Directing JavaScript with Arrows

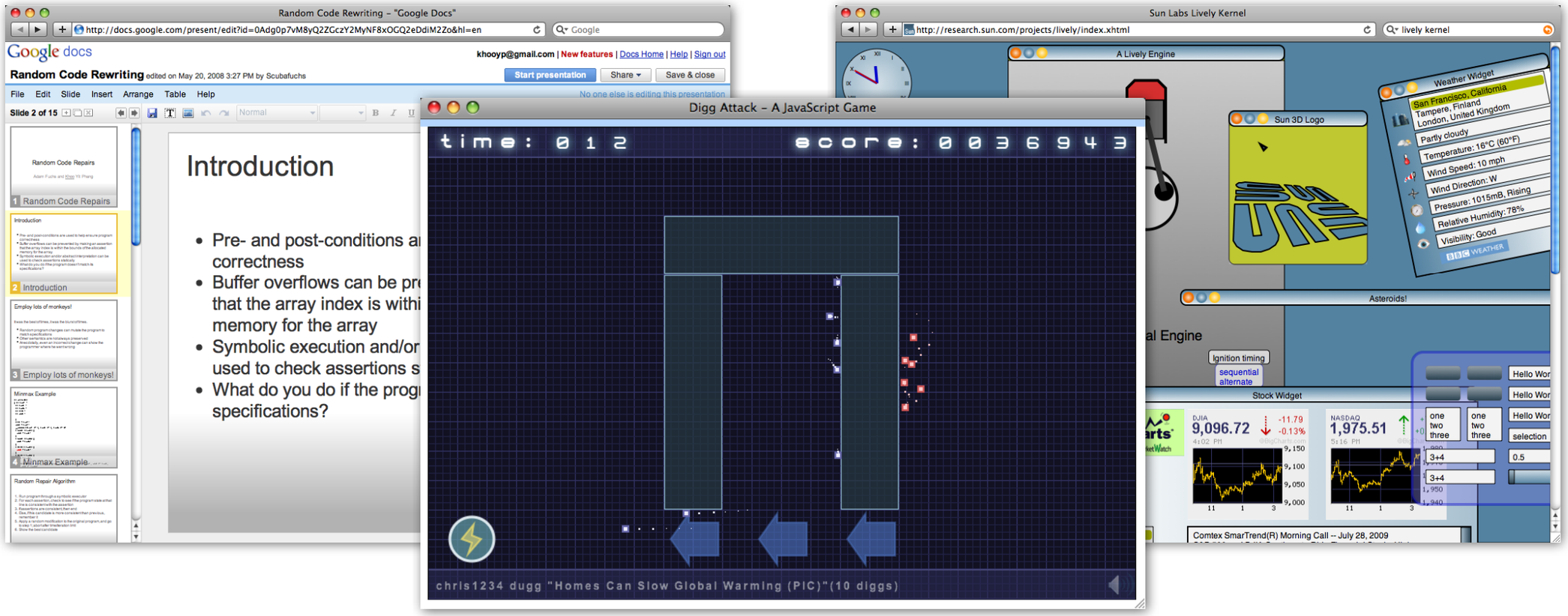
Khoo Yit Phang, Michael Hicks, Jeffrey S. Foster, Vibha Sazawal
University of Maryland
October 26, 2009

JavaScript: *de-facto* language for Web 2.0

- JavaScript increasingly used to write sophisticated, interactive web applications:

Productivity Applications

Development Environments



Games

UI programming with JavaScript

- JavaScript is single-threaded; event-driven programming only way for interactive apps
- E.g., trigger when “click” on HTML element:

```
element.addEventListener(
  “click”, function(){alert(“clicked!”)})
```

- E.g., break-up long-running computation:

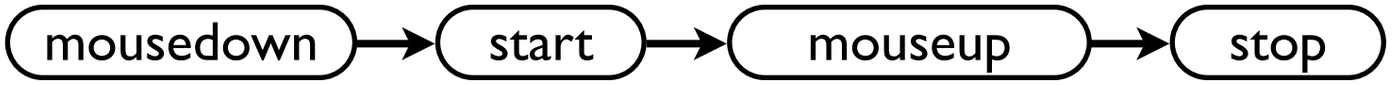
```
function longcompute() {
  /* ... */ setTimeout(longcompute, 0) }
```

Understanding event-driven control flow

- Often interested in *sequences* of events

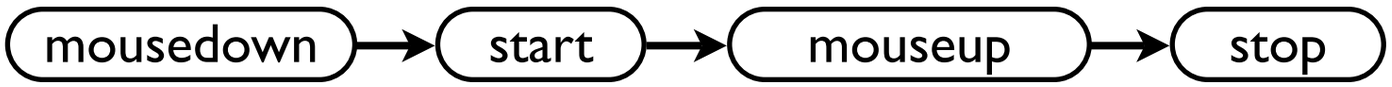
Understanding event-driven control flow

- Often interested in *sequences* of events



Understanding event-driven control flow

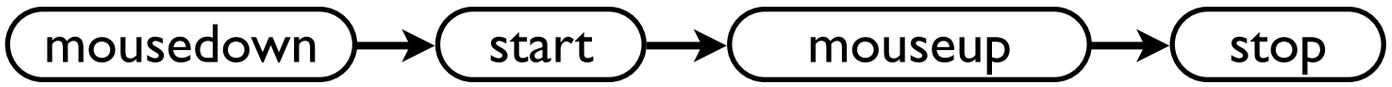
- Often interested in *sequences* of events




```
function start(event) {
  A.removeEventListener("mousedown", start);
  A.addEventListener("mouseup", stop);
  A.style.background = "yellow";
}

function stop(event) {
  A.removeEventListener("mouseup", stop);
  A.style.background = "white";
}

A.addEventListener("mousedown", start);
```

Understanding event-driven control flow

- Often interested in *sequences* of events

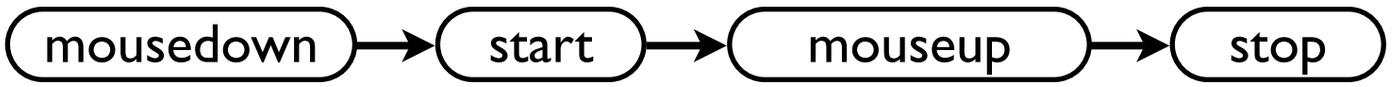



```
function start(event) {
  A.removeEventListener("mousedown", start);
  A.addEventListener("mouseup", stop);
  A.style.background = "yellow";
}

function stop(event) {
  A.removeEventListener("mouseup", stop);
  A.style.background = "white";
}

A.addEventListener("mousedown", start);
```

Control flow is indirect, convoluted, hard to understand

Understanding event-driven control flow

- Often interested in *sequences* of events



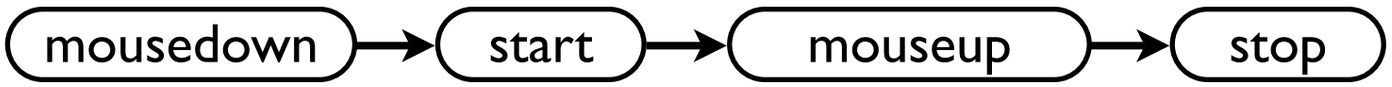

```
function start(event) {
  A.removeEventListener("mousedown", start);
  A.addEventListener("mouseup", stop);
  A.style.background = "yellow";
}

function stop(event) {
  A.removeEventListener("mouseup", stop);
  A.style.background = "white";
}

A.addEventListener("mousedown", start);
```

Control flow is indirect, convoluted, hard to understand

Control flow "plumbing" interspersed with "action"

Maintaining event-driven programs

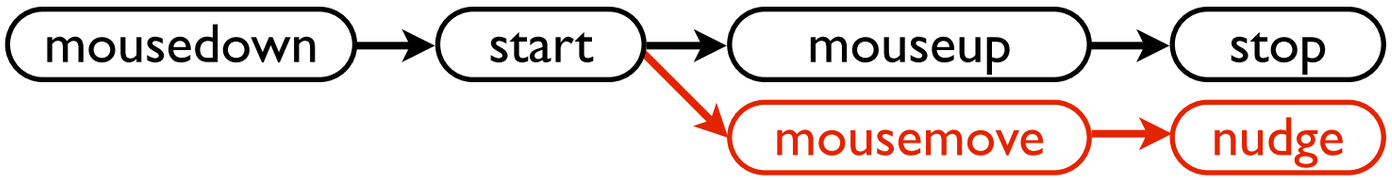
- Modifications are annoyingly difficult




```
function start(event) {
  A.removeEventListener(“mousedown”, start);
  A.addEventListener(“mouseup”, stop);
  A.style.background = “yellow”;
}

function stop(event) {
  A.removeEventListener(“mouseup”, stop);
  A.style.background = “white”;
}

A.addEventListener(“mousedown”, start);
```

Maintaining event-driven programs

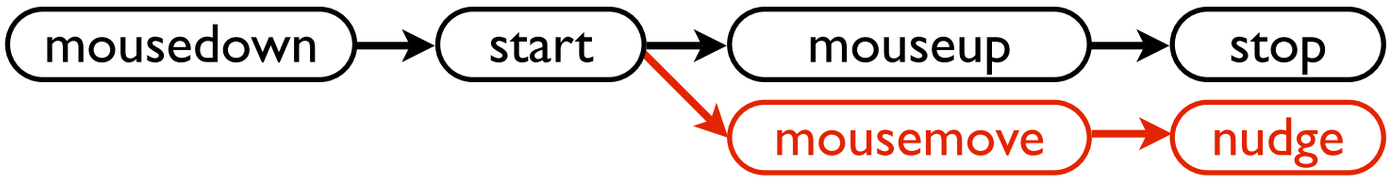
- Modifications are annoyingly difficult




```
function start(event) {
  A.removeEventListener(“mousedown”, start);
  A.addEventListener(“mouseup”, stop);
  A.style.background = “yellow”;
}

function stop(event) {
  A.removeEventListener(“mouseup”, stop);
  A.style.background = “white”;
}

A.addEventListener(“mousedown”, start);
```

Maintaining event-driven programs

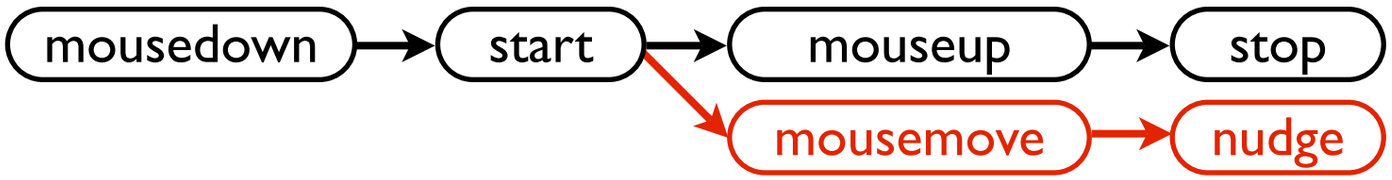
- Modifications are annoyingly difficult




```
function start(event) {
  A.removeEventListener(“mousedown”, start);
  A.addEventListener(“mouseup”, stop);
  A.addEventListener(“mousemove”, nudge);
  A.style.background = “yellow”;
}
function stop(event) {
  A.removeEventListener(“mouseup”, stop);
  A.removeEventListener(“mousemove”, nudge);
  A.style.background = “white”;
}
function nudge(event) { /* ... */ }
A.addEventListener(“mousedown”, start);
```

Maintaining event-driven programs

- Modifications are annoyingly difficult

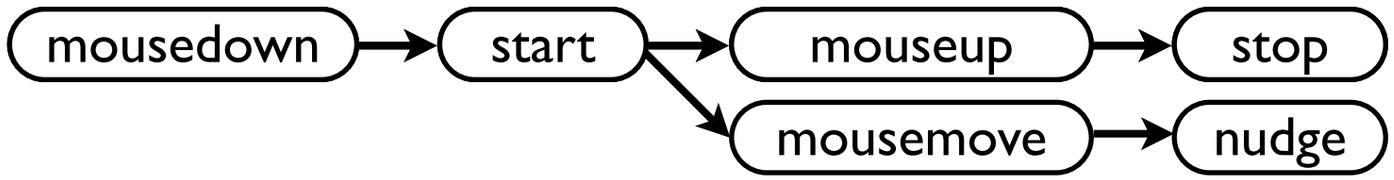


Changes strewn throughout code

```
function start(event) {
  A.removeEventListener("mousedown", start);
  A.addEventListener("mouseup", stop);
  A.addEventListener("mousemove", nudge);
  A.style.background = "yellow";
}
function stop(event) {
  A.removeEventListener("mouseup", stop);
  A.removeEventListener("mousemove", nudge);
  A.style.background = "white";
}
function nudge(event) { /* ... */ }
A.addEventListener("mousedown", start);
```

Maintaining event-driven programs

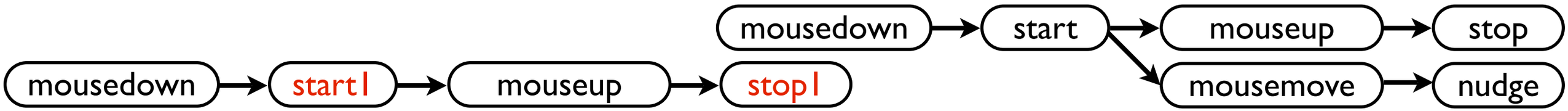
- What if you want old *and* new behavior?




```
function start(event) {
  A.removeEventListener(“mousedown”, start);
  A.addEventListener(“mouseup”, stop);
  A.addEventListener(“mousemove”, nudge);
  A.style.background = “yellow”;
}
function stop(event) {
  A.removeEventListener(“mouseup”, stop);
  A.removeEventListener(“mousemove”, nudge);
  A.style.background = “white”;
}
function nudge(event) { /* ... */ }
A.addEventListener(“mousedown”, start);
```

Maintaining event-driven programs

- What if you want old *and* new behavior?

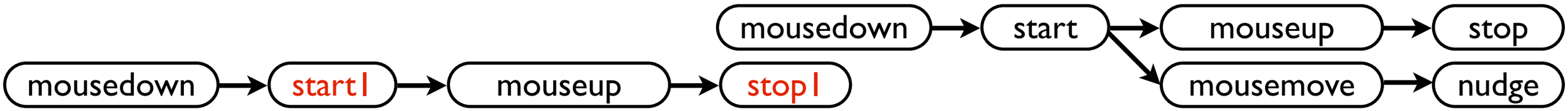



```
function start1(event) {
  A.removeEventListener(“mousedown”, start1);
  A.addEventListener(“mouseup”, stop1);
  A.style.background = “yellow”;
}

function stop1(event) {
  A.removeEventListener(“mouseup”, stop1);
  A.style.background = “white”;
}

A.addEventListener(“mousedown”, start1);
```

```
function start(event) {
  A.removeEventListener(“mousedown”, start);
  A.addEventListener(“mouseup”, stop);
  A.addEventListener(“mousemove”, nudge);
  A.style.background = “yellow”;
}
function stop(event) {
  A.removeEventListener(“mouseup”, stop);
  A.removeEventListener(“mousemove”, nudge);
  A.style.background = “white”;
}
function nudge(event) { /* ... */ }
A.addEventListener(“mousedown”, start);
```

Maintaining event-driven programs

- What if you want old *and* new behavior?

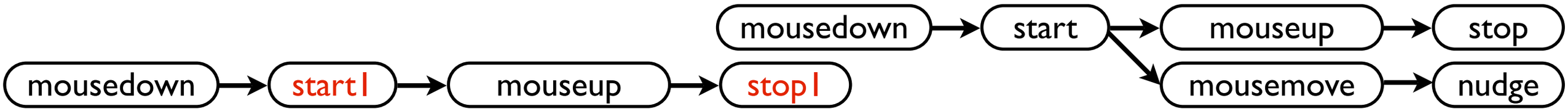



```
function start1(event) {
  A.removeEventListener(“mousedown”, start1);
  A.addEventListener(“mouseup”, stop1);
  A.style.background = “yellow”;
}

function stop1(event) {
  A.removeEventListener(“mouseup”, stop1);
  A.style.background = “white”;
}

A.addEventListener(“mousedown”, start1);
```

```
function start(event) {
  A.removeEventListener(“mousedown”, start);
  A.addEventListener(“mouseup”, stop);
  A.addEventListener(“mousemove”, nudge);
  A.style.background = “yellow”;
}
function stop(event) {
  A.removeEventListener(“mouseup”, stop);
  A.removeEventListener(“mousemove”, nudge);
  A.style.background = “white”;
}
function nudge(event) { /* ... */ }
A.addEventListener(“mousedown”, start);
```

Maintaining event-driven programs

- What if you want old *and* new behavior?



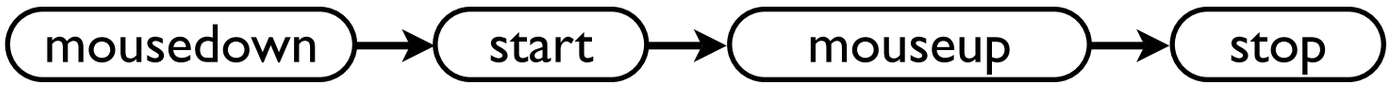

```
function start1(event) {
  A.removeEventListener("mousedown", start1);
  A.addEventListener("mouseup", stop1);
  A.style.background = "yellow";
}

function stop1(event) {
  A.removeEventListener("mouseup", stop1);
  A.style.background = "white";
}

A.addEventListener("mousedown", start1);
```

```
function start(event) {
  A.removeEventListener("mousedown", start);
  A.addEventListener("mouseup", stop);
  A.addEventListener("mousemove", nudge);
  A.style.background = "yellow";
}
function stop(event) {
  A.removeEventListener("mouseup", stop);
  A.removeEventListener("mousemove", nudge);
  A.style.background = "white";
}
function nudge(event) { /* ... */ }
A.addEventListener("mousedown", start);
```

Need to duplicate the entire code, and remember to rename carefully!

Re-using event-driven components

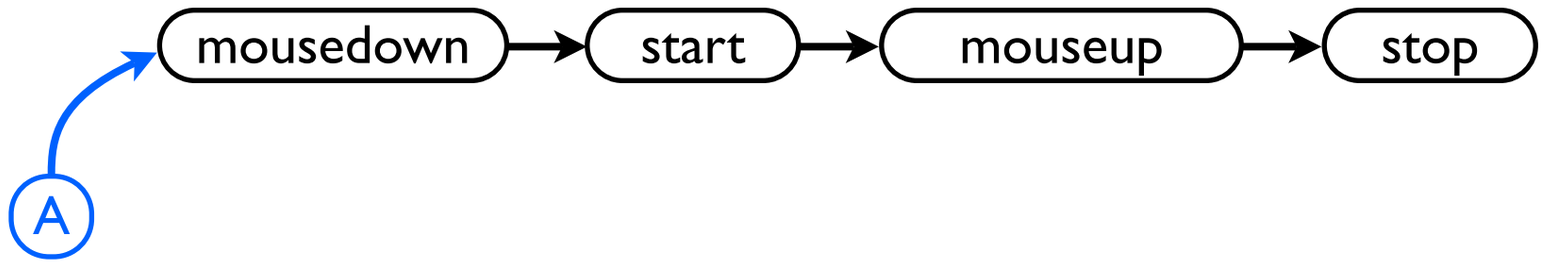
- Say, want to maximize re-use (it's good!)




```
function start(event) {
  A.removeEventListener("mousedown", start);
  A.addEventListener("mouseup", stop);
  A.style.background = "yellow";
}
function stop(event) {
  A.removeEventListener("mouseup", stop);
  A.style.background = "white";
}
A.addEventListener("mousedown", start);
```

Re-using event-driven components

- Say, want to maximize re-use (it's good!)

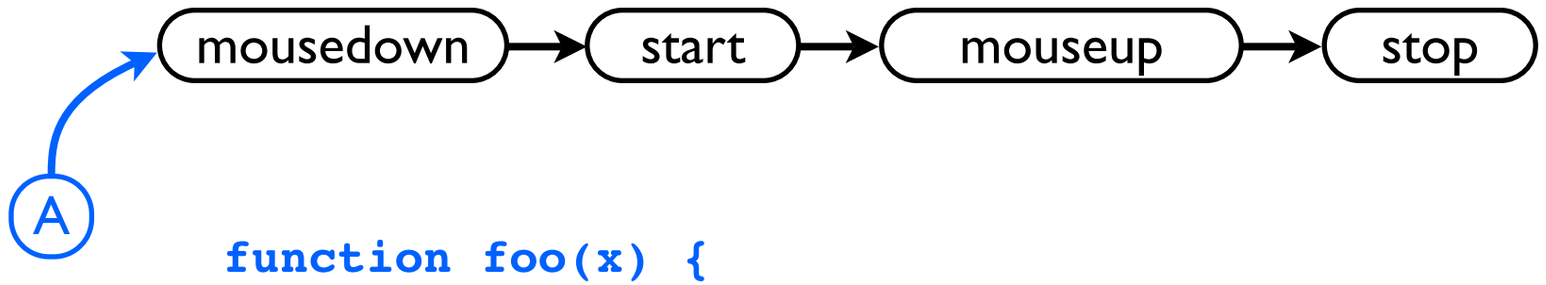


We can factor out the target

```
function start(event) {
  A.removeEventListener("mousedown", start);
  A.addEventListener("mouseup", stop);
  A.style.background = "yellow";
}
function stop(event) {
  A.removeEventListener("mouseup", stop);
  A.style.background = "white";
}
A.addEventListener("mousedown", start);
```

Re-using event-driven components

- Say, want to maximize re-use (it's good!)

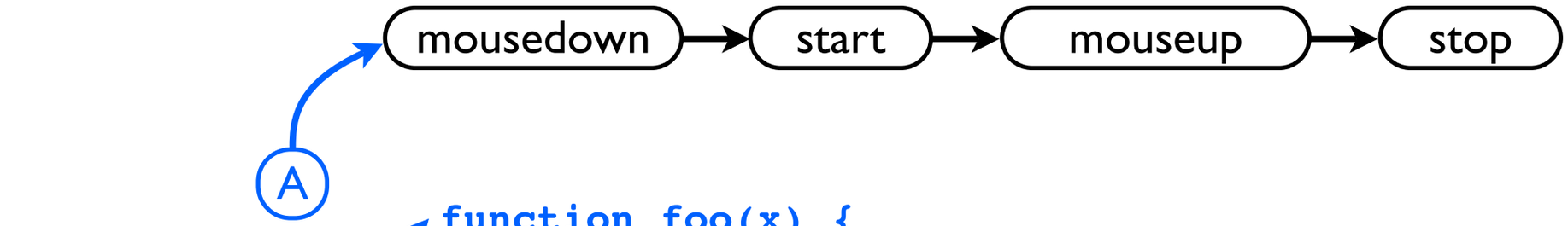


Parameterize using a closure

```
function foo(x) {
  function start(event) {
    x.removeEventListener("mousedown", start);
    x.addEventListener("mouseup", stop);
    x.style.background = "yellow";
  }
  function stop(event) {
    x.removeEventListener("mouseup", stop);
    x.style.background = "white";
  }
  x.addEventListener("mousedown", start);
}
foo(A); foo(B);
```

Re-using event-driven components

- Say, want to maximize re-use (it's good!)

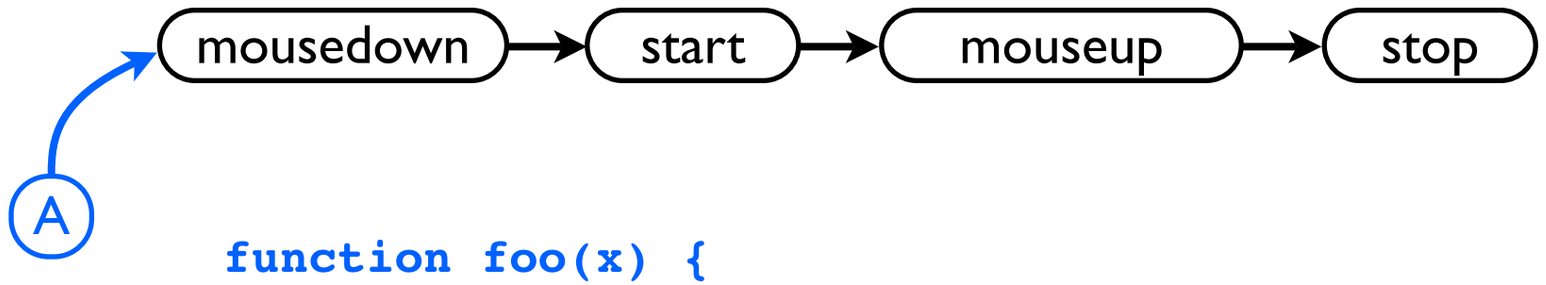


Parameterize using a closure

```
function foo(x) {
  function start(event) {
    x.removeEventListener("mousedown", start);
    x.addEventListener("mouseup", stop);
    x.style.background = "yellow";
  }
  function stop(event) {
    x.removeEventListener("mouseup", stop);
    x.style.background = "white";
  }
  x.addEventListener("mousedown", start);
}
foo(A); foo(B);
```

Re-using event-driven components

- Say, want to maximize re-use (it's good!)

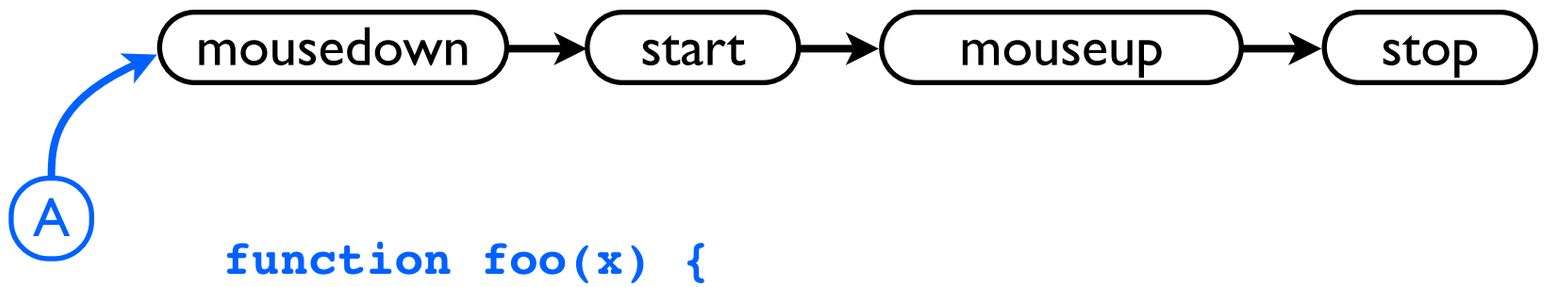


But, we cannot decouple start and stop

```
function foo(x) {
  function start(event) {
    x.removeEventListener("mousedown", start);
    x.addEventListener("mouseup", stop);
    x.style.background = "yellow";
  }
  function stop(event) {
    x.removeEventListener("mouseup", stop);
    x.style.background = "white";
  }
  x.addEventListener("mousedown", start);
}
foo(A); foo(B);
```

Re-using event-driven components

- Say, want to maximize re-use (it's good!)

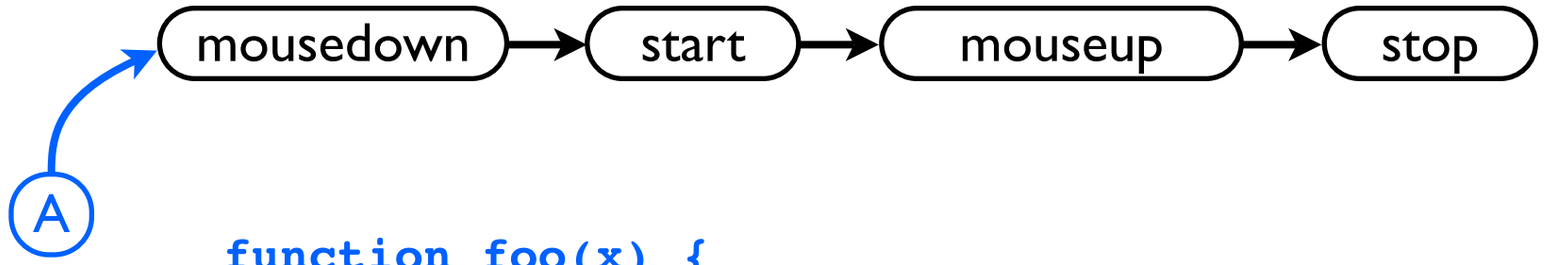


But, we cannot decouple start and stop

```
function foo(x) {
  function start(event) {
    x.removeEventListener("mousedown", start);
    x.addEventListener("mouseup", stop);
    x.style.background = "yellow";
  }
  function stop(event) {
    x.removeEventListener("mouseup", stop);
    x.style.background = "white";
  }
  x.addEventListener("mousedown", start);
}
foo(A); foo(B);
```

Re-using event-driven components

- Say, want to maximize re-use (it's good!)



But, we cannot decouple start and stop

In fact, the closure makes it worse

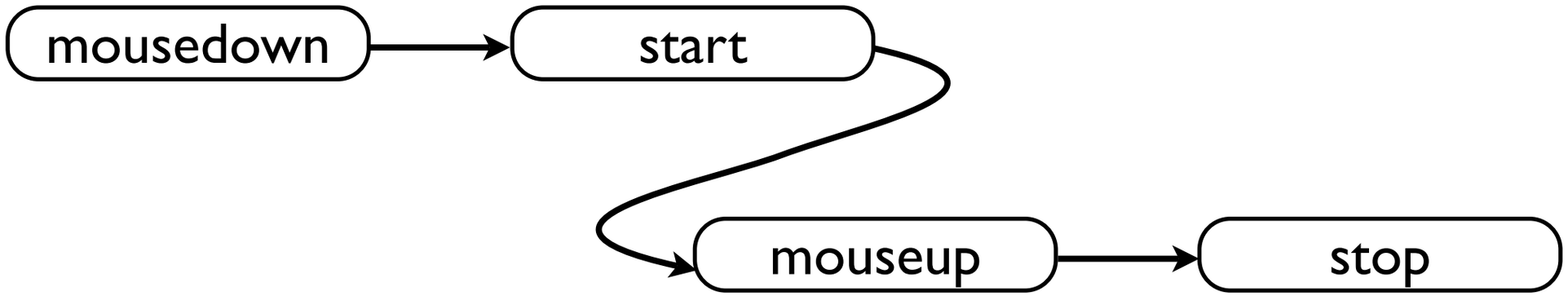
```
function foo(x) {
  function start(event) {
    x.removeEventListener("mousedown", start);
    x.addEventListener("mouseup", stop);
    x.style.background = "yellow";
  }
  function stop(event) {
    x.removeEventListener("mouseup", stop);
    x.style.background = "white";
  }
  x.addEventListener("mousedown", start);
}
foo(A); foo(B);
```

Event-driven programs with *Arrowlets*

- *Arrowlets* is a JavaScript library to make composing events easy
  - easy-to-understand control flow
  - "plumbing" separate from "action"
  - small, modular compositions enables re-use

Event-driven control flow with Arrowlets

- The state machine (from before):

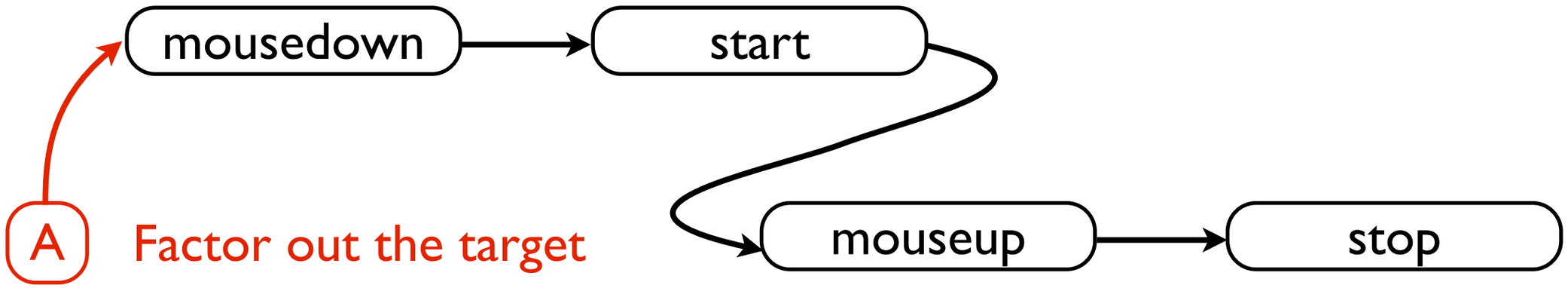


- Re-write start/stop without plumbing:

```
function start(target, event)
{     target.style.background = “yellow”;
      return target;                                  }
function stop(target, event,)
{     target.style.background = “white”;
      return target;                                  }
```

Event-driven control flow with Arrowlets

- The state machine (from before):

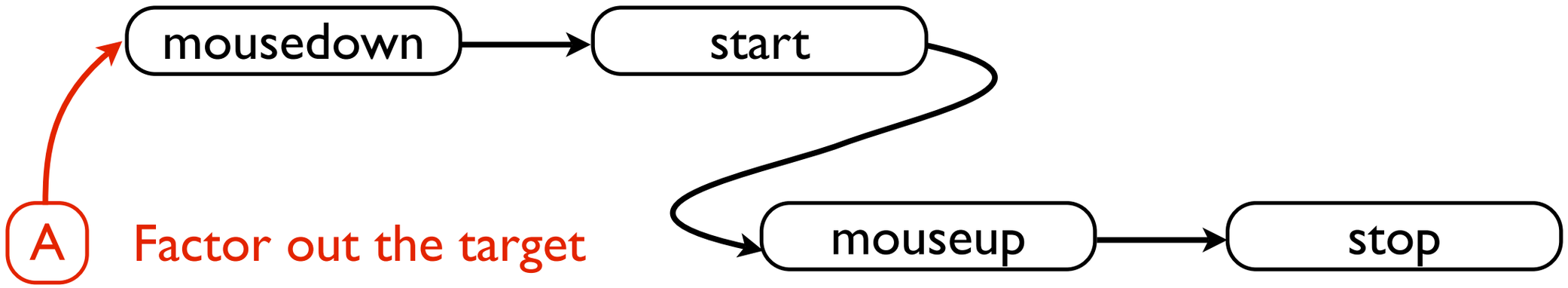


- Re-write start/stop without plumbing:

```
function start(target, event)
{     target.style.background = “yellow”;
      return target;                          }
function stop(target, event,)
{     target.style.background = “white”;
      return target;                          }
```

Event-driven control flow with Arrowlets

- The state machine (from before):

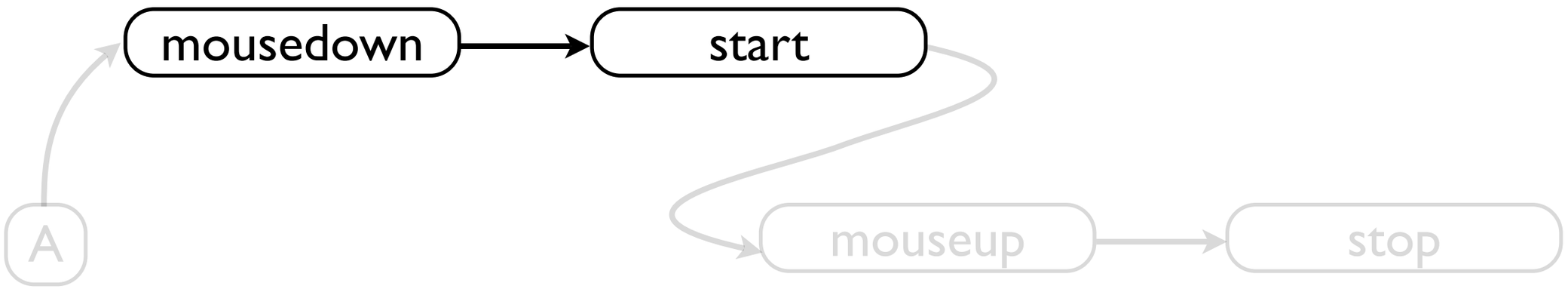


- Re-write start/stop without plumbing:

```
function start(target, event)
{     target.style.background = “yellow”;
      return target;                          }
function stop(target, event,)
{     target.style.background = “white”;
      return target;                          }
```

Event-driven control flow with Arrowlets

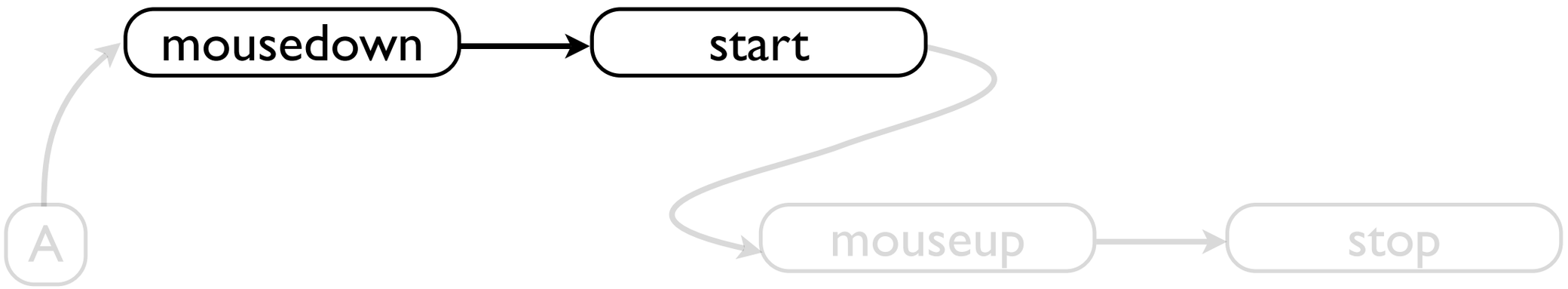
- The state machine:



- Compose using Arrowlets:

Event-driven control flow with Arrowlets

- The state machine:

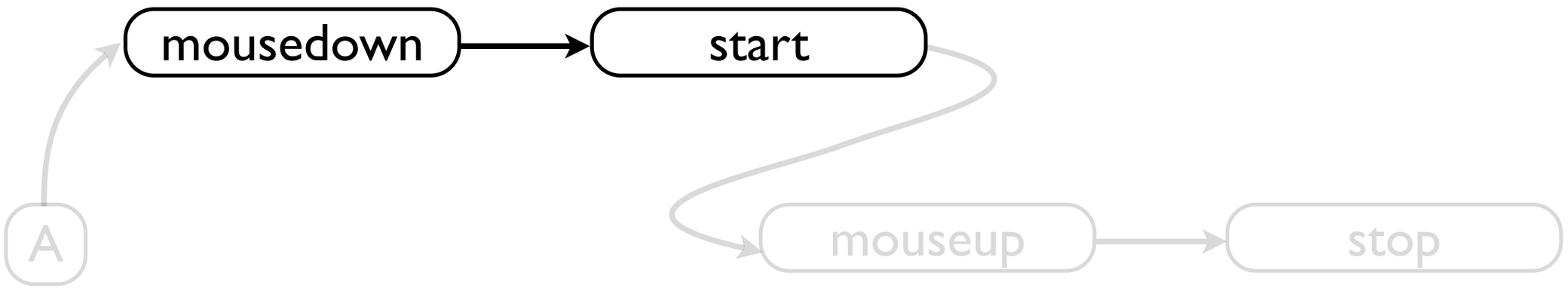


- Compose using Arrowlets:

```
var step1 = EventA("mousedown").bind(start);
```

Event-driven control flow with Arrowlets

- The state machine:



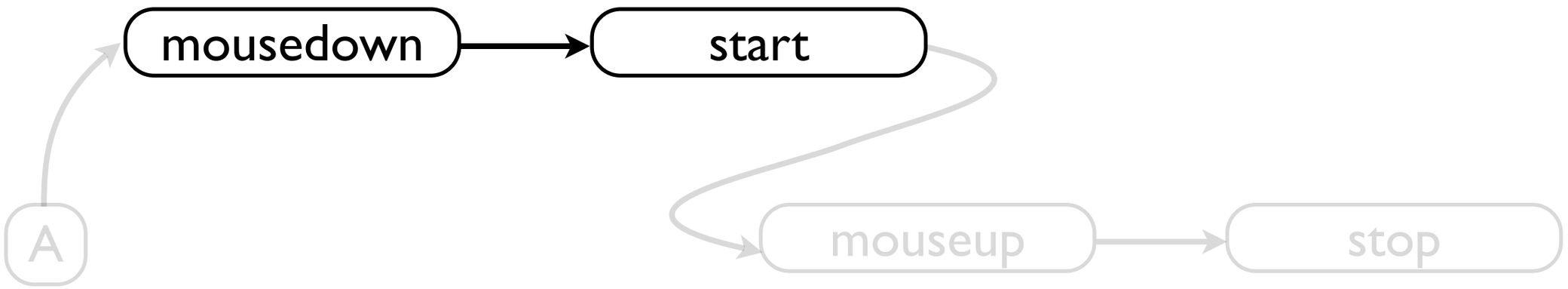
- Compose using Arrowlets:

```
var step1 = EventA("mousedown").bind(start);
```

Wait for "mousedown" on the input

Event-driven control flow with Arrowlets

- The state machine:



- Compose using Arrowlets:

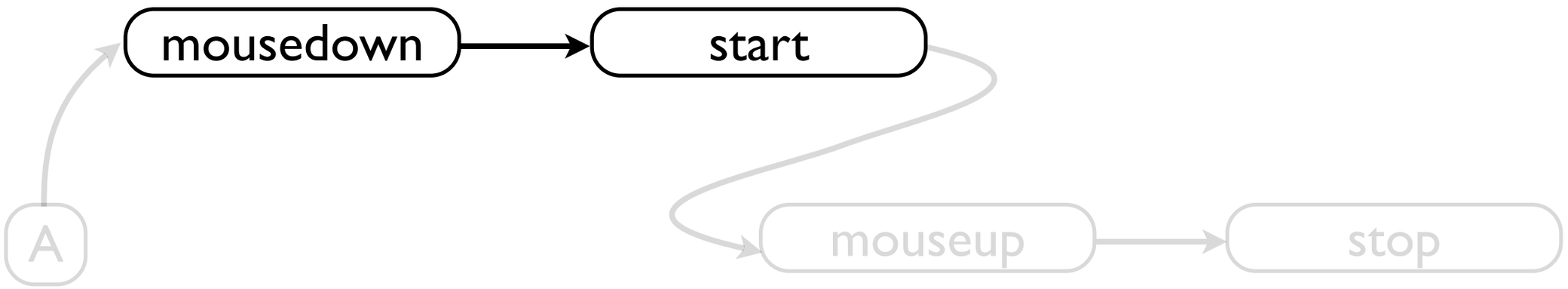
```
var step1 = EventA("mousedown").bind(start);
```

Wait for "mousedown" on the input

then

Event-driven control flow with Arrowlets

- The state machine:



- Compose using Arrowlets:

```
var step1 = EventA("mousedown").bind(start);
```

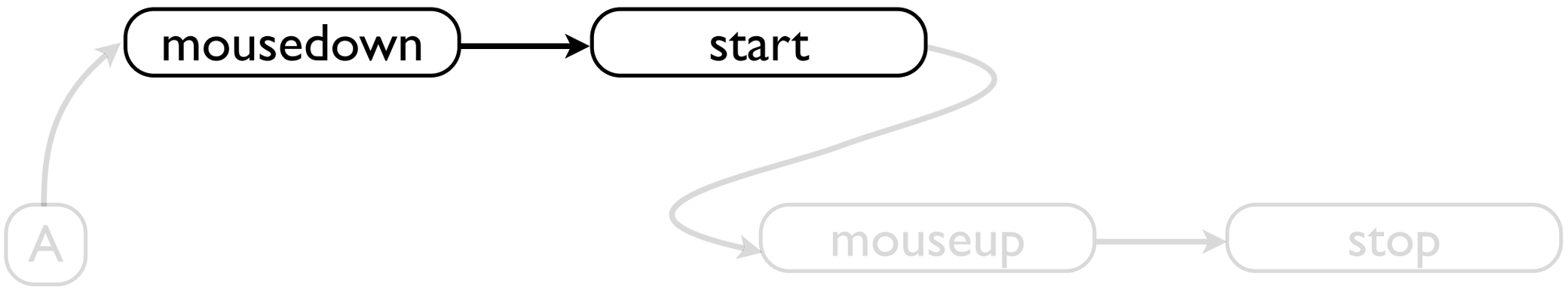
Wait for "mousedown" on the input

call start

then

Event-driven control flow with Arrowlets

- The state machine:

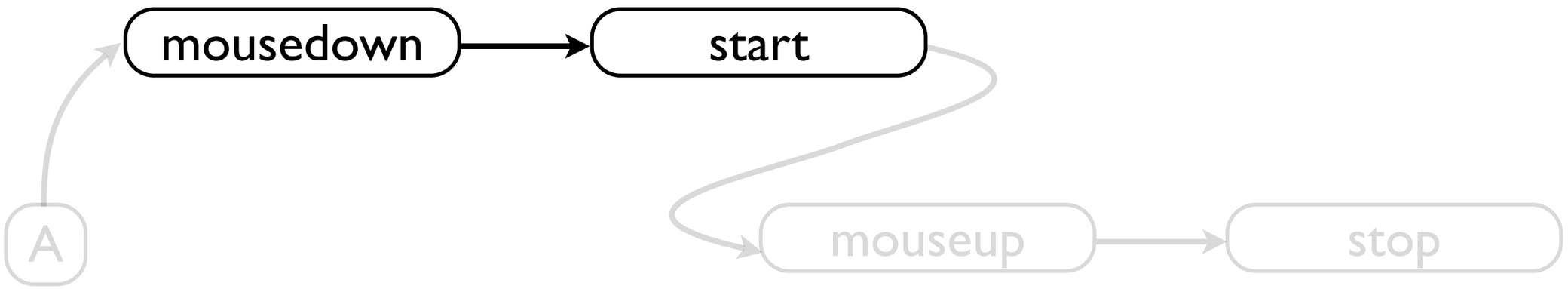


- Compose using Arrowlets:

```
var step1 = EventA("mousedown").bind(start);
```

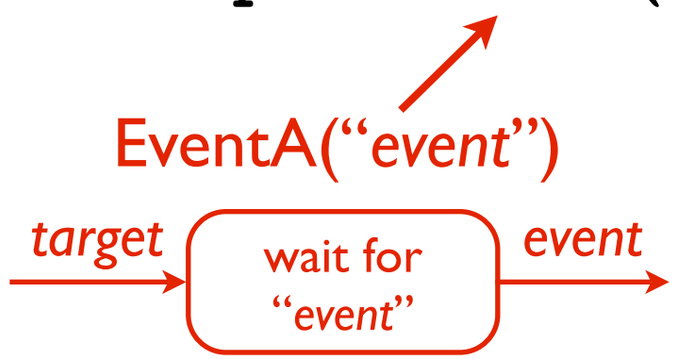
Event-driven control flow with Arrowlets

- The state machine:



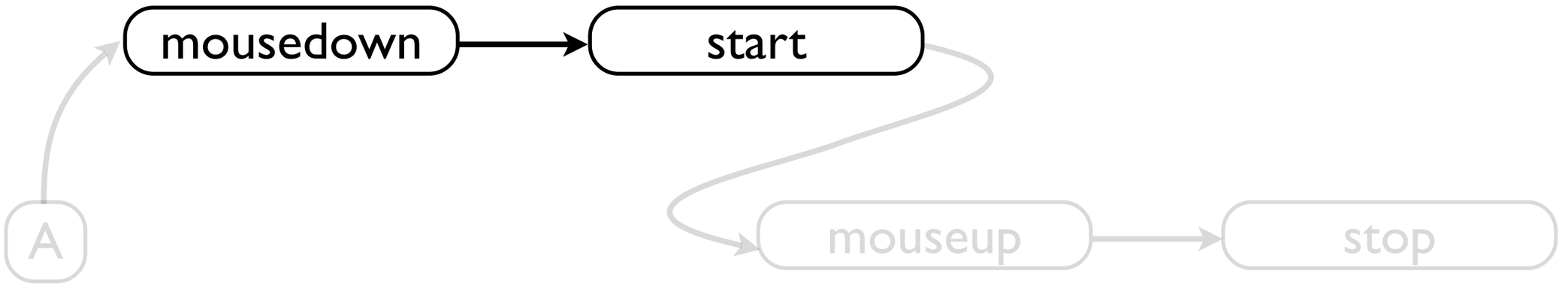
- Compose using Arrowlets:

```
var step1 = EventA("mousedown").bind(start);
```



Event-driven control flow with Arrowlets

- The state machine:

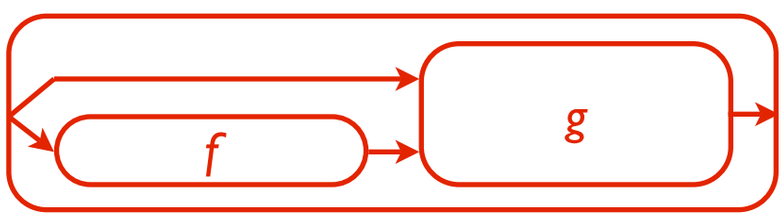


- Compose using Arrowlets:

```
var step1 = EventA("mousedown").bind(start);
```

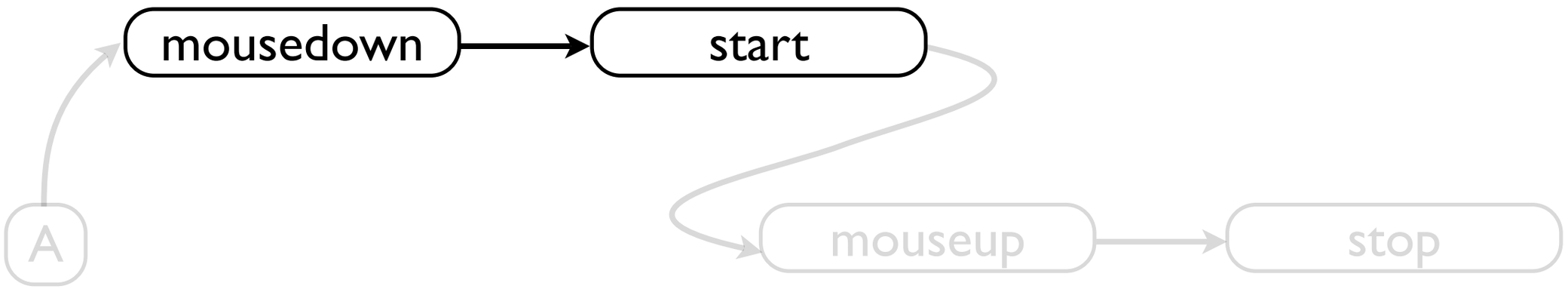
EventA("*event*")

f.bind(*g*) ≈ *g*(*f*(*x*), *x*)



Event-driven control flow with Arrowlets

- The state machine:

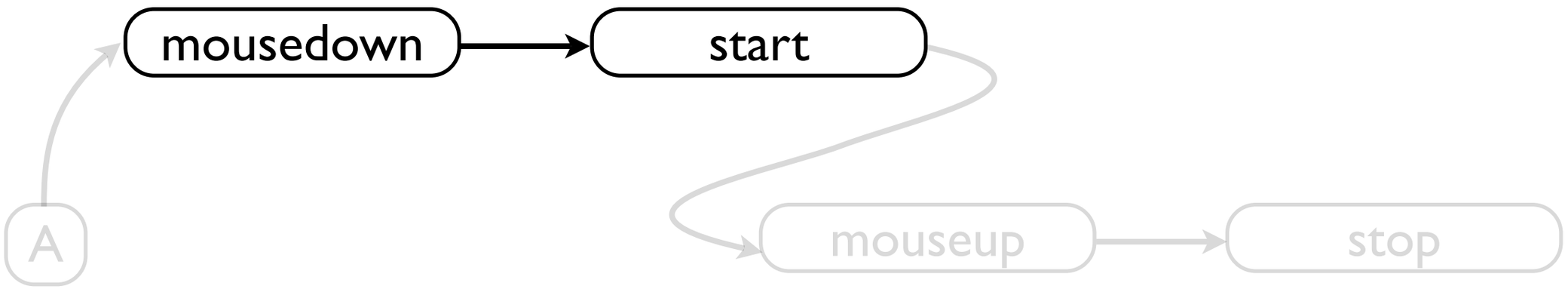


- Compose using Arrowlets:

```
var step1 = EventA("mousedown").bind(start);
```

Event-driven control flow with Arrowlets

- The state machine:



- Compose using Arrowlets:

Combinators compose arrows

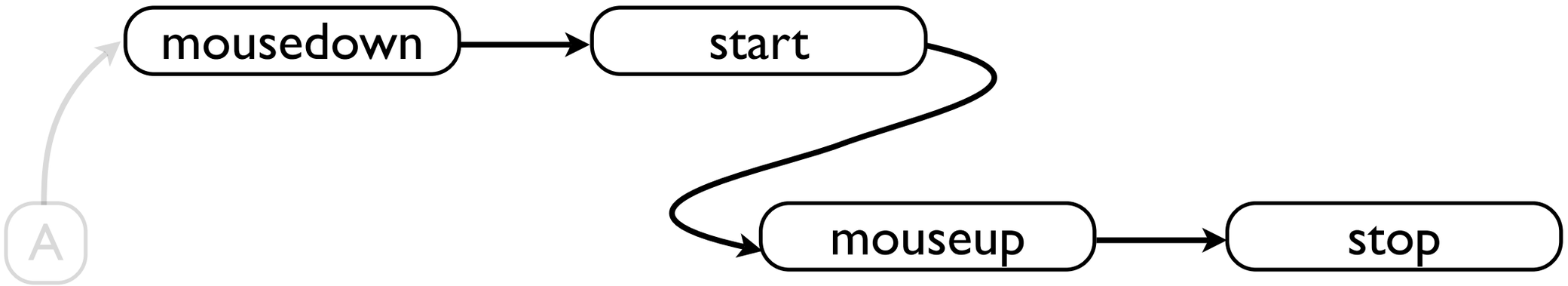
```
var step1 = EventA("mousedown").bind(start);
```

Arrows include:

- wrapped asynchronous functions
- regular functions

Event-driven control flow with Arrowlets

- The state machine:

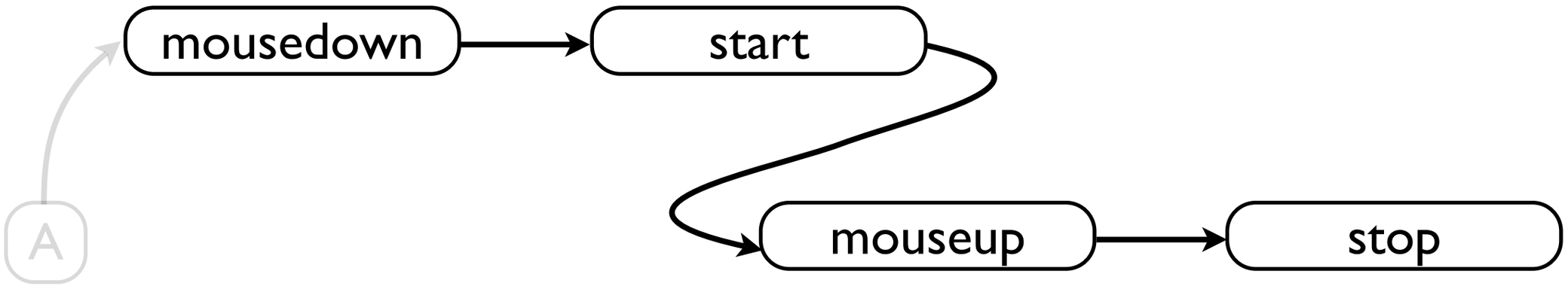


- Compose using Arrowlets:

```
var step1 = EventA(“mousedown”).bind(start);
var step2 = EventA(“mouseup”).bind(stop);
var step1and2 = step1.next(step2);
```

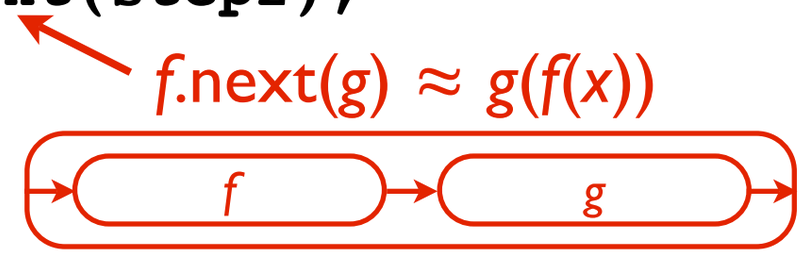
Event-driven control flow with Arrowlets

- The state machine:



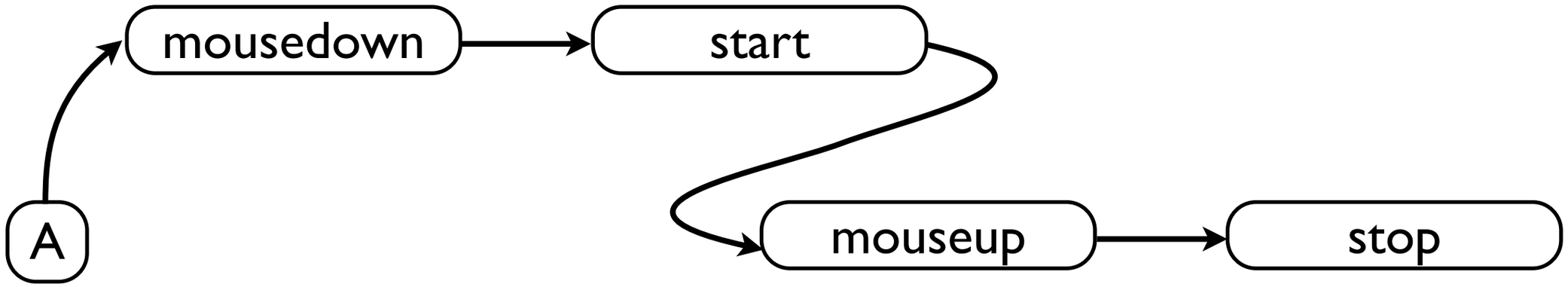
- Compose using Arrowlets:

```
var step1 = EventA("mousedown").bind(start);
var step2 = EventA("mouseup").bind(stop);
var step1and2 = step1.next(step2);
```



Event-driven control flow with Arrowlets

- The state machine:

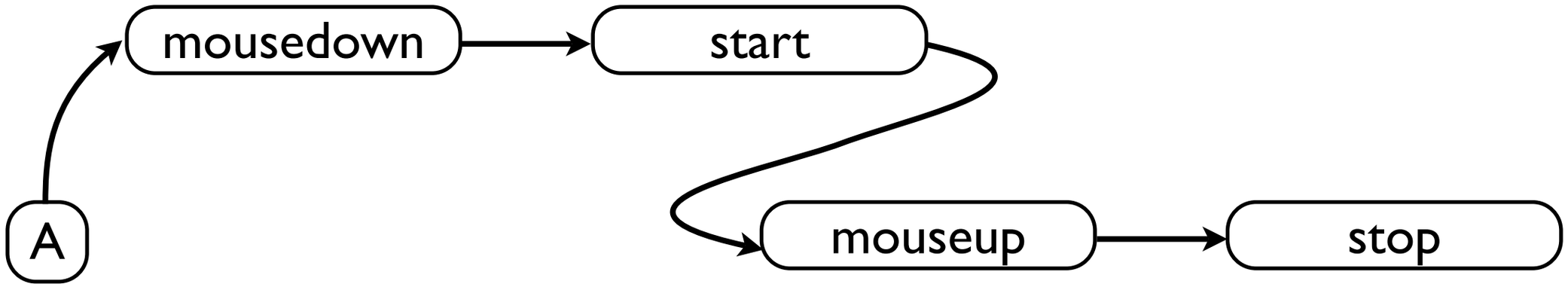


- Compose using Arrowlets:

```
var step1 = EventA("mousedown").bind(start);
var step2 = EventA("mouseup").bind(stop);
var step1and2 = step1.next(step2);
step1and2.run(A);
```

Event-driven control flow with Arrowlets

- The state machine:



- Compose using Arrowlets:

```
var step1 = EventA("mousedown").bind(start);
var step2 = EventA("mouseup").bind(stop);
var step1and2 = step1.next(step2);
step1and2.run(A);
```

f.run(*x*) begins running the composition with initial input *x*.

Comparing the old way to Arrowlets

Old way:

```
function start(event) {
  A.removeEventListener(“mousedown”, start);
  A.addEventListener(“mouseup”, stop);
  A.style.background = “yellow”;
}

function stop(event) {
  A.removeEventListener(“mouseup”, stop);
  A.style.background = “white”;
}

A.addEventListener(“mousedown”, start);
```

Arrowlets:

```
function start(event, target) {
    target.style.background = “yellow”;
    return target;
}

function stop(event, target) {
    target.style.background = “white”;
    return target;
}

var step1 = EventA(“mousedown”).bind(start);
var step2 = EventA(“mouseup”).bind(stop);
var step1and2 = step1.next(step2);
step1and2.run(A);
```

Comparing the old way to Arrowlets

Old way:

```
function start(event) {
  A.removeEventListener(“mousedown”, start);
  A.addEventListener(“mouseup”, stop);
  A.style.background = “yellow”;
}

function stop(event) {
  A.removeEventListener(“mouseup”, stop);
  A.style.background = “white”;
}

A.addEventListener(“mousedown”, start);
```

Arrowlets:

```
function start(event, target) {
    target.style.background = “yellow”;
    return target;
}

function stop(event, target) {
    target.style.background = “white”;
    return target;
}

var step1 = EventA(“mousedown”).bind(start);
var step2 = EventA(“mouseup”).bind(stop);
var step1and2 = step1.next(step2);
step1and2.run(A);
```

“Plumbing” is completely separate from “action”, and in one place

Comparing the old way to Arrowlets

Old way:

```
function start(event) {
  A.removeEventListener(“mousedown”, start);
  A.addEventListener(“mouseup”, stop);
  A.style.background = “yellow”;
}

function stop(event) {
  A.removeEventListener(“mouseup”, stop);
  A.style.background = “white”;
}

A.addEventListener(“mousedown”, start);
```

Arrowlets:

```
function start(event, target) {
    target.style.background = “yellow”;
    return target;
}

function stop(event, target) {
    target.style.background = “white”;
    return target;
}

var step1 = EventA(“mousedown”).bind(start);
var step2 = EventA(“mouseup”).bind(stop);
var step1and2 = step1.next(step2);
step1and2.run(A);
```

Comparing the old way to Arrowlets

Old way:

```
function start(event) {
  A.removeEventListener(“mousedown”, start);
  A.addEventListener(“mouseup”, stop);
  A.style.background = “yellow”;
}

function stop(event) {
  A.removeEventListener(“mouseup”, stop);
  A.style.background = “white”;
}

A.addEventListener(“mousedown”, start);
```

Arrowlets:

```
function start(event, target) {
    target.style.background = “yellow”;
    return target;
}

function stop(event, target) {
    target.style.background = “white”;
    return target;
}

var step1 = EventA(“mousedown”).bind(start);
var step2 = EventA(“mouseup”).bind(stop);
var step1and2 = step1.next(step2);
step1and2.run(A);
```

Target no longer hard-coded

Comparing the old way to Arrowlets

Old way:

```
function start(event) {
    A.removeEventListener(“mousedown”, start);
    A.addEventListener(“mouseup”, stop);
    A.style.background = “yellow”;
}

function stop(event) {
    A.removeEventListener(“mouseup”, stop);
    A.style.background = “white”;
}

A.addEventListener(“mousedown”, start);
```

Arrowlets:

```
function start(event, target) {
    target.style.background = “yellow”;
    return target;
}

function stop(event, target) {
    target.style.background = “white”;
    return target;
}

var step1 = EventA(“mousedown”).bind(start);
var step2 = EventA(“mouseup”).bind(stop);
var step1and2 = step1.next(step2);
step1and2.run(A);
```

Target no longer hard-coded

Event handlers are decoupled

Comparing the old way to Arrowlets

Old way:

```
function start(event) {
  A.removeEventListener(“mousedown”, start);
  A.addEventListener(“mouseup”, stop);
  A.style.background = “yellow”;
}

function stop(event) {
  A.removeEventListener(“mouseup”, stop);
  A.style.background = “white”;
}

A.addEventListener(“mousedown”, start);
```

Arrowlets:

```
function start(event, target) {
    target.style.background = “yellow”;
    return target;
}

function stop(event, target) {
    target.style.background = “white”;
    return target;
}

var step1 = EventA(“mousedown”).bind(start);
var step2 = EventA(“mouseup”).bind(stop);
var step1and2 = step1.next(step2);
step1and2.run(A);
```

Target no longer hard-coded

Event handlers are decoupled

Composition is modular

Comparing the old way to Arrowlets

Old way:

```
function start(event) {
  A.removeEventListener(“mousedown”, start);
  A.addEventListener(“mouseup”, stop);
  A.style.background = “yellow”;
}

function stop(event) {
  A.removeEventListener(“mouseup”, stop);
  A.style.background = “white”;
}

A.addEventListener(“mousedown”, start);
```

Arrowlets:

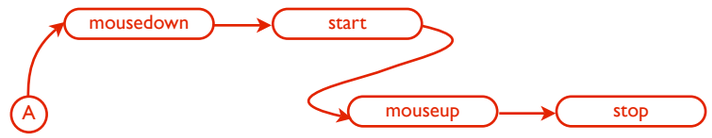
```
function start(event, target) {
    target.style.background = “yellow”;
    return target;
}

function stop(event, target) {
    target.style.background = “white”;
    return target;
}

var step1 = EventA(“mousedown”).bind(start);
var step2 = EventA(“mouseup”).bind(stop);
var step1and2 = step1.next(step2);
step1and2.run(A);
```

Target no longer hard-coded

Event handlers are decoupled

Composition is modular

Straightforward translation of state machine

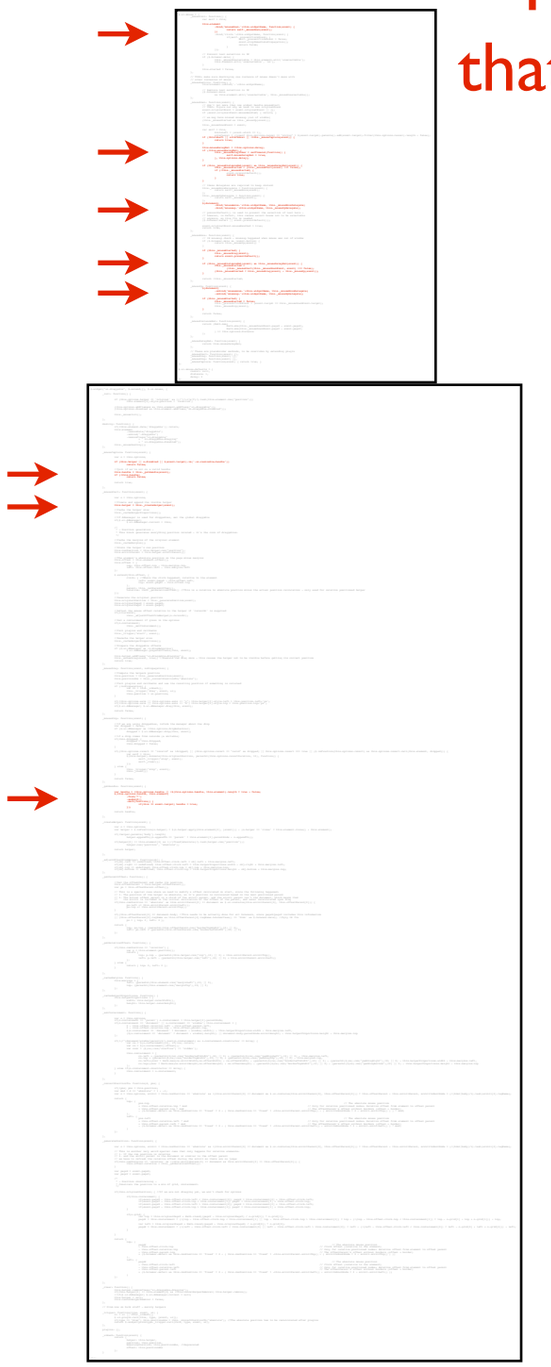


Real example: drag-and-drop

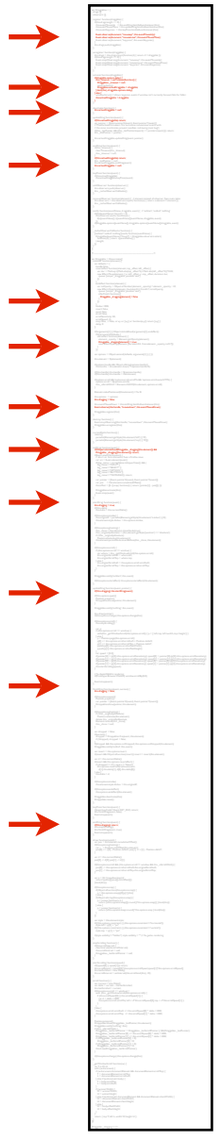
- Common in many JavaScript libraries
  - Browsers lack built-in support
- Rich interaction in state machine:
  - ≥ 3 states (depending on features)
  - includes branches and loops

Drag-and-drop in 4 popular JavaScript libraries

→ The red lines are the "plumbing" code that register event handlers or implement the state machine

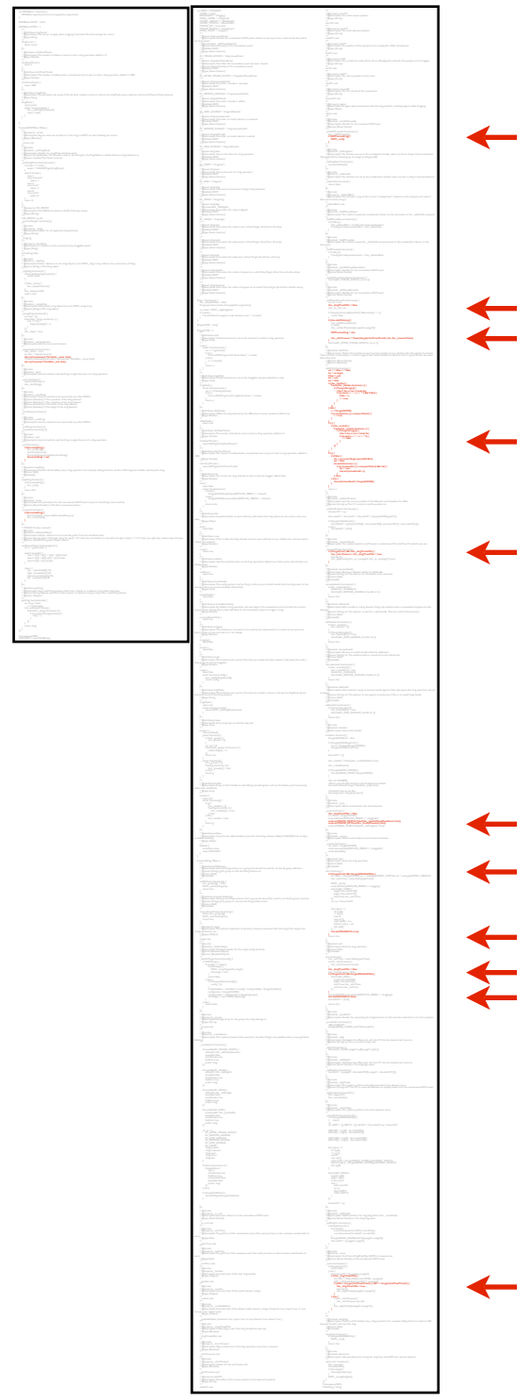


Libary A
(532 lines)



Libary B
(445 lines)

Libary C
(243 lines)



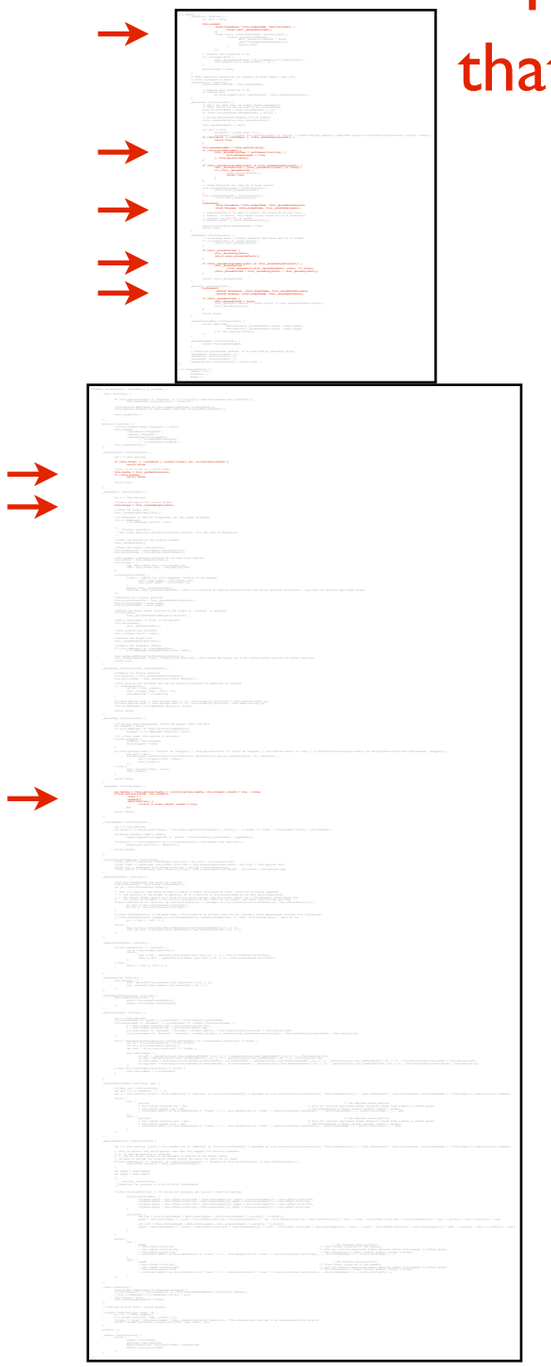
Libary D
(1321* lines)

* a *lot* of comments

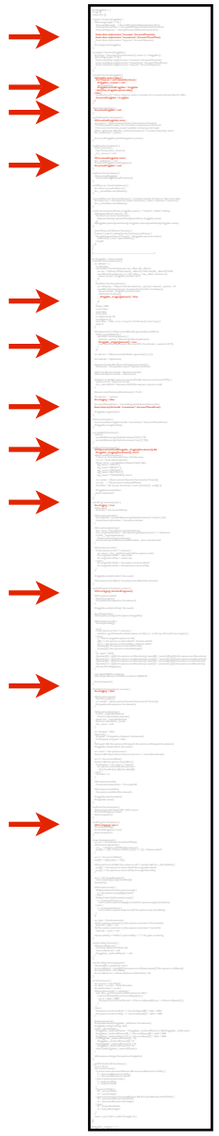
Drag-and-drop in 4 popular JavaScript libraries

→ The red lines are the "plumbing" code that register event handlers or implement the state machine

They are all over the code!



Libary A
(532 lines)



Libary B
(445 lines)

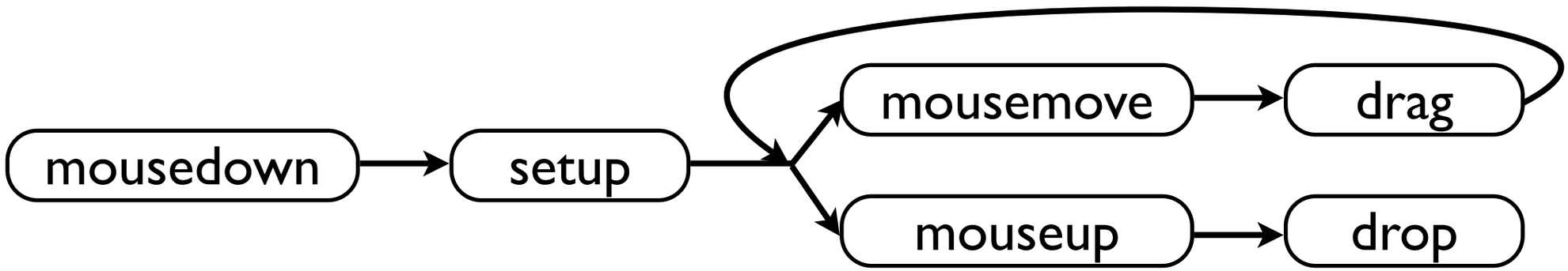
Libary C
(243 lines)

Libary D
(1321* lines)

* a *lot* of comments

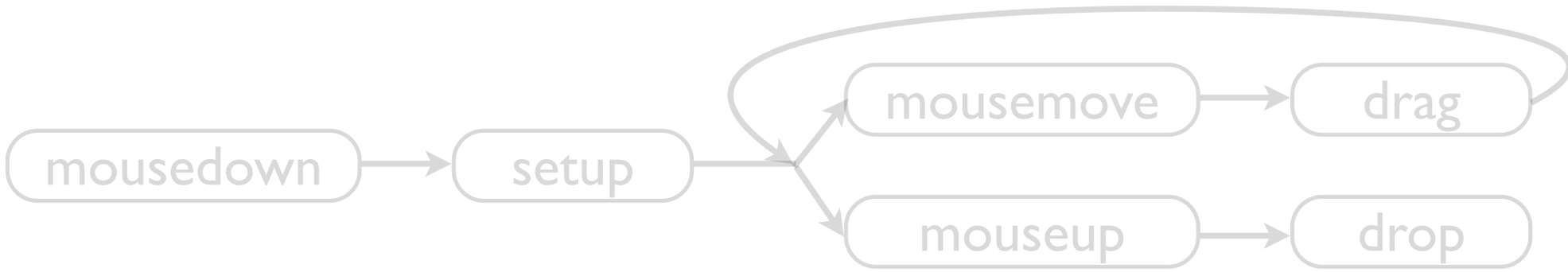
Drag-and-drop using Arrowlets

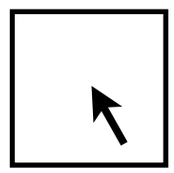
- State machine:



Drag-and-drop using Arrowlets

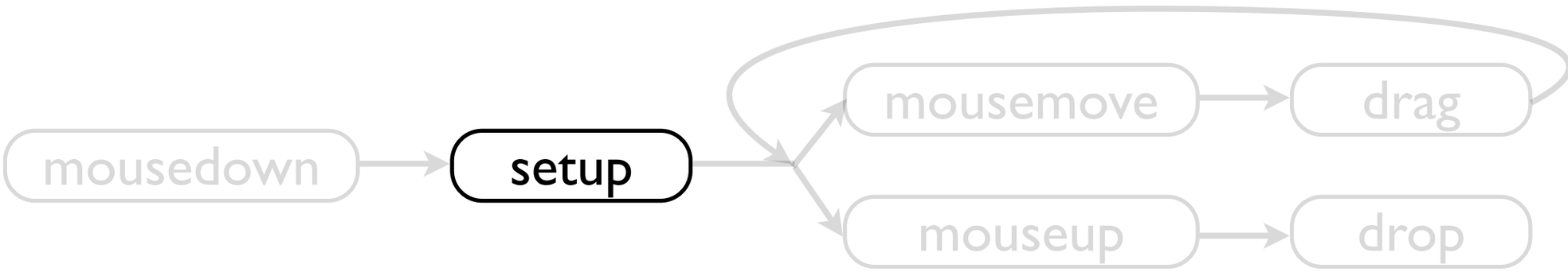
- State machine:





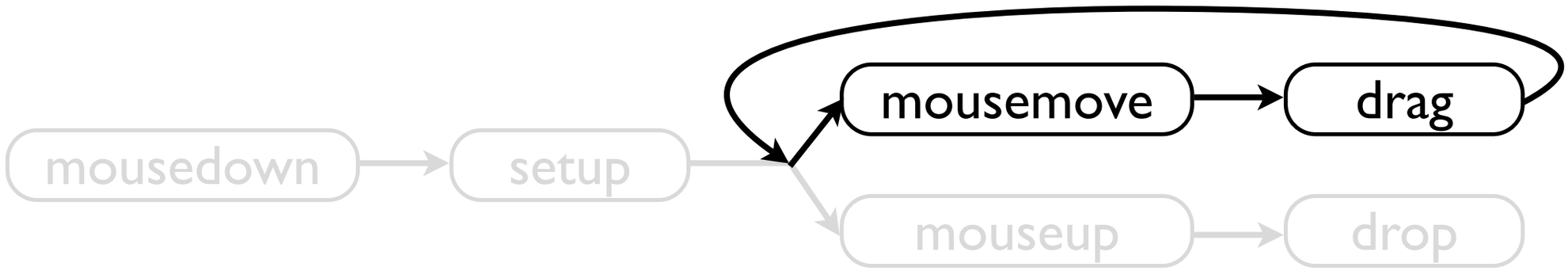
Drag-and-drop using Arrowlets

- State machine:



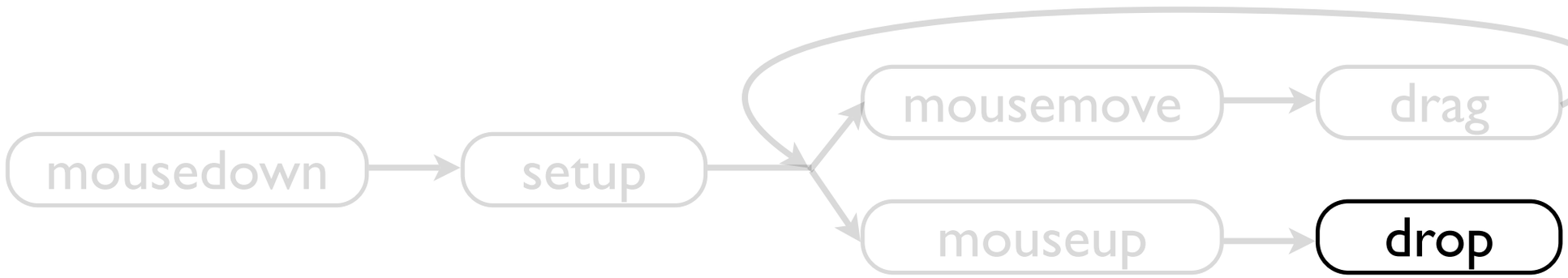
Drag-and-drop using Arrowlets

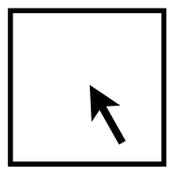
- State machine:



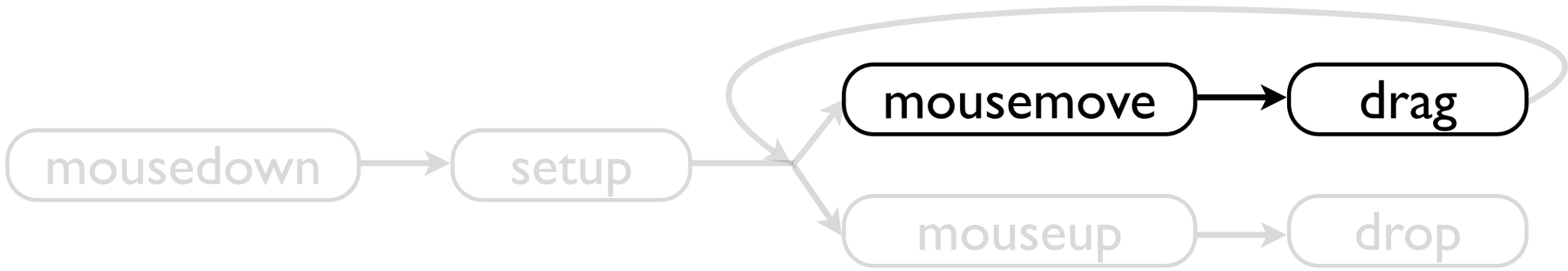
Drag-and-drop using Arrowlets

- State machine:



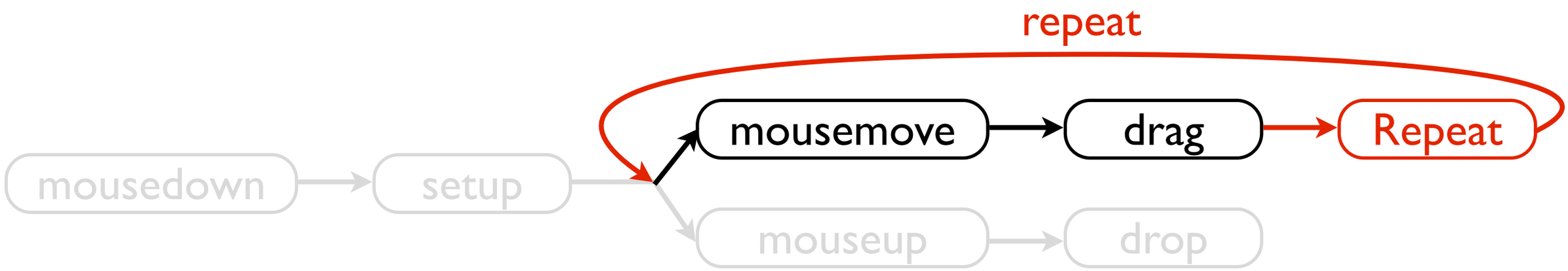


Repeating drag



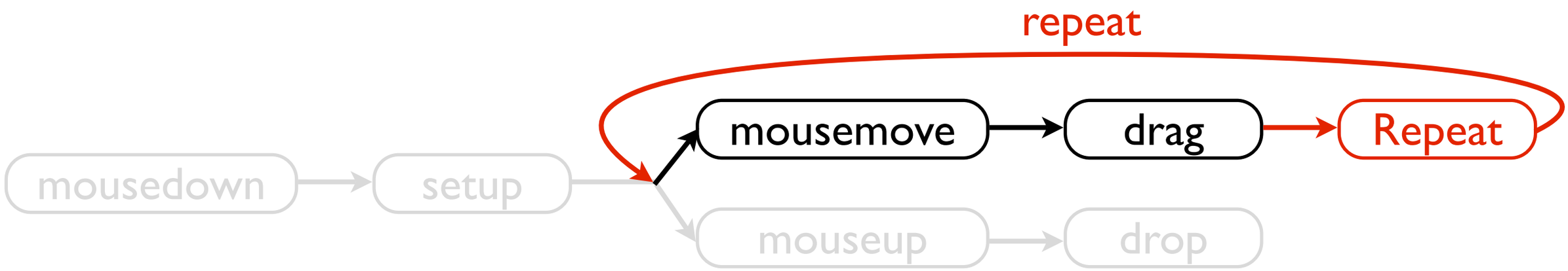

```
EventA("mousemove").bind(drag)
```

Repeating drag




```
(       (EventA(“mousemove”).bind(drag)).next(Repeat)

).repeat();
```

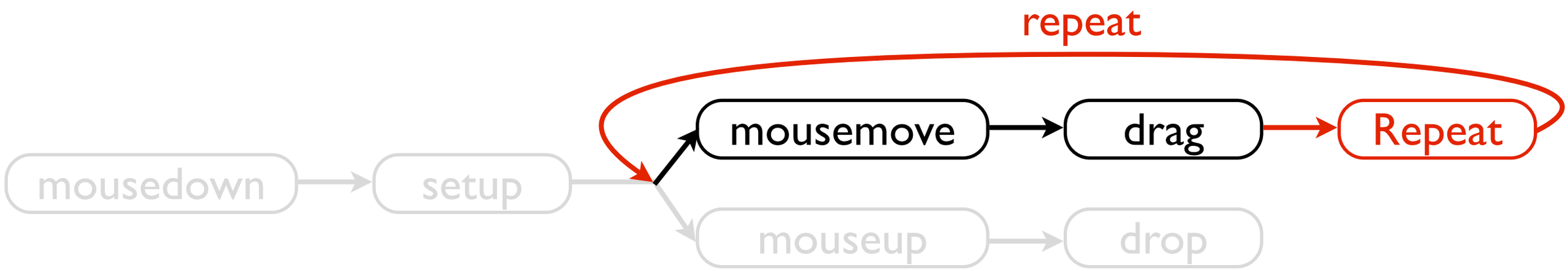
Repeating drag




```
(       (EventA("mousemove").bind(drag)).next(Repeat)

).repeat();
```

Repeat takes an input, and wraps it in an object tagged "Repeat"

Repeating drag



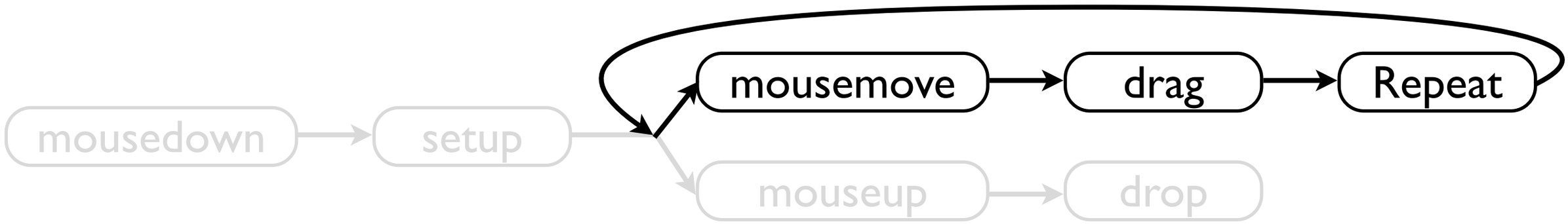

```
(       (EventA("mousemove").bind(drag)).next(Repeat)

).repeat();
```

Repeat takes an input, and wraps it in an object tagged "Repeat"

f.repeat() runs *f*, and if *f* outputs:

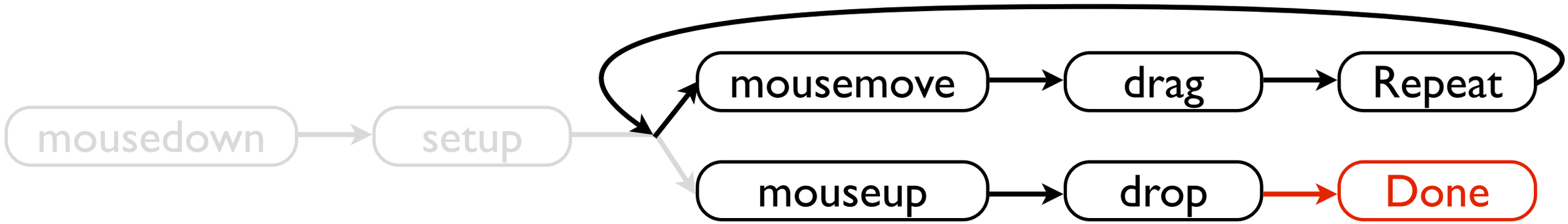
- Repeat(*x*), then runs *f* again with *x* as input
- Done(*x*), then outputs *x*

Dragging or dropping



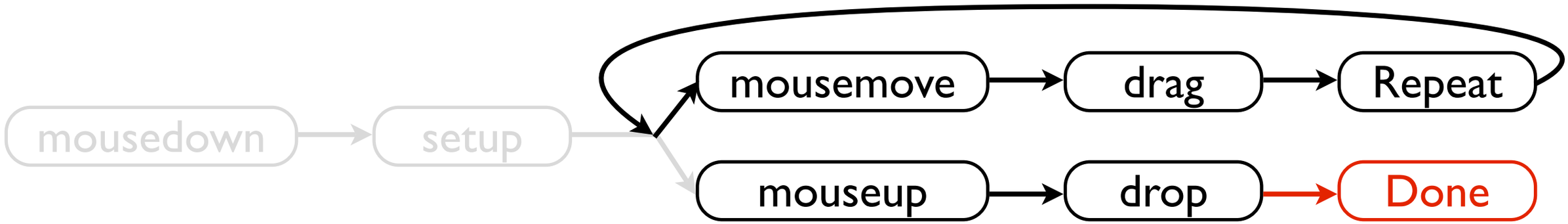

```
(       (EventA(“mousemove”).bind(drag)).next(Repeat)

).repeat();
```

Dragging or dropping




```
(       (EventA("mousemove").bind(drag)).next(Repeat)
        (EventA("mouseup").bind(drop)).next(Done)
).repeat();
```

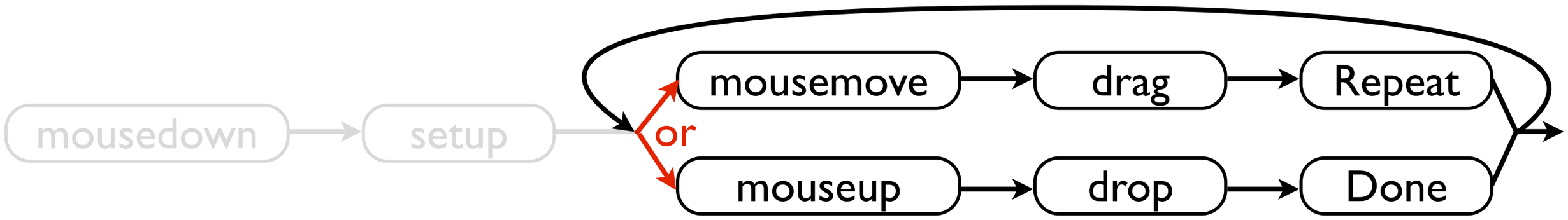
Dragging or dropping




```
(       (EventA("mousemove").bind(drag)).next(Repeat)
        (EventA("mouseup").bind(drop)).next(Done)
).repeat();
```

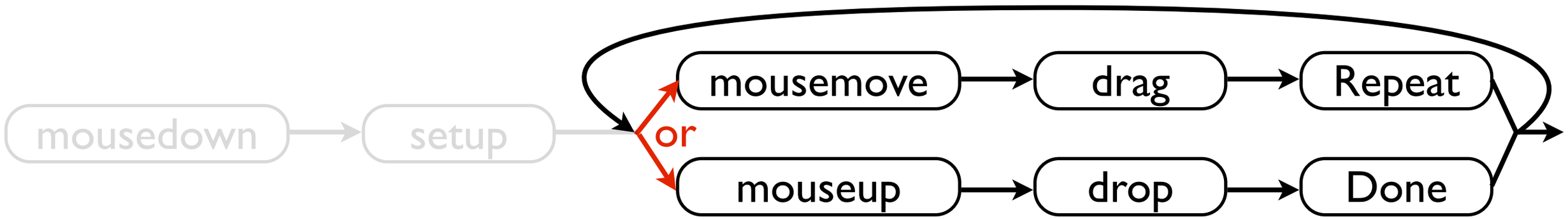
Tag to stop repeating

Dragging or dropping




```
(    ( (EventA("mousemove").bind(drag)).next(Repeat) )
  .or( (EventA("mouseup").bind(drop)).next(Done) )
).repeat();
```

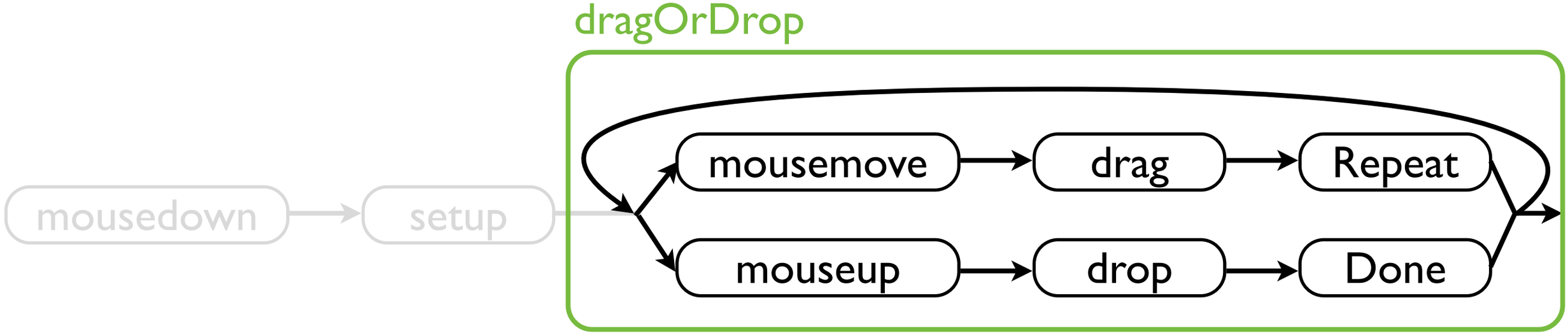
Dragging or dropping




```
(     ( (EventA("mousemove").bind(drag)).next(Repeat) )
   .or( (EventA("mouseup").bind(drop)).next(Done) )
).repeat();
```

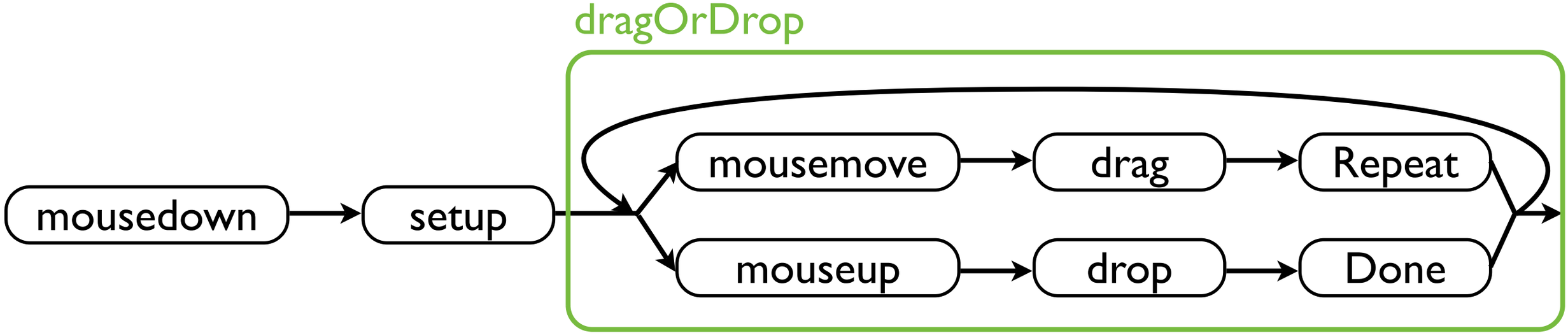
f.or(*g*) allows only *f* or *g* to run–whichever is triggered first–and cancels the other

Dragging or dropping



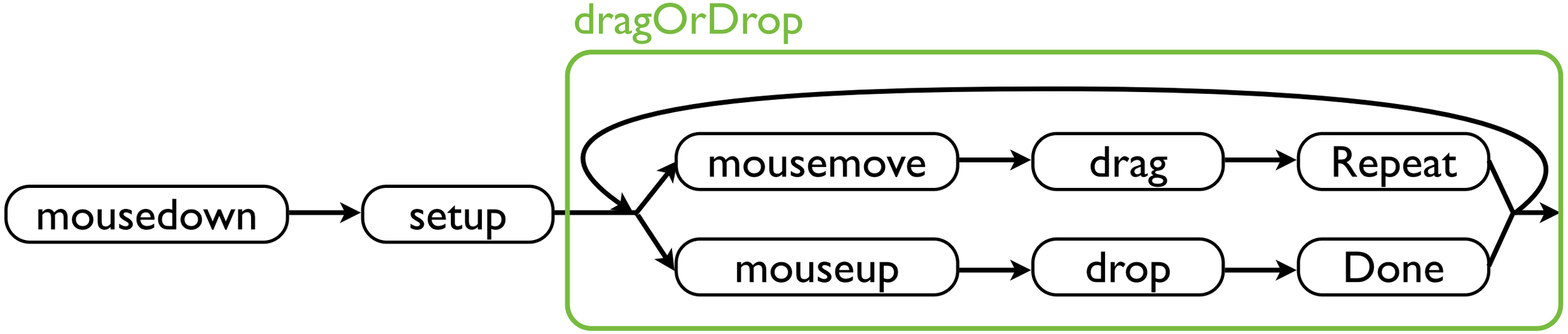

```
var dragOrDrop =
    (     ( (EventA(“mousemove”).bind(drag)).next(Repeat) )
      .or( (EventA(“mouseup”).bind(drop)).next(Done) )
    ).repeat();
```

Setup, install and run




```
var dragOrDrop =
    (     ( (EventA(“mousemove”).bind(drag)).next(Repeat) )
      .or( (EventA(“mouseup”).bind(drop)).next(Done) )
    ).repeat();

var dragAndDrop =
    (EventA(“mousedown”).bind(setup)).next(dragOrDrop);
```

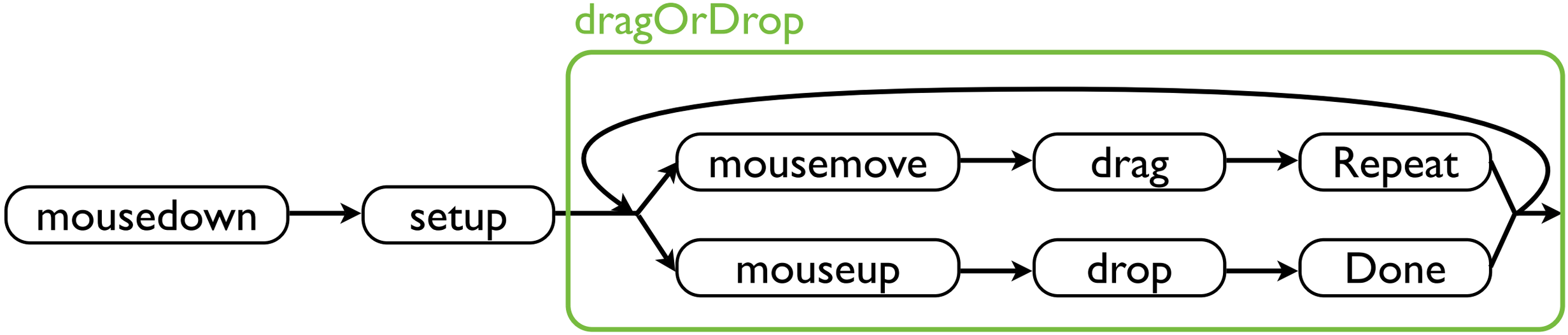
Setup, install and run




```
var dragOrDrop =
    (     ( (EventA(“mousemove”).bind(drag)).next(Repeat) )
      .or( (EventA(“mouseup”).bind(drop)).next(Done) )
    ).repeat();

var dragAndDrop =
    (EventA(“mousedown”).bind(setup)).next(dragOrDrop);

dragAndDrop.run(target);
```

This is the entire control flow of basic drag-and-drop!

Canceling drag-and-drop

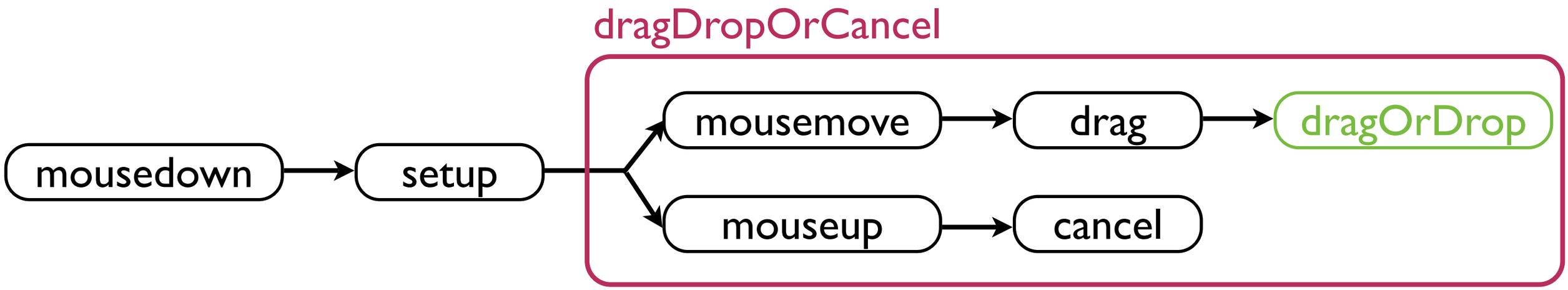


- Want different handler to cancel drag-and-drop:
  - after "mousedown" but before "mousemove"
  - in old way, need to modify setup, drag and drop

Canceling drag-and-drop

dragOrDrop

Canceling drag-and-drop



- Add a branch between setup and dragOrDrop:

```
var dragDropOrCancel =
      ((EventA("mousemove").bind(drag)).next(dragOrDrop))
   .or((EventA("mouseup").bind(cancel)));

var dragAndDropWithCancel =
   (EventA("mousedown").bind(setup)).next(dragDropOrCancel);
```

Re-use dragOrDrop

Re-use drag-and-drop in many ways

- Trigger on “mouseover”:

```
(EventA("mouseover").bind(setup))
  .next(dragDropOrCancel)
```

- Jigsaw game:

```
(nextPiece
  .next(EventA("click").bind(setup))
  .next((dragOrDrop.next(repeatIfWrongPlace)).repeat())
).repeat()
```

- Every composition can be used simultaneously; no need to duplicate code!

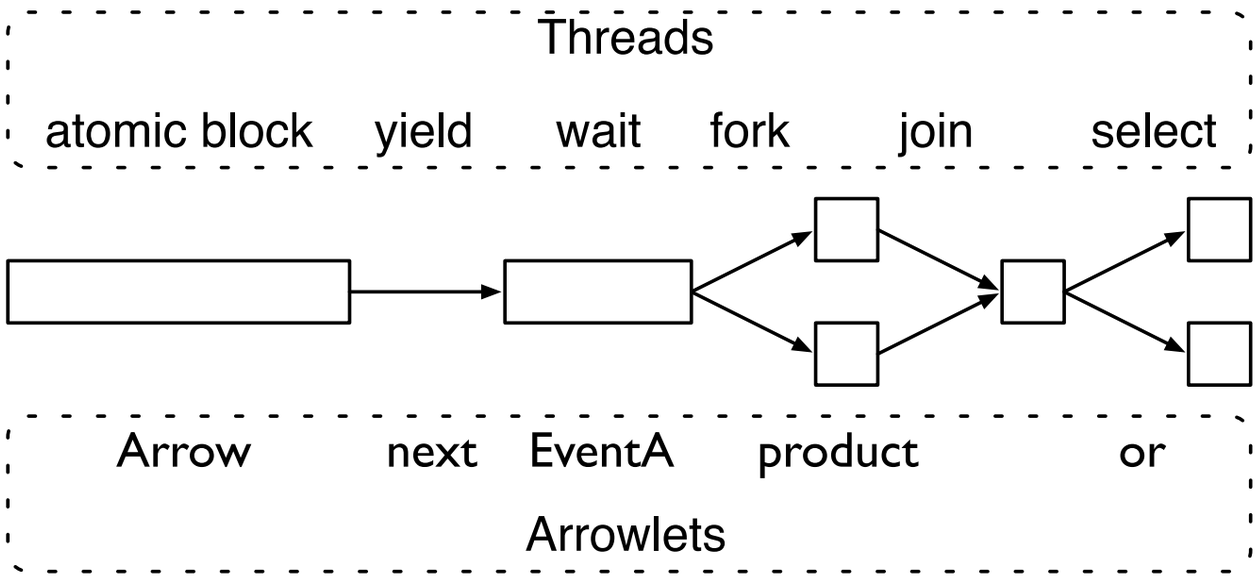
Implementation

- API, semantics inspired by Haskell's *Arrows*
- *Continuation-passing style* under the hood
- *Trampoline* to overcome call stack limit
  - also to limit setTimeout() latency

More implementation details available in paper

Related Work

- *Flapjax* (OOPSLA '09): JavaScript library based on functional reactive programming
  - compose *event streams* vs. state machines for Arrowlets
- Threads and events:



Conclusion

- We believe Arrowlets to be an elegant approach to event-driven programming in JavaScript
- Arrowlets enables finer modularity, flexible composition, and is easier to understand
- Available at http://www.cs.umd.edu/projects/PL/arrowlets

This slide is intentionally left blank.

What about Monads?

Deferred (Twisted):

```
function add1(x) {
  return Deferred(x + 1);
}
```

```
function add2(x) {
  return add1(x)
    .addCallback(add1);
}
```

```
function addN(N, x) {
  var a = add1(x);
  for (var i = 1; i < N; i++)
    a = a.addCallback(add1)
  return a;
}
```

Arrowlets:

```
function add1(x) {
  return x + 1;
}
```

```
var add2 = add1.next(add1);
```

```
var addN = function(N) {
  var a = add1;
  for (var i = 1; i < N; i++)
    a = a.next(add1);
  return a;
}.bindapp();
```

Implementation of *Arrowlets*

Haskell's *Arrows*

- Arrowlets is based on Haskell's Arrows:

```
class Arrow a where
    arr   :: (b -> c) -> a b c
    (>>>) :: a b c -> a c d -> a b d
```

- Type class `Arrow a` supports operations:
  - `arr f` : lifts function `f` into the type `Arrow a`
  - `f >>> g` : composes arrows `f` and `g` in sequence

Arrows in JavaScript

- Simplest arrows are functions (->):

```
instance Arrow (->) where
    arr f     = f          {- identity function -}
    (f >>> g) = g (f x)    {- function composition -}
```

- In JavaScript, augment Function prototype:

```
Function.prototype.A = function() { /* arr */
    return this;
}
Function.prototype.next = function(g) { /* >>> */
    var f = this;
    g = g.A(); /* ensure g is a function */
    return function(x) { return g(f(x)); }
}
```

CPS and Event Arrows

- Problem: `addEventListener` and friends "continue" via a callback parameter
- Solution: use *continuation-passing style*:

```
function CpsA(cps) { /* constructor */
    this.cps = cps; /* cps :: (x, k) -> () */
}
Function.prototype.CpsA = function() { /* lift */
    var f = this;
    /* wrap f in CPS function with "callback" k */
    return new CpsA(function(x, k) {
        k(f(x));
    });
}
```

CPS and Event Arrows

- Finally, wrap `addEventListener` with CpsA:

```
function SimpleEventA(eventname) {
    if (!(this instanceof SimpleEventA)) /*“new” idiom*/
        return new 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);
});
```