1 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
```

Questions

1.1 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
    """
```

1.2   **Tutorial:** Write a function similar to `keep_ints` like before, 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`.

```
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
    """
```

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.

```
x = 4
def add_num(x):
    return lambda y: x + y


add_two = add_num(2)
add_two(3)
```

Frames

Global frame
```
                    x   4
              add_num  •
              add_two  •
```

f1: add_num [parent=Global]
```
                    x   2
              Return
              value    •
```

f2: λ <line 3> [parent=f1]
```
                    y   3
              Return   5
              value
```

Objects

func add_num(x) [parent=Global]

func λ(y) <line 3> [parent=f1]

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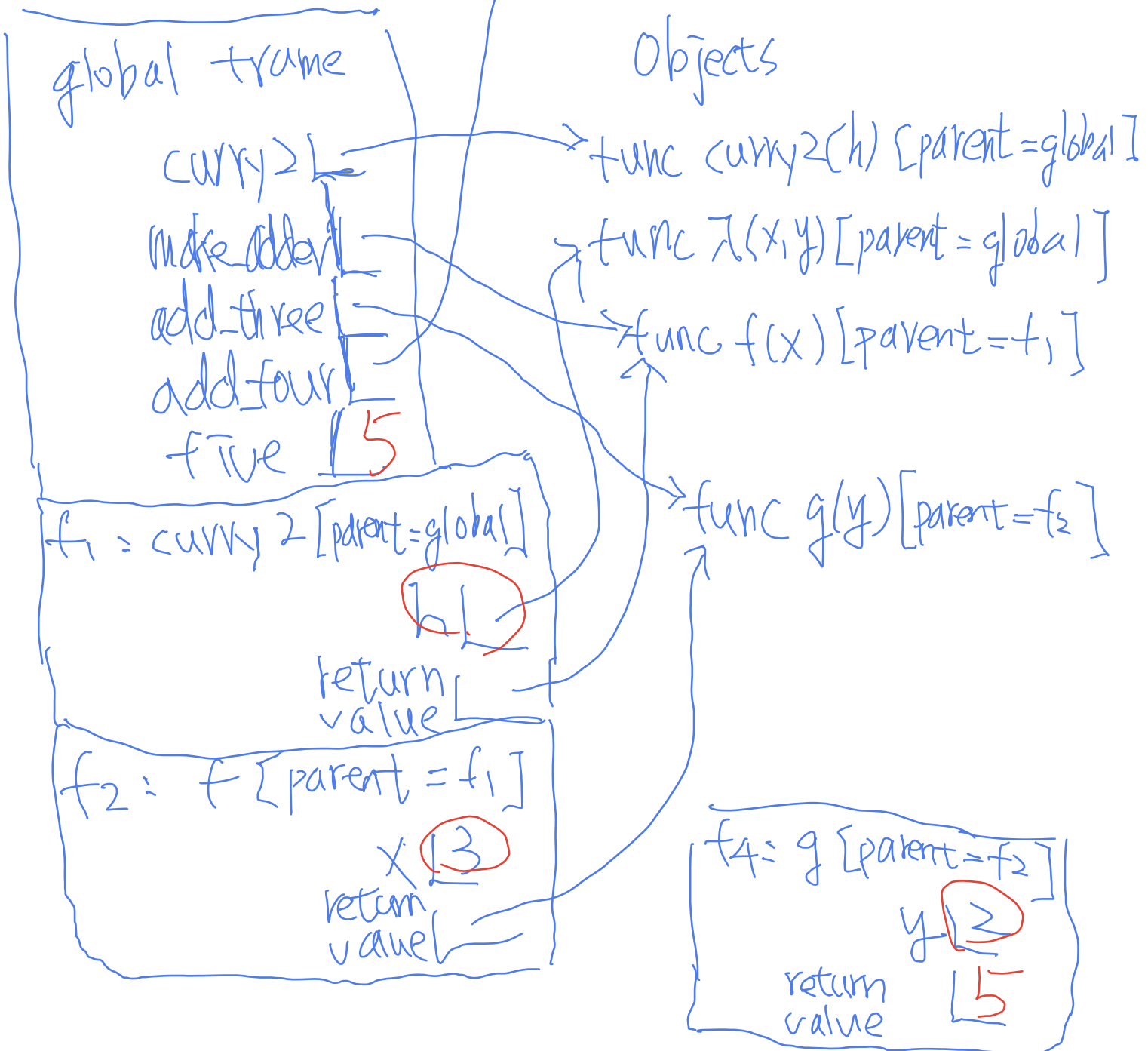
1.3 Draw the environment diagram that results from executing the code below.

```
1  def curry2(h):
2      def f(x):
3          def g(y):
4              return h(x, y)
5          return g
6      return f
7  make_adder = curry2(lambda x, y: x + y)
8  add_three = make_adder(3)
9  add_four = make_adder(4)
10 five = add_three(2)
```
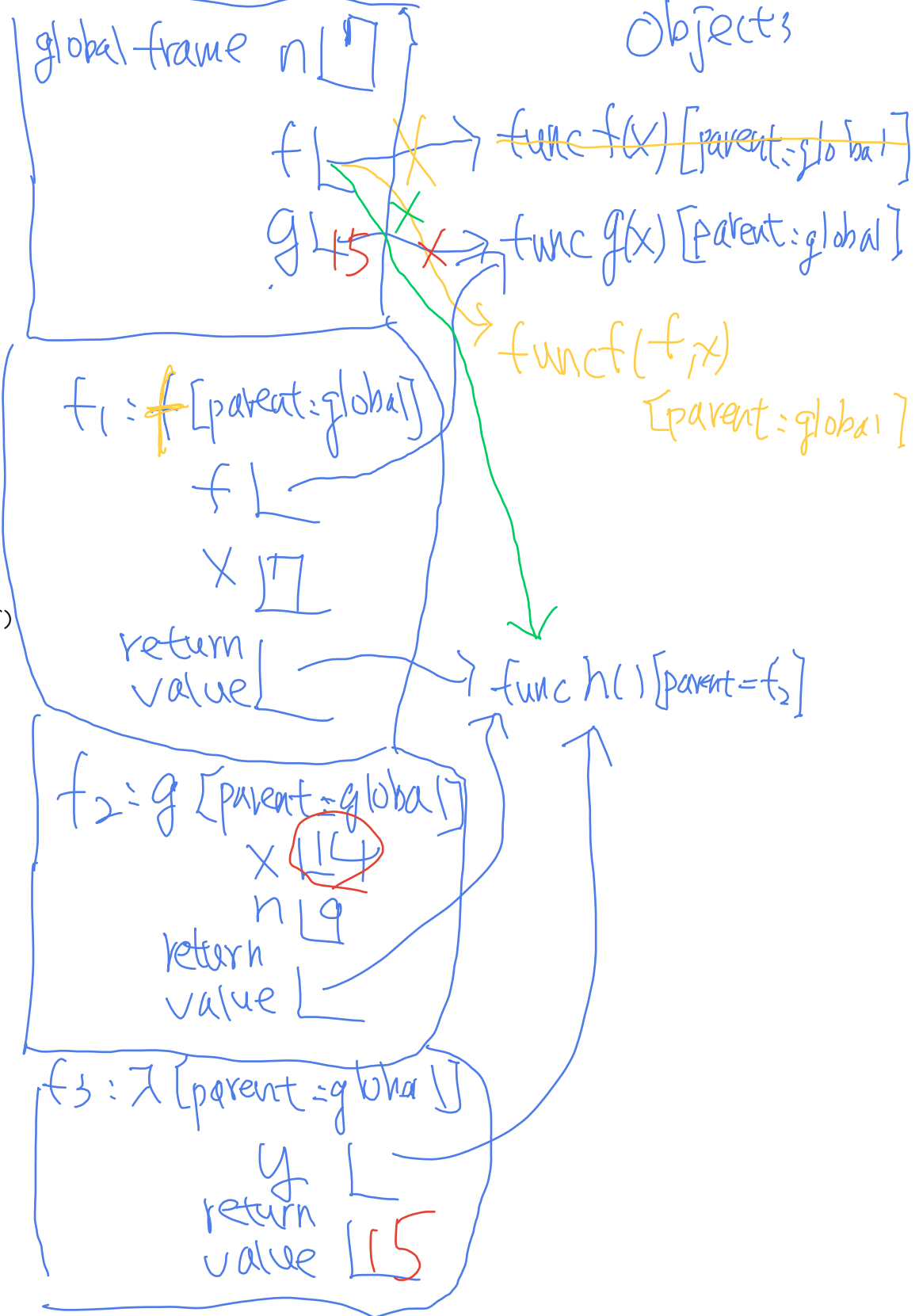
1.4   Write `curry2` as a lambda function.

1.5   **Tutorial:** Draw the environment diagram that results from executing the
code below.

```
1   n = 7
2
3   def f(x):
4       n = 8
5       return x + 1
6
7   def g(x):
8       n = 9
9       def h():
10          return x + 1
11      return h
12
13  def f(f, x):
14      return f(x + n)
15
16  f = f(g, n)
17  g = (lambda y: y())(f)
```

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

1.6   *The following question is more challenging than the previous ones. Nonethe-less, it's a fun problem to try.*
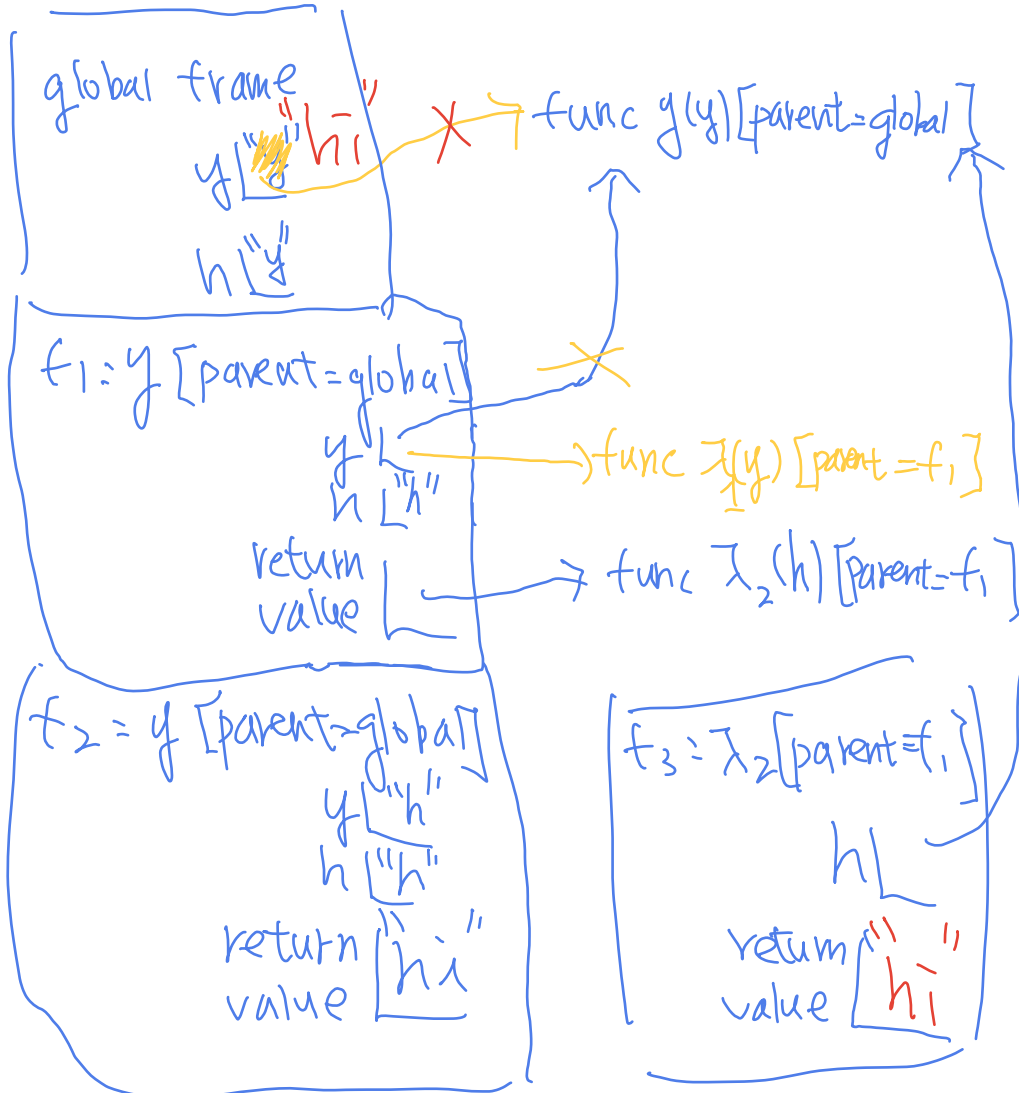
Draw the environment diagram that results from executing the code below.

Note that 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"

```
1  y = "y"
2  h = y
3  def y(y):
4      h = "h"
5      if y == h:
6          return y + "i"
7      y = lambda y: y(h)
8      return lambda h: y(h)
9  y = y(y)(y)
```

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 to the right.

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

Note that 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.

. **Small Note:** 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!

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


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

Questions

1.7 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.

```python
def print_delayed(x):
    """Return a new function. This new function, when called,
    will print out x and return another function with the same
    behavior.
    >>> f = print_delayed(1)
    >>> f = f(2)
    1
    >>> f = f(3)
    2
    >>> f = f(4)(5)
    3
    4
    >>> f("hi")
    5
    <function print_delayed> # a function is returned
    """
    def delay_print(y):

        _____
```

```
        return _____
    return delay_print
```

1.8 **Tutorial:** 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".

```
def print_n(n):
    """
    >>> 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 _____
            print("done")
        else:
            print(x)
        return _____
    return _____
```