# Evaluator: Recursive Calls

```
(define (evaluate e env d)
  (cond
   [(boolean? e) e]
   ....
   [(plus? e) (+ (evaluate (plus-left e) env d)
                 (evaluate (plus-right e) env d))]
   ....
   [(app? e)
    (define f (evaluate (app-func e) env d))
    (evaluate (function-body f)
              (make-sub (function-arg-name f)
                        (evaluate (app-arg e) env d)
                        (function-env f))
              d)]
   [(lambda? e)
    (make-function (lambda-arg-name e)
                   (lambda-body e)
                   env)]
   ....))
```

# Print List: Tail Form

```
(define (print-list strs)
  (cond
   [(empty? strs) (void)]
   [else (displayln (first strs))
         (print-list (rest strs))]))
```

# Print Reverse: Non-Tail Form

```
(define (print-list-rev strs)
  (cond
   [(empty? strs) (void)]
   [else (print-list-rev (rest strs))
         (displayln (first strs))]))
```

# Print Reverse: Tail Form

```
(define (print-list-rev* strs todo)
  (cond
   [(empty? strs) (perform-todos todo)]
   [else (print-list-rev* (rest strs)
                          (cons (first strs)
                                todo))]))

(define (perform-todos todos)
  (cond
   [(empty? todos) (void)]
   [else (displayln (first todos))
         (perform-todos (rest todos))]))
```

# Enumerate: Tail Form

```
(define (enumerate strs pos)
  (cond
   [(empty? strs) (void)]
   [else
    (printf "~a. ~a\n" pos (first strs))
    (enumerate (rest strs) (+ pos 1))]))
```

# Count Down: Non-Tail Form

```
(define (count-down strs pos)
  (cond
   [(empty? strs) (void)]
   [else
    (count-down (rest strs) (+ pos 1))
    (printf "~a. ~a\n" pos (first strs))]))
```

# Count Down: Tail Form

```
(define (count-down* strs pos todo)
  (cond
   [(empty? strs) (perform-todos todo)]
   [else (count-down* (rest strs)
                      (+ pos 1)
                      (cons (list pos (first strs))
                            todo))]))

(define (perform-todos todos)
  (cond
   [(empty? todos) (void)]
   [else
    (define todo (first todos))
    (printf "~a. ~a\n" (first todo) (second todo))
    (perform-todos (rest todos))]))
```

# Print with Counter: Tail Form

```
(define (print-list+count strs n)
  (cond
   [(empty? strs) (void)]
   [else
    (define s (first strs))
    (printf "~a ~a\n" n s)
    (print-list+count (rest strs)
                      (+ n (string-length s)))]))
```

# Reverse Print with Counter: Non-Tail Form

```
(define (print-list-rev+count strs)
  (cond
   [(empty? strs) 0]
   [else
    (define s (first strs))
    (define n (print-list-rev+count (rest strs)))
    (printf "~a ~a\n" n s)
    (+ n (string-length s))]))
```

# Reverse Print with Counter: Tail Form

```
(define (print-list-rev+count* strs todo)
  (cond
   [(empty? strs) (perform-todos todo 0)]
   [else (print-list-rev+count*
          (rest strs)
          (cons (first strs)
                todo))]))

(define (perform-todos todos n)
  (cond
   [(empty? todos) (void)]
   [else
    (define s (first todos))
    (printf "~a ~a\n" n s)
    (perform-todos (rest todos)
                   (+ n (string-length s)))]))
```

# Print Tree: Non-Tail Form

```
(define-struct tree (left val right))

(define (print-tree t)
  (cond
   [(empty? t) (void)]
   [else
    (print-tree (tree-left t))
    (displayln (tree-val t))
    (print-tree (tree-right t))]))
```

# Print Tree: Tail Form

```
(define (print-tree* t todos)
  (cond
   [(empty? t) (perform-todos todos)]
   [else
    (print-tree* (tree-left t)
                 (cons t todos))]))

(define (perform-todos todos)
  (cond
   [(empty? todos) (void)]
   [else
    (define t (first todos))
    (displayln (tree-val t))
    (print-tree* (tree-right t) (rest todos))]))
```

# Print with Depth: Non-Tail Form

```
(define (print-tree+depth t d)
  (cond
   [(empty? t) (void)]
   [else
    (print-tree+depth (tree-left t) (+ d 1))
    (printf "~a ~a\n" d (tree-val t))
    (print-tree+depth (tree-right t) (+ d 1))]))
```

# Print with Depth: Tail Form

```
(define (print-tree+depth* t d todos)
  (cond
   [(empty? t) (perform-todos todos)]
   [else
    (print-tree+depth* (tree-left t)
                       (+ 1 d)
                       (cons (list d t) todos))]))

(define (perform-todos todos)
  (cond
   [(empty? todos) (void)]
   [else
    (define d (first (first todos)))
    (define t (second (first todos)))
    (printf "~a ~a\n" d (tree-val t))
    (print-tree+depth* (tree-right t)
                       (+ 1 d)
                       (rest todos))]))
```

# Tree Node Enumerate: Non-Tail Form

```
(define (enumerate t pos)
  (cond
   [(empty? t) pos]
   [else
    (define new-pos (enumerate (tree-left t) pos))
    (printf "~a ~a\n" new-pos (tree-val t))
    (enumerate (tree-right t) (+ 1 new-pos))]))
```

# Tree Node Enumerate: Tail Form

```
(define (enumerate* t pos todos)
  (cond
   [(empty? t) (perform-todos pos todos)]
   [else
    (enumerate* (tree-left t)
                pos
                (cons t todos))]))

(define (perform-todos pos todos)
  (cond
   [(empty? todos) pos]
   [else
    (define t (first todos))
    (printf "~a ~a\n" pos (tree-val t))
    (enumerate* (tree-right t)
                (+ 1 pos)
                (rest todos))]))
```

# Tree Increment: Non-Tail Form

```
(define (tree-inc t)
  (cond
   [(empty? t) #f]
   [else
    (make-tree (tree-inc (tree-left t))
               (+ 1 (tree-val t))
               (tree-inc (tree-right t)))]))
```

# Tree Increment: Tail Form

```
(define-struct got-left (t))
(define-struct got-right (t left-t))

(define (tree-inc* t todos)
  (cond
   [(empty? t) (perform-todos empty todos)]
   [else
    (tree-inc* (tree-left t)
               (cons (make-got-left t)
                     todos))]))
```

# Tree Increment: Tail Form (Continued)

```
(define (perform-todos new-t todos)
  (cond
   [(empty? todos) new-t]
   [else
    (define todo (first todos))
    (cond
     [(got-left? todo)
      (define t (got-left-t todo))
      (tree-inc* (tree-right t)
                 (cons (make-got-right t new-t)
                       (rest todos)))]
     [(got-right? todo)
      (define t (got-right-t todo))
      (define left-t (got-right-left-t todo))
      (perform-todos (make-tree left-t
                                (+ 1 (tree-val t))
                                new-t)
                     (rest todos))])]))
```

# Tree Increment: Tail Form

```
; A todo is either
;   - empty
;   - (make-got-left tree todo)
;   - (make-got-right tree tree todo)
(define-struct got-left (t rest))
(define-struct got-right (t left-t rest))

(define (tree-inc* t todo)
  (cond
   [(empty? t) (perform-todo empty todo)]
   [else
    (tree-inc* (tree-left t)
               (make-got-left t
                              todo))]))
```

# Tree Increment: Tail Form (Continued)

```
(define (perform-todo new-t todo)
  (cond
   [(empty? todo) new-t]
   [(got-left? todo)
    (define t (got-left-t todo))
    (tree-inc* (tree-right t)
               (make-got-right
                t
                new-t
                (got-left-rest todo)))]
   [(got-right? todo)
    (define t (got-right-t todo))
    (define left-t (got-right-left-t todo))
    (perform-todo (make-tree left-t
                             (+ 1 (tree-val t))
                             new-t)
                  (got-right-rest todo))]))
```

# Tree Increment by Depth: Non-Tail Form

```
(define (tree-dinc t d)
  (cond
   [(empty? t) #f]
   [else
    (make-tree (tree-dinc (tree-left t)
                          (+ d 1))
               (+ d (tree-val t))
               (tree-dinc (tree-right t)
                          (+ d 1)))]))
```

Both accumulator and results ⇒ general case

# Tree Increment by Depth: Tail Form

```
; A todo is either
;   - empty
;   - (make-got-left tree num todo)
;   - (make-got-right tree num tree todo)
(define-struct got-left (t d rest))
(define-struct got-right (t d left-t rest))

(define (tree-dinc* t d todo)
  (cond
   [(empty? t) (perform-todo empty todo)]
   [else
    (tree-dinc* (tree-left t)
                (+ d 1)
                (make-got-left t
                               d
                               todo))]))
```

# Tree Increment by Depth: Tail Form (Continued)

```
(define (perform-todo new-t todo)
  (cond
   [(empty? todo) new-t]
   [(got-left? todo)
    (define t (got-left-t todo))
    (define d (got-left-d todo))
    (tree-dinc* (tree-right t)
                (+ d 1)
                (make-got-right
                 t d new-t
                 (got-left-rest todo)))]
   [(got-right? todo)
    (define t (got-right-t todo))
    (define d (got-right-d todo))
    (define left-t (got-right-left-t todo))
    (perform-todo (make-tree left-t
                             (+ d (tree-val t))
                             new-t)
                  (got-right-rest todo))]))
```

# Tree Search: Non-Tail Form

```
(define (tree-search t s)
  (cond
   [(empty? t) #f]
   [else
    (or (equal? s (tree-val t))
        (tree-search (tree-left t) s)
        (tree-search (tree-right t) s))]))
```

# Tree Search: Tail Form

```
; A todo is either
;  - empty
;  - (make-try-right tree any todo)
(define-struct try-right (t s rest))

(define (tree-search* t s todo)
  (cond
   [(empty? t) (perform-todo todo)]
   [else
    (if (equal? s (tree-val t))
        #t ; better than the original!
        (tree-search* (tree-left t)
                      s
                      (make-try-right
                       (tree-right t)
                       s
                       todo)))]))
```

# Tree Increment by Depth: Tail Form (Continued)

```
(define (perform-todo todo)
  (cond
   [(empty? todo) #f]
   [(try-right? todo)
    (tree-search* (try-right-t todo)
                  (try-right-s todo)
                  (try-right-rest todo))]))
```