Programming Paradigms CSI2120 – Winter 2018

Jochen Lang
EECS, University of Ottawa
Canada

Université d'Ottawa | University of Ottawa

uOttawa

L'Université canadienne
Canada's university

uOttawa.ca

Scheme: Functional Programming

- **Input/Output in Scheme**
- **Setting varibles with set!**
- **Looping with do**
- **Sorting**

Input/Output

- **display – prints to the screen (REPL buffer)**
  - (display "hello world")
  - 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 numbers, 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
```

Example: Using the Input

- **A function that reads a number and calls the function sqrt and prints the output.**

```
(define (find-sqrt)
  (let ((n (ask-number)))
    (display "The sqrt of ")
    (display n)
    (display " is ")
    (display (sqrt n))
    (newline)))
=> find-sqrt
(find-sqrt)
Enter a number: type 5
prints The sqrt of 5 is 2.23606797749979
```

Reading a File into a List

- **File i/o works as expected**
  - File streams are called 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 opens a file and applies a procedure to every token read.**
  - 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 `filename` and 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 to pass 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)))))))))
```

Reminder: The function set!

- **The function set! allows us to assign a value to a variable**

  (set! a-num (+ 3 4))

  (set! a-num (+ 1 a-num))
- **Note that in Scheme functions which modify their arguments are given names that end with an exclamation mark !**

Example: Encapsulated set

```
(define light-switch (let ((lit #f)) (lambda ()
      (set! lit (not lit))
      (if lit 'on 'off))))
```

- The variable lit is only accessible within the define
- It is a kind of a static variable inside a function

Stack – Using Variables

```
(define a-stack '())

(define (empty?)
  (null? a-stack))

(define (push e)
  (begin
    (set! a-stack
      (cons e a-stack))
    a-stack ))
```

```
(define (pop)
  (if (empty?)
      ()
      (begin
      (set! a-stack
         (cdr a-stack))
      a-stack)))

(define (top)
  (if (empty?)
      ()
      (car a-stack)))
```

Using a Stack

```
(define a-stack ())
=> ()
(empty?)
=> #t
(push 5)
=> (5)
(top)
=> 5
(pop)
=> ()
```

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
```

Sorting Vectors and Lists

- **Sorting functions available in Scheme**
  - dialect dependent
  - 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 algorithm**
  - Split list into two
  - 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 range from a 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 two using sub-list**
  - two base cases, empty 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 assuming the input is sorted**

```
(define (mergelists L M)
  (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-routines**

```
(define (mergesort L)
  (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? L) (<= (length L) 1))
      L   ; no need to sort
      (let loop ((left '()) (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)
```

Scheme: Functional Programming

- **Input/Output in Scheme**
- **Setting variables with set!**
  - stack with a top-level stack
- **Looping with do**
- **Sorting**
  - Mergesort
  - Quicksort