Discussion 2: Higher-Order Functions, Self Reference, Lambda Expressions

This is an online worksheet that you can work on during discussions and tutorials. Your work is not graded and you do not need to submit anything.

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 `compose1` below takes in two functions as arguments and returns a function that is the composition of the two arguments.

```
def compose1(f, g):
    def h(x):
        return f(g(x))
    return h
```

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

A Note on 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
```

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
```

Q1: Keep Ints

Write a function that takes in a function `cond` and a number `n` and prints numbers from 1 to `n` where calling `cond` on that number returns `True`.

```
def keep_ints(cond, n):
    """Print out all integers 1..i..n where cond(i) is true
    >>> def is_even(x):
    ...     # Even numbers have remainder 0 when divided by 2.
    ...     return x % 2 == 0
    >>> keep_ints(is_even, 5)
    2
    4
    """
    "*** YOUR CODE HERE ***"

    i = 1
```

Q2: (Tutorial) Make Keeper

Write a function similar to `keep_ints` like in Question 1 #, but now it takes in a number `n` and returns a function that has one parameter `cond`. The returned function prints out numbers from 1 to `n` where calling `cond` on that number returns `True`.

```
5           ...     return x % 2 == 0
6           >>> make_keeper(5)(is_even)
7           2
8           4
9       """
10      "*** YOUR CODE HERE ***"
11      def g(cond):
12          i = 1
13          while i<=n:
14              if (cond(i)):
15                  print(i)
16              i+=1
```

HOFs in Environment Diagrams

Recall that 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.

Server error! Your code might have an INFINITE LOOP or be running for too long.
The server may also be OVERLOADED. Or you're behind a FIREWALL that blocks access.
Try again later. This site is free with NO technical support. [#UnknownServerError]

(see UNSUPPORTED FEATURES)

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

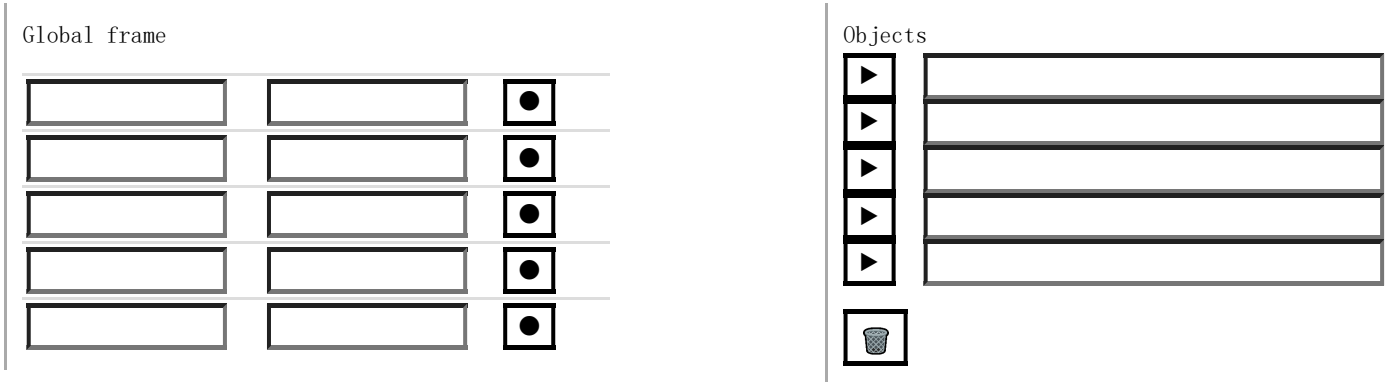
As illustrated above, higher order functions that return a function have their return value represented with a pointer to the function object.

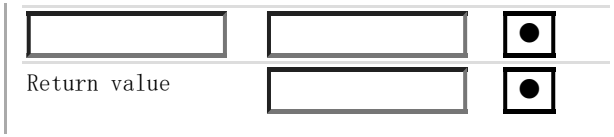
Q3: Curry2 Diagram

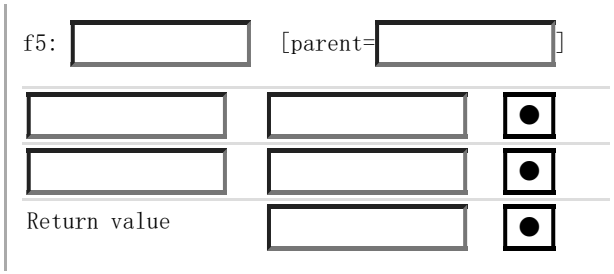
Draw the environment diagram that results from executing the code below.

```
def curry2(h):
    def f(x):
        def g(y):
            return h(x, y)
        return g
    return f

make_adder = curry2(lambda x, y: x + y)
add_three = make_adder(3)
add_four = make_adder(4)
five = add_three(2)
```


Global frame
Objects
f1: [parent=]
Return value
f2: [parent=]
Return value
f3: [parent=]
Return value
f4: [parent=]


Return value


f5:
[parent=
]
Return value

Q4: Curry2 Lambda

Write `curry2` as a lambda function.

```
"*** YOUR CODE HERE ***"
curry2 = lambda h: lambda x: lambda y: h(x, y)

```

Q5: (Tutorial) HOF Diagram Practice

Draw the environment diagram that results from executing the code below.

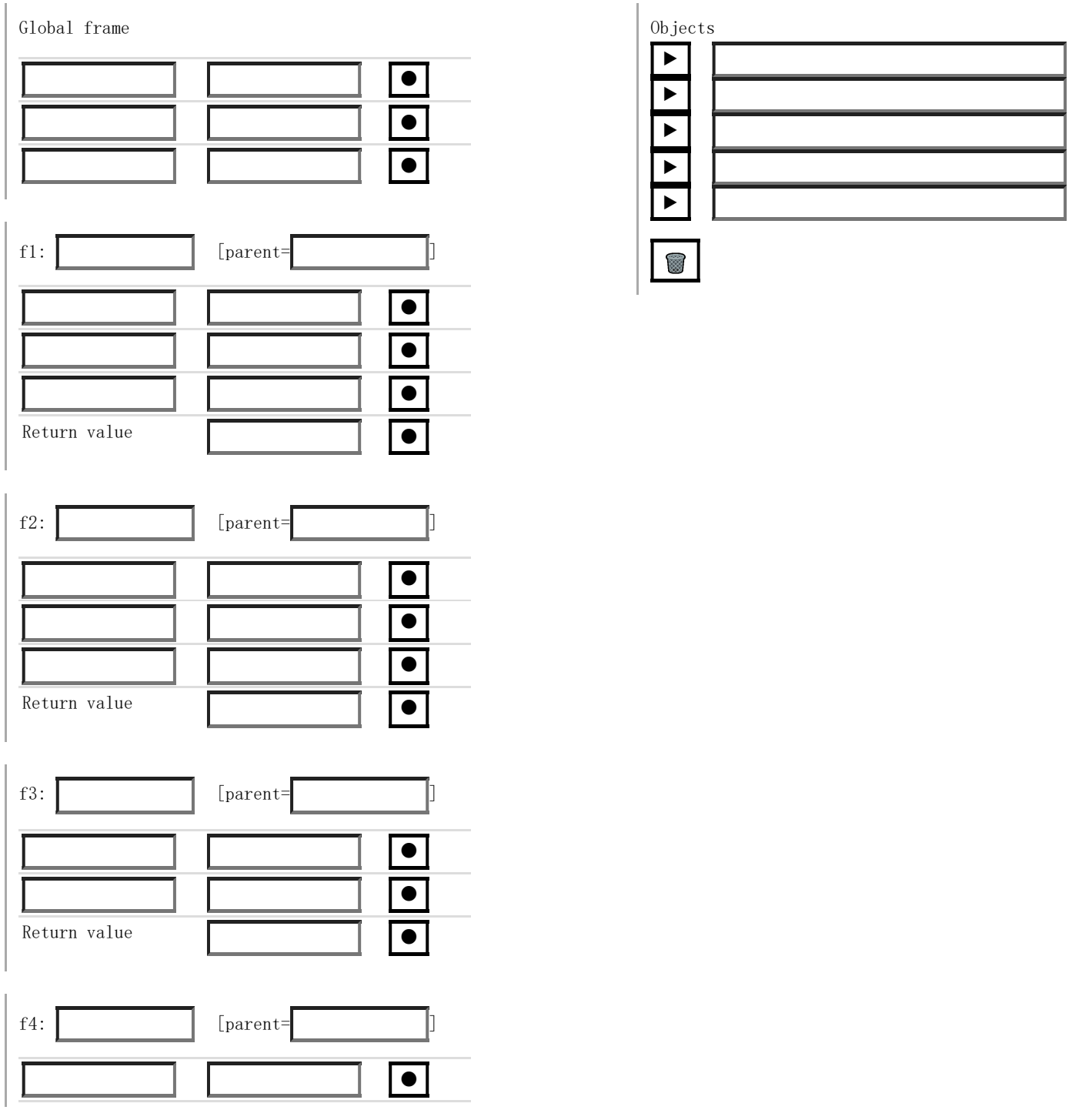
```
n = 7

def f(x):
    n = 8
    return x + 1

def g(x):
    n = 9
    def h():
        return x + 1
    return h

def f(f, x):
    return f(x + n)

f = f(g, n)
g = (lambda y: y())(f)
```


Global frame
f1: [parent=]
Return value
f2: [parent=]
Return value
f3: [parent=]
Return value
f4: [parent=]
Objects

Return value		

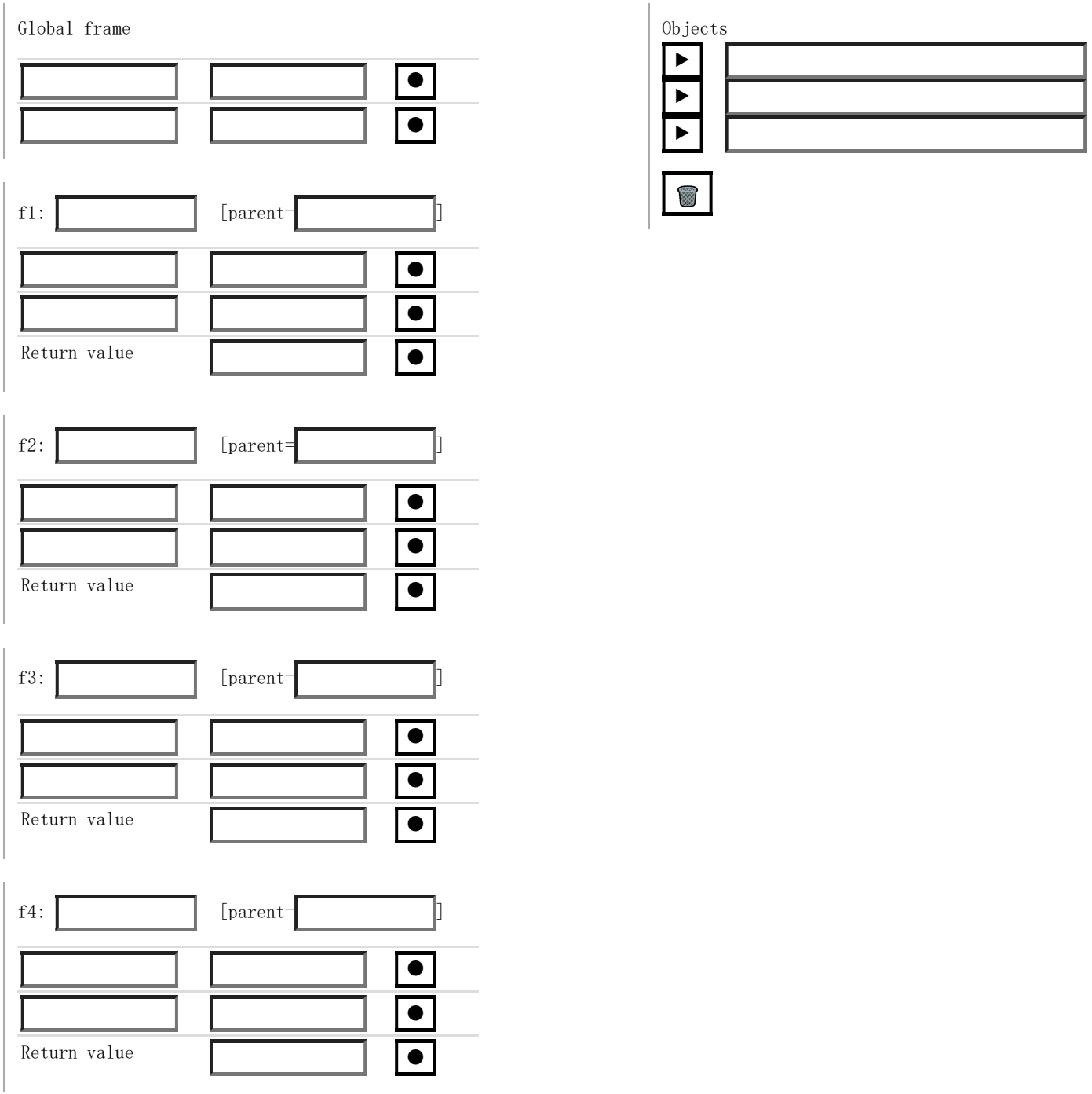
Q6: YY Diagram

The following question is more challenging than the previous ones. Nonetheless, it's a fun problem to try.

Draw the environment diagram that results from executing the code below.

> Tip: Using the `+` operator with two strings results in the second string being appended to the first. For example `"C" + "S"` concatenates the two strings into one string `"CS"`.

```
y = "y"
h = y
def y(y):
    h = "h"
    if y == h:
        return y + "i"
    y = lambda y: y(h)
    return lambda h: y(h)
y = y(y)(y)
```


Global frame
Objects
f1:
[parent=
]
Return value
f2:
[parent=
]
Return value
f3:
[parent=
]
Return value
f4:
[parent=
]
Return value

Video walkthrough (https://www.youtube.com/watch?v=MlRfJaGBeAY&feature=youtu.be)

Self Reference

Self-reference refers to a particular design of HOF, where a function eventually returns itself. In particular, a self-referencing function will not return a function **call**, but rather the function object itself. As an example, take a look at the `print_all` function:

```
def print_all(x):
    print(x)
    return print_all
```

Self-referencing functions will oftentimes employ helper functions that reference the outer function, such as the example below, `print_sums`.

```
def print_sums(n):
    print(n)
    def next_sum(k):
        return print_sums(n + k)
    return next_sum
```

A call to `print_sums` returns `next_sum`. A call to `next_sum` will return the result of calling `print_sums` which will, in turn, return another function `next_sum`. This type of pattern is common in self-referencing functions.

> A Note on Recursion: This differs from recursion because typically each new call returns a new function rather than a function call. We have not yet covered recursion so don't worry too much about what this means!

Q7: (Tutorial) Warm Up: Make Keeper Redux

These exercises are meant to help refresh your memory of topics covered in lecture and/or lab this week before tackling more challenging problems.

In this question, we will explore the execution of a self-reference function, `make_keeper_redux`, based off Question 2 (/~cs61a/sp21/disc/disc02/#q2), `make_keeper`. The function `make_keeper_redux` is similar to `make_keeper`, but now the returned function also returns **another function** with the same behavior. Feel free to paste and modify your code for `make_keeper` below.

(Hint: you only need to add one line to your `make_keeper` solution. What is currently missing from `make_keeper_redux`?)

```
        >>> k = make_keeper_redux(11)(multiple_of_4)
        4
        8
        >>> k = k(ends_with_1)
        1
        11
        >>> k
        <function do_keep>
        """
        # Paste your code for make_keeper here!

        def do_keep(func):
            i = 1
            while i <= n:
                if func(i):
                    print(i)
                i += 1
            return make_keeper_redux(n)

        return do_keep

```

Q8: Print Delayed

Write a function `print_delayed` that delays printing its argument until the next function call. `print_delayed` takes in an argument `x` and returns a new function `delay_print`. When `delay_print` is called, it prints out `x` and returns another `delay_print`.

```
        >>> f = print_delayed(1)
        >>> f = f(2)
        1
        >>> f = f(3)
        2
        >>> f = f(4)(5)
        3
        4
        >>> f("hi") # a function is returned
        5
        <function delay_print>
        """

        def delay_print(y):
            print(x)
            return print_delayed(y)

        return delay_print

```

Q9: (Tutorial) Print N

Write a function `print_n` that can take in an integer `n` and returns a repeatable print function that can print the next `n` parameters. After the nth parameter, it just prints "done".

```
    """
    >>> f = print_n(2)
    >>> f = f("hi")
    hi
    >>> f = f("hello")
    hello
    >>> f = f("bye")
    done
    >>> g = print_n(1)
    >>> g("first")("second")("third")
    first
    done
    done
    <function inner_print>
    """

    def inner_print(x):
        if n <= 0:
            print("done")
        else:
            print(x)
        return print_n(n-1)

    return inner_print

```