Q9: Make Tree

```
(define (make-tree label branches)
    (cons label branches)
)
```

Python

```
def make_tree(label, branches=[]):
    return [label] + branches
```

Python but it's LL:

```
def make_tree(label, branches=Link.empty):
    return Link(label, branches)
```

```
(define (label tree)
    (car tree)
)
```

```
def label(tree):
    return tree[0]
```

```
def label(tree):
    return tree.first
```

```
(define (branches tree)
    (cdr tree)
)
```

```
def branches(tree):
    return tree[1:]
```

```
def label(tree):
    return tree.rest
```

- note that when we defined tree ADT's in Python, we used regular lists, BUT we could just as easily have used LL → we wouldn't even need to change any code that uses these constructor/selector functions, because we trust the abstraction barrier to take care of things (this is why branches(t) is better than t[1:] → what if our ADT was using LL?)
- also note that "default arguments" don't exist in Scheme → this is the same reason we can't call (cons 2) and have to use (cons 2 nil)
- think about: how to check if is_leaf? (see end of notes)

Q10: Tree sum

```
(define (tree-sum tree)
    (if (null? (branches tree))
        (label tree)
        (+ (label tree) (sum(map tree-sum (branches t))))
    )
)
```

*sum, sum on next page

Sln on wksht:

```
(define (tree-sum tree)
    (+ (label tree)
       (sum (map tree-sum (branches tree))))
    ))
```

Python ver 1
Python ver 2

```
def tree-sum (tree):
    if is_leaf (tree):
        return label(tree)
    else:
        return label(t) + sum ( map(tree-sum, branches(tree)))
def tree-sum (tree):
    return (label(tree) + sum(map(tree-sum, branches(tree))))
```

These Python implementation are interesting because if map & sum are the in-built Python versions, it'll work w/ the Python list ADT tree implementation but if map & sum are the LL versions we've defined, they'll work for the LL version ADT
→ hooray for abstraction!

```
(define (sum lst)
    (if (null? lst)
        0
        (+ (car lst)
           (sum (cdr lst)))
    )
)
```

(Python has a built-in sum for regular lists)

```
def sum(lnk):
    if lnk == Link.empty:
        return 0
    return lnk.first + sum_lnk(lnk.rest)
```

note we can skip writing "else" because we only reach this line → if lnk != Link.empty

→ this makes it even more Scheme-like!

```
def is_leaf(t):
    return len(branches(t)) == 0
```

(we're assuming t isn't null, which is generally dangerous but ok in our sandbox))

```
def is_leaf(t):
    return branches(t) == Link.empty
```

```
def is_leaf(t):
    return not branches(t)
```

↳ this works for either Python implementation because Link.empty and [] are both falsy! Abstraction's great ..

```
(define (is-leaf tree)
        (null? (branches tree))
)
```

- this is a little different from the LL implementation because the Link.empty equivalent is just nil, which has a built-in procedure for checking. A more direct translation would be:

```
(define (is-leaf tree)
        (eq? (branches tree) nil)
)
```