Lecture 8: Implementing Reactive Programming 2

More Continuation-passing style
Implementing event handlers
schmarrows

education version of arrowlets
Our goal

Write a JavaScript program with this behavior:

print 1
print 2
wait for a click
print 3
print 4
sleep for 1 second
print 5
print 6

Design an **DSL** with readable syntax, embedded in JS.

Namely, *shallowly* embedded: constructs are calls to a JS lib

*Deep embedding* is when the program exists as data, eg AST
Our starter skeleton

<body onload="start()"/>

function start() { ?? }
function f1() { print(1); }
function f2() { print(2); }
function click() { document.getElementById("myBtn").onclick = ??; }
function f3() { print(3); }
function f4() { print(4); }
function sleep() { setTimeout(??, 1000); }
function f5() { print(5); }
function f6() { print(6); }

Replace ?? with functions or function calls.
The complete starter skeleton

```html
<html>
<head></head>
<body onload= "start()">

<div></div>
<button id="myBtn">Click me</button>

<script>
function start() {
}
function f1() { print(1); }
...
function f6() { print(6); }

function print(s) {
    document.body.children[0].appendChild(document.createTextNode(s));
}
</script>
</body>
```
1. A working solution, but unsatisfying

function start() { f1(); }
function f1() { print(1); f2(); }
function f2() { print(2); click(); }
function click() { document.getElementById("myBtn").onclick = f3; }
function f3() { print(3); f4(); }
function f4() { print(4); sleep(); }
function sleep() { setTimeout(f5, 1000); }
function f5() { print(5); f6(); }
function f6() { print(6); }

We transfer the control correctly including via event handlers!
Why is this program unsatisfying?

Modifies the implementation of work functions f1..6
- Typically, we have only references to functions, not the code
- This also prevents us from calling any of these functions twice

The hardcoded calls to f1..6 are equivalent to goto:

```
goto Label1:
...
Label1:   ...
```
2. A fix: parameterize the work functions

```javascript
function f1k(k) { print(1); k(); }
function f2k(k) { print(2); k(); }
function clickk(k){ document.getElementById("myBtn").onclick = k; }
function f3k(k) { print(3); k(); }
function f4k(k) { print(4); k(); }
function sleepk(k){ setTimeout(k, 1000); }
function f5k(k) { print(5); k(); }
function f6k(k) { print(6); k(); }
```

The value passed to k is a continuation

a closure that executes “the rest of the program execution”

Continuation-passing style: when functions call continuations rather than returning (we’ll name such functions f1k)

A cps program never returns (only at the very end)

continuation calls are tail calls (no need to push frames to call stack)
Setting up the continuations

Should we define continuations from the end of the execution? Yes, easier than from the front.

```javascript
function e() { }
function k6() { f6k(e); }
function k56() { f5k(k6); }
function ks56() { sleepk(k56); }
function k4s56() { f4k(ks56); }
function k34s56() { f3k(k4s56); }
function kc34s56() { clickk(k34s56); }
function k2c34s56() { f2k(kc34s56); }
function k12c34s56() { f1k(k2c34s56); }

function start() { k12c34s56(); }
```

`k2c34s56` calls `f2` telling it to what continuation to continue via parameter `k`.
3. Better yet: avoid direct modification of f1..6

```javascript
function cps(f) { return function(k) { f(); k() } }

// cps(f1) turns function f1() { print(1) }
// into function f1k(k) { print(1); k(); }

function e() { }
function k6() { cps(f6)(e); }
function k56() { cps(f5)(k6); }
function ks56() { sleepk(k56); }
function k4s56() { cps(f4)(ks56); }
function k34s56() { cps(f3)(k4s56); }
function kc34s56() { clickk(k34s56); }
function k2c34s56() { cps(f2)(kc34s56); }
function k12c34s56() { cps(f1)(k2c34s56); }

function start() { k12c34s56(); }
```
Exercise

Draw the connections between functions after the code on the previous slide executes.

Instructions:
- Each function is a box
- Boxes a, b are connected if a calls b
  - Use the small black box to show that b is a continuation of a
Taking stock

Our work functions f1..6 are unmodified (yay!)

But do we want our programmers write this code?!?

```javascript
function e() { }
function k6() { cps(f6)(e); }
function k56() { cps(f5)(k6); }
function ks56() { sleepk(k56); }
function k4s56() { cps(f4)(ks56); }
function k34s56() { cps(f3)(k4s56); }
function kc34s56() { clickk(k34s56); }
function k2c34s56() { cps(f2)(kc34s56); }
function k12c34s56() { cps(f1)(k2c34s56); }
```
Are you bored?

For those needing a challenge, implement function `seq` that allows composition as close as possible to this:

```javascript
// compose f1..6, don’t call them yet
let c = seq(f1)(f2)(f3)(f4)(f5)(f6)

c() // f1..f6 are called now
```

Write `seq` with lambdas only, without arrays or objects.
For the rest of us ...

How our composition code should look like.
We want to write code as close as possible to this:

```javascript
var c = f1.next(f2) // compose the steps
  .click()
  .next(f3)
  .next(f4)
  .sleep()
  .next(f5)
  .next(f6);

c.run() // now run the composition
```
4. Stepping stone (forward composition)

As a prep step, we change this **backward** composition

```javascript
function e() { }
function k6() { wrap(f6)(e); }
function k56() { wrap(f5)(k6); }
function ks56() { sleep(k56); }
...
function k12c34s56() { wrap(f1)(k2c34s56); }
```

into a **forward** composition, roughly as follows

```javascript
function f12() { ... composes f1 and f2 ... }
function f12c() { ... composes f12 and the wait on click ... }
function f12c3() { ... composes f12c and f3 ... }
...
```
The actual composition code

```javascript
function cps(f) { return function(k) { f(); k() } }

var f1k = cps(f1);
var f12k = compose(f1k, f2);
var f12ck = compose_click(f12k);
var f12c3k = compose(f12ck, f3);
var f12c34k = compose(f12c3k, f4);
var f12c34sk = compose_sleep(f12c34k);
var f12c34s5k = compose(f12c34sk, f5);
var f12c34s56k = compose(f12c34s5k, f6);
```

**Overview of our approach:** The crux is writing the compose operator. Once we have it, we’ll just trivially wrap it in .next. Compose accepts f1k and f2 and produces comp.
The actual composition code

```javascript
var f1k = cps(f1);
var f12k = compose( f1k, f2);
// This example omits the click call, for simplicity.
var f123k = compose( f12k, f3);
var f1234k = compose(f123k, f4);

function start() { f1234k(e); }
```

We start the program with the empty continuation `e` which terminates the execution.

The pattern of the composition is now apparent.
All we need set the continuation calls ↓. See the next slide.
The composition code (without events)

```javascript
function compose(fk, g) {
    function fgk(k) {
        function gk() {
            g();
            k();
        }
        fk(gk);
    }
    return fgk;
}

var f1k = cps(f1);
var f12k = compose(f1k, f2);
var f123k = compose(f12k, f3);
var f1234k = compose(f123k, f4);

f1234k(e);
```
Compose with the click action

```javascript
function compose_click(fk) {
    return function (k) {
        function click_() {
            document.getElementById("myBtn2").onclick = k;
        }
        fk(click_);
    }
}
var f1k = wrap(f1);
var f12k = compose(f1k, f2);
var f12ck = compose_click(f12k);
var f12c3k = compose(f12ck, f3);
```
5. Final step: wrap it all in methods

Function.prototype.click = function() {
    var fk = this;
    return compose_click(fk);
}
Function.prototype.sleep = function() {
    var fk = this;
    return compose_sleep(fk);
}
Function.prototype.next = function(g) {
    var fk = this;
    return compose(fk,g);
}
Add begin and .run()

begin
.next(f1)
.next(f2)
.click()
.next(f3)
.next(f4)
.sleep()
.next(f5)
.next(f6)
.run()

var begin = function(k) { k() } // note that begin == cps(e)

Function.prototype.run = function() {
  var fk = this;
  fk(e);
}
Towards full arrowlets

We still need to:

- pass events to targets through the “pipeline”
  `EventA("mousedown").bind(start);`

- remove handlers after the event happens
  So that the second click does not reexecute f3, f4, ...
Hint for the seq puzzle

Problem: Implement function `seq` that allows composition as close as possible to this.

```javascript
// compose the functions
var c = seq(f1)(f2)(f3)(f4)(f5)(f6)()

// run the composition
console.log(c())
```

marks the end of composition
CPS Function Arrows
function calls in continuation-passing style
Now let’s return to Arrowlets

With the schmarrow language under our belt, we are ready to examine how cps implements the more serious arrowlets language.

We will implement Arrowlets in two steps:

CPS Function Arrows:

- compose functions with a continuation;
- Recall: CPS functions “never” return, always continue

Simple Async Event Arrows

- CPS functions that register their continuations as a handler for the particular event
How is arrowlets different in syntax

Differences from our schmarrow language:

cps functions now take one argument \( x \)
  this is in addition to the continuation argument \( k \)

Arrowlets uses classes
  .CpsA() and .next(k) rather than cps(f) and compose(k, f)
CPS Function Arrows in JS

Programs we want to write in this section.

```javascript
function f1(x) { return x + 1; }
function f2(x) { print x; }

var f1k = f1.CpsA(); // in schmarrows, this was cps(f1)
var f2k = f2.CpsA();

var comp = f1k.next(f2k);
comp.run(1); /* prints 2 */
```

Still no events, just functions. But with CPS Arrows in hand, adding events will be trivial.

**Key idea:** CpsA “lifts” the function f into a cps function. next then composes the cps functions.
Step 1: Transform function into cps functions

```javascript
function f1(x) { return x + 1; }
function f2(x) { print x; }

var f1k = f1.CpsA();
var f2k = f2.CpsA();

var comp = f1k.next(f2k);
comp.run(1);  /* prints 2 */
```

Recall syntax change from
wrap(f) → f.CpsA()
Step 1: Transform function into cps functions

```javascript
function f1(x) { return x + 1; }
function f2(x) { print x; }

var f1k = f1.CpsA();
var f2k = f2.CpsA();

var comp = f1k.next(f2k);
comp.run(1); /* prints 2 */
```
Step 2: Compose continuations

function f1(x) { return x + 1; }
function f2(x) { print x; }

var f1k = f1.CpsA();
var f2k = f2.CpsA();

var comp = f1k.next(f2k);
comp.run(1);  /* prints 2 */

Recall syntax change from
compose(k1, f2) → k1.next(k2)
where k2 is CPS version of f2

k: the next continuation to call
Step 3: Apply the composed chain to value

function f1(x) { return x + 1; }
function f2(x) { print x; }

var f1k = f1.CpsA();
var f2k = f2.CpsA();

var comp = f1k.next(f2k);
comp.run(1); /* prints 2 */

In Arrowlets diagram:

How are CpsA and next implemented?
Implementation steps

1. Define the CpsA class
2. Add CpsA() method to Function class
   (to turn ordinary Functions into CpsA objects)
3. Define CpsA.run(x)
4. Define CpsA.next(k) to compose two CpsA objects
Step 1: Define the CpsA class

// create a class (prototype) CpsA with a field cps storing the continuation

// Constructor for the CpsA class
function CpsA(cps) {
    // cps is a function in CPS style, returns void
    this.cps = cps; // i.e., cps is an function with type (x, k) → ()
    // can be invoked by CpsAObj.cps(x,k)
}

// CpsA.CpsA() method
// will see why this is useful in the implementation of CpsA.next
CpsA.prototype.CpsA = function() { return this; }
Step 2: Add CpsA() method to the Function class

// convert a direct-style function into a cps function stored in a CpsA object

// JS syntax: add method CpsA() to the JS Function class
Function.prototype.CpsA = function() {
    var f = this;
    // wrap the regular (direct-style) function f in a CpsA object
    return new CpsA(function(x, k) {
        k(f(x));
        // note: f(x) in k(f(x)) is not a tail call
    }));
}

x: argument to pass to f
k: the continuation to execute after f(x) returns

When called, we will first call f(x), and then pass the value to k
What is f? It’s the function that we are converting to cps
Step 3: Define CpsA.run(x)

// run calls the CPS function stored in the CpsA object,
// passing it the input argument and an empty continuation.
// The empty continuation will stop the chain of tail calls, ending the
// evaluation

CpsA.prototype.run = function(x) {
    this.cps(x, function(y) {});
}
Step 4: Define CpsA.next(k) to compose CpsA objs

CpsA.prototype.next = function(g) {
    var f = this;
    // if g is a function, then g.CpsA() will convert it into cps
    // but if g already is cps, then g.CpsA() is an identity function
    // (see implementation of CpsA.CpsA() method 3 slides back)
    g = g.CpsA();
    // f and g are now CpsA objects. Now let’s create a new function
    // such that when called, it would execute their composition
    var fgComposed =
        function(x, k) {
            f.cps(x, function(y) {
                g.cps(y, k);
            });
        }
    // and wraps the composition in a new CpsA object
    var r = new CpsA(fgComposed); return r;
}
Example

```javascript
function f1(x) { return x + 1; }
function f2(x) { print x; }
var f1k = f1.CpsA();
var f2k = f2.CpsA();

var comp = f1k.next(f2k);
comp.run(1);   /* prints 2 */
```

```javascript
CpsA.prototype.next = function(g) {
  var f = this;
  g = g.CpsA();
  var fgComposed =
      function(x, k) {
        f.cps(x, function(y) {
          g.cps(y, k); })
      };
  var r = new CpsA(fgComposed);
  return r;
}
```
Example

```javascript
function f1(x) { return x + 1; }
function f2(x) { print x; }
var f1k = f1.CpsA();
var f2k = f2.CpsA();

var comp = f1k.next(f2k);
comp.run(1); /* prints 2 */

CpsA.prototype.next = function(g) {
    var f = this;
g = g.CpsA();
    var fgComposed =
        function(x, k) {
            f.cps(x, function(y) {
                g.cps(y, k); })
        }; 
    var r = new CpsA(fgComposed);
    return r;
}
```
Example

```javascript
function f1(x) { return x + 1; }
function f2(x) { print x; }
var f1k = f1.CpsA();
var f2k = f2.CpsA();

var comp = f1k.next(f2k);
comp.run(1); /* prints 2 */

CpsA.prototype.next = function(g) {
  var f = this;
  g = g.CpsA();
  var fgComposed =
    function(x, k) {
      f.cps(x, function(y) {
        g.cps(y, k);
      });
    };
  var r = new CpsA(fgComposed);
  return r;
}
```
Example

```javascript
function f1(x) { return x + 1; }
function f2(x) { print x; }
var f1k = f1.CpsA();
var f2k = f2.CpsA();

var comp = f1k.next(f2k);
comp.run(1); /* prints 2 */

CpsA.prototype.next = function(g) {
  var f = this;
  g = g.CpsA();
  var fgComposed =
    function(x, k) {
      f.cps(x, function(y) {
        g.cps(y, k); })
    };
  var r = new CpsA(fgComposed);
  return r;
}
```
Example

```javascript
function f1(x) { return x + 1; }
function f2(x) { print x; }
var f1k = f1.CpsA();
var f2k = f2.CpsA();

var comp = f1k.next(f2k);
comp.run(1);  /* prints 2 */

CpsA.prototype.next = function(g) {
    var f = this;
    g = g.CpsA();
    var fgComposed =
        function(x, k) {
            f.cps(x, function(y) {
                g.cps(y, k); })
        };
    var r = new CpsA(fgComposed);
    return r;
}
```
Example

```javascript
function f1(x) { return x + 1; }
function f2(x) { print x; }
var f1k = f1.CpsA();
var f2k = f2.CpsA();

var comp = f1k.next(f2k);
comp.run(1); /* prints 2 */

CpsA.prototype.next = function(g) {
  var f = this;
  g = g.CpsA();
  var fgComposed =
    function(x, k) {
      f.cps(x, function(y) {
        g.cps(y, k);
      });
    }
  var r = new CpsA(fgComposed);
  return r;
}
```

Conceptually comp == r
Simple Async Event Arrows

let’s add events
Where are we?

Function Arrows:
  compose direct-style functions

CPS Function Arrows:
  compose continuation-passing-style functions
  CPS functions “never” return, always continue

next, Simple Async Event Arrows
  CPS functions that register their continuations as a handler for the particular event
Example of programs we want to write

```javascript
var count = 0;

// this is a regular handler, nothing special here
function clickTargetA (event) {
    var target = event.currentTarget;
    target.textContent = "You clicked me! " + ++count;
    return target;
}

SimpleEventA("click") // wait for click event
    .next( clickTargetA ) // call handler on target passed to next
    .run(document.getElementById("target")); // select the target
```
Reuse of code is now possible

Same code composition run on a different target

SimpleEventA("click")
  .next( clickTargetA )
  .run(document.getElementById("anotherTarget"));
Defining the SimpleEventA class

// Class constructor
function SimpleEventA(eventname) {
    if (!(this instanceof SimpleEventA))
        return new SimpleEventA(eventname);
    this.eventname = eventname;
}

Explanation:
If the constructor SimpleEventA is called as a regular function (i.e., without new), it calls itself again as a constructor to create a new SimpleEventA object.

This allows us to omit new when using SimpleEventA, for example in .next( SimpleEventA("click") )
Making SimpleEventA a “subclass” of CpsA

// First, create a function that will be invoked when the continuation
// associated with SimpleEventA is called, here k is the continuation that will
// be called when the event handler is invoked

// Next, we make SimpleEventA a subclass of CpsA by setting its prototype
// field to be a CpsA object (JS doesn’t really have inheritance)

SimpleEventA.prototype = new CpsA(function(target, k) {
    var f = this;

    // register event handler as part of the continuation
    function handler(event) {
        target.removeEventListener(f.eventname, handler, false);
        k(event);
    }
    target.addEventListener(f.eventname, handler, false);
});
Example

```javascript
var s = SimpleEventA("click")
    .next( clickTargetA );
    .run(document.getElementById("target"));

function SimpleEventA(eventname) {
    ...
    this.eventname = eventname;
}

SimpleEventA.prototype = new CpsA(
    function(target, k) {
        var f = this;
        function handler(event) {
            target.removeEventListener(f.eventname, handler, false);
            k(event);
        }
        target.addEventListener(f.eventname, handler, false);
    });
```

```javascript
Example
```
Example

```javascript
var s = SimpleEventA("click")
    .next( clickTargetA );
    .run(document.getElementById("target"));

function SimpleEventA(eventname) {
  ...
  this.eventname = eventname;
}

SimpleEventA.prototype = new CpsA(
  function(target, k) {
    var f = this;
    function handler(event) {
      target.removeEventListener(f.eventname, handler, false);
      k(event);
    }
    target.addEventListener(f.eventname, handler, false);
  });
```

Example

```javascript
var s = SimpleEventA("click")
  .next( clickTargetA );
  .run(document.getElementById("target"));

function SimpleEventA(eventName) {
  ...
  this.eventname = eventName;
}

SimpleEventA.prototype = new CpsA( function(target, k) {
  var f = this;
  function handler(event) {
    target.removeEventListener(f.eventname, handler, false);
    k(event);
  }
  target.addEventListener(f.eventname, handler, false);
});
```
Example

```javascript
var s = SimpleEventA("click")
  .next( clickTargetA )
  .run(document.getElementById("target"));

function SimpleEventA(eventName) {
  ...  
  this.eventname = eventName;
}

SimpleEventA.prototype = new CpsA(  
  function(target, k) {
    var f = this;
    function handler(event) {
      target.removeEventListener(f.eventname, handler, false);
      k(event);
    }
    target.addEventListener(f.eventname, handler, false);
  }
);  
```
Example

```javascript
var s = SimpleEventA("click")
  .next( clickTargetA );
  .run(document.getElementById("target"));

function SimpleEventA(eventname) {
  ...
  this.eventname = eventname;
}

SimpleEventA.prototype = new CpsA(function(target, k) {
  var f = this;
  function handler(event) {
    target.removeEventListener(f.eventname, handler, false);
    k(event);
  }
  target.addEventListener(f.eventname, handler, false);
});
```

![Diagram of event handling process]

The diagram above illustrates the event handling process for the `SimpleEventA` function. The `eventName` is passed as a parameter to the constructor, and the event listener is added to the target object. When an event occurs, the `handler` function is executed, which can subsequently call another function (represented by `k(event)`). This represents a basic example of how event listeners can be managed using a custom event handling function `SimpleEventA`. This approach allows for more flexible and reusable code when managing different types of events and their associated actions.
Example

```javascript
var s = SimpleEventA("click")
    .next( clickTargetA );
    .run(document.getElementById("target"));

function SimpleEventA(eventname) {
    ...
    this.eventname = eventname;
}

SimpleEventA.prototype = new CpsA(
    function(target, k) {
        var f = this;
        function handler(event) {
            target.removeEventListener(f.eventname, handler, false);
            k(event);
        }
        target.addEventListener(f.eventname, handler, false);
    });
```
Another example (wait for two clicks)

SimpleEventA("click")
  .next( clickTargetA )
  // as the next step of the pipeline, wait for click and call handler
  .next( SimpleEventA("click").next( clickTargetA ) )
  .run(document.getElementById("target"));

Same as this program (because .next is associative):

SimpleEventA("click")
  .next( clickTargetA )
  .next( SimpleEventA("click") )
  .next( clickTargetA )
  .run(document.getElementById("target"));
Another example (wait for two clicks)

```
SimpleEventA("click")
    .next( clickTargetA )
    // as the next step of the pipeline, wait for click and call handler
    .next( SimpleEventA("click").next( _clickTargetA_ ) )
    .run(document.getElementById("target"))
```

```
SimpleEventA.prototype = new CpsA(function(target, k) {
    var f = this;
    function handler(event) {
        target.removeEventListener(f.eventname, handler, false);
        k(event);
    }
    target.addEventListener(f.eventname, handler, false);
    k(event);
});
```
Full Async Event Arrows

a realistic system
Full Asynchronous Event Arrows

Function Arrows:
- compose functions in a wrapper function

CPS Function Arrows:
- compose functions with a continuation;
- CPS functions “never” return, always continue

Simple Async Event Arrows
- CPS functions that register their continuations to handle a particular event

next, Full Async Event Arrows
- We will add combinators needed by drag and drop
  see last lecture and the paper for details if interested
Full Async Arrows

Want to support multiple arrows in flight
  i.e., wait for multiple events at once
Only one of the events will happen
  So we must be able to cancel one of the two waiting events
Solution: Build AsyncA,
  – AsyncA extends CpsA to support tracking progress and cancellation
  – When AsyncA is run, it returns a progress arrow
Using AsyncA, we build EventA,
  Which extends SimpleEventA to track progress and cancellation.
Example

The next example shows how to perform an animation of bubblesort. We want to sleep for 100ms between each iteration. How to do this nicely in JS?

The Arrowlets code on next slide looks almost like a vanilla bubblesort. The key is the repeat(100) operator that calls the body every 100 ms.
Example

```javascript
var bubblesortA = function(x) {
    var list = x.list, i = x.i, j = x.j;
    if (j + 1 < i) {
        if (list.get(j) > list.get(j + 1)) {
            list.swap(j, j + 1);
        }
        return Repeat({ list: list, i: i, j: j + 1 });
    } else if (i > 0) {
        return Repeat({ list: list, i: i - 1, j: 0 });
    } else {
        return Done();
    }
};
AsyncA().repeat(100);
/* list is an object with methods get and swap */
bubblesortA.run({list:list, i: list.length, j: 0 });
```
Summary

• In compiling arrowlet constructs, we introduced the concept of continuations

• Continuations represent “the rest of the program”
  – A function that takes in program state and is invoked after the current function terminates
  – Hence it conceptually “never returns” to its caller

• We have seen examples of how this can be used to implement event handlers

• We will see how this is implemented in Rx and Vega in the next lecture
Extra material

How to implement tail recursion elimination when the compiler does not support it
Trampoline

Consider this code in CPS:

\[
\begin{align*}
\text{def } f_3(x,k) \{ \text{ print } x \}
\text{def } f_2(x,k) \{ k(x+2, \{ \}) \}
\text{def } f_1(x,k) \{ k(x/2, f_3); \}
\text{def } g(x,k) \{ k(x, f_2); \}
\end{align*}
\]

\[
\begin{align*}
g(1, f_1)
\end{align*}
\]

\{ \} = “end” sentinel

Call stack:

<table>
<thead>
<tr>
<th>function</th>
<th>arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>1, f1</td>
</tr>
<tr>
<td>f1</td>
<td>1, f2</td>
</tr>
<tr>
<td>f2</td>
<td>1/2, f3</td>
</tr>
<tr>
<td>f3</td>
<td>1/2 + 2, { }</td>
</tr>
</tbody>
</table>

Stack keeps growing!
Trampoline: Poor man’s tail-call optimization

Implement an iterator that keeps executing continuations:

Now we can write the following:

```plaintext
trampoline(x, cont) {
    while (cont != {})
        (x, cont) = cont(x);
}

f3(x) { (print x, {}) }
f2(x) { (x+2, f3) }
f1(x) { (x/2, f2) }
g(x) { x }

tramarine(1, g)
```

Call stack:

<table>
<thead>
<tr>
<th>function</th>
<th>arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>trampoline</td>
<td>1, g</td>
</tr>
<tr>
<td>g</td>
<td>1</td>
</tr>
</tbody>
</table>
Trampoline: Poor man’s tail-call optimization

Implement an iterator that keeps executing continuations:

Now we can write the following:

\[
\text{trampoline}(x, \text{cont}) = \begin{cases} \\
(x, \text{cont}) = \text{cont}(x); \\
\end{cases}
\]

\[
\text{while } (\text{cont} \neq \{\}) { \\
\text{f3(x) } \{ \text{print x, \{\}} \} \\
\text{f2(x) } \{ \text{x+2, f3} \} \\
\text{f1(x) } \{ \text{x/2, f2} \} \\
\text{g(x) } \{ \text{x} \} \\
\text{trampoline}(1, \text{g})
\]

Call stack:

<table>
<thead>
<tr>
<th>function</th>
<th>arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>trampoline</td>
<td>1, g</td>
</tr>
<tr>
<td>f1</td>
<td>1/2</td>
</tr>
</tbody>
</table>

Similarly for the rest
Home exercise

Encode our earlier example using trampoline

```javascript
function add1(x) { return x+1; }
function add2(x) { return x+2; }
var add3 = add1.next(add2);
add3(1);

trampoline(x, cont) {
  while (cont !== { }) {
    (x, cont) = cont(x);
  }
  x
}
```

```python
def add2(x) { (x+2, {}) }
def add1(x) { (x+1, add2) }
def add3(x) { (x, add1) }
```