# Tree Zippers

**(3 points.)** To start, run `python new-homework.py [language] zippers`

**Background.** A **zipper** is a representation of an aggregate data structure that specifies a current location within the structure. Zippers allow navigation through the data structure and support operations to update its contents. An example of a zipper is our representation of a file system: at any time, we have a current working directory, but we can move through the directory structure and update the files and directories as desired. Zippers are particularly well suited to making immutable updates to such structures. In this exercise, you will implement a zipper for general trees.

In our language for specifying data types, we can represent a Zipper with two data type declarations:

```
data Maybe a = None | Some a
data Zipper a = Zipper { node :: a
                       , lefts :: List a
                       , rights :: List a
                       , path :: Maybe (Zipper a)
                       , changed :: Bool }
```

Let's unpack this. First, recall that the `a` in these expressions a "type variable" that stands in for another type, like `Int` or `Double` or `List String`.

Second, the type `Maybe a` represents a data value that is either missing, denoted `None`, or a value of type `a`, labeled `Some a`.

Third, a `Zipper a` represents a zipper for a tree whose nodes are of type `a`. It is a record with four fields:

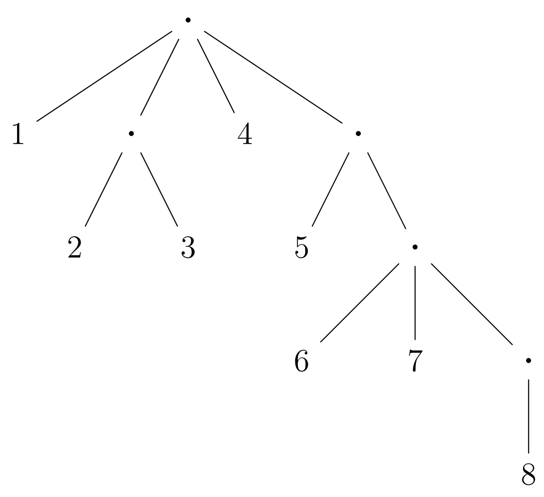
1. `node` - the current node under focus,
2. `lefts` - the list of left siblings of the current node,
3. `rights` - the list of right siblings of the current node,
4. `path` - the *path* from the current node to the root; if the current node is the root node, this is `Nothing`; otherwise it is a Zipper focused on the parent of the current node, and
5. `changed` - an indicator of whether the nodes within this subtree have been changed, and thus in aggregate differ from the values stored in the parent node.

The idea here is that nodes represent the *subtree* at that position in the tree.

Let's look at a simple example. We can think of the nested list

```
[1, [2, 3], 4, [5, [6, 7, [8]]]]
```

as a tree that looks like



We initialize our zipper with this tree

```
loc = vector_zipper([1, [2, 3], 4, [5, [6, 7, [8]]]])

#=> {node: [1, [2, 3], 4, [5, [6, 7, [8]]]],
#    lefts: [], rights: [],
#    path: None, changed: False}
```

The

`vector_zipper`

function is a zipper creator that is configured for this kind of nested list. As we will see below, it is easy to configure zippers for any tree-like structure. (The name `loc` stands for "location," reflecting the idea that the zipper is a pointer to a position in the tree.)

Now, we pass `loc` through a chain of functions[1] from the zipper library to produce a new zipper

```
pipe( loc, [down, right, down] )
#=> {node: 2,
#    lefts: [], rights: [3],
#    path: {node: [2,3], lefts: [1], rights: [4, [5, [6, 7, [8]]]]],
#           path: {...}, changed: False},
#    changed: False }
```

When we change the node

```
pipe( loc, [down, right, down, [replace, 10]] )
#=> {node: 10,
#    lefts: [], rights: [3],
#    path: {node: [2,3], lefts: [1], rights: [4, C], path: ..., changed: False},
#    changed: True }
```

[1]A python definition of the `pipe` function is given in the Resources section.

the change is recorded at the current focus, but the parent node will only be changed when we move up to view it, either directly or indirectly:

```
pipe( loc, [down, right, down, [replace, 10], up] )
#=> {node: [10, 3],
#    lefts: [1], rights: [4, [5, [6, 7, [8]]]],
#    path: {node: [1, [2, 3], 4, [5, [6, 7, [8]]]],
#           lefts: [], rights: [],
#           path: None, changed: False},
#    changed: True }
pipe( loc, [down, right, down, [replace, 10], root] )
#=> [1, [10, 3], 4, [5, [6, 7, [8]]]]

# A few other examples
pipe( loc, [down, [right, 3], down, right, [replace, 100], root] )
#=> [1, [10, 3], 4, [5, 100]]
pipe( loc, [down, [right, 2], [edit, lambda x: 2*x], right, down,
            right, [replace, 99], root] )
#=> [1, [10, 3], 8, [5, 99]]
pipe( loc, [down, [insert_right, 17], [right, 2], down,
            [insert_left, 1], root] )
#=> [1, 17, [1, 2, 3], 4, [5, [6, 7, [8]]]]
```

Notice that parts of the tree are replicated in the path. This works well with structure sharing. R's copy-on-write discipline is effective here – nodes are shared until one is changes, exactly as we would like. With Python and other languages, we can use persistent data structures for the same effect.

**Task.** Create a tree zipper library that creates zippers for a configured tree-like data structure (e.g., trees, file systems, xml files, pathnames and urls). Use a language of your choice, both R and Python, for instance, are appropriate. If you use Python, I suggest you use the `pyrsistent` package to manage the vectors and dicts in your data structure; `NamedTuple` may also be useful.

The basic interface to implement is as follows:

- `zipper(root, is_branch, children, make_node)`

  Returns a new zipper pointing to root node *root*. The remaining arguments are functions that determine how a zipper operates on the data. For a

  ```
  Zipper a
  ```

  object:

  - `is_branch :: a -> Bool`

    Returns true if the given node is a branch node (i.e., can have children), and false for leaf nodes.

  - `children :: a -> List a`

    Returns a list of a node's children. This will only be applied to branch nodes, but it can be defined to given an empty list for leaf nodes if desired.

  - `make_node :: a -> List a -> a`

    `make_node(node, new_children)`

    returns a new copy of *`node`* with the specified children *`new_children`*. (Any data in the node is preserved, but the children are replaced.)

  These three functions should be associated with the zipper object so that zipper methods can use them given only the zipper itself. (See e.g., *`up`*.)

- `node :: Zipper a -> a`

  *`node(loc)`* returns the node at *`loc`*'s current focus.

- `root :: Zipper a -> a`

  *`root(loc)`* returns the root node of *`loc`* and thus gives the underlying tree

- `up :: Zipper a -> Maybe (Zipper a)`

  *`up(loc)`* returns the updated zipper pointing to the parent of the node at *`loc`*, or `None` if *`loc`* points to the root.

- `down :: Zipper a -> Int -> Maybe (Zipper a)`

  *`down(loc, index=0)`* returns the updated zipper pointing to the *`index`*th child of the current node. But if indexed child does not exist or if the current node is a leaf node, return `None`.

- `left :: Zipper a -> Int -> Maybe (Zipper a)`

  *`left(loc, by=1)`* returns the updated zipper moving left by *`by`* siblings from the current node. If insufficient siblings exist to the left of the current node, return `None`.

- `right :: Zipper a -> Int -> Maybe (Zipper a)`

  *`right(loc, by=1)`* returns the updated zipper moving right by *`by`* siblings from the current node. If insufficient siblings exist to the right of the current node, return `None`.

- `replace :: Zipper a -> a -> Zipper a`

  *`replace(loc, new_node)`* returns the updated zipper replacing the current node with *`new_node`*, without moving.

- `edit :: Zipper a -> (a -> b -> a) -> Zipper a`

  *`edit(loc, f, *args)`* returns the updated zipper replacing the current node with the result of *`f(node, ...args)`* applied to the current node. Here, *`args`* can be an empty/missing list, and only the supplied args are passed.

- `insertLeft :: Zipper a -> a -> Zipper a`

  `insertLeft(loc, new_node)` returns the updated zipper with `new_node` inserted to the left of the current node. The zipper location is not moved.

- `insertRight :: Zipper a -> a -> Zipper a`

  `insertRight(loc, new_node)` returns the updated zipper with `new_node` inserted to the right of the current node. The zipper location is not moved.

- `appendChild :: Zipper a -> a -> Zipper a`

  `appendChild(loc, new_node)` returns the updated zipper with `new_node` inserted as the last child of the current (branch) node. The zipper location is not moved.

- `remove :: Zipper a -> Maybe (Zipper a)`

  `remove(loc)` returns the updated zipper with the current node removed, or `None` if the current node is the root. After the node is removed, the zipper is moved to the next position in depth-first traversal order (i.e., the first right sibling of the node or the parent if there are no right siblings.

This seems like a lot of functions, but they are all quite short, so it's a lost faster than it looks.

Create one or more *factory functions* for creating zippers on common data structures, such as `vector_zipper(root)` that creates a zipper for a nested list like in the examples above. (Other nice possibilities include files/directories, xml files, web pages, URLs/pathnames, ...)

Note: You are free to modify the signatures of the above functions to fit the idioms of your language. For example, languages with pipe operators, like R, it's convenient to have the zipper always be the first argument; but in languages with automatic currying, like Haskell, it makes more sense to have the zipper be the last argument.

Here are a few tips:

1. You want to arrange the left and right siblings list so that the nodes nearest to the current node are easiest to access. For instance, with standard lists, you could have the nearest nodes be first; which means that `lefts` would be kept in reverse order. But this varies with ease of access, e.g., `x[-1]` easily gets the last item python.
2. The three functions in the zipper specification (`is_branch` etc) do not change, but should be accessible given the zipper itself. In this way, the function `up` can create new nodes and access children in the same way for any type of zipper.
3. Try to isolate the checks for None (and the cases that produce them) from the main computations.

**Extensions.** Feel free to enhance your interface with additional useful functions. For instance, `leftmost`, `rightmost`, `lefts`, and `rights` for moving to and accessing siblings; `append_child` and `prepend_child` for inserting a child node without moving, `next` and `prev` for iterating post-order traversal; `path_to` and `move_to` for navigating non-locally in the tree; and so forth.

You may also want to include type annotations/hints/declarations in your code, though this is optional. (It's a good example for practicing using Python's `typing` module.)

**Requirements.**

☐ Implement a zipper library implementing the above interface in your chosen language.

☐ Write a brief script that shows your library in operation on some tree type.

☐ Supply a test suite for your functions. Ensure all tests pass.

**Resources.**

```
from functools import reduce

def pipe( arg, fs, extra_args=[] ):
    """Executes a function pipe, passing each result as input to the next function.

    Given a list of functions [f1, f2, ..., fn], compute f1(args) and pass
    the result to f2, pass that result to f3, and so on, returning the result
    from the call to fn.

    In general, each fi will be either a function or a list [func, *argsi];
    given the result from the previous stage (or args) as result, apply
    func to result and any args in argsi and extra_args in order.

    The list extra_args is passed to every function.
    """
    def rf( result, f ):
        if isinstance(f, list):
            return f[0](result, *(f[1:] + extra_args[:]))
        else:
            return f(result, *extra_args)
    return reduce(rf, fs, arg)
```