Data Abstraction

Jia Xiaodong

September 6, 2021

Data Structures

For example, I want to sort a bunch of numbers:

Data Structures

For example, I want to sort a bunch of numbers:

```
// returns them in sorted order
function sort(a,b,c,d...) { ??? }
```

Data Structures

For example, I want to sort a bunch of numbers:

```
// returns them in sorted order
function sort(a,b,c,d...) { ??? }
```

Better?

```
function sort(list) { ... return list; }
```

What is data?

...we can think of data as defined by some collection of selectors and constructors, together with specified conditions that these functions must fulfil in order to be a valid representation. — SICP §2.1.3

Church encoding *

Ex. 2.6

Give a definition for plus, representing natural numbers in the following way:

```
const zero = f => x => x;
function succ(n) {
        return f => x => f(n(f)(x));
}
```

Church encoding *

Ex. 2.6

Give a definition for plus, representing natural numbers in the following way:

```
const zero = f => x => x;
function succ(n) {
        return f => x => f(n(f)(x));
}
```

```
const one = succ(zero);
```

Church encoding *

Ex. 2.6

Give a definition for plus, representing natural numbers in the following way:

```
const zero = f => x => x;
function succ(n) {
        return f => x => f(n(f)(x));
}
```

```
const one = succ(zero);
f => x => f(zero(f)(x))
```

Church encoding *

Ex. 2.6

Give a definition for plus, representing natural numbers in the following way:

```
const zero = f => x => x;
function succ(n) {
        return f => x => f(n(f)(x));
}
```

```
const one = succ(zero);
f => x => f(zero(f)(x))
f => x => f((x => x)(x))
```

Church encoding *

Ex. 2.6

Give a definition for plus, representing natural numbers in the following way:

```
const zero = f => x => x;
function succ(n) {
        return f => x => f(n(f)(x));
}
```

```
const one = succ(zero);
f => x => f(zero(f)(x))
f => x => f((x => x)(x))
f => x => f(x)
```

Church encoding *

Ex. 2.6

Give a definition for plus, representing natural numbers in the following way:

```
const zero = f => x => x;
function succ(n) {
        return f => x => f(n(f)(x));
}
```

```
const one = succ(zero);
f => x => f(zero(f)(x))
f => x => f((x => x)(x))
f => x => f(x)

const two = succ(one);
```

Church encoding *

Ex. 2.6

Give a definition for plus, representing natural numbers in the following way:

```
const zero = f => x => x;
function succ(n) {
        return f => x => f(n(f)(x));
}
```

```
const one = succ(zero);
f => x => f(zero(f)(x))
f => x => f((x => x)(x))
f => x => f(x)

const two = succ(one);
f => x => f(one(f)(x))
```

Church encoding *

Ex. 2.6

Give a definition for plus, representing natural numbers in the following way:

```
const zero = f => x => x;
function succ(n) {
        return f => x => f(n(f)(x));
}
```

```
const one = succ(zero);
f => x => f(zero(f)(x))
f => x => f((x => x)(x))
f => x => f(x)

const two = succ(one);
f => x => f(one(f)(x))
f => x => f(((x => f(x))(x))
```

Church encoding *

Ex. 2.6

Give a definition for plus, representing natural numbers in the following way:

```
const zero = f => x => x;
function succ(n) {
        return f => x => f(n(f)(x));
}
```

```
const one = succ(zero);
f => x => f(zero(f)(x))
f => x => f((x => x)(x))
f => x => f(x)

const two = succ(one);
f => x => f(one(f)(x))
f => x => f(((x => f(x))(x))
f => x => f(f(x))
```

Church encoding *

Ex. 2.6

Give a definition for plus, representing natural numbers in the following way:

```
const zero = f => x => x;
function succ(n) {
        return f => x => f(n(f)(x));
}
```

```
const one = succ(zero);
f => x => f(zero(f)(x))
f => x => f((x => x)(x))
f => x => f(x)
```

```
const two = succ(one);
f => x => f(one(f)(x))
f => x => f(((x => f(x))(x))
f => x => f(f(x))
```

```
const three = succ(two);
f => x => f(two(f)(x))
f => x => f((x => ffx)(x))
f => x => fffx
```

Church encoding *

Ex. 2.6

Give a definition for plus, representing natural numbers in the following way:

```
const zero = f => x => x;
function succ(n) {
        return f => x => f(n(f)(x));
}
```

```
const one = succ(zero);
f => x => f(zero(f)(x))
f => x => f((x => x)(x))
f => x => f(x)
```

```
const two = succ(one);
f => x => f(one(f)(x))
f => x => f(((x => f(x))(x))
f => x => f(f(x))
```

```
const three = succ(two);
f => x => f(two(f)(x))
f => x => f((x => ffx)(x))
f => x => fffx
```

```
const four = succ(three);
f => x => f(three(f)(x))
f => x => f((x => fffx)(x))
f => x => ffffx
```

Church encoding *

Cont.

```
const one = f => x => fx
const three = f => x => fffx
const four = f => x => ffffx
```

Church encoding *

Cont.

```
const one = f => x => fx
const three = f => x => fffx
const four = f => x => ffffx

three(f)(x) === fffx
one(f)(fffx) === f fffx
```

Church encoding *

Cont.

```
const one = f => x => fx
const three = f => x => fffx
const four = f => x => ffffx

three(f)(x) === fffx
one(f)(fffx) === f fffx

four(f)(x) === one(f)(three(f)(x))
```

Church encoding *

Cont.

```
const one = f => x => fx
const three = f => x => fffx
const four = f => x => ffffx

three(f)(x) === fffx
one(f)(fffx) === f fffx

four(f)(x) === one(f)(three(f)(x))

function plus(a, b) {
    return f => x => a(f)(b(f)(x));
}
```

Church encoding *

Cont.

Give a definition for `pred`.

Church encoding *

Cont.

Give a definition for pred.

```
const one = f => x => f(x);
const two = f => x => f(f(x));
```

Church encoding *

Cont.

Give a definition for pred.

```
const one = f => x => f(x);
const two = f => x => f(f(x));

function contain(n) { return p => p(n); }
function extract(c) { return c(u => u); }       // c is a container
```

Church encoding *

Cont.

Give a definition for pred.

```
const one = f => x => f(x);
const two = f => x => f(f(x));

function contain(n) { return p => p(n); }
function extract(c) { return c(u => u); }       // c is a container

function inc(c) { return h => h(c(f)); }
const init = u => x
```

Church encoding *

Cont.

Give a definition for pred.

```
const one = f => x => f(x);
const two = f => x => f(f(x));

function contain(n) { return p => p(n); }
function extract(c) { return c(u => u); }       // c is a container

function inc(c) { return h => h(c(f)); }
const init = u => x
inc(init) = h => h(x)           // contain(x)
inc(inc(init)) = i => i(f(x))   // contain(f(x))
```

Church encoding *

Cont.

Give a definition for pred.

```
const one = f => x => f(x);
const two = f => x => f(f(x));

function contain(n) { return p => p(n); }
function extract(c) { return c(u => u); }       // c is a container

function inc(c) { return h => h(c(f)); }
const init = u => x
inc(init) = h => h(x)           // contain(x)
inc(inc(init)) = i => i(f(x))   // contain(f(x))

function pred(n) { return f => x => extract(n(inc)(init); }
```

Church encoding *

Cont.

Give a definition for pred.

```
const one = f => x => f(x);
const two = f => x => f(f(x));

function contain(n) { return p => p(n); }
function extract(c) { return c(u => u); }       // c is a container

function inc(c) { return h => h(c(f)); }
const init = u => x
inc(init) = h => h(x)                // contain(x)
inc(inc(init)) = i => i(f(x))    // contain(f(x))

function pred(n) { return f => x => extract(n(inc)(init); }

function pred(n) { return f => x =>
    extract(n(c => h => h(c(f)))(init); }
```

Church encoding *

Quest

New definitions. Implement succ and pred.

```
const zero = f => x => x;
const one = f => x => f(zero, () => zero(f)(x));
const two = f => x => f(one, () => one(f)(x));
```

Church encoding *

Quest

New definitions. Implement succ and pred.

```
const zero = f => x => x;
const one = f => x => f(zero, () => zero(f)(x));
const two = f => x => f(one, () => one(f)(x));
```

```
function succ(n) { f => x => f(n, () => x); }
```

Church encoding *

Quest

New definitions. Implement succ and pred.

```
const zero = f => x => x;
const one = f => x => f(zero, () => zero(f)(x));
const two = f => x => f(one, () => one(f)(x));
```

```
function succ(n) { f => x => f(n, () => x); }

function pred(n) { f => x => n((m, n) => m)(zero); }

function plus(a, b) { a((m, n) => succ(n())) (b); }
```

Pair

A pair is a collection of two items. We assign one to the *head*, and the other to the *tail* of the pair.

Pair

A pair is a collection of two items. We assign one to the *head*, and the other to the *tail* of the pair.

Ex. 2.4 *

Possible implementation:

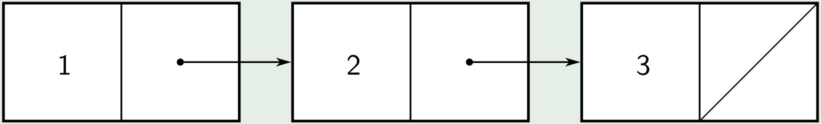
```
function pair(x,y) { return f => f(x, y); }
function head(p) { return p((x, y) => x); }
function tail(p) { return p((x, y) => y); }
```

List

`null` is a (empty) list. A list is a pair whose tail is a list.

Example

```
list(1, 2, 3) === pair(1, pair(2, pair(3, null)))
```
[1]



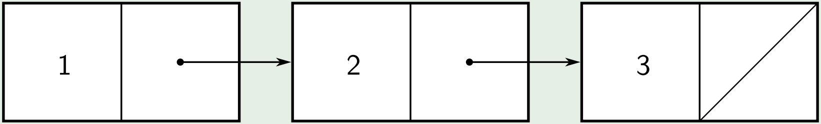
[1] This actually evaluates to `false`. What I mean by === here, for the lack of a better way to write it, is that they mean the same thing.

List

`null` is a (empty) list. A list is a pair whose tail is a list.

Example

```
list(1, 2, 3) === pair(1, pair(2, pair(3, null)))
```
[1]



Predeclared functions:

- LISTS documentation
- S2 Language Spec

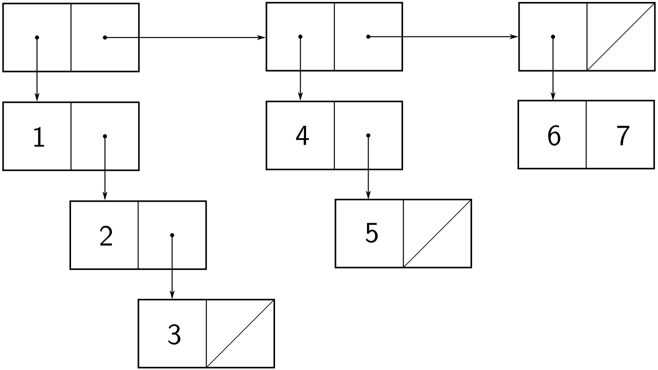
[1] This actually evaluates to `false`. What I mean by === here, for the lack of a better way to write it, is that they mean the same thing.

S5 Q1

Draw box and pointer diagram and give the printed representation for `list(list(1, 2, list(3)), list(4, 5), pair(6, 7));`

S5 Q1

Draw box and pointer diagram and give the printed representation for `list(list(1, 2, list(3)), list(4, 5), pair(6, 7));`



S5 Q1

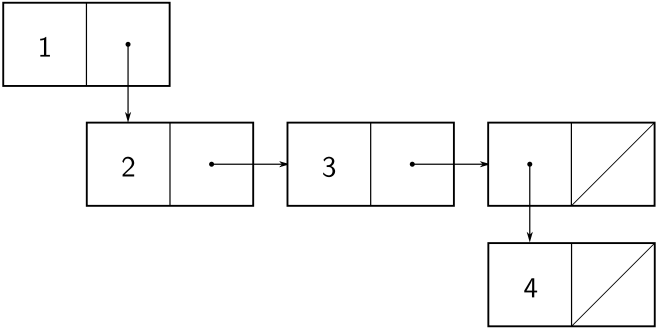
Cont.

Draw box and pointer diagram and give the printed representation for `pair(1, list(2, 3, pair(4, null)));`

S5 Q1

Cont.

Draw box and pointer diagram and give the printed representation for `pair(1, list(2, 3, pair(4, null)));`



S5 Q1

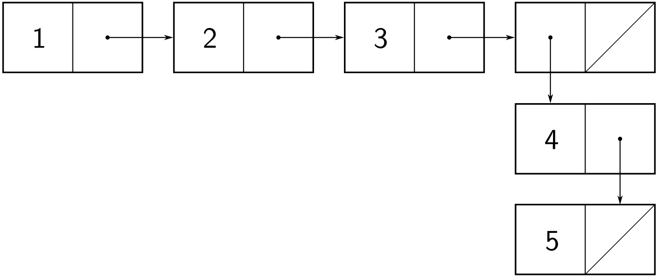
Cont.

Draw box and pointer diagram and give the printed representation for `pair(1, pair(2, list(3, list(4, 5))));`

S5 Q1

Cont.

Draw box and pointer diagram and give the printed representation for `pair(1, pair(2, list(3, list(4, 5))));`



S5 Q2

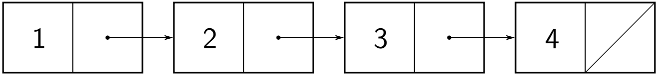
```
function reverse(lst) {
    return is_null(lst)
        ? null
        : pair(reverse(tail(lst)), head(lst));
}
```

Evaluate `reverse(list(1, 2, 3, 4));`.

S5 Q2

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

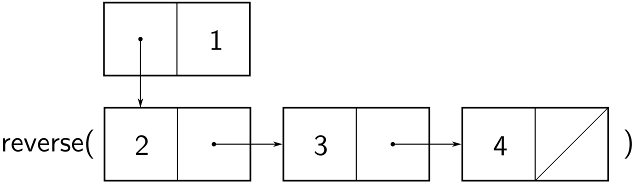
Evaluate `reverse(list(1, 2, 3, 4));`.



S5 Q2

Cont.

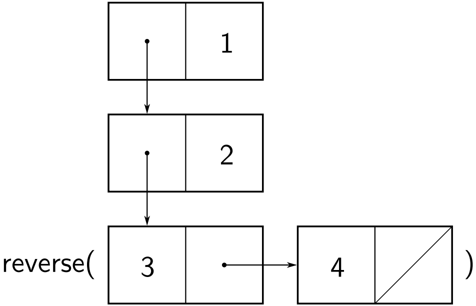
```
function reverse(lst) {
    return is_null(lst)
        ? null
        : pair(reverse(tail(lst)), head(lst));
}
```



S5 Q2

Cont.

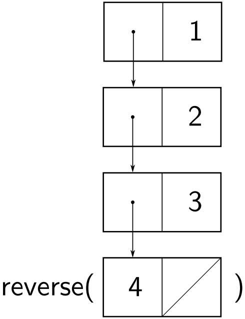
```
function reverse(lst) {
    return is_null(lst)
        ? null
        : pair(reverse(tail(lst)), head(lst));
}
```



S5 Q2

Cont.

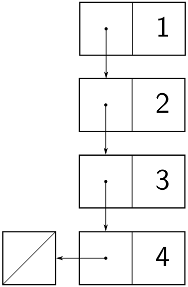
```
function reverse(lst) {
    return is_null(lst)
        ? null
        : pair(reverse(tail(lst)), head(lst));
}
```



S5 Q2

Cont.

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



S5 Q3

Write expressions using `lst`, `head`, `tail` that will return 1 with

```
lst = list(7, list(6, 5, 4), 3, list(2, 1));
```

S5 Q3

Write expressions using `lst`, `head`, `tail` that will return 1 with

```
lst = list(7, list(6, 5, 4), 3, list(2, 1));
```

- `head tail tail tail` gets us to `list(2, 1)`.

S5 Q3

Write expressions using `lst`, `head`, `tail` that will return 1 with
`lst = list(7, list(6, 5, 4), 3, list(2, 1));`

- `head tail tail tail` gets us to `list(2, 1)`.
- Then, `head tail` gets us 1.

S5 Q3

Write expressions using `lst`, `head`, `tail` that will return 1 with `lst = list(7, list(6, 5, 4), 3, list(2, 1));`

- `head tail tail tail` gets us to `list(2, 1)`.
- Then, `head tail` gets us 1.

`head(tail(head(tail(tail(tail(lst)))))).`

S5 Q3

Cont.

Write expressions using `lst`, `head`, `tail` that will return 1 with

```
lst = list(list(7), list(6, 5, 4), list(3, 2), 1);
```

S5 Q3

Cont.

Write expressions using `lst`, `head`, `tail` that will return 1 with

```
lst = list(list(7), list(6, 5, 4), list(3, 2), 1);
```

```
head(tail(tail(tail(lst))))
```

S5 Q3

Cont.

Write expressions using lst, head, tail that will return 1 with

```
lst = list(7, list(6), list(5, list(4)),
        list(3, list(2, list(1))));
```

S5 Q3

Cont.

Write expressions using `lst`, `head`, `tail` that will return 1 with

```
lst = list(7, list(6), list(5, list(4)),
        list(3, list(2, list(1))));
```

- `head tail tail tail` gets us to `list(3, list(...))`.

S5 Q3

Cont.

Write expressions using lst, head, tail that will return 1 with

```
lst = list(7, list(6), list(5, list(4)),
        list(3, list(2, list(1))));
```

- head tail tail tail gets us to list(3, list(...)).
- Then head tail gets us to list(2, list(1)).

S5 Q3

Cont.

Write expressions using `lst`, `head`, `tail` that will return 1 with

```
lst = list(7, list(6), list(5, list(4)),
        list(3, list(2, list(1))));
```

- `head tail tail tail` gets us to `list(3, list(...))`.
- Then `head tail` gets us to `list(2, list(1))`.
- Then `head tail` gets us to `list(1)`.

S5 Q3

Cont.

Write expressions using `lst`, `head`, `tail` that will return 1 with

```
lst = list(7, list(6), list(5, list(4)),
        list(3, list(2, list(1))));
```

- `head tail tail tail` gets us to `list(3, list(...))`.
- Then `head tail` gets us to `list(2, list(1))`.
- Then `head tail` gets us to `list(1)`.
- Then `head` gives us 1.

S5 Q3

Cont.

Write expressions using `lst`, `head`, `tail` that will return 1 with

```
lst = list(7, list(6), list(5, list(4)),
        list(3, list(2, list(1))));
```

- `head tail tail tail` gets us to `list(3, list(...))`.
- Then `head tail` gets us to `list(2, list(1))`.
- Then `head tail` gets us to `list(1)`.
- Then `head` gives us 1.

```
head(head(tail(head(tail(head(tail(tail(tail(lst)))))))))
```

S5 Q3

Cont.

Write expressions using `lst`, `head`, `tail` that will return 1 with

```
lst = list(7,
        list(list(6, 5), list(4), 3, 2), list(list(1)));
```

S5 Q3

Cont.

Write expressions using lst, head, tail that will return 1 with

```
lst = list(7,
        list(list(6, 5), list(4), 3, 2), list(list(1)));
```

- head tail tail gets us to list(list(1)).

S5 Q3

Cont.

Write expressions using lst, head, tail that will return 1 with

```
lst = list(7,
        list(list(6, 5), list(4), 3, 2), list(list(1)));
```

- head tail tail gets us to list(list(1)).
- Then head head gets us 1.

S5 Q3

Cont.

Write expressions using `lst`, `head`, `tail` that will return 1 with

```
lst = list(7,
        list(list(6, 5), list(4), 3, 2), list(list(1)));
```

- `head tail tail` gets us to `list(list(1))`.
- Then `head head` gets us 1.

```
head(head(head(tail(tail(lst)))))
```

S5 IC-Q1

Write function `every_second` that takes in a list and returns a list containing every other element, starting from the first element.

S5 IC-Q1

Write function every_second that takes in a list and returns a list containing every other element, starting from the first element.

```
function every_second(lst) {
    function h(res, n, max) {
        return n >= max
            ? res
            : h(pair(list_ref(lst, n), res), n + 2, max);
    }
    return h(null, 1, length(lst));
}
```

S5 IC-Q1

Cont.

Write function `every_second` that takes in a list and returns a list containing every other element, starting from the first element.

S5 IC-Q1

Cont.

Write function every_second that takes in a list and returns a list containing every other element, starting from the first element.

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

S5 IC-Q2

Write a function that takes in a list of numbers and returns a list containing (1) the sum of even-ranked numbers and (2) the sum of odd-ranked numbers.

S5 IC-Q2

Write a function that takes in a list of numbers and returns a list containing (1) the sum of even-ranked numbers and (2) the sum of odd-ranked numbers.

```
function every_second_odd(lst) ...
function every_second_even(lst) ...
function sum(lst) ...
```

S5 IC-Q2

Write a function that takes in a list of numbers and returns a list containing (1) the sum of even-ranked numbers and (2) the sum of odd-ranked numbers.

S5 IC-Q2

Write a function that takes in a list of numbers and returns a list containing (1) the sum of even-ranked numbers and (2) the sum of odd-ranked numbers.

```
function sum(lst){
    function s(e, o, lst, iseven) {
```

S5 IC-Q2

Write a function that takes in a list of numbers and returns a list containing (1) the sum of even-ranked numbers and (2) the sum of odd-ranked numbers.

```
function sum(lst){
    function s(e, o, lst, iseven) {
        return is_null(lst)
            ? list(e, o)
```

S5 IC-Q2

Write a function that takes in a list of numbers and returns a list containing (1) the sum of even-ranked numbers and (2) the sum of odd-ranked numbers.

```
function sum(lst){
    function s(e, o, lst, iseven) {
        return is_null(lst)
            ? list(e, o)
            : iseven
                ? s(e + head(lst), o, tail(lst), false)
                : s(e, o + head(lst), tail(lst), true);
    }
```

S5 IC-Q2

Write a function that takes in a list of numbers and returns a list containing (1) the sum of even-ranked numbers and (2) the sum of odd-ranked numbers.

```
function sum(lst){
    function s(e, o, lst, iseven) {
        return is_null(lst)
            ? list(e, o)
            : iseven
                ? s(e + head(lst), o, tail(lst), false)
                : s(e, o + head(lst), tail(lst), true);
    }
    return s(0, 0, lst, true);
}
```

Q6

Lexicographic order

Write a function `lexico(xs, ys)` that takes in lists of characters `xs` and `ys` and returns `true` iff `xs > ys` lexicographically.

```
function lexico(xs, ys) {
    if (is_null(xs)) {
        return false;
    } else if (is_null(ys)) {
        return true;
    } else if (head(xs) === head(ys)) {
        return lexico(tail(xs), tail(ys));
    } else if (head(xs) > head(ys)) {
        return true;
    } else {
        return false;
    }
}
```

Q7

Substring

Write a function substr(xs, ys) that takes in lists of characters xs and ys and returns true iff ys is a substring of xs.

```
function substr(xs, ys) {
```

Q7

Substring

Write a function substr(xs, ys) that takes in lists of characters xs and ys and returns true iff ys is a substring of xs.

```
function substr(xs, ys) {
    function trial(xs, ys) {
        if (is_null(xs)) { return is_null(ys); }
        else if (is_null(ys)) { return true; }
        else if (head(xs) === head(ys)) {
            return trial(tail(xs), tail(ys));
        } else { return false; }
    }
```

Q7

Substring

Write a function substr(xs, ys) that takes in lists of characters xs and ys and returns true iff ys is a substring of xs.

```
function substr(xs, ys) {
    function trial(xs, ys) {
        if (is_null(xs)) { return is_null(ys); }
        else if (is_null(ys)) { return true; }
        else if (head(xs) === head(ys)) {
            return trial(tail(xs), tail(ys));
        } else { return false; }
    }
    function step(xs) {
        if (is_null(xs)) { return false; }
        else {
            return trial(xs, ys) || step(tail(xs));
        }
    }
```

Q7

Substring

Write a function substr(xs, ys) that takes in lists of characters xs and ys and returns true iff ys is a substring of xs.

```
function substr(xs, ys) {
    function trial(xs, ys) {
        if (is_null(xs)) { return is_null(ys); }
        else if (is_null(ys)) { return true; }
        else if (head(xs) === head(ys)) {
            return trial(tail(xs), tail(ys));
        } else { return false; }
    }
    function step(xs) {
        if (is_null(xs)) { return false; }
        else {
            return trial(xs, ys) || step(tail(xs));
        }
    }
    return step(xs);
}
```