More list processing

Jia Xiaodong

September 13, 2021

Admin matters

- Code style

Admin matters

- Code style
- Plagiarism

Admin matters

- Code style
- Plagiarism
- Mastery check

Built in operations

- `append(xs, ys)`
- `reverse(xs)`
- `for_each(f, xs)`
- `map(f, xs)`
- `filter(pred, xs)`
- `accumulate(f, init, xs)`
- Online reference

Map

- `map` takes in parameters `(f, xs)`

Map

- `map` takes in parameters `(f, xs)`
  - `xs` is a list of type `T`

Map

- `map` takes in parameters `(f, xs)`
  - `xs` is a list of type `T`
  - `f` is a function of type `T => any`.

Map

- `map` takes in parameters `(f, xs)`
  - `xs` is a list of type `T`
  - `f` is a function of type `T => any`.
- In short: `map(f, xs)` brings `xs` from `list(e1, e2, ...)` to `list(f(e1), f(e2), ...)`.

Map

- `map` takes in parameters `(f, xs)`
  - `xs` is a list of type `T`
  - `f` is a function of type `T => any`.
- In short: `map(f, xs)` brings `xs` from `list(e1, e2, ...)` to `list(f(e1), f(e2), ...)`.

Ex: Negating a list

```
map(x => -x, xs)
```

Accumulate

- `accumulate` takes in 3 parameters, `(f, init, xs)`.

Accumulate

- `accumulate` takes in 3 parameters, `(f, init, xs)`.
  - `xs` is a list of type `T`.

Accumulate

- `accumulate` takes in 3 parameters, `(f, init, xs)`.
  - `xs` is a list of type `T`.
  - `init` is a variable of type `U`.

Accumulate

- `accumulate` takes in 3 parameters, `(f, init, xs)`.
  - `xs` is a list of type `T`.
  - `init` is a variable of type `U`.
  - `f` is a function of type `(T, U) => U`

Accumulate

- `accumulate` takes in 3 parameters, `(f, init, xs)`.
  - `xs` is a list of type `T`.
  - `init` is a variable of type `U`.
  - `f` is a function of type `(T, U) => U`
- Imagine `xs, init` as a flat sequence of elements: `list(n1, n2, ..., nk, init)`. Then `accumulate` returns `f(n1, f(n2, ... f(nk-1, f(nk, init)) ...))`

Filter

- `filter` takes in 2 parameters, `(pred, xs)`.

Get even elements

```
filter(x => x % 2 === 0, list(1, 2, 3, 4, 5))
```

Filter

- `filter` takes in 2 parameters, `(pred, xs)`.
  - `xs` is a list of type `T`.

Get even elements

```
filter(x => x % 2 === 0, list(1, 2, 3, 4, 5))
```

Filter

- `filter` takes in 2 parameters, `(pred, xs)`.
  - `xs` is a list of type `T`.
  - `pred` is a function of type `T => true/false`

Get even elements

```
filter(x => x % 2 === 0, list(1, 2, 3, 4, 5))
```

Trees

Definition

A tree is a list of either elements or trees.

Example

Draw `list(list(1, 2), 3, 4)`. Compare with Fig. 2.6 in the textbook.

S6 Q1

Implement `map` using `accumulate`.

[1] Here again ~ is used to represent something like `equals()` or $\simeq$.

S6 Q1

Implement `map` using `accumulate`.

- `accumulate(pair, init, xs)` ~ `identity` [1].

[1] Here again ~ is used to represent something like `equals()` or $\simeq$.

S6 Q1

Implement `map` using `accumulate`.

- `accumulate(pair, init, xs)` ~ `identity` [1].

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

Challenge

Implement `filter` with `accumulate`.

[1] Here again ~ is used to represent something like `equals()` or $\simeq$.

S6 Q2

Use `filter` to write `remove_duplicates`.

S6 Q2

Use `filter` to write `remove_duplicates`.

```
function remove_duplicates(xs) {
    function pred(v) {
```

S6 Q2

Use `filter` to write `remove_duplicates`.

```
function remove_duplicates(xs) {
    function pred(v) {
        function f(x, y) {


            // return accumulate(f, *, xs) ... ?
```

S6 Q2

Use `filter` to write `remove_duplicates`.

```
function remove_duplicates(xs) {
    function pred(v) {
        function f(x, y) {
            return x === v ? y + 1 : y;
        }
        // return accumulate(f, *, xs) ... ?
```

S6 Q2

Use `filter` to write `remove_duplicates`.

```
function remove_duplicates(xs) {
    function pred(v) {
        function f(x, y) {
            return x === v ? y + 1 : y;
        }

        return accumulate(f, 0, xs) < 2;
    }
    return filter(pred, xs);
}
```

S6 Q2

Alternative

```
function remove_duplicates(xs) {
    return is_null(xs)
        ? null
```

S6 Q2

Alternative

```
function remove_duplicates(xs) {
    return is_null(xs)
        ? null
        : pair(head(xs),
```

S6 Q2

Alternative

```
function remove_duplicates(xs) {
    return is_null(xs)
        ? null
        : pair(head(xs),
            remove_duplicates(
```

S6 Q2

Alternative

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

S6 Q3

```
function makeup_amount(x, coins) {
    if (x === 0) {
        return list(null);
    } else if (x < 0 || is_null(coins)) {
        return null;
    } else {
        // Combinations that don't use the head coin.

        // Combinations after we remove the head coin.

        // Combinations that use the head coin.

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

S6 Q3

```
function makeup_amount(x, coins) {
    if (x === 0) {
        return list(null);
    } else if (x < 0 || is_null(coins)) {
        return null;
    } else {
        // Combinations that don't use the head coin.
        const combi_A = makeup_amount(x, tail(coins));
        // Combinations after we remove the head coin.

        // Combinations that use the head coin.

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

S6 Q3

```
function makeup_amount(x, coins) {
    if (x === 0) {
        return list(null);
    } else if (x < 0 || is_null(coins)) {
        return null;
    } else {
        // Combinations that don't use the head coin.
        const combi_A = makeup_amount(x, tail(coins));
        // Combinations after we remove the head coin.
        const combi_B = makeup_amount(x - head(coins), tail(coins));
        // Combinations that use the head coin.

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

S6 Q3

```
function makeup_amount(x, coins) {
    if (x === 0) {
        return list(null);
    } else if (x < 0 || is_null(coins)) {
        return null;
    } else {
        // Combinations that don't use the head coin.
        const combi_A = makeup_amount(x, tail(coins));
        // Combinations after we remove the head coin.
        const combi_B = makeup_amount(x - head(coins), tail(coins));
        // Combinations that use the head coin.
        const combi_C = map(x => pair(head(coins), x), combi_B);
        return append(combi_A, combi_C);
    }
}
```

S6 Q4

Use `accumulate` to write `remove_duplicates`.

```
function remove_duplicates(xs) {
```

S6 Q4

Use `accumulate` to write `remove_duplicates`.

```
function remove_duplicates(xs) {

        accumulate(
```

S6 Q4

Use `accumulate` to write `remove_duplicates`.

```
function remove_duplicates(xs) {

        accumulate(
            (x, ys) => is_null(member(x, ys))
                ? pair(x, ys)
                : ys,
```

S6 Q4

Use `accumulate` to write `remove_duplicates`.

```
function remove_duplicates(xs) {

        accumulate(
            (x, ys) => is_null(member(x, ys))
                ? pair(x, ys)
                : ys,
            null, xs);
}
```

S6 Q4

Use `accumulate` to write `remove_duplicates`.

```
function remove_duplicates(xs) {
    return
        accumulate(
            (x, ys) => is_null(member(x, ys))
                ? pair(x, ys)
                : ys,
            null, xs);
}
```

S6 Q5

Write a function `subsets(xs)` that returns the set (a list) of all subsets of `xs`.

S6 Q5

Write a function `subsets(xs)` that returns the set (a list) of all subsets of `xs`.

```
function subsets(xs) {
```

S6 Q5

Write a function `subsets(xs)` that returns the set (a list) of all subsets of `xs`.

```
function subsets(xs) {
        if (is_null(xs)) {
            return list(null)
        }
```

S6 Q5

Write a function `subsets(xs)` that returns the set (a list) of all subsets of `xs`.

```
function subsets(xs) {
        if (is_null(xs)) {
            return list(null)
        }
        else {
            const subset_a = subsets(tail(xs));
```

S6 Q5

Write a function `subsets(xs)` that returns the set (a list) of all subsets of `xs`.

```
function subsets(xs) {
        if (is_null(xs)) {
            return list(null)
        }
        else {
            const subset_a = subsets(tail(xs));
            const subset_b =
                map(x => pair(head(xs), x), subset_a);
```

S6 Q5

Write a function `subsets(xs)` that returns the set (a list) of all subsets of `xs`.

```
function subsets(xs) {
        if (is_null(xs)) {
            return list(null)
        }
        else {
            const subset_a = subsets(tail(xs));
            const subset_b =
                map(x => pair(head(xs), x), subset_a);
            return append(subset_a, subset_b);
        }
}
```

S6 Q6

Write a function `permute(xs)` that returns a list of all permutations of `xs`.

S6 Q6

Write a function `permute(xs)` that returns a list of all permutations of `xs`.

```
function permutations(xs) {









    }
}
```

S6 Q6

Write a function `permute(xs)` that returns a list of all permutations of `xs`.

```
function permutations(xs) {
    if(is_null(xs) {
        return list(xs);
    }









    }
}
```

S6 Q6

Write a function `permute(xs)` that returns a list of all permutations of `xs`.

```
function permutations(xs) {
    if(is_null(xs) {
        return list(xs);
    }
    else {




    }
}
```

S6 Q6

Write a function `permute(xs)` that returns a list of all permutations of `xs`.

```
function permutations(xs) {
    if(is_null(xs) {
        return list(xs);
    }
    else {


                permutations(remove(x, xs)) ),


    }
}
```

S6 Q6

Write a function `permute(xs)` that returns a list of all permutations of `xs`.

```
function permutations(xs) {
    if(is_null(xs) {
        return list(xs);
    }
    else {
        return map(x =>
            map(z => pair(x, z),
                permutations(remove(x, xs)) ),
            xs);
    }
}
```

Q7

Write `accumulate_n` that accumulates a list of lists.

Q7

Write `accumulate_n` that accumulates a list of lists.

```
function accumulate_n(op, init, seqs) {
    return is_null(head(seqs))
        ? null
        : pair(



        );
}
```

Q7

Write `accumulate_n` that accumulates a list of lists.

```
function accumulate_n(op, init, seqs) {
    return is_null(head(seqs))
        ? null
        : pair(
            accumulate(op, init, map(head, seqs)),

        );
}
```

Q7

Write accumulate_n that accumulates a list of lists.

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

Q8

Write `insert(x, xs)` that puts `x` at the correct spot in `xs`.

Q8

Write `insert(x, xs)` that puts `x` at the correct spot in `xs`.

```
function insert(x, xs) {




}
```

Q8

Write `insert(x, xs)` that puts `x` at the correct spot in `xs`.

```
function insert(x, xs) {
    if (is_null(xs)) {
        return list(x);
    }









}
```

Q8

Write `insert(x, xs)` that puts `x` at the correct spot in `xs`.

```
function insert(x, xs) {
    if (is_null(xs)) {
        return list(x);
    }
    else {
        const y = head(xs);
        return x < y
            ? pair(x, pair(y, tail(xs)))
            : pair(y, insert(x, tail(xs)));
    }
}
```

Q8

Implement insertion sort using `insert`.

Q8

Implement insertion sort using `insert`.

```
function sort(xs) {
    return accumulate(
        (x, acc) => insert(x, acc),
        list(),
        xs);
}
```