

Computer Science I

Professor Tom Ellman
Lecture 10

Reversing a List

```
(define reverse (lambda (lst) ...?...))
```

```
Welcome to DrScheme.  
> (reverse '(a b c))  
(c b a)  
> (reverse '(a))  
(a)  
> (reverse '())  
()
```

The REVERSE Procedure

```
(define reverse  
  (lambda (lst)  
    (if (null? lst)  
        '()  
        (snoc (reverse (cdr lst)) (car lst))))))
```

Putting Something at the End of a List

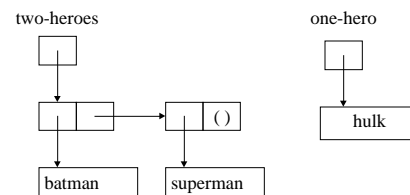
```
(define snoc (lambda (lst item) ...?...))
```

```
Welcome to DrScheme.  
> (snoc '() 'z)  
(z)  
> (snoc '(y) 'z)  
(y z)  
> (snoc '(x y) 'z)  
(x y z)
```

Behavior of SNOC

```
Welcome to DrScheme.  
> (define one-hero 'hulk)  
> (define two-heroes '(batman superman))  
> (define three-heroes (snoc two-heroes one-hero))  
> three-heroes  
(batman superman hulk)  
> two-heroes  
(batman superman)
```

How to Implement SNOC?



three-heroes

two-heroes

one-hero

batman

superman

hulk

- Suppose we just create one new CONS cell and insert two new pointers as shown in the diagram above?
- The variables ONE-HERO and THREE-HEROES have the right values, but TWO-HEROES now has a new (wrong) value!

The SNOC Procedure

```

(define snoc
  (lambda (lst item)
    (if (null? lst)
        (cons item '())
        (cons (car lst) (snoc (cdr lst) item)))))

```

- Scheme makes a copy of the first argument to SNOC.
- In this case, Scheme will copy the list (batman superman).

two-heroes

three-heroes

one-hero

batman

superman

batman

superman

hulk

Discussion of the REVERSE Procedure Definition using SNOC

- It gives the right answer.
- It does a lot of extra, unnecessary work.
- REVERSE calls SNOC over and over.
- Each time Scheme evaluates (snoc X Y), Scheme makes a copy of the entire list X.

Trace of Evaluation of REVERSE

```

(reverse '(a b c))
(snoc (reverse '(b c)) 'a)
(snoc (snoc (reverse '(c)) 'b) 'a)
(snoc (snoc (snoc '() 'c) 'b) 'a)
(snoc (snoc '(c) 'b) 'a)
(snoc '(c b) 'a)
(c b a)

```

Executing Our REVERSE Procedure

- Suppose Scheme reverses (a b c).
- Scheme makes copies of (c b), (c) and ().
- Suppose Scheme reverses (a b c d e).
- Scheme makes copies of (e d c b), (e d c), (e d), (e) and ().
- ...Etc...
- Parts of the reversed list get copied many times.

A New Approach to REVERSE

```
(define reverse
  (lambda (lst) (reverse-helper lst '())))

(define reverse-helper
  (lambda (lst answer)
    (if (null? lst)
        answer
        (reverse-helper (cdr lst)
                        (cons (car lst) answer)))))
```

What is REVERSE-HELPER?

- It takes two arguments LST and ANSWER.
- It returns: (APPEND (REVERSE LST) ANSWER).
- It does not actually call the APPEND procedure.
- Instead it takes the CAR of LST and uses CONS to put it at the beginning of ANSWER.
- It calls itself recursively on (CDR LST) and (CONS (CAR LST) ANSWER).

Trace of Evaluation of New REVERSE Procedure

```
(reverse '(a b c))
(reverse-helper '(a b c) '())
(reverse-helper '(b c) '(a))
(reverse-helper '(c) '(b a))
(reverse-helper '() '(c b a))
(c b a)
```

The Accumulator Method

- A recursive function has a special parameter called the “accumulator”.
- Each time the function calls itself recursively, it augments the accumulator.
 - E.g. CONSing something into an accumulator list.
 - E.g. Adding something to an accumulator number.
- When recursion stops, the accumulator has the final answer.

Summing the Numbers on a List using the Accumulator Method

```
(define sum (lambda (lst) ...?...(sum-helper ...?...)?...))
(define sum-helper (lambda (...?...)?...))
```

Welcome to DrScheme.

```
> (sum '(1 2 3))
```

```
6
```

```
> (sum '(7))
```

```
7
```

```
> (sum '())
```

```
0
```

Summing the Numbers on a List using the Accumulator Method

```
(define sum
  (lambda (lst) (sum-helper lst 0)))

(define sum-helper
  (lambda (lst N)
    (if (null? lst)
        N
        (sum-helper (cdr lst) (+ (car lst) N)))))
```

Trace of Evaluation of New SUM Procedure

```
(sum '(1 2 3 4 5))
(sum-helper '(1 2 3 4 5) 0)
(sum-helper '(2 3 4 5) 1)
(sum-helper '(3 4 5) 3)
(sum-helper '(4 5) 6)
(sum-helper '(5) 10)
(sum-helper '() 15)
15
```

Finding the Decimal Representation of a Number

```
(define digits (lambda (n) ...?...))

Welcome to DrScheme.
> (digits 1066)
(1 0 6 6)
> (digits 1492)
(1 4 9 2)
> (digits 10)
(1 0)
> (digits 7)
(7)
> (digits 0)
(0)
```

Quotient and Remainder

```
Welcome to DrScheme.
> (quotient 1492 10)
149
> (remainder 1492 10)
2
> (quotient 149 10)
14
> (remainder 149 10)
9
```

The DIGITS Procedure with an Accumulation Parameter

```
(define digits
  (lambda (n) (digits-helper n '())))

(define digits-helper
  (lambda (n lst)
    (if (= n 0)
        lst
        (digits-helper (quotient n 10)
                        (cons (remainder n 10) lst)))))
```

Trace of Evaluation of DIGITS Procedure

```
(digits 1492)
(digits-helper 1492 '())
(digits-helper 149 '(2))
(digits-helper 14 '(9 2))
(digits-helper 1 '(4 9 2))
(digits-helper 0 '(1 4 9 2))
(1 4 9 2)
```

Converting a List of Decimal Digits into a Number

```
(define value (lambda (lst) ...?...))

Welcome to DrScheme.
> (value '(1 0 6 6))
1066
> (value '(1 4 9 2))
1492
> (value '(7))
7
> (value '())
0
```

The Value Procedure with an Accumulation Parameter

```
(define value
  (lambda (lst) (value-helper lst 0)))

(define value-helper
  (lambda (lst answer)
    (if (null? lst)
        answer
        (value-helper (cdr lst)
                      (+ (car lst) (* 10 answer))))))
```

Trace of Evaluation of VALUE Procedure

```
(value 1492)
(value-helper '(1 4 9 2) 0)
(value-helper '(4 9 2) 1)
(value-helper '(9 2) 14)
(value-helper '(2) 149)
(value-helper '() 1492)
1492
```