Is there any method in this madness?

We've written lots of Scheme programs. Often it seems that we need to pull them out of thin air.

Is there a systematic method for developing programs?

There are general approaches to problem-solving.

We can also classify solutions to problems into general types, as we've begun to see when writing higher-order functions.

A general design recipe

1. Understand the input data.
   What do the inputs mean and what are the allowed values? E.g., if it's a list, can it be nested? Is it always of a particular length? If it's a number, can it be negative?

2. Describe the function.
   Write a short comment describing what the function is supposed to do before you start writing the code.

3. Write a few example cases and what the result should be.
   When the problem statement divides the input into several categories, test each one.

4. Write the function.
   This is the hard part!

5. Test the function on the examples.
   Did the function produce the output you expected? If not, trace through the execution of the function to see why.

A general design recipe

1. Understand the input data.
   What do the inputs mean and what are the allowed values? E.g., if it's a list, can it be nested? Is it always of a particular length? If it's a number, can it be negative?

2. Describe the function.
   Write a short comment describing what the function is supposed to do before you start writing the code.

3. Write a few example cases and what the result should be.
   When the problem statement divides the input into several categories, test each one.

4. Write the function.
   This is the hard part! When you've clearly defined the problem, you can ask yourself if it looks like one of the patterns you've encountered before.

5. Test the function on the examples.
   Did the function produce the output you expected? If not, trace through the execution of the function to see why.
Hierarchies of program schemes

Simplify and Conquer  Divide and Conquer

Most general

Flat Recursion  Deep Recursion

Flat Reduction  Deep Reduction

Flat Mapping  Deep Mapping

Most specific

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

\[
\text{(define map-flat} \rightarrow \text{map-flat)}
\]

\[
\text{(lambda } \text{(fn lst})) \rightarrow \text{lambda)}
\]

\[
\text{(if (null? lst))} \rightarrow \text{if)}
\]

\[
\text{'()} \rightarrow \text{'()}
\]

\[
\text{(cons (fn (first lst))} \rightarrow \text{cons)}
\]

\[
\text{(map-flat fn (rest lst))})}) \rightarrow \text{(map-flat fn (rest lst))})}
\]

In order to apply flat mapping to a problem, we need only to choose a procedure \text{fn}.\]
Deep mapping

\[
\text{map-deep} \equiv \lambda (fn \ tree) \ (\text{cond} \ ((\text{null?} \ tree) \ '()) \ ((\text{not} \ (\text{pair?} \ tree)) \ (fn \ tree) \ (\text{else}) \ (\text{cons} \ (\text{map-deep} \ fn \ (\text{first} \ tree)) \ (\text{map-deep} \ fn \ (\text{rest} \ tree))))))
\]

In order to apply deep mapping to a problem, we need only to choose a procedure \( fn \).

Flat mapping examples

\[
\text{double-all} \equiv \lambda (lst) \ (\text{map-flat} \ (\lambda (x) \ (* \ 2 \ x)) \ lst)))
\]

\[
\text{increment-all} \equiv \lambda (lst) \ (\text{map-flat} \ (\lambda (x) \ (+ \ 1 \ x)) \ lst)))
\]

We can see precisely how \text{double-all} and \text{increment-all} are similar to each other and how they are different.

Hierarchies of program schemes

<table>
<thead>
<tr>
<th>Simplify and Conquer</th>
<th>Divide and Conquer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flat Recursion</td>
<td>Deep Recursion</td>
</tr>
<tr>
<td>Flat Reduction</td>
<td>Deep Reduction</td>
</tr>
<tr>
<td>Flat Mapping</td>
<td>Deep Mapping</td>
</tr>
</tbody>
</table>

Most general

Most specific
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.

Flat reduction

\[
\text{Flat reduction examples}
\]

\[
\begin{align*}
\text{(define sum)} & \quad \text{(lambda (lst))}
\text{ (lambda (lst) (reduce-flat lst + 0)))}
\text{(define length)} & \quad \text{(lambda (lst))}
\text{ (reduce-flat lst (lambda (x y) (+ 1 y)) 0)))}
\end{align*}
\]

In order to apply flat reduction to a problem, we need only to choose a procedure \text{fun} and a value \text{base}.

We can see precisely how \text{sum} and \text{length} are similar to each other and how they are different.
item-sum* as tree morphism

Deep reduction

In order to apply deep reduction to a problem, we need only to choose procedures rec-fun, base-fun, and null-val.

Deep reduction examples

Hierarchies of program schemes

We can see precisely how item-count* and item-sum* are similar to each other and how they are different.
Flat mapping is a special case of flat reduction that uses **cons** to rebuild the list with mapped values, starting with the empty list:

(\texttt{(define map-flat (lambda (fn lst) (reduce-flat lst (lambda (x y) (cons (fn x) y)) '()))})

Deep mapping is a special case of deep reduction that uses **cons** to rebuild the tree (i.e., nested list) with mapped values and '() to replace null values:

(\texttt{(define map-deep (lambda (fn tree) (reduce-deep tree cons fn '()))})

Hierarchies of program schemes

\begin{tabular}{c|c|c}
\textbf{Simplify and Conquer} & \textbf{Divide and Conquer} & \textbf{Most general} \\
\hline
\textbf{Flat Recursion} & \textbf{Deep Recursion} & \\
\textbf{Flat Reduction} & \textbf{Deep Reduction} & \\
\textbf{Flat Mapping} & \textbf{Deep Mapping} & \textbf{Most specific}
\end{tabular}

Patterns of recursion

We have studied two patterns of recursion:

- Flat recursion
- Deep 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 rec-fun base-fun stop?)
    ;; If the stopping condition is true for the input list
    (if (stop? lst)
        ;; Return the base case by applying BASE-FUN to the input list
        ;; (which it might ignore)
        (base-fun lst)
        ;; Otherwise, the recursive case uses REC-FUN to combine the
        ;; first item on the list with the result of the recursive call
        ;; on the rest
        (rec-fun (first lst)
          (recur-flat (rest lst)
                      rec-fun
                      base-fun
                      stop?))))

To apply flat recursion to a problem, we need only to choose the procedures rec-fun, base-fun, and stop?.

Flat recursion examples

(define sum
  (lambda (lst)
    (recur-flat lst
      ;; recursive case function
      +
      ;; base case function
      (lambda (x) 0)
      ;; stopping condition
      null?)))

Flat recursion examples

(define length
  (lambda (lst)
    (recur-flat lst
      ;; recursive case function
      (lambda (x y) (+ 1 y))
      ;; base case function
      (lambda (x) 0)
      ;; stopping condition
      null?)))

Using flat recursion to insert an item into an ordered list

(define insert
  (lambda (n lst)
    (recur-flat ...
      )))

> (insert 3 '(1 2 4))
(1 2 3 4)
Using flat recursion to insert an item into an ordered list

```scheme
(define insert
  (lambda (n lst)
    (recur-flat lst
      ;; recursive case function
      cons
      ;; base case function
      (lambda (listy)
        (cons n listy))
      ;; stopping condition
      (lambda (listy)
        (or (null? listy)
            (<= n (first lst)))))))
```

Using flat recursion to count symbols in a list

```scheme
(define count-symbols
  (lambda (lst)
    (recur-flat ...
      ;; recursive case function
      (lambda (item count)
        (if (symbol? item)
            (+ 1 count)
            count))
      ;; base case function
      (lambda (lst) 0)
      ;; stopping condition
      null?)))
```

Using flat recursion to count symbols in a list

```scheme
(define insert
  (lambda (n lst)
    (recur-flat lst
      ;; recursive case function
      cons
      ;; base case function
      (lambda (listy)
        (cons n listy))
      ;; stopping condition
      (lambda (listy)
        (or (null? listy)
            (<= n (first lst)))))))
```

Returning a flat-recursive function

```scheme
(define recur-flat-fn
  (lambda (rec-fun base-fun stop?)
    (letrec ((recur-flat
                  (lambda (lst)
                    (if (stop? lst)
                        (base-fun lst)
                        (rec-fun (first lst)
                                  (recur-flat (rest lst)))))
                  recur-flat)))
    A flat recursion procedure is built from the procedures rec-fun, base-fun, and stop?.
```
Flat recursion examples revisited

(define \textit{sum} (lambda (lst)
    (recur-flat lst +
        (lambda (x) 0)
        null?))))

(define \textit{sum} (recur-flat-fn + (lambda (x) 0) null?))

Deep recursion

(define \textit{recur-deep} (lambda (lst rec-fun base-fun null-val stop?)
    (cond
        ((null? lst) null-val)
        ((stop? lst) (base-fun lst))
        (else
            (rec-fun (recur-deep (first lst) rec-fun base-fun null-val stop?)
                (recur-deep (rest lst) rec-fun base-fun null-val stop?)))))

In order to apply deep recursion to a problem, we need only to choose the procedures \textit{rec-fun}, \textit{base-fun}, and \textit{stop}.

Flat recursion examples revisited

(define \textit{length} (lambda (lst)
    (recur-flat lst (lambda (x y) (+ 1 y))
        (lambda (x) 0)
        null?))))

(define \textit{length} (recur-flat-fn (lambda (x y) (+ 1 y)) (lambda (x) 0) null?))

Deep recursion examples

(define \textit{item-count*} (lambda (lst)
    (recur-deep lst +
        (lambda (x) 1)
        0
        (lambda (x) (not (pair? x))))))

(define \textit{item-sum*} (lambda (lst)
    (recur-deep lst +
        (lambda (x) x)
        (lambda (x) (not (pair? x))))))
**Returning a deep-recursive function**

```
(define recur-deep-fn
  (lambda (rec-fun base-fun null-val stop?)
    (letrec ((recur-deep
                (lambda (lst)
                  (cond
                    ((null? lst) null-val)
                    ((stop? lst) (base-fun lst))
                    (else
                      (rec-fun (recur-deep (first lst))
                               (recur-deep (rest lst)))))))
     recur-deep)))
```

A deep recursion procedure is built from the procedures \textit{rec-fun}, \textit{base-fun}, \textit{null-val}, and \textit{stop}.

**Deep recursion examples revisited**

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

```
(define item-count*
  (recur-deep-fn +
  (lambda (x) 1)
  0
  (lambda (x) (not (pair? x)))))
```

**Hierarchies of program schemes**

<table>
<thead>
<tr>
<th>Simplify and Conquer</th>
<th>Divide and Conquer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flat Recursion</td>
<td>Deep Recursion</td>
</tr>
<tr>
<td><strong>How do these relate?</strong></td>
<td></td>
</tr>
<tr>
<td>Flat Reduction</td>
<td>Deep Reduction</td>
</tr>
<tr>
<td>Flat Mapping</td>
<td>Deep Mapping</td>
</tr>
</tbody>
</table>

Most general

Most specific
Flat reduction is a special case of flat recursion that stops when its argument is null, and then returns a constant:

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

Deep reduction is a special case of deep recursion that stops when its argument is not a pair:

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

Hierarchies of program schemes

<table>
<thead>
<tr>
<th>Simplify and Conquer</th>
<th>Divide and Conquer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flat Recursion</td>
<td>Deep Recursion</td>
</tr>
<tr>
<td>Flat Reduction</td>
<td>Deep Reduction</td>
</tr>
<tr>
<td>Flat Mapping</td>
<td>Deep Mapping</td>
</tr>
</tbody>
</table>

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

Problem

Solution

Sub

Sub

Simplify and Conquer examples

(define sum
  (lambda (lst)
    (simplify-and-conquer lst first rest
     +
     (lambda (x) 0)
     null?)))

(define length
  (lambda (lst)
    (simplify-and-conquer lst first rest
     (lambda (x y)
       (+ 1 y))
     (lambda (x) 0)
     null?)))

Divide and Conquer

Problem

Solution

Sub

Sub

Sub

Sub
**Divide and Conquer**

\[
\text{define divide-and-conquer} \\
\quad \text{(lambda (structure select1 select2 rec-fun base-fun null-val stop?)}} \\
\quad \quad \text{(cond ((null? structure) null-val) (stop? structure) (base-fun structure) else (rec-fun (divide-and-conquer (select1 structure)) select1 select2 rec-fun base-fun null-val stop?) (divide-and-conquer (select2 structure) select1 select2 rec-fun base-fun null-val stop?)))))
\]

In order to apply *divide-and-conquer* to a problem, we need to choose procedures `select1, select2, rec-fun, base-fun, and stop.`

**Divide and Conquer example**

\[
\text{define item-count*} \\
\quad \text{(lambda (lst) (divide-and-conquer lst first rest + (lambda (x) 1) 0 (lambda (x) (not (pair? x))))))}
\]

**Divide and Conquer example**

\[
\text{define item-sum*} \\
\quad \text{(lambda (lst) (divide-and-conquer lst first rest + (lambda (x) x) 0 (lambda (x) (not (pair? x))))))}
\]

**Hierarchies of program schemes**

- Simplify and Conquer
  - Flat Recursion
  - Flat Reduction
  - Flat Mapping
- Divide and Conquer
  - Deep Recursion
  - Deep Reduction
  - Deep Mapping

*How do these relate?*
Flat recursion is simplify-and-conquer in which the selection functions are \texttt{first} and \texttt{rest}:

\begin{verbatim}
(define recur-flat
  (lambda (lst rec-fun base-fun stop?)
    (simplify-and-conquer lst first rest rec-fun base-fun stop?)))
\end{verbatim}

Deep recursion is divide-and-conquer in which the selection functions are \texttt{first} and \texttt{rest}:

\begin{verbatim}
(define recur-deep
  (lambda (lst rec-fun base-fun null-val stop?)
    (divide-and-conquer lst first rest rec-fun base-fun null-val stop?)))
\end{verbatim}

Hierarchies of program schemes

```plaintext
Hierarchies of program schemes
```

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 \texttt{select}, \texttt{diminish}, and test whether the structure is \texttt{empty?}, and by choosing a function to \texttt{augment} the accumulator.
Accumulation

(define accumulation
  (lambda (structure accumulator empty? select diminish augment)
    (if (empty? structure)
        accumulator
        (accumulation (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? first rest cons)))

(define digits
  (lambda (n)
    (accumulation n '()
                 zero?
                 (lambda (x)
                          (remainder x 10))
                 (lambda (x)
                          (quotient x 10))
                 cons)))

Hierarchies of program schemes

<table>
<thead>
<tr>
<th>Simplify and Conquer</th>
<th>Divide and Conquer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flat Recursion</td>
<td>Deep Recursion</td>
</tr>
<tr>
<td>Flat Reduction</td>
<td>Deep Reduction</td>
</tr>
<tr>
<td>Flat Mapping</td>
<td>Deep Mapping</td>
</tr>
</tbody>
</table>

Most general

Most specific
Hierarchies of program schemes

Accumulation

List–List Accumulation
List–Number Accumulation
Number–List Accumulation
Number–Number Accumulation

Most general

Most specific