More functions and recursion

Jia Xiaodong

August 30, 2021

Anonymous functions

```
const g = param => { /* body */ }
```

Short aside: difference between *parameter* and *argument*:

- A parameter is what the function depends on. For the above, the parameter of `g` is `param`.
- An argument is what you give the function. For example, `g(5)`, then 5 is the argument.

A very convenient notation for functions, since we will be dealing with plenty of them. They are especially good when it's a simple one liner that is kept anonymous. We will see uses of them later on.

Higher order functions

Remember this?

```
function fact_helper(n, res) {
    return n === 1
        ? res
        : fact_helper(n - 1, n * res);
}

function factorial(n) {
    return fact_helper(n, 1);
}
```


In the previous session we saw this example. For the sake of abstraction we would like to hide the helper function.

Higher order functions

Cont.

Makes more sense?

```
function factorial(n) {
    function fact_helper(n, res) {
        return n === 1
            ? res
            : fact_helper(n - 1, n * res);
    }

    return fact_helper(n, 1);
}
```


The way to do this is to just put it inside the factorial function. Now nobody outside this function can access it. Luckily Source allows us to do this — not all languages support this. But does this actually hide it? How do we know? We shall see that later. Meanwhile we should discuss more of our new abilities with functions.

Higher order functions

Functions as return value

Functions of functions (functionals):

$$I = \int f(t)\,\mathrm{d}t$$

Functions that return functions aren't actually that foreign. It is also not difficult to come up with examples where something like this might be needed.

Higher order functions

Functions as arguments

Say I want the smallest of two things.

```
const min = (a, b) => a < b ? a : b
```

What if I am comparing timings in HH:MM format and I want the earliest?

```
const min = (a, b, f) => f(a) < f(b) ? a : b
```

```
function hhmm_to_mins(a) { ... }
min(a, b, hhmm_to_mins);
```


Passing in functions as arguments is a good way to further abstract behaviour. This is a simple example. An ordinary function to find the minimum of two things is only good for things that can be directly compared: numbers, strings, etc.

By accepting in a function that converts the objects we are comparing into their magnitudes, we can make a more robust `min` function that can compare anything we like, as long as we can quantify them.

Scoping

- We give names to things.
- We may give many things the same name. (e.g. c: Speed of light, specific heat capacity, etc.)
- What gives us the context for our names?

Assigning names to things is a great thing. But sometimes names can collide. In mathematics this might happen since we don't have enough alphabets. When writing our programs bad things happen not when we run out of names, but when we accidentally use the wrong name. This can happen for instance if everyone decides to name their helper functions the same thing. Best case, the program doesn't even run. Worst case, you get bugs and don't even know where they're coming from.

So the program needs to know what the names are pointing to.

Scopes

- *A name occurrence refers to the closest surrounding declaration.*
- Scopes are our context where we find our names.
- The most common context are blocks: `{...}`.
- To find what a name refers to, look at the current scope, and then outwards. Take the first one you come across.
- Names in an outer scope can be hidden by definitions in an inner scope.

The convention for establishing the context for our names is to take the closest definition. In this case, the "closeness" is defined by the number of scopes.

Scopes are formed by blocks. The only exceptions are the global scope, which is just there by default, and the scope of anonymous function parameters and bodies.

To find definitions for our names, we work our way outwards. What this means is that we can't search "into" scopes: there is no way for an outer scope to access the contents of an inner scope. This justifies the factorial example shown previously.

Furthermore, this also means that we can hide names by declaring them again in an inner scope. However, note that you cannot declare the same thing twice in the same scope (at least for now)! Now why would you want to hide names? At times, it is a good bit of safety to do so. Going back to the factorial function, the `fact_helper` declares `n` that hides the parameter `n` of the `factorial` function. This is good safety — the helper function should not have access to the actual parameter, and leave that up to the wrapper function instead.

Exercise

```
hello = "world"
function n(hello){
   const g = hello => display;
   g(hello);
}
n("hello")(hello);
```

```
const n = 1;
{
    const n = 2;
    {
        const n = 3;
        {
            display(n);
        }
        const n = 4;
    }
}
```

The first example makes use of the fact that Source treats functions as first-class citizens. So you can return functions, take functions in as arguments, etc. just like any other variable. The naming in this example is quite poor to cause some confusion. `n("hello")` passes `"hello"` to function `n` as the parameter `hello`. Now remember that the `hello` of `g` is completely independent! So is that in the global scope. So when we call `g(hello)`, we actually call it with `g("hello")`. This causes `g` to return `display` which may or may not be what you expected...

The second example shows that in Source, programs aren't just interpreted blindly from top to down. If that were so, then `display` would fire at least once. But it does not, and an error is given immediately as it detects the redeclaration of names.

Example: series *

Let us make a polynomial series generator. A series is something like

$$S(x) = \sum_{n=0}^{k} a_n x^n = a_0 + a_1 x + a_2 x^2 + \cdots$$

A disposable solution:

```
function sum(x) {
    return a0 + a1 * x + a2 * x * x + ...
}
```


Let us consider a more useful problem. Most of you might be familiar with series expansions of functions. So let us try to make an all-purpose series generator. Of course the naive approach is to just hard code everything, but this would be only good for one purpose. We want something more robust.

How do we do this? First of all, of course we need to terminate the sum at k. Next, the coefficients have to be created dynamically and not hard coded. The perfect solution would be to accept a function passed in as an argument. Finally, we don't want to evaluate the generator again every time we feed in a different x. Optimally we want the generator to return a function like `sum` so that we don't have to regenerate the entire series.

Example: series *

Cont.

```
function series_generator(k, coeff) {
    function gen_helper(n, series) {
        return n === k
            ? series
            : gen_helper(n + 1,
                x => series(x) + coeff(n) * math_pow(x, n));
        }
    return gen_helper(0, x => 0);
}
```


So, after our discussion let us write down the form of the generator. For the series to terminate at k we would need some kind of counter. One way to do this is with a helper function. The base case is quite clear here.

Now, how do we add a term to the series? We get into some trouble. Remember that we want `series` to actually be a *function*, so if you just did `1 + series` it would be wrong. So we have to redefine `series` to be an extension of the old `series`. That's it!

How do we call the helper? What are the default arguments? Well we start from $k = 0$ and the series defaults to 0.

Example: series *

Demonstration

$$e^x \approx \sum_{n=0}^{k} \frac{x^n}{n!}$$

```
function exp_coeff(n) {
    return 1 / factorial(n);
}
const exp_series = series_generator(5, exp_coeff);
```

Try it out!

Let us start with a simple example. The definition of the exponential function is as given. It is fairly easy to create this. Here are the errors with our approximation using 5 terms,

- Delta @ 0.5 : 0.00028377070012819416
- Delta @ 1 : 0.009948495125712054
- Delta @ 1.5 : 0.08325157033806452
- Delta @ 2 : 0.3890560989306504

And using 10 terms.

- Delta @ 0.5 : 2.818771882573401e-10
- Delta @ 1 : 3.0288585284310443e-7
- Delta @ 1.5 : 0.000018363725341252746
- Delta @ 2 : 0.0003435768847959153

Example: series *

Demonstration, cont.

$$\sin(x) \approx \sum_{n=0}^{k} \frac{(-1)^n x^{2n+1}}{(2n+1)!}$$

```
function sin_coeff(n) {
    function minus_one(n) {
        return ((n - 1) / 2) % 2 === 0 ? 1 : -1;
    }
    return n % 2 === 0 ? 0 : minus_one(n) / factorial(n);
}
const sin_series = series_generator(5, sin_coeff);
```

Try it out! (same link as before)

Another example is the sine function, it is usually given in odd powers of x so we have to do a bit of converting to suite our purposes. Note that here if we generate up to the fifth term we actually only get 2 terms due to that fact.

Example: series *

Challenge

Fourier trigonometric series for function f with period $2L$:

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \sin\left(\frac{n\pi x}{L}\right) + \sum_{n=1}^{\infty} b_n \cos\left(\frac{n\pi x}{L}\right)$$

A challenge. So far we have only generated series in terms of polynomials. This is only good for power series. There are plenty of other series. One such example is the Fourier series. The Fourier series expansion can be found for any periodic function in terms of other orthogonal functions. Without getting too much into the mathematics, this means that we forgot to abstract one thing — the function of x. Therefore the challenge is for you to abstract that functionality out.

Primitive recursion *

Definition

The following are primitive recursive:

- Constant function: 0
- Successor function: $S(x) = x + 1$
- Projection function[1]: $P_i(\mathbf{x}) = \mathbf{x}_i$

Recursion: if f, g are primitive recursive, h is primitive recursive if

$$h(0, \mathbf{x}) = f(\mathbf{x})$$
$$h(S(y), \mathbf{x}) = g(y, h(y, \mathbf{x}), \mathbf{x})$$

[1] In subsequent slides $\mathbf{x}$ is the vector of arguments given to the function (i.e. represents $x_1, x_2, \ldots$), and $\mathbf{x}_i$ is the i-th element of the vector (i.e. x_i).

If I asked you what do you think can someone compute given all the resources in the world you would perhaps either say that of course there might be ridiculous questions not even an omnipotent deity can compute. For example the task of enumerating all real numbers is something even a god with infinite time and space will not be able to accomplish.

Anyway our goal here is not to answer that question, but to address what it means to even compute. Of course if you were a god perhaps you would not be able to enumerate the real numbers but you could do some startling things that humans might not be able to do. So what is something to represent a reasonable human computation?

One of the earliest concepts of this came in the form of *primitive recursion*.

Primitive recursion *

Definition

A function f is defined from t by *iteration* if

$$f(\mathbf{x}, n) = t^n(\mathbf{x})$$

Theorem

Minus some formalities, primitive recursion and iteration are equivalent.

Proof.

Iteration is primitive recursion because

$$f(\mathbf{x}, 0) = \mathbf{x}$$
$$f(\mathbf{x}, n+1) = t(f(\mathbf{x}, n))$$

□

Another very simple idea is just doing something over and over again. That is called iteration. The two are pretty much the same.

Iteration can be easily converted to primitive recursion, with h as the identity and g as t.

Primitive recursion *

Proof.

Primitive recursion can be converted into recursion. Take

$$t(\mathbf{x}, n, z) := (\mathbf{x}, n+1, h(\mathbf{x}, n, z))$$

Then

$$(\mathbf{x}, n, f(\mathbf{x}, n)) = t^n(\mathbf{x}, 0, g(\mathbf{x}))$$

Example: factorial

Factorial is defined as follows:

$$f(0) = g := 1$$
$$f(n+1) = h(n, f(n)) := (n+1) \cdot f(n)$$

Continuing with the proof, primitive recursion can also be converted to iteration, minus some details.

An example: factorial. As primitive recursion, factorial seems very familiar to us. Something to note is that we don't need $\mathbf{x}$ here so we omitted it.

Primitive recursion *

Example: factorial

Let us make it iterative. Then using the recipe,

$$t(n, z) := (n + 1, (n + 1) \cdot z)$$
$$(n + 1, n!) = t^n(0, 1)$$

This is what we get using the recipe. Quite expected and unsurprising, since this function is quite a simple one. Unfortunately, I have quietly pulled a fast one — the $\cdot$ operation was used as though we knew what it was. In fact, as an exercise you can find how to write multiplication in a recursive form.

S4 Q1

Write a function computing elements of Pascal's triangle, i.e. $\binom{\text{row}}{\text{col}}$. The following relationships might be helpful:

$$\binom{r}{c} = \binom{r-1}{c-1} + \binom{r-1}{c} \qquad \binom{r}{1} = \binom{r}{r} = 1$$

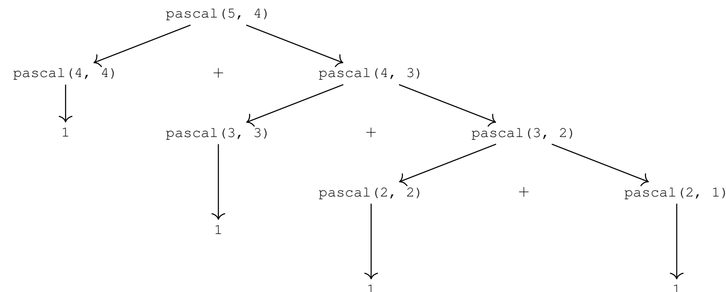
```
function pascal(row, col) {
    return col === 1 || col === row
        ? 1
        : pascal(row - 1, col - 1) + pascal(row - 1, col);
}
```


From the equation provided we see that this is quite similar to the Fibonacci numbers.

S4 Q2

Draw the tree illustration the process generated by `pascal(5, 4)`.

Quite standard. If this diagram confuses you, try covering up the 1's. Instead, replace the function call directly with the 1's instead of drawing another arrow. That should give you a clear idea of the number of deferred operations.



S4 IC-Q1

What do the following evaluate to?

```
compose(math_sqrt, math_log)(math_E)
compose(math_log, math_sqrt)(math_E * math_E)
```

```
(z => math_sqrt(math_log(z)))(math_E)
```

```
(y => math_log(math_sqrt(z)))(math_E * math_E)
```


This is simply substituting in the return results. Here perhaps we also highlight the fact that the bound variable names do not matter.

S4 IC-Q2

```
const compose = (f, g) => x => f(g(x));
function thrice(f) {
    return compose(compose(f, f), f);
}
```

```
thrice(h);
compose(compose(h, h), h)
compose(x => h(h(x)), h)
y => (x => h(h(x))(h(y))

thrice(h)(z);
(x => h(h(x))(h(z))
h(h(h(z)))
```


Let us evaluate a call to thrice. It may seem complicated but really we have done nothing but substitution.

- Substitute return value of `thrice`.
- Evaluate first argument (applicative order reduction).
- Substitute return value of `compose` with argument `f <- x=>h(h...)` and `g<-h`
- Substitute return value of `thrice`, and then substituting argument `y<-z`.
- Substitute return value of `x=>...` using argument `x<-h(z)`.

S4 IC-Q3

```
const compose = (f, g) => x => f(g(x));
function repeated(f, n) {
    return n === 0
        ? x => x
        : compose(f, repeated(f, n - 1));
}
```

```
repeated(f, 2);
compose(f, repeated(f, 1))
compose(f, compose(f, repeated(f, 0)))
compose(f, compose(f, x => x))
compose(f, y => f((x => x)(y)))
z => f((y => f((x => x)(y)))(z))

repeated(f, 2)(a);
f((y => f((x => x)(y)))(a))
f((f((x => x)(a))))
f((f((a))))
```

Again the name gives it away. But let us perform the analysis anyway. It is just careful work, but as long as you follow the rules, it is like clockwork.

- Substitute return value for `repeated`.
- Evaluate arguments. Substitute return value.
- Evaluate arguments of second `compose`.
- Evaluate second `compose` with arguments `f<-f` and `g<-x=>x`.
- Evaluate final `compose` with arguments `f<-f` and `g<-y => ...`.

- Substitute in return value, and call function with argument `z<-a`.
- Call `y => ...` with argument `y<-a`.
- Call `x => ...` with argument `x<-a`.

S4 IC-Q3

Cont.

```
const compose = (f, g) => x => f(g(x));
function thrice(f) {
    return compose(compose(f, f), f);
}
```

For what value of n will `((thrice(thrice))(f))(0)` return the same value as `(repeated(f, n))(0)`?

```
// thrice(h)(z) ---> h(h(h(z)))
(thrice(thrice))(f);
thrice(thrice(thrice(f)))

((thrice(thrice))(f))(0);
(thrice(thrice(thrice(f))))(0)
g(g(g(0))) // g = thrice(thrice(f))
g(g(h(h(h(0))))) // h = thrice(f)
g(g(h(h(f(f(f(0)))))))
```


The first expression may seem quite complex but actually we already know the behaviour of `thrice`. Using our previously established results, we can evaluate this easily.

- A substitution. The only difference is some brackets.
- Substitute using previous results.
- Introduce an abbreviation to reduce clutter. Also substitute in the first `thrice` using the same result (the commented line).
- Introduce another abbreviation. Evaluate the most inner `thrice` since it is being called with an argument.
- Evaluate the most inner `thrice` since it is being called with an argument.

The pattern emerges. `f(0)` is just a number, and so is `f(f(f(0)))`. The `h(f...)` will be expanded into `f(f(f(f...)))`. How many compositions of `f` is this? How many will there be after we evaluate all the `h`'s? How many will have occurred after evaluating all 3 `g`'s?

S4 IC-Q3

Cont.

```
// thrice(h)(z) ---> h(h(h(z)))
((thrice(thrice))(f))(0);
(thrice(thrice(thrice(f))))(0)
g(g(g(0)))             // g = thrice(thrice(f))
g(g(h(h(h(0)))))       // h = thrice(f)
g(g(h(h(f(f(f(0)))))))
g(g(h(f(f(f(a)))))) // a = f(f(f(0)))
g(g(f(f(f(b)))))       // b = f(f(f(a)))
g(g(c))                // c = f(f(f(b))) = ffffffa = fffffffff0
g(fffffffffc)
fffffffffd             // d = fffffffffc = ffffffff fffffffff0
ffffffff ffffffff fffffffff0
```

27.

The expansion follows the same rules as before so I will not write it all out again. The only thing to note is the slightly different notation. Here, because the number of brackets would be utterly disgusting, I have just dropped them. Thus `f(f(f(f(f(f(f(f(f(0)))))))))` `===` `fffffffff0`. In fact this is always fine as long as we follow certain rules, which we will not get into here.

S4 IC-Q4a

```
((thrice(thrice))(add1))(6);
```

```
aaaaaaaaa aaaaaaaaa aaaaaaaaa6
```

33.

Using the previous result the next question is fairly easy.

S4 IC-Q4b

```
((thrice(thrice))(x => x))(compose);
```

```
fffffffff fffffffff fffffffffc
fffffffff fffffffff ffffffffc
c
```

Again this is of the same form as the previous question. However the function being composed this time is the identity function. This is quite silly — even if you got the number of compositions wrong, you can still get this correct. The identity function remains as itself regardless of the number of times it self-composes.

S4 IC-Q4c,d

```
((thrice(thrice))(square))(2);
```

```
sssssssss sssssssss sssssssss2   // 2^1
sssssssss sssssssss ssssssss4    // 2^2
sssssssss sssssssss sssssss16    // 2^4
```

$2^{2^{27}} \gtrapprox 2^{100 \text{ million}}$

Part c is quite trivial for the same reason as part b. In any way it is the exact same question as part d.

You probably won't be able to check your answers to this question by running it on the interpreter. Using our previous results we conclude that this expression is equivalent to 27 repeated squarings. A hasty conclusion would be a result of 2^{27}. However if it was such a small number, the interpreter would not have failed. The following makes it clear that the answer is instead $2^{2^{27}}$.

$$(2)^2 = 2^2$$
$$(2^2)^2 = 2^4$$
$$(2^4)^2 = 2^8$$
$$(2^8)^2 = 2^{16}$$

Just for your info

$$2^{10} \approx 10^3 \qquad 2^{20} \approx 10^6 \qquad 2^{30} \approx 10^9$$