Mutability and advanced control structures

Jia Xiaodong

October 11, 2021

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.

A brief aside that might help explain why we miss out on a "feature" (mutability) in the first place. We have already mentioned primitive recursion before some time ago in another aside. Recursion is an answer to "what can be computed". In 1936 two more models were proposed.

The first is similar to what we have been doing. Functions are the basis of calculation. We have seen a taste of this in Church encoding. In the calculus everything can be created from the ground up from a few simple rules, without needed any "built-in" functions.

The second is the Turing machine, which is more physical. It is a machine in every sense of the word This is the more common model used in CS for matters relating to complexity. The power of the machine comes from the tape. Remove the tape, and the machine will become what is known as a finite automaton. These automata are very weak in comparison.

The two models are equivalent. In fact a proposal is that all reasonable methods of computation are equivalent in power to the Turing machine.

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.

What we have been doing so far is fairly similar to the λ-calculus. However most computers do not operate in this fashion. Very roughly, modern computers consist of a processor that reads and modifies memory according to instructions, which are also stored in memory. This is the von Neumann architecture. This is similar to the Turing machine. There once existed machines (e.g. LISP machines) that operated in other fashions, but nowadays they are rare.

Most programming languages have a way to mutate (change) data. If you pick up a programming 101 book in any other language (including JS) the first few chapters will introduce you to this already. Most data structures and algorithms also mutate data. We have also seen that some things are either troublesome, or become slower if we do not allow mutation. Mutation seems to be intertwined with our daily experience and problem solving.

Therefore we would also like to try this out.

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.

We introduce a new keyword `let` that allows mutation. In other words, we can point the name at something else. `const` does not allowing reassignment.

The reason why this is added later is because it is harder to reason about your programs when things are being mutated everywhere. For example one day if your screen goes blank, you do not know if it's your graphics card that failed, the drivers that failed, the port that failed, the wire that failed, the application that failed, the OS that failed, the screen that failed, since they are all changing (mutating) the state of screen. Yet is is quite necessary to do something to the screen if we are to hope to display anything on it.

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

Q: `arr[4] = ?` `arr[4] = undefined`

The array is created by writing its elements down between square brackets, kind of like the same fashion as a list. The elements are 0-indexed, meaning the first element is given the index of 0. The reason for this is perhaps more easily seen in other languages such as C, where arrays are just an address to a contiguous chunk of memory and the first cell is the address + 0.

The square brackets has 2 uses, both as a way to declare arrays and as a way to access elements of arrays. Accessing elements that do not exist result in `undefined`.

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

The square brackets also allow us to set any element.

In fact, as this example demonstrates, the elements do not even need to be contiguous. However one thing is that the index must be a natural number.

While loops

Definition

A while loop is made as such:

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

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

The while loop checks the expression in brackets, and if it is true, then it executes the statements. Otherwise it returns `undefined` and the program carries on. Please be careful with your reasoning when using loops. It is very common to commit off-by-one errors where the loop runs either an extra step or one step too little especially due to fumbling between things like `<` and `<=`.

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) {...}
```


A for loop is a more compact while loop since most of the time in a loop you want to (a) initialize a counter, (b) check the conditions, and (c) modify the counter.

The example demonstrates it in action. We count from `i = 0` until `i = array_length(arr) - 1`, which allows us to operate on the entire array element by element using `arr[i]` in the body.

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?

It is not very recommended to use these controls all the time since it makes things hard to reason about. There are cases where they are useful though. For example, an infinite loop waiting for things to happen.

In any case, there is a limitation in that they only apply to the closest enclosing loop. To break out of two loops at once, you will have to use a guard variable or something like that. What this means is to have the outer loop test for `if (notgood) { break; }` and the inner loop set `notgood = true; break;`.

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.

These are the scoping rules we have covered previously.

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.

The environment rules are the same as the scoping rules we have previously covered. The only addition is the drawing of environments. This is more straightforward than you think it is. A good way to practice is to come up with complicated programs (perhaps those in midterms or RAs) and draw them, and check against the visualiser in SA.

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`.

Using the environment model it should be clear that the `x` inside and outside the function body are two different things. In the environment model, calling a function will create a new frame with the correct values assigned to parameters.

This is slightly more complicated. We will be seeing more of this in the future. You can try writing a similar function using lists. Does it modify the argument that way?

If you are interested, you can look up *call by value* and *call by reference*. This means, on a function call, sometimes parameters are assigned values (like numbers), and sometimes parameters are assigned references (like arrows to a piece of data, like a list).

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));
    }
}
```


This is very similar to the actual definition of `filter`, just that we add in `set_tail`.

S9 Q3

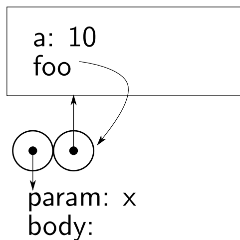
Draw the environment at the breakpoints.

```
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

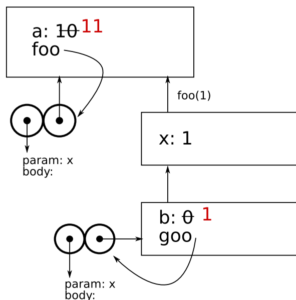
```
let a = 10;
function foo(x) {
    let b = 0;
    function goo(x) {
        let a = 30;
    }
    if (x <= 2) {
        a = a + x;
        b = b + x;
    } else {
        goo(x - 1);
    }
    a = a + x;
    b = b + x;
    goo(3);
}
// Breakpoint #1
foo(1);
```



S9 Q3

Breakpoint 2

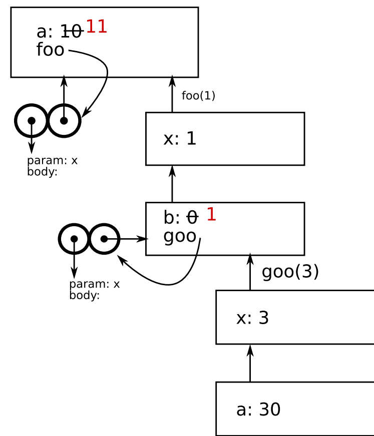
```
let a = 10;
function foo(x) {
    let b = 0;
    function goo(x) {
        let a = 30;
    }
    if (x <= 2) {
        a = a + x;
        b = b + x;
    } else {
        goo(x - 1);
    }
    a = a + x;
    b = b + x;
    // Breakpoint #2
    goo(3);
}
foo(1);
```



S9 Q3

Breakpoint 3

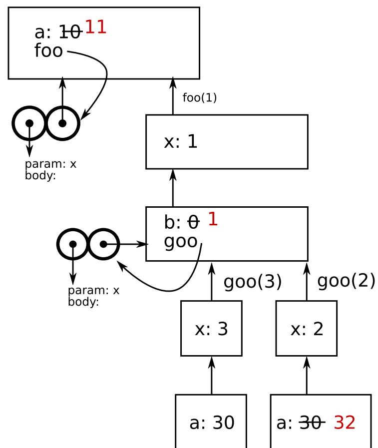
```
let a = 10;
function foo(x) {
    let b = 0;
    function goo(x) {
        let a = 30;
    }
    if (x <= 2) {
        a = a + x;
        b = b + x;
    } else {
        // Breakpoint #3
        goo(x - 1);
    }
    a = a + x;
    b = b + x;
    goo(3);
}
foo(1);
```



S9 Q3

Breakpoint 4

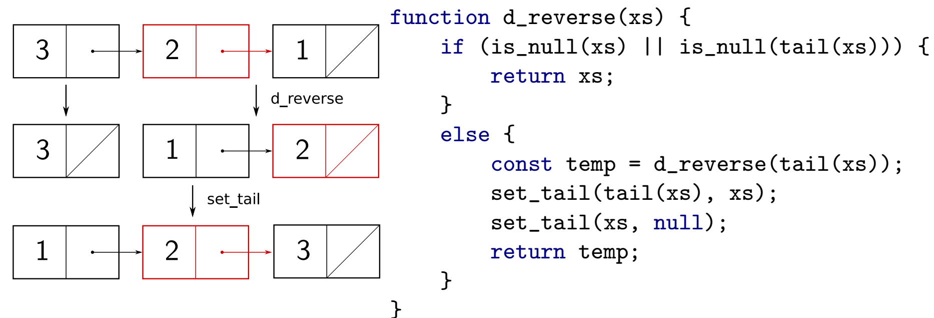
```
let a = 10;
function foo(x) {
    let b = 0;
    function goo(x) {
        let a = 30;
    }
    if (x <= 2) {
        a = a + x;
        b = b + x;
        // Breakpoint #4
    } else {
        goo(x - 1);
    }
    a = a + x;
    b = b + x;
    goo(3);
}
foo(1);
```



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]]]]]]
```




```
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;
    }
}
```


The simplest way of doing this is actually with something like `append`. You would traverse the whole list and `set_tail` on the last element with the current head. However this is very slow. A better solution comes with the observation that if you keep track of the first item in the tail, after reversing the tail, it will become the last item. So that is how we can append the head to the last item in the list without traversing the entire list.

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);
```


The problem with this procedure is of course it does not know if it has counted a pair previously. The scenario where the function never terminates is the easiest — we just create a list that points back at itself, which will cause an infinite loop. It is a good exercise to draw these lists out and trace the execution of the function.