

## Computer Science I

Professor Tom Ellman  
Lecture 24

## Is there any method in this madness?

- We have written lots of Scheme programs.
- Often it seems that we must pull them out of thin air.
- Is there a systematic method for *developing* program?
- No, but there is a systematic method of *classifying* the programs we have written.

## Patterns of Mapping

- We have studied two patterns of mapping:
  - Flat mapping.
  - Deep mapping.
- Each of these patterns can be implemented as a higher order procedure.
- Any specific example of flat or deep mapping can be implemented by supplying suitable arguments to the higher order procedure.

## Flat Mapping

```
(define map-flat
  (lambda (fn lst)
    (if (null? lst)
        '()
        (cons (fn (car lst)) (map-flat fn (cdr lst))))))
```

In order to apply flat mapping to a problem, we need only to choose a procedure FN.

## Deep Mapping

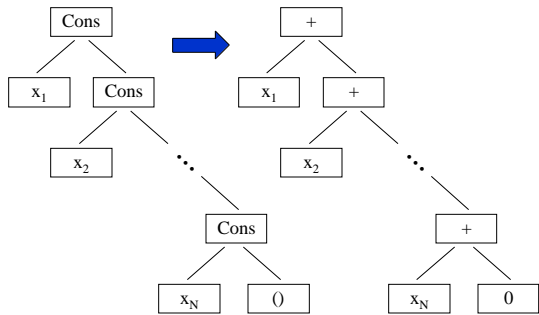
```
(define map-deep
  (lambda (fn tree)
    (if (not (pair? tree))
        (fn tree)
        (cons (map-deep fn (car tree))
              (map-deep fn (cdr tree))))))
```

In order to apply deep mapping to a problem, we need only to choose a procedure FN.

## Patterns of Reduction

- We have studied two patterns of reduction:
  - Flat reduction.
  - Deep reduction.
- Each of these patterns can be implemented as a higher order procedure.
- Any specific example of flat or deep reduction can be implemented by supplying suitable arguments to the higher order procedure.

## SUM as List Morphism



## Flat Reduction

```
(define reduce-flat
  (lambda (lst fun base)
    (if (null? lst)
        base
        (fun (car lst) (reduce-flat (cdr lst) fun base))))))
```

In order to apply flat reduction to a problem, we need only to choose a procedure FUN and a value BASE.

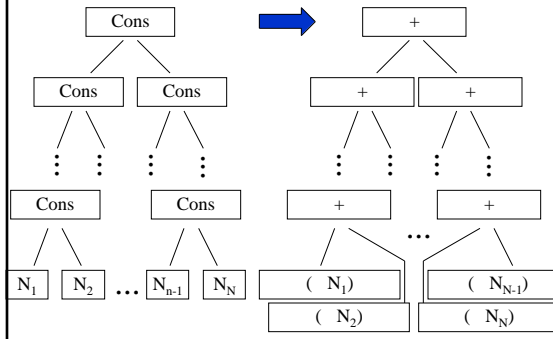
## Flat Reduction Examples

```
(define sum (lambda (lst)
  (reduce-flat lst
    +
    0)))

(define length (lambda (lst)
  (reduce-flat lst
    (lambda (x y) (+ 1 y))
    0)))
```

We can see precisely how sum and length are similar to and different from each other.

## ITEM-SUM as Tree Morphism



## Deep Reduction

```
(define reduce-deep
  (lambda (lst ind-fun base-fun)
    (if (not (pair? lst))
        (base-fun lst)
        (ind-fun (reduce-deep (car lst) ind-fun base-fun)
          (reduce-deep (cdr lst) ind-fun base-fun)))))
```

In order to apply deep reduction to a problem, we need only to choose procedures IND-FUN and BASE-FUN.

## Deep Reduction Examples

```
(define item-count
  (lambda (lst)
    (reduce-deep lst
      +
      (lambda (x) (if (null? x) 0 1)))))

(define item-sum
  (lambda (lst)
    (reduce-deep lst
      +
      (lambda (x) (if (null? x) 0 x)))))
```

We can see precisely how item-count and item-sum are similar to and different from each other.

## Relating Mapping Patterns to Reduction Patterns

- Flat and deep mapping are special cases of flat and deep reduction.
  - Flat mapping is flat reduction of the CONS function with the empty list as the base value.
  - Deep mapping is deep reduction of the CONS function.
- Flat and deep mapping can be implemented using the higher order procedures for flat and deep reduction.

## Defining Flat and Deep Mapping in Terms of Flat and Deep Reduction

```
(define map-flat
  (lambda (fn lst)
    (reduce-flat lst
      (lambda (x y) (cons (fn x) y))
      '())))

(define map-deep
  (lambda (fn tree) (reduce-deep tree cons fn)))
```

## Relating Mapping Patterns to Reduction Patterns

- Flat mapping is flat reduction that uses CONS to rebuild the list with mapped values, starting with the empty list.
- Deep mapping is deep reduction that uses CONS to rebuild the tree with mapped values.

## Patterns of Recursion

- We have studied two patterns of recursion:
  - Flat recursion: (“Cdr Recursion”).
  - Deep recursion: (“Car-Cdr Recursion”).
- Each of these patterns can be implemented as a higher order procedure.
- Any specific example of flat or deep recursion can be implemented by supplying suitable arguments to the higher order procedure.

## Flat Recursion

```
(define recur-flat
  (lambda (lst ind-fun base-fun stop?)
    (if (stop? lst)
        (base-fun lst)
        (ind-fun (car lst)
                  (recur-flat (cdr lst) ind-fun base-fun))))))
```

In order to apply flat recursion to a problem, we need only to choose the procedures IND-FUN, BASE-FUN and STOP?.

## Flat Recursion Examples

```
(define sum
  (lambda (lst)
    (recur-flat lst + (lambda (x) 0) null?)))

(define length
  (lambda (lst)
    (recur-flat lst
      (lambda (x y) (+ 1 y))
      (lambda (x) 0)
      null?)))
```

## Deep Recursion

```
(define recur-deep
  (lambda (lst ind-fun base-fun stop?)
    (if (stop? lst)
        (base-fun lst)
        (ind-fun (recur-deep (car lst) ind-fun base-fun stop?)
                     (recur-deep (cdr lst) ind-fun base-fun stop?))))))
```

In order to apply deep recursion to a problem, we need only to choose the procedures IND-FUN, BASE-FUN and STOP?.

## Deep Recursion Examples

```
(define item-count
  (lambda (lst)
    (recur-deep lst
                +
                (lambda (x) (if (null? x) 0 1))
                (lambda (x) (not (pair? x))))))
```

```
(define item-sum
  (lambda (lst)
    (recur-deep lst
                +
                (lambda (x) (if (null? x) 0 x))
                (lambda (x) (not (pair? x))))))
```

## Relating Reduction Patterns to Recursion Patterns

- Flat and deep reduction are special cases of flat and deep recursion.
  - Flat reduction is flat recursion with a special stopping condition and a special constant-valued base function.
  - Deep reduction is deep recursion with a special stopping condition.
- Flat and deep reduction can be implemented using the higher order procedures for flat and deep recursion.

## Defining Flat and Deep Reduction in Terms of Flat and Deep Recursion

```
(define reduce-flat
  (lambda (lst fun base)
    (recur-flat lst fun (lambda (x) base) null?)))
```

```
(define reduce-deep
  (lambda (lst ind-fun base-fun)
    (recur-deep lst
                ind-fun
                base-fun
                (lambda (x) (not (pair? x))))))
```

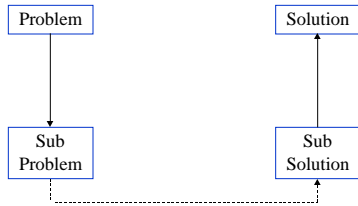
## Relating Reduction Patterns to Recursion Patterns

- Flat reduction is flat recursion that stops when its argument is null, and then returns a constant.
- Deep reduction is deep recursion that stops when its argument is not a pair.

## Generalizing Flat and Deep Recursion

- Flat and deep recursion are special cases of even more general patterns of computation.
- Flat recursion is a special case of a pattern called “Simplify and Conquer”.
- Deep recursion is a special case of a pattern called “Divide and Conquer”.

## Simplify and Conquer



## Simplify and Conquer

```
(define simplify-and-conquer
  (lambda (structure select1 select2 ind-fun base-fun stop?)
    (if (stop? structure)
        (base-fun structure)
        (ind-fun (select1 structure)
                  (simplify-and-conquer (select2 structure)
                                       select1
                                       select2
                                       ind-fun
                                       base-fun
                                       stop?))))))
```

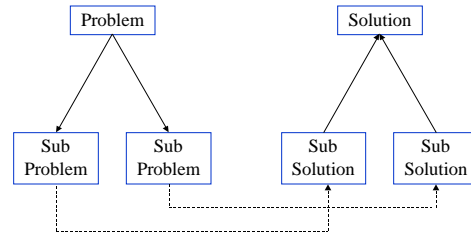
In order to apply simplify-and-conquer to a problem, we need to choose procedures SELECT1, SELECT2, IND-FUN, BASE-FUN and STOP?.

## Simplify and Conquer Examples

```
(define sum
  (lambda (lst)
    (simplify-and-conquer lst car cdr + (lambda (x) 0) null?)))

(define length
  (lambda (lst)
    (simplify-and-conquer lst
                          car
                          cdr
                          (lambda (x y) (+ 1 y))
                          (lambda (x) 0)
                          null?)))
```

## Divide and Conquer



## Divide and Conquer

```
(define divide-and-conquer
  (lambda (structure select1 select2 ind-fun base-fun stop?)
    (if (stop? structure)
        (base-fun structure)
        (ind-fun (divide-and-conquer (select1 structure)
                                     select1 select2
                                     ind-fun base-fun
                                     stop?)
                  (divide-and-conquer (select2 structure)
                                       select1 select2
                                       ind-fun base-fun
                                       stop?))))))
```

In order to apply divide-and-conquer to a problem, we need to choose procedures SELECT1, SELECT2, IND-FUN, BASE-FUN and STOP?.

## Divide and Conquer Example

```
(define item-count
  (lambda (lst)
    (divide-and-conquer lst
                        car
                        cdr
                        +
                        (lambda (x) (if (null? x) 0 1))
                        (lambda (x) (not (pair? x))))))
```

## Divide and Conquer Example

```
(define item-sum
  (lambda (lst)
    (divide-and-conquer lst
      car
      cdr
      +
      (lambda (x) (if (null? x) 0 x))
      (lambda (x) (not (pair? x))))))
```

## Defining Flat/Deep Recursion in Terms Simplify/Divide and Conquer

```
(define recur-flat
  (lambda (lst ind-fun base-fun stop?)
    (simplify-and-conquer lst car cdr ind-fun base-fun stop?)))

(define recur-deep
  (lambda (lst ind-fun base-fun stop?)
    (divide-and-conquer lst car cdr ind-fun base-fun stop?)))
```

## Relating Recursion Patterns to Simplify/Divide and Conquer Patterns

- Flat recursion is simplify-and-conquer in which the selection functions are CAR and CDR.
- Deep recursion is divide-and-conquer in which the selection functions are CAR and CDR.

## Defining the Accumulator Method as a Higher Order Procedure

- Accumulator procedures all involve repeatedly selecting data from a given structure, diminishing the given structure, and using the selected data to augment the accumulator.
- We get a variety of accumulator procedures by choosing functions to SELECT, DIMINISH and test whether the structure is EMPTY?, and by choosing a function to AUGMENT the accumulator.

## Accumulation

```
(define accumulation
  (lambda (structure accumulator empty? select diminish augment)
    (accumulation-helper structure accumulator)))

(define accumulation-helper
  (lambda (structure accumulator empty? select diminish augment)
    (if (empty? structure)
        accumulator
        (accumulation-helper (diminish structure)
                             (augment (select structure)
                                     accumulator)
                             empty?
                             select
                             diminish
                             augment))))
```

## Applying the Accumulator Method to a Problem

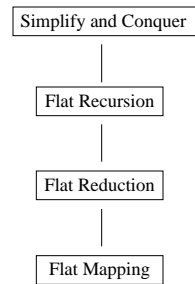
- Choose a procedure EMPTY? that tests whether the given structure is as small as possible.
- Choose a procedure SELECT to take something from given structure.
- Choose a procedure DIMINISH to make the given structure smaller.
- Choose a procedure AUGMENT to make the put something into the accumulator.

## Accumulation Examples

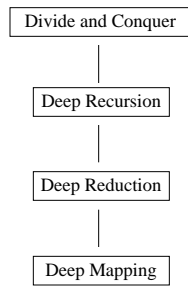
```
(define reverse  
  (lambda (lst)  
    (accumulation lst '() null? car cdr cons)))
```

```
(define digits  
  (lambda (n base)  
    (accumulation n  
      '()  
      zero?  
      (lambda (x) (remainder x base))  
      (lambda (x) (quotient x base))  
      cons)))
```

## Hierarchies of Program Schemes



## Hierarchies of Program Schemes



## Hierarchies of Program Schemes

