Today

- Administrative
  - Weekly Survey
  - Assignment 2

- Roadmap
  - Continuations
  - High-order Functions
  - Examples
  - Algebraic Proofs

Scheme: higher order func - map

- A higher order function used to apply another function to every element of a list:
  - Syntax: (map <func> <arg-list>)
  - Arguments: a function and arguments lists
  - <func> must be a function taking as many arguments as there are in <arg-list> and returning a single value
  - The dynamic order in which <func> is applied to the elements of the <arg-list> is unspecified.

  Examples:
  - (map + '(1 2 3) '(4 5 6))
    (5 7 9)
  - (map (lambda (x) (+ 1 x)) '(1 2 3))
    (2 3 4)
  - (map abs '(-1 2 -3 -4))
    (1 2 3 4)

Scheme: higher order func - for-each

- Another high order function (map)

  Syntax: (for-each <func> list)
  - Guaranteed to apply function <func> to each element of a list.
  - Value returned by for-each is unspecified.

  Implementation:
  - (for-each proc list)
  - (cond ((null? list) (for-each proc '())) (else (proc (car list)) (for-each proc (cdr list)))))

  Example:
  - (for-each display 'a b c d e f ()
    abcd ef)

Scheme: higher order func - reduce

- A high order function (we have already seen map)

  Syntax: (reduce func initial-list)
  - Reduce a list of values to a single value by repeatedly applying a binary function to the list of values

  Implementation:
  - (define (reduce func init-list)
    (if (null? init-list) id
      (func (car init-list) (reduce func (cdr init-list)))))

  Example:
  - (reduce + '(1 2 3 4 5) 0) ; 15
  - (reduce * '(1 2 4 6 8 10) 1) ; 3840
  - (reduce append '()' '(1 2 3) '(4 5 6) '(7 8)) ; '(1 2 3 4 5 6 7 8)
  - (reduce expt '()' '(2 2 2 2) 1) ; 65536

Scheme: vectors

- Another compound structure (lists, paris)

  Syntax: (vector <v-list>)
  - Similar to lists, it can hold heterogeneous elements

  What’s the problem with lists?
  - Access time ... Vectors are accessed in constant time.

  Example:
  - (make-vector 5) ; creates a vector of length 5 initialized to 0
  - (vector-length (make-vector 10)) ; 10
  - (vector-ref (make-vector 10) 2) ; returns the third element of
  - (vector-set! (make-vector 10) 2 'x) ; replaces element 2 by x
  - (vector-fill! (make-vector 10) 'x) ; fills each element with x
  - (vector->list (make-vector 10) 'x) ; returns list (x x x ... x)
  - (list->vector '(a b c)) ; returns vector 'a b c'
  - (for-each (lambda (x) (vector-set! (make-vector 10) 2 x)))
Scheme: continuations

• What is a continuation?
  – The current continuation at any point in the execution of a program is an abstraction of the rest of the program.
  – A continuation of the evaluation of an expression \( E \) in a surrounding context \( C \) represents the entire future of the computation, which waits for the value of \( E \).
  – E.g.

<table>
<thead>
<tr>
<th>Context C and expression E</th>
<th>Intuitive continuation of E in C</th>
</tr>
</thead>
<tbody>
<tr>
<td>(define x 5)</td>
<td><code>(lambda (e) (+ x 1))</code></td>
</tr>
<tr>
<td>(define x 5)</td>
<td><code>(remainder (+ x 1) x)</code></td>
</tr>
<tr>
<td>(define x 5)</td>
<td><code>(lambda (e) (cons 1 (cons 2 (cons 3 e))))</code></td>
</tr>
<tr>
<td>(define x 5)</td>
<td><code>(lambda (e) (+ 5 e))</code></td>
</tr>
</tbody>
</table>

- A more precise notation of the continuation of \( E \)

<table>
<thead>
<tr>
<th>Context C and expression E</th>
<th>Intuitive continuation of E in C</th>
</tr>
</thead>
<tbody>
<tr>
<td>(define x 5)</td>
<td><code>(lambda (e) (+ x 1)) (lambda (e) (remainder (+ x 1) x)) (lambda (e) (cons 1 (cons 2 (cons 3 e)))) (lambda (e) (+ 5 e))</code></td>
</tr>
</tbody>
</table>

- Scheme provides a primitive that captures a continuation of an expression \( E \) in a context \( C \)

• Syntax

<table>
<thead>
<tr>
<th>call-with-current-continuation (func (param))</th>
<th>call/cc (func (param))</th>
</tr>
</thead>
<tbody>
<tr>
<td>call/cc takes a parameter which is a receiver function of one parameter.</td>
<td></td>
</tr>
<tr>
<td>call/cc captures its own continuation and passes it to the receiver, i.e., call/cc applies the receiver to its own continuation.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Context C and expression E</th>
<th>Value of C</th>
<th>Application of call/cc</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(define x 5)</td>
<td><code>(lambda (e) (+ x 1))</code></td>
<td>(call/cc (lambda (cont) (set! aFuture cont) 8))</td>
<td>10</td>
</tr>
<tr>
<td>(define error '())</td>
<td><code>()</code></td>
<td>(call/cc (lambda (cont) (set! error cont) (even? 9)))</td>
<td><code>()</code></td>
</tr>
</tbody>
</table>

- E.g.

<table>
<thead>
<tr>
<th>Context C and expression E</th>
<th>Context C and the capturing</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(define aFuture '())</code></td>
<td><code>(lambda (e) (remainder (+ x 1) x))</code></td>
</tr>
<tr>
<td><code>(define x 5)</code></td>
<td><code>(lambda (e) (+ x 1))</code></td>
</tr>
<tr>
<td><code>(define aFuture '())</code></td>
<td><code>(lambda (e) (remainder (+ x 1) x))</code></td>
</tr>
<tr>
<td><code>(define x 5)</code></td>
<td><code>(lambda (e) (+ x 1))</code></td>
</tr>
</tbody>
</table>

- How to use call/cc to change continuation?
  1. We make \( n \) the body of a function.
  2. Let us have one argument passed to that function.
  3. Let us call this argument "return".
  4. Use call/cc.

- Primary usage:
  - Exception handling
  - Backtracking

- E.g.

<table>
<thead>
<tr>
<th>Context C and expression E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(define aFuture '())</code></td>
<td><code>()</code></td>
</tr>
<tr>
<td><code>(define x 5)</code></td>
<td><code>()</code></td>
</tr>
<tr>
<td><code>(define aFuture '())</code></td>
<td><code>()</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Context C and expression E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(define aFuture '())</code></td>
<td><code>()</code></td>
</tr>
<tr>
<td><code>(define x 5)</code></td>
<td><code>()</code></td>
</tr>
<tr>
<td><code>(define aFuture '())</code></td>
<td><code>()</code></td>
</tr>
</tbody>
</table>

- Scheme: continuations

• Syntax

<table>
<thead>
<tr>
<th>call/cc(func (param))</th>
</tr>
</thead>
<tbody>
<tr>
<td>call/cc takes a parameter which is a receiver function of one parameter.</td>
</tr>
<tr>
<td>call/cc captures its own continuation and passes it to the receiver, i.e., call/cc applies the receiver to its own continuation.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Context C and expression E</th>
<th>Value of C</th>
<th>Application of call/cc</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(define x 5)</code></td>
<td><code>(lambda (e) (+ x 1))</code></td>
<td>(call/cc (lambda (cont) (set! aFuture cont) 8))</td>
<td>10</td>
</tr>
<tr>
<td><code>(define error '())</code></td>
<td><code>()</code></td>
<td>(call/cc (lambda (cont) (set! error cont) (even? 9)))</td>
<td><code>()</code></td>
</tr>
</tbody>
</table>

- Scheme: continuations

• Syntax

<table>
<thead>
<tr>
<th>call-with-current-continuation (func (param))</th>
<th>call/cc (func (param))</th>
</tr>
</thead>
<tbody>
<tr>
<td>call/cc takes a parameter which is a receiver function of one parameter.</td>
<td></td>
</tr>
<tr>
<td>call/cc captures its own continuation and passes it to the receiver, i.e., call/cc applies the receiver to its own continuation.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Context C and expression E</th>
<th>Value of C</th>
<th>Application of call/cc</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(define x 5)</code></td>
<td><code>(lambda (e) (+ x 1))</code></td>
<td>(call/cc (lambda (cont) (set! aFuture cont) 8))</td>
<td>10</td>
</tr>
<tr>
<td><code>(define error '())</code></td>
<td><code>()</code></td>
<td>(call/cc (lambda (cont) (set! error cont) (even? 9)))</td>
<td><code>()</code></td>
</tr>
</tbody>
</table>

- Scheme: continuations

• Syntax

<table>
<thead>
<tr>
<th>call-with-current-continuation (func (param))</th>
<th>call/cc (func (param))</th>
</tr>
</thead>
<tbody>
<tr>
<td>call/cc takes a parameter which is a receiver function of one parameter.</td>
<td></td>
</tr>
<tr>
<td>call/cc captures its own continuation and passes it to the receiver, i.e., call/cc applies the receiver to its own continuation.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Context C and expression E</th>
<th>Value of C</th>
<th>Application of call/cc</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(define x 5)</code></td>
<td><code>(lambda (e) (+ x 1))</code></td>
<td>(call/cc (lambda (cont) (set! aFuture cont) 8))</td>
<td>10</td>
</tr>
<tr>
<td><code>(define error '())</code></td>
<td><code>()</code></td>
<td>(call/cc (lambda (cont) (set! error cont) (even? 9)))</td>
<td><code>()</code></td>
</tr>
</tbody>
</table>
Scheme: continuations

- E.g.
  ```lisp
  => (define some-flag #t)
  => (define (my-abortable-proc escape-proc)
      (display "in my-abortable-proc ")
      (if some-flag
          (escape-proc "ABORTED")
          (display " still in my-abortable-proc ")
          " NOT ABORTED ")
  => (my-abortable-proc display)
  in my-abortable-proc ABORTED still in my-abortable-proc NOT ABORTED
  ```

- Another control structure (we have already seen cond and if)

- Syntax:
  ```lisp
  (begin exp_0 exp_1 ... exp_n)
  ```
  - expressions are evaluated from left to right
  - The value of the rightmost expression is the value of the entire expression. Other expressions are for side-effects only

- Example:
  ```lisp
  => (define x 3)
  => (begin
      (set! x (+ x 1))
      (+ x x) ) ; returns 8
  ```

Scheme: I/O

- Input/output
  - (read ...) ; reads and returns an expression
  - (read-char ...) ; reads and returns a character
  - (peek-char ...) ; returns next available character w/o updating
  - (char-ready? ...) ; returns #t if char has been entered
  - (write OBJECT ...) ; outputs the object (strings are in quotes, )
  - (newline) ; outputs end-of-line

- Reading/writing files
  - (open-input-file)  
  - (open-output-file)

- Example:
  ```lisp
  > (display "hello world ")
  > (define r (read ))
  > (display r)
  ```

Scheme: assoc function

- assoc does lookup in a list

- E.g:
  ```lisp
  => (define NAMES '( (Smith Pat Q) (Jones Chris J) (Walker Kelly T) (Thompson Shelby P)))
  => (assoc 'Smith NAMES)
  (Smith Pat Q)
  => (assoc 'Walker NAMES)
  (Walker Kelly T)
  ```

- assoc returns the first sublist if more than one sublist with the same key exist.

Scheme: I/O - cont’d

- Reading an input file
  ```lisp
  (define (echo filename)
    (let ( (p (open-input-file filename ) ) )
      ( let loop ( ( obj (read p)))
        (if (eof-object? obj ) #t
            (begin
              (write obj )
              (newline)
              (loop ( read p) ) ) ) ) ) )
  ```

Scheme: tree example

- Constructing a tree:
  ```lisp
  => (define TREE
      '(dog
        (bird (aardvark () ()) (cat () ()) )
        (possum (frog () ()) (wolf () ()) ) )
  ```

- Simple tree functions: (empty?, root, left-subtree, right-subtree)
  ```lisp
  => (defineempty? (λ (tree) (null? tree))
    (λ (tree) (null? (cadr tree)))
    (λ (tree) (null? (caddr tree)))
  )
  ```

- Error handling:
  ```lisp
  => (define-error-term (λ (tree) #t)
    (λ (tree) (empty? tree))
    (λ (tree) (error term)))
  ```
Scheme: tree example - cont’d

• Tree searching:
  ```scheme
  (define contains?
    (lambda (tree sym)
      (cond ((empty? tree) #f)
            ((equal? (root tree) sym) #t)
            (else (or (contains? (left-subtree tree))
                      (contains? (right-subtree tree)))))))
  ```

• Alternate tree searching:
  ```scheme
  (define bst-contains?
    (lambda (tree sym)
      (cond ((empty? tree) #f)
            ((= (root tree) sym) #t)
            ((> (root tree) sym) (bst-contains? (left-subtree tree) sym))
            (else (bst-contains? (right-subtree tree) sym)))))
  ```

Scheme: tree example - cont’d

• Tree traversal:
  ```scheme
  (define pre-order
    (lambda (tree)
      (if (null? tree) '()
          (append (list (car tree))
                  (pre-order (cadr tree))
                  (pre-order (caddr tree)))))
  ```

  ```scheme
  (define in-order
    (lambda (tree)
      (if (null? tree) '()
          (append (in-order (cadr tree))
                  (list (car tree))
                  (in-order (caddr tree)))))
  ```

  ```scheme
  (define post-order
    (lambda (tree)
      (if (null? tree) '()
          (append (post-order (cadr tree))
                  (post-order (caddr tree))
                  (list (car tree)))))
  ```

  ```scheme
  (define TREE
    (dog bird aardvark cat possum frog wolf))
  ```

• Backtracking:
  - It is an approach to find a target path in a tree, e.g. just a path, all paths, the shortest path

Software Verification

• Software crisis
  - Quality of software?

• Enforce roles/responsibilities
  - Architects ➔ Programmers ➔ QA
  - QA test the program on many different inputs
  - However, subtle bugs may remain undiscovered, only to appear at random, inconvenient or dangerous moments.

• Implement quality processes (e.g. CMM, ISO…)
  - However, bugs may remain undiscovered

• Use Algebraic proofs
  - Especially useful for safety-critical software (e.g. Air control, …)

Software Verification: proving properties of programs

• Example:
  ```scheme
  (define (append X Y)
    (if (null? X)
        Y
        (cons (car X) (append (cdr X) Y))))
  ```

  ```scheme
  (define (length X)
    (if (null? X) 0
        (+ 1 (length (cdr X)))))
  ```

  Theorem: \((\text{length } (\text{append } X X)) = (\text{length } X) + (\text{length } X)\) for all lists \(X\) and \(Y\)
Software Verification: prove outline

- Use mathematical induction on the length of X
  - A two-part method of proving a theorem involving an integral parameter.
    - First, the theorem is verified for the smallest admissible value of the integer.
    - Then it is proven that if the theorem is true for any value of the integer, it is true for the next greater value. The final proof contains the two parts.
  - First, prove that the theorem is true for lists of length 0 [Basis]
  - Then, prove that if the theorem is true for lists of length $N$, then it is also true for lists of length $N+1$ [Inductive Step]
  - This implies that the theorem is true for lists of any length (i.e., for any list)

Software Verification: structural induction

- Actually, we will use a variation of induction that emphasizes the structure of lists, not their length.
  - First, prove that the theorem is true for $X = '()$ i.e., when $X$ has length 0 [Basis]
  - Then, prove that if the theorem is true for $X = L$, then it is also true for $X = (\text{cons} \ E \ L)$ [Inductive Step]
    - Note: if $L$ has length $N$, then $(\text{cons} \ E \ L)$ has length $N + 1$

Software Verification: preliminaries

- Before using induction (or any other technique) to prove a complex property of a program, write down the basis properties that can be trivially verified by inspecting the program code
  - The inductive proof should only use these properties of the code.
  - The code itself plays no other role in a proof of correctness and can be henceforth ignored.
  - Note: If the basic properties are wrong, then the entire proof is wrong. So be sure to get them right!!

Software Verification: basic properties of append

- Code:
  ```scheme```
  ```
  (define (append X Y)
    (if (null? X) Y
     (cons (car X) (append (cdr X) Y))))
  ```
  ```
  - Using $X = '()$ (append '() Y) = Y [1]
  - Using $X = (\text{cons} \ E \ L)$ (append (cons E L) Y) = (cons E (append L Y)) [2]
  ```
  ```
  - Note:
    - if $X = (\text{cons} \ E \ L)$, then $(\text{car} \ X) = E$, $(\text{cdr} \ X) = L$
  ```

Software Verification: basic properties of length

- Code:
  ```scheme```
  ```
  (define (length X)
    (if (null? X) 0
     (+ 1 (length (cdr X))))
  ```
  ```
  - Using $X = '()$ (length '()) = 0 [3]
  - Using $X = (\text{cons} \ E \ L)$ (length (cons E L)) = 1 + (length L) [4]
  ```
  ```
  - Note:
    - if $X = (\text{cons} \ E \ L)$, then $(\text{cdr} \ X) = L$
  ```

Software Verification: summary of basic properties

- (append '() Y) = Y [1]
- (append (cons E L) Y) = (cons E (append L Y)) [2]
- (length '()) = 0 [3]
- (length (cons E L)) = 1 + (length L) [4]
- Using these basic properties, we shall prove (by induction) a more complex property:
  - Theorem: (length (append X Y)) = (length X) + (length Y)
Software Verification: basis proof

- Using induction on the structure of X

  - **Basic:** when X = ' ()

    \[
    \begin{align*}
    (\text{length} \ (\text{append} \ X \ Y)) &= (\text{length} \ (\text{append} \ '() \ Y)) \\
    &= (\text{length} \ Y) \quad \text{[by 1]} \\
    &= 0 + (\text{length} \ Y) \\
    &= (\text{length} \ X) + (\text{length} \ Y) \quad \text{[by 3]} \\
    
    \end{align*}
    \]

Therefore, the theorem is true when X = ' ()

Software Verification: inductive step

- Suppose that the theorem holds for X = L, i.e. suppose that
  \[(\text{length} \ (\text{append} \ L \ Y)) = (\text{length} \ L) + (\text{length} \ Y)\]
  \(\text{Inductive hypothesis}\)

- Now, prove that the theorem holds for X = (cons E L)

Software Verification: proof of inductive step

- If X = (cons E L) then

  \[
  \begin{align*}
  (\text{length} \ (\text{append} \ X \ Y)) &= (\text{length} \ (\text{append} \ (\text{cons} \ E \ L) \ Y)) \\
  &= (\text{length} \ (\text{cons} \ E \ (\text{append} \ L \ Y))) \quad \text{[by 2]} \\
  &= 1 + (\text{length} \ L) + (\text{length} \ Y) \quad \text{[by 4]} \\
  &= (\text{length} \ (\text{cons} \ E \ L) + (\text{length} \ Y) \quad \text{[by inductive hypothesis]} \\
  &= (\text{length} \ X) + (\text{length} \ Y) \quad \text{[by 4]} \\
  
  \end{align*}
  \]

Software Verification: proving properties of programs

- Example 2:

  \[
  \begin{align*}
  \text{(define} \ (\text{member} \ A \ X) \ &= \text{cond} \ (\text{null?} \ X) \ #f \\
  \text{(equal?} \ A \ (\text{car} \ X)) \ #t \\
  \text{(else} \ (\text{member} \ A \ (\text{cdr} \ X)))) \\
  
  \end{align*}
  \]

  - Prove the following:
    \[
    \begin{align*}
    \text{Theorem: } &\text{If } (\text{member} \ A \ X) \text{ then} \\
    & (\text{member} \ A \ (\text{append} \ X \ Y)) \text{ for any } A, \text{ and any lists } X \text{ and } Y. \\
    
    \end{align*}
    \]

Software Verification: outline of proof

- As before, we will use structural induction on X
  - First, prove that the theorem is true for X = ' () [basis]
  - Then, prove that if the theorem is true for X = L, then it is true for X = (cons E L) [inductive step]
  - However, this time proof will be more complex, because the program can terminate its recursion in two ways.
Software Verification: basic properties of member

- Code:

  (define (member A X)
      (cond ((null? X) #f)
            ((equal? A (car X)) #t)
            (else (member A (cdr X)))))

- Using X = '() (member A '()) = #f  [5]

- Using X = (cons A L) (member A (cons A L)) = #t  [6]

- Note:
  - (car X) = A

Software Verification: basic properties of member – cont’d

- Code:

  (define (member A X)
      (cond ((null? X) #f)
            ((equal? A (car X)) #t)
            (else (member A (cdr X)))))

- Using X = (cons E L) where E != A (member A (cons E L)) = (member A L)  [7]

- Note:
  - (car X) = E, (cdr X) = L

Software Verification: summary of basic properties

- member Code:
  - (member A '()) = #f  [5]
  - (member A (cons A L)) = #t  [6]
  - If E=A then
    (member A (cons E L)) = (member A L)  [7]

- append Code:
  - (append '() Y) = Y  [1]
  - (append (cons E L) Y) = (cons E (append L Y))  [2]

Software Verification: proof of theorem

- Theorem: If (member A X) then (member A (append X Y))

- Proof: by induction on X

  - Basis: If X = '() then
    (member A X)
    = (member A '())
    = #f  [by 5]

     Therefore, the premise of the theorem is false.
     Therefore, the theorem is trivially true (for X = '())

Software Verification: inductive step

- Suppose the theorem is true for X=L, i.e.
  if (member A L) then
    (member A (append L Y))  [Inductive hypothesis]

- Now, prove the theorem is true for X = (cons E L)

Software Verification: proof of inductive step

- Let X = (cons E L), there are two cases

  - Case 1: E = A

      Therefore, X = (cons A L)
      Therefore, (member A (append X Y))
      = (member A (append (cons A L) Y))  [by 2]
      = (member A (cons A (append L Y)))  [by 6]

     Therefore, the conclusion of the theorem is true
     Therefore, the theorem itself is trivially true for X = (cons A L)
Software Verification: proof of inductive step – cont’d

- Let \( X = (\text{cons} \ E \ L) \), there are two cases

- Case 2: \( X = (\text{cons} \ E \ L) \) where \( E \neq A \)

  \[
  \text{if} (\text{member} \ A \ X) = \#t \ \text{then}
  \]
  \[
  (\text{member} \ A \ (\text{cons} \ E \ L))
  \]
  \[
  = \ ??
  \]
  \[
  = (\text{member} \ A \ L)
  \]
  \[
  = (\text{member} \ A \ (\text{append} \ L \ Y)) \quad \text{[by inductive hypothesis]}
  \]
  \[
  = (\text{member} \ A \ (\text{append} \ E \ L \ Y)) \quad \text{[by 7]}
  \]
  \[
  = (\text{member} \ A \ (\text{append} \ (\text{cons} \ E \ L) \ Y)) \quad \text{[by 2]}
  \]
  \[
  = (\text{member} \ A \ (\text{append} \ X \ Y))
  \]

  Therefore, the theorem is true when \( X = (\text{cons} \ E \ L) \) and \( E \neq A \)

Software Verification: summary of proof of theorem

- For any \( A \) and any list \( Y \)

- Basis
  
  The theorem holds for \( X = '() \)

- Inductive Step:
  
  If the theorem holds for \( X = L \), then it holds for \( X = (\text{cons} \ E \ L) \)
  
  Therefore, by the principle of structural induction, the theorem holds for any \( A \) and \( Y \) lists \( X \) and \( Y \).

- Theorem:
  
  If \( (\text{member} \ A \ X) \) then
  
  \( (\text{member} \ A \ (\text{append} \ X \ Y)) \)

Software Verification

- Interested to learn more...
  
  - CSC465: Formal Methods in Software Design