Low-Level Functional Programming (using Racket)

Robert Keller
September 2012
What’s “Low-Level” About This?

- “low-level” refers to the construction of functions by explicitly creating and decomposing lists a few elements at a time.

- Previously we used higher-order functions to do most of the non-trivial work in a functional decomposition.

- Now we are going to use recursion.
A list is either:
- **empty**, () or
- **non-empty**, in which case it has both a
  - **first**
  - **rest**

Most list definitions deal with these two cases separately.

Definitions are typically a form of **inductive definition**, in which empty is the basis.
Defining Functions on Lists

• Suppose we want to define a function taking an arbitrary list as an argument.

• It is sufficient to:
  
  define the function on the empty list, and

  define the function on a general non-empty list.

  called the “basis”

  called the “induction step” or “recursion”
Recursion vs. Induction

- Recursion tends to be a “top-down” view,

  whereas

- Induction tends to be a “bottom up” view.
Induction

• ‘Induction is just a fancy way of saying “... and so on”’.

• ‘Recursion is just a fancy way of saying ??’

<Think of some comparably pithy colloquialism>.
Recursion Manifesto

• “Let recursion do the work for you.”
  CS 60 notes, Robert Keller

In other words, *assume* you can do what you’ve set out to do for the general case.
Do the special case ad hoc.
Example: Sorting

• “Sorting” means to arrange into order.

• To sort a list of numbers:

  If there is just one number, or no numbers, there is nothing to do. (basis)

  Otherwise, let F be the first number, and R be the list of numbers after F.

  Sort R, to get S, then insert F in the proper place in S.
Sorting by Insertion

(define (isort L)
  (if (null? L)
      L
      (insert (first L) (isort (rest L))))

Let recursion do the work for you!

But what about insert?
(define (insert F R)
  (cond
    [(null? R) (list F)]
    [(<= F (first R)) (cons F R)]
    [else (cons (first R) (insert F (rest R)))]))

Let recursion do the work for you!
Computation by Rewriting

\[(\text{insert} \ 4 \ '(1 \ 2 \ 5)) \Rightarrow\]
\[(\text{cons} \ 1 \ (\text{insert} \ 4 \ '(2 \ 5))) \Rightarrow\]
\[(\text{cons} \ 1 \ (\text{cons} \ 2 \ (\text{insert} \ 4 \ '(5)))) \Rightarrow\]
\[(\text{cons} \ 1 \ (\text{cons} \ 2 \ (\text{cons} \ 4 \ '(5)))) \Rightarrow\]
\[(\text{cons} \ 1 \ (\text{cons} \ 2 \ '(4 \ 5)) \Rightarrow\]
\['(1 \ 2 \ 4 \ 5)\]

(define (insert F R)
  (cond
    [(null? R) (list F)]
    [(<= F (first R)) (cons F R)]
    [else (cons (first R) (insert F (rest R)))]))
Computation by Rewriting

\[
(isort '(4 2 5 1)) \Rightarrow \\
(insert 4 (isort '(2 5 1))) \Rightarrow \\
(insert 4 (insert 2 (isort '(5 1)))) \Rightarrow \\
(insert 4 (insert 2 (insert 5 (isort '(1))))) \Rightarrow \\
(insert 4 (insert 2 (insert 5 (insert 1 (isort '()))))) \Rightarrow \\
(insert 4 (insert 2 (insert 5 (insert 1 '())))) \Rightarrow \\
(insert 4 (insert 2 (insert 5 '(1)))) \Rightarrow \\
(insert 4 '(1 2 5)) \Rightarrow \\
'(1 2 4 5)
\]

\[
(define (isort L)
  (if (null? L)
      L
      (insert (first L) (isort (rest L)))))
\]
Define from a low-level:

- map
- reduce
- member
Example: map

\[
\text{(define (map F L)} \\
\text{\hspace{1em} (if (null? L)} \\
\text{\hspace{2em} (')} \\
\text{\hspace{2em} (cons (F (first L))} \\
\text{\hspace{3em} (map F (rest L))))})
\]

Let recursion do the work for you!
Example

- Define the function `member` which tests whether the first argument is an element of the list in the second argument. If it is, then the entire list suffix beginning with that element is returned. Otherwise `#f` is returned.

```
(define (member X L)
    (if (null? L)
        #f
        (if (equal? X (first L))
            L
            (member X (rest L))))
```
Alternate Version: Use cond

Better readability than with nested if’s:

(define (member X L)
  (cond
   [(null? L)                 #f]
   [(equal? X (first L))  L]
   [else                      (member X (rest L))]]))
Quiz: Define from a low-level:

• (drop P L) drops from list L elements X where (P X) is true

• (numsTo N) generates ‘(0 1 2 3 . . . N)
Matching with
Two or More List Arguments

- Some functions have more than one list argument.

- Induction might, *or might not*, use rules that dichotomize **both** lists.
Example: append

(define (append L M)
  (if (null? L) M
      (cons (first L)
            (append (rest L) M))))
The Merge Pattern

• This **important** pattern arises many times in different applications.

• It is similar to insert, but deals with two lists.

• Construct a function that merges two lists of numbers:
  
  The argument lists are already in ascending order. The result is to be in ascending order as well.
(test (merge '(3 5 8 10) '(2 6 7 9 10 11)) '(2 3 5 6 7 8 9 10 10 11))

(define (merge L M)
  (cond
   ((null? L) M)
   ((null? M) L)
   ((<= (first L) (first M)) (cons (first L) (merge (rest L) M)))
   (else (cons (first M) (merge L (rest M)))))))

; Merge retains duplicates
Using Merge for Sorting

(define (msort L)
  (cond
    ((null? L)       ())
    ((null? (rest L)) L)
    (else          (merge (msort (everyother L))
                         (msort (everyother (rest L))))))

(check-expect (msort '(5 1 9 3 2 6 4 8 7 11 10)) '(1 2 3 4 5 6 7 8 9 10 11))

Exercise: Define everyother.
Using Merge Pattern for Sets

• Assume we want to build a library for manipulating sets.

• Assume that the elements are drawn from a set for which there is an ordering relation (such as numbers).
Sets as Lists

• For mathematical sets:
  Duplicate elements do not matter
  Order does not matter

• For lists, the above are not true.
  To represent sets as lists, we will agree:
    • A representation will not have duplicates.
    • The elements of a representation will always be in order.

• These conventions allow efficiency gains by using the merge technique.
Operations on Two Sets

• union: The elements that are in either of the sets.

• intersection: The elements that are in both.

• difference: The elements that are in the first, but not the second.
(define (union A B)
  (cond
    ((null? A) B)
    ((null? B) A)
    ((= (first A) (first B)) (cons (first A) (union (rest A) (rest B))))
    ((< (first A) (first B)) (cons (first A) (union (rest A) B)))
    (else (cons (first B) (union A (rest B))))))

(check-expect (union '(1 2 4 6) '(3 4 5 6 7)) '(1 2 3 4 5 6 7))
(define (intersection A B)
  (cond
    ((null? A) ()
     ((null? B) ())
     ((= (first A) (first B)) (cons (first A) (intersection (rest A) (rest B))))
     ((< (first A) (first B)) (intersection (rest A) B))
     (else (intersection A (rest B))))))

(check-expect (intersection '(1 2 4 6) '(3 4 5 6 7)) '(4 6))
Quiz: difference

(define (difference A B))

(check-expect (difference '(1 2 4 6) '(3 4 5 6 7)) '(1 2))
Similar Example

- Consider the representation of numbers as lists of prime factors with multiplicity, e.g.
  400 represented as \((2 4) (5 2)\)
  with factors in **increasing** order.

- Want to define functions \(gcd\) and \(lcm\) on this representation:
  \((gcd \ R \ S)\) representation of greatest common divisor of \(R\) and \(S\)
  \((lcm \ R \ S)\) representation of least common multiple of \(R\) and \(S\)

- \((check-expect (gcd '((3 5) (5 2) (7 4)) '((3 2) (7 3) (11 2))) '((3 2) (7 3)))\)
  \((check-expect (lcm '((3 5) (5 2) (7 4)) '((3 2) (7 3) (11 2))) '((3 5) (5 2) (7 4) (11 2)))\)
Unicalc Example

• Representing “quantities” in 3 parts:
  numeric multiplier
  symbolic numerator
  symbolic denominator

• e.g.
  \( \frac{2}{3} \)
  ‘(kg meter)
  ‘(second second) \[ 2/3 \text{ kg meter/sec}^2 \]
Normalized Quantities

• Units are sorted

• No units common to numerator and denominator (could be cancelled otherwise)

• Only “basic” units: Non-basic units must first be converted to basic ones using the database and recursion.
Basic Unit

- Any unit having a definition in the database association list is \textit{not} basic.
- These are \textit{not} basic, as they have a definition in the database:
  
  \begin{align*}
  \text{inch} & \quad (0.02539954113 \text{ (meter)(\()}) \\
  \text{hour} & \quad (60 \text{ (minute)(\()}) \\
  \text{minute} & \quad (60 \text{ (second)(\())})
  \end{align*}

- These are \textbf{basic}, as they have no definition in the database:
  
  \begin{itemize}
  \item meter
  \item kg
  \item second
  \end{itemize}
Loop-Free

• It is assumed that the database has no circular definitions.

• Thus, any non-basic unit can be resolved, using recursion, to a combination of basic units.

• The number of steps required can be arbitrary.
Example

• Find a normalized quantity representing 1 day:
• > (assoc 'day unicalc-db)
  (day (24 (hour) ()))

• > (assoc 'hour unicalc-db)
  (hour (60 (minute) ()))

• > (assoc 'minute unicalc-db)
  (minute (60 (second) ()))

• > (assoc 'second unicalc-db)
  #f
  The quantity is ‘(86400 (second)()).
Keep symbolic units sorted

- Easier to divide and multiply (using merge pattern)
Mutual Recursion

• This is recursion wherein one function calls another other than itself, but that function calls the original function, etc.

• Instead of: We have:

F calls G

G calls F
Mutual Recursion in Unicalc

• To normalize a Quantity:
  Normalize each unit in the **numerator**, multiplying those results together.

  Normalize each unit in the **denominator**, multiplying those results together.

  **Cancel** units common to the resulting quantity.
Normalizing a Single Unit

- Look up the unit using assoc.
- If the unit is basic (not found), \textit{create a} Quantity for it.

\begin{verbatim}
\texttt{\texttt{\textbackslash r (normalize-unit \textquotesingle \texttt{second}) (1 (second) ())}}
\end{verbatim}
Normalizing a Single Unit

• (Look up the unit using assoc.)
• If the unit is not basic (was found), a numerator and denominator are obtained.

• Each is a list of units.

• Normalize each unit, and multiply the results together, then divide the numerator product by the denominator product.

• Finally, combine with the multiplier in the database.
Example

• psi stands for “pounds per square inch

```lisp
> (normalize-unit 'psi)
(6895.006417815267 (kg) (meter second second))
```

• I’m going to conceptualize the normalization process as a tree. This does not imply that you need to construct a tree.
Conceptualization as a Tree

```
> (assoc 'psi unicalc-db)
(psi (1 (pound_force) (inch inch)))
```

```
 psi
  /   \
 1  pound_force  inch inch
```
Recursive Expansion

\[
\text{psi} = \left(\text{pound} \cdot \text{force} \left(1 \cdot \text{pound} \cdot \text{free}\_\text{acceleration}\right)\right)\]

> (assoc 'pound_force unicalc-db)
  (pound_force (1 (pound free_acceleration) ()))
Recursive Expansion

> (assoc 'pound unicalc-db)
(pound (0.45359237 (kg) ()))
Recursive Expansion

> (assoc 'free_acceleration unicalc-db)
(free_acceleration (9.80665 (meter) (second second)))
Recursive Expansion

> (assoc 'inch unicalc-db)
  (inch (0.02539954113 (meter) ()))
Cancellation

1 pound_force

1 pound free_acceleration

0.45359237 kg 9.80665 meter
0.02539954113 second second

psi

inch inch

0.02539954113 meter

meter
After Cancellation

psi

1 pound_force

1 pound free_acceleration

0.45359237 kg 9.80665

0.02539954113 inch inch

0.02539954113 meter

(\(\frac{1}{\text{psi}}\) \(\times 0.45359237\) 9.80665) \(\times 0.02539954113\) \(\times 0.02539954113\) = 6895.006417815267 (kg) (meter second second)
Possible Mutual Recursion

- normalize-unit could call a function, say normalize-list (i.e. list of units)
More Mutual Recursion Possibilities
(to avoid duplicating work)
Getting Scheme to Trace Calls

• Put in your source file:
  (require (lib "trace.ss"))

• Execute, e.g.:
  (trace normalize-unit …)

• To stop tracing, execute:
  (untrace normalize-unit …)
Example trace Output

```
> (normalize-unit 'psi)
  | (normalize-unit psi)
  | | (normalize-unit pound_force)
  | | | (normalize-unit pound)
  | | | | (normalize-unit kg)
  | | | | | (1 (kg) ())
  | | | | (0.45359237 (kg) ())
  | | | (normalize-unit free_acceleration)
  | | | | (normalize-unit meter)
  | | | | | (1 (meter) ())
  | | | | (normalize-unit second)
  | | | | | | (1 (second) ())
  | | | | | | (normalize-unit second)
  | | | | | | | (1 (second) ())
  | | | | | | | (9.80665 (meter) (second second))
  | | | | | | | | (4.4482216152605 (kg meter) (second second))

| (normalize-unit inch)
| | (normalize-unit meter)
| | | (1 (meter) ())
| | | (0.02539954113 (meter) ())
| | (normalize-unit inch)
| | | (normalize-unit meter)
| | | | (1 (meter) ())
| | | | (0.02539954113 (meter) ())
| | | (6895.006417815267 (kg) (meter second second))
| | | (6895.006417815267 (kg) (meter second second))
```

result of (normalize-unit pound_force)
Using Auxiliary Functions

- Often the function to be defined is not directly definable in a natural or efficient way using recursion. A helper or auxiliary function may be necessary.

- Example: reverse
Inefficient Reverse $O(n^2)$

```
(define (reverse L)
  (if (null? L)
      ()
      (append (reverse (rest L))
              (list (first L))))
```
Use helper function to make efficient reverse O(n)

\[
\text{(define (reverse-helper L M)}
\]
\[
\quad \text{(if (null? L)}
\]
\[
\quad \quad M
\]
\[
\quad \quad \text{(reverse-helper (rest L) (cons (first L) M))})
\]
\[
\text{)}
\]
\[
\text{(define (reverse L)}
\]
\[
\quad \text{(reverse-helper L ()})
\]
\[
\text{)}
\]

M is called an \textit{accumulator} argument.
Mixed Functional Programming Examples

• Use low-level or high-level, whatever fits best
  Maybe start with low-level, and then use high-level retrospectively, or vice-versa

• Radix conversion
  Tail recursion

• Tree and graph searching
Tail Recursion

- Sometimes recursive definitions have “cleanup” to be done after the recursive call.

- Example:

  (define (fac N)
    (if (< N 2)
      1
      (* N (fac (- N 1))))

  “cleanup happens after fac call”
• Functions that require cleanup are non-tail-recursive.
• They build up data on the call stack.
• The depth of recursion could be limited as a consequence.

\[
\begin{array}{cccc}
(fac 4) & (fac 4) & (fac 4) & (fac 4) \\
(fac 3) & (fac 3) & (fac 3) & (fac 3) \\
(fac 2) & (fac 2) & (fac 2) & (fac 2) \\
(fac 1) & (fac 1) & (fac 1) & (fac 1)
\end{array}
\]
Tail Recursion

- Functions that require cleanup are non-tail-recursive.
- They build up data on the stack.
- The depth of recursion could be limited as a consequence.

- Equivalent tail-recursive function (uses a helper):
  (define (fac N) (fac-helper N 1))
  (define (fac-helper N Acc)
    (if (< N 2)
        Acc
        (fac-helper (- N 1) (* N Acc)))))

  no cleanup  happens before call
No Stack Build-up

\[(\text{fac-helper} \ 4 \ 1) \quad (\text{fac-helper} \ 3 \ 4) \quad (\text{fac-helper} \ 2 \ 12) \quad (\text{fac-helper} \ 1 \ 24)\]

answer: 24

With tail-recursion, recursive call can be replaced with go-to.

Only certain compilers/languages do this:

- Scheme/Racket
- Prolog

Others don’t or can’t:

- C++
- Java
Convert Number to Binary

• Example:

(toBinary 37) ⇔

\[
\begin{array}{cccccc}
1 & 0 & 0 & 1 & 0 & 1 \\
\end{array}
\]

\[
1 \times 32 + 0 \times 16 + 0 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 \\
2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0
\]

• First try:

divide by 2, record remainder, continue with quotient until 0
Convert Number from Binary

• Construct fromBinary, e.g.
  (fromBinary ‘(1 0 0 1 0 1)) \(\rightarrow\) 37

• Considerations:
  Do we need an accumulator?
  Can it be done with tail-recursion?
  Try it and see.
An Approach

- Write **iterative pseudo-code**, then construct tail-recursive equivalent.

- \( L = \ldots \text{list to be converted} \ldots; \)
  \[ \text{Result} = 0; \]
  \[ \text{while}( L \neq [ ] ) \]
  \[ \{ \]
  \[ \text{Result} = 2*\text{Result} + \text{first}(L); \]
  \[ \text{L} = \text{rest}(L); \]
  \[ \} \]
  \[ \ldots \text{answer is in Result} \ldots \]
McCarthy’s Transformation

• Any iterative, imperative (i.e. assignment-based) program can be transformed to a system of mutual tail recursions.
Exercises

• Compare “obvious” and tail-recursive forms of:
  length function
  sum of a list
  reduce
  reverse
Essential Non-Tail Recursions

• Some functions don’t admit a tail-recursive version (unless reverse is used before or after):

  Examples:
  • map, keep, drop
  • append

  To avoid stack build-up, we might consider using reverse.
append Elimination
(aka “appendectomy”)

- When maximum efficiency is desired, uses of append should be avoided.
- It is often possible to get rid of append by defining versions of functions with an extra accumulator argument.
- Example:

  (define (nodes Graph)
      (remove_duplicates (append (map first Graph)
                                   (map second Graph))))

- Show how to avoid append by generalizing map to take an accumulator.
reverse elimination

• Some functions naturally build lists in reverse.

• Rather than immediately reversing the result, consider leaving it as is (in reversed form) and exploiting this fact at a later stage of the functional composition.

• Some functions, such as map, keep, drop, … work equally well whether or not the data is in reverse order.