Lab 9 [Regular Sections]

Goals:

- To practice writing data definitions for trees, writing functions that consumes trees, and writing functions that produce trees.
- To practice writing data definitions and functions for mutually recursive structures

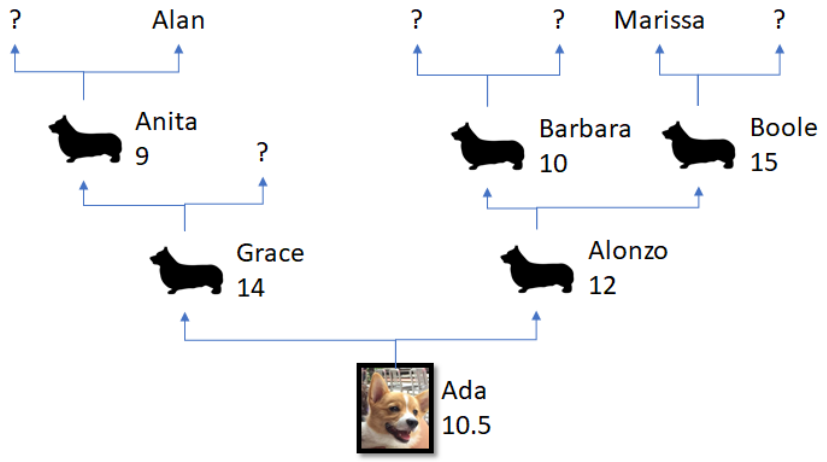
Simple Binary Trees

A Corgi is a high-class breed of dog known for its short legs, high enthusiasm, and adorable face. Some people care very strongly that they know the genetics of their Corgi, even though all dogs are good dogs. Regardless of whether you care about such things, you have been hired by a wealthy English Corgi enthusiast to write a Corgi Tracking System.

A **Corgi** has a *name* (a string), a *height* (a number), a *mom*, and a *dad*. Both the mom and dad of a Corgi are Corgi Family Trees.

A **Corgi Family Tree** is either a Corgi, the value `#false` (which indicates that we don't know anything about their parent), or a string indicating the name of the Corgi (which indicates that we don't know their height or parentage, just their name).

Problem 1A. Create a data definition for a **Corgi Family Tree** and a **Corgi**. Note that this includes the template and several examples. Model the following Corgi Family Tree as one of your examples. Note the question marks indicate a Corgi without any information (the value `#false`).



Problem 1B. Write a function `count-corgis` that consumes a Corgi Family Tree and produces the number of dogs in the tree. If a Corgi's parent is "?", then they are not counted towards the total (but do count parents where we only know their name).

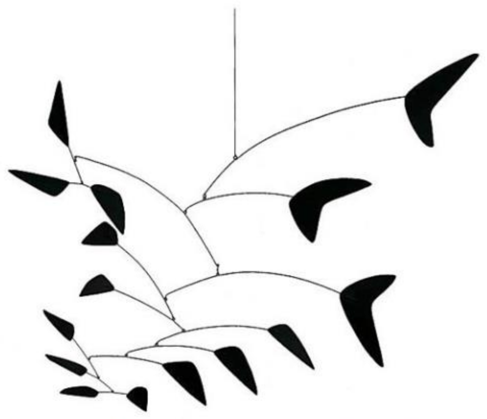
Problem 1C. Write a function `count-moms` that consumes a Corgi Family Tree and produces the number of Corgi moms in the tree. Again, do not count question marks, but do count dogs with just names. In the tree above, there are 4 corgi moms.

Problem 1D. Write a function `has-corgi?` that consumes a Corgi Family Tree and a name, then produces whether or not the given Corgi is in the tree.

Problem 1E. Write a function `find-corgis-by-height` that consumes a Corgi Family Tree and a threshold height, then produces a list of all the Corgi's names that are BELOW the given height. The order of the names is up to you. Hint: You may find it helpful to use the built-in `append` function, which joins two lists together (as opposed to `cons`, which adds an element to a list).

Calder's Binary Mobiles

We've created a data structure that refers to itself twice. But now we'll create a data structure that has multiple parts, some of which refer to each other. This will be a Binary Mobile, like the image below.

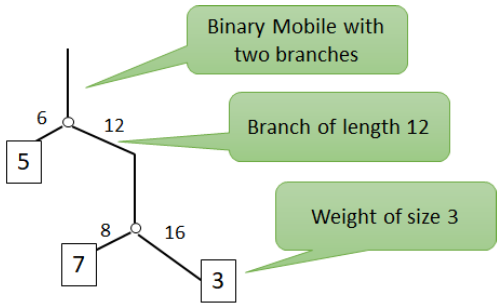

Alexander Calder's binary mobile, "Three Black Fishtails," 1960, sheet metal, wire and paint.

A **binary mobile** consists of two **branches**, a *left* branch and a *right* branch. Each **branch** is a rod of a certain *length*, from which hangs an *attachment*, either a **weight** or another **binary mobile**.

A **branch** is constructed from a rod *length* (which must be a number) together with one *attachment* hanging from the end of the rod.

An **attachment** may be either a `Number` (representing a simple **weight**, which is a number) or another **binary mobile**.

Here is a more abstract representation of this idea.



Problem 2A. Write the *data definition* [including an example and template] for a **binary mobile**. Note that there are ***three*** data definitions involved here (`BinaryMobile`, `Branch`, `Attachment`). Thus, there need to be at least *three* kinds of examples, and *three* templates.

Problem 2B. Define a procedure **total-weight** that returns the total weight of a binary mobile. For this problem, assume the rod weight is negligible [If you are a real sculptor, rod weight would be proportional to the length, of course]. Recall that any function on a `BinaryMobile` involves three functions (on a `BinaryMobile`, a `Branch`, and an `Attachment`, since they are mutually referring).

Problem 2C. Design a predicate **balanced?** that tests whether a binary mobile is balanced. A binary mobile is said to be **balanced?** if the *torque* applied by its left branch is equal to the torque applied by its right branch **and** if each of the attachments hanging off its branches are **balanced?**.

Note: *torque = length x weight.* That is, *torque* is the *length* of a rod multiplied by the *weight* of the attachment hanging from that rod. Torque is only well-defined on a `Branch`.