

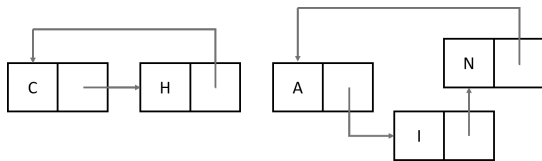
## CS 61A

August 11, 2021

### 1 Environment Diagrams

#### 1. 2 Chainz

2 Chainz accidentally scrambled his chains! Now there's just one long link that reads "CANHI." Fill in each blank in the code example below so that its environment diagram is the following.



```
a = Link("C", Link("A", Link("N", Link("H", Link("I")))))
b = _____
a.rest = b.rest.rest
a._____ = b.rest
b.rest = _____
a.rest.rest = _____
b.rest.rest.rest = _____
```

```
a = Link("C", Link("A", Link("N", Link("H", Link("I")))))
b = a.rest
a.rest = b.rest.rest
a.rest.rest.rest = b.rest
b.rest = a.rest
a.rest.rest = a
b.rest.rest.rest = b
```

---

## 2 OOP

---

1. The `DLList` class is a spin off of the normal `Link` class we learned in class; each `DLList` link has a `prev` attribute that keeps track of the previous link and a `next` attribute that keeps track of the next link. Fill in the following methods for `DLList`.

(a) `class DLList:`

```
    """
```

```
    >>> lst = DLList(6, DLList(1))
```

```
    >>> lst.value
```

```
    6
```

```
    >>> lst.next.value
```

```
    1
```

```
    >>> lst.prev.value
```

```
AttributeError: 'NoneType' object has no attribute 'value'
```

```
    """
```

```
    empty = None
```

```
    def __init__(self, value, next=empty, prev=empty):
```

```
        _____
```

```
        _____
```

```
        _____
```

```
    def __init__(self, value, next=empty, prev=empty):
```

```
        self.value = value
```

```
        self.next = next
```

```
        self.prev = prev
```

```
(b) def add_last(self, value):
    """
    >>> lst = DLList(6)
    >>> lst.add_last(1)
    >>> lst.value
    6
    >>> lst.next.value
    1
    >>> lst.next.prev.value
    6
    """
    pointer = self
    while _____:

        _____

    _____ = DLList(_____)

def add_last(self, value):
    pointer = self
    while pointer.next != DLList.empty:
        pointer = pointer.next
    pointer.next = DLList(value, DLList.empty, pointer)
```

(c) **def** add\_first(self, value):

"""

>>> lst = DLLList('A')

>>> lst.add\_first(1)

>>> lst.value

1

>>> lst.next.value

'A'

>>> lst.next.prev.value

1

>>> lst.add\_first(6)

>>> lst.value

6

>>> lst.next.next.prev.value

1

"""

old\_first = DLLList(\_\_\_\_\_)

\_\_\_\_\_ = \_\_\_\_\_

\_\_\_\_\_ = \_\_\_\_\_

**if** \_\_\_\_\_:

\_\_\_\_\_

**def** add\_first(self, value):

old\_first = DLLList(self.value, self.next, self)

self.value = value

self.next = old\_first

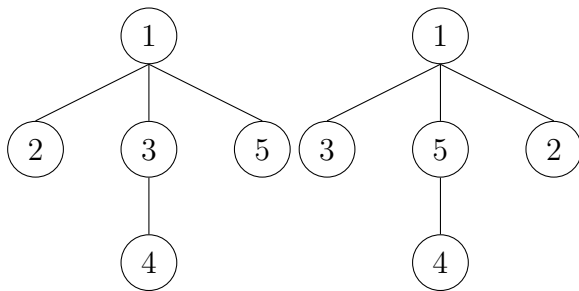
**if** old\_first.next != DLLList.empty:

old\_first.next.prev = old\_first

### 3 Trees

1. Implement `rotate`, which takes in a tree and rotates the labels at each level of the tree by one to the left destructively. This rotation should be modular (That is, the leftmost label at a level will become the rightmost label after running `rotate`). You do NOT need to rotate across different branches.

For example, given tree `t` on the left, `rotate(t)` should mutate `t` to give us the right.



```

def rotate(t):
    """
    >>> t1 = Tree(1, [Tree(2), Tree(3, [Tree(4)]), Tree(5)])
    >>> rotate(t1)
    >>> t1
    Tree(1, [Tree(3), Tree(5, [Tree(4)]), Tree(2)])
    >>> t2 = Tree(1, [Tree(2, [Tree(3), Tree(4)]),
                      Tree(5, [Tree(6)])])
    >>> rotate(t2)
    >>> t2
    Tree(1, [Tree(5, [Tree(4), Tree(3)]),
              Tree(2, [Tree(6)])])
    """
    branch_labels = _____

```

```

n = len(t.branches)

```

```

for _____:

```

```

    _____

```

```

    _____

```

```

    _____

```

```
def rotate(t):
    branch_labels = [b.label for b in t.branches]
    n = len(t.branches)
    for i in range(n):
        branch = t.branches[i]
        branch.label = branch_labels[(i + 1) % n]
        rotate(branch)
```

2. Define `tree_sequence`, a generator that iterates through a tree by first yielding the root value and then yielding the values from each branch.

```
def tree_sequence(t):
    """
    >>> t = tree(1, [tree(2, [tree(5)]), tree(3, [tree(4)])])
    >>> print(list(tree_sequence(t)))
    [1, 2, 5, 3, 4]
    """
```

```
def tree_sequence(t):
    yield label(t)
    for branch in branches(t):
        for value in tree_sequence(branch):
            yield value
```

Alternate solution:

```
def tree_sequence(t):
    yield label(t)
    for branch in branches(t):
        yield from tree_sequence(branch)
```

Thinking about the solution in terms of the recursive leap of faith: assume that each call to `tree_sequence(branch)` yields the values in that branch in the proper order. Then all we have to do is yield each value from that branch for each branch in order after yielding the root value.

We utilize the common strategy of performing an action on the current node's value, and then using tree recursion to repeat this action for each branch of the list of branches.

In the alternate solution, `yield from` allows us to yield a list of values, aka the list of all results from recursively calling `tree_sequence`. This is equivalent to yielding each element through a for loop.

---

## 4 Tree Recursion

---

1. Define `all_sums`, a generator that iterates through all the sums that can be formed by adding the elements in `lst`.

```
def all_sums(lst):  
    """  
    >>> list(all_sums([]))  
    [0] #sum nothing  
    >>> list(all_sums([1, 2]))  
    [3, 2, 1, 0] #1 + 2, 2, 1, 0  
    >>> list(all_sums([1, 2, 3]))  
    [6, 5, 4, 3, 3, 2, 1, 0] #repeat sums are ok! (3 and 2+1)  
    """  
  
    if len(lst) == 0:  
        yield 0  
    else:  
        for sum_rest in all_sums(lst[1:]):  
            yield sum_rest + lst[0]  
            yield sum_rest
```

2. Fill in `combine_to_61`, which takes in a list of positive integers and returns `True` if a contiguous sublist (i.e. a sublist of adjacent elements) combine to 61. You can **combine** two adjacent elements by either summing them or multiplying them together. If there is no combination of summing and multiplying that equals 61, return `False`.

```
def combine_to_61(lst):
    """
    >>> combine_to_61([3, 4, 5])
    False # no combination will produce 61
    >>> combine_to_61([2, 6, 10, 1, 3])
    True # 61 = 6 * 10 + 1
    >>> combine_to_61([2, 6, 3, 10, 1])
    False # elements must be contiguous
    """

    def helper(lst, num_so_far):

        if _____:
            return True

        elif _____:
            return False

        with_sum = _____ and \
            helper(_____, _____)

        with_mul = _____ and \
            helper(_____, _____)

        return with_sum or with_mul

    return _____
```



```
def combine_to_61(lst):
    def helper(lst, num_so_far):
        if num_so_far == 61:
            return True
        elif not lst:
            return False
        with_sum = num_so_far + lst[0] <= 61 and helper(lst
            [1:], num_so_far + lst[0])
        with_mul = num_so_far * lst[0] <= 61 and helper(lst
            [1:], num_so_far * lst[0])
        return with_sum or with_mul
    return helper(lst, 0)
```

---

## 5 Linked Lists

---

1. Complete the implementation of `iter_link`, which takes in a linked list and returns a generator which will iterate over the values of the linked list in order. Your function should support deep linked lists.

```
def iter_link(lnk):
    """
    Yield the values of a linked list in order; your function
    should support deep linked lists.
    >>> lst1 = Link(1, Link(2, Link(3, Link(4))))
    >>> list(iter_link(lst1))
    [1, 2, 3, 4]
    >>> lst2 = Link(1, Link(Link(2, Link(3)), Link(4, Link(5))))
    >>> print(lst2)
    <1 <2 3> 4 5>
    >>> iter_lst2 = iter_link(lst2)
    >>> next(iter_lst2)
    1
    >>> next(iter_lst2)
    2
    >>> next(iter_lst2)
    3
    >>> next(iter_lst2)
    4
    """
    if lnk is not Link.empty:
        if type(_____) is Link:
            _____
        else:
            _____
            _____
```

```
def iter_link(lnk):
    if lnk is not Link.empty:
        if type(lnk.first) is Link:
            yield from iter_link(lnk.first)
        else:
            yield lnk.first
    yield from iter_link(lnk.rest)
```

2. Write a function `combine_two`, which takes in a linked list of integers `lnk` and a two-argument function `fn`. It returns a new linked list where every two elements of `lnk` have been combined using `fn`.

```
def combine_two(lnk, fn):
    """
    >>> lnk1 = Link(1, Link(2, Link(3, Link(4))))
    >>> combine_two(lnk1, add)
    Link(3, Link(7))
    >>> lnk2 = Link(2, Link(4, Link(6)))
    >>> combine_two(lnk2, mul)
    Link(8, Link(6))
    """
    if _____:

        return _____

    elif _____:

        return _____

    combined = _____

    return _____
```

```
def combine_two(lnk, fn):
    if lnk is Link.empty:
        return Link.empty
    elif lnk.rest is Link.empty:
        return Link(lnk.first)
    combined = fn(lnk.first, lnk.rest.first)
    return Link(combined, combine_two(lnk.rest.rest, fn))
```

## 6 Higher-Order Functions

1. Write a function, `make_digit_remover`, which takes in an integer from 0-9, `i`. It returns another function which takes in an integer, and removes all digits from right to left up to and including the first occurrence of `i`. If `i` does not occur in the integer, this function returns the original number.

```
def make_digit_remover(i):
    """
    >>> remove_two = make_digit_remover(2)
    >>> remove_two(232018)
    23
    >>> remove_two(23)
    0
    >>> remove_two(99)
    99
    """
    def remove(_____):

        removed = _____

        while _____ > 0:

            _____

            removed = removed // 10

            if _____:

                _____

        return _____

    return _____

def make_digit_remover(i):
    def remove(n):
        removed = n
        while removed > 0:
            digit = removed % 10
            removed = removed // 10
            if digit == i:
                return removed
        return n
    return remove
```

2. Write a function, `curry_forever`, which takes in a two-argument function, `f`, and an integer, `arg_num`. It returns another function that allows us to enter `arg_num` amount of numbers into `f` one by one.

```
def curry_forever (f, arg_num, base=0):
    """
    >>> g = curry_forever(add, 4)
    >>> g(1)(2)(3)(4) # 1 + 2 + 3 + 4
    10
    """
```

```
def helper(arg_num, amt):
```

```
    if arg_num == 0:
```

```
        return
```

```
def curry_forever (f, arg_num, base=0):
    def helper(arg_num, amt)
        if arg_num == 0:
            return amt
        return lambda x: helper(arg_num - 1, f(amt, x))
    return helper(arg_num, base)
```

---

## 7 Scheme

---

1. You are creating a computer from scratch. In their rawest form, computers use 0s and 1s to compose commands and data. Fill in a function that takes a list of boolean values representing an **unsigned binary number** and returns its **decimal representation**. Each `#t` in the list represents a 1 and each `#f` represents a 0, with the **first** element in the list being the **rightmost** (smallest) binary digit and the **last** element being the **leftmost** (largest) binary digit.

```
;Doctests
scm> (binary (list #f #t)) ; 10
2
scm> (binary (list #t #f #t #t)) ; 1101
13
scm> (binary (list #t #t #f #f #t)) ; 10011
19
scm> (binary (list #f)) ; 0
0
```

```
(define (binary bin-list)
  (cond
    ((null? _____)
     _____
    )
    ((_____ )
     _____
    )
    (else
     _____
    )
  )
)
```

```
(define (binary bin-list)
  (cond
    ((null? bin-list)
     0
    )
    ((car bin-list)
     (+ 1 (* 2 (binary (cdr bin-list))))
    )
    (else
     (* 2 (binary (cdr bin-list)))
    )
  )
)
```

2. Now, write the binary to decimal function, but in tail recursive form. Note that the `expt` function takes in a base and an exponent. For example, `(expt 2 3)` raises 2 to the third power, returning 8.

```
;Doctests
scm> (binary-tail (list #f #t)) ; 10
2
scm> (binary-tail (list #t #f #t #t)) ; 1101
13
scm> (binary-tail (list #t #t #f #f #t)) ; 10011
19
scm> (binary-tail (list #f)) ; 0
0
```

```
(define (binary-tail bin-list)
  (define (helper bin-list i sum)
    (cond
      ((null? _____)
       _____
      )
      ((_____ )
       _____
      )
      (else
       _____
      )
    )
  )
  (helper _____)
)
```



```
(define (binary-tail bin-list)
  (define (helper bin-list i sum)
    (cond
      ((null? bin-list)
       sum)
      ((car bin-list)
       (helper
        (cdr bin-list) (+ 1 i) (+ sum (expt 2 i)))
      )
      (else
       (helper
        (cdr bin-list) (+ 1 i) sum)
      )
    )
  )
  (helper bin-list 0 0)
)
```

3. Given the function `run`, write the helper function `duplicate` that takes in a list of integers, `lst`, and an integer `n`. The `duplicate` function takes each element of the list and duplicates it by its value (i.e. If the first number in the list is 2, add 2 as the next element in the list so we have a total of two 2's in the list).

```
;Doctests
scm> (define lst (cons 1 (cons 3 (cons 2 nil))))
lst
scm> (run lst)
(1 3 3 3 2 2)
scm> (run (cons 2 (cons 2 (cons 1 nil))))
(2 2 2 2 1)

(define run
  (lambda (lst)
    (duplicate lst 0)
  )
)

(define duplicate
```

)

```

(define duplicate
  (lambda (lst n)
    (cond ((null? lst) '())
          ((< n (car lst))
           (cons (car lst) (duplicate lst
                                       (+ n 1))))
          (else
           (duplicate (cdr lst) 0))
          )
    )
  )
)

```

## 8 Tail Recursion

1. Implement `slice`, which takes in a list `lst`, a starting index `i`, and an ending index `j`, and returns a new list containing the elements of `lst` from index `i` to `j - 1`.

;Doctests

```

scm> (slice '(0 1 2 3 4) 1 3)
(1 2)
scm> (slice '(0 1 2 3 4) 3 5)
(3 4)
scm> (slice '(0 1 2 3 4) 3 1)
()

```

```

(define (slice lst i j)

```

```

)

```

```

(define (slice lst i j)
  (cond ((or (null? lst) (>= i j)) nil)
        ((= i 0) (cons (car lst) (slice (cdr lst) i (-
                                           j 1))))
        (else (slice (cdr lst) (- i 1) (- j 1)))))

```

2. Now implement `slice` with the same specifications, but make your implementation tail recursive.

You may wish to use the built-in `append` function, which takes in two lists and returns a new list containing the elements of the two lists concatenated together.

```
(define (slice lst i j)
```

```
)
```

```
(define (slice lst i j)
  (define (slice-tail lst i j lst-so-far)
    (cond ((or (null? lst) (>= i j)) lst-so-far)
          ((= i 0) (slice-tail (cdr lst) i (- j 1) (
            append lst-so-far (list (car lst)))))
          (else (slice-tail (cdr lst) (- i 1) (- j 1) lst
            -so-far))))
  (slice-tail lst i j nil))
```

Alternate Solution:

```
(define (slice lst i j)
  (define (slice-tail lst index lst-so-far)
    (cond ((or (null? lst) (= index j)) lst-so-far)
          ((<= i index) (slice-tail (cdr lst) (+ index 1)
            (append lst-so-far (list (car lst)))))
          (else (slice-tail (cdr lst) (+ index 1) lst-so-
            far))))
  (if (< i j) (slice-tail lst 0 nil) nil))
```