var
                                 bindings))))))
```

```scheme
(define (lgg-simplify pat bindings)
  (let lp ((pat pat))
    (cond ((null? pat) '())
          ((var? pat)
           (list ': (expand-var pat bindings)))
          ((list? pat)
           (map lp pat))
          (else pat))))

(define (expand-var var bindings)
  (let ((binding (lookup-variable var bindings)))
    (let lp ((stuff (binding-patterns binding))
             (accum '()))
      (cond ((null? stuff) accum)
            ((var? (car stuff))
             (let ((more-stuff
                    (expand-var (car stuff) bindings)))
               (lp (lset-union equal? (cdr stuff) more-stuff)
                   accum)))
            (else
             (lp (cdr stuff)
                 (lset-adjoin equal?
                              accum
                              (lgg-simplify (car stuff)
                                            bindings))))))))
```

```
;;; Variables

(define (new-var)
  (list '@ (generate-uninterned-symbol)))

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

;;; Bindings connect patterns with variables.

(define (new-bindings) '())
(define (empty-bindings? b) (null? b))

(define (lookup-binding key bindings)
  (assoc key bindings))

(define lookup-variable
  (association-procedure eqv? cadr))

(define (extend-bindings pats var bindings)
  (cons (list pats var) bindings))

(define (binding-patterns binding) (car binding))
(define (binding-variable binding) (cadr binding))
```

More examples

```
(pp (lgg '(separates pacific canada japan)
         '(separates pacific mexico japan)))

((separates pacific (@ |G21|) japan)
 (((canada mexico) (@ |G21|))))

(pp (lgg (list (list 'a 'b) (list (list 'a 'b) 'c))
         (list 'd (list 'd 'c))))
(((@ |G29|) ((@ |G29|) c))
 ((((a b) d) (@ |G29|))))

;;; But this is different...
(pp (lgg (list (list 'a 'b) (list (list 'a 'b) 'c))
         (list 'd (list 'e 'c))))
(((@ |G30|) ((@ |G31|) c))
 ((((a b) e) (@ |G31|)) (((a b) d) (@ |G30|))))
```

```
(lgg-internal '(separates pacific canada japan)
              '(separates pacific mexico japan)
              (new-bindings)
              (lambda (pat* bindings*)
                (lgg-internal '(separates pacific usa japan)
                              pat*
                              bindings*
                              (lambda (pat** bindings*)
                                (pp `(,pat** ,bindings*))))))

((separates pacific (@ |G33|) japan)
 (((usa (@ |G32|)) (@ |G33|))
  ((canada mexico) (@ |G32|))))


(define (generalize pat . pats)
  (let lp ((pats pats) (accum pat) (bindings (new-bindings)))
    (if (null? pats)
        (lgg-simplify accum bindings)
        (lgg-internal accum (car pats) bindings
                      (lambda (pat* bindings*)
                        (lp (cdr pats) pat* bindings*))))))
```

```
(define separations
 '((separates pacific canada japan)
   (separates pacific mexico japan)
   (separates pacific usa japan)
   (separates atlantic canada denmark)
   (separates atlantic canada france)
   (separates atlantic canada germany)
   (separates atlantic canada italy)
   (separates atlantic canada spain)
   (separates atlantic canada sweden)
   (separates atlantic canada uk)
   (separates atlantic mexico denmark)
   (separates atlantic mexico france)
   (separates atlantic mexico germany)
   (separates atlantic mexico italy)
   (separates atlantic mexico spain)
   (separates atlantic mexico sweden)
   (separates atlantic mexico uk)
   (separates atlantic usa denmark)
   (separates atlantic usa france)
   (separates atlantic usa germany)
   (separates atlantic usa italy)
   (separates atlantic usa spain)
   (separates atlantic usa sweden)
   (separates atlantic usa uk)))
```

```
(pp (apply generalize separations))

(separates (: (atlantic pacific))
           (: (usa mexico canada))
           (: (uk sweden spain italy germany
                  france denmark japan)))
```

```
;;;                    Detecting Software Clones

(define p3
  '(define (lgg pat1 pat2)
     (lgg-internal pat1 pat2
                   (new-bindings)
                   (lambda (pat* bindings*)
                     (list pat* bindings*)))))

(define p4
  '(define (unify x y)
     (unify:internal x y
                     (match:new-dict)
                     (lambda (dict) dict))))

(pp (lgg-compare p3 p4))
(define ((: (unify lgg)) (: (x pat1)) (: (y pat2)))
  ((: (unify:internal lgg-internal))
   (: (x pat1))
   (: (y pat2))
   ((: (match:new-dict new-bindings)))
   (lambda ((: (dict pat*)) . (@ |G243|))
     (: (dict (list pat* bindings*))))))
```

```
;;;;                    More Seriously, from SRFI 1


;;; A dotted list is a finite list (possibly of length 0)
;;; terminated by a non-nil value. Any non-cons, non-nil
;;; value (e.g., "foo" or 5) is a dotted list of length 0.
;;;
;;; <dotted-list> ::= <non-nil,non-pair>
;;;               |   (cons <x> <dotted-list>)

;;; A dotted list is a finite list (possibly of length 0)
;;; terminated by a nil value.
;;; <proper-list> ::= ()
;;;               |   (cons <x> <proper-list>)
;;; Note that this definition rules out circular lists --
;;; and this function is required to detect this case and
;;; return false.

;;; Reference implementations given in SRFI 1 below:
```

```
(define p5
  '(define (dotted-list? x)
     (let lp ((x x) (lag x))
       (if (pair? x)
           (let ((x (cdr x)))
             (if (pair? x)
                 (let ((x (cdr x)) (lag (cdr lag)))
                   (and (not (eq? x lag)) (lp x lag)))
                 (not (null? x))))
           (not (null? x))))))

(define p6
  '(define (proper-list? x)
     (let lp ((x x) (lag x))
       (if (pair? x)
           (let ((x (cdr x)))
             (if (pair? x)
                 (let ((x (cdr x)) (lag (cdr lag)))
                   (and (not (eq? x lag)) (lp x lag)))
                 (null? x)))
           (null? x)))))
```

```
;;;              Discovering Abstractions Automagically

(pp (lgg-compare p5 p6))

(define ((: (proper-list? dotted-list?)) x)
  (let lp ((x x) (lag x))
    (if (pair? x)
        (let ((x (cdr x)))
          (if (pair? x)
              (let ((x (cdr x)) (lag (cdr lag)))
                (and (not (eq? x lag))
                     (lp x lag)))
              ((: (null? not)) (: (x (null? x))))))
        ((: (null? not)) (: (x (null? x)))))))
```

```
;;; These could both be made as instances of an abstraction:

(define (list-check-tail check?)
  (lambda (x)
    (let lp ((x x) (lag x))
      (if (pair? x)
          (let ((x (cdr x)))
            (if (pair? x)
                (let ((x (cdr x)) (lag (cdr lag)))
                  (and (not (eq? x lag))
                       (lp x lag)))
                (list-tail-check x)))
          (check? x)))

(define dotted-list?
  (list-check-tail (lambda (x) (not (null? x)))))

(define proper-list? (list-check-tail null?))

;;; Could this be automagically discovered by a compiler?
```

Other Applications

Analogies
Inductive Generalization from Data

Cheating (Plagiarism) Detection
Copyright Infringement Detection

Polymorphic Type Inference
Finding the minimal union type