FIT2102 Programming Paradigms Lecture 5

Combinators
Lambda Calculus

Faculty of Information Technology

MONASH University

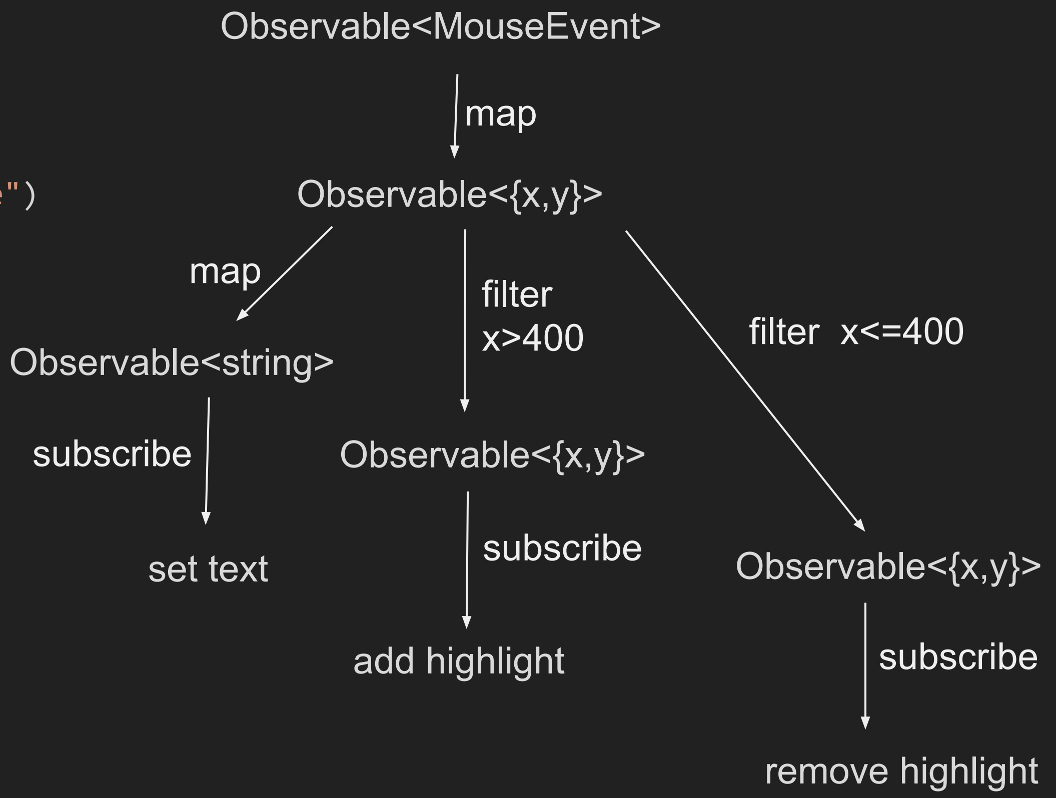
Learning Outcomes

- Create interactive programs using Observable
- Create new functions from old functions using Combinators
- Create powerful declarative programs
  using Higher-order Functions and Combinators
- Relate the lambda calculus to functional programming
- Apply conversion and reduction rules to simplify lambda expressions

Observable Trees

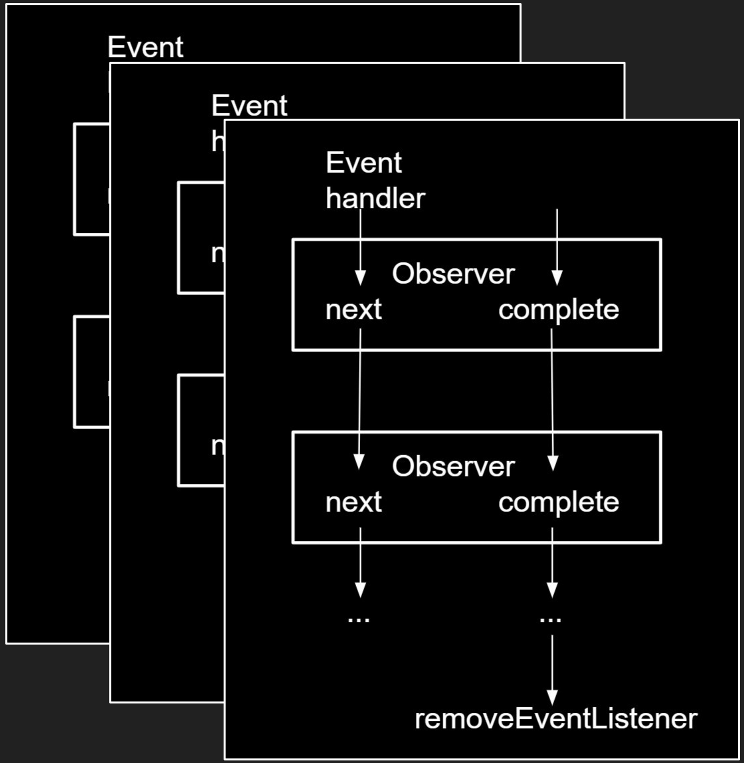
The Observable `o` has three separate child subscriptions:

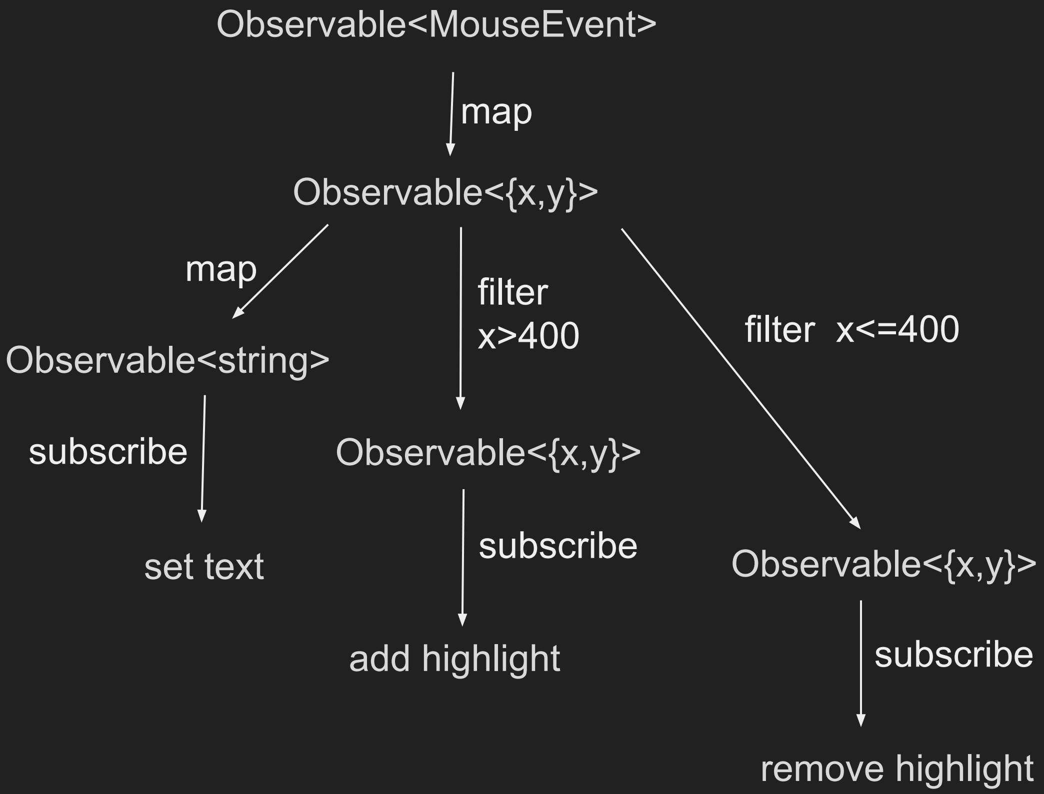
```
const
  pos = document.getElementById("pos"),
  o = Observable
    .fromEvent<MouseEvent>(document, "mousemove")
    .map(({clientX, clientY})=>
           ({x: clientX, y: clientY}));

o.map(({x,y}) => `${x},${y}`)
 .subscribe(s => pos.innerHTML = s);

o.filter(({x}) => x > 400)
 .subscribe(_ =>
     pos.classList.add('highlight'));

o.filter(({x}) => x <= 400)
 .subscribe(_ =>
     pos.classList.remove('highlight'));
```



Observer Chains

A separate observer chain is created for each subscribe, each with their own event listener:

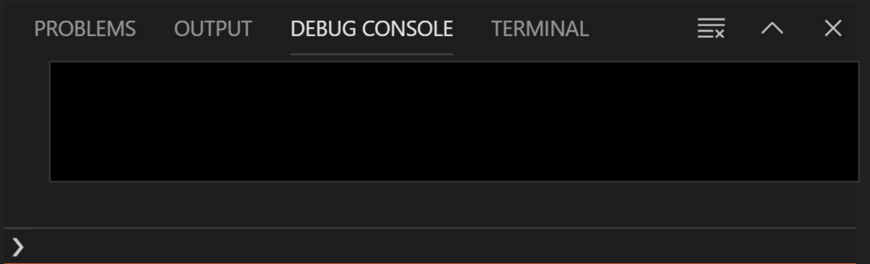




Beware of impurity in Observable chains with multiple subscribes

```
const pos = document.getElementById("pos")!,
  o = Observable
        .fromEvent<MouseEvent>(document, "mousedown")
        .map(({clientX, clientY})=>{
          console.log(`x=${clientX}, y=${clientY}`)  <= side effect: print to console
          return ({x: clientX, y: clientY})
        });

o.map(({x,y}) => `${x},${y}`)
 .subscribe(s => pos.innerHTML = s);

o.filter(({x}) => x > 400)
 .subscribe(_ => pos.classList.add('highlight'));

o.filter(({x}) => x <= 400)
 .subscribe(_ => pos.classList.remove('highlight'));
```

Because the three subscribes cause three separate observer chains to be created, each mousedown event causes the effect to occur three times:



Cons Lists

Can we create lists with only lambda (anonymous) functions?

```
const cons = (x, y) => f => f(x, y)

const aList = cons('Lists', cons("don't", cons("get", cons('any',
                  cons('simpler', cons('than', cons('this',
                      undefined)))))))

const head = list => list((x, y)=> x)

const rest = list => list((x, y)=> y)

head(rest(rest(aList)))

> "get"
```

Cons Lists - Curried

Can we create lists with only lambda (anonymous) functions?

```
const cons = x => y => f => f(x)(y)

const aList = cons('Lists')(cons("don't")(cons("get")(cons('any')(
                    cons('simpler')(cons('than')(cons('this')
                        (null)))))))

const head = list => list(x=> y=> x)

const rest = list => list(x=> y=> y)

head(rest(rest(aList)))
```

Combinators

Combinators are functions which are expressions of only their parameters

They let us combine and transform other functions in various ways

```
const
    I = x=> x
  ,
    K = x=> y=> x
```

Identity: more useful than you might think!
e.g. test map: a.map(I) == a

A function that ignores the second parameter.
Where have we seen this before?

Cons list - with I and K combinators

```
const
    cons = x=> y=> f=> f(x)(y)

const aList =                   cons(3)(null)
```

f => f (3) (null)

```
const
    I = i=> i,
    K = x=> y=> x,
    head = l=> l(K),
    tail = l=> l(K(I))
```

K(I) ≡ K(i=> i) I := i=> i

≡ (x=> y=> x)(i=> i) K := x=> y=> x

≡ y=> (i=> i) beta reduction

≡ y=> x=> x alpha equivalence

(pseudo lambda calculus with JS notation)

Exercise 1

To be announced...

Reducing reduce

```
const aList = cons(1)(cons(2)(cons(3)(undefined)))

function reduce(f,i,l) {
    if (l) {
        return reduce(f, f(i,head(l)), tail(l))
    } else {
        return i;
    }
};

console.log(reduce((x,y)=>x+y, 0, aList))
> 6
```

Reducing reduce

```
const aList = cons(1)(cons(2)(cons(3)(undefined)))

function reduce(f,i,l) {
   return l ?
              reduce(f, f(i,head(l)), tail(l))
            :
              i;

}

console.log(reduce((x,y)=>x+y, 0, aList))
> 6
```

Reducing reduce

```
const aList = cons(1)(cons(2)(cons(3)(undefined)))

function reduce(f,i,l) {
    return l ? reduce(f, f(i,head(l)), tail(l)) : i;
}
console.log(reduce((x,y)=>x+y, 0, aList))
> 6
```

Reducing reduce

```
const aList = cons(1)(cons(2)(cons(3)(undefined)))

const reduce = (f,i,l) =>
   l ? reduce(f, f(i,head(l)), tail(l)) : i

console.log(reduce((x,y)=>x+y, 0, aList))
> 6
```

Reducing reduce

```
const aList = cons(1)(cons(2)(cons(3)(undefined)))

const reduce = f => i => l =>
   l ? reduce(f)(f(i,head(l)))(tail(l)) : i

console.log(reduce((x,y)=>x+y)(0)(aList))
> 6
```

Reducing reduce

```
const aList = cons(1)(cons(2)(cons(3)(undefined)))

const reduce = f => i => l =>
   l ? reduce(f)(f(i)(head(l)))(tail(l)) : i

console.log(reduce(x=> y=> x+y)(0)(aList))
> 6
```

Reducing reduce

```
const aList = cons(1)(cons(2)(cons(3)(undefined)))

const fold = f=> i=> l=> l ? fold(f)(f(i)(head(l)))(tail(l)) : i

console.log(fold(x=> y=> x+y)(0)(aList))
> 6
```

Reducing reduce

```
const aList = cons(1)(cons(2)(cons(3)(undefined)))

const fold = f=> i=> l=> l ? fold(f)(f(i)(head(l)))(tail(l)) : i,
      forEach = f=>l=>fold(_=>v=>f(v))(undefined)(l)

forEach(console.log)(aList)

> 1
> 2
> 3
```

K(f) ≡ K(v=> void)

≡ (x=> y=> x)(v=> void)

≡ y=> (v=> void)

≡ _=> v=> void

f := v=> void

K := x=> y=> x

Reducing reduce

```
const aList = cons(1)(cons(2)(cons(3)(undefined)))

const fold = f=> i=> l=> l ? fold(f)(f(i)(head(l)))(tail(l)) : i,
      forEach = f=>l=>fold(K(f))(undefined)(l)

forEach(console.log)(aList)

> 1
> 2
> 3
```

Reducing reduce

```
const aList = cons(1)(cons(2)(cons(3)(undefined)))

const fold = f=> i=> l=> l ? fold(f)(f(i)(head(l)))(tail(l)) : i,
      forEach = f=>l=>fold(K(f))(undefined)(l),
      reverse = l=> fold(c=>v=>cons(v)(c))(undefined)(l)

forEach(console.log)(reverse(aList))

> 3
> 2
> 1
```

Reducing reduce

```
const aList = cons(1)(cons(2)(cons(3)(undefined)))

const fold = f=> i=> l=> l ? fold(f)(f(i)(head(l)))(tail(l)) : i,
      forEach = f=>l=>fold(K(f))(undefined)(l),
      reverse = fold(c=>v=>cons(v)(c))(undefined)⇦ Tacit or Point-Free Style

forEach(console.log)(reverse(aList))

> 3
> 2
> 1
```

Reducing reduce

```
const aList = cons(1)(cons(2)(cons(3)(undefined)))

const fold = f=> i=> l=> l ? fold(f)(f(i)(head(l)))(tail(l)) : i,
      forEach = f=>l=>fold(K(f))(undefined)(l),
      flip = f=>a=>b=>f(b)(a),
      reverse = fold(c=>v=>cons(v)(c))(undefined)

forEach(console.log)(reverse(aList))

> 3
> 2
> 1
```

Reducing reduce

```
const aList = cons(1)(cons(2)(cons(3)(undefined)))

const fold = f=> i=> l=> l ? fold(f)(f(i)(head(l)))(tail(l)) : i,
      forEach = f=>l=>fold(K(f))(undefined)(l),
      flip = f=>a=>b=>f(b)(a),
      reverse = fold(flip(cons))(undefined)

forEach(console.log)(reverse(aList))

> 3
> 2
> 1
```

Exercise 2

To be announced...

Compose

```
const compose = (f, g) => x => f(g(x))

const marks = ['80.4','100.000','90','99.25',...],
      students = ['tim','sally','sam','cindy',...]

const parseMarks = compose(map(Number), fromArray),
      joined = zip(a=>b=>[a,b])(fromArray(students))(parseMarks(marks))

forEach(console.log)(joined)
```

```
Array(2) ["Valentino Dalton", 84.51]
Array(2) ["Hayden Walton", 42.85]
Array(2) ["Jane Bryant", 57.03]
Array(2) ["Ronald Hayes", 52.99]
Array(2) ["Journey Bradshaw", 65.39]
Array(2) ["Matias Guzman", 35.57]
Array(2) ["Jaylah Hunt", 11.88]
Array(2) ["Dangelo Russell", 61.11]
Array(2) ["Giovani Hendricks", 61.7]
...
```

Exercise 3

To be announced...

Lambda Calculus

$I = \lambda x . x$ lambda calculus expression

```
I = x => x
```

JavaScript

Lambda Calculus - application

$(\lambda x \,.\, x)\, y$ lambda calculus expression

```
(x => x)(y)
```

JavaScript

Lambda Calculus

$I = \lambda x . x$ I-Combinator

$K = \lambda x y . x$ K-Combinator

$$
\begin{aligned}
K\, I &= (\lambda x y . x)(\lambda x . x) \\
&= \lambda y . x\ [x := \lambda x . x] && \Leftarrow \text{Beta reduction} \\
&= \lambda y .\ (\lambda x . x) \\
&= \lambda yx . x && \Leftarrow \text{Equivalent due to currying} \\
&= \lambda xy . y && \Leftarrow \text{Alpha equivalence}
\end{aligned}
$$

Lambda's are always curried, i.e.:

$$\lambda x y . x = \lambda x . \lambda y . x$$

Variables are free or bound:

u bound

$$\lambda x y . x z = \lambda a b . a c = \lambda u . \lambda v . u w$$

↖ ↑ ↗ Alpha equivalence

⇧ *w* free

$$\lambda x . M x = M$$

⇧ Eta conversion

Lambda Calculus

Three operations:

- Alpha Equivalence
  - expressions are equivalent if their variables are renamed
- Beta Reduction
  - application of functions involves substituting the argument into the expression
- Eta Conversion
  - wrapping a simple lambda around an expression does not change the expression

Lambda expressions are anonymous *(although we’ve been making “macros” (e.g. I,K) with =)*

- They can’t refer to themselves! *(but there’s a trick for recursion: the Y-combinator)*

Conclusions

Lambda calculus is a ridiculously simple model of computation

And yet it is Turing Complete:

i.e. can compute everything that a Turing machine can compute,
just as powerful as any other programming language

Unlike Turing Machines, lambda calculus is the basis of real languages!

(from LISP to JavaScript to Haskell)