CMPT325: Functional Programming Techniques

Dr. B. Price & Dr. R. Greiner

16th September 2004
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)
Recursion

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

Steps in magic formula:
Recursion

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

- Steps in magic formula:
  - Reduce problem to simpler, *self-similar* problems
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
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
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
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

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 $n$, $fct(n)$, 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 (int i = 1; i <= n; i++)
        fct *= i;
    return fct;
}
```
Factorial Example

- Counts ordered \( n \)-tuples drawable from \( n \) items without replacement

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

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

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

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

- The factorial of $n$, $fct(n)$ 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:

```java
int fct(int n)
    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.
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 $fct(5) = 1 \times 2 \times 3 \times 4 \times 5$
    
    $= fct(5)$
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 $fct(6) = 6 \times fct(5)$
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 \( n \times fct(n - 1) \)
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 $fct(5) = 1 \times 2 \times 3 \times 4 \times 5$

▶ If we have computed $fct(5)$ we could get $fct(6) = 6 \times fct(5)$

▶ In general we can compute $fct(n)$ as $n \times fct(n - 1)$

$fct(5) = 1 \times 2 \times 3 \times 4 \times 5$
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 \( n \times fct(n - 1) \)
  
  \[ fct(5) = 1 \times 2 \times 3 \times 4 \times 5 \]
  
  \[ fct(5) = 5 \times fct(4) \]
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 $n \times fct(n-1)$

$$fct(5) = 1 \times 2 \times 3 \times 4 \times 5$$

$$fct(5) = 5 \times fct(4)$$

  $$fct(4) = 4 \times fct(3)$$
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 $n \times fct(n - 1)$

  $fct(5) = 1 \times 2 \times 3 \times 4 \times 5$
  $fct(5) = 5 \times fct(4)$
  $fct(4) = 4 \times fct(3)$
  $fct(3) = 3 \times fct(2)$
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 $n \times fct(n - 1)$
  - $fct(5) = 1 \times 2 \times 3 \times 4 \times 5$
  - $fct(5) = 5 \times fct(4)$
    - $fct(4) = 4 \times fct(3)$
      - $fct(3) = 3 \times fct(2)$
        - $fct(2) = 2 \times fct(1)$
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:

\[
fct(n) = \begin{cases} 
1 & \text{if } n = 0 \\
n \times fct(n - 1) & \text{otherwise}
\end{cases}
\]
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"
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:
Recursive Factorial

- Self-similar substructure is captured with a conditional function:

\[
\text{fct}(n) = \begin{cases} 
1 & \text{if } n = 0 \\
 n \times \text{fct}(n - 1) & \text{otherwise}
\end{cases}
\]

- \(n = 0\) is the "base case" and \(n > 0\) is the "recursive case"

- Notice:
  - \(\text{fct}(n - 1)\) is a simpler problem than \(f(n)\)
Recursive Factorial

- Self-similar substructure is captured with a conditional function:

\[
fct(n) = \begin{cases} 
1 & \text{if } n = 0 \\
\times fct(n - 1) & \text{otherwise}
\end{cases}
\]

- \(n = 0\) 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)
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
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

\[
\text{(LABELS ((fact (n))}
\quad \text{(IF (= n 0)}
\quad \quad 1
\quad \quad (\text{* n (fact (- n 1))))))
\quad ))
\]

\[
\rightarrow (1 24 868331761881188649551819440128000000)
\]
Recursive Factorial in Pure Lisp

\[
\text{(LABELS ((fact (n))
             (IF (= n 0)
                 1
                 (* n (fact (- n 1))))
           ))
\]

(LIST
  (fact 1)
  (fact 4)
  (fact 33)
) → (1 24 86833176188118864955181944012800000000 )
Recursive Factorial in Semi-pure Lisp

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

Be careful not to clobber a function with the same name or unintentionally use a previously defined predicate!
Recursive Factorial in Semi-pure Lisp

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

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

(fact 1) → 1
```
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
```
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 if and only if the atom a is contained in list s.
Recursions with Lists: contains

- Define a function "contains(s,a)" which returns true if and only if the atom a is contained in list s.

- Can we see shared subproblems here?
Recursion
Overview

Recusions 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)
Recursion Overview

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)
  ;(OR (EQ 1 3) (contains '(1 2) 3)

As Lisp code

```lisp
(defun contains (s a)
  (cond ((null s) nil)
        ((equal (car s) a) t)
        (t (contains (cdr s) a)))
)```

Dr. B. Price & Dr. R. Greiner
CMPT325: Functional Programming Techniques
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?

  \[
  \begin{align*}
  \text{(contains '() 3)} &= \text{NIL} \\
  \text{(contains '(3) 3)} &= \text{T} \\
  \text{(contains '(2 3) 3)} \\
  \text{(contains '(1 2 3) 3)} \\
  \quad &; (\text{OR (EQ 1 3) (contains '(1 2) 3)})
  \end{align*}
  \]

- As Lisp code

  \[
  \text{(DEFUN contains (s a)} \\
  \quad \text{ (COND ((NULL s) nil) (EQUAL (CAR s) a) t) ( t (contains (CDR s) a))})
  \]
Alternative Version of `contains`

- Original Version

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

- Alternative Version emphasizing functional perspective

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

Effectively, we are using `or` to compose the value of subproblems.

Boolean functions can be written in compact intuitive form.
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)))))
Alternative Version of contains

- Original Version

\[
(\text{DEFUN contains (s a)} \\
\quad (\text{COND ((NULL s) nil) \\
\quad \quad ((\text{EQUAL (CAR s) a) t) \\
\quad \quad \quad (t (\text{contains (CDR s) a))))}))
\]

- Alternative Version emphasizing functional perspective

\[
(\text{DEFUN contains (s a)} \\
\quad (\text{AND (NOT (NULL s)) \\
\quad \quad (\text{OR (EQUAL (CAR s) a) \\
\quad \quad \quad (contains (CDR s) a))}))
\]

- Effectively, we are using or to compose value of subproblems
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

\[
\text{(contains } '(1 2 3) 3) \\
\text{(contains } '(2 3) 3) \\
\text{(contains } '(3) 3) \\
\rightarrow T \\
\rightarrow T \\
\rightarrow T \\
\rightarrow T
\]
Tail Recursion

- The very last recursive call to contains determines its value

(contains '(1 2 3) 3)
  (contains '(2 3) 3)
    (contains '(3) 3)
      → T
    → T
  → T
(contains '(1 2 3) 4)
  (contains '(2 3) 4)
    (contains '(3) 4)
      (contains '() 4)
        → NIL
    → NIL
  → NIL
→ NIL
Tail Recursion

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

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

- Results in highly efficient code
Tail Recursion

- 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
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
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

```lisp
(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
Recursion Overview

Typical Structure of Recursions We’ll See

- Recursive Analysis
  1. Identify trivial (base) cases with immediate answers (e.g. atom, (), nil, 0, 1, ...)
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, ÷, ...)

Dr. B. Price & Dr. R. Greiner
CMPT325: Functional Programming Techniques
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 my-length

- Can we see shared substructure?

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

- Can we see shared substructure?
  
  \( \text{(my-length '()) } \rightarrow 0 \)
  
  \( \text{(my-length '(a) ) } \rightarrow 1 \)
  
  \( \text{(my-length '(a b) ) } \rightarrow 2 \)

- Analysis
Recursive Version of my-length

- Can we see shared substructure?

  (my-length '() ) → 0
  (my-length '(a) ) → 1
  (my-length '(a b) ) → 2

- Analysis

  1. What is trivial (base) case?
Recursive Version of my-length

- Can we see shared substructure?
  - `(my-length '()) → 0`
  - `(my-length '(a)) → 1`
  - `(my-length '(a b)) → 2`

- Analysis
  1. What is trivial (base) case?
     - '() → 0
Recursive Version of my-length

- Can we see shared substructure?

  \(\text{my-length } '(()) \rightarrow 0\)
  \(\text{my-length } '(a) \rightarrow 1\)
  \(\text{my-length } '(a\ b) \rightarrow 2\)

- Analysis

  1. What is trivial (base) case?
     \( '(()) \rightarrow 0\)
  2. How can we reduce toward this case?
Recursive Version of my-length

- Can we see shared substructure?
  - `(my-length '()) \rightarrow 0`
  - `(my-length '(a) ) \rightarrow 1`
  - `(my-length '(a b) ) \rightarrow 2`

- Analysis
  1. What is trivial (base) case?
     - '() \rightarrow 0
  2. How can we reduce toward this case?
     - `(CDR the-list)`
Recursive Version of my-length

- Can we see shared substructure?
  
  (my-length '(()) ) → 0
  (my-length '(a) ) → 1
  (my-length '(a b) ) → 2

- Analysis
  
  1. What is trivial (base) case?
     ’() → 0
  2. How can we reduce toward this case?
     (CDR the-list)
  3. How to compose value of problem from value of reduced problem?
Recursive Version of my-length

▶ Can we see shared substructure?

(my-length ’() ) → 0  
(my-length ’(a) ) → 1  
(my-length ’(a b) ) → 2

▶ Analysis

1. What is trivial (base) case?  
   ’() → 0

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

3. How to compose value of problem from value of reduced problem?  
   (+ 1 reduced-value)
Lisp Implementation of my-length

(defun my-length (any-list)
  "returns length of 'any-list'"
  (cond (null any-list) 0
        (t (+ 1 (my-length (cdr any-list)))))
)
Lisp Implementation of my-length

(defun my-length (any-list)
  "returns length of 'any-list"
  (COND ( (NULL any-list) 0)
          (t (+ 1 (my-length (CDR any-list)) ) )
  )
)

▷ Base case
Lisp Implementation of my-length

(defvar my-length (any-list)
  "returns length of 'any-list'"
  (cond ((null any-list) 0)
         (t (+ 1 (my-length (cdr any-list)))))
)

- Base case
- Recursive case
  - Reduction
  - Composition
Lisp Implementation of my-length

(defun my-length (any-list)
  "returns length of 'any-list'"
  (COND ( (NULL any-list) 0)
         (t (+ 1 (my-length (CDR any-list)) )))
)

- Base case
- Recursive case
  - Reduction
  - Composition
- What type of recursion?
Lisp Implementation of my-length

(defun my-length (any-list)
  "returns length of 'any-list'"
  (COND ( (NULL any-list) 0)
        ( t (+ 1 (my-length (CDR any-list)) ) )
        )
)

- Base case
- Recursive case
  - Reduction
  - Composition
- What type of recursion? CDR-recursion
Recursive Version of my-append

- Samples of behavior:

\[(\text{my-append } () (\text{a}) ) \rightarrow (\text{a}) \quad ; \quad '(\text{a})\]
\[(\text{my-append } (\text{b}) (\text{a}) ) \rightarrow (\text{b a}) \quad ; \quad (\text{CONS } '\text{b }'(\text{a}))\]
\[(\text{my-append } (\text{c b}) (\text{a}) ) \rightarrow (\text{c b a}) \quad ; \quad (\text{CONS } '\text{c} \quad (\text{CONS } '\text{b }'(\text{a})))\]
Recursive Version of my-append

- Samples of behavior:

  (my-append '() '(a) ) → (a) ; '(a)
  (my-append '(b) '(a) ) → (b a) ; (CONS 'b '(a))
  (my-append '(c b) '(a) ) → (c b a) ; (CONS 'c
  (CONS 'b '(a))

- Analysis
Recursive Version of my-append

Samples of behavior:

\[
\begin{align*}
(\text{my-append } '(()) '(a) ) & \rightarrow (a) & \Rightarrow '(a) \\
(\text{my-append } '(b) '(a) ) & \rightarrow (b \ a) & \Rightarrow (\text{CONS } 'b ' '(a)) \\
(\text{my-append } '(c \ b) '(a) ) & \rightarrow (c \ b \ a) & \Rightarrow (\text{CONS } 'c \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad (\text{CONS } 'b ' '(a)))
\end{align*}
\]

Analysis

1. What is trivial (base) case?
Recursive Version of my-append

Samples of behavior:

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

Analysis

1. What is trivial (base) case?
   \[(() \ a \ \rightarrow (a))\]
Recursive Version of my-append

- Samples of behavior:
  
  (my-append '(a) '(a)) → (a) ; (a)
  (my-append '(b) '(a)) → (b a) ; (CONS 'b '(a))
  (my-append '(c b) '(a)) → (c b a) ; (CONS 'c (CONS 'b '(a)))

- Analysis

  1. What is trivial (base) case?
     () a → (a)

  2. How can we reduce toward this case?
Recursive Version of my-append

▶ Samples of behavior:

\[
\begin{align*}
\text{(my-append '() '(a) )} & \rightarrow (a) \quad ; \quad '(a) \\
\text{(my-append '(b) '(a) )} & \rightarrow (b \ a) \quad ; \quad (\text{CONS } 'b \ '(a)) \\
\text{(my-append '(c b) '(a) )} & \rightarrow (c \ b \ a) \quad ; \quad (\text{CONS } 'c \\
& \quad (\text{CONS } 'b \ '(a)))
\end{align*}
\]

▶ Analysis

1. What is trivial (base) case?
   () a $\rightarrow$ (a)

2. How can we reduce toward this case?
   (CDR first-list)
Recursive Version of my-append

- **Samples of behavior:**

  \[
  \text{(my-append '() '(a) ) } \rightarrow (a) ; \ (a)
  \]

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

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

- **Analysis**

  1. What is trivial (base) case?
     
     \((\ ) \ a \rightarrow (a)\)

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

  3. How to compose value of problem from value of reduced problem?
Recursive Version of my-append

- **Samples of behavior:**

  \[
  \begin{align*}
  \text{(my-append } &\, (\text{'() '}(a)) \to (a) \quad ; \quad \text{'(a)} \\
  \text{(my-append } &\, (\text{'(b) '}(a)) \to (\text{b a}) \quad ; \quad (\text{CONS 'b '}(a)) \\
  \text{(my-append } &\, (\text{'(c b) '}(a)) \to (\text{c b a}) \quad ; \quad (\text{CONS 'c} \\
  \quad &\text{(CONS 'b '}(a)))
  \end{align*}
  \]

- **Analysis**

  1. What is trivial (base) case?
     
     (\text{() a } \to (a))

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

  3. How to compose value of problem from value of reduced problem?
     
     (\text{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)))))
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
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
Recursion

Overview

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?
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 \textit{my-equal}

- Suppose we want to implement 'equal' with eq

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

\[
\text{; (EQ 'a 'b)}
\]

\[
\text{; (EQ 'a 'b)}
\]

\[
\text{; (EQ \text{(CAR '(a)) (CAR '(a))})}
\]

\[
\text{; (AND \text{(EQ (CAR '(a \ b)) (CAR '(a \ b)))}}
\]

\[
\text{; \text{(EQ (CDR '(a \ b)) (CDR '(a \ b)))}}
\]
Recursion

Recursion Analysis of my-equal

▶ Suppose we want to implement 'equal' with eq

\[
\begin{align*}
\text{my-equal } &\ 'a 'a \rightarrow t ;(\text{EQ 'a 'b}) \\
\text{my-equal } &\ 'a 'b \rightarrow \text{nil} ;(\text{EQ 'a 'b}) \\
\text{my-equal } &\ '(a) '(a) \rightarrow t ;(\text{EQ (CAR '(a)) (CAR '(a))}) \\
\text{my-equal } &\ '(a b) '(a b) \rightarrow t \\
&\hspace{1cm};(\text{AND (EQ (CAR '(a b)) (CAR '(a b))}) \\
&\hspace{1cm};(\text{EQ (CDR '(a b)) (CDR '(a b))})
\end{align*}
\]

▶ Analysis
Recursive Analysis of my-equal

- Suppose we want to implement 'equal' with eq

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

- Analysis

1. What is trivial (base) case?
Recursive Analysis of my-equal

- Suppose we want to implement 'equal' with eq

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

- Analysis

1. What is trivial (base) case? (EQ x y) where x,y atoms
Recursive Analysis of my-equal

- Suppose we want to implement 'equal' with eq

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

- Analysis

1. What is trivial (base) case? (EQ x y) where x,y atoms
2. How can we reduce toward this case?
Recursive Analysis of my-equal

- Suppose we want to implement 'equal' with eq

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

- 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
Recursive Analysis of my-equal

- Suppose we want to implement 'equal' with eq

\[(\text{my-equal } 'a 'a ) \rightarrow t \quad ;(\text{EQ } 'a 'b)\]
\[(\text{my-equal } 'a 'b ) \rightarrow \text{nil} \quad ;(\text{EQ } 'a 'b)\]
\[(\text{my-equal } '(a) '(a) ) \rightarrow t \quad ;(\text{EQ (CAR '(a)) (CAR '(a))})\]
\[(\text{my-equal } '(a b) '(a b) ) \rightarrow t\]
\quad ;(\text{AND (EQ (CAR '(a b)) (CAR '(a b))})\]
\quad ;(\text{EQ (CDR '(a b)) (CDR '(a b))})\]

- Analysis

1. What is trivial (base) case? \((\text{EQ } x y)\) where \(x, y\) atoms
2. How can we reduce toward this case?
   - Use \text{CAR and CDR}
3. Composition operator?
Recurse Analysis of my-equal

- Suppose we want to implement 'equal' with eq

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

- 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`

```
(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))
  )
```

- **Base case**
Recursive Implementation of my-equal

(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) ))

▷ Base case

▷ Recursive case
  ▷ Reduction
  ▷ Composition
Recursive Implementation of *my-equal*

```lisp
(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))
  ))
```

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

(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)))

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

- Original Implementation

```lisp
(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

```lisp
(defun my-equal (s1 s2)
  (or (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))))))
```
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 \((s_1, s_2)\) of lists jointly containing the original elements of \(s\) and the difference in length between \(s_1\) and \(s_2\) is at most 1

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

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

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

- **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

(DEFUN length (L)
    (IF (NULL L) 0 (+ 1 (length (CDR L)))))

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

(DEFUN length (L)
  (IF (NULL L) 0 (+ 1 (length (CDR L))))

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

(LAMBDA (lst1)
  (IF (NULL lst1) 0
       (+ 1
         (LAMBDA (lst2)
           (IF (NULL lst2) 0
                (+ 1
                  (LAMBDA (lst3)
                    (IF (NULL lst3) 0
                        (+ 1
                          (LAMBDA (lst4)
                            . . .
                          (CDR lst3))
                        (CDR lst2))
                    (CDR lst1))
              ) (CDR lst1))
          ) (CDR lst1))
        ))
Recursion as Substitution

(DEFUN length (L)
  (IF (NULL L) 0 (+ 1 (length (CDR L)))))

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

(LAMBDA (lst1)
  (IF (NULL lst1) 0
    (+ 1 (LAMBDA (lst2)
      (IF (NULL lst2) 0
        (+ 1 (LAMBDA (lst3)
          (IF (NULL lst3) 0
            (+ 1 (LAMBDA (lst4)
              . . .
            ) (CDR lst3))
          ) (CDR lst2))
        ) (CDR lst1))
      )))
  ) (CDR lst1)) )
Recursion as Substitution

(DEFUN length (L)
  (IF (NULL L) 0 (+ 1 (length (CDR L)))))

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

(LAMBDA (lst1)
  (IF (NULL lst1) 0
    (+ 1 (LAMBDA (lst2)
      (IF (NULL lst2) 0
        (+ 1 (LAMBDA (lst3)
          (IF (NULL lst3) 0
            (+ 1
              (CDR lst2))
          (CDR lst2))
        (CDR lst1))
      (CDR lst2))
    (CDR lst1))))
Recursion as Substitution

(DEFUN length (L)
  (IF (NULL L) 0 (+ 1 (length (CDR L)))))

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

(LAMBDA (lst1)
  (IF (NULL lst1) 0
   (+ 1 (LAMBDA (lst2)
     (IF (NULL lst2) 0
      (+ 1 (LAMBDA (lst3)
        (IF (NULL lst3) 0
         (+ 1 (LAMBDA (lst4)
           . . .
         ) (CDR lst3))
       ) (CDR lst2))
     ) (CDR lst1))
  ))
Recursion as Self-Referential Variables

\[(\text{LAMBDA (dummy)}\)
\[
\text{(} \)
\]

\[\text{'any-old-value) }\]

- 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
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 )

- 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
Recursion as Self-Referential Variables

\[
\text{( (LAMBDA (dummy)} \\
\text{  ( (LAMBDA (length)} \\
\text{  ) (LAMBDA (L)}} \\
\text{  (IF (NULL L) 0} \\
\text{    (+ 1 (funcall dummy (CDR L)))))))} \\
\text{ ) 'any-old-value }\]

► 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
Recursion as Self-Referential Variables

```lisp
((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 )
```

- 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
Recursion as Self-Referential Variables

( (LAMBDA (dummy)
  ( (LAMBDA (length)
    (SETF dummy length)
    (FUNCALL length '(a b c d))
  )
  )
)
)

\[ \text{\texttt{\textquotesingle any-old-value \texttt{\rightarrow 4}}} \]

- 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

\[
\text{LABELS (}
\text{((length (L)
\text{  (IF (NULL L) 0
\text{  (+ 1 (length (CDR L))))
\text{  )) }) )
\text{ (length '(a b c d)) })
\rightarrow 4
\]

▶ Pure Lisp with LABELS is therefore sufficient to compute any function
LABELS as Self-Referential Variables

- Self-reference requires a SETF
- But variable "dummy" is inside a LAMBDA closure so all side-effects are isolated
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))))))
(l)
(length '(a b c d)) → 4
```
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

  \[
  \text{(LABELS ((length (L))
             (IF (NULL L) 0
                (+ 1 (length (CDR L))))))}
  \text{ (length '(a b c d)) } \rightarrow 4
  \]

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