WEEK 6 STUDIO

AGENDA

- Functional Expressionism
- Studio

FUNCTIONAL EXPRESSIONISM

TASK 4

```
const zero_r = f => x => x;
const one_r = f => x => f(zero_r, zero_r(f)(x));
const two_r = f => x => f(one_r, one_r(f)(x));
const three_r = f => x => f(two_r, two_r(f)(x));

const to_int = r => r((iter_count, x) => x + 1)(0);

const inc_r = r => f => x => f(r, r(f)(x));

const add_r = (r1, r2) =>
    r1((iter_count, x) => inc_r(x))(r2);

to_int(add_r(two_r, three_r));  // should return 5
```

TASK 4

```
const add_r = (r1, r2) =>
    f => x => r1(f)(r2(f)(x)); // returns 5 as well
```

Why is this code wrong?

- Because it does not follow the specification of the representation of a repeater.

TASK 5

```
const zero_r = f => x => x;
const one_r = f => x => f(zero_r, zero_r(f)(x));
const two_r = f => x => f(one_r, one_r(f)(x));
const three_r = f => x => f(two_r, two_r(f)(x));

const to_int = r => r((iter_count, x) => x + 1)(0);

const decrement_r = r =>
    r((iter_count, x) => iter_count)(zero_repeater);

to_int(decrement_r(three_r));  // should return 2
```

Note that O(1) is only possible with **normal-order evaluation** for this particular representation. More info here.

ATTENDANCE

STUDIO 6

QUESTION 1

Write the function `map` *using* `accumulate`*.*
Name your function `my_map`

- Approach: Understand what both accumulate and map does by looking at the definition of these two functions.

QUESTION 1

```
function my_map(f, lst) {
    return accumulate((x, ys) => pair(f(x), ys), null, lst);
}

const list1 = list(1, 2, 3, 4, 5, 6);
my_map(x => x + 1, list1);
```

QUESTION 2

- *Write a function called* `remove_duplicates` *that*
  - *Takes in a list as its only argument*
  - *Returns a list with duplicate elements removed*
  - *Order of the elements in the returned list does not matter*
  - ***Use filter in your function***

- Approach: Take the head, then remove all other occurences of the head from the tail of the list. Pair the two results together. Use recursion and wishful thinking

QUESTION 2

```
function remove_duplicates(lst) {
    return is_null(lst)
        ? null
        : pair(head(lst),
               remove_duplicates(
                   filter(x => !equal(head(lst), x),
                          tail(lst))));
}
function remove_duplicates2(lst) {
    return is_null(lst)
        ? null
        : pair(head(lst),
           filter(x => !equal(head(lst), x),
                  remove_duplicates2(tail(lst))));
}
```

HOW DO THEIR APPROACHES DIFFER?

- First one is intuitive. First take the head. Then, from the filtered list which does not contain any other occurences of the head, remove the other duplicates present, and pair these two results together.
- Second one is different, it says first take the head, then from the list which already has its duplicate elements removed, filter all occurences of the head.
- First one filters from the start of the list till the end, but the second filters from the end of the list till the start.

QUESTION 3

- *Write a function called* `remove_duplicates` *that*
  - *Takes in a list as its only argument*
  - *Returns a list with duplicate elements removed*
  - *Order of the elements in the returned list does not matter*
  - ***Use accumulate in your function***

- Approach: Same thing. Understand what both functions achieve and how they behave, then work from there.

QUESTION 3

```
function remove_duplicates(lst) {
    return accumulate(
        (ele, result) => is_null(member(ele, result))
            ? pair(ele, result)
            : result,
        null,
        lst);
}
```

HOW DOES IT WORK?

- Start from the last element and build the unique list all the way to the start. If element is already in the list, just return the list, else pair the element with the list.
- Can you use filter instead of member? Yes, but it'll be quite inefficient!

QUESTION 4

Our friend Louis Reasoner has a pocket full of change. He wants to buy a snack that will cost him x cents, and he wants to know all the ways in which he can use his change to make up that amount.

Write a function which takes as parameters the amount x and a list of all the coins Louis has in his pocket, and returns a list of lists, such that each sub-list of the result contains a valid combination to make up x. A combination may appear more than once, since it may be using different coins of the same denomination.

APPROACH

- What does `makeup_amount` return me? It returns the number of ways I can make change for amount `x` with the list of coins that I have.
    - It returns a list of the number of ways to give change, which is a list of lists.
- Think about the different cases that make up all of the number of ways to give change.
    - Case 1: I use the coin. So this coin will be part of my solution. The implication for my recursive call is this: the amount will decrease, and the list of coins will also decrease, since I have used the coin.
    - Case 2: I don't use the coin. So this coin is not part of my solution. The implication is this: the amount will not decrease, but the list of coins will still decrease, since I don't want this coin to be part of my solution.

QUESTION 4

```
function makeup_amount(x, coins) {
    if (x === 0) {
        return list(null);
    } else if (x < 0 || is_null(coins)) {
        return null;
    } else {
        // Combinations that do not use the head coin.
        const wo_head = makeup_amount(x, tail(coins));

        // Combinations that make use of the head coin.
        const sub = makeup_amount(x - head(coins), tail(coins));
        const w_head = map(c => pair(head(coins), c), sub);
        return append(wo_head, w_head);
    }
}
```

SOME QUESTIONS TO ASK YOURSELF

- What does `sub` represent?
  - It represents a number of ways I can make change for (`x - head(coins)`) amount of money with (`tail(coins)`) coins. i.e. a smaller version of the original problem.
- Why still need to tail(coins) when I choose to use the coin?
  - Because coins are finite, if I use it, it is gone.
- Why do you need to `map` the `head` in?
  - Because `sub` represents only the smaller version of the problem, and not the actual problem that we are solving for. Try working this out to understand what it means.
- Why do the base cases return different results?
  - Think of it in terms of what the question is asking for, and what kind of result `makeup_amount` returns.
- Question: How many of the same solutions do appear in the list?
  - Simplified answer: n choose k.

QUESTION 5

The function `accumulate_n` *is similar to* `accumulate` *except that it takes as its third argument a list of lists, which are all assumed to have the same number of elements. It applies the designated accumulation function to combine all the first elements of the sequences, all the second elements of the sequences, and so on, and returns a list of the results.*

APPROACH

- Same as before, think of it using recursion and wishful thinking.
- What is the base case?
  - `accumulate_n` takes in a list of lists. So the base case is if each individual list is an empty list, which is null.
- What is the smaller version of the problem?
  - The a list containing the tail of each individual list in the original list of lists.
- How to piece them together?
  - Accumulate the head of each individual list, then call `accumulate_n` on the smaller problem. Finally, do something to combine these two pieces together.

QUESTION 5

```
function accumulate_n(op, init, seq) {
    return is_null(head(seq))
        ? null
        : pair(
            accumulate(
                (ele, result) => op(head(ele), result),
                init,
                seq),
            accumulate_n(op, init, map(ele => tail(ele), seq)));
}

const s = list(list(1, 2, 3), list(4, 5, 6),
               list(7, 8, 9), list(10, 11, 12));
accumulate_n((x, y) => x + y, 0, s);
```

Can we do better?

QUESTION 5

```
function accumulate_n2(op, init, seq) {
    return is_null(head(seq))
        ? null
        : pair(
            accumulate(op, init, map(head, seq)),
            accumulate_n(op, init, map(tail, seq)));
}

const s = list(list(1, 2, 3), list(4, 5, 6),
               list(7, 8, 9), list(10, 11, 12));
accumulate_n2((x, y) => x + y, 0, s);
```

HOW DO THEY DIFFER?

- First observation is trivial.
  - `map(tail, seq)` removes the redundancy of doing `ele => tail(ele)`.
- Second observation in the `accumulate` function call, however, is not trivial.
- For the first version, you are taking the `seq` and then applying the function `op` on the `head` of each element, which is a list, in `seq`.
- For the second version, you first mapping `head` to `seq`, which gives a list containing all the `head` elements of each individual list in `seq` i.e. `list(1, 4, 7, 10)`.

QUESTION 6

Write a function called accumulate_tree that behaves like accumulate but can also work on trees.

Definition: A tree of a certain data type is a list whose elements are of that type, or trees of that type. Recursion!

- Approach: Look at how a tree is defined and represented.
- As above, use the idea of wishful thinking to help out with your recursion!

QUESTION 6

```
function accumulate_tree(op, init, tree) {
    return accumulate(
        (ele, result) => !is_list(ele)
                ? op(ele, result)
                : op(accumulate_tree(op, init, ele), result),
        init,
        tree);
}

accumulate_tree(
    (x, y) => x + y,
    0,
    list(1, 2, list(3, 4), list(5, list(6, 7))));
```

- How does it work?
    - Check each element of the tree. If the element is a tree, then call `accumulate_tree` on the element to get a result. Else, just apply the function `op` to the element and the accumulated result, and continue accumulating.

QUESTION 7 (EXTRA, BUT IMPORTANT!)

In this question, lists represent *sets*. Each element of the set appears exactly once in its list representation, and the order does not matter. So the list `list(1, 2, 3)` represents the same set as the list `list(3, 2, 1)`.

In this question, you are supposed to compute all subsets of a give set. Your function `subsets` takes as argument a list, representing the given set, and needs to return a list of lists, each representing a unique subset of the given set.

```
function subsets(xs) {
    ...
}
```

Example call:

```
subsets(list(1, 2, 3));
// Result: list(list(),
//              list(1), list(2), list(3),
//              list(1,2), list(1,3), list(2,3),
//              list(1,2,3))
```

APPROACH

- How to approach? Like `makeup_amount` actually, think of how to break the problems down using recursion and wishful thinking.
- You either want the element to be in your set, or not.
- Note the base case. What does `subsets` return you?

QUESTION 7 (EXTRA, BUT IMPORTANT!)

```
// SOLUTION 1:
function subsets(xs) {
    if (is_null(xs)) {
        return list(null);
    } else {
        const subsets_rest = subsets(tail(xs));
        const x = head(xs);
        const has_x = map(s =>pair(x, s), subsets_rest);
        return append(subsets_rest, has_x);
    }
}
```

QUESTION 7 (EXTRA, BUT IMPORTANT!)

```
// SOLUTION 2:
function subsets(xs) {
    return accumulate(
        (x, ss) => append(ss, map(s => pair(x, s), ss)),
        list(null),
        xs);
}
```

HOW DO THEY DIFFER?

- First solution hinges on the idea of wishful thinking very heavily, just like `makeup_amount`.
- Second solution builds the list from the bottom.
    - So I look at an element of the list `xs`. If I want it, the I map it to every single subset that I had from the previous result, `ss`. Then, I just append it together with the list of subsets without the current element, which is also `ss`. This gives the current result of all the subsets of the elements we have already looked at. Finally, repeat this process with the rest of the elements.
- The second solution follows the idea of want or don't want more explicitly.
- Try both of them out with only 3 elements to see the process of want or don't want!

QUESTION 8 (EXTRA, BUT IMPORTANT!)

A *permutation* of a list s is a list with the same elements as s, but in a possibly different order. For example, the list `list(3,1,2)` is a permutation of `list(1,2,3)`. Write a function `permutations` that takes a list s as argument and returns a list of all permutations of s.

```
function permutations(s) {
    ...
}
```

Example call:

```
permutations(list(1, 2, 3));
// Result: list(list(1,2,3), list(1,3,2),
//              list(2,1,3), list(2,3,1),
//              list(3,1,2), list(3,2,1))
```

- Approach: Try it out yourself!

QUESTION 8 (EXTRA, BUT IMPORTANT!)

```
function permutations(s) {
    return is_null(s)
        ? list(null)
        : accumulate(append, null,
            map(x => map(p => pair(x, p),
                         permutations(remove(x, s))),
                s));
}
```