Functional Programming

- Functions
- *Lisp*'s Foundations
  Data Structures
  Built-in Functions + Predicates
  Evaluation
  Lambda-Expressions
  Special Forms
  Functional Arguments + Label Lambda-Expressions
- "Extensions" to *Lisp*
  Side Effects
  Numbers
  Dotted-Pair, Association & Property Lists
  Lisp qua Procedural Language
- Issues
  ⇒ Recursion, Variables, Efficiency,
  – Funarg Problem (Scoping)
  – Program=Data (Eval, nlambda, OOp)
  – Lambda Calculus
  – SECD machine
- Example (polynomials)

Recursion Example: Factorial

- $\texttt{fct}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times \texttt{fct}(n-1) & \text{otherwise} \end{cases}$
- Trivial Case: Arg is 0
  Recall $\texttt{fct}(0) = 1$
  (Notice: no further recurring needed.)
- Reduction:

  Can write $\texttt{fct}(n)$
  as function involving $\texttt{fct}(n-1)$:

  Reduction Operator: -1
  as $n \rightsquigarrow n-1$

  Composition Operator: $n \times \ldots$
  as $\texttt{fct(n)} \rightsquigarrow n \times \texttt{fct(n-1)}$

Recursion (in Lisp)

- In *Lisp*, 3 primary kinds:
  `CAR` recursion
  `CDR` recursion
  `CAR/CDR` recursion
- Analysis
  1. Trivial case of problem
     (no further recurring needed)
     atom, empty list, 0, 1, ...
  2. Reduction operator
     (eventually transform general ↦ trivial)
     `CAR`, `CDR`, `-1`, ÷, ...
  3. Composition operator
     (Get value of general case from values of reduced cases)
     general value = f(*reduced value*)

Example: `LENGTH`

- Behavior:
  → (LENGTH '(A B C))
  3
  → (LENGTH ())
  0
  → (LENGTH '((A) B))
  2
- Recursive: `CDR`-recursion
  [Call arg `L1`]
  1. Trivial case:
     `(NULL L1)` Here: value is `0`
  2. Reduction operator: `CDR` on `L1`
     on `(A B C)`
     Reduced-value = `LENGTH` of `(B C)`
     = 2
  3. Composition operator:
     `(+ 1 reduced-value)`

Recursive Functions

```
(DEFUN length (l1)
 (COND  ((NULL l1) 0 )
        ( t        (+     1
                          (length (CDR l1)))
  )) )


(DEFUN fct (n)
 (COND  ((EQ n 0)  1 )
        ( t        (TIMES n
                          (fct (- n 1)))
  )) )


(DEFUN append (l1 l2)
 (COND  ((NULL l1) l2 )
        ( t        (CONS  (CAR l1)
                          (append (CDR l1) l2))
  )) )
```


Example: `ALL-ATOMP`

- Behavior (examples):

  → (all-atomp '(A B C))
  t
  → (all-atomp '(A (B C)))
  nil
  → (all-atomp nil)
  t

- Each member of List is an `atom`.
  ⇒ Needs `CDR` Recursion!

```
(DEFUN all-atomp (s)
 (COND  ((NULL s)                t)
        ((NOT (ATOM (CAR s)))  nil)
        ( t     (all-atomp (CDR s))) ))
```

Tail recursion:
Efficient, as value of internal `all-atomp` is simply returned by outer `all-atomp`.

Example: `EQUALITY`

- Behavior (examples):

  → (EQUAL '(A (B C) D) '(A (B C) D))
  t
  → (EQUAL '(A (B C) D) '(A B (C D))
  nil
  → (EQUAL 'A 'A)
  t

- Behavior (def'n):

  `EQUAL` of a_1 and a_2 is $\left\{ \begin{array}{ll} \texttt{t} & \text{if same s-expr} \\ \texttt{nil} & \text{otherwise} \end{array} \right\}$

- Must check EACH Position of S-Expressions.
  ⇒ Needs `CAR`/`CDR` Recursion!

`EQUAL` Function

```
(DEFUN equal (x y)
 (COND  ((ATOM x)   (COND ((ATOM y) (EQ x y))
                          (t      nil)))
        ((ATOM y)  nil)
        ((EQUAL (CAR x) (CAR y))
         (EQUAL (CDR x) (CDR y)))
        ( t        nil) )))
```

Notes:
First 2 clauses are *TRIVIAL Cases*
3rd clause is CONJUNCTION of reduced cases!

```
(DEFUN equal (x y)
 (COND  ((ATOM x)  (AND (ATOM y) (EQ x y)))
        ((ATOM y)  nil)
        ( t        (AND (EQUAL (CAR x) (CAR y))
                     (EQUAL (CDR x) (CDR y))) )))
```

Other examples: See p42.

Variables

- What are *Variables*?
  "Constructs" for "holding" values.
- From where do values arise?
  (which are then stored in *Variables*)
  By application of functions.
- Purposes of *Variables*?
  Passing Parameters
  Efficiency!

Efficiency via Variables – I

- Compare:
  - `Y := F(x) × F(x)` with
  - `Z := F(x)`
    `Y := Z × Z`

  First: computes `F(x)` twice;
  Second: only once!
- In *Lisp*:
  - `(LAMBDA (x) (APPEND (FOO x) (FOO x)))` vs
  - `(LAMBDA (x) ((LAMBDA (z) (APPEND z z)) (FOO x)))`
- View `(LAMBDA (z) (APPEND z z))` as a
  "HELPING FUNCTION"

`Append` Function

- Code:

```
(DEFUN append (l1 l2)
  (COND  ((NULL l1) l2 )
         ( T         (CONS (CAR l1)
                           (append (CDR l1) l2))
   )) )
```

- Efficiency:

length[l1]	length[l2]	# iterations?
0	5	1
1	5	2
6	5	7
10	5	11
10	100	11
10	0	11
10	100,000	11

So...

- RunTime[`Append`] $=$ O(length[l1])

  [Ie, `Append` is LINEAR in length of 1st arg]

Efficiency via Variables – II

- Consider `REVERSE` function
  `(REVERSE '(A B C D E))` ⇝ `(E D C B A)`
- Proposed solution:

```
(DEFUN reverse (l)
  (COND ((NULL l) nil)
        ( t       (APPEND (reverse (CDR l))
                          (LIST (CAR l))) )))
```

- Inefficient: Traverses (partial) list at each step!

  ...

  2: Build up `(E)`
  then Traverse it to put `D` at end.

  3: Build up `(E D)`
  then Traverse it to put `C` at end.

  ...

  5: Build up `(E D C B)`
  then Traverse it to put `A` at end.

Trace of `Reverse`

```
→ (reverse '(a b c d e))
   Enter reverse (a b c d e)
       Enter reverse (b c d e)
           Enter reverse (c d e)
               Enter reverse (d e)
                   Enter reverse (e)
                       Enter reverse nil
                       Exit reverse nil
                       Enter append nil (e)
                       Exit append (e)
                   Exit reverse (e)
                   Enter append (e) (d)
                   Exit append (e d)
               Exit reverse (e d)
               Enter append (e d) (c)
               Exit append (e d c)
           Exit reverse (e d c)
           Enter append (e d c) (b)
           Exit append (e d c b)
       Exit reverse (e d c b)
       Enter append (e d c b) (a)
```



```
        Exit append (e d c b a)
    Exit reverse (e d c b a)
(e d c b a)
```

Efficient `REVERSE` Code

- In "Pseudo-*PASCAL*"

```
x ← L
answer ← nil
while  x ≠ NIL do
  [ answer ← (CONS (CAR x) answer)
  [ x      ← (CDR x)
return [answer]
```

- Efficient:
  Traverses `L`-list only once
  building up `answer` while reducing `x`

- Uses a *Collector Variable*

Collector Variable

- A **collector variable** is
  an *extra* argument in function
  that represents the calculation so far.

  When function is done
  (typically by exhausting another argument)
  it simply returns the collector variable
  as value of function.

- Here: Use *helper function* `RevLoop`
  whose 2^{nd} arg is collector.
  [Starting at `nil`].

```
(DEF reverse (LAMBDA (l)
   (RevLoop l nil)))
```

Efficient `REVERSE` Code (In Lisp)

```
(DEF reverse (LAMBDA (l)
  (RevLoop l nil)

(DEFUN RevLoop (x ans)
  (COND  ((NULL x)  ans)
         ( t         (RevLoop (CDR x)
                              (CONS (CAR x) ans))
  ))))
```

NOTES:

`RevLoop` is *helping function*.

Special kind of recursion: "Tail-end Recursion"

— composition operation is "Identity"

[Last thing done is Recursive call]

⇒ As efficient as ITERATION

[in some implementations: store+branch]

Trace of Efficient `Reverse`

```
→ (reverse '(a b c d ))
   1 Enter Reverse (a b c d)
     1 Enter RevLoop (a b c d) nil
       2 Enter RevLoop (b c d) (a)
         3 Enter RevLoop (c d) (b a)
           4 Enter RevLoop (d) (c b a)
             5 Enter RevLoop nil (d c b a)
             5 Exit RevLoop (d c b a)
           4 Exit RevLoop (d c b a)
         3 Exit RevLoop (d c b a)
       2 Exit RevLoop (d c b a)
     1 Exit RevLoop (d c b a)
   1 Exit Reverse (d c b a)
(d c b a)
```


Lisp `REVERSE` Code #2

```
(DEFUN reverse (l)
  ((LABEL RevLoop (LAMBDA (x ans)
    (COND ((NULL x) ans)
          ( t        (RevLoop (CDR x)
                              (CONS (CAR x) ans))
            )))
   l nil))
```

NOTES:

Use of `LABEL` makes helping function

local + temporary

[Even though `reverse` uses `DEFUN`]

Efficiency in General

- Q: Is $\langle \mathtt{fn}_1 \rangle$ *more efficient* than $\langle \mathtt{fn}_2 \rangle$?
  wrt expected *Run Time Cost*
  for LARGE problems
  [worst case]

- Defined in terms of
  # of Function Applications
  as a function of "Size" of Argument(s)

- "Size"
  Usually Assymptotic
  "... for sufficiently large lists..."
  wrt LISP: Usually "length of list"

Efficiency – Classes

- "Constant Order" $O(1)$
  \# of Function Applications
  is INDEDEPENT of args
  …No recursion

  [Eg, `(LAMBDA (x) (CAR (CDR x)))`]

- "Linear Order" $O(n)$
  (n is size of argument)
  1 Recursive call,
  but CONSTANT work at each pass

  [Eg, `APPEND ...(CONS (CAR x) (APPEND (CDR x) y))...`]
  [Here n is length of `x`.]

Efficiency – Classes (con't)

- "Polynomial Order" $O(n^2)$, $O(n^5)$, …
  Recursion,
  with Linear (poly) work at each level

  Eg, bad **reverse**:
  …(APPEND (reverse (CDR x)) …) …
  $\nwarrow$ linear at each level
  so POLY order

  Recall good **reverse**:
  …(RevLoop (CDR x) (CONS (CAR x) ans))
  $\nwarrow$ constant at each level
  so LINEAR order

- "Exponential Order" $O(2^n)$, $O(n^n)$, …
  More than 1 recursive call
  at each level (over same args)

`Fib` Function

- Function:
  Fib(0) = 1; Fib(1) = 1;
  Fib($n+2$) = Fib($n+1$) + Fib(n)

n	1	2	3	4	5	6	7	8	…
Fib(n)	1	1	2	3	5	8	13	21	…

- Bad Code: [Exponential Order]
  Good Code: [Linear Order]

- Pure Lisp $\not\equiv$ InEfficient Code

Code for `Fib` Function

- Bad Code: [Exponential Order]

```
(DEFUN Fib1 (n)
 (COND   ((LESSp n 3)  1 )
         ( t          (PLUS  (Fib1 (- n 1) )
                             (Fib1 (- n 2) )
   ))))
```

- Good Code: [Linear Order]

```
(DEFUN Fib2 (n)
  (FibHelp n 1 1 0))

(DEFUN FibHelp (n I FibI Prev)
 (COND  ((EQ n I) FibI )
        ( t       (FibHelp  (Plus I 1)
                            (Plus FibI Prev)
                            FibI ))) )
```

?? Better Code for `Fib` ??

$$\begin{pmatrix} f_{n+1} \\ f_n \end{pmatrix} = \begin{pmatrix} f_n + f_{n-1} \\ f_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} f_n \\ f_{n-1} \end{pmatrix}$$

$$\begin{pmatrix} f_2 \\ f_1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

$$\begin{aligned}
\begin{pmatrix} f_{k+2} \\ f_{k+1} \end{pmatrix} &= \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} f_{k+1} \\ f_k \end{pmatrix} \\
&= \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} f_k \\ f_{k-1} \end{pmatrix} \\
&= \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}^2 \begin{pmatrix} f_k \\ f_{k-1} \end{pmatrix} \\
&= \quad \cdots \\
&= \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}^k \begin{pmatrix} f_2 \\ f_1 \end{pmatrix} \\
&= \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}^k \begin{pmatrix} 1 \\ 1 \end{pmatrix}
\end{aligned}$$

Towards Efficient Code for `Fib`

Example:

$$\begin{aligned}
\begin{pmatrix} f_{18} \\ f_{17} \end{pmatrix} &= \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}^{16} \begin{pmatrix} f_2 \\ f_1 \end{pmatrix} \\
&= \left[\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}^8\right]^2 \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\
&= \left\{\left[\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}^4\right]^2\right\}^2 \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\
&= \left[\left\{\left[\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}^2\right]^2\right\}^2\right]^2 \begin{pmatrix} 1 \\ 1 \end{pmatrix}
\end{aligned}$$

. . . involves just 4 matrix multiplications
(of fixed-size 2×2 matrix) !

In gen'l: f_{2^k+2} requires k matrix multiplications!
f_n requires $O(\ln n)$ operations!

$O(\lg_2 n)$ Code for `Fib`

- Code is straightforward
  (even in PureLisp!)

```
(defun fib (n)
  (cond ((< n 2) 1)
        (t (1-1-of (m-times (fib-help (tobinary (- n 2))
                                      '⟨( 1 1 ; 0 1 )⟩)
        '⟨( 1 ; 1 )⟩)) )))

(defun fib-help (bin mat)
  (cond ((null bin) '⟨( 1 0 ; 0 1 )⟩)
        ((eq 1 (car bin))
         (m-times mat (fib-help (cdr bin)
                                (m-times mat mat))))
        (t (fib-help (cdr bin)
                     (m-times mat mat))) ))
```

- Need
  Integer to Binary conversion — `tobinary`
  Encoding of Matrices — $\langle \begin{pmatrix} x \\ y \end{pmatrix} \rangle$
  Matrix operations — `1-1-of`, `m-times`