Recursion and Iteration

• We've mentioned before that Scheme uses recursion where most other languages (such as Python) use special iterative constructs.

• This puts a special burden on Scheme interpreters to handle iterative recursions, known as tail recursions well.

• From the reference manual:

  "Implementations of Scheme must be properly tail-recursive. Procedure calls that occur in certain syntactic contexts called tail contexts are tail calls. A Scheme implementation is proper tail-recursive if it supports an unbounded number of simultaneously active tail calls."

Tail Contexts

• Tail contexts are defined inductively (or recursively). The "bases" are:

  (lambda (ARGUMENTS) EXPR1 EXPR2 ... EXPRn)

  (define (NAME ARGUMENTS) EXPR1 EXPR2 ... EXPRn)

  ('EXPR)

  (cond (COND-EXPR1 EXPR11 EXPR12 ... EXPR1n)
        (COND-EXPR2 EXPR21 EXPR22 ... EXPR2n) ...)

  (and EXPR1 ... EXPRn)

  (or EXPR1 ... EXPRn)

  (begin EXPR1 ... EXPRn)

• If an expression is in a tail context, then certain parts of it become tail contexts all by themselves:

  (else EXPR)

  (cond (EXPR1 EXPR11 EXPR12 ... EXPR1n)
        (EXPR2 EXPR21 EXPR22 ... EXPR2n) ...)

  (if EXPR THEN-EXPR ELSE-EXPR)

• Tail contexts are defined one particular way.

  • These decrease the memory devoted to keeping track of which function expressions involved in each function call are tail-contexts.

  • This eliminates need for simply retaining expressions involved in each function call.

  • In effect, Scheme turns recursive calls into iterations by replacing those calls with one of the function's tail-context expressions instead of simply returning.

  • A function is tail-recursive if all of the function's tail-contexts result in a recursive call on that same function as a tail-context.

  • In a language like Scheme where expressions can be tail-contexts all by themselves, this is a powerful tool for implementing recursive functions in a tail-context.

• Goals:

  • Keep the number of expressions stored in a call to that function.

  • Keep track of all function expressions that are tail-contexts.

  • Recursion and Iteration

Prime Numbers

(define (prime? x)
  "True iff X is prime."
  (cond ((< x 2) #f)
        ((= x 2) #t)
        (#t ?))

• This decreases the memory used by keeping track of which functions expressions involved in each function call are tail-contexts.

• Tail contexts are used to implement recursive calls by replacing those calls with one of the function's tail-context expressions instead of simply returning.

• A function is tail-recursive if all of the function's tail-contexts result in a recursive call on that same function as a tail-context.

• In a language like Scheme where expressions can be tail-contexts all by themselves, this is a powerful tool for implementing recursive functions in a tail-context.
Prime Numbers

(define (prime? x)
  "True iff X is prime."
  (define (no-factor? k lim)
    "LIM has no divisors >= K and < LIM."
    (cond ((< x 2) #f)
          ((= x 2) #t)
          (#t (no-factor? (+ k 1) lim))))
  (cond ((< x 2) #f)
        ((= x 2) #t)
        (#t (no-factor? 2 (floor (+ (sqrt x) 2))))))

Tail-Recursive Length?

On several occasions, we've computed the length of a linked list like this:

;; The length of list L
(define (length L)
  (if (eqv? L '()) 0 (+ 1 (length (cdr L)))))

but this is not tail recursive. How do we make it so?

Try a helper method:

;; The length of list L
(define (length L)
  (define (length+ ?)
    (length+ ?))
  (length+ L))
Tail-Recursive Length?

On several occasions, we've computed the length of a linked list like this:

;; The length of list L
(define (length L)
  (if (eqv? L '()) ; Alternative: (null? L)
      0
      (+ 1 (length (cdr L)))))

but this is not tail recursive. How do we make it so?

Try a helper method:

;; The length of list L
(define (length L)
  (define (length+ n R)
    ; "The length of R plus N."
    (if (null? R) n
      (length+ (+ n 1) (cdr R))))

(length+ 0 L))

Standard List Searches: assoc, etc.

The functions assq, assv, and assoc classically serve the purpose of Python dictionaries. An association list is a list of key/value pairs. The Python dictionary 

{1: 5, 3: 6, 0: 2} might be represented

((1 . 5) (3 . 6) (0 . 2))

The functions access this list, returning the pair whose car matches a key argument.

The difference between the methods is that
– assq compares using eq? (Python is).
– assv uses eqv? (which is like Python == on numbers and like is otherwise).
– assoc uses equal? (does "deep" comparison of lists).

;; The first item in L whose car is eqv? to key, or #f if none.
(define (assv key L)
  (cond ((null? L) #f)
        ((eqv? key (caar L)) (car L))
        (else (assv key (cdr L))))

This is a tail-recursive function.

Why caar (car (car ...))?
– L has the form ((key1 . val1) (key2 . val2) ...).
– So the car of L is (key1 . val1), and its key is therefore (car (car L)) (or caar for short).

A classic: reduce

;; Assumes f is a two-argument function and L is a list.
;; If L is (x1 x2...xn), the result of applying f n-1 times
to give (f (f (... (f x1 x2) x3) x4) ...).
;; If L is empty, returns f with no arguments.
;; [Simply Scheme version.]
(define (reduce f L)
  (cond ((null? L) (f)) ; Odd case with no items
        ((null? (cdr L)) (car L)) ; One item
        (else (reduce f (cons (f (car L) (cadr L)) (cddr L)))))))

E.g.: (reduce + '(2 3 4))
-calls-> (reduce + (5 4))
-calls-> (reduce + (9))
-yields-> 9
Reduce Solution (2)

Assumes \( f \) is a two-argument function and \( L \) is a list. If \( L \) is \((x_1 x_2 \ldots x_n)\), the result of applying \( f \) \( n-1 \) times to give \((f (f \ldots (f x_1 x_2) x_3) x_4) \ldots\). If \( L \) is empty, returns \( f \) with no arguments.

\[
\begin{align*}
\text{define } & \text{reduce } f \ L \\
& \text{define } \text{reduce-tail } \text{accum } R \\
& \quad \text{cond} \\
& \quad \quad \text{if } \text{null? } R \text{ then } \text{accum} \\
& \quad \quad \quad \text{else } \text{reduce-tail } (f \ \text{accum} \ \text{car } R) \ \text{cdr } R)
\end{align*}
\]

Another Example

We've seen \texttt{map} many times.

An obvious recursive solution:

\[
\begin{align*}
\text{define } & \text{map } f \ L \\
& \text{if } \text{null? } L \text{ then } () \\
& \quad \text{else } \text{cons } (f \ \text{car } L) \ \text{map } f \ \text{cdr } L)
\end{align*}
\]

Making map tail recursive

Need to pass along the partial results and add to them.

Problem: \texttt{cons} adds to the front of a list, so we end up with a reverse of what we want.

\[
\begin{align*}
\text{define } & \text{map+ } \text{partial-result } \text{rest} \\
& \quad \text{if } \text{null? } \text{rest} \text{ then } \text{partial-result} \\
& \quad \quad \text{else } \text{map+ } (\text{cons } (f \ \text{car } \text{rest}) \ \text{partial-result}) \ \text{cdr } \text{rest})
\end{align*}
\]

\[
\text{reverse } \text{map+ } () \ \text{L}
\]

Another Example

Consider the problem of shuffling together two lists, \( L_1 \) and \( L_2 \). The result consists of the first element of \( L_1 \), then the first of \( L_2 \), then the second of \( L_1 \), etc., until one or the other list has no more values.

Obvious recursive solution:

\[
\begin{align*}
\text{define } & \text{shuffle1 } L_1 \ L_2 \\
& \text{if } \text{null? } L_1 \text{ then } () \\
& \quad \text{else } \text{cons } (\text{car } L_1) \ \text{shuffle1 } L_2 \ \text{cdr } L_1)
\end{align*}
\]

And Finally, Reverse

Actually, we can use the very problem that \texttt{cons} creates to solve it! That is, consing items from a list from left to right results in a reversed list:

\[
\begin{align*}
\text{define } & \text{reverse } L \\
& \text{define } \text{reverse+ } \text{partial-result } \text{rest} \\
& \quad \text{if } \text{null? } \text{rest} \text{ then } \text{partial-result} \\
& \quad \quad \text{else } \text{reverse+ } (\text{cons } (\text{car } \text{rest}) \ \text{partial-result}) \ \text{cdr } \text{rest})
\end{align*}
\]

\[
\text{reverse+ } () \ \text{L}
\]

Another Example

Consider the problem of shuffling together two lists, \( L_1 \) and \( L_2 \). The result consists of the first element of \( L_1 \), then the first of \( L_2 \), then the second of \( L_1 \), etc., until one or the other list has no more values.

Obvious recursive solution:

\[
\begin{align*}
\text{define } & \text{shuffle1 } L_1 \ L_2 \\
& \text{if } \text{null? } L_1 \text{ then } () \\
& \quad \text{else } \text{cons } (\text{car } L_1) \ \text{shuffle1 } L_2 \ \text{cdr } L_1)
\end{align*}
\]

What about \texttt{reverse}?

\[
\begin{align*}
\text{define } & \text{reverse+ } \text{partial-result } \text{rest} \\
& \quad \text{if } \text{null? } \text{rest} \text{ then } \text{partial-result} \\
& \quad \quad \text{else } \text{reverse+ } (\text{cons } (\text{car } \text{rest}) \ \text{partial-result}) \ \text{cdr } \text{rest})
\end{align*}
\]

\[
\text{reverse+ } () \ \text{L}
\]

Another Example

Obvious recursive solution:

\[
\begin{align*}
\text{define } & \text{shuffle1 } L_1 \ L_2 \\
& \text{if } \text{null? } L_1 \text{ then } () \\
& \quad \text{else } \text{cons } (\text{car } L_1) \ \text{shuffle1 } L_2 \ \text{cdr } L_1)
\end{align*}
\]

Problem: \texttt{cons} adds to the front of a list, so we end up with a reverse of what we want.

Make sure to pass along the partial results and add to them.

Another Example

Consider the problem of shuffling together two lists, \( L_1 \) and \( L_2 \). The result consists of the first element of \( L_1 \), then the first of \( L_2 \), then the second of \( L_1 \), etc., until one or the other list has no more values.

Obvious recursive solution:

\[
\begin{align*}
\text{define } & \text{shuffle1 } L_1 \ L_2 \\
& \text{if } \text{null? } L_1 \text{ then } () \\
& \quad \text{else } \text{cons } (\text{car } L_1) \ \text{shuffle1 } L_2 \ \text{cdr } L_1)
\end{align*}
\]
Another Example

Consider the problem of shuffling together two lists, L1 and L2. The result consists of the first item of L1, then the first of L2, then the second of L1, etc., until one or the other list has no more values.

Obvious recursive solution:

(define (shuffle1 L1 L2)
  "The list consisting of the first element of L1, then the first of L2, then the second of L1, etc., until the elements of one or the other list is exhausted."
  (if (null? L1) '()
      (cons (car L1) (shuffle1 L2 (cdr L1))))
)

Not tail recursive. Again, we can use a helper method:

(define (shuffle L1 L2)
  (define (shuffle+ reversed-result L1 L2)
    (if (null? L1) (reverse reversed-result)
        (shuffle+ (cons (car L1) reversed-result) L2 (cdr L1))))
  (shuffle+ '() L1 L2))

Another Example

Consider the problem of shuffling together two lists, L1 and L2. The result consists of the first item of L1, then the first of L2, then the second of L1, etc., until one or the other list has no more values.

Obvious recursive solution:

(define (shuffle1 L1 L2)
  "The list consisting of the first element of L1, then the first of L2, then the second of L1, etc., until the elements of one or the other list is exhausted."
  (if (null? L1) '()
      (cons (car L1) (shuffle1 L2 (cdr L1))))
)

Not tail recursive. Again, we can use a helper method:

(define (shuffle L1 L2)
  (define (shuffle+ reversed-result L1 L2)
    (if (null? L1) (reverse reversed-result)
        (shuffle+ (cons (car L1) reversed-result) L2 (cdr L1))))
  (shuffle+ '() L1 L2))

Another Example

Consider the problem of shuffling together two lists, L1 and L2. The result consists of the first item of L1, then the first of L2, then the second of L1, etc., until one or the other list has no more values.

Obvious recursive solution:

(define (shuffle1 L1 L2)
  "The list consisting of the first element of L1, then the first of L2, then the second of L1, etc., until the elements of one or the other list is exhausted."
  (if (null? L1) '()
      (cons (car L1) (shuffle1 L2 (cdr L1))))
)

Not tail recursive. Again, we can use a helper method:

(define (shuffle L1 L2)
  (define (shuffle+ reversed-result L1 L2)
    (if (null? L1) (reverse reversed-result)
        (shuffle+ (cons (car L1) reversed-result) L2 (cdr L1))))
  (shuffle+ '() L1 L2))