Road Map Revisited

- Functions: Done!

- *Lisp's Foundations*: Done!

- Functional Programming
  - Recursion, Variables, Efficiency,
  - Funarg Problem (Scoping)
  - Program=Data (*eval*, *nlambda*, *oop*)
  - Lambda Calculus
  - SECD machine

- “Extensions” to Pure *Lisp*

- Example (polynomials)
Recursion

- Recursion is a problem-solving technique (a.k.a. divide-and-conquer)

- Steps in magic formula:
  - Reduce problem to simpler, self-similar problems
  - Solve the simpler problems
  - Compose results to solve the main problem

- Decomposition is also used in procedural programming
  - In recursion, subproblems are similar to original

- Recursion is the central model of computation in pure functional programming

Factorial Example

- Counts ordered $n$-tuples drawable from $n$ items without replacement

- The factorial of $n$, $\text{fct}(n)$ is the product of the first $n$ integers:

\[
\prod_{i=1,n}^{} i = 1 \times 2 \times \cdots \times (n-1) \times n
\]

- Procedurally we could write this as a loop:

```c
int fct(int n) {
    int fct := 1;
    FOR i := 1 TO n DO
        fct := fct * i;
    return fct;
}
```
Factorial’s Self-Similar Substructure

- In general computing \( fct(n) \) for different \( n \)'s repeats a lot of work
  - \( fct(6) = 1 \times 2 \times 3 \times 4 \times 5 \times 6 \), but \( fct(5) = 1 \times 2 \times 3 \times 4 \times 5 = fct(5) \)
  - If we have computed \( fct(5) \) we could get \( fct(6) = 6 \times fct(5) \)
- In general we can compute \( fct(n) \) as \( n \times fct(n - 1) \)
  - \( fct(5) = 1 \times 2 \times 3 \times 4 \times 5 \)
  - \( fct(5) = 5 \times fct(4) \)
    - \( fct(4) = 4 \times fct(3) \)
    - \( fct(3) = 3 \times fct(2) \)
    - \( fct(2) = 2 \times fct(1) \)
- \( fct(1) \) is undecomposable. We specify an answer: \( fct(1) = 1 \)

Recursive Factorial

- Self-similar substructure is captured with a conditional function:
  \[
  fct(n) = \begin{cases} 
  1 & \text{if } n = 0 \\ 
  n \times fct(n - 1) & \text{otherwise} 
  \end{cases}
  \]
- \( n = 0 \) is the "base case" and \( n > 0 \) is the "recursive case"
- Notice:
  - \( fct(n - 1) \) is a simpler problem than \( f(n) \)
  - \( n - 1 \) is a reduction operator (reduces problem to a simpler one)
  - Reduction operator progresses to base case so recursion terminates
  - Composition operator \( \times \) in \( n \times fct(n - 1) \) creates solution to original problem from subproblems
Recursive Factorial in Pure Lisp

(LABELS ((fact (n))
  (IF (= n 0)
    1
    (* n (fact (- n 1)))
  ))
(LIST
  (fact 1)
  (fact 4)
  (fact 33) )
→(1 24 8683317618811886495518194401280000000)

Recursive Factorial in Semi-pure Lisp

- The DEFUN form assigns the global function symbol fact to a closure with the arguments and body given

(DEFUN fact (n)
  "returns factorial of the non-negative integer n"
  (IF (= n 0)
    1
    (* n (fact (- n 1)))
  ))

(fact 1) →1
(fact 4) →24

- Be careful not to clobber a function with the same name or unintentionally use a previously defined predicate!
Recursions with Lists: contains

Define a function "contains(s,a)" which returns true ⇔ the atom a is contained in list s.

Can we see shared subproblems here?

(contains '(()) 3) → NIL
(contains '((3) 3) → T
(contains '((2) 3) 3)
(contains '((1) 2) 3) 3)

As Lisp code

(DEFUN contains (s a)
  (COND ((NULL s) nil)
         ((EQUAL (CAR s) a) t)
         (t (contains (CDR s) a))))

Alternative Version of contains

Original Version

(DEFUN contains (s a)
  (COND ((NULL s) nil)
         ((EQUAL (CAR s) a) t)
         (t (contains (CDR s) a)))))

Alternative Version emphasizing functional perspective

(DEFUN contains (s a)
  (AND (NOT (NULL s))
       (OR (EQUAL (CAR s) a)
           (contains (CDR s) a)))))

Effectively, we are using or to compose value of subproblems

Boolean functions can be written in compact intuitive form
Tail Recursion

- The very last recursive call to contains determines its value:

  \[
  \text{contains '(1 2 3) 3)\\n  \quad \text{(contains '(2 3) 3)\\n  \quad \quad \text{(contains '(3) 3)\\n  \quad \quad \quad \rightarrow T\\n  \quad \quad \rightarrow T\\n  \quad \rightarrow T\\n  \text{(contains '(1 2 3) 4)\\n  \quad \text{(contains '(2 3) 4)\\n  \quad \quad \text{(contains '(3) 4)\\n  \quad \quad \quad \rightarrow NIL\\n  \quad \quad \rightarrow NIL\\n  \quad \rightarrow NIL\\n  \rightarrow NIL}
  \]

- Modern compilers
  - Detect "Tail recursion"
  - Convert the computation to an iteration
  - Eliminate the recursive function calls

- Results in highly efficient code

- We can write code in a functional style obtaining freedom from side-effects and elegant formulations while obtaining the efficiency of highly-optimized compiled code
Three Types of Simple List Recursions

- Three types of recursions on a single list:
  - CAR recursion
  - CDR recursion
  - CAR/CDR recursion

- Type of recursion identified by reductions employed

- contains uses "CDR" for reduction

```
(DEFUN contains (s a)
  (COND ((NULL s) nil)
         ((EQUAL (CAR s) a) t)
         (t (contains (CDR s) a))))
```

Typical Structure of Recursions We’ll See

- Recursive Analysis
  1. Identify trivial (base) cases with immediate answers
     (e.g. atom, (), nil, 0, 1, ...)
  2. Find reduction operator(s) to transform general towards trivial
     (e.g. CAR, CDR, -1, ÷, ...)
  3. Create a composition operator to calculate answers in terms of reduced cases
     (e.g. AND, CONS, +, MAX, MIN, ...)
Recursive Version of \texttt{my-length}

▷ Can we see shared substructure?

\begin{align*}
\text{(my-length '()) } & \rightarrow 0 \\
\text{(my-length '(a) ) } & \rightarrow 1 \\
\text{(my-length '(a b) ) } & \rightarrow 2 \\
\end{align*}

▷ Analysis

1. What is trivial (base) case?
\texttt{'}() \rightarrow 0

2. How can we reduce toward this case?
\texttt{(CDR the-list)}

3. How to compose value of problem from value of reduced problem?
\texttt{(\(+ 1 \text{ reduced-value}\))}

\textbf{Lisp Implementation of my-length}

\begin{verbatim}
(defun my-length (any-list)
  "returns length of 'any-list"
  (COND ((NULL any-list) 0)
         (t (+ 1 (my-length (CDR any-list)))))
)
\end{verbatim}

▷ Base case

▷ Recursive case

▷ Reduction

▷ Composition

▷ What type of recursion? CDR-recursion
Recursive Version of my-append

Samples of behavior:

- \( \text{(my-append '() '(a))} \rightarrow (a) \); 'a
- \( \text{(my-append '(b) '(a))} \rightarrow (b a) \); (CONS 'b '(a))
- \( \text{(my-append '(c b) '(a))} \rightarrow (c b a) \); (CONS 'c (CONS 'b '(a))

Analysis

1. What is trivial (base) case?
   - () a \( \rightarrow (a) \)
2. How can we reduce toward this case?
   - (CDR first-list)
3. How to compose value of problem from value of reduced problem?
   - (CONS (FIRST first-list) reduced-value)

Lisp Implementation of my-append

(defun my-append (first-list second-list)
  (COND ( (NULL first-list) second-list)
        ( t (CONS (CAR first-list)
                     (my-append (CDR first-list)
                                 second-list)))))

Base case

Recursive case

- Reduction
- Composition

What type of recursion? CDR-recursion
Recursive Analysis of my-equal

- Suppose we want to implement 'equal' with eq

\[
\begin{align*}
\text{(my-equal 'a 'a )} & \rightarrow t ; (\text{EQ 'a 'b}) \\
\text{(my-equal 'a 'b )} & \rightarrow \text{nil} ; (\text{EQ 'a 'b}) \\
\text{(my-equal '(a) '(a) )} & \rightarrow t ; (\text{EQ (CAR '(a)) (CAR '(a))}) \\
\text{(my-equal '(a b) '(a b) )} & \rightarrow t \\
& ; \ (\text{AND (EQ (CAR '(a b)) (CAR '(a b))}) \\
& ; \ (\text{EQ (CDR '(a b)) (CDR '(a b))})
\end{align*}
\]

- Analysis

1. What is trivial (base) case? (EQ x y) where x,y atoms
2. How can we reduce toward this case?
   - Use CAR and CDR
3. Composition operator?
   - (AND reduced-car-value reduced-cdr-value)

Recursive Implementation of my-equal

\[
\begin{align*}
\text{(DEFUN my-equal (s1 s2) )}
\text{(COND ((AND (ATOM s1) (ATOM s2))}
\ & \ (\text{EQ s1 s2})
\text{)}
\text{ ((AND (CONSP s1) (CONSP s2))}
\ & \ (\text{AND (my-equal (CAR s1) (CAR s2))}
\ & \ (\text{my-equal (CDR s1) (CDR s2))}) )
\ & \ (t \ nil) ))
\end{align*}
\]

- Base case
- Recursive case
  - Reduction
  - Composition
- What type of recursion? CAR-CAR-recursion
Alternative Implementation of my-equal

▶ Original Implementation

(DEFUN my-equal (s1 s2)
  (COND ((AND (ATOM s1) (ATOM s2))
       (EQ s1 s2))
     ((AND (CONSP s1) (CONSP s2))
      (AND (my-equal (CAR s1) (CAR s2))
           (my-equal (CDR s1) (CDR s2)))
     (t nil)))

▶ Alternative version emphasizing functional perspective

(DEFUN my-equal (s1 s2)
  (OR (AND (ATOM s1) (ATOM s2) (EQ s1 s2))
       (AND (CONSP s1) (CONSP s2)
            (my-equal (CAR s1) (CAR s2))
            (my-equal (CDR s1) (CDR s2))))

Efficient Implementation of my-equal

▶ Alternative version

(DEFUN my-equal (s1 s2)
  (OR (AND (ATOM s1) (ATOM s2) (EQ s1 s2))
       (AND (CONSP s1) (CONSP s2) ;; eliminate!
            (my-equal (CAR s1) (CAR s2))
            (my-equal (CDR s1) (CDR s2))))

▶ Efficient Version

(DEFUN my-equal (s1 s2)
  (COND ((ATOM s1)
           (AND (ATOM s2) (EQ s1 s2))
           ((ATOM s2) nil)
           ((my-equal (CAR s1) (CAR s2))
            (my-equal (CDR s1) (CDR s2))))
Other Problems to Try

- split(s) which returns a pair (s1 . s2) of lists jointly containing the original elements of s and the difference in length between s1 and s2 is at most 1

  \[
  \text{split( '(a b c d) ) } \rightarrow (\text{(a c) (b d)}) \\
  \text{split( '(a b c d e) ) } \rightarrow (\text{(a c e) (b d)})
  \]

- even-list(s) which returns true (e.g. T) if list s has even length

  \[
  \text{even-list( '(a b c d) ) } \rightarrow \text{T} \\
  \text{even-list( '(a b c d e) ) } \rightarrow \text{nil}
  \]

- flatten(s) which returns list containing atoms of s all at the top level

  \[
  \text{flatten( '( (a b) ((c) d) ) ) } \rightarrow (a b c d)
  \]

Recursion as Substitution

\[
(\text{DEFUN length (L)} \\
(\text{IF (NULL L) 0 (+ 1 (length (CDR L)))))})
\]

- Need \( n \) substitutions to evaluate \( n \)-element lists!

\[
(\text{LAMBDA (lst1)} \\
(\text{IF (NULL lst1) 0 (+ 1 (LAMBDA (lst2) \\
(\text{IF (NULL lst2) 0 (+ 1 (LAMBDA (lst3) \\
(\text{IF (NULL lst3) 0 (+ 1 (LAMBDA (lst4) \\
(\text{... \\
\text{) (CDR lst3))}})
\text{) (CDR lst2))}})
\text{) (CDR lst1))})})}
\]
Recursion as Self-Referential Variables

( (LAMBDA (dummy)
  ( (LAMBDA (length)
    (SETF dummy length)
    (FUNCALL length '(a b c d))
  ) (LAMBDA (L)
    (IF (NULL L) 0
        (+ 1 (funcall dummy (CDR L)))
    )
  ) ) 'any-old-value ) → 4
)

▷ Local environment with dummy variable
▷ Write "length" which calls "dummy"
▷ Pass "length" to inner environment
▷ Set dummy to length so "length" calls itself
▷ Use recursive function in body and get result

LABELS as Self-Referential Variables

▷ Self-reference requires a SETF
▷ But variable "dummy" is inside a LAMBDA closure so all side-effects are isolated
▷ The LABELS construct performs the previous expansion for us

(LABELS ((length (L)
     (IF (NULL L) 0
         (+ 1 (length (CDR L)))
     )
   )
(length '(a b c d)) ) → 4

▷ Pure Lisp with LABELS is therefore sufficient to compute any function