CSI2120 Programming Paradigms

Jochen Lang

jlang@uottawa.ca

Scheme: Functional Programming

- Input/Output in
- Vectors in Scher
- Looping with do
- Sorting

Jochen Lang, EECS
jlang@uOttawa.ca

Input/Output

- display – prints [illegible]een (REPL buffer)
  - (display "hell[illegible]
  - hello world
- read – function that returns keyboard entries
  - Reads a line from the REPL buffer and returns; nothing printed, e.g.:
    `(read)` *type* `234` *return*
  - Combine with display
    `(display (read))` *type* `"hello world"` *return*
    prints `hello world`
- newline – for formatted output

Example: Number Entry

- Function that reads inputs, tests for a number until one is received.

```
(define (ask-number)
  (display "Enter a number: ")
  (let ((n (read)))
    (if (number? n) n (ask-number))))
=> ask-number
(ask-number)
Enter a number: type "Hello"
Enter a number: type Hello
Enter a number: type (+ 3 54)
Enter a number: type 2
prints 2
```

Reading a File into a List

- File i/o works as
  - File streams / ports
  - Different flavours of file commands, here: open-input-file (others include open-input-output-file and open-output-file), close-input-port

```
(let ((p (open-input-file "short.scm")))
  (let f ((x (read p)))   ; reading from file
    (if (eof-object? x)   ; check for eof
        (begin
          (close-input-port p)
          '())
        (cons x (f (read p))))))
```

File I/O Inside a Top-level Define

- Function that op[illegible] and applies a procedure to every token rea[illegible]
  - Two arguments filename and proc

```
(define proc-in-file
  (lambda (filename proc)
    (let ((p (open-input-file filename)))
      (let ((v (proc p)))
        (close-input-port p)
        v))))
```

  - Note that procedure is applied to the path
    - must read in supplied procedure proc

Example: Output to File with a Top-level Define

- Write to file `fil` the output from the function `proc`

```
(define proc-out-file
  (lambda (filename proc)
    (let ((p (open-output-file filename)))
      (let ((v (proc p)))
        (close-output-port p)
        v))))
```

  - `proc-out-file` only opens and closes file
    - must write in supplied procedure proc

Printing a List to File

- Call proc-out-file with a function that recursively goes over the list
  - Define a lambda function to proc-out-file
  - Here the lambda makes use of a named let expression

```
(proc-out-file "list.out"
  (lambda (p)
    (let ((list-to-be-printed '(1 2 3 4)))
    (let f ((l list-to-be-printed))
      (if (not (null? l))
          (begin
            (write (car l) p)
            (newline p)
            (f (cdr l)))))))))
```

Iteration: do

- (do ((*var init update*) ...) (*test resultIfTrue* ...) *exprIfTestFalse* ...)

```
(define fibonacci
  (lambda (n)
    (if (= n 0)
        0
        (do ((i n (- i 1))          ; i=1, --i
             (a1 1 (+ a1 a2))       ; a1=1, a1+=a2
             (a2 0 a1))             ; a2=0, a2=a1
            ((= i 1) a1)))))

(fibonacci 6)
=> 8
```

Vectors in Scheme

- Vectors in Schem rogeneous structures that allow access to the var nts using an index.
  - requires less an lists
  - elements access is constant time
- Basic commands
  - create a vector with k elements
    `(make-vector` *k* `)`
  - create with an initial value for the elements
    `(make-vector` *k init* `)`
  - test if an object is a vector
    `(vector?` *obj* `)`
  - make a constant vector
    `'#(` *element_0 ... element_k-1* `)`
- Ex: `'#(0 (2 2 2 2) "Anna")`

Constructors and Accessors

- Top-level define for

```
(vector     'a 'b
=> #(a b c)
(define v (vector 2 (+ 1 2)))
=> v
v
=> #(2 3)
(vector-ref  v 0)   ; index starts at 0
2
(vector-length  v)
2
(vector-set! v 1 10)
=> #(2 10)
```

More Examples of vector-set!

```
(let ((v (vector 'a                    'e)))
  (vector-set! v 2
=> #(a b x d e)
(define vector-fill!
  (lambda (v x)
    (let ((n (vector-length v)))
      (do ((i 0 (+ i 1)))
          ((= i n))
        (vector-set! v i x)))))
=> vector-fill!
(let ((v (vector 1 2 3)))
  (vector-fill! v 0) v)
=> #(0 0 0)
```


Jochen Lang, EECS
jlang@uOttawa.ca

Converting a List into a Vector

- Scheme functi tor and list conversions

```
(list->vecto 3 4))
(vector->lis >vector '(1 2 3 4)))
```

- Our own functi

```
(define vec->li
  (lambda (s)
    (do ((i (- (vector-length s) 1) (- i 1))
         (ls '() (cons (vector-ref s i) ls)))
        ((< i 0) ls))))
=> vec-li
(vec->li '#(a b c))
=> (a b c)
(let ((v '#(1 2 3 4 5)))
  (apply * (vec->li v)))
=> 120
```

A Vector Function Example

- Looping from be end

```
(define vector-s
  (lambda (vec)
    (let ((size (vector-length vec)))
      (do ((position 0 (+ position 1))
           (total 0
            (+ total (vector-ref vec position))))
          ((= position size) total)))))
=> vector-sum
(vector-sum #( 2 5 7 3 4))
=> 21
```

Another Example

- This time counting from the end to the beginning

```
(define vector-sum
  (lambda (vec)
    (do ((remaining (vector-length vec)
                    (- remaining 1))
         (total 0 (+ total (vector-ref vec
                    (- remaining 1)))))
        ((zero? remaining) total))))
=> vector-sum
(vector-sum #( 2 5 7 3 4))
=> 21
```

Sorting Vectors and Lists

- Sorting functions [illegible] n Scheme
  - dialect depen[illegible]
  - Racket has `sort` (while MIT Scheme has `quick-sort` and `merge-sort` accepting a list or vector) with an order predicate (here less-than). `sort` only accepts lists
- The predicate *test* must have the general form

```
(and (test x y) (test y x))
=> #f
```

- Examples

```
(sort  '(3 4 2 1 2 5) <)
=>   (1 2 2 3 4 5)
(sort '(0.5 1.2 1.1) >)
=>  (1.2 1.1 .5)
```

List Functions Needed for Merge-Sorting

- Recursive algorit
  - Split list into
  - until only one element
  - merge lists on the way up maintaining the order
- Our Implementation will use helper function
  - `split` ; splitting a list into two
    - `sub-list` ; Defined with a helper routine that extracts a sub-list
  - `merge-list` ; merging two lists in order

Extracting a Sub-list from a List

- Extracting a ran list with an additional offset

```
(define (sub L start stop ctr)
   (cond
     ((null? L) L)
     ((< ctr start)
        (sub (cdr L) start stop (+ ctr 1)))
     ((> ctr stop) '() )
     (else (cons (car L)
        (sub (cdr L) start stop (+ ctr 1))))))
=> sub
(sub '(a b c d e f g h) 3 7 0)
=> (d e f g h)
```

Split a List into Two

- Split a list into t ub-list
  - two base cas list and lists of 1

```
(define (split L)
  (let ((len (length L)))
    (cond ((= len 0) (list L L) )
          ((= len 1) (list L '()))
          (else (list (sub L 1 (quotient len 2) 1)
                      (sub L (+ (quotient len 2) 1)
                           len 1)))))))
=> split
(split '(a b c d e f g h))
=> ((a b c d) (e f g h))
```

Merging Two Lists

- Merging in order[illegible] the input is sorted

```
(define (mergeli
  (cond ( (null? L) M)
        ((null? M) L)
        ((< (car L)(car M))
              (cons (car L)
                    (mergelists (cdr L) M)))
        (else (cons (car M)
                    (mergelists L (cdr M))))))
=> mergelists
(mergelists '(1 5 10) '(2 6 7))
=> (1 2 5 6 7 10)
```

Merge-Sort Main Routine

- Assemble the sub

```
(define (mergesor
  (cond ((null? L
        ((= 1 (length L)) L)
        ((= 2 (length L))
              (mergelists (list (car L))(cdr L)))
        (else (mergelists
                 (mergesort (car (split L)) )
                 (mergesort (car (cdr (split L))))))))
=> mergesort
(mergesort  '(3 4 2 1 8 6 10))
=> (1 2 3 4 6 8 10)
```

Quick Sort

```
(define (qsort L
  (if (or (null?          ength L) 1))
      L   ; no n          t
      (let loop          ) (right '()) ; for
                  (pivot (car L)) (rest (cdr L)))
        (if (null? rest)
            (append (qsort left)(list pivot)(qsort right))
            (if (<= (car rest) pivot)
                (loop (append left (list (car rest)))
                      right pivot (cdr rest))
                (loop left (append right (list (car rest)))
                      pivot (cdr rest)))))))
=> qsort
(qsort  '(7 4 2 1 8 6 10))
=> (1 2 4 6 7 8 10)
```


Jochen Lang, EECS
jlang@uOttawa.ca

Scheme: Functional Programming

- Input/Output in
- Looping with do
- Vectors
  - basic functions
  - looping forwards and backwards
- Sorting
  - Mergesort
  - Quicksort

Jochen Lang, EECS
jlang@uOttawa.ca