L7: Mutable Data

CS1101S: Programming Methodology

Low Kok Lim

October 1, 2021

Outline

- State (3.1)
- Mutable Data (3.3)

Module Overview

- **Unit 1 — Functions** (textbook Chapter 1)
  - Getting acquainted with the elements of programming, using **functional abstraction**
  - Learning to read programs, and using the **substitution model**
  - Example applications: runes, curves
- **Unit 2 — Data** (textbook Chapter 2)
  - Getting familiar with data: pairs, lists, trees
  - Searching in lists and trees, sorting of lists
  - Example application: sound processing

Module Overview

- **Unit 3 — State** (parts of textbook Chapter 3)
  - Programming with **stateful abstractions**
  - **Mutable data** processing
  - Arrays, loops, searching in and sorting of arrays
  - Reading programs using the **environment model**
  - Example applications: robotics, image/video processing

today

- **Unit 4 — Beyond** (parts of textbook Chapters 3 and 4)
  - Streams
  - Metalinguistic abstraction
  - Computing with Register Machines / Logic Programming / Concurrency

Outline

- State (3.1)
- Mutable Data (3.3)

So far ...

- Our computation has been ***functional***
  - Given same argument, function always returns same result*
  - ***Memoryless*** / ***stateless*** — each function call is independent of the past, and of the future
  - Makes it easy to reason about program behavior
    - Use ***substitution model of evaluation***

Functional Programming

- **Example:**
  - `factorial(5)` always gives 120
    - No matter how many times you call it, or when you call it

- Compare with a **bank account**:
  - Suppose it starts with $100
  - Function `withdraw` returns the balance if there is enough $, otherwise also displays error message

```
withdraw(40); → 60
withdraw(40); → 20
withdraw(40); → 20 "Insufficient funds"
withdraw(15); → 5
```

State

- Identical calls to `withdraw` produce different results

- Bank account has "**memory**"
  - It remembers something about the past
  - It has ***state***

- Functional programming does not allow our programs to have state
  - We need to use ***assignment***

Simple Bank Account — Using Assignment

[Show in Playground]

```
function make_account(initial_balance) {
    let balance = initial_balance;

    function withdraw(amount) {
        if (balance >= amount) {
            balance = balance - amount;
            return balance;
        } else {
            display("Insufficient funds");
            return balance;
        }
    }
    return withdraw;
}

const W1 = make_account(100);
W1(40); ➔ 60
W1(40); ➔ 20
W1(40); ➔ 20 "Insufficient funds"
```

Simple Bank Account — Functional Approach

```
function fn_make_account(initial_balance) {
    const balance = initial_balance;

    function withdraw(amount) {
        if (balance >= amount) {
            return fn_make_account(balance - amount);
        } else {
            display("Insufficient funds");
            return fn_make_account(balance);
        }
    }
    return withdraw;
}

const W1 = fn_make_account(100);
const W2 = W1(40); → fn_make_account(60)
const W3 = W2(40); → fn_make_account(20)
const W4 = W3(40); → fn_make_account(20) "Insufficient funds"
```

[Show in Playground]

Variable Declaration Statement

```
let name = expression;
```

- Declares a ***variable*** *`name`* in the current scope and initializes its value to the value of *`expression`*
- From now on, *`name`* will evaluate to the value of *`expression`*
- Note that from Source §3 onwards, **function parameters** are **variables**

Assignment Statement

```
name = expression;
```

- *`name`* is a **variable**; not evaluated
- *`expression`* is evaluated, then its value is ***assigned*** to the variable *`name`*
- From now on, *`name`* will evaluate to the value of *`expression`*

Example

```
let balance = 100;

balance; → 100

balance = balance - 20;

balance; → 80

balance = balance - 20;

balance; → 60
```

Multiple Accounts

```
const W1 = make_account(100);
const W2 = make_account(100);

W1(50); → 50
W2(70); → 30
W2(40); → 30 "Insufficient funds"
W1(40); → 10
```

- `W1` and `W2` are completely **independent**
  - Each has its own state variable `balance`
  - Withdrawals from one do not affect the other

Assignment: Pros

- Assignment allows us to create objects with ***state***
- State allows objects to behave differently over time

Assignment: Cons

- Harder to reason about programs
  - Harder to debug
  - Harder to verify correctness

- **Substitution model of evaluation breaks down!**
  - Not powerful enough to explain state
  - Need a more sophisticated model — ***Environment Model***

Substitution Model Breaks Down

- Consider

```
function make_simplified_withdraw(balance) {
    return amount => {
        balance = balance - amount;
        return balance;
    }
}
```

- Use **substitution model** to evaluate

```
(make_simplified_withdraw(25))(20);
```

Substitution Model Breaks Down

- Use substitution model to evaluate

```
(make_simplified_withdraw(25))(20);

(amount => { balance = 25 - amount; return 25; })(20);

balance = 25 - 20; return 25; // WRONG!
```

- It returns `25`, which is wrong!

Why Substitution Model Breaks Down?

- Substitution model considers a constant/variable as **just a name for a value**
  - Its value will not change
  - Therefore, one can be substituted for the other

- But **assignment** considers a variable as a **“container” holding a value**
  - The contents of the container may be **changed over time**
  - The container is maintained in a structure called an ***environment***

Outline

- State (3.1)
- Mutable Data (3.3)

Mutable Data

- **Assignment** gives us the ability to create ***mutable*** data, i.e. data that can be modified
  - E.g. bank account

- In Source §1 and §2, all our data were ***immutable***. We had
  - Constructors, selectors, predicates, printers
  - But no ***mutators***

Mutable Pairs

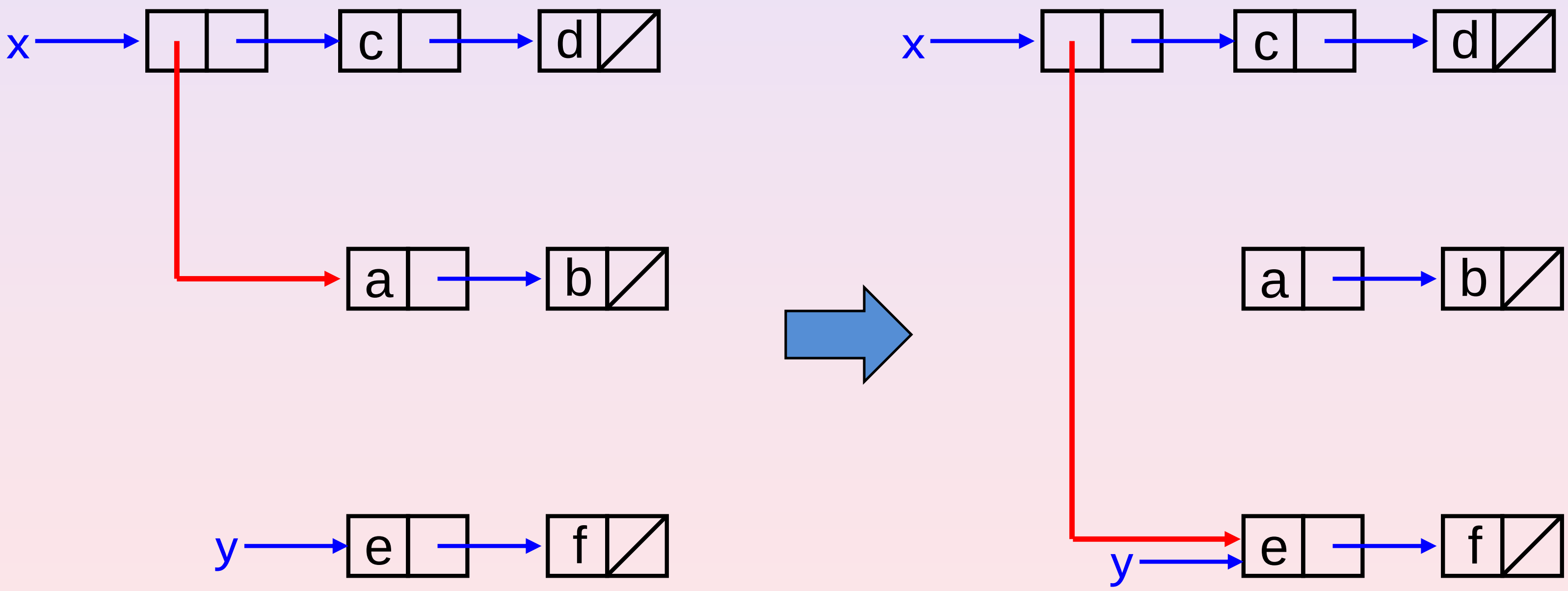
- Now we will allow ***mutators*** in order to create ***mutable data structures***

- After creating a pair with `pair`
  - The **head** can be changed using **`set_head`**
  - The **tail** can be changed using **`set_tail`**

- Mutating mutable pairs
  - `set_head(p, x)` changes **head** of pair `p` to `x`
  - `set_tail(p, x)` changes **tail** of pair `p` to `x`

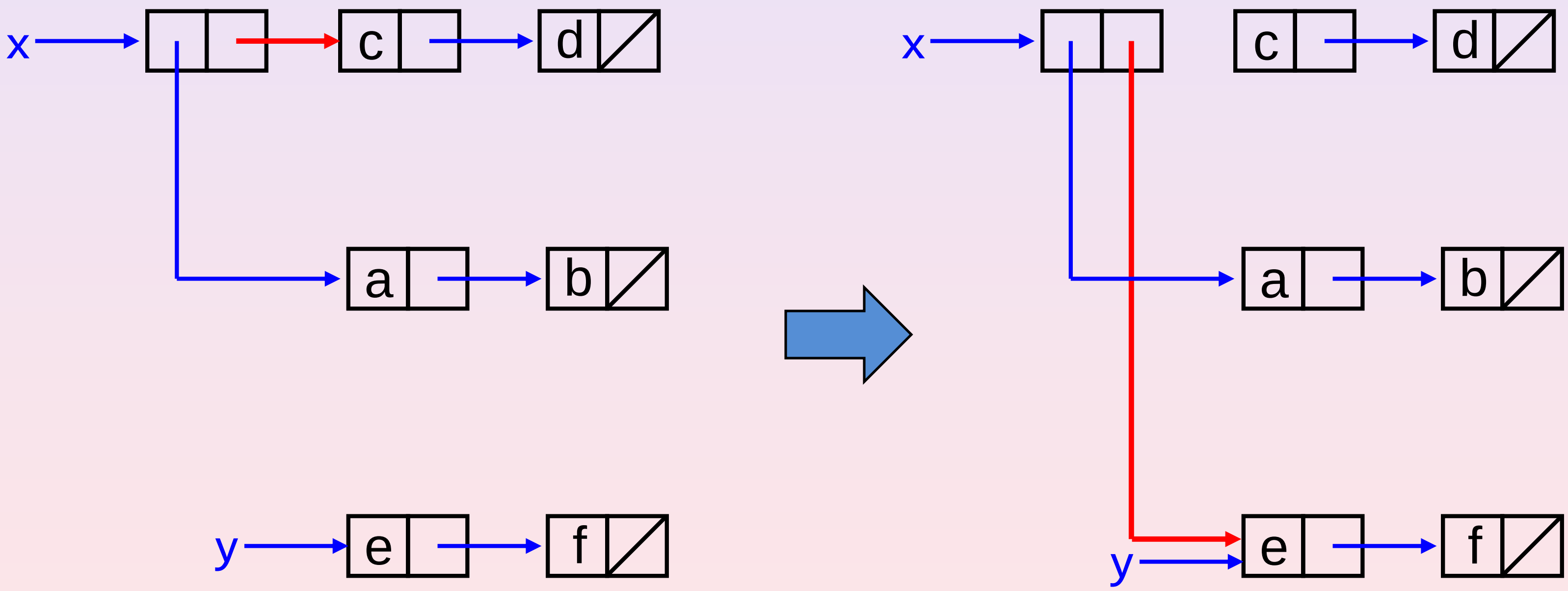
set_head Example

- Effect of `set_head(x, y)`



`set_tail` Example

- Effect of `set_tail(x, y)`



Be Careful with Mutators!

- **Example:**

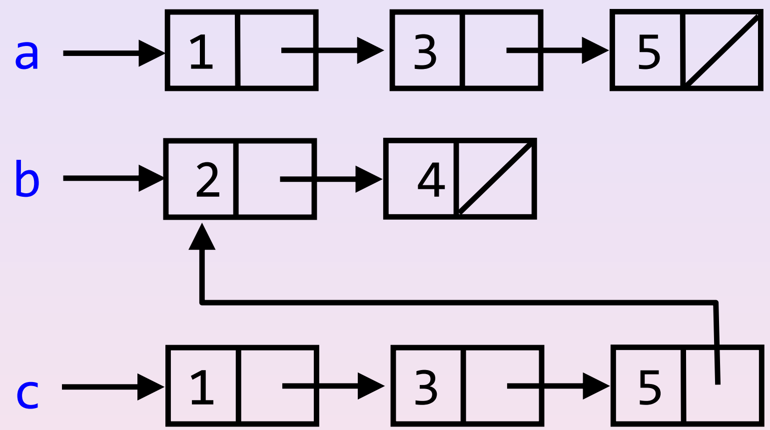
```
const a = list(1, 3, 5);
const b = list(2, 4);
const c = append(a, b);
c; → [1, [3, [5, [2, [4, null]]]]]

set_head(b, 9);
b; → [9, [4, null]
c; → [1, [3, [5, [9, [4, null]]]]]
```

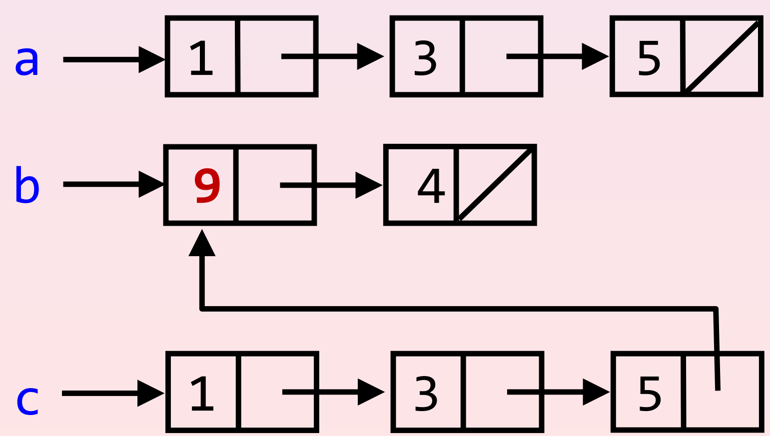
- Mutating b changes c as well !!!
- What is happening?

Mutation and Sharing

- **Before** `set_head(b, 9)`



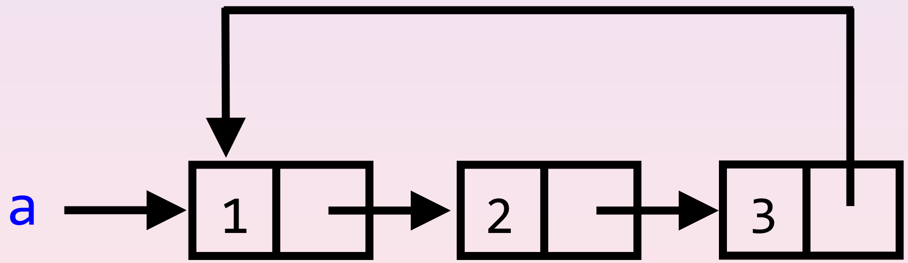
- **After** `set_head(b, 9)`



Be Careful with Mutators!

- Another example:

```
const a = list(1, 2, 3);
set_tail(tail(tail(a)), a);
```



Cyclical structure!

- What is `length(a)`?!

Mutable (“Destructive”) List Processing — Append

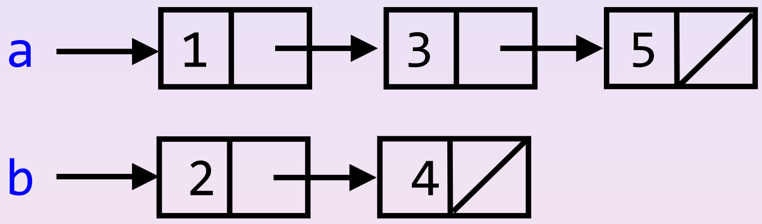
- **Wanted:**

  A function to **append** two lists and return the result list
  - **No new pair must be created**
  - Result list is constructed from existing pairs of input lists

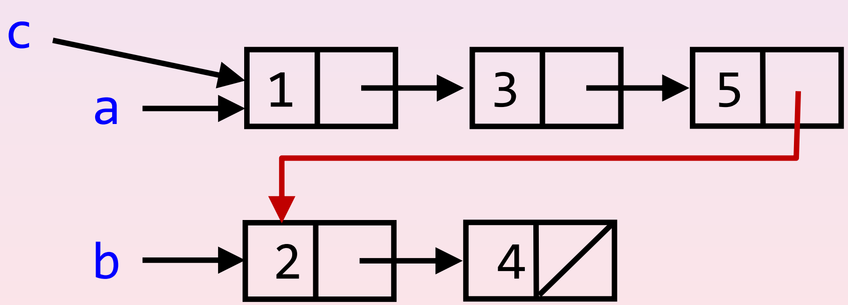
“Destructive” Append

- **Example:**

```
const a = list(1, 3, 5);
const b = list(2, 4);
```




```
const c = d_append(a, b);
```



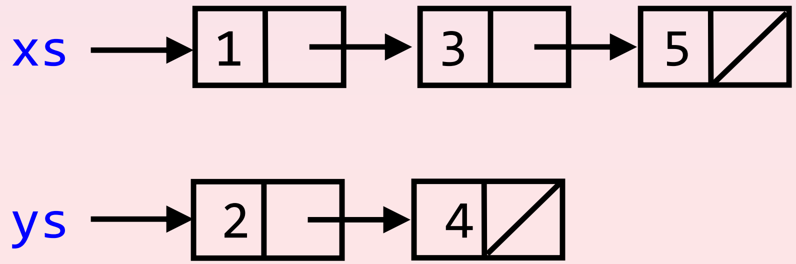
c; ➔ [1, [3, [5, [2, [4, null]]]]]
a; ➔ [1, [3, [5, [2, [4, null]]]]]
b; ➔ [2, [4, null]]

"Destructive" Append

- **Implementation:**

[Show in Playground]

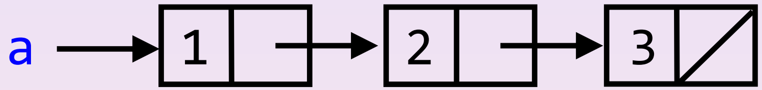
```
function d_append(xs, ys) {
    if (is_null(xs)) {
        return ys;
    } else {
        set_tail(xs, d_append(tail(xs), ys));
        return xs;
    }
}
```



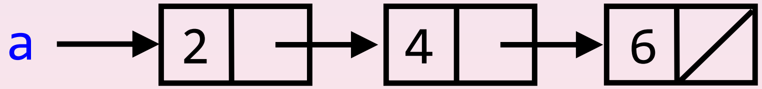
“Destructive” Map

- **Example:**

```
const a = list(1, 2, 3);
```




```
d_map(x => x * 2, a);
```



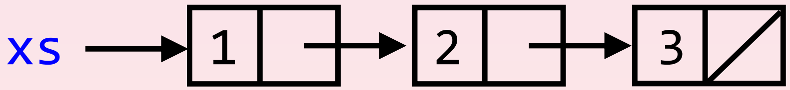

```
a; ➔ [2, [4, [6, []]]]
```

“Destructive” Map

- **Implementation:**

```
function d_map(fun, xs) {
    if (!is_null(xs)) {
        set_head(xs, fun(head(xs)));
        d_map(fun, tail(xs));
    } else { }
}
```

[Show in Playground]



Summary

- **Assignment** allows us to create **state**
- **Assignment** allows us to create **mutable data** structures
- **Substitution model** breaks down with **assignment**
- `set_head` and `set_tail` mutate **pairs** & **lists**
- Be careful when mutating things!