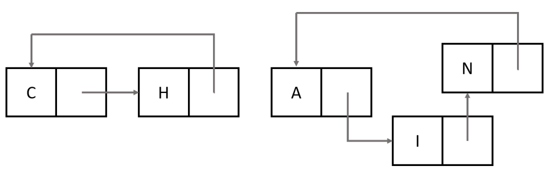
Final Review

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 = _______________________________
```

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):

        ________________________________________

        ________________________________________

        ________________________________________
```

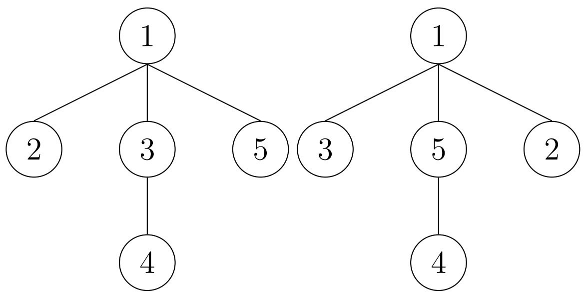
(b)
```
def add_last(self, value):
    """
    >>> lst = DLList(6)
    >>> lst.add_last(1)
    >>> lst.value
    6
    >>> lst.next.value
    1
    >>> lst.next.prev.value
    6
    """
    pointer = self
    while ______________________________:

        ____________________________________

    ______________ = DLList(___________________________)
```

(c)
```
def add_first(self, value):
    """
    >>> lst = DLList('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 = DLList(______________________________)

    ________________ = _________________________________

    ________________ = _________________________________

    if ________________________________:

        __________________________________________________
```

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 ______________________________________:

        ______________________________________________

        ______________________________________________

        ______________________________________________
```

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]
    """
```

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)
    """
```

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 ______________________________
```

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:

            ______________________________________

        ________________________________________
```

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 __________________________________
```

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 occurance 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 ___________________________
```

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 __________________________________________

    ____________________________________________________
```

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
      ____________________________________
    )
  )
)
```

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 ________________)
)
```

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








)
```

8 Tail Recursion

1. Implement `slice`, which takes in a 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)

)
```

2. Now implement slice with the same specifications, but make you 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)

)
```