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)]
   ....))
```

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)]
   ....))
```

In C, these recursive calls to `evaluate` will be bad...

Print List: Tail Form

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

Tail form ⇒ No problem for C

Print Reverse: Non-Tail Form

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

Non-tail form ⇒ need a “todo” stack

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))]))
```

Converting to Tail Recursion

- Add a “todo” accumulator to the program
- Convert a non-tail call to push onto the “todo” stack
- When not recurring, call **`perform-todos`**
- Define **`perform-todos`** to continue with the first “todo” item

Enumerate: Tail Form

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

Tail form ⇒ No problem for C

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))]))
```

Non-tail form ⇒ need a “todo” stack

Each “todo” has a number and string

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))]))
```

“Todo” Records

- Before a non-tail call, package *all* needed data into a “todo” value
- Have `perform-todos` unpack a “todo” item

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)))]))
```

Tail form ⇒ No problem for C

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))]))
```

Non-tail form ⇒ need a “todo” stack

Each “todo” has a string and *recursion result*

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

Recursion Results

If the non-tail program used the recursion result:

- Define **`perform-todos`** to accept the value
- Pass a “result” value to each call to **`perform-todos`**

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))]))
```

Non-tail form ⇒ need a “todo” stack

Each “todo” needs `t`

Performing a “todo” calls back to `print-tree`

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))]))
```

Tree Recursion

- Multiple recursive calls means that `perform-todo` calls back to the main function

Improved Notation

`struct` and **`match`**

Print Tree: Non-Tail Form

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

(define (print-tree t)
  (match t
   ['() (void)]
   [(tree l v r)
    (print-tree l)
    (displayln v)
    (print-tree r)]))
```

Non-tail form ⇒ need a “todo” stack

Each “todo” needs `v` and `r`

Print Tree: Tail Form

```
(struct finish-node (v r todo))

(define (print-tree* t todo)
  (match t
   ['() (perform-todos todo)]
   [(tree l v r)
    (print-tree* l (finish-node v r todo))]))

(define (perform-todo todo)
  (match todo
   ['() (void)]
   [(finish-node v r todo)
    (displayln v)
    (print-tree* r todo)]))
```

“Todo” Structures

- Use a struct instead of a list to hold multiple pieces in a “todo” record
- Then, you might as well build the list structure into the record

Print with Depth: Non-Tail Form

```
(define (print-tree+depth t d)
  (match t
   ['() (void)]
   [(tree l v r)
    (print-tree+depth l (+ d 1))
    (printf "~a ~a\n" d v)
    (print-tree+depth r (+ d 1))]))
```

Non-tail form ⇒ need a “todo” stack

Each “todo” needs **`v`**, **`r`**, and **`d`**

Print with Depth: Tail Form

```
(struct finish-node (v r todo))

(define (print-tree+depth* t d todo)
  (match t
   ['() (perform-todo todo)]
   [(tree l v r)
    (print-tree+depth* l
                       (+ 1 d)
                       (finish-node v r d todo))]))

(define (perform-todos todos)
  (match todo
   ['() (void)]
   [(finish-node v r d todo)
    (printf "~a ~a\n" d v)
    (print-tree+depth* r (+ 1 d) todo)]))
```

Tree Node Enumerate: Non-Tail Form

```
(define (enumerate t pos)
  (match t
   ['() pos]
   [(tree l v r)
    (define new-pos (enumerate l pos))
    (printf "~a ~a\n" new-pos v)
    (enumerate r (+ 1 new-pos))]))
```

Non-tail form ⇒ need a “todo” stack

Each “todo” needs `v`, `r`, and recursion result

Tree Node Enumerate: Tail Form

```
(define (enumerate* t pos todo)
  (match t
   ['() (perform-todo pos todo)]
   [(tree l v r)
    (enumerate* l pos
                (finish-node v r todo))]))

(define (perform-todo pos todo)
  (match todo
   ['() pos]
   [(finish-node v r todo)
    (printf "~a ~a\n" pos v)
    (enumerate* r (+ 1 pos) todo)]))
```

Tree Increment: Non-Tail Form

```
(define (tree-inc t)
  (match t
   ['() '()]
   [(tree l v r)
    (tree (tree-inc l)
          (+ 1 v)
          (tree-in r))]))
```

Non-tail form ⇒ need a “todo” stack

First “todo” needs **`v`**, **`r`**, and recursion result

Second “todo” needs both a saved result, **`v`**, and recursion result

Tree Increment: Tail Form

```
(struct node-right (v r todo))
(struct finish-node (l v todo))

(define (tree-inc* t todo)
  (match t
    ['() (perform-todo '() todo)]
    [(tree l v r)
     (tree-inc* l
                (node-right v r todo))]))

(define (perform-todo new-t todo)
  (match todo
    ['() new-t]
    [(node-right v r todo)
     (tree-inc* r (finish-node new-t (+ 1 v) todo))]
    [(finish-node l v todo)
     (perform-todo (tree l v new-t) todo)]))
```

Multiple Non-Tail Calls

- Multiple non-tail require multiple “todo” variants

Tree Increment by Depth: Non-Tail Form

```
(define (tree-dinc t d)
  (match t
   ['() '()]
   [(tree l v r)
    (make-tree (tree-dinc l (+ d 1))
               (+ d v)
               (tree-dinc r (+ d 1)))]))
```

Both accumulator and results ⇒ general case

Tree Increment: Tail Form

```
(struct node-right (v r d todo))
(struct finish-node (l v todo))

(define (tree-dinc* t d todo)
  (match t
    ['() (perform-todo '() todo)]
    [(tree l v r)
     (tree-dinc* l
                 (node-right v r todo))]))

(define (perform-todo new-t todo)
  (match todo
    ['() new-t]
    [(node-right v r d todo)
     (tree-dinc* r (+ d 1)
                 (finish-node new-t (+ d v) todo))]
    [(finish-node l v todo)
     (perform-todo (tree l v new-t) todo)]))
```

Tree Search: Non-Tail Form

```
(define (tree-search t s)
  (match t
   ['() #f]
   [(tree l v r)
    (or (equal? s v)
        (tree-search l s)
        (tree-search r s))]))
```

Tree Search: Tail Form

```
(struct try-right (r s rest))

(define (tree-search* t s todo)
  (match t
   ['() (perform-todo todo)]
   [(tree l v r)
    (if (equal? s v)
        #t ; better than the original!
        (tree-search* l
                      s
                      (try-right r s todo)))]))

(define (perform-todo todo)
  (match todo
    ['() #f]
    [(try-right r s todo)
     (tree-search* r s todo)]))
```

MiniRacket

See **`miniracket3a.rkt`** for **`struct`** and **`match`** conversion

See **`miniracket4.rkt`** for “todo” conversion

See **`miniracket4a.rkt`** for better abstraction