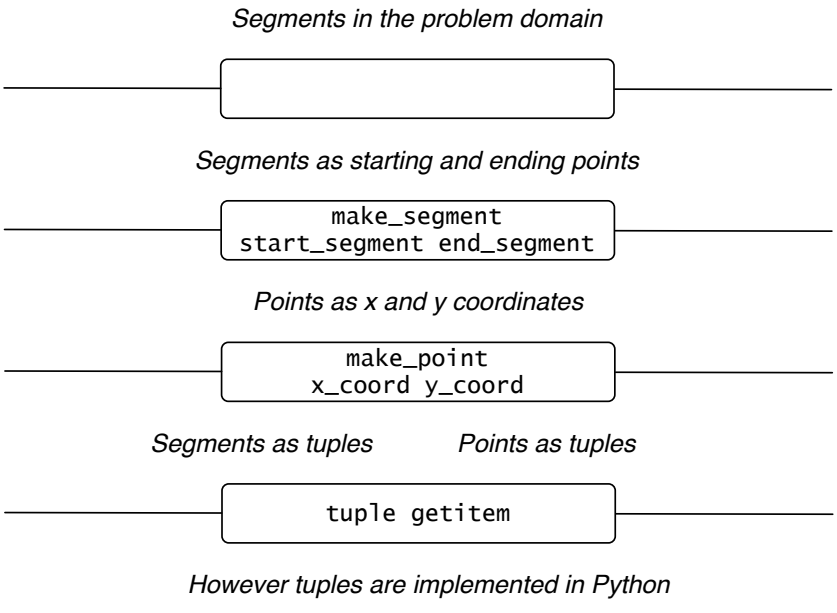
CS61A Notes – Week 5: Sequences

Abstract Data Types (ADT)

In the previous lab you were asked to implement constructors and selectors to represent points and segments in an abstract manner. An example of an acceptable implementation is:

```
def make_point(x, y):
    return (x, y)

def x_point(p):
    return p[0]

def y_point(p):
    return p[1]

def make_segment(start_point, end_point):
    return (start_point, end_point)

def start_point(s):
    return s[0]

def end_point(s):
    return s[1]
```

The abstraction barriers implicit in the representation implemented above are made explicit in the diagram below:



Note that none of the functions in the implementation directly call any functions or access data across the abstraction barrier below them. This type of abstraction is very useful when writing complex code, especially when other programmers are sharing the same codebase.

If the implementation of a function or underlying representation of data needs to be modified, no code above the abstraction barrier need to be changed as long as all assumptions about the input and output of the lower-level functions are maintained; we'll elaborate on this in the exercises below.

EXERCISES

1. Andy Abstrakshon was working on adding the `midpoint_segment f`unction from this week's lab, which takes a segment as the argument and returns its midpoint. However, he does not know about data abstraction and produced the flawed code below. Add this new function name to the proper place in the diagram above and then correct Andy's code so that it no longer breaks any abstraction barriers.

```
def midpoint_segment(seg):
    p1, p2 = start_segment(seg), end_segment(seg)
    new_x = (p1[0] + p2[0])/2
    new_y = (p1[1] + p2[1])/2
    return (new_x, new_y)
```

2. The advantage of working with abstract data types is that it separates the implementation details from the higher-level code using the ADT. For example suppose that Andy was working with Bob on a partner-project. Andy can work on implementing the segment functions without worrying about the way Bob will implement the point functions. That is, Andy can make the segment ADT while Bob simultaneously makes the point ADT.

   To emphasize this, consider your fixed midpoint_segment from the previous exercise. Since Bob just spent 2 weeks working on the Pig project, which uses higher-order functions, Bob decided to implement the point ADT with the following:

```
def make_point(x, y):
    def point(z):
        if z == 1:
            return x
        else:
            return y
    return point
```

   Finish defining the x-point and y-point functions. Verify that your solution to part 1 which obeys the abstraction barrier for points works just as well with this implementation of points.

Sequences

From the pig project, we discovered the utility of having structures that contain multiple values. Today, we are going to cover some ways we can model and create these structures, which are called sequences.

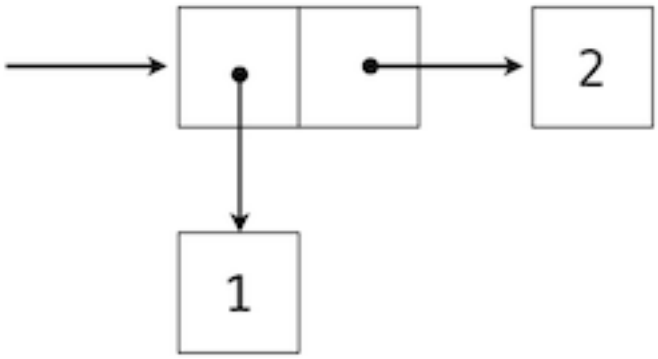
A sequence is an ordered collection of data values. Unlike a pair, which has exactly two elements, a sequence can have an arbitrary (but finite) number of ordered elements.

A sequence is not a particular abstract data type, but instead a collection of behaviors that different types share. That is, there are many kinds of sequences, but they all share certain properties. In particular:

1. **Length**: a sequence has a finite length.
2. **Element selection**: A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

Nested pairs and "box and pointer" notation

We have seen pairs implemented using both tuples and functions. When you want to draw a pair (or other sequencing structures) on paper, computer scientists often use "box and pointer" diagrams. For example, the pair `(1, 2)` would be represented as the following box and pointer diagram:



Box and pointer diagrams are useful, since they show the structure and elements of a pair or sequence very clearly. The steps to construct such a diagram are as follows:

a. Represent a pair as two horizontally adjacent boxes.
b. Draw arrows from inside the boxes to the contents of the pair, which can also be pairs (or any other value).
c. Draw a *starting arrow* to the first entry in your diagram. Note that the starting arrow in the above diagram points to the *entirety of the two boxes*, and not just the first box.
d. Don't forget your starting arrow! Your diagram will spontaneously burst into flames without it.

The arrows are also called **pointers**, indicating that the content of a box "points" to some value.

EXERCISES

1. Draw a box and pointer diagram for the following code:

```
>>> x = make_pair(make_pair(42, 28), make_pair(2, 3))
```

2. Given the definition of `x` above, what will `getitem_pair(getitem_pair(x, 1), 0)` return?

Recursive Lists

In lab we created our own type of sequence, called a **recursive list** (or **rlist**). Recursive lists are chains of pairs. Each pair will hold a value in our list (referred to as the **first** element of the pair) and a pointer to the **rest** of the recursive list (another pair). In lab we defined the following constructors and selectors:

```
empty_rlist = None

def make_rlist(first, rest = empty_rlist):
    return (first, rest)

def first(s):
    return s[0]

def rest(s):
    return s[1]
```

Remember to use the name `empty_rlist` (instead of None) when working with recursive lists in order to respect the abstraction, or Prof. Hilfinger will set your paper on fire.

To formalize our definition, a recursive list is something that fits either of the following two descriptions:

1. A pair whose `first` item can be anything, and whose `rest` item is itself a recursive list.
2. The empty_rlist.

Since we want recursive lists to be a *sequence*, we should be able to find the **length** of a recursive list and be able to **select any element** from the recursive list.

Functions to find such things are in the lecture notes, but let's try re-implementing them in the exercises below:

EXERCISES

1. Draw the box and pointer diagram for this rlist:

```
make_rlist(1,
  make_rlist(2,
    make_rlist(make_rlist(3, empty_rlist),
      make_rlist(4,
        make_rlist(5, empty_rlist)))))
```

2. Using calls to `make_rlist`, create the rlist that is printed as :

```
(3, (2, (1, ('Blastoff', None))))
```

3. Write the function `len_rlist` that takes an rlist and returns its length. (This is in the lecture notes as well.)

4. Write the function `last` that takes an rlist and returns the last item in that rlist (assume the function `getitem_rlist` is not yet defined).

5. Write the function `getitem_rlist` that takes an rlist and an index and returns the element at that index:

```
>>> x = make_rlist(2, make_rlist(3, empty_rlist))
>>> getitem_rlist(x, 1)
3
```

Tuples

Tuples are another kind of sequence, which we also talked about in lab. Since we spent a lot of time on them in lab, we will only briefly review them here. You can make a tuple by enclosing a comma separated set of elements inside parentheses.

```
>>> (1, 2, 3)
(1, 2, 3)
```

Since tuples are a type of sequence, that means we should have a way to get the length of the sequence:

```
>>> len((1, 2, 3))
3
```

And we should have a way to pull elements out:

```
>>> (1, 2, 3)[1]
2
```

In fact we've seen tuples in the first project when you return multiple values from a function:

```
>>> def foo(a, b):
...     return a, b
...
>>> tup = foo(1, 2)
>>> tup
(1, 2)
```

EXERCISES

1. Write a function, `sum`, which uses a while loop to calculate the sum of the elements of a tuple. (Note: `sum` is actually already built into Python!)

```
>>> sum((1, 2, 3, 4, 5))
15
```

Sequence Iteration with For Loops

In a lot of our sequence questions so far, we've ended up with code that looks like this:

```
i = 0
while i < len(sequence):
    elem = sequence[i]
    # do something with elem
    i += 1
```

This particular construct happens to be incredibly useful because it gives us a way to look at each element in a sequence. In fact, iterating through a sequence is so common that Python actually gives us a special piece of syntax to do it, called the for loop:

```
for elem in sequence:
    # do something with elem
```

Look at how much shorter that is! More generally, sequence can by any expression that evaluates to a sequence, and `elem` is simply a variable name. On the first iteration through this loop, `elem` will be bound to the first element in sequence in the current environment. On the second iteration, `elem` will be bound to the first element in sequence in the current environment. On the second iteration, `elem` gets rebound to the second element. This process repeats until `elem` has been bound to each element in the sequence, at which point the for loop terminates.

EXERCISES

1. Implement `sum` one more time, this time using a for loop.

2. Now use a for loop to write a function, `filter`, that takes a predicate of one argument and a sequence and returns a tuple. (A predicate is a function that returns `True` or `False`.) This tuple should contain the same elements as the original sequence, but *without* the elements that do not match the predicate (i.e. the predicate returns `False` when you call it on that element).

```
>>> filter(lambda x: x % 2 == 0, (1, 4, 2, 3, 6))
(4, 2, 6)
```

3. It's often useful to pair values in a sequence with names. If we do that, then instead of having to remember the index of the value you are interested in, you only need to remember the name associated with that value. (This has proven to be so useful that Python actually has something like this built-in, called dictionaries, which you will learn about later.) In this section, we'll build an abstract data type called an association list.

   We can implement this as a sequence of 2-tuples. The first element (index 0) of the tuple will be the 'key' associated with the value, and the second element (index 1) will be the 'value' itself.

   a. Write a constructor, `make_alist`, that takes a sequence of keys and a sequence of corresponding values, and returns an association list (that is, a sequence of key-value tuples).

      ```
      >>> make_alist(('a', 'b','c'), (1, 2, 3))
      (('a', 1), ('b', 2), ('c', 3))
      ```

   b. Write a selector, `lookup`, that takes an alist and a key and returns the value that corresponds to that key, or `None` if the key is not in the alist.

   c. Finally, write a procedure, change, that takes an alist, a key, and a value. It should return a new alist that matches the original, but with the following difference:
      - c.i. If the key is already present in the original alist, replace its corresponding value with the argument value
      - c.ii. Otherwise, add the key and value to the end of the alist.