14th Sept 2020

Studio 6
List Processing

CS1101S AY20/21 SEM 1

Studio 03A

Chen Xihao
Year 2 Computer Science

chenxihao@u.nus.edu
@BooleanValue

Studio 6

Agenda

- Recap
  - Box and pointer diagrams
  - Trees
  - List processing
  - Tree processing
- Studio Sheet
- In-class Studio Sheet

Recap - BoxeRs and PointeRs

Recap

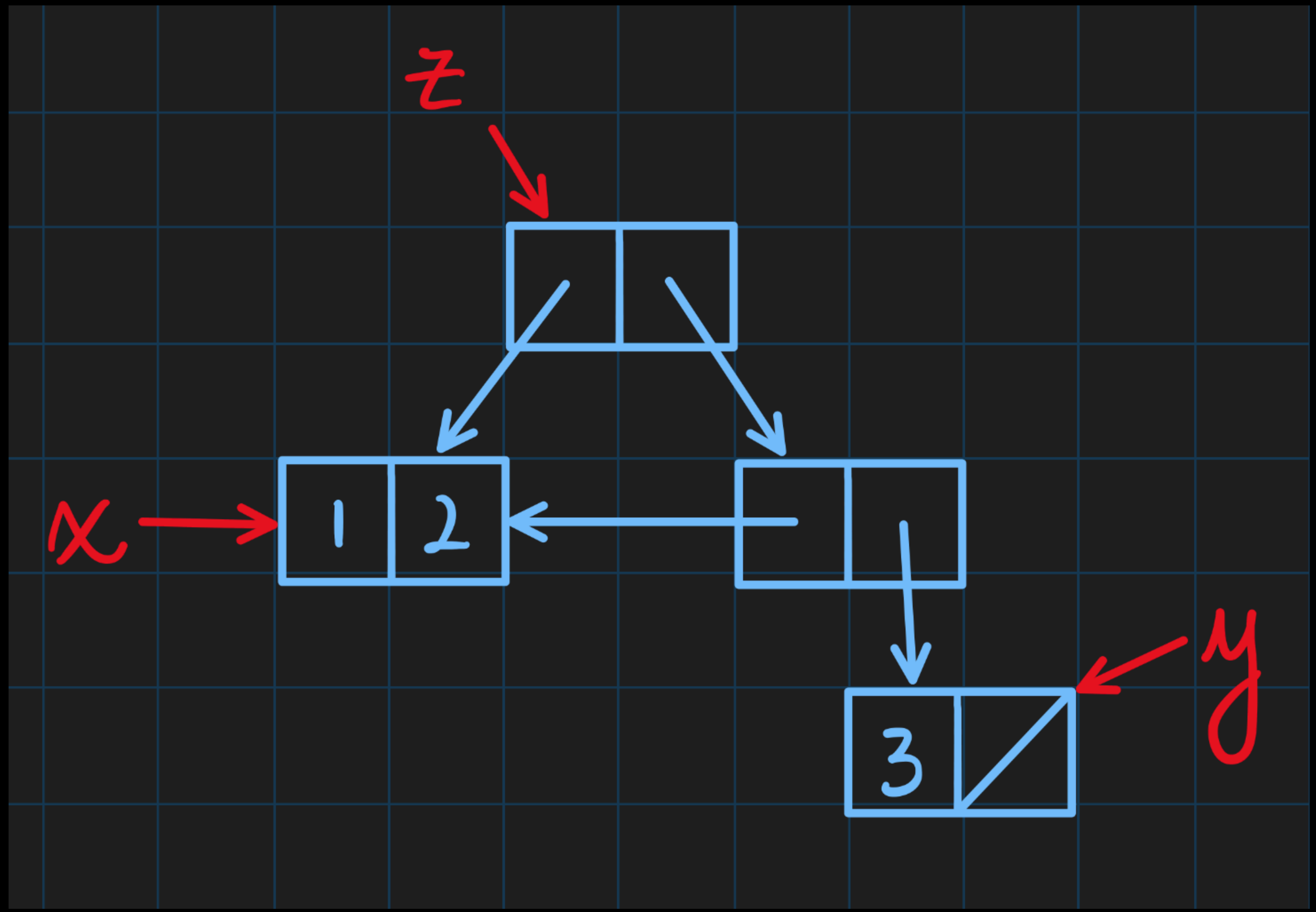
Box and Pointer Diagrams

- Let's start with an exercise:
  - Draw the box-and-pointer diagram for this programme:
    - `const x = pair(1, 2);`
    - `const y = list(3);`
    - `const z = pair(x, pair(x, y));`

Recap

Box and Pointer Diagrams

- `const x = pair(1, 2);`
- `const y = list(3);`
- `const z = pair(x, pair(x, y));`



Recap

Box and Pointer Diagrams

- When do we draw arrows? When do we not?
  - Primitives: draw in the box
  - Data structures: draw an arrow

Recap

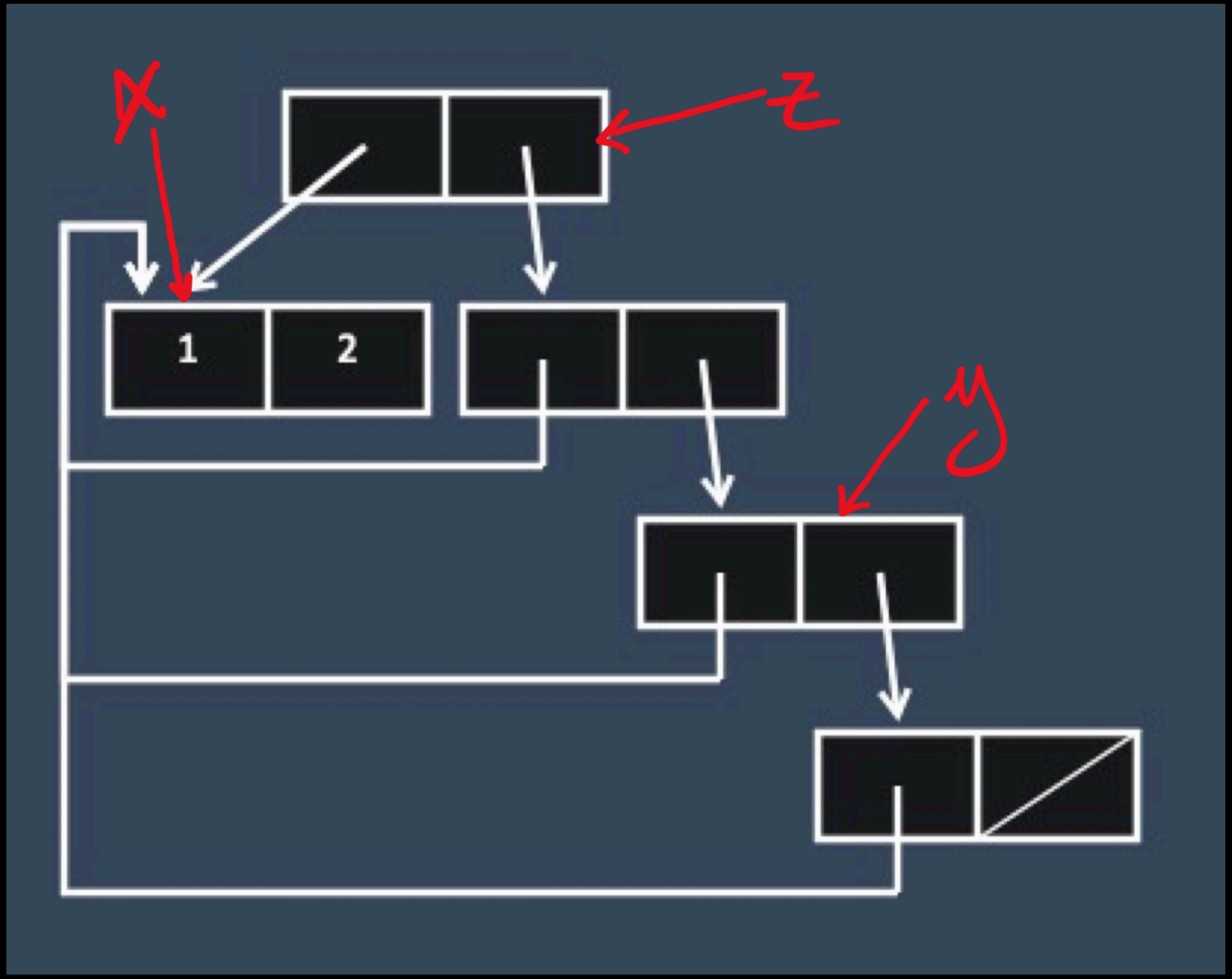
Box and Pointer Diagrams

- Consider this new programme:
  - `const x = pair(1, 2);`
  - `const y = list(x, x);`
  - `const z = pair(x, pair(x, y));`
- What are the results of evaluating the following:
  - `x === pair(1, 2);            // returns ?`
  - `head(tail(z)) === x;         // returns ?`
  - `head(tail(tail(z))) === x;  // returns ?`

Recap

Box and Pointer Diagrams

- Consider this new programme:
  - `const x = pair(1, 2);`
  - `const y = list(x, x);`
  - `const z = pair(x, pair(x, y));`
- What are the results of evaluating the following:
  - `x === pair(1, 2);                // returns false`
  - `head(tail(z)) === x;             // returns true`
  - `head(tail(tail(z))) === x;   // returns true`



Recap

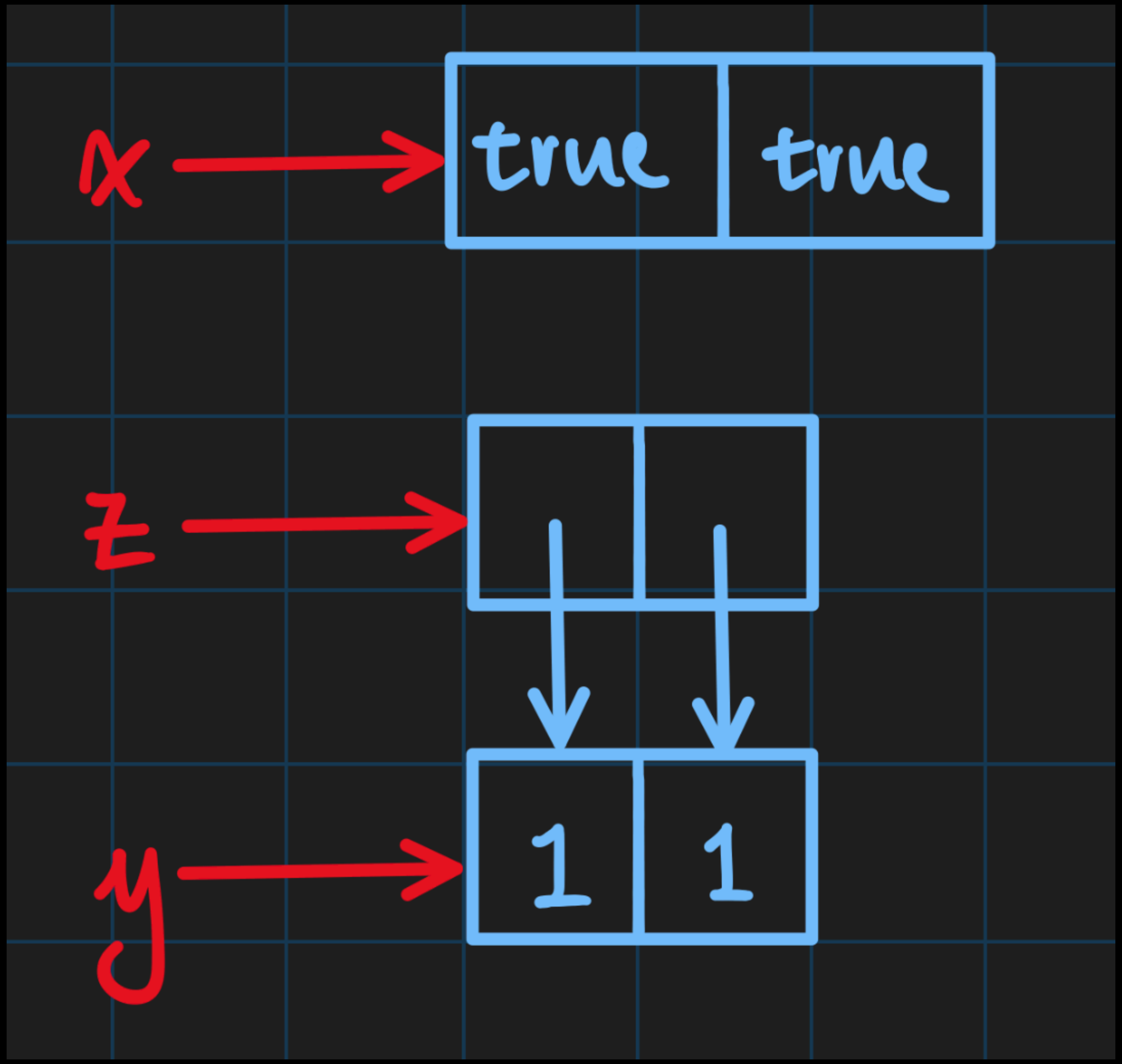
Box and Pointer Diagrams

- We create new data structures when using:
  - `pair`
  - `list`
- Identities:
  - We <u>duplicate</u> the values in data structures if they are primitives
  - We <u>re-direct</u> the pointer in data structures if they are non-primitives

Recap

Box and Pointer Diagrams

- Example:
  - Duplicated values:
    - `const x = pair(true, true);`
  - Redirected pointers:
    - `const y = pair(1, 1);`
    - `const z = pair(y, y);`



Recap

Box and Pointer Diagrams

- Side note:
  - Named data structures:
    - Remember to show the write the name and point to the structure
  - Not shown in SourceAcademy but good practice

Recap - Trees

Recap

Trees

- New data structure!
- Definition:
    - A tree (of some data type) is a ***list***
        - whose elements are of this data type,
        - or trees of such data types.

Recap

Trees

- For example:
  - A tree of <u>numbers</u> is a list whose elements are <u>numbers</u>, or trees of <u>numbers</u>
- Recall: lists are either null or pairs with tails as lists

Recap

Trees

- For example:
  - A tree of <u>numbers</u> is a list whose elements are <u>numbers</u>, or trees of <u>numbers</u>
  - `list(1);  // is a tree of numbers`

Recap
Trees

- For example:
  - A tree of numbers is a list whose elements are numbers, or trees of numbers.
  - `list(list(1));  // is a tree of numbers`
    - `// since list(1) is a tree of numbers`
  - `list(list(1, 2, 3), 1, list(1, 2, 3));  // is a tree of numbers`
    - `// can be a mixture of trees and single numbers`

Recap
Trees

- Exercise: Discuss whether these are trees (of numbers)
  - `list(1, 2, 3);`
  - `list(null, 1);`
  - `list(1, list(2, 3), pair(4, 5));`

Recap

Trees

- Exercise: Discuss whether these are trees (of numbers)
  - `list(1, 2, 3);  // yes`
  - `list(null, 1);  // yes`
    - `// ‘null’ can be considered as a list of numbers`
    - `// so ‘null’ is technically also a tree of numbers`
  - `list(1, list(2, 3), pair(4, 5));  // no`
    - `// not a tree of numbers`
    - `// also not even a list of numbers…`

Recap
Trees

- Visualisation of a binary tree:
    - Each element has:
        - its own value
        - left subtree (some other tree)
        - right subtree (some other other tree)

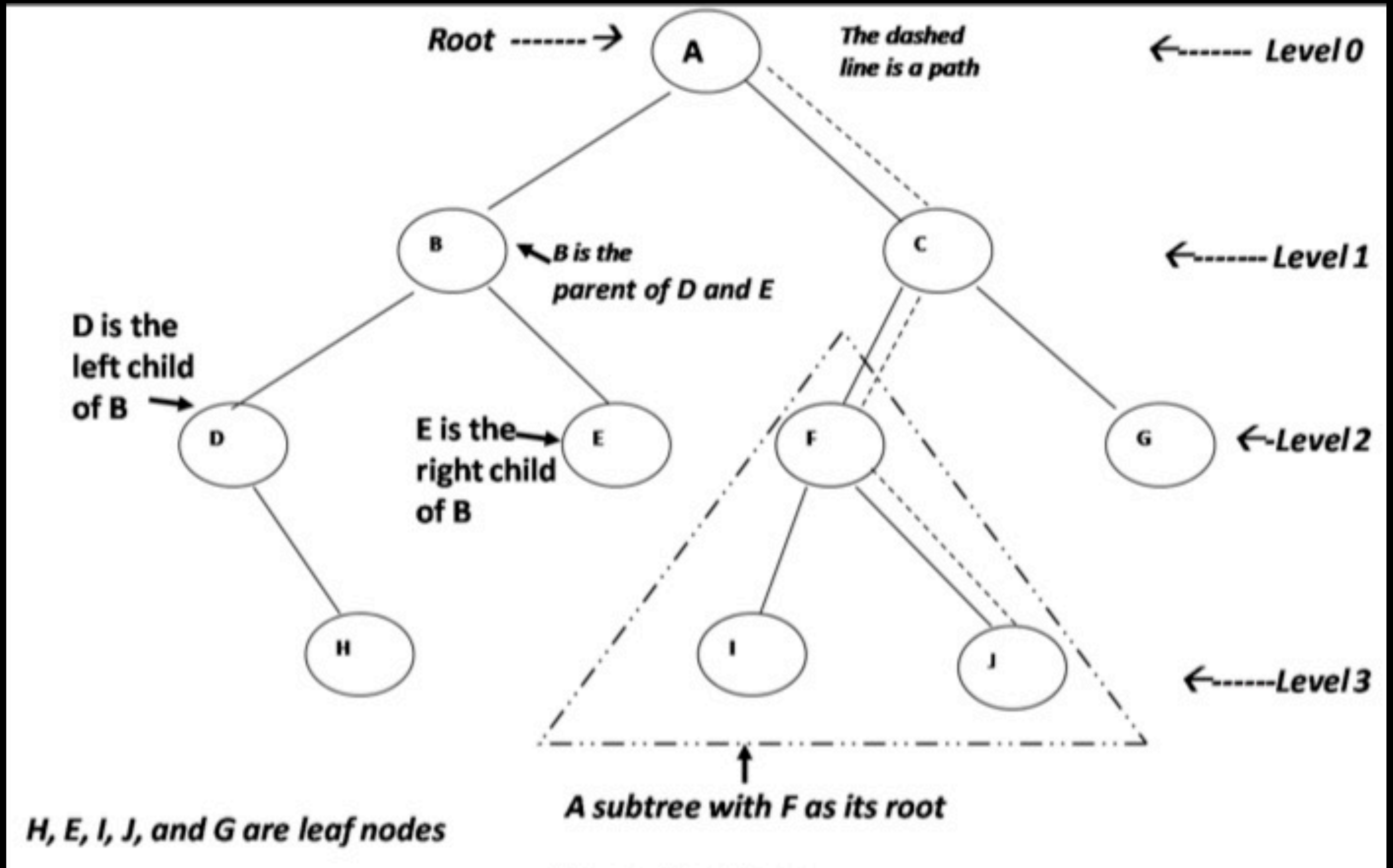


Figure: Tree Terms

Recap

Trees

- Why is the root at the top?
  - The computer scientist who designed this convention probably has never seen an actual tree before…

Recap
Trees - Summary

- Trees: another data structure
  - Recursive in nature

Recap - List Processing

Recap

List Processing

- LISTS library
  - `map, filter, accumulate`
  - Three of the most important functions in programming
  - Why important to you:
    - Exams will ask you to solve stuff using only these functions.

Recap

List Processing - Map

- Map: ‘to assign in a mathematical or exact correspondence’
  - `map(f, xs)`
  - Not the same as 地图 pls
- Usage: applies the function `f` to all elements inside `xs`
  - `original xs: list(a1, a2, a3, … an)`
  - `mapped list: list(f(a1), f(a2), f(a3), …, f(an))`

Recap

List Processing - Map

- Example:
  - `map(x => x + 1, list(1, 2, 3))`
  - `// returns list(2, 3, 4)`

Recap

List Processing - Map

- Note:
  - Map always returns a list, of the same length!
    - Alters each element, but not the ‘structure’ of the list
  - The function `f` does NOT need to return the same data type
    - Example:
      - `map(x => true, list(1, 2, 3));`
      - `// returns list(true, true, true)`

Recap

List Processing - Map

- Important uses:
    - Pairing each element with some other element
        - `map(x => pair(1, x), list(1, 2, 3))`
        - `// returns list(pair(1, 1), pair(1, 2), pair(1, 3))`
    - Extension: pre-pending items to each list
        - ```
          map(
            ys => pair(a, ys),  // pre-pending `a` (aka pairing `a` with current list)
            list(xs1, xs2, xs3, …, xsn)  // list of lists
          )
          ```
          ```

Recap

List Processing - Map

- Important uses:
  - Copying lists
    - `map(x => x, list(1, 2, 3))`
    - `// returns list(1, 2, 3), a copy of the original list`
  - Reminder: duplication rule applies
    - Elements are only duplicated if they are primitives
    - Data structures are not duplicated (pointers are used)

Recap

List Processing - Filter

- Filter: ‘to remove by means of a filter’
  - `filter(condition, xs)`
  - Removes elements that do not pass the boolean condition
  - Allows elements that pass the condition to be in the resulting list
- Usage:
  - `filter(x => is_even(x), list(1, 2, 3));`
  - `// returns list(2);`

Recap

List Processing - Filter

- Note:
  - Filter, like map, returns a list
  - Filter does not alter the elements in the list
  - Filter may return a shorter list, or a list of the same length
  - Condition (officially `pred`) MUST return a boolean

Recap

List Processing - Accumulate

- Accumulate: ‘to gather or pile up especially little by little’
  - `accumulate(f, initial, xs)`
- Parameters:
  - `f`: a binary operator
    - takes in two arguments, `x` and `y`
    - return value must be the same type as `y`
  - `initial`: the initial value of accumulation, same type as `y`
  - `xs`: a list with elements of the same type as `x`

Recap

List Processing - Accumulate

- For the sake of understanding, let’s assume that everything are of the same type.
    - For example:
        - \`f\` operates on numbers: (number, number) -> number
        - \`initial\` is some number
        - \`xs\` is a list of numbers

Recap

List Processing - Accumulate

```
const mult = (x, y) => x * y;
```

- Trace:
  - `accumulate(mult, 1, list(1, 2, 3));`
  - `> mult(1, accum(mult, 1, list(2, 3)))`
  - `> mult(1, mult(2, accum(mult, 1, list(3))))`
  - `> mult(1, mult(2, mult(3, accum(mult, 1, list()))))`
  - `> mult(1, mult(2, mult(3, 1)))`
  - `> mult(1, mult(2, 3))`
  - `> mult(1, 6)`
  - `>> 6`

Recap

List Processing - Accumulate

- In the example, we evaluated `mult(3, 1)` first
  - Start by operating on the last element with the initial value
- In general, if we have `accumulate(op, i, list(a1, a2, a3, …, an))`, we can visualise it like this:
  - `op(a1, op(a2, op(a3, … op(an, i) ) )`
  - Always operate on the last element in the list first
    - Curious qn: how do we detect this?

Recap

List Processing - Accumulate

- Remember we made the assumption about having the same types?
- We can have different data types in accumulate
  - `const collect_strs = (n, str) => stringify(n) + str;`
  - `accumulate(collect_strs, “”, list(1, 2, 3));`
    - `// returns the string “123”`

Recap

List Processing - Accumulate

- Let’s take a closer look:
  - `const collect_strs = (n, str) => stringify(n) + str;`
    - `// (number, string) -> string`
  - `accumulate(collect_strs, “”, list(1, 2, 3));  // returns the string “123”`
    - `// (function, string, list of numbers) -> string`
- Match the types!

Recap

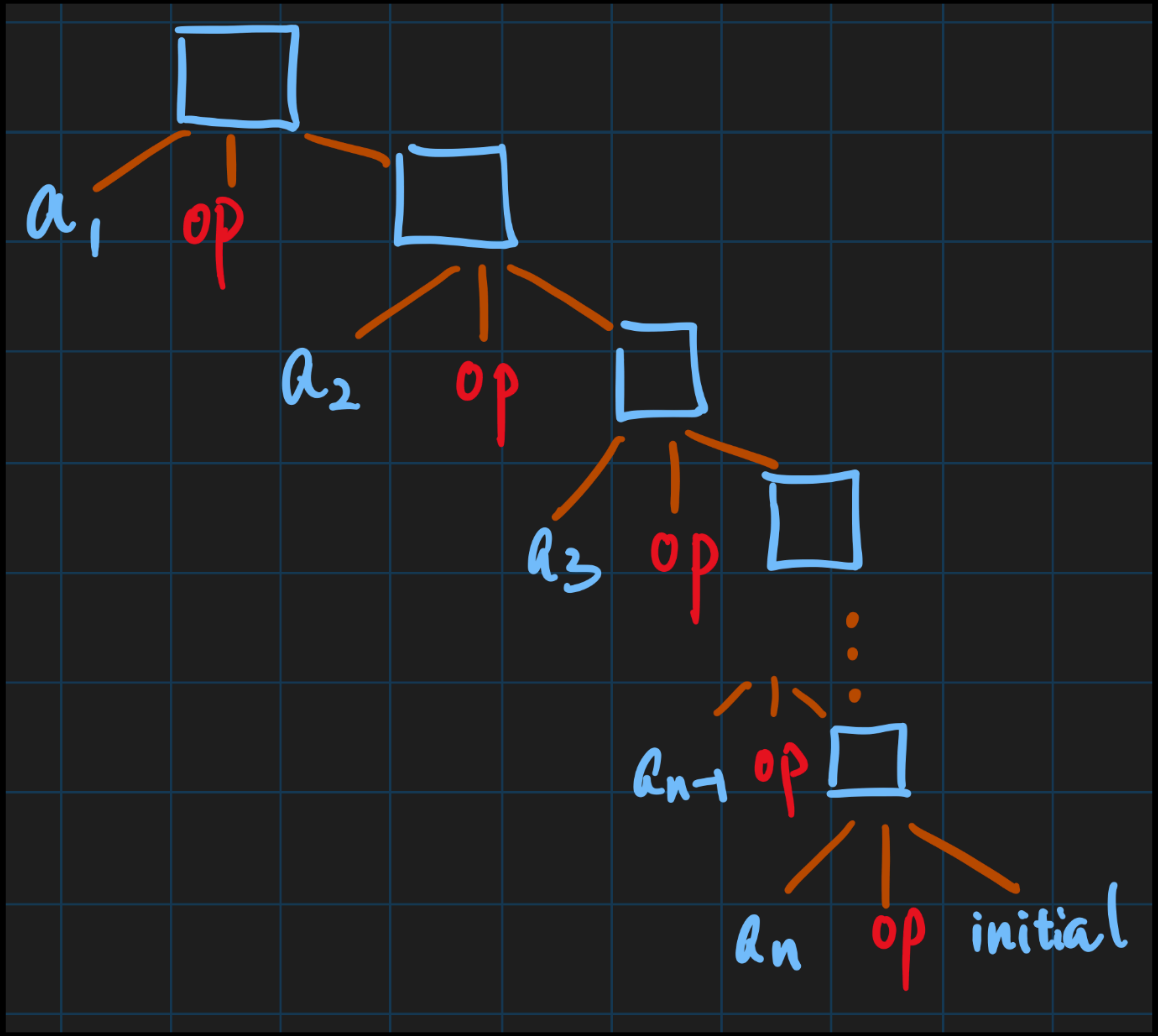
List Processing - Accumulate

- Let's take a closer look:
  - `const collect_strs = (n, str) => stringify(n) + str;`
    - `// (number, string) -> string`
  - `accumulate(collect_strs, "", list(1, 2, 3));  // returns the string "123"`
    - `// (function, string, list of numbers) -> string`
- Match the types!

Recap

List Processing - Accumulate

- Another representation using the abstract syntax tree
    - or at least i think it is
- Notice how all the `op` calls are on towards the right fo the tree
    - Accumulate is also called the ‘right-fold’ operation (just for your curiousity)



Recap

List Processing - Accumulate

- Important usage: appending multiple lists together
- If we lists xs_1, xs_2, xs_3, …, xs_n, and we want to append them together:
  - `append(xs_1, append(xs_2, append(xs_3, …, append(xs_n, null))));`
  - Manual and tedious and cumbersome and bad
- Using accumulate:
  - `accumulate(append, null, list(xs_1, xs_2, xs_3, …, xs_n));`
  - Try it out!

Recap

List Processing - Summary

- Map, Filter and Accumulate
  - Essential functions for data structures

Any questions?

End of Recap

Studio Sheet

Recap

Question 1

- Write `map` using `accumulate`
  - ```
    function my_map(f, xs) {
         // your answer here
     }
    ```
  - Recall: `accumulate(bin_op, initial, xs)`
    - (renamed to `bin_op` for clarity)
    ```

Recap

`my_map(f, xs)`

Question 1

- Write `map` using `accumulate`
  - Observe that:
    - `map` applies `f` to every element`
      - `` `f`: (typeA) -> typeB ``
    - `accumulate` applies `bin_op` to every pair of elements
      - `` `bin_op`: (typeA, typeB) -> typeB ``

Recap
Question 1

`my_map(f, xs)`

- Write \`map\` using \`accumulate\`
  - Let’s start by duplicating the list
    - `Recall: map(x => x, xs)`
  - Similarly:
    - `accumulate((x, ys) => pair(x, ys), null, xs)`
  - Now we have a duplicated list

Recap

Question 1

`my_map(f, xs)`

- Write `map` using `accumulate`
  - `accumulate((x, ys) => pair(x, ys), null, xs)`
    - Current element: `x`
    - Just apply `f` to it!
  - `> accumulate((x, ys) => pair(f(x), ys), null, xs)`

Recap

Question 2

- Write `remove_duplicates` using `filter`
  - ```
    function remove_duplicates(xs) {
        // your answer here
    }
    ```
    ```

Recap

Question 2

`remove_duplicates(xs)`

- Write `remove_duplicates` using `filter`
  - Naively:
    - Keep track of a list without duplicates `ys`
    - Check the current element `x`
      - If `x` is in `ys`: go to the next element
      - If `x` is not in `ys`: add `x` into `ys`

Recap
Question 2

- Write `remove_duplicates` using `filter`
    - We need something to check if an element is in some list
        - `member(v, xs) -> {list}`
        - Returns the first sublist whose head is identical to v, returns null if element does not occur in list. O(n) time
        - Example:
            - `member(1, list(0, 1, 2, 3));  // returns: list(1, 2, 3)`

Recap

Question 2

remove_duplicates(xs)

- Write `remove_duplicates` using `filter`
    - Now we are ready to remove duplicates
    - Runtime: O(n^2)

```
function remove_duplicates(xs) {
    // xs: list of items not went thru yet
    // ys: list that we have processed
    function helper(xs, ys) {
        if (is_null(xs)) {
            return ys;
        } else {
            const x = head(xs);

            if (is_null(member(x, ys))) {
                // x is not found in ys
                return helper(tail(xs), pair(x, ys));
            } else {
                // x is found in ys (there will be a duplicate)
                return helper(tail(xs), ys);
            }
        }
    }
    return helper(xs, null);
}
```

Recap
Question 2

`remove_duplicates(xs)`

- Write `remove_duplicates` using `filter`
  - Wait… we didn't use filter!
  - Recall on recursion:
    - What can we do with the current element `x`?
    - Remove all duplicates in the unprocessed list!
    - Gives us a list that doesn't have the current element
    - Now we can move on to the next element

Recap

Question 2

`remove_duplicates(xs)`

- Write `remove_duplicates` using `filter`
  - High level idea:
    - Current list: list(1, 2, 3, 4, 3, 1)
    - Current element: 1, unprocessed list: list(2, 3, 4, 3, 1)
    - Remove `1` from the unprocessed list
      - `filter(1, list(2, 3, 4, 1))  // returns list(2, 3, 4, 3)`
    - Continue with the unprocessed list (list(2, 3, 4, 3))

Recap

Question 2

- Write `remove_duplicates` using `filter`
  - Now we are ACTUALLY ready to remove duplicates!

```
function remove_duplicates(xs) {
    if (is_null(xs)) {
        return null;
    } else {
        const x = head(xs);
        const unprocessed_list = tail(xs);
        const unprocessed_list_x_removed = filter(x, unprocessed_list);

        return pair(x,
                remove_duplicates(unprocessed_list_x_removed)
        );
}
```

Recap

Question 2

`remove_duplicates(xs)`

- Write `remove_duplicates` using `filter`
  - Runtime complexity?
    - Still O(n^2)
  - Can we do better than this?
    - Nope

Recap

Question 3

- Coin change but now give the exact permutations!
  - Recall:
    - We went through how to count the number of ways to change coins
  - Now:
    - We want to actually find the ways themselves

Recap

Question 3

- Incomplete solution:

```
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 combi_A = ...

        // Combinations that do not use the head coin for the remaining amount.
        const combi_B = ...

        // Combinations that use the head coin.
        const combi_C = ...

        return append(combi_A, combi_C);
    }
}
```

Recap

Question 3

- Example execution:
  - `makeup_amount(22, list(1, 10, 5, 20, 1, 5, 1, 50))`
  - ```
    > list(
          list(1, 10, 5, 1, 5),
          list(1, 10, 5, 5, 1),
          list(1, 20, 1),
          list(10, 20, 1),
          list(10, 5, 1, 5, 1),
          list(20, 1, 1)
      )
    ```
    ```

Recap
Question 3

- Explanation:
  - `list(1, 10, 5, 20, 1, 5, 1, 50)  // these are all the coins we have`
  - `list(1, 10, 5, 20, 1, 5, 1, 50)  // combi 1`
  - `list(1, 10, 5, 20, 1, 5, 1, 50)  // combi 2`
  - `list(1, 10, 5, 20, 1, 5, 1, 50)  // combi 3`
  - `list(1, 10, 5, 20, 1, 5, 1, 50)  // combi 4`
  - `list(1, 10, 5, 20, 1, 5, 1, 50)  // combi 5`
  - `list(1, 10, 5, 20, 1, 5, 1, 50)  // combi 6`

Recap
Question 3

- Discover the recurrence relation:
  - Every coin has a chance of being selected to get every permutation
  - Notice that every coin is treated uniquely (contrast with lecture version)
    - e.g. combinations 3 and 4 are “different”
  - Systematically, go through the list of coins from left to right
  - At each coin, we decide whether to pick it, or not

Recap

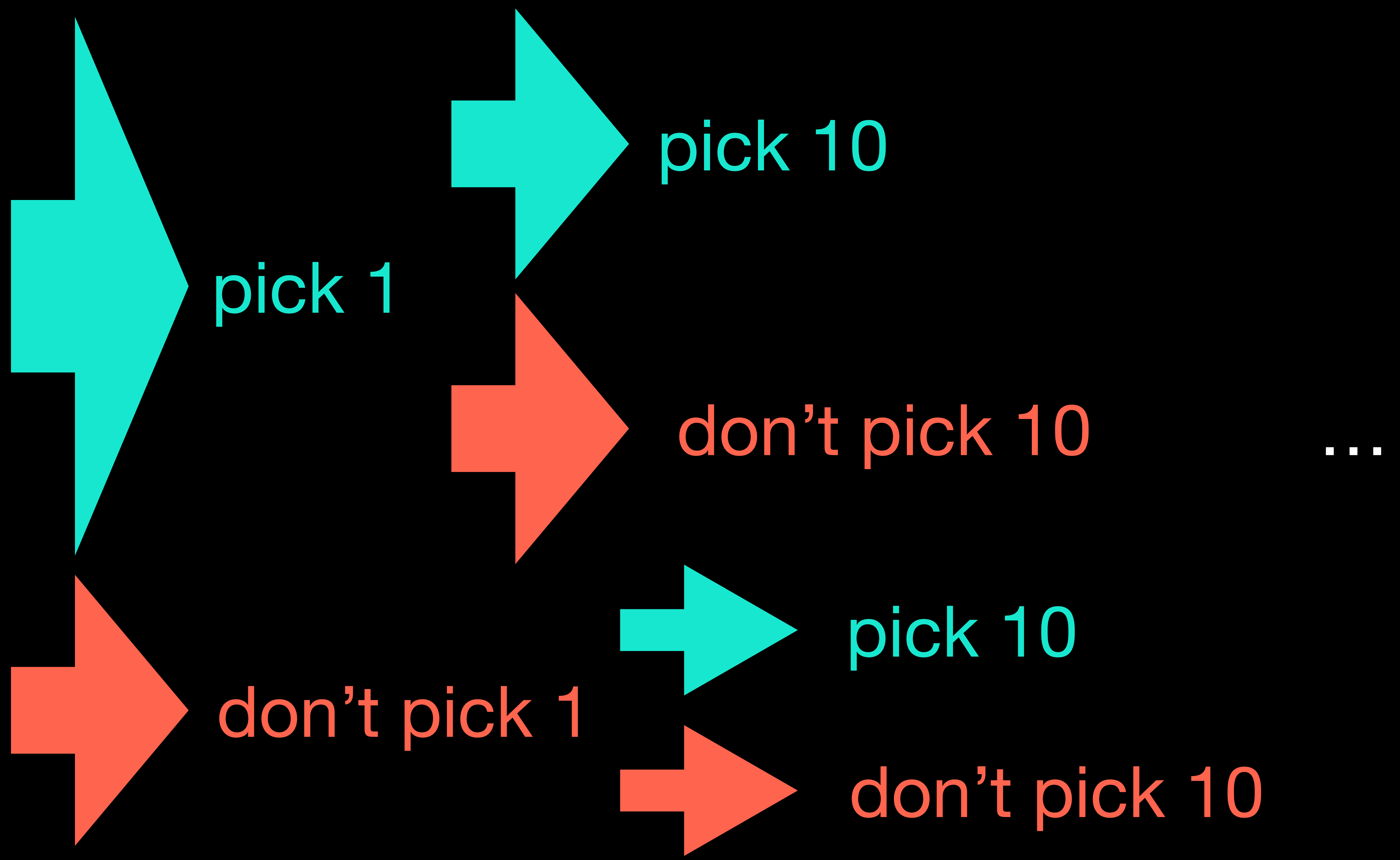
Question 3

- Intuition:
  - makeup_amount(amt, coins) =
    - combinations from picking the current coin, union with
    - combinations from NOT the picking current coin
  - // current coin: head(coins)
  - // union: we can use the `append` function

Recap

Question 3

- Observe
  - `list(1, 10, 5, 20, 1, 5, 1, 50)`
  - `list(1, 10, 5, 20, 1, 5, 1, 50)   // combi 1`
  - `list(1, 10, 5, 20, 1, 5, 1, 50)   // combi 2`
  - `list(1, 10, 5, 20, 1, 5, 1, 50)   // combi 3`
  - `list(1, 10, 5, 20, 1, 5, 1, 50)   // combi 4`
  - `list(1, 10, 5, 20, 1, 5, 1, 50)   // combi 5`
  - `list(1, 10, 5, 20, 1, 5, 1, 50)   // combi 6`



Recap

Question 3

- Naively:
  - `makeup_amount(amt, coins) = append(`

```
    makeup_amount(amt - head(coins), tail(coins)), // pick head(coins)
    makeup_amount(amt, tail(coins));  // don’t pick head(coins)
```

  - Close… but we have a problem
    - Notice that this will never include the head coin!

Recap

Question 3

- Why?
  - makeup_amount(amt, coins) = append(

```
makeup_amount(amt - head(coins), tail(coins)), // pick head(coins)
makeup_amount(amt, tail(coins));  // don't pick head(coins)
```

  - list(1, 10, 5, 20, 1, 5, 1, 50)   // combi 1
  - list(1, 10, 5, 20, 1, 5, 1, 50)   // combi 2
  - list(1, 10, 5, 20, 1, 5, 1, 50)   // combi 3
  - list(1, 10, 5, 20, 1, 5, 1, 50)   // combi 4
- We need to prepend `1` to every combination in the list returned by this sub-problem

Recap

Question 3

- Naively:
  - ```
    makeup_amount(amt, coins) = append(
        makeup_amount(amt - head(coins), tail(coins)), // pick head(coins)
        makeup_amount(amt, tail(coins));  // don’t pick head(coins)
    ```
- More strictly:
  - ```
    makeup_amount(amt, coins) = append(
        (append head(coins) to every member in makeup_amount(amt - head(coins), tail(coins))),
        // pick head(coins)
        makeup_amount(amt, tail(coins));  // don’t pick head(coins)
    ```

Recap

Question 3

```
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 combi_A = makeup_amount(x, tail(coins));

        // Combinations that do not use the head coin for the remaining amount.
        const combi_B = makeup_amount(x - head(coins), tail(coins));

        // Combinations that use the head coin.
        const headCoin = head(coins);
        const combi_C = map(combi => pair(headCoin, combi), combi_B);

        return append(combi_A, combi_C);
    }
}
```

don’t pick head(coins)

pick head(coins),
but we didn’t include
the headCoin itself

so we have to
prepend it here!

End of File