CS 61A Spring 2023 — Environment Diagrams, Higher-Order Functions

Discussion 2: February 1, 2023

Note on web discussions: In order to use the environment diagrams on the site, please log in using the account you use for Okpy.

Call Expressions

Call expressions, such as `square(2)`, apply functions to arguments. When executing call expressions, we create a new frame in our diagram to keep track of local variables:

1. Evaluate the operator, which should evaluate to a function.
2. Evaluate the operands from left to right.
3. Draw a new frame, labelling it with the following:
   - A unique index (`f1`, `f2`, `f3`, ...).
   - The **intrinsic name** of the function, which is the name of the function object itself. For example, if the function object is `func square(x) [parent=Global]`, the intrinsic name is `square`.
   - The parent frame (`[parent=Global]`).
4. Bind the formal parameters to the argument values obtained in step 2 (e.g. bind `x` to 3).
5. Evaluate the body of the function in this new frame until a return value is obtained. Write down the return value in the frame.

If a function does not have a return value, it implicitly returns `None`. In that case, the "Return value" box should contain `None`.

Note: Since we do not know how built-in functions like `min(...)` or imported functions like `add(...)` are implemented, we do not draw a new frame when we call them, since we would not be able to fill it out accurately.

Q1: Call Diagram

Let's put it all together! Draw an environment diagram for the following code. You may not have to use all of the blanks provided to you.

See the web version of this resource for the environment diagram.

Video diagram

Q2: Nested Calls Diagrams

Draw the environment diagram that results from executing the code below. You may not need to use all of the frames and blanks provided to you.

See the web version of this resource for the environment diagram.

Video walkthrough

Lambda Expressions

A lambda expression evaluates to a function, called a lambda function. For example, `lambda y: x + y` is a lambda expression, and can be read as "a function that takes in one parameter `y` and returns `x + y`."

A lambda expression by itself evaluates to a function but does not bind it to a name. Also note that the return expression of this function is not evaluated until the lambda is called. This is similar to how defining a new function using a def statement does not execute the function's body until it is later called.

```
>>> what = lambda x : x + 5
>>> what
<function <lambda> at 0xf3f490>
```

Unlike `def` statements, lambda expressions can be used as an operator or an operand to a call expression. This is because they are simply one-line expressions that evaluate to functions. In the example below, `(lambda y: y + 5)` is the operator and `4` is the operand.

```
>>> (lambda y: y + 5)(4)
9
>>> (lambda f, x: f(x))(lambda y: y + 1, 10)
11
```

Q3: Lambda the Environment Diagram

Draw the environment diagram for the following code and predict what Python will output.

See the web version of this resource for the environment diagram.

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

Higher Order Functions

A **higher order function** (HOF) is a function that manipulates other functions by taking in functions as arguments, returning a function, or both. For example, the function `compose` below takes in two functions as arguments and returns a function that is the composition of the two arguments.

```
def composer(func1, func2):
    """Return a function f, such that f(x) = func1(func2(x))."""
    def f(x):
        return func1(func2(x))
    return f
```

HOFs are powerful abstraction tools that allow us to express certain general patterns as named concepts in our programs.

HOFs in Environment Diagrams

An **environment diagram** keeps track of all the variables that have been defined and the values they are bound to. However, values are not necessarily only integers and strings. Environment diagrams can model more complex programs that utilize higher order functions.

See the web version of this resource for the environment diagram.

Lambdas are represented similarly to functions in environment diagrams, but since they lack instrinsic names, the lambda symbol ( ) is used instead.

The parent of any function (including lambdas) is always the frame in which the function is defined. It is useful to include the parent in environment diagrams in order to find variables that are not defined in the current frame. In the previous example, when we call `add_two` (which is really the lambda function), we need to know what `x` is in order to compute `x + y`. Since `x` is not in the frame `f2`, we look at the frame's parent, which is `f1`. There, we find `x` is bound to 2.

As illustrated above, higher order functions that return a function have their return value represented with a pointer to the function object.

Q4: Make Adder

Draw the environment diagram for the following code:

See the web version of this resource for the environment diagram.

There are 3 frames total (including the Global frame). In addition, consider the following questions:

1. In the Global frame, the name `add_ten` points to a function object. What is the intrinsic name of that function object, and what frame is its parent?
2. What name is frame `f2` labeled with (`add_ten` or)? Which frame is the parent of `f2`?
3. What value is the variable `result` bound to in the Global frame?

You can try out the environment diagram at tutor.cs61a.org. To see the environment diagram for this question, click here.

1. The intrinsic name of the function object that `add_ten` points to is   (specifically, the lambda whose parameter is `k`). The parent frame of this lambda is `f1`.
2. `f2` is labeled with the name . The parent frame of `f2` is `f1`, since that is where   is defined.
3. The variable `result` is bound to 19.

Q5: Make Keeper

Write a function that takes in a number `n` and returns a function that can take in a single parameter `cond`. When we pass in some condition function `cond` into this returned function, it will print out numbers from 1 to `n` where calling `cond` on that number returns `True`.

```
def make_keeper(n):
    """Returns a function which takes one parameter cond and prints
    out all integers 1..i..n where calling cond(i) returns True.

    >>> def is_even(x):
    ...     # Even numbers have remainder 0 when divided by 2.
    ...     return x % 2 == 0
    >>> make_keeper(5)(is_even)
    2
    4
    """
    def do_keep(cond):
        i = 1
        while i <= n:
            if cond(i):
                print(i)
            i += 1
    return do_keep
```

Currying

One important application of HOFs is converting a function that takes multiple arguments into a chain of functions that each take a single argument. This is known as **currying**. For example, the function below converts the `pow` function into its curried form:

```
>>> def curried_pow(x):
        def h(y):
            return pow(x, y)
        return h

>>> curried_pow(2)(3)
8
```

This is useful if, say, you needed to calculate a lot of powers of 2. Using the normal `pow` function, you would have to put in 2 as the first argument for every function call:

```
>>> pow(2, 3)
8
>>> pow(2, 4)
16
>>> pow(2, 10)
1024
```

With `curried_pow`, however, you can create a one-argument function specialized for taking powers of 2 one time, and then keep using that function for taking powers of 2:

```
>>> pow_2 = curried_pow(2)
>>> pow_2(3)
8
>>> pow_2(4)
16
>>> pow_2(10)
1024
```

This way, you don't have to put `2` in as an argument for every call! If instead you wanted to take powers of 3, you could quickly make a similar function specialized in taking powers of 3 using `curried_pow(3)`.

Another point that will be relevant once you learn about sequences: Currying is also helpful in contexts where only one-argument functions are allowed, such as with the `map` function. The `map` applies a one-argument function to every term in a sequence. If we wanted to take the power of 2 for every number in a sequence using `map`, we would be forced to use the one-argument function `pow_2` (as defined in the example above) instead of using `pow` with 2 as the first argument.

Q6: Currying

Write a function `curry` that will curry any two argument function.

```
def curry(func):
    """
    Returns a Curried version of a two-argument function FUNC.
    >>> from operator import add, mul, mod
    >>> curried_add = curry(add)
    >>> add_three = curried_add(3)
    >>> add_three(5)
    8
    >>> curried_mul = curry(mul)
    >>> mul_5 = curried_mul(5)
    >>> mul_5(42)
    210
    >>> curry(mod)(123)(10)
    3
    """
    def first(arg1):
        def second(arg2):
            return func(arg1, arg2)
        return second
    return first
```

HOFs and Lambdas

Q7: Make Your Own Lambdas

For each of the following expressions, write functions `f1`, `f2`, `f3`, and `f4` such that the evaluation of each expression succeeds, without causing an error. Be sure to use lambdas in your function definition instead of nested `def` statements. Each function should have a one line solution.

```python
def f1():
    """
    >>> f1()
    3
    """
    return 3

def f2():
    """
    >>> f2()()
    3
    """
    return lambda: 3

def f3():
    """
    >>> f3()(3)
    3
    """
    return lambda x: x

def f4():
    """
    >>> f4()()(3)()
    3
    """
    return lambda: lambda x: lambda: x
```

Q8: Lambdas and Currying

Write a function `lambda_curry2` that will curry any two argument function like with `curry`, but this time using lambdas.

Your solution to this problem should only be one line.

```
def lambda_curry2(func):
    """
    Returns a Curried version of a two-argument function FUNC.
    >>> from operator import add, mul, mod
    >>> curried_add = lambda_curry2(add)
    >>> add_three = curried_add(3)
    >>> add_three(5)
    8
    >>> curried_mul = lambda_curry2(mul)
    >>> mul_5 = curried_mul(5)
    >>> mul_5(42)
    210
    >>> lambda_curry2(mod)(123)(10)
    3
    """
    return lambda arg1: lambda arg2: func(arg1, arg2)
```

Extra Practice

This question is particularly challenging, so it is recommended to attempt if you are feeling confident on the previous questions or are studying for the exam.

Q9: Match Maker

Implement `match_k`, which takes in an integer `k` and returns a function that takes in a variable `x` and returns `True` if all the digits in `x` that are `k` apart are the same.

For example, `match_k(2)` returns a one argument function that takes in `x` and checks if digits that are 2 away in `x` are the same.

`match_k(2)(1010)` has the value of `x = 1010` and digits 1, 0, 1, 0 going from left to right. `1 == 1` and `0 == 0`, so the `match_k(2)(1010)` results in `True`.

`match_k(2)(2010)` has the value of `x = 2010` and digits 2, 0, 1, 0 going from left to right. `2 != 1` and `0 == 0`, so the `match_k(2)(2010)` results in `False`.

Important: You may not use strings or indexing for this problem. You do not have to use all the lines, one staff solution does not use the line directly above the while loop.

Hint: Floor dividing by powers of 10 gets rid of the rightmost digits.

```python
def match_k_alt(k):
    """ Return a function that checks if digits k apart match

    >>> match_k_alt(2)(1010)
    True
    >>> match_k_alt(2)(2010)
    False
    >>> match_k_alt(1)(1010)
    False
    >>> match_k_alt(1)(1)
    True
    >>> match_k_alt(1)(2111111111111111)
    False
    >>> match_k_alt(3)(123123)
    True
    >>> match_k_alt(2)(123123)
    False
    """
    def check(x):
        while x // (10 ** k):
            if (x % 10) != (x // (10 ** k)) % 10:
                return False
            x //= 10
        return True
    return check
```

Here's an alternate solution:

```
# Alternate solution
def match_k(k):
    """ Return a function that checks if digits k apart match

    >>> match_k(2)(1010)
    True
    >>> match_k(2)(2010)
    False
    >>> match_k(1)(1010)
    False
    >>> match_k(1)(1)
    True
    >>> match_k(1)(2111111111111111)
    False
    >>> match_k(3)(123123)
    True
    >>> match_k(2)(123123)
    False
    """
    # BEGIN SOLUTION ALT="____________________________" NO PROMPT
    def check(x):
        i = 0
        while 10 ** (i + k) < x:
            if (x // 10**i) % 10 != (x // 10 ** (i + k)) % 10:
                return False
            i = i + 1
        return True
    return check
# END SOLUTION
```