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?

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

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.

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

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
   ```

   _____

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

   _____

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

   _____

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

   _____

(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
```

_____

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)
```

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))
```

The Basics of Vectors

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

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

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.

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

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

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.

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)
  #()
```

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?

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

(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
```