IMMUTABLE DICTIONARIES AND TREES 6

COMPUTER SCIENCE 61A

July 5, 2012

1 Dictionaries

Recall that *dictionaries* are data structures that map *keys* to *values*. Dictionaries are usually unordered (unlike real-world dictionaries) – in other words, the key-value pairs are not arranged in the dictionary in any particular order.

1.1 IDicts

There are many different ways to implement dictionaries. The implementation that we use in this course defines an *immutable* dictionary – that is, once the dictionary is created, it cannot be modified. This might seem odd, since there there exists the function `idict_insert`, which "inserts" a new key-value pair into an existing IDict. However, instead of modifying the existing IDict, `idict_insert` returns a duplicate of the original IDict, but with the new key-value pair added. The original IDict remains the same.

The constructors and selectors that we use for our IDict ADT are:

- `make_idict(*mappings)`: The constructor, which takes a tuple of key-value pairs and returns an IDict with the corresponding key-value mappings.
- `idict_select(idict, key)`: A selector, which takes an IDict and a key, and returns the value associated with the key.
- `idict_keys(idict)`: A selector, which takes an IDict and returns a tuple of the keys belonging to the IDict.

1.2 Questions

1. As warm up, define a function `associate`, which takes in two tuples – a tuple of keys and a tuple of values – and returns an IDict of key-value pairs.

```
def associate(keys, values):
    """Returns an IDict that maps the keys to values
    >>> idict_str(associate(('hot', 'internet'), ('dog', 'tubes')))
    ('hot' -> 'dog', 'internet' -> 'tubes')
    """
```

2. Define a function `most_common_letter`, which takes in a `string` and returns the letter that occurs the most often. If the `string` is empty, just return the empty string (which is `''`, two consecutive single quotes). If multiple letters in the `string` occur with equal frequency, return just one of them. You may find it useful to use an IDict in your solution.

```
def most_common_letter(s):
    """Returns the most common letter in s.
    >>> most_common_letter('abracadabra')
    'a'
    """
```

2 Trees

In computer science, *trees* are recursive data structures that are widely used in various settings. This is a diagram of a simple tree.

Notice that the tree branches downward – in computer science, the *root* of a tree starts at the top, and the *leaves* are at the bottom.

Trees consist of two components: datum, and children.

1. **Datum**: Each tree houses one item (datum). The data could be numbers, strings, tuples, etc.
2. **Children**: This is a sequence containing all of its *children* (each of which are trees).

Some terminology regarding trees:

- **Parent node**: A node that has children. Parent nodes can have multiple children.
- **Child node**: A node that has a parent. A child node can only belong to one parent.
- **Root**: The top node. There is only one root. Because every other node branches directly or indirectly from the root, it is possible to start from the root and reach any other node in the tree. The root is, of course, a parent – it is the only node that is not a child. For example, the node that contains the 2 at the top is the root.
- **Leaf**: Nodes that have no children. For example, the nodes that contain the bottom 2, 5, 11, and 4 are leaves. The node that contains 9 is not a leaf, since it has one child.
- **Subtree**: Notice that each child of a parent is itself the root of a smaller tree (for example, the node containing 7 is the root of another tree). This is why trees are *recursive* data structures: trees are made up of subtrees, which are trees themselves.
- **Depth**: How far away a node is from the root. In other words, how many generations away from the root is the specific child node? In the diagram, the node containing 7 has depth 1; the node containing 6 has depth 2. We define the root of a tree to have depth 0.

- **Height**: The depth of the lowest leaf. In the diagram, the nodes containing 5, 11, and 4 are all the "lowest leaves," and they have depth 3. Thus, the entire tree has height 3.

In Computer Science, there are many different types of trees – some vary in the number of children each node has, and others vary in the structure of the tree.

2.1 ITrees

In lecture, we introduced one implementation of an immutable tree: basically, a tree that cannot be modified after creation.

- `make_itree(datum, children=())`: The constructor function.
  - `datum`: The item that is stored in the root of the tree.
  - `children`: A tuple of child trees. Each element in the tuple is itself an ITree. The default value of `children` is an empty tuple, which would make the resulting tree a leaf.
- `itree_datum(t)`: Returns the item stored in the ITree `t`.
- `itree_children(t)`: Returns a tuple of the children of the ITree `t`. If `t` is a leaf, the return value will be an empty tuple.
- `itree_items(t)`: Returns a tuple of all the items in the ITree `t`. This returns the item stored in the root of `t`, as well as the items stored in the child nodes of `t`, and their children, and so on.
- `itree_map(func, t)`: Returns a new ITree whose items are the results of applying `func` to all the items of the ITree `t`.
- `itree_str(t)`: Returns a string representation of an ITree. `itree_str` actually has two other parameters with default values, but you don't have to worry about those.

Remember, treat ITrees as an Abstract Data Type. Use the constructors and selectors defined above to deal with ITrees. Do NOT try to treat ITrees as nested tuples - that would be a Data Abstraction Violation!

2.2 Questions

1. Define a function `square_tree(t)` that squares every item in `t`. You can assume that every item is a number.

   Hint: Use one (or more) of the ITree functions provided above to solve this problem!

   ```
   def square_tree(t):
       """Return a new ITree whose items are all the items of t,
   ```

```
    but squared.
    """
```

2. Define a function `height(t)` that returns the height of an ITree. The height of an ITree is defined as the length of the *longest* path from the root node down to a leaf node. If an ITree just consists of a root with no children, its height is 0.

   If it helps, there is a Python built-in function `max` that takes an arbitrarily long sequence of numbers and returns the maximum value in the sequence.

```
def height(t):
    """Returns the height of the ITree t."""
```

3. Define a function `itree_size(t)`, which returns ho many items the ITree `t` contains. *Note:* Try to solve this problem using recursion, and without using `itree_items`.

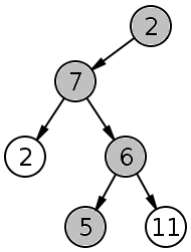
```
def itree_size(t):
    """Returns the number of items in ITree t."""
```

4. Define a function `itree_max(t)` which, given an ITree of numbers, returns the largest number. *Note:* Try to solve this problem using recursion, and without using `itree_items`.

```
def itree_max(t):
    """Returns the max item in the ITree t."""
```

5. Define the procedure `find_path` that, given an `itree` and a datum `datum`, returns the nodes along the path required to get from the root of `itree` to `datum`. If `datum` is not present in `itree`, return `False`. Assume that the elements in `itree` are unique.

   For instance, for the following tree, `find_path` should return:




```
>>> find_path(itree_ex, 5)
(2, 7, 6, 5)
```

   Hint: You can use the `is_leaf` function to check if an ITree is a leaf:

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

def find_path(itree, datum):
```

3 Deep Recursion

Up until now, our recursive procedures operating on data structures (tuples, IRLists, ITrees) usually applied some operation to each datum/value, and joined that to the result of recursively operating on the rest of the data structure. For example, the `map` procedure on tuples:

```
def map(fn, tup):
    if not tup:
        return tup
    else:
        return (fn(tup[0]),) + map(fn, tup[1:])
```

We can read the `map` function as doing:

function `map(fn, tup)`

if `tup` **is empty**
Return the empty tuple.

else:
Apply `fn` to `tup[0]`, and join it to the result of `map`-ing `fn` to the rest of `tup`.

```
>>> mytuple = (1, 4, 6, 9)
>>> map(lambda x: x**2, mytuple)
(1, 16, 36, 81)
```

This assumes that the input tuple only contains numbers. Suppose we wanted to extend our map function to be able to handle nested tuples, i.e.:

```
>>> deeptuple = (4, (3, 2), (5,), (2, (6, 7)), 10)
>>> deepmap(lambda x: x**2, deeptuple)
(16, (9, 4), (25,), (4, (36, 49)), 100)
```

There is now an extra case we have to check for: if the element at `tup[0]` is itself a tuple, then we need to apply `fn` to each element in the tuple `tup[0]`, and then join *this* result to the result of `deepmap`-ing the rest of `tup`:

function `deepmap(fn, tup)`

if `tup` **is empty**
Return the empty tuple.

if `tup[0]` **is a tuple**
`deepmap` the tuple `tup[0]`, and join it to the result of `deepmap`-ing the rest of `tup`.

else:
Apply `fn` to `tup[0]`, and join it to the result of `map`-ing `fn` to the rest of `tup`.

We can use the following helper procedure to check if something is a tuple or not:

```
def is_tuple(thing):
    return type(thing) is tuple
```

Finally, we can translate our idea into code:

```
def deepmap(fn, tup):
    if not tup:
        return tup
    elif is_tuple(tup[0]):
        return (deepmap(fn, tup[0]),) + deepmap(fn, tup[1:])
    else:
        return (fn(tup[0]),) + deepmap(fn, tup[1:])
```

3.1 Questions

1. Louis Reasoner is copying the `deepmap` procedure, and accidently makes a typo:

```
def deepmap_oops(fn, tup):
    if not tup:
        return tup
    elif is_tuple(tup[0]):
        return deepmap_oops(fn, tup[0]) + deepmap_oops(fn, tup[1:])
    else:
        return (fn(tup[0]),) + deepmap_oops(fn, tup[1:])
```

What does this procedure end up doing if given a tuple containing nested tuples? For instance, what will `deepmap_oops` return for the following?

```
>>> deepmap_oops(lambda x: x*x, (1, (2, (3, 4)), 5, (6,)))
```

2. Define a procedure `deepreverse` that, given a tuple, reverses the tuple, and any nested tuple.

```
>>> mytuple = ('i', 'read', ('the', ('news', 'today')), ('oh', 'boy'))
>>> deepreverse(mytuple)
(('boy', 'oh'), (('today', 'news'), 'the'), 'read', 'i')
```

```
def deepreverse(tup):
```

Note: For the following IRList procedures, you'll want to use the `could_be_irlist` that, given a thing, returns `True` if the thing might be an IRList.

3. Define the procedure `irlist_deepmap` that, given a function `fn` and an IRList, performs a deep map.

```
>>> myirlist = irlist_populate(irlist_populate(4, 3),
                  irlist_populate(5,
                    irlist_populate(8, 6, 2)))
>>> irlist_str(myirlist)
```

```
<<4, 3>, <5, <8, 6, 2>>
>>> irlist_str(irlist_deepmap(lambda x: x*x, myirlist))
<<16, 9>, <25, <64, 36, 4>>

def irlist_deepmap(fn, irlist):
```

4. Define the procedure `irlist_flatten` that, given an IRList that possibly contains nested IRLists, returns a new IRList with all elements 'un-nested':

```
>>> myirlist = irlist_populate(irlist_populate(4, 3),
                 irlist_populate(5,
                   irlist_populate(8, 6, 2)))
>>> irlist_str(myirlist)
<<4, 3>, <5, <8, 6, 2>>
>>> irlist_str(irlist_flatten(myirlist))
<4, 3, 5, 8, 6, 2>

def irlist_flatten(irl):
```