L6: List and Tree Processing

SWS3012: Structure and Interpretation of Computer Programs

Martin Henz

July 10, 2023

Outline

- List Processing (2.2.1)
- Higher-order List Processing (2.2.1 & 2.2.3)
- Trees and Tree Processing (2.2.2)
- Continuation-Passing Style

Outline

- List Processing (2.2.1)
- Higher-order List Processing (2.2.1 & 2.2.3)
- Trees and Tree Processing (2.2.2)
- Continuation-Passing Style

Review: Pairs

```
const p = pair(1, 2);  // creates a pair

head(p);  // accesses first component

tail(p);  // accesses second component
```

Review: List Discipline

- **Definition:**

  A ***list*** is either `null` or a **pair whose tail is a list**

- **Examples:**

```
null

pair(8, null)

pair(1, pair(2, pair(3, pair(4, null))))
```

Review: Shortcut for Constructing Lists

```
pair(1, pair(2, pair(3, pair(4, null))));
```

can be written as

```
list(1, 2, 3, 4);
```

- Displayed in **box notation** as

```
[1, [2, [3, [4, null]]]]
```

- Displayed in **list notation** as

```
list(1, 2, 3, 4)
```

Review: Test for Empty List

```
is_null(null)  → true

is_null(pair(1, null)) → false

is_null(pair(1, 2))    → false

is_null(list(1, 2, 3)) → false
```

Refinement of List Discipline

- **Definition:**

  A ***list of a certain data type*** is `null` or a **pair** whose **head is of that data type** and whose **tail is a list of that data type**

- **Examples:**

```
// a list of numbers
const listA = pair(1, pair(2, null));

// a list of booleans
const listB = list(true, true, false, true);

// a list of strings
const listC = list("X", "Y", "Z");

const listD = null;  // can be a list of any type
```

Review: Computing the Length of a List

- **Specification:**

  The ***length*** of the empty list is 0, and the length of a non-empty list is one more than the length of its tail

- **Implementation (recursive version):**

```
function length(xs) {
    return is_null(xs)
           ? 0
           : 1 + length(tail(xs));
}
```

[Show in Playground]

Review: Computing the Length of a List

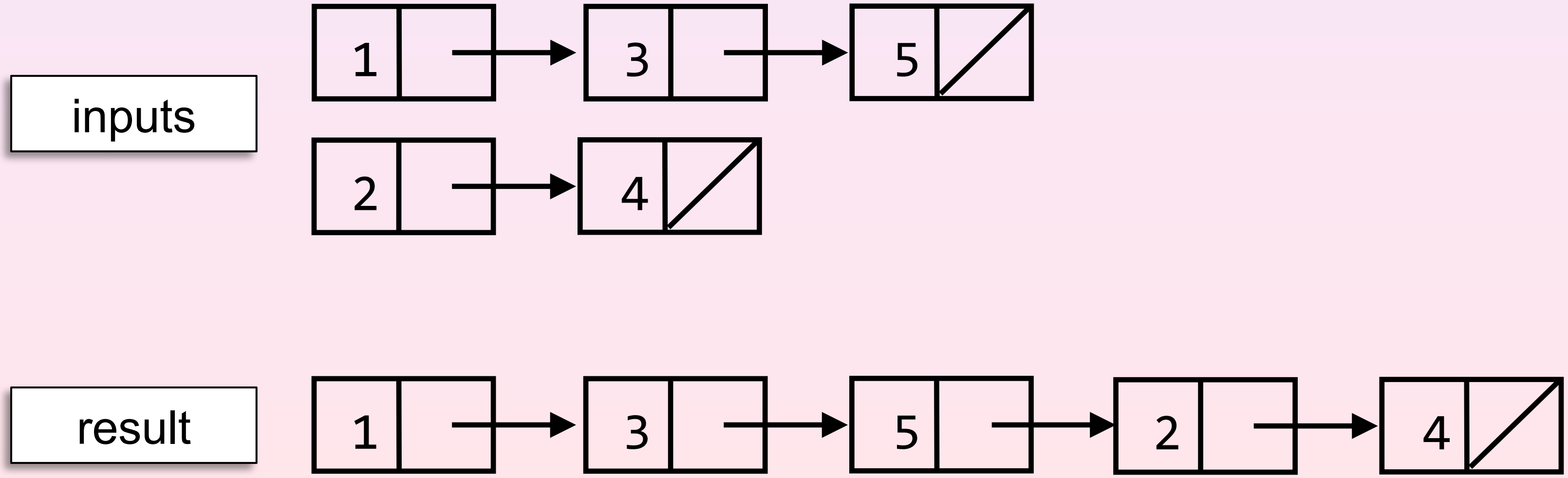
- **Iterative version:**

```
function length_iter(xs) {

    function len(ys, counted_so_far) {
        return is_null(ys)
            ? counted_so_far
            : len(tail(ys), counted_so_far + 1);
    }

    return len(xs, 0);
}
```

Show in Playground

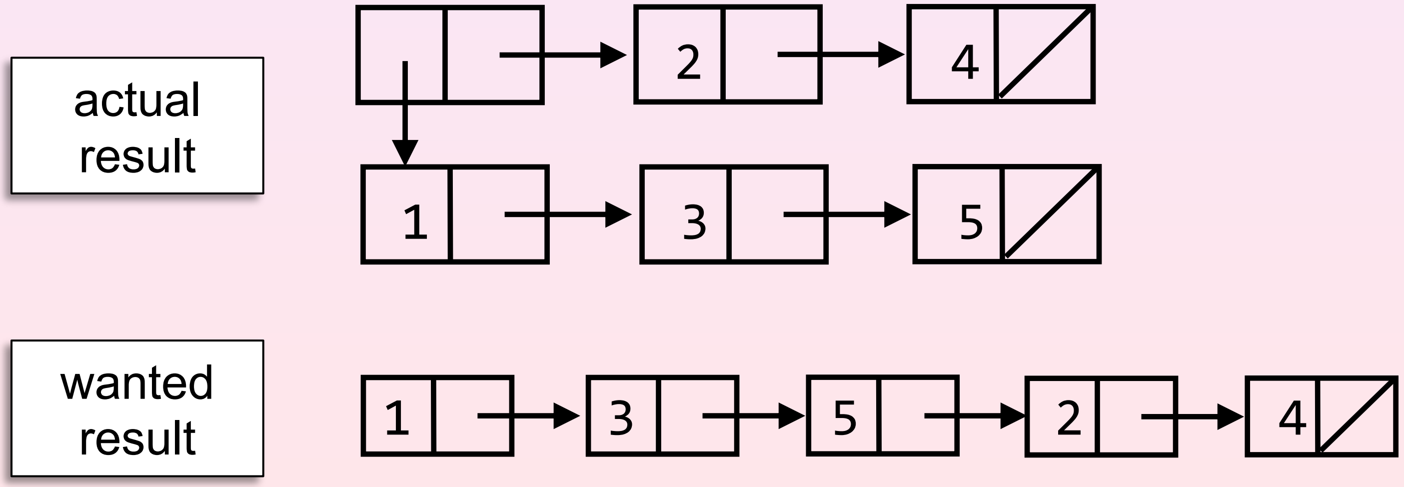
Appending Two Lists

- **Wanted:** Append `list(1, 3, 5)` and `list(2, 4)` to produce result equal to `list(1, 3, 5, 2, 4)`
  - `append(list(1, 3, 5), list(2, 4))`
    ➔ `list(1, 3, 5, 2, 4)`



Appending Two Lists

- **First attempt:** `const append1 = pair;`
- Does it work?
  - `append1(list(1, 3, 5), list(2, 4))`
    ➔ `pair(list(1, 3, 5), list(2, 4))`



Strategy for Append

- `append(list1, list2)`:
  - If `list1` is empty, return `list2`
  - Otherwise, wishful thinking!
    - Append the tail of `list1` to `list2`
    - Form a pair of the head of `list1` and the result

Implementation of Append in Source

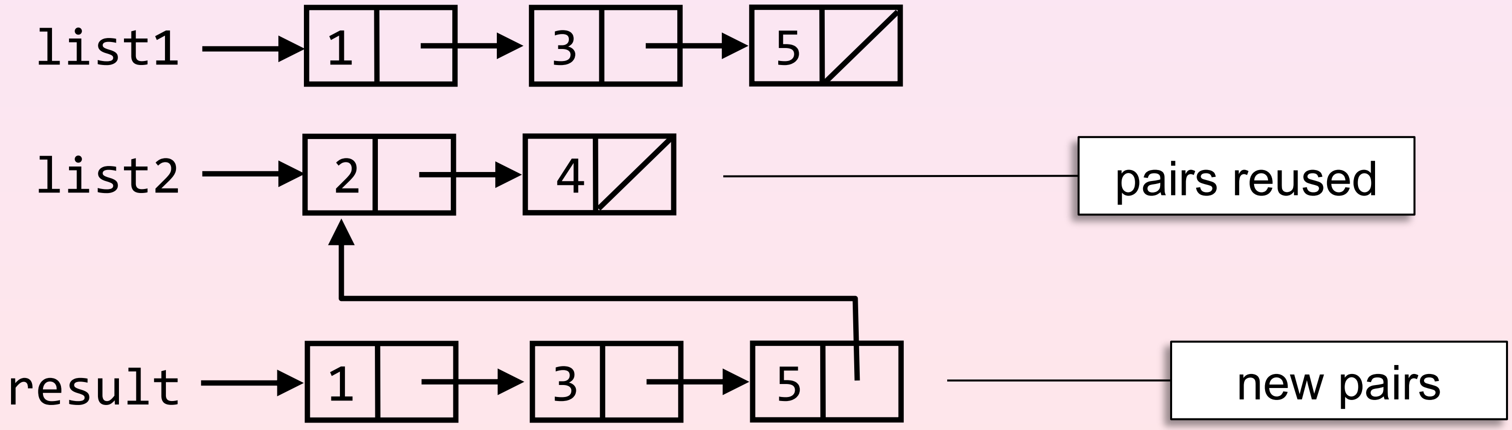
- **Second attempt:**

```
function append2(xs, ys) {
    return is_null(xs)
           ? ys
           : pair(head(xs), append2(tail(xs), ys));
}
```

- Order of growth in time?
- Order of growth in space?

Append: Example Run

```
function append2(xs, ys) {
    return is_null(xs)
           ? ys
           : pair(head(xs), append2(tail(xs), ys));
}
const list1 = list(1, 3, 5);
const list2 = list(2, 4);
const result = append2(list1, list2);
```



Reversing a List

- **First attempt:**

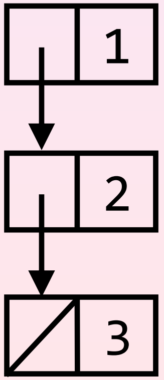
```
function reverse1(lst) {
    return is_null(lst)
           ? null
           : pair(reverse1(tail(lst)), head(lst));
}
```

Show in Playground

- Does it work?
  - `reverse1(list(1, 2, 3))`

    ➔ `pair(pair(pair(null, 3), 2), 1)`

actual result



Reversing a List

- **Second attempt:**

```
function reverse2(lst) {
    return is_null(lst)
           ? null
           : append(reverse2(tail(lst)),
                    list(head(lst)));
}
```

Show in Playground

- Does it work?
  - `reverse2(list(1, 2, 3))`
    ➔ `list(3, 2, 1)`

- Correct! But what about its run time?

Reversing a List

- **Third attempt:**

[Show in Playground](#)

```
function reverse3(xs) {
    function rev(original, reversed) {
        return is_null(original)
               ? reversed
               : rev(tail(original),
                     pair(head(original), reversed));
    }
    return rev(xs, null);
}
```

- Order of growth in time?
- Order of growth in space?

Outline

- List Processing (2.2.1)
- Higher-order List Processing (2.2.1 & 2.2.3)
- Trees and Tree Processing (2.2.2)
- Continuation-Passing Style

Example: Scaling a List

- Let us **scale** all elements of a list by a factor k:

```
function scale_list(xs, k) {
    return is_null(xs)
           ? null
           : pair(k * head(xs),
                  scale_list(tail(xs), k));
}

scale_list(list(1, 2, 3), 10);
➔ list(10, 20, 30)
```

[Show in Playground]

Example: Squaring a List

- Let us **square** all elements of a list:

```
function square_list(xs) {
    const square = x => x * x;
    return is_null(xs)
           ? null
           : pair(square(head(xs)),
                  square_list(tail(xs)));
}

square_list(list(1, 2, 3, 4));
➔ list(1, 4, 9, 16)
```

Show in Playground

Abstraction: `map`

- ***Mapping*** means applying a given function `f` **element-wise** to a given **list** `xs`
- The result is a **list** consisting of the results of applying `f` to each element of `xs`

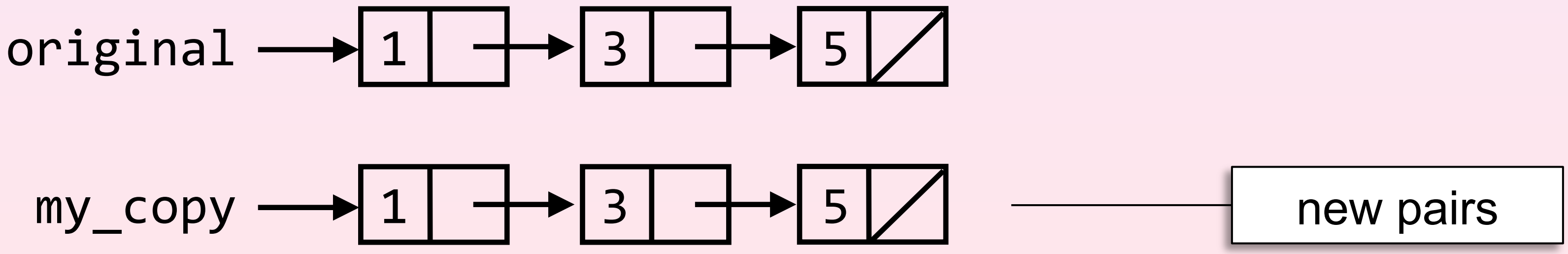
```
function map(f, xs) {
    return is_null(xs)
           ? null
           : pair(f(head(xs)), map(f, tail(xs)));
}
```

Rewriting using map Abstraction

```
function map(f, xs) {
    return is_null(xs)
           ? null
           : pair(f(head(xs)), map(f, tail(xs)));
}

function scale_list(xs, k) {
    return map(x => k * x, xs);
}

function square_list(xs) {
    return map(x => x * x, xs);
}
```

Making a Copy of a List

```
function copy(xs) {
    return map(x => x, xs);
}

const original = list(1, 3, 5);

const my_copy = copy(original);
```



Example: Taking Even Numbers from a List

- Take only even numbers from a list:

```
function even_numbers(xs) {
    return is_null(xs)
           ? null
           : head(xs) % 2 === 0
           ? pair(head(xs), even_numbers(tail(xs)))
           : even_numbers(tail(xs));
}

even_numbers(list(1, 2, 3, 4, 5, 6));
➔ list(2, 4, 6)
```

[Show in Playground]

Abstraction: filter

```
function filter(pred, xs) {
    return is_null(xs)
           ? null
           : pred(head(xs))
           ? pair(head(xs), filter(pred, tail(xs)))
           : filter(pred, tail(xs));
}
```

- Rewriting even_numbers using filter abstraction:

```
function even_numbers(xs) {
    return filter(x => x % 2 === 0, xs);
}
```

Show in Playground

Example: Summing Elements of a List

- Compute the sum of all elements of a list of numbers:

```
function list_sum(xs) {
    return is_null(xs)
           ? 0
           : head(xs) + list_sum(tail(xs));
}

list_sum( list(1, 2, 3) );
➔ 1 + list_sum( list(2, 3) )
➔ 1 + (2 + list_sum( list(3) ))
➔ 1 + (2 + (3 + list_sum(null)))
➔ 1 + (2 + (3 + 0))
➔ 6
```

right-to-left folding

[Show in Playground]

Abstraction: accumulate

```
function accumulate(op, initial, xs) {
    return is_null(xs)
           ? initial
           : op(head(xs),
                accumulate(op, initial, tail(xs)));
}
```

- Rewriting `list_sum` using `accumulate` abstraction:

```
function list_sum(xs) {
    return accumulate((x, y) => x + y, 0, xs);
}
```

Show in Playground

The Trio

```
function map(f, xs) {
    return is_null(xs)
           ? null
           : pair(f(head(xs)), map(f, tail(xs)));
}
function accumulate(op, initial, xs) {
    return is_null(xs)
           ? initial
           : op(head(xs), accumulate(op, initial, tail(xs)));
}
function filter(pred, xs) {
    return is_null(xs)
           ? null
           : pred(head(xs))
           ? pair(head(xs), filter(pred, tail(xs)))
           : filter(pred, tail(xs));
}
```

Outline

- List Processing (2.2.1)
- Higher-order List Processing (2.2.1 & 2.2.3)
- Trees and Tree Processing ([2.2.2](2.2.2))
- Continuation-Passing Style

Trees

- **Definition:**

  A ***tree*** ***of a certain data type*** is a **list** whose **elements** are of **that data type**, or **trees of that data type**

- **Examples:**

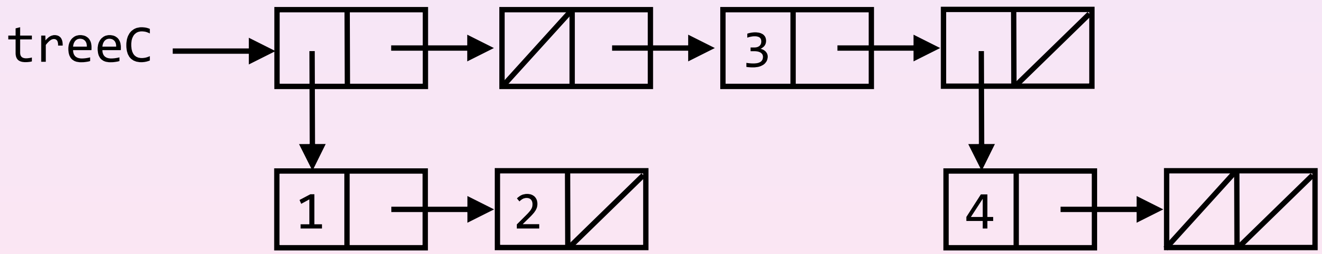
```
// trees of numbers
const treeA = list(1, 2, 3, 4);
const treeB = list(list(1, 2), list(3, 4));
const treeC = list(list(1, 2), null, 3, list(4, null));
const treeD = null;  // can be a tree of any type
```

- **Caveat:** Cannot consider `null` and **pair** as “certain data type”
  - So, we cannot have **trees of** `null`**s** and **trees of pairs**

Example Trees

- **Example tree of numbers:**

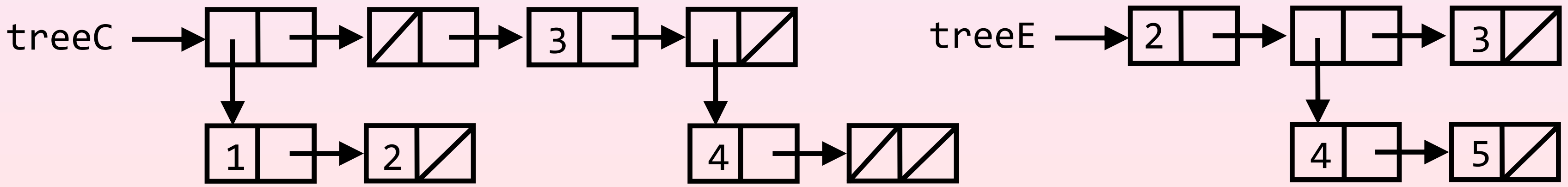
```
const treeC = list(list(1, 2), null, 3, list(4, null));
```



Alternative Definition of Trees

- A ***tree* of a certain data type** is
  - either `null`
  - or a **pair**
    - whose **tail is a tree of that data type** and
    - whose **head** is
      - either **of that data type**
      - or **a tree of that data type**

- Example trees of numbers:



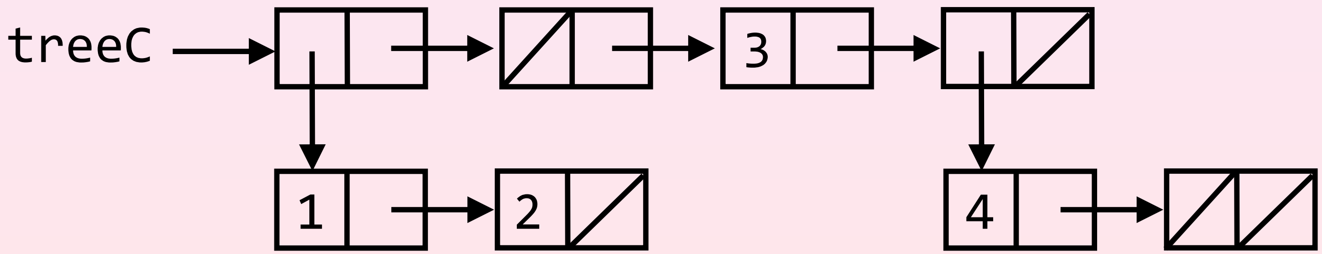
Counting Data Items in a Tree

- **Wanted:** `count_data_items(tree)` returns the number of data items in a given `tree`

- **Example:**

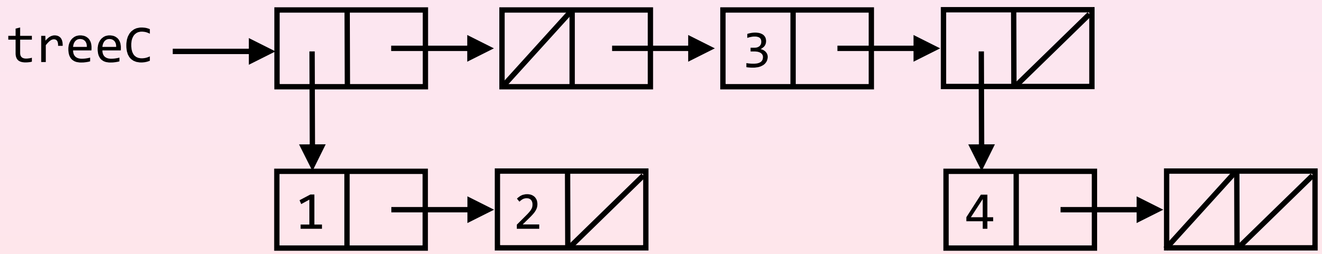
```
const treeC = list(list(1, 2), null, 3, list(4, null));
count_data_items(treeC);
```

➔ 4



Counting Data Items in a Tree

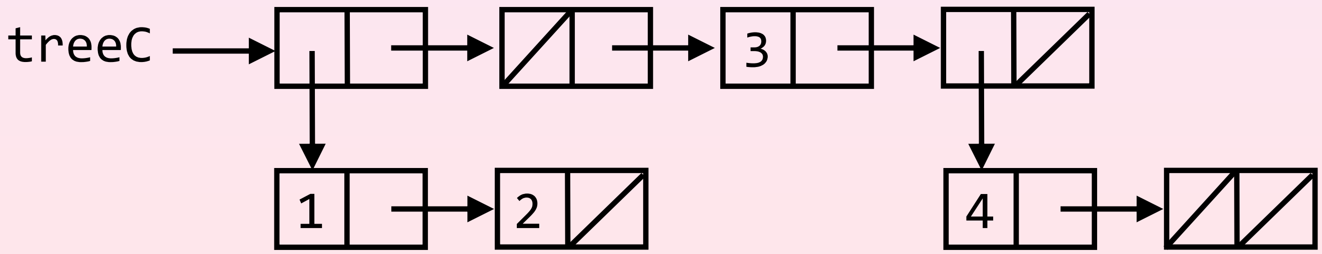
- **Idea:**
  1) Every **tree** is a **list**
  2) If the **list** is **empty**, we **return 0**
  3) If the **list** is **not empty**, we **add** the **number of data items of the head** to the **number of data items of the tail**
     - If the **head** is a **tree/list**, we **count its data items**;
       if it's **not a tree/list**, it's a data item and we **count 1**



Counting Data Items in a Tree

```
function count_data_items(tree) {
    return is_null(tree)
           ? 0
           : ( is_list(head(tree))
               ? count_data_items(head(tree))
               : 1 )
             +
             count_data_items(tail(tree));
}
```

Show in Playground



Scaling a Tree

- **Wanted:** `scale_tree(tree, k)` returns a new tree with the data items of `tree` scaled by the factor `k`

- **Example:**

```
const treeC = list(list(1, 2), null, 3, list(4, null));
```

```
scale_tree(treeC, 10);
```

➔ `list(list(10, 20), null, 30, list(40, null))`

Scaling a Tree

- **Recall:** A *tree* is a list whose elements are data items, or trees
- **Idea:** Map over the list — if element is a data item, scale element, if not, scale tree

- **Implementation:**

```
function scale_tree(tree, k) {
    return map(sub_tree =>
                  !is_list(sub_tree)
                  ? k * sub_tree
                  : scale_tree(sub_tree, k),
               tree);
}
```

Show in Playground

Abstraction: Mapping over Trees

```
function map_tree(f, tree) {
    return map(sub_tree =>
                   !is_list(sub_tree)
                   ? f(sub_tree)
                   : map_tree(f, sub_tree),
               tree);
}
```

scale_tree using map_tree

```
function scale_tree(tree, k) {
    return map_tree(data_item => data_item * k,
                    tree);
}
```

Show in Playground

Outline

- List Processing (2.2.1)
- Higher-order List Processing (2.2.1 & 2.2.3)
- Trees and Tree Processing (2.2.2)
- Continuation-Passing Style

A Closer Look at append

Show in Playground

```
function append(xs, ys) {
    return is_null(xs)
           ? ys
           : pair(head(xs), append(tail(xs), ys));
}
```

- Can we do this using an iterative process?

Iterative append

- **First attempt:**

[Show in Playground]

```
function append_iter(xs, ys) {
    return is_null(xs)
           ? ys
           : append_iter(tail(xs),
                         pair(head(xs), ys));
}

append_iter(list(1, 2, 3), list(4, 5, 6));
➔ list(3, 2, 1, 4, 5, 6)
```

Iterative append

- **Second attempt (using reverse):**

Show in Playground

```
function append_iter(xs, ys) {
    return is_null(xs)
           ? ys
           : append_iter(tail(xs), pair(head(xs), ys));
}
function append(xs, ys) {
    return append_iter(reverse(xs), ys);
}

append(list(1, 2, 3), list(4, 5, 6));
➔ list(1, 2, 3, 4, 5, 6)
```

Another Iterative Version of append

```
function append(xs, ys) {  // Recursive process
    return is_null(xs)
           ? ys
           : pair(head(xs), append(tail(xs), ys));
}

function app(current_xs, ys, c) {  // Iterative process
    return is_null(current_xs)
           ? c(ys)
           : app(tail(current_xs), ys,
                 x => c(pair(head(current_xs), x)));
}
function append_iter(xs, ys) {
    return app(xs, ys, x => x);
}
```

Show in Playground

Continuation-Passing Style

```
function app(current_xs, ys, c) {  // Iterative process
    return is_null(current_xs)
           ? c(ys)
           : app(tail(current_xs), ys,
                 x => c(pair(head(current_xs), x)));
}
function append_iter(xs, ys) {
    return app(xs, ys, x => x);
}
```

Show in Playground

- **Programming Pattern: CPS**
  - Passing the deferred operation as a function in an extra argument is called "**Continuation-Passing Style**" (**CPS**)
  - We can convert ***any*** recursive function this way!

Another CPS Example

- Recall the “divine” fractal solution from Lecture L3

```
function fractal_5(rune, n) {
   return n === 1
          ? rune
          : beside(rune,
                   fractal_5(stack(rune, rune), n - 1));
}
```

Show in Playground

Another CPS Example

```
function fractal_5(rune, n) {  // Recursive process
   return n === 1
          ? rune
          : beside(rune, fractal_5(stack(rune, rune), n - 1));
}

function frac(rune, n, c) {  // Iterative process
    return n === 1
           ? c(rune)
           : frac(stack(rune, rune), n - 1,
                   res => c(beside(rune, res)));
}
function fractal_5_iter(rune, n) {
    return frac(rune, n, rune => rune);
}
```

Show in Playground

Summary

- **List processing**: `length`, `append`, `reverse`
- **Higher-order list processing** with `map`, `filter`, `accumulate`
- **Trees**, **tree processing**, and **higher-order tree processing**
- **Continuation-Passing Style** can be used to turn ***any*** function **iterative**