CS320 Handout 11
Pairs and Lists
Three Descriptions of Pairs

**Input Expression**

\[(\text{cons } 1 \ 2)\]

**Box and Pointer**

![Box and Pointer Diagram]

**Output Representation** – using *dotted pair notation*

\[(1 . \ 2)\]
Three Descriptions of Lists

Input Expression

\[(\text{define } L (\text{list } 2 (+ 1 2) (\text{list } \#t \ #f)))\]

Recall that
\[(\text{list } \langle v_1 \rangle \langle v_2 \rangle \cdots \langle v_n \rangle)\]
is equivalent to
\[(\text{cons } \langle v_1 \rangle (\text{cons } \langle v_2 \rangle (\cdots (\text{cons } \langle v_n \rangle \text{nil}) \cdots)))\]

Box and Pointer

Output Representation

\[(2 \ 3 \ (#t \ #f))\]

Using dotted pair notation, you may write instead:
\[(2 . (3 . (#t . (#f . ())))))\]
But this is more verbose and less readable!
“Box and Pointer” More Informative Than “Output Representation”

<table>
<thead>
<tr>
<th>Input Expression</th>
<th>Output Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(list (list 1 2) (list 3 2))</td>
<td>((1 2) (3 2))</td>
</tr>
<tr>
<td>(let ((x (list 2))) (list (cons 1 x) (cons 3 x)))</td>
<td>((1 2) (3 2))</td>
</tr>
</tbody>
</table>

Box and Pointer

![Box and Pointer Diagram]

1 2 3 2
Several Hand Exercises

;; A Scheme function that prints list structures in the same way ;; that a Scheme interpreter does:

(define (print-list-structure1 x)
  (cond ((null? x) (display "()"))
        ((not (pair? x)) (display x))
        (else (display "(")
               (print-list-structure1 (car x))
               (display " . ")
               (print-list-structure1 (cdr x))
               (display ")")))))

;; But the preceding is not the standard way of printing out ;; list structures. The standard way is implemented by the ;; following Scheme function:

(define (print-list-structure2 x)
  (define (print-contents x)
    (display (car x))
    (cond ((null? (cdr x)) () )
          ((not (pair? (cdr x)))
           (display " . ")
           (display (cdr x)))
          (else
           (display " ")
           (print-contents (cdr x)))))
  (cond ((null? x) (display "()"))
        ((not (pair? x)) (display x))
        (else (display "(")
               (print-contents x)
               (display ")")))))
;; A Scheme expression that prints out (1 2 3) -- yes, very easy:
;; (list 1 2 3)

;; A Scheme expression that prints out (1 2 . 3):
;; Is it (list 1 (cons 2 3)) ? NO
;; Is it (cons 1 (cons 2 3)) ? YES

;; A Scheme expression that prints out (1 . 2 3):
;; Is it (cons 1 (list 2 3)) ? NO
;; Is it (display "(1 . 2 3)") ? YES
;; Is it (list 1 "." 2 3) ? NO
;; Is it (list 1 "." 2 3) ? YES