!

Plagiarism.

- All work in CS 114 is to be done individually. The penalty for plagiarism on assignments (first offense) is a mark of 0 on the assignment and a 5% reduction of the final grade, consistent with School of Computer Science policy. In addition, a letter detailing the offense is sent to the Associate Dean of Undergraduate Studies, meaning that subsequent offenses will carry more severe penalties, up to **suspension** or **expulsion**.
- To avoid inadvertently incurring this penalty, you should discuss assignment issues with other students only in a very broad and high-level fashion. Do not take notes during such discussions, and avoid looking at anyone else's code, on screen or on paper. If you find yourself stuck, contact the ISA or instructor for help, instead of getting the solution from someone else. Do not consult other books, library materials, Internet sources, or solutions (yours or other students') from other courses or other terms.

Assignment Guidelines.

- This assignment covers material in Module 9.
- Submission details:
  - Solutions to these questions must be placed in files `a09q1.py`, `a09q2.py`, and `a09q3.py`.
  - All solutions must be submitted to MarkUs. No solutions will be accepted through email, even if you are having issues with MarkUs.
  - Verify using MarkUs and your basic test results that your files were properly submitted and are readable on MarkUs.
  - Be sure to review the Academic Integrity policy on the Assignments page.
- Use a tolerance of 0.0001 for all tests that use `check.within`.
- Restrictions:
  - Unless the question specifically describes exceptions, you are restricted to using the functions and techniques covered in or before Module 9.
  - Read each question carefully for additional restrictions.
- **The solutions you submit must be entirely your own work. Do not look up either full or partial solutions on the Internet or in printed sources.**

Write the docstring and annotations carefully, for every function you write.
We will not mark code that is not documented and annotated.

!

Run this in Jupyter to download a file containing starter code:

```
!wget https://student.cs.uwaterloo.ca/~cs114/src/Assignment-09.ipynb
```

1. Most Children.

Exercise

Write a function `most_children(fam)`. It takes a `Family` and returns a `Tuple[int, str]` indicating the number of children and name of the person from `fam` who has the most children.
For example, in the `tully` tree that we've looked at before, the person with the most children is `'Catelyn'`, who has 5 children. So:
`most_children(tully) ⇒ (5, 'Catelyn')`
In the case of a tie:

- When it is a tie between two "descendants", the leftmost wins.
  So in `example1`, *Cy* wins over *Io*.
  `most_children(example1) ⇒ (3, 'Cy')`
- When it is a tie between the "parent" and one of the descendants, the "parent" wins.
  So in `example2`, *Nat* wins over *Ollie*.
  `most_children(example2) ⇒ (4, 'Nat')`

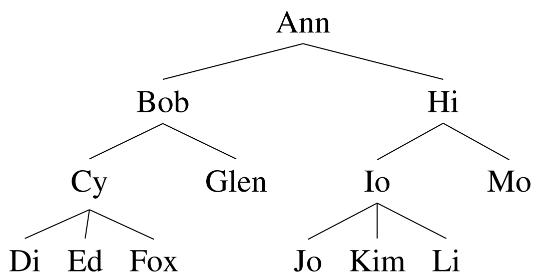

Diagram of `example1`

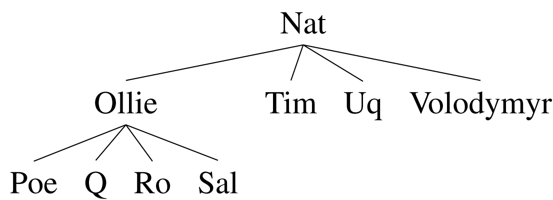

Diagram of `example2`

2. Changing a `LLT`. Recall the way we defined a leaf-labelled tree, `LLT`:

```
LLT = Union[int, List['LLT']]
```

Exercise

Write a function `tree_map(t: LLT, f: Callable)`. It returns a new `LLT` that has the same shape as `t`, but where each leaf has been transformed by the function `f`.

For example, we define a few simple `Callable` values:

```
def add1(x: int) -> int: return x + 1
def sqr(x: int) -> int: return x*x
def double(n: int) -> int: return n * 2
```

We can now make some tests:

```
check.expect("TM1",
             tree_map([2, [[4], 6], 0, [1]], add1), [3, [[5], 7], 1, [2]])
check.expect("TM2",
             tree_map([2, [[4], 6], 0, [1]], double), [4, [[8], 12], 0, [2]])
check.expect("TM3", tree_map(7, sqr), 49)
check.expect("TM4", tree_map([[[[[[[7]]]]]]], add1), [[[[[[[8]]]]]]])
```

3. No loops.

Start with the following function:

```
def zero(f: Callable, x0: float, x1: float) -> float:
    """Return a value between x0 and x1 such that the function f(x)
    returns a value that is close to 0 within 0.0001

    Requires:
      f is continuous on [x0, x1)
      f(x0) and f(x1) have opposite signs
      x0 <= x1
    """
    while abs(f(x0)) > 0.0001:
        mid_point = (x0 + x1) / 2

        if f(mid_point) * f(x0) < 0:
            x1 = mid_point
        else:
            x0 = mid_point

    return x0
```

Exercise: Rewrite the function `zero` without using any loops. Use recursion only.