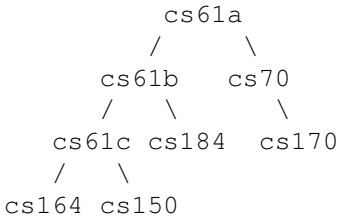
1 Trees (with a capital T)

At Depth University, a student must complete at least one advanced class to graduate. However, every advanced class has a prerequisite, which may itself have a prerequisite, and so on. Write a procedure `fast-grad` that, given a prerequisite tree, with constructor `make-tree` and selectors `datum` and `children`, returns the shortest possible list of courses needed to graduate. If there is a tie, `fast-grad` may return any of the shortest lists. You may assume that all leaf nodes are advanced classes, and vice versa.

For example, `fast-grad` called on the following tree can return `(cs61a cs70 cs170)` or `(cs61a cs61b cs184)`. Either one is correct.



SOLUTION:

```
(define (fast-grad tree)
  (cons (datum tree)
        (forest-grad (children tree))))

(define (forest-grad forest)
  (min-length (map fast-grad forest)))

(define (min-length lol)
  (cond ((null? lol) '())
        ((null? (cdr lol)) (car lol))
        (else
          (let ((rest (min-length (cdr lol))))
            (if (< (length (car lol)) (length rest))
                (car lol)
                rest)))))
```

2 Environment Diagram

Draw the full environment diagram that is generated from typing the following expressions. What does each line return?

```
STk> (define (bo-peep)
       (let ((sheep 1000))
         (define (herd msg)
           (cond ((eq? msg 'one) (set! sheep (/ sheep 2)) sheep)
                 ((eq? msg 'two)
                   (let ((m (herd 'one)))
                        (set! sheep (* m 4))
                        sheep))))
         herd))

STk> (define flock (bo-peep))

STk> (flock 'two)
```

_____

SOLUTION: 2000

Please use `envdraw` to check your environment diagram.

3 Concurrency

Assume the following has been typed in at the prompt:

```
STk> (define x 5)
STk> (define y 8)

STk> (define s (make-serializer))
STk> (define t (make-serializer))
```

a)
```
(parallel-execute (s (lambda () (set! x (+ x 1))))
                  (t (lambda () (set! x (+ x 2)))))
```

Is deadlock possible? _____

What are the possible results? _____

SOLUTION:

Is deadlock possible? NO
What are the possible results? `x = 6, 7, 8`

b)
```
(parallel-execute (s (t (lambda () (set! y (+ x 1)))))
                  (t (s (lambda () (set! x (* y 2))))))
```

Is deadlock possible? _____

What are the possible results? _____

SOLUTION:

Is deadlock possible? YES
What are the possible results? `x = 12, y = 6` and `x = 16, y = 17`

4 Streams

What are the first seven elements of the following stream?

```
(define mystery (cons-stream 1
                  (cons-stream 2
                    (stream-map (lambda (x y) (+ x (* 2 y)))
                                mystery
                                (stream-cdr mystery)))))
```

mystery: _____ _____ _____ _____ _____ _____ _____

SOLUTION:

mystery: 1 2 5 12 29 70 169

5 Metacircular Evaluator (Part 1)

Recall that `mceval.scm` tests true or false using the `true?` and `false?` procedure:

```
(define (true? x) (not (eq? x false)))
(define (false? x) (eq? x false))
```

Suppose we type the following definition into MCE:

```
mceval> (define true false)
```

What would be returned by the following expression?

```
mceval> (if (= 2 2) 3 4)
```

SOLUTION: 3

6 Metacircular Evaluator (Part 2)

Suppose we type the following into MCE:

```
mceval> (define 'x (* x x))
```

This expression evaluates without error! Remember that expressions such as `'x` are automatically expanded to be the list `(quote x)` prior to evaluation. Knowing this, what would be returned by the following expressions?

a) `mceval> quote`

SOLUTION: `(compound-procedure (x) ((* x x)) <procedure-env>)`

b) `mceval> (quote 10)`

SOLUTION: `10`

7 Analyzing Evaluator

Which of the following interactions will execute faster, slower, or the same in the analyzing evaluator than in the original metacircular evaluator?

Circle FASTER, SLOWER, or SAME for each.

a)
```
STk> (define (gauss-recur n) ;; sum of #s from 1 to n
       (if (= n 1)
           1
           (+ n (gauss-recur (- n 1)))))
STk> (gauss-recur 1000)
```

Analyzing will be...

FASTER SLOWER SAME

SOLUTION: FASTER

b)
```
STk> (define (gauss n) ;; sum of #s from 1 to n
       (/ (* (+ n 1) n) 2))
STk> (gauss 1000)
```

Analyzing will be...

FASTER SLOWER SAME

SOLUTION: SAME

8 Lazy Evaluator

Consider the following interactions in the lazy evaluator:

```
lazy> (define w 100)
lazy> (define (foo x y)
        (x y))
lazy> (define q
        (foo (lambda (z) (set! w 50) z)
             (begin (set! w 10) 3)))
```

What are the values of the following statements when typed at the prompt immediately afterwards?

1. `w`: _____
2. `q`: _____
3. `w`: _____

SOLUTION:

1. `w`: 50
2. `q`: 3
3. `w`: 10

9 Logic Programming

Write rules for the query (logic) evaluator to define the `subseq` relation. `subseq` is a relation between two lists. The query evaluator should return a solution if the elements in the first list are present in the second list in the same order. For example:

```
query> (subseq (a b c) (a z b y c))  ;; returns a solution
(subseq (a b c) (a z b y c))

query> (subseq (a b c) (a z c b)) ;; wrong order, so no solution
```

SOLUTION:

```
(assert! (rule (same ?x ?x)))

(assert! (rule (subseq () ?ls)))

(assert! (rule (subseq (?car1 . ?cdr1) (?car1 . ?cdr2))
               (subseq ?cdr1 ?cdr2)))

(assert! (rule (subseq (?car1 . ?cdr1) (?car2 . ?cdr2))
               (and (subseq (?car1 . ?cdr1) ?cdr2)
                    (not (same ?car1 ?car2)))))
```

10 Vectors

Write a procedure `vector-remove` that, given a vector and an element, returns a new vector with the occurrences of that element removed. The size of the new vector should be exactly the number of elements minus the ones removed.

You may find the `count` procedure helpful, which, given a vector and an element, returns the number of occurrences of that element in that vector. **Do not use** `vector->list` **or** `list->vector` **in this problem. Use no data aggregates other than vectors.**

```
STk> (count #(1 2 3 1) 1)
2
STk> (vector-remove #(1 2 3 1) 1)
#(2 3)
STk> (vector-remove #(1 2 3) 4)
#(1 2 3)
```

SOLUTION:

```
(define (vector-remove vect el)
  (define (helper old new old-i new-i)
    (if (= old-i (vector-length old))
        new
        (if (eq? (vector-ref old old-i) el)
            (helper old new (+ old-i 1) new-i)
            (begin (vector-set! new new-i (vector-ref old old-i))
                   (helper old new (+ old-i 1) (+ new-i 1))))))
  (helper vect (make-vector (- (vector-length vect) (count vect el))) 0 0))
```