Mutability and advanced control structures

Jia Xiaodong

October 11, 2021

Two models of computing *

- What can and cannot be computed?

Two models of computing *

- What can and cannot be computed?
- Church, 1936: λ-calculus.

Two models of computing *

- What can and cannot be computed?
- Church, 1936: λ-calculus.
  - Functions can be defined.

Two models of computing *

- What can and cannot be computed?
- Church, 1936: λ-calculus.
  - Functions can be defined.
  - Functions can be called.

Two models of computing *

- What can and cannot be computed?
- Church, 1936: λ-calculus.
  - Functions can be defined.
  - Functions can be called.
- Turing, 1936 (later): Turing machines.

Two models of computing *

- What can and cannot be computed?
- Church, 1936: λ-calculus.
  - Functions can be defined.
  - Functions can be called.
- Turing, 1936 (later): Turing machines.
  - A machine he devised to abstract "computation through a purely mechanical process".

Two models of computing *

- What can and cannot be computed?
- Church, 1936: λ-calculus.
  - Functions can be defined.
  - Functions can be called.
- Turing, 1936 (later): Turing machines.
  - A machine he devised to abstract "computation through a purely mechanical process".
  - Consists of a memory tape, a head, a machine state, and a function that makes decisions.

What have we been doing? *

- We have been using a functional programming style, in spirit of λ-calculus.

What have we been doing? *

- We have been using a functional programming style, in spirit of λ-calculus.
- Computers nowadays mostly are in the style of the von Neumann architecture.

What have we been doing? *

- We have been using a functional programming style, in spirit of λ-calculus.
- Computers nowadays mostly are in the style of the von Neumann architecture.
- Most programming languages have a way to mutate data.

What have we been doing? *

- We have been using a functional programming style, in spirit of λ-calculus.
- Computers nowadays mostly are in the style of the von Neumann architecture.
- Most programming languages have a way to mutate data.
- Most data structures and algorithms also mutate data.

What have we been doing? *

- We have been using a functional programming style, in spirit of λ-calculus.
- Computers nowadays mostly are in the style of the von Neumann architecture.
- Most programming languages have a way to mutate data.
- Most data structures and algorithms also mutate data.
- We would also like to mutate data, then.

Mutability

- `const` constant declaration.

[1] See lecture notes for pitfalls.

Mutability

- `const` constant declaration.
- `let` *variable* declaration.

[1] See lecture notes for pitfalls.

Mutability

- `const` constant declaration.
- `let` *variable* declaration.
- Mutability leads to more complicated reasoning.[1]. Yet if done right, is more "natural" to work with.

[1] See lecture notes for pitfalls.

Arrays

The Array

The array is initialized with a list of comma delimited items within square brackets. The items are 0-indexed. Elements can be accessed with the dereference operator `[]`.

Arrays

The Array

The array is initialized with a list of comma delimited items within square brackets. The items are 0-indexed. Elements can be accessed with the dereference operator `[]`.

Example

```
let arr = [1, 2, 3, 4];
```

Q: `arr[0] = ?, arr[1] = ?, arr[2] = ?`

Arrays

The Array

The array is initialized with a list of comma delimited items within square brackets. The items are 0-indexed. Elements can be accessed with the dereference operator `[]`.

Example

```
let arr = [1, 2, 3, 4];
arr[0] = 1, arr[1] = 2, arr[2] = 3
```

Arrays

The Array

The array is initialized with a list of comma delimited items within square brackets. The items are 0-indexed. Elements can be accessed with the dereference operator `[]`.

Example

```
let arr = [1, 2, 3, 4];
```

`arr[0] = 1`, `arr[1] = 2`, `arr[2] = 3`

Q: `arr[4] = ?`

Arrays

The Array

The array is initialized with a list of comma delimited items within square brackets. The items are 0-indexed. Elements can be accessed with the dereference operator `[]`.

Example

```
let arr = [1, 2, 3, 4];
arr[0] = 1, arr[1] = 2, arr[2] = 3
arr[4] = undefined
```

Arrays

Insertion

We can insert or modify any element by simply dereferencing it and setting it.

Arrays

Insertion

We can insert or modify any element by simply dereferencing it and setting it.

Example

```
let arr = [1, 2, 3, 4];
arr[3] = 3; arr[4] = 4; arr[6] = 6;
```

Q: `arr = ?`

Arrays

Insertion

We can insert or modify any element by simply dereferencing it and setting it.

Example

```
let arr = [1, 2, 3, 4];
arr[3] = 3; arr[4] = 4; arr[6] = 6;
```

```
[1, 2, 3, 3, 4, undefined, 6]
```

While loops

Definition

A while loop is made as such:

```
while (expression) {
    statements
}
```

This is something like `expression ? statements : undefined`, over and over again.

For loops

Definition

A for loop is made as such:

```
for (expr a; expr b; expr c) {
    statements
}
```

This is like

```
expr a;
while(expr b;) {
    expr c;
    statements;
}
```

For loops

Definition

A for loop is made as such:

```
for (expr a; expr b; expr c) {
    statements
}
```

This is like

```
expr a;
while(expr b;) {
    expr c;
    statements;
}
```

Traversing arrays

```
for (let i = 0; i < array_length(arr); i = i + 1) {...}
```

Loop controls

Control statements

If you wish to escape the closest loop prematurely, use `break`. If you wish to skip one iteration immediately, use `continue`.

Question

Two nested loops:

```
while(...) {
    while(...) {
        //I am here
    }
}
```

How to break out of both loops?

Crisis

Substitution model does not work any more. We need a new way to keep track of our names!

Remember that time...

- Every new context creates a new *scope*.
- The most common context are blocks: ....
- Names in an inner scope inherit those defined outside of it.
- Names can be overridden by definitions in the current scope.
- You cannot go "into" an inner scope from an outer scope to retrieve definitions!
- **In conclusion:** To find what a name refers to, look at the current scope, and then outwards. Take the first one you come across.

Environment Model

- We use environments (boxes) to denote our scopes.

Environment Model

- We use environments (boxes) to denote our scopes.
- Name definitions add an entry to the environment they are in.

Environment Model

- We use environments (boxes) to denote our scopes.
- Name definitions add an entry to the environment they are in.
- Functions have some weird syntax (refer to lecture notes).

Environment Model

- We use environments (boxes) to denote our scopes.
- Name definitions add an entry to the environment they are in.
- Functions have some weird syntax (refer to lecture notes).
- New scopes create child environments.

Environment Model

- We use environments (boxes) to denote our scopes.
- Name definitions add an entry to the environment they are in.
- Functions have some weird syntax (refer to lecture notes).
- New scopes create child environments.
  - All whole program resides in the program environment.

Environment Model

- We use environments (boxes) to denote our scopes.
- Name definitions add an entry to the environment they are in.
- Functions have some weird syntax (refer to lecture notes).
- New scopes create child environments.
  - All whole program resides in the program environment.
  - The parent is the closest *enclosing environment*.

Environment Model

- We use environments (boxes) to denote our scopes.
- Name definitions add an entry to the environment they are in.
- Functions have some weird syntax (refer to lecture notes).
- New scopes create child environments.
  - All whole program resides in the program environment.
  - The parent is the closest *enclosing environment*.
- Duplicate names in the same environment cannot exist.

Environment Model

- We use environments (boxes) to denote our scopes.
- Name definitions add an entry to the environment they are in.
- Functions have some weird syntax (refer to lecture notes).
- New scopes create child environments.
  - All whole program resides in the program environment.
  - The parent is the closest *enclosing environment*.
- Duplicate names in the same environment cannot exist.
  - However, they can be overwritten depending on the statement.

Environment Model

- We use environments (boxes) to denote our scopes.
- Name definitions add an entry to the environment they are in.
- Functions have some weird syntax (refer to lecture notes).
- New scopes create child environments.
  - All whole program resides in the program environment.
  - The parent is the closest *enclosing environment*.
- Duplicate names in the same environment cannot exist.
  - However, they can be overwritten depending on the statement.
- Environments are never destroyed.

Environment Model

- We use environments (boxes) to denote our scopes.
- Name definitions add an entry to the environment they are in.
- Functions have some weird syntax (refer to lecture notes).
- New scopes create child environments.
  - All whole program resides in the program environment.
  - The parent is the closest *enclosing environment*.
- Duplicate names in the same environment cannot exist.
  - However, they can be overwritten depending on the statement.
- Environments are never destroyed.
- When evaluating a name, search outwards starting from the current environment. First match is returned. Otherwise, invalid name.

S9 Q1

```
function change(x, new_value) {
    x = new_value;
}
let x = 0;
change(x, 1);
```

What is the value of `x` after evaluation?

S9 Q1

```
function change(x, new_value) {
    x = new_value;
}
let x = 0;
change(x, 1);
```

What is the value of `x` after evaluation?

`x = 0`.

S9 Q2

Write `d_filter(xs)` that acts like `filter` but modifies the list `xs` in place.

S9 Q2

Write `d_filter(xs)` that acts like `filter` but modifies the list `xs` in place.

```
function d_filter(pred, xs) {




}
```

S9 Q2

Write `d_filter(xs)` that acts like `filter` but modifies the list `xs` in place.

```
function d_filter(pred, xs) {
    if (is_null(xs)) {
        return xs;
    }




}
```

S9 Q2

Write `d_filter(xs)` that acts like `filter` but modifies the list `xs` in place.

```
function d_filter(pred, xs) {
    if (is_null(xs)) {
        return xs;
    }
    else if (pred(head(xs))) {
        set_tail(xs, d_filter(pred, tail(xs)));
        return xs;
    }



}
```

S9 Q2

Write `d_filter(xs)` that acts like `filter` but modifies the list `xs` in place.

```
function d_filter(pred, xs) {
    if (is_null(xs)) {
        return xs;
    }
    else if (pred(head(xs))) {
        set_tail(xs, d_filter(pred, tail(xs)));
        return xs;
    }
    else {
        return d_filter(pred, tail(xs));
    }
}
```

S9 Q3

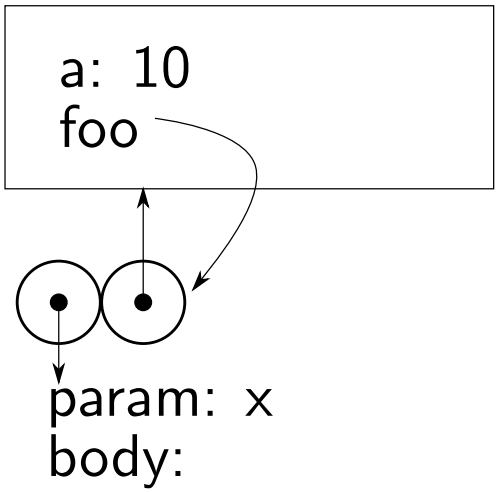
Draw the environment at the breakpoints.

```
let a = 10;
function foo(x) {
    let b = 0;
    function goo(x) {
        let a = 30;
    }
    if (x <= 2) {
        a = a + x;
        b = b + x;
        // Breakpoint #4
    } else {
        // Breakpoint #3
        goo(x - 1);
    }
    a = a + x;
    b = b + x;
    // Breakpoint #2
    goo(3);
}
// Breakpoint #1
foo(1);
// Breakpoint #5
```

S9 Q3

Breakpoint 1

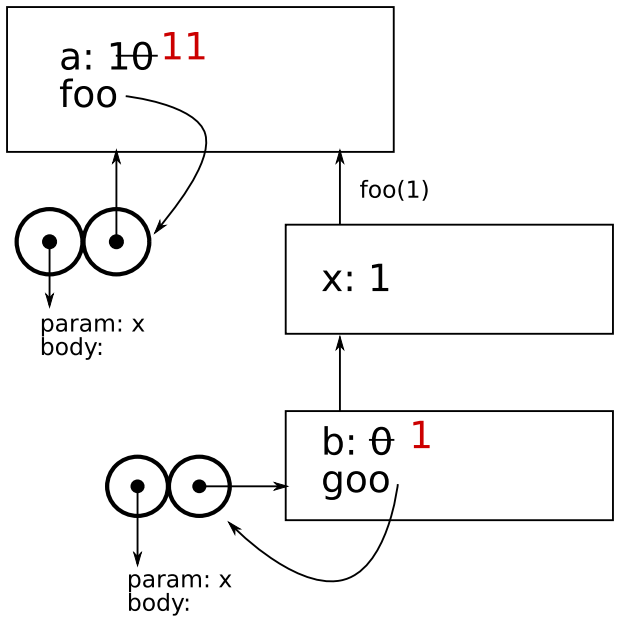
```
let a = 10;
function foo(x) {
    let b = 0;
    function goo(x) {
        let a = 30;
    }
    if (x <= 2) {
        a = a + x;
        b = b + x;
    } else {
        goo(x - 1);
    }
    a = a + x;
    b = b + x;
    goo(3);
}
// Breakpoint #1
foo(1);
```



S9 Q3

Breakpoint 2

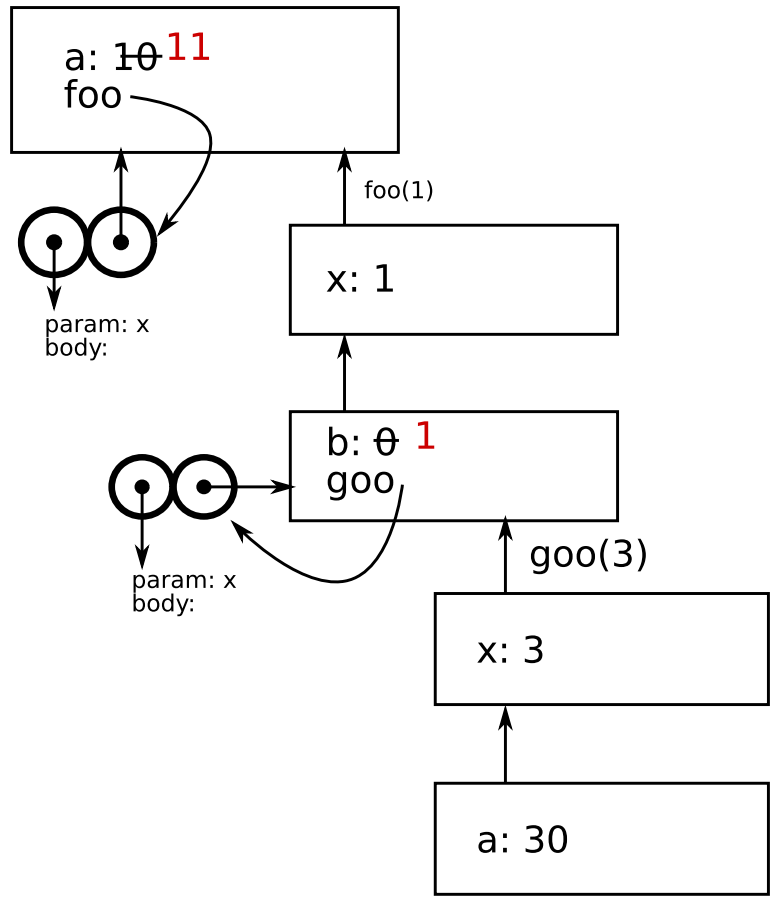
```
let a = 10;
function foo(x) {
    let b = 0;
    function goo(x) {
        let a = 30;
    }
    if (x <= 2) {
        a = a + x;
        b = b + x;
    } else {
        goo(x - 1);
    }
    a = a + x;
    b = b + x;
    // Breakpoint #2
    goo(3);
}
foo(1);
```



S9 Q3

Breakpoint 3

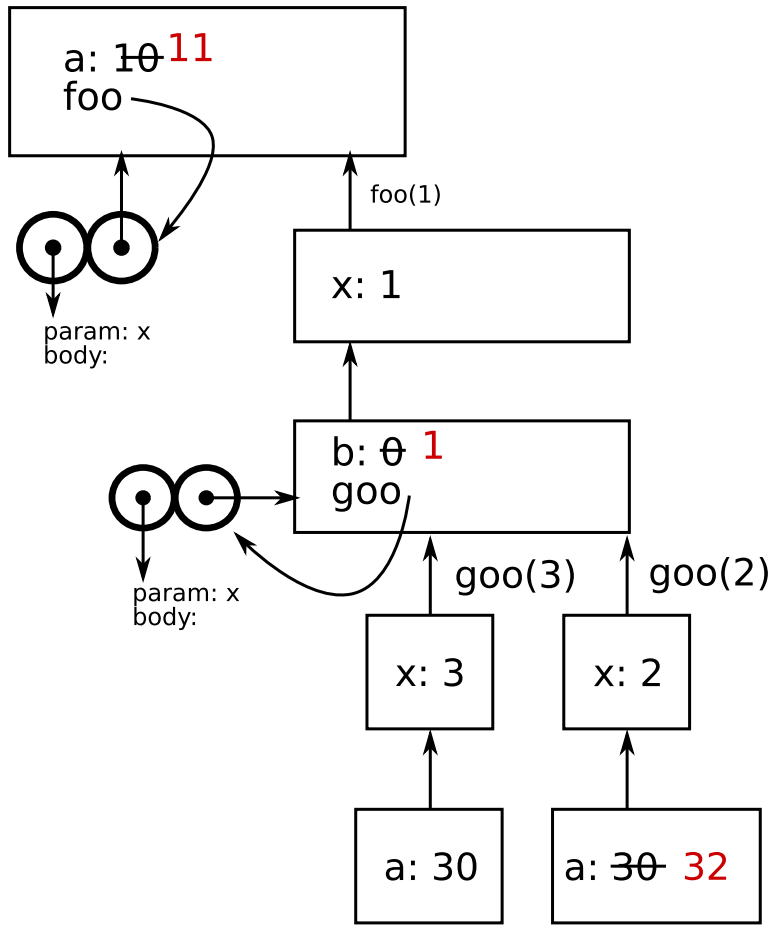
```
let a = 10;
function foo(x) {
    let b = 0;
    function goo(x) {
        let a = 30;
    }
    if (x <= 2) {
        a = a + x;
        b = b + x;
    } else {
        // Breakpoint #3
        goo(x - 1);
    }
    a = a + x;
    b = b + x;
    goo(3);
}
foo(1);
```



S9 Q3

Breakpoint 4

```
let a = 10;
function foo(x) {
    let b = 0;
    function goo(x) {
        let a = 30;
    }
    if (x <= 2) {
        a = a + x;
        b = b + x;
        // Breakpoint #4
    } else {
        goo(x - 1);
    }
    a = a + x;
    b = b + x;
    goo(3);
}
foo(1);
```



Q1

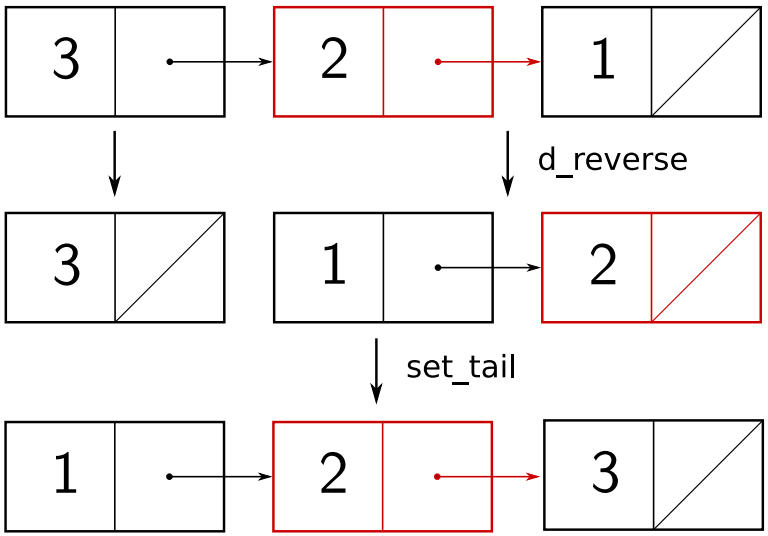
Write `d_reverse(xs)`, just like `d_filter` you did just now.

```
const L = list(1, 2, 3, 4, 5, 6);
d_reverse(L); // returns [6, [5, [4, [3, [2, [1, null]]]]]]
```

Q1

Write `d_reverse(xs)`, just like `d_filter` you did just now.

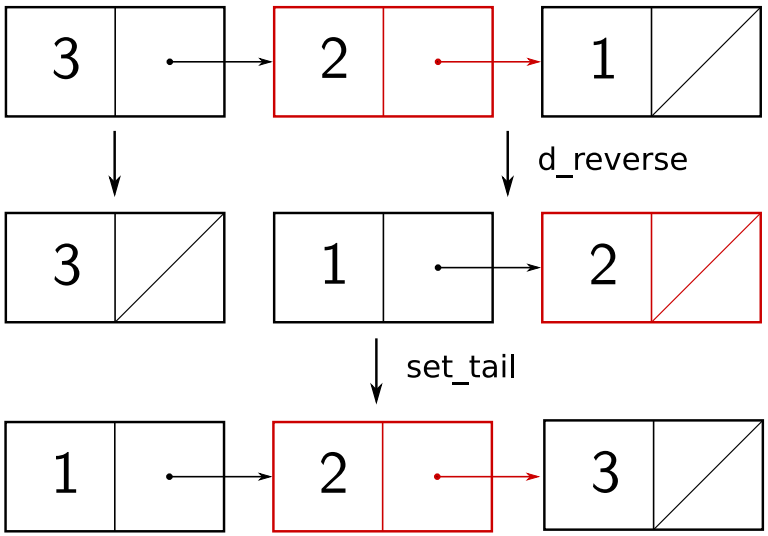
```
const L = list(1, 2, 3, 4, 5, 6);
d_reverse(L); // returns [6, [5, [4, [3, [2, [1, null]]]]]]
```



Q1

Write `d_reverse(xs)`, just like `d_filter` you did just now.

```
const L = list(1, 2, 3, 4, 5, 6);
d_reverse(L); // returns [6, [5, [4, [3, [2, [1, null]]]]]]
```




```
function d_reverse(xs) {
    if (is_null(xs) || is_null(tail(xs))) {
        return xs;
    }
    else {
        const temp = d_reverse(tail(xs));
        set_tail(tail(xs), xs);
        set_tail(xs, null);
        return temp;
    }
}
```

Q2

Ex. 3.16

Consider the following function that counts the number of pairs in a list. Give examples of lists made of 3 pairs that will cause the function to return 3, 4, 7, or never return at all.

```
function count_pairs(x) {
    if (!is_pair(x)) {
        return 0;
    } else {
        return 1 + count_pairs(head(x)) + count_pairs(tail(x));
    }
}
```

Q2

Ex. 3.16

Consider the following function that counts the number of pairs in a list. Give examples of lists made of 3 pairs that will cause the function to return 3, 4, 7, or never return at all.

```
function count_pairs(x) {
    if (!is_pair(x)) {
        return 0;
    } else {
        return 1 + count_pairs(head(x)) + count_pairs(tail(x));
    }
}
```

```
// returns 3
const three = list(1, 2, 3);
count_pairs(three);
```

Q2

Ex. 3.16

Consider the following function that counts the number of pairs in a list. Give examples of lists made of 3 pairs that will cause the function to return 3, 4, 7, or never return at all.

```
function count_pairs(x) {
    if (!is_pair(x)) {
        return 0;
    } else {
        return 1 + count_pairs(head(x)) + count_pairs(tail(x));
    }
}
```

```
// returns 3
const three = list(1, 2, 3);
count_pairs(three);

// infinite loop
const loop = list(1, 2, 3);
set_tail(tail(tail(loop)), loop);
count_pairs(loop);
```

```
// returns 4
const four_a = pair(null, null);
const four_b = pair(four_a, four_a);
const four = pair(four_b, null);
count_pairs(four);
```

Q2

Ex. 3.16

Consider the following function that counts the number of pairs in a list. Give examples of lists made of 3 pairs that will cause the function to return 3, 4, 7, or never return at all.

```
function count_pairs(x) {
    if (!is_pair(x)) {
        return 0;
    } else {
        return 1 + count_pairs(head(x)) + count_pairs(tail(x));
    }
}
```

```
// returns 3
const three = list(1, 2, 3);
count_pairs(three);

// infinite loop
const loop = list(1, 2, 3);
set_tail(tail(tail(loop)), loop);
count_pairs(loop);
```

```
// returns 4
const four_a = pair(null, null);
const four_b = pair(four_a, four_a);
const four = pair(four_b, null);
count_pairs(four);

// returns 7
const seven_a = pair(null, null);
const seven_b = pair(seven_a, seven_a);
const seven = pair(seven_b, seven_b);
count_pairs(seven);
```