```
;; Question 1 [60 seconds]
;;
;; What value does running this program produce?

(define s "summer")
(local [(define s "spring")
        (define (is-favourite? x)
          (string=? x s))]
   (is-favourite? "summer"))

;; A. true
;; B. false
```

```
;; Question 2 [60 seconds]
;;
;; Consider the following expression. Which of the following
;; is the correct first evaluation step?

(local [(define x 3)
        (define (timesx y)
          (* x y))]
  (timesx 5))

;; A. (define x_0 3)
;;    (local [(define (timesx y)
;;              (* x_0 y))]
;;      (timesx 5))

;; B. (define x_0 3)
;;    (define (timesx_0 y)
;;      (* x_0 y))
;;    (timesx_0 5)

;; C. (define (timesx_0 y)
;;      (* 3 y))
;;    (timesx_0 5)
```

```
;; Question 3
;;
;; When the following expression is evaluated how many times is the + primitive
;; called?

(local [(define a (+ 2 3))]
  (* a a))

;; A. 0
;; B. 1
;; C. 2
;; D. 3
```

```
;; Question 4
;;
;; How many definitions are lifted when the following program is run?

(define (play-with-strings s t)
  (local [(define s2 (substring s 0 3))
          (define (t2 x) (string-append t x "turquoise"))
          (define s3 "purple")]
    (string-append s2 (t2 "white") s3)))

(string-append (play-with-strings "orange" "red")
               (play-with-strings "yellow" "green"))

;; A. 3
;; B. 2
;; C. 5
;; D. 6
```

3 ways to use local:

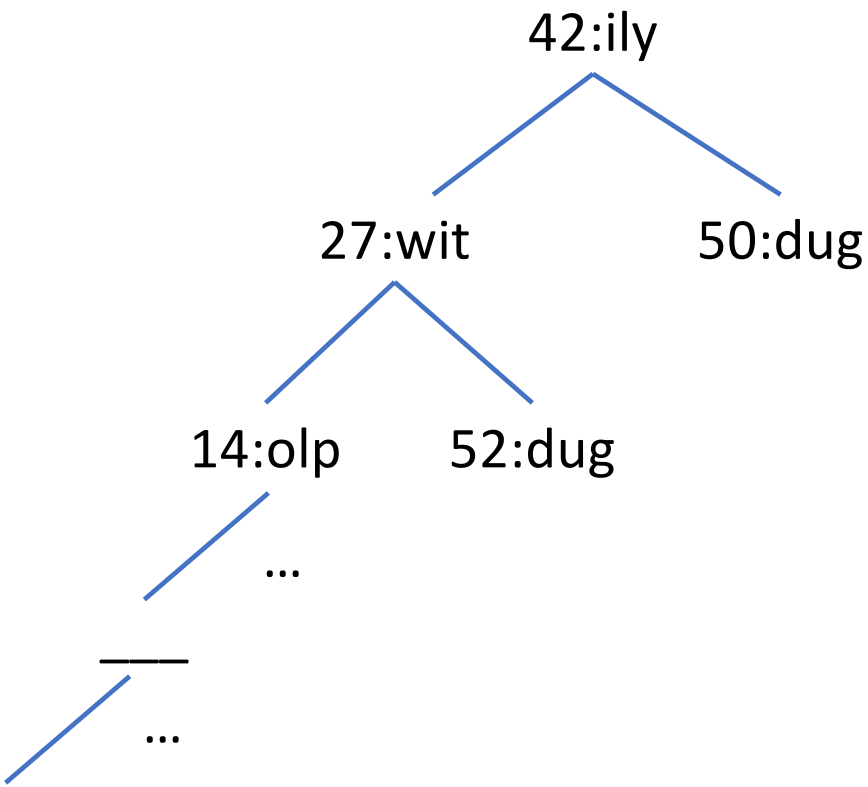
- eliminate identical recursive calls
- encapsulate 2 or more definitions
- improve code clarity by giving names to intermediate values

refactoring

- a fundamental technique for improving code
- take a correct running program, refactor to improve structure

3 ways to use local - either via refactoring, or when initially writing code

	Eliminate…	Encapsulate 2…	Give names to vals
Via refactoring			
While originally writing code			


42:ily
27:wit
50:dug
14:olp
52:dug
…
___
…

```
(@htdf all-labels--region all-labels--lor)
(@signature Region -> ListOfString)
(@signature ListOfRegion -> ListOfString)
;; produce labels of all regions in region (including root)
(check-expect (all-labels--lor empty) '())
(check-expect (all-labels--region S1) (list "one"))
(check-expect (all-labels--lor LOR123) (list "one" "two" "three"))
(check-expect (all-labels--region G4)
              (list "one" "two" "three"
                    "four" "five" "six"))

(@template-origin Region)

(define (all-labels--region r)
  (cond [(single? r) (list (single-label r))]
        [else
         (all-labels--lor (group-subs r))]))

(@template-origin ListOfRegion)

(define (all-labels--lor lor)
  (cond [(empty? lor) empty]
        [else
         (append (all-labels--region (first lor))
                 (all-labels--lor (rest lor)))]))
```

```
#| Refactor using local to encapsulate. |#

(@htdf all-with-color)
(@signature Color Region -> ListOfRegion)
;; produce all regions with given color
(check-expect (all-with-color "red" S1) (list S1))
(check-expect (all-with-color "blue" S1) empty)
(check-expect (all-with-color "red"
                              (make-group "blue"
                                          (list G4
                                                (make-single "X" 90 "red"))))
              (list G1 S1 (make-single "X" 90 "red")))

(@template-origin Region ListOfRegion encapsulated)

(define (all-with-color c r)
  (local [(define (all-with-color--region c r)
            (cond [(single? r)
                   (if (string=? (single-color r) c) (list r) empty)]
                  [else
                   (if (string=? (group-color r) c)
                       (cons r (all-with-color--lor c (group-subs r)))
                       (all-with-color--lor c (group-subs r)))]))

          (define (all-with-color--lor c lor)
            (cond [(empty? lor) empty]
                  [else
                   (append (all-with-color--region c (first lor))
                           (all-with-color--lor c (rest lor)))]))]

    (all-with-color--region c r)))
```

```
(@htdf all-with-color--region all-with-color--lor)
(@signature Color Region -> ListOfRegion)
(@signature Color ListOfRegion -> ListOfRegion)
;; produce all regions with given color
(check-expect (all-with-color--lor "red" empty) '())
(check-expect (all-with-color--region "red" S1) (list S1))
(check-expect (all-with-color--region "blue" S1) '())
(check-expect (all-with-color--lor "red" LOR123) (list S1))
(check-expect
 (all-with-color--region "red"
                         (make-group "blue"
                                     (list G4
                                           (make-single "X" 90 "red'
 (list G1 S1 (make-single "X" 90 "red")))

(@template-origin Region)

(define (all-with-color--region c r)
  (cond [(single? r) (if (string=? (single-color r) c) (list r) '()
        [else
         (if (string=? (group-color r) c)
             (cons r (all-with-color--lor c (group-subs r)))
             (all-with-color--lor c (group-subs r)))]))

(@template-origin ListOfRegion)

(define (all-with-color--lor c lor)
  (cond [(empty? lor) empty]
        [else
         (append (all-with-color--region c (first lor))
                 (all-with-color--lor c (rest lor)))]))
```