CS98-52 Lecture 4

Wednesday, October 2nd, 2019

Anonymous Factorial in HW3

> **Question 6**
> The recursive factorial function can be written as a single expression by using a conditional expression.
>
> ```
> >>> fact = lambda n: 1 if n == 1 else mul(n, fact(sub(n, 1)))
> >>> fact(5)
> 120
> ```
>
> However, this implementation relies on the fact (no pun intended) that fact has a name, to which we refer in the body of fact. To write a recursive function, we have always given it a name using a def or assignment statement so that we can refer to the function within its own body. In this question, your job is to define fact recursively without giving it a name!
>
> Write an expression that computes n factorial using only call expressions, conditional expressions, and lambda expressions (no assignment or def statements). Note in particular that you are not allowed to use make_anonymous_factorial in your return expression. The sub and mul functions from the operator module are the only built-in functions required to solve this problem.

Solution

```
lambda n: (lambda f: f(n,f))(lambda n, f: 1 if n == 1 else n*f(n-1,f)))(5)
```

With just call and lambda expressions, you can even create iterative and recursive procedures.

Data Representations

Back then, 61A was taught in Scheme, an older programming language with a minimal set of features. Scheme only had call expressions and ifs and elses, but no while loops, lists, or dictionaries. That means student had to build their own versions of these feature.

Next week, we will cover how to build the object system from functions! Let's learn how to do this in Python.

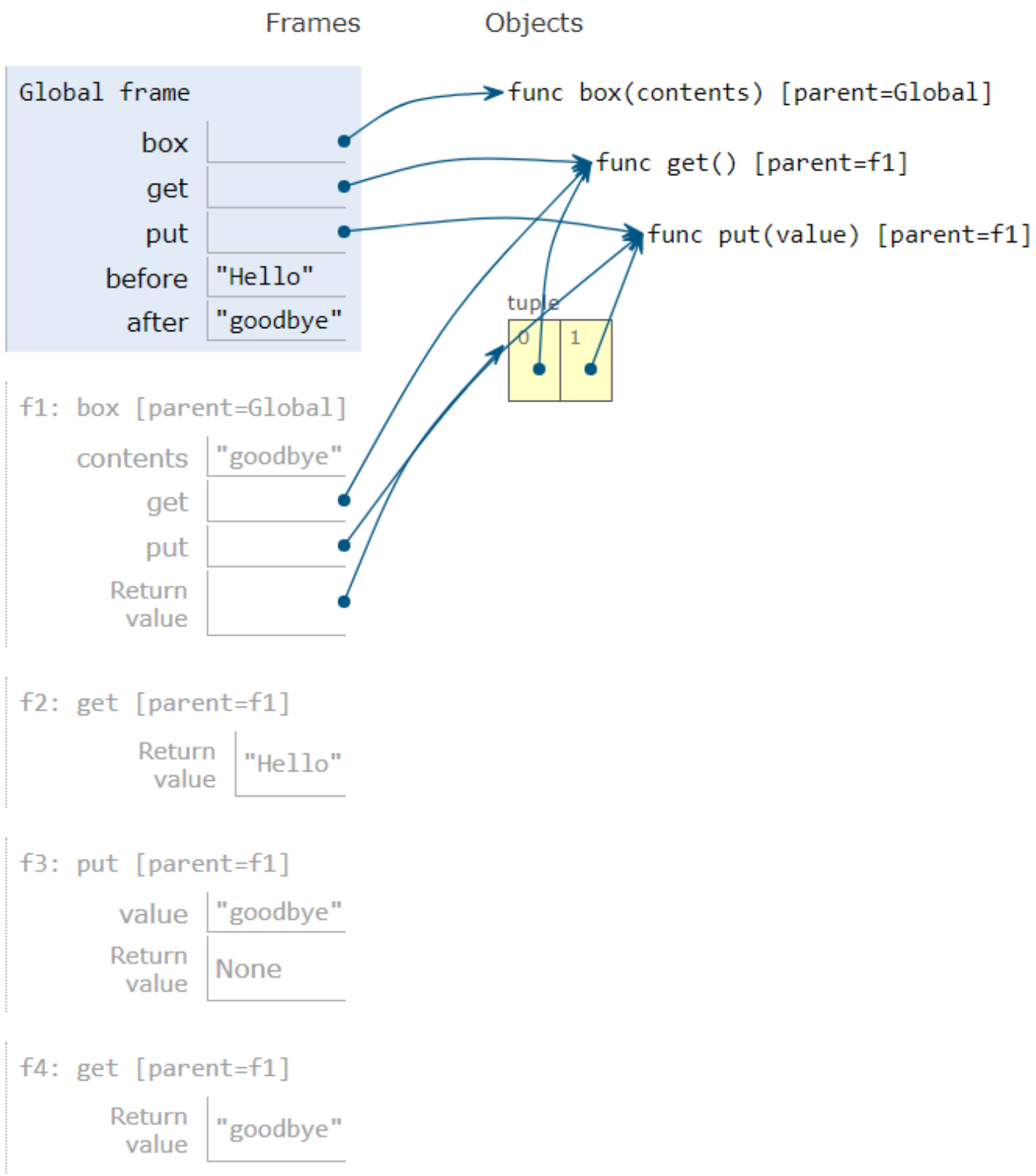
Functions with Shared Local State

```
def box(contents):
    def get():
        return contents
    def put(value):
        nonlocal contents
        contents = value
    return get, put

get, put = box('Hello')
```

```
before = get()
put('goodbye')
after = get()
```

We have two functions, one that returns the box's contents, and the other that changes what the contents are.



The environment diagram shows us a lie that we've been told so far about how Python executes multiple assignments. You can't return more than one value. What Python is doing is instead building a tuple, and then it immediately gets broken apart when the return value is assigned to the same number of elements.

Pairs Implemented as Functions

```
def pair(x, y):
    def dispatch(m):
        if m == 'first':
            return x
        elif m == 'second':
            return y
    return dispatch
```

The constructor is a higher order function, and the inner function represents the pair (x,y). Even though this is a list of functions, we can just represent the numbers with box and pointer notation.

```
>>> p = pair(3,pair(4,5))
>>> p('first')
3
>>> p('second')('first')
4
```

> **Determining whether something is a function**
> To check if something is a built-in function:
>
> ```
> >>> type(abs) == type(print)
> True
> >>> type(abs)
> class.built_in_function
> >>> type(pair)
> class.function
> >>> def user_function(x):
> ...     return x
> ...
> >>> type(pair) == type(user_function)
> True
> ```

Linked Lists

An empty list is called “nil” and represented as None, and a non-empty list is represented as a pair, where the first element of the pair is the first element of the lest, and the second is the rest of the list.

One implementation of linked lists is to use the `pair` function above:

```
>>> pair(3,pair(4,pair(5,nil)))
```

Or you could also use normal lists:

```
>>> [3,[4,[5,nil]]]
```

To check how long a linked list with the `pair` function is:

```
nil = None
def list_len(s):
    if s is nil:
```

```
        return 0
    else:
        return 1 + list_len(s('second'))
```

And to append an element to the middle of a list built with the `pair` function:

```
def append(s,x):
    if s is nil:
        return pair(x, nil)
    else:
        first, rest = s('first'), s('second')
        return pair(first, append(rest,x))
```

Creating your own list system

Let's try to emulate the 61A of old, where students had to build their own list system.

```
def make_list():
    s = nil
    def dispatch(m):
        if m == 'append':
            nonlocal s
            s = append(s,x)
        if m == 'len':
            return list_len(s)
        if m == 'print':
            return lambda i: list_get(s,i)
    return dispatch
```

This got kind of long though. We could probably do this without a nonlocal statement, or at least within this function:

```
def make_list():
    s = make_box(nil)
    def dispatch(m):
        if m == 'append':
            return lambda x: s('put')(append(s('get'),x))
        if m == 'len':
            return list_len(s)
        if m == 'print':
            return list_print(s)
    return dispatch
```

We've built a pretty close approximation of the Python list using just functions, but we don't yet have indexing:

```
def list_get(s,i):
    if i == 0:
        return s('first')
    else:
        return list_get(s('second'),i-1)
```

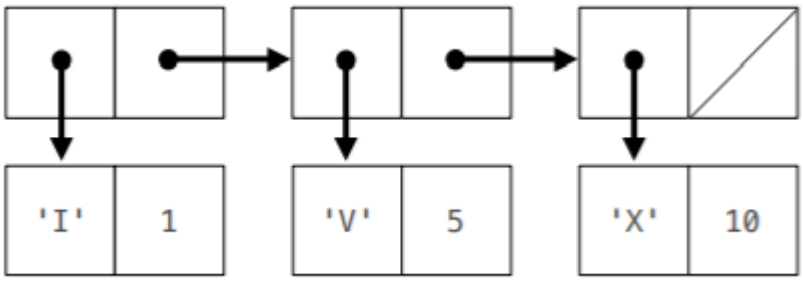
```
def make_list():
    s = make_box(nil)
    def dispatch(m):
        if m == 'append':
            return lambda x: s('put')(append(s('get'),x))
        if m == 'len':
            return list_len(s)
        if m == 'print':
            return list_print(s)
        if m == 'get':
            return list_get(s,)
    return dispatch
```

An Inefficient Dictionary Implementation

We can use a linked list of key-value pairs to implement a dictionary behavior, even if it is rather inefficient.

It is inefficient in the sense that dictionaries run on constant time, fetching a result in the same amount of time whether it has 1 or 1000 entries. Our implementation will not do that, because the code required to implement such a solution is much too long to cover in one lecture.

```
>>> d = dict_dispatch()
>>> d('set')('I', 1)
>>> d('set')('V', 5)
>>> d('set')('X', 10)
```



Dispatch Dictionaries

What we've done so far: enumerating different messages in conditional statements, isn't very convenient. Equality tests are repetitive and we can't add new messages without re-writing the dispatch function.

A dispatch dictionary has messages as keys and functions or data objects as values. The dictionary will handle the message look-up logic and we can concentrate on implementing behavior.

Instead of:

```
def box_dispatch(contents):
    def dispatch(m):
        if m == 'contents':
            return contents
        if m == 'put':
            def put(value):
                nonlocal contents
```

```
                contents = value
            return put
    return dispatch
```

We can write:

```
def box_dict(contents):
    def put(value):
        d['contents']=value
    d = {'contents': contents, 'put': put}
    return d
```

Here, we've used the built-in dictionary implementation in Python, but you are free to use the dictionary implementation we wrote above to implement this behavior.

Extra topics

What we will discuss here is a little disconnected from what we've learnt today:

Constraint Programming

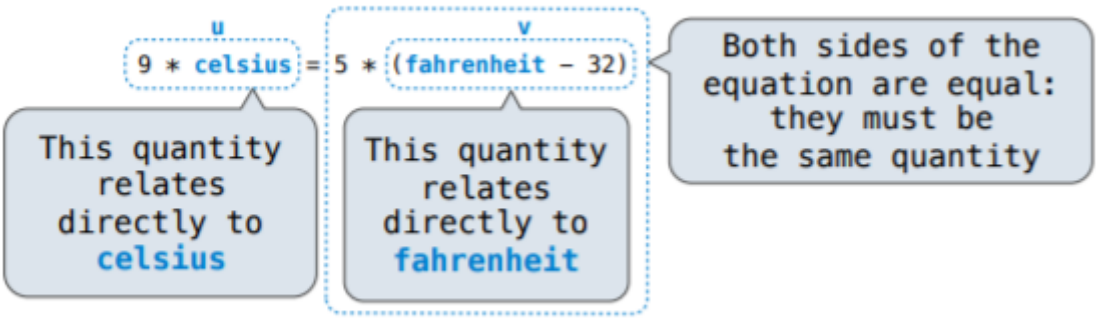
Here, we are going to learn how to solve for variables in an equation.

- Algebraic equations are declarative. They describe a relationship among different quantities.
- Python functions are procedural: the describe how to compute a result from a set of input arguments. They are one-way.

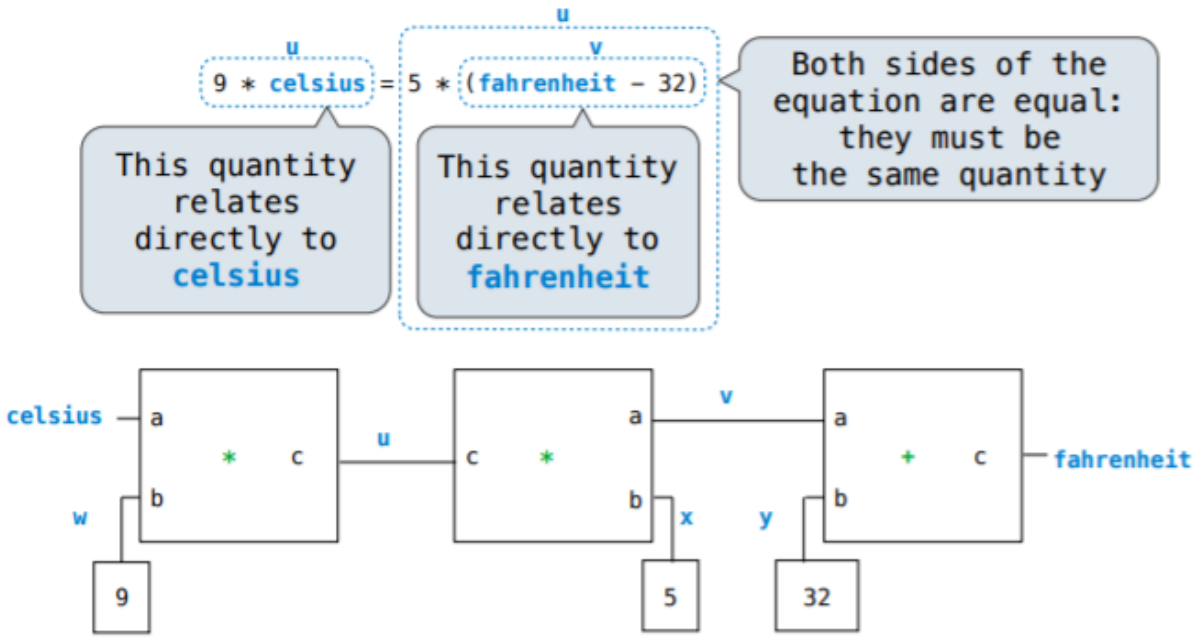
Constraint programming states that we want to implement behavior so that we can define the relationship between quantities, provide values for known quantities, and our computer will calculate values for unknown quantities. In other words, we want a general means of combination. In even simpler terms: how do we make Python behave more like algebra?

A constraint network for temperature conversion

The big idea is that we will give all intermediate quantities a value!



By doing so, we can define relationships between these values in a way that our computer will be able to solve for unknown values.



You can read more about the actual code here (scroll to the bottom): CS61A Code