Functional Programming

Pure Functional Programming

- Computation is largely performed by applying functions to values.
- The value of an expression depends only on the values of its sub-expressions (if any).
  - Evaluation does not produce side effects.
  - The value of an expression cannot change over time.
  - No notion of state.
  - Computation may generate new values, but not change existing ones.
Advantages

• Simplicity
  – No explicit manipulation of memory.
  – Values are independent of underlying machine with assignments and storage allocation.
  – Garbage collection.

• Power
  – Recursion
  – Functions as first class values
    • Can be value of expression, passed as argument, placed in data structures. Need not be named.

Scheme (A Dialog of LISP)

• Primarily an interpreted language.

• Interacting with the interpreter
  – Supply an expression to be evaluated
  – Bind a name to a value (could be a function)

> 3.14159 ; a number evaluates to itself
   3.14159

> (define pi 3.14159) ; bind name to value
   pi

> pi
   3.14159
Expressions

• Prefix notation
  \( \text{op operand1 operand2 operand3 ... } \)
  
  \[
  \begin{array}{c}
  \text{>} \hspace{0.1cm} (* \hspace{0.1cm} 5 \hspace{0.1cm} 7) \\
  \hspace{0.3cm} ; \hspace{0.3cm} 35
  \\
  \text{>} \hspace{0.1cm} (+ \hspace{0.1cm} 4 \hspace{0.1cm} (* \hspace{0.1cm} 5 \hspace{0.1cm} 7)) \\
  \hspace{0.3cm} ; \hspace{0.3cm} 39
  \end{array}
  \]

• General Expression Evaluation
  \( E_1 \ E_2 \ E_3 \ldots \ E_K \)
  – Each \( E_i \) is evaluated (in unspecified order)
  – \( E_i \) must evaluate to a function which is applied to the values of \( E_2 \ldots E_K \)
  – Innermost (call-by-value) evaluation

Expressions (2)

• Uniform syntax is useful for manipulating programs as data.

• Some “special forms” are evaluated differently.
Function Definitions

- Lambda expressions (a special form) evaluate to anonymous functions:

\[
\text{(lambda (formal-parameters) expression)}
\]

> (define square (lambda (x) (* x x)))
; square
> (square 5)
; 25
> ( (lambda (x) (* x x)) 5)
; 25

Quoting

- Quote (a special form) prevents evaluation of its operand:

> (define pi 3.14159)
; pi
> pi
; 3.14159
> (quote pi)
; pi
> 'pi
; pi

> (define f *)
; f
> (f 2 3)
; 6
> (define f '*)
; f
> (f 2 3)
; Error * not a procedure
Lists

- A list is a sequence of zero or more values (symbols, numbers, booleans, functions, other lists).
- A list is written by enclosing its elements in parentheses.
- Some expressions are lists.

List length

```
> (length '() )
; 0
> (length '(a b c) )
; 3
> (length '((a b c)) )
; 1
> (length '((a b) c) )
; 2
> (length 'a)
; 2
> (length '(+ 2 3) )
; 3
```
Operations on Lists

\( \text{(null? } X) \) ; true if \( x \) is the empty list, else false

\( \text{(car } x) \) ; First element in the list \( x \)

\( \text{(cdr } x) \) ; List of all elements of \( x \) except the first

\( \text{(cons } a \ x) \) ; List whose car is \( a \) and whose cdr is \( x \)

\( \text{(append } x \ y) \) ; The concatenation of lists \( x \) and \( y \)

\( \text{(list } e_1 \ e_2 \ldots \ e_k) \) ; List of values of \( e_1 \ldots e_k \)

- Operators do not change lists:

\[ (> \text{(define } l \ ‘(b c d))) \]
\[ (> l) \]
\[ (> \text{(cons } 'a \ l)) \]
\[ (> (car '(a b c))) \]
\[ (> l) \]

\[ (> \text{(define } l \ ‘(b c d))) \]
\[ (> \text{(null? } 'a)) \]
\[ (> \text{(null? } '(a))) \]
\[ (> \text{(null? } '())) \]
\[ (> \text{(car } '(a b c))) \]
\[ (> \text{(car } '((a b) c))) \]
\[ (> \text{list } e_1 \ e_2 \ldots \ e_k) \]

Operations on Lists (2)

\[ (> \text{(null? } 'a)) \]
\[ (> \text{(cdr } '(a b c))) \]
\[ (> \text{(null? } 'a)) \]
\[ (> \text{(null? } '(a))) \]
\[ (> \text{(null? } '())) \]
\[ (> \text{(car } '(a b c))) \]
\[ (> \text{(car } '((a b) c))) \]
\[ (> \text{(list } 'a \ (+\ 3\ 4) \ 'b)) \]
Examples: factorial, length

(define factorial
  (lambda (n)
    (if (equal? n 0)
      1
      (* n (factorial (- n 1))))))

(define length
  (lambda (x)
    (if (empty? x)
      0
      (+ 1 (length (cdr x))))))

Example: flatten

(define flatten
  (lambda (x)
    (cond ((null? x) x)
          ((list? (car x))
           (append (flatten (car x))
                  (flatten (cdr x))))
          (else (cons (car x)
                      (flatten (cdr x)))))))

> (flatten '(a (b (c) d) e) )
; (a b c d e)
Scoping

• Let special form
\[
(\text{let}
  ((x_1 e_1) (x_2 e_2) \ldots (x_k e_k))
  f_1 f_2 \ldots f_n)
\]
  1. Each \( e_i \) is evaluated (in parallel)
  2. Each \( f_j \) is evaluated with \( x_i \) denoting the value of \( e_i \)
  3. The result is the value of \( f_n \)

• Let* special form (sequential let)
\[
(\text{let*}
  ((x_1 e_1) (x_2 e_2) \ldots (x_k e_k))
  f_1 f_2 \ldots f_n)
\]
  Bindings to \( x_i \)'s take place sequentially. That is, 
  \( e_{i+1} \) is evaluated with the binding of \( x_i \) to \( e_i \) already in effect.

Scope Examples

> (+ (square 3) (square 3))
; 18
> (let ((sq3 (square 3))) (+ sq3 sq3))
; 18
> (define x 0)
; x
> (let ((x 2) (y x)) y)
; 0
> (let* ((x 2) (y x)) y)
; 2
Lexical Scoping

> (define g
  (lambda (f)
    (let ((x 5)) (f) )))

> (let ((x 0))
  (g (lambda () (+ 1 x)) ))
; 1

Naming vs. Assignment

C

int f(int i)
{
  int x = 0;
  if (i > 10)
    x = 1;
  else    
    x = 2;
  return x;
}

Scheme

(define f
  (lambda (i)
    (let ((x 0))
      ; arbitrary code
      ; ...
      x)))
; What does f return?
Naming vs. Assignment (2)

Scheme

(define f
  (lambda (i)
    (let ((x 0))
      (if (> i 10)
        (let ((x 1)) x)
        (let ((x 2)) x)))
    x))

; What does f return?

C

int f(int i)
{
  int x = 0;
  if (i > 10) {
    int x = 1;
  }
  else {
    int x = 2;
  }
  return x;
}

Efficiency: Cons vs. Append

> (define x1 '())
> (define x2 (cons 'c x1))
> (define x3 (cons 'b x2))
> (define x4 (cons 'a x3))

> x1
; ()
> x2
; (c)
> x3
; (b c)
> x4
; (a b c)

Cons cells
Conditionals

- Booleans
  
  \#t \#f

- Predicates
  
  number? symbol? list? equal? etc.

- If special form
  
  (if P E_1 E_2)

- Cond special form
  
  (cond (P_1 E_1) (P_2 E_2) ... (P_k E_k)
         (else E_{k+1}))

Efficiency: Cons vs. Append (2)

> (define x1 '(a))
> (define x2
      (append x1 '(b)))
> (define x3
      (append x2 '(c)))

> x1
  ; (a)
> x2
  ; (a b)
> x3
  ; (a b c)

Cons cells

Efficiency: Cons vs. Append (2)
Efficiency: Tail Recursion

• When a recursive function returns the result of its recursive call, there is no need to maintain a stack of activation records.

```
(define last (lambda (x)
    (cond ((null? x) '())
          ((null? (cdr x)) (car x))
          (else (last (cdr x))))))
```

Efficiency: Tail Recursion (2)

```
(define length (lambda (x)
    (if (null? x) 0 (+ 1 (length (cdr x))))))

(define length (lambda (x) (length-help x 0)))

(define length-help
    (lambda (x n)
        (if (null? x)
            n
            (length-help (cdr x) (+ 1 n))))))
```
Applications: Insertion Sort

(define insert
   ;; insert a number, n, into a sorted list, x.
   (lambda (n x)
      (cond ((null? x) (list n))
            ((< n (car x)) (cons n x))
            (else (cons (car x) (insert n (cdr x)))))))

(define insertion-sort
   (lambda (x)
      (cond ((null? x) x)
            (else (insert (car x) (insertion-sort (cdr x)))))))

Applications: Merge Sort

; Merge two sorted lists
(define merge (lambda (x y)
   (cond ((null? x) y)
         ((null? y) x)
         ((< (car x) (car y)) (cons (car x) (merge (cdr x) y)))
         (else (cons (car y) (merge x (cdr y)))))))
Applications: Merge Sort (2)

(define split (lambda (x) (split-help x '())))

(define split-help
  (lambda (x y)
    (if (<= (length x) (length y))
      (list x y)
      (split-help (cdr x) (cons (car x) y)))))

Applications: Merge Sort (3)

; Mergesort a list
(define mergesort (lambda (x)
    (if (< (length x) 2) x (let* ((halfs (split x))
      (r1 (mergesort (car halfs)))
      (r2 (mergesort (cadr halfs)))
      (merge r1 r2))))))
Association Lists

- List of key/value pairs

```lisp
(assoc exp x) ; Find the first pair in x whose
               ; key has the same value as exp

> (define alist `((a 1) (b 2) (c 3) (b 4)))
 ; alist
> (assoc 'b alist)
 ; (b 2)
```

Higher Order Functions: Map

- Let \( l_1, l_2, \ldots, l_k \) be lists of equal length, \( n \). Let \( a_{i,j} \) be the \( j \)th element of \( l_i \). Then:

```lisp
(map f l_1 l_2 \ldots l_k) = 
  ((f a_{1,1} a_{2,1} \ldots a_{k,1}) (f a_{1,2} a_{2,2} \ldots a_{k,2})
   \ldots (f a_{1,n} a_{2,n} \ldots a_{k,n}))
```

```lisp
> (map + '(1 2 3) '(4 5 6))
 ; (5 7 9)
> (map cons '(a b c) '(() (c d) (e)))
 ; ((a) (b c d) (c e))
> (map + '(1 2 3) '(4 5 6) '(7 8 9))
 ; (12 15 18)
> (map list '(a b c))
 ; ((a) (b) (c))
```
Higher Order Functions (2)

- How to write a tail recursive version of length without adding helper functions to top level namespace?

\[
(\text{define length}
  \ (\text{lambda} \ (x))
  \ (\text{let} \ ((\text{helper} \ (\text{lambda} \ (x \ n))
    \ (\text{if} \ (\text{null?} \ x)
      \ n
      \ (\text{helper} \ (\text{cdr} \ x) \ (+ \ 1 \ n))))))
  \ (\text{helper} \ x \ 0)))
\]

- But this doesn’t work. Scheme does not allow you to use a symbol before it is bound in a let, so we can’t make the recursive call.

Higher Order Functions (3)

- Add an additional parameter to the helper function for the function to call.
- Pass the helper function to itself (after it is defined), so that it can make the recursive call.
- Each time a recursive call is made, the helper function is passed along so the recursion can continue.

\[
(\text{define length}
  \ (\text{lambda} \ (x))
  \ (\text{let} \ ((\text{helper} \ (\text{lambda} \ (\text{help} \ x \ n))
    \ (\text{if} \ (\text{null?} \ X)
      \ n
      \ (\text{help} \ \text{helper} \ (\text{cdr} \ x) \ (+ \ 1 \ n))))))
  \ (\text{helper} \ \text{helper} \ x \ 0)))
\]
Example: Searching

- A directed acyclic graph in Scheme

```scheme
(define g '(((a (b c d)) (b ()) (c (e)) (d (f g)))
           (e ()) (f (h i j)) (g ()) (h ()))
           (i ()) (j ()))))
```

```
A
B  C  D
  E  F  G
    H I  J
```

Searching (2)

```scheme
(define successors
  (lambda (node graph)
    (cadr (assoc node graph))))

(define path-extensions
  (lambda (path graph)
    ;; find all one node extensions of the path
    (map (lambda (node) (cons node path))
         (successors (car path) graph)))))

(define expand (lambda (graph goal paths)
    (if (null? paths) '
      (let ((first-path (car paths))
            (remaining-paths (cdr paths)))
        (if (equal? goal (car first-path))
            first-path
            (expand graph goal
                     (append (path-extensions first-path graph) remaining-paths))))))

(define search (lambda (graph start goal)
                 (reverse (expand graph goal (list (list start)))))
```
Example: Expression Evaluator

- Consider the following grammar for expressions with “scheme-like” semantics:

\[
E ::= (\text{num \ val})
| (\text{var \ symbol})
| (\text{plus \ } E \ E)
| (\text{times \ } E \ E)
| (\text{let \ (params) \ } E)
\]

\[
\text{params ::= (symbol \ } E) | (\text{symbol \ } E \ \text{params})
\]
Expression Evaluator (2)

(define lookup
  (lambda (x env)
    (cadr (assoc x env)))))

(define bind
  (lambda (params env)
    (append
     (map (lambda (y)
           (list (car y)
                 (eval (cadr y) env))) params)
     env)))

Expression Evaluator (3)

(define eval
  (lambda (exp env)
    (cond
     ((equal? (car exp) 'num) (cadr exp))
     ((equal? (car exp) 'var)
      (lookup (cadr exp) env))
     ((equal? (car exp) 'plus)
      (+ (eval (cadr exp) env)
          (eval (caddr exp) env)))
     ((equal? (car exp) 'let)
      (eval (caddr exp)
        (bind (cadr exp) env)))))))
Expression Evaluator (4)

> (eval '(let ((x (num 2)) (y (num 3)))
    (plus (var x) (var y))) '())
  ; 5

> (eval '(let ((x (num 1)))
    (plus
      (let ((x (num 2)) (y (num 3)))
        (plus (var x) (var y)))
      (var x))) '())
  ; 6