MUTABLE DATA AND VECTORS 9

COMPUTER SCIENCE 61AS

The Basics of Mutable Data

1. `set!` is a special form that takes in two arguments. What must be true about the first argument so that the expression does not error?

   Solution: The first argument to `set!` must be a symbol that is already defined in the current environment.

2. When should `set!` be used versus `set-car!`/`set-cdr!`?

   Solution: `set!` should be used when we want to fully change what a variable's value is. `set-car!` and `set-cdr!` should be used when we want to change part of a list structure.

3. Why would we ever want to use mutation versus just making a brand new structure with the intended values? Try to come up with a non-contrived example if possible.

   Solution: It all depends on the problem. Using mutation to modify lists can often be more efficient since we don't have to remake the entire structure. Also, thinking back to OOP, if multiple objects have this list as a state variable, mutating the list means we don't have to update all of the references. This is not only more convenient, but will often also lead to a lot less bugs in larger projects.

4. True or false: Drawing out box-and-pointer diagrams help most students significantly when dealing with mutable data problems.

   Solution: True. Drawing it out is so much easier than keeping it all in your head!

Practice with Mutable Data

1. Say what Scheme prints, and draw the corresponding box-and-pointer diagram. In the case of an error, write "Error" and don't draw the diagram.

   (a)
   ```
   (define a (list 1 2))
   (set! (cadr a) 3)
   a
   ```

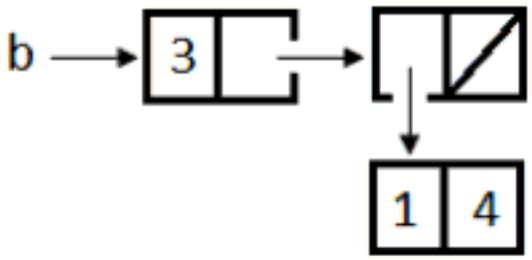
   _____

   Solution: Error. Remember, the first argument to `set!` must be a symbol.

   (b)
   ```
   (define b (list 3 6))
   (set-car! (cdr b) (cons 1 4))
   b
   ```

   _____

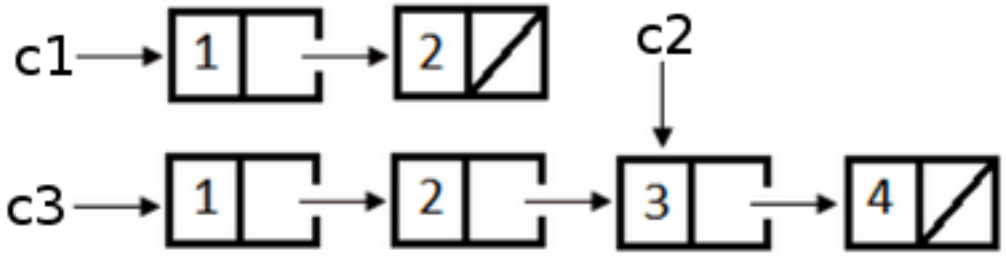
   Solution: `(3 (1 . 4))`

   (c)
   ```
   (define c1 (list 1 2))
   (define c2 (list 3 4))
   (define c3 (append c1 c2))
   c3
   ```

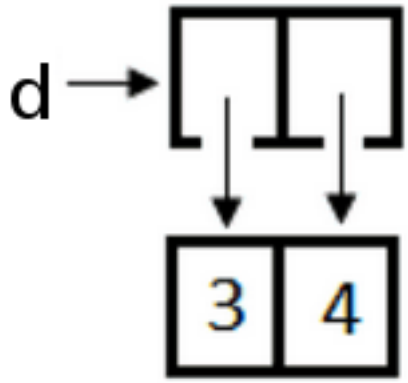
   _____

Solution: `(1 2 3 4)`



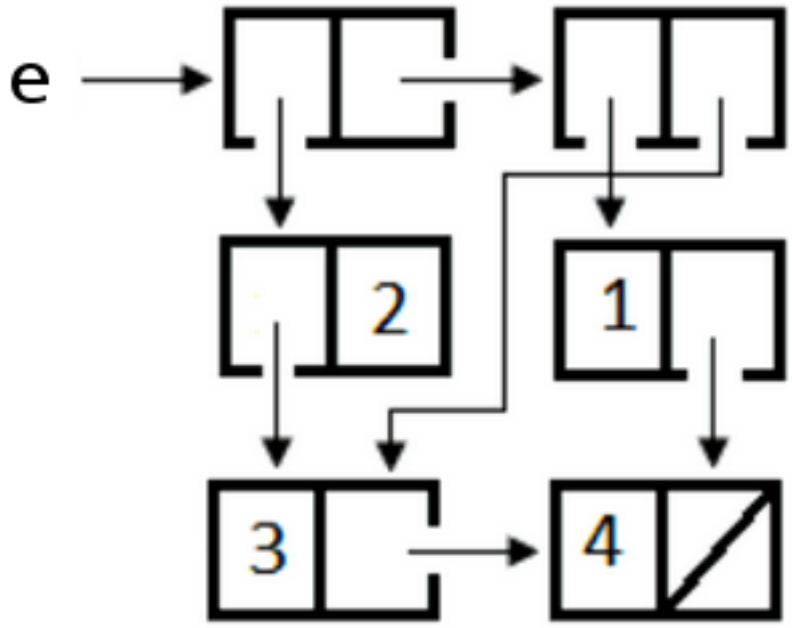
(d)
```
(define d (cons 1 2))
(set-car! d (cons 2 4))
(set-cdr! d (car d))
(set-car! (cdr d) 3)
d
```

Solution: `((3 . 4) 3 . 4)`



(e)
```
(define e (list (cons 5 2) 6))
(set-car! (car e) (list 3 4))
(set-car! (cdr e) (cons 1 (cdaar e)))
(set-cdr! (cdr e) (caar e))
e
```

Solution: `(((3 4) . 2) (1 4) 3 4)`



2. Write a procedure `remove-alternates!`, which removes every other element in a list, starting from the second one. You should not allocate any new pairs; just rearrange the existing pairs. (Hint: as with all mutation problems, try drawing out exactly how you want the box-and-pointer diagram to change!)

```
> (define x '(1 2 3 4 5 6))
> (remove-alternates! x)
  (1 3 5)
> x
  (1 3 5)
```

Solution:

```
(define (remove-alternates! lst)
    (if (not (or (null? lst) (null? cdr lst)))
        (set-cdr! lst (remove-alternates! (cddr lst))))
    lst)
```

3. Write a procedure `pairup!`, which pairs consecutive elements in a list. You should not

allocate any new pairs; just rearrange the existing pairs. Assume that the list is always of even length.

```
> (define x '(1 2 3 4 5 6))
> (pairup! x)
> x
  ((1 . 2) (3 . 4) (5 . 6))
```

Solution:

```
(define (pairup! lst)
    (if (not (null? lst))
        (let ((rest (cddr lst)))
            (set-cdr! (cdr lst) (cadr lst))
            (set-car! (cdr lst) (car lst))
            (set-car! lst (cdr lst))
            (set-cdr! lst rest)
            (pairup! rest))))
```

The Basics of Vectors

1. Describe at least 3 differences between vectors and lists.

   Solution: The size of a vector is fixed once it is made, while lists can always change size. Accessing any item in a vector takes constant time. This means, regardless of how big the vector is, the time to get to any one item doesn't increase! Lists, on the other hand, require on average linear time to access a random element. Vectors tend to be used in situations where the number of items is known in advance and random access to items is necessary. Lists, while slower to access items, are more useful when the number of items is not known in advance or random access is not needed.

2. A vector v has length x. What is the index of the first and last elements in v?

   Solution: 0 and $x - 1$.

3. For each of the following, indicate whether a vector or list would be better suited for the job (more efficient). If they are equally efficient or there is no clear winner, write "either". Explain your answers.

   (a) A roster of all of the students in CS 61AS, where students may add or drop the class at any time.

Solution: A list, since we do not know the number of students in the class ahead of time. However, you could argue that a vector would work as well since the number of students will never exceed a certain number.

(b) Stack of flashcards to remember the names of each of the first fifty US presidents.

Solution: A vector, since there is a clear mapping from index number to the name of the president. Additionally, we would want to quickly be able to find a president's name given his number (random access). Also, we have a bounded number of flashcards.

(c) A library of all of your music with the ablility to quickly play any one at will.

Solution: This problem doesn't have a clear winner. Random access to the songs would require a vector. However, vectors also require the length to be known in advance, which means there will be a limit to the number of songs. You'll learn more about trade-offs like these and different data structures that could help solve the problem in CS 61B!

Practice with Vectors

1. Write a procedure `square-vect!` which takes in a vector of numbers as argument and squares each one of its arguments. You should not create a new vector; simply change the values in the original argument.

Solution:

```
(define (square-vect! vec)
    (define (square x) (* x x))
    (define (loop i)
        (if (< i (vector-length vec))
            (let ((old-num (vector-ref vec i)))
                (vector-set! vec i (square old-num))
                (loop (+ i 1)))
            vec))
    (loop 0))
```

2. Write a procedure `duplicate` which takes in a vector and a number indicating how many times to duplicate each value.

```
> (duplicate (vector 1 2 3) 3)
  #(1 1 1 2 2 2 3 3 3)

> (duplicate (vector 4 5 6) 0)
  #()
```

Solution:

```
(define (duplicate vec n)
  (define (loop new-vec i)
    (if (< i (vector-length new-vec))
        (let ((num (vector-ref vec (div i n))))
          (vector-set! new-vec i num)
          (loop new-vec (+ i 1)))
        new-vec))
  (loop (make-vector (* n (vector-length vec))) 0))
```

`div` does an integer division: `(div 5 2)` will return 2.

3. You have to guess a number between 0 and 100, and with each guess, you will be told whether it was too high or two low. The best way to guess is then to guess 50, because then no matter what the answer is you have eliminated at least half the candidates. If you then find out it was too high, you would guess 25, to eliminate half of the remaining candidates, and so on.

   The binary search algorithm is based on a similiar idea. We are given a vector of numbers sorted in ascending order, and we want to find out if a certain number is in this vector. Instead of blindly looking through the entire list, we look at the middle element, and compare it to the number we are searching for. Depending on the result, we recurse on the lower or upper part of the vector (or if we actually find the element, we return `#t`).

   (a) What is the run-time of the binary search algorithm described above?

   Solution: $\Theta(log(n))$, where n is the length of the vector. Because each iteration of the binary search algorithm eliminates half of the elements, it will take at most $log(n)$ steps to either find the item or realize it is not there. Each iteration consists of finding the middle element and doing at most three comparisons. Thus, each iteration takes $\Theta(1)$ time. Thus, the entire algorithm takes $\Theta(1)$ time for each iteration, with $\Theta(log(n))$ iterations, leading to a runtime of $\Theta(log(n))$.

   (b) What would the run-time of this algorithm be if we tried using it on a list, rather than a

vector? Why?

> **Solution:** $\Theta(nlog(n))$. Using the same logic as above, we will still have at most $\Theta(log(n))$ iterations. However, each iteration will now take n time. This is because finding the middle element requires walking through $n/2$ items, which is in $\Theta(n)$. (It would have been quicker to just go through all of the elements one-by-one!)

(c) Write the procedure `bsearch`, which takes as input a sorted vector and an element, and using binary search determines whether the element is in the vector. It returns `#t` if the element is in the vector, and `#f` otherwise. You should not create new vectors, since that is inefficient. (Note: to get the middle index, you should do something like `(div (+ low high) 2)`). `div` does an integer division, so `(div 5 2)` will return 2.

```
> (define my-vec (vector 2 9 12 15 19 24 100))
> (bsearch my-vec 18)
  #f
> (bsearch my-vec -1)
  #f
> (bsearch my-vec 100)
  #t
```

Solution:

```
(define (bsearch vec item)
    (define (loop low high)
        (if (> low high)
            #f
            (let* ((mid (div (+ low high) 2)))
                   (elmnt (vector-ref vec mid)))
                (cond ((= item elmt) #t)
                      ((> item elmt)
                       (loop low (- mid 1)))
                      ((> item elmt)
                       (loop (+ mid 1) high))))))
    (loop 0 (- (vector-length vec) 1)))
```