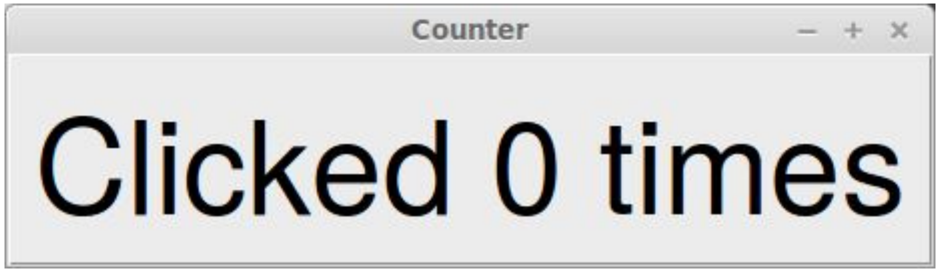
Logic, Categories, and Graphical User Interfaces

Jennifer Paykin, Steve Zdancewic,
Neel Krishnaswami

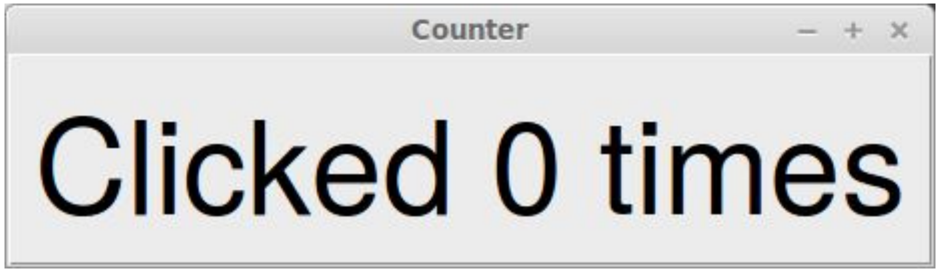
Wesleyan University
4/21/2015

GUIs



Clicked 0 times

GUIs

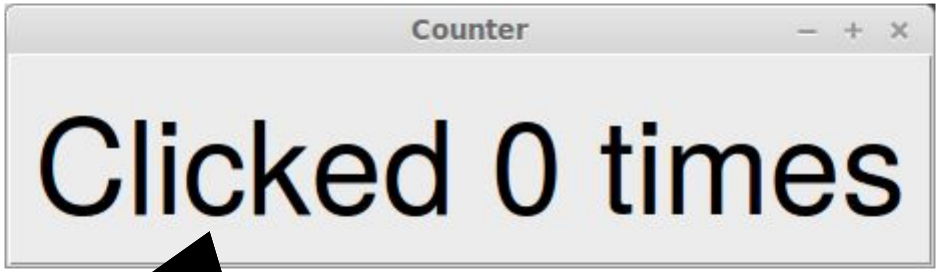


Clicked 0 times

Widgets

Callbacks

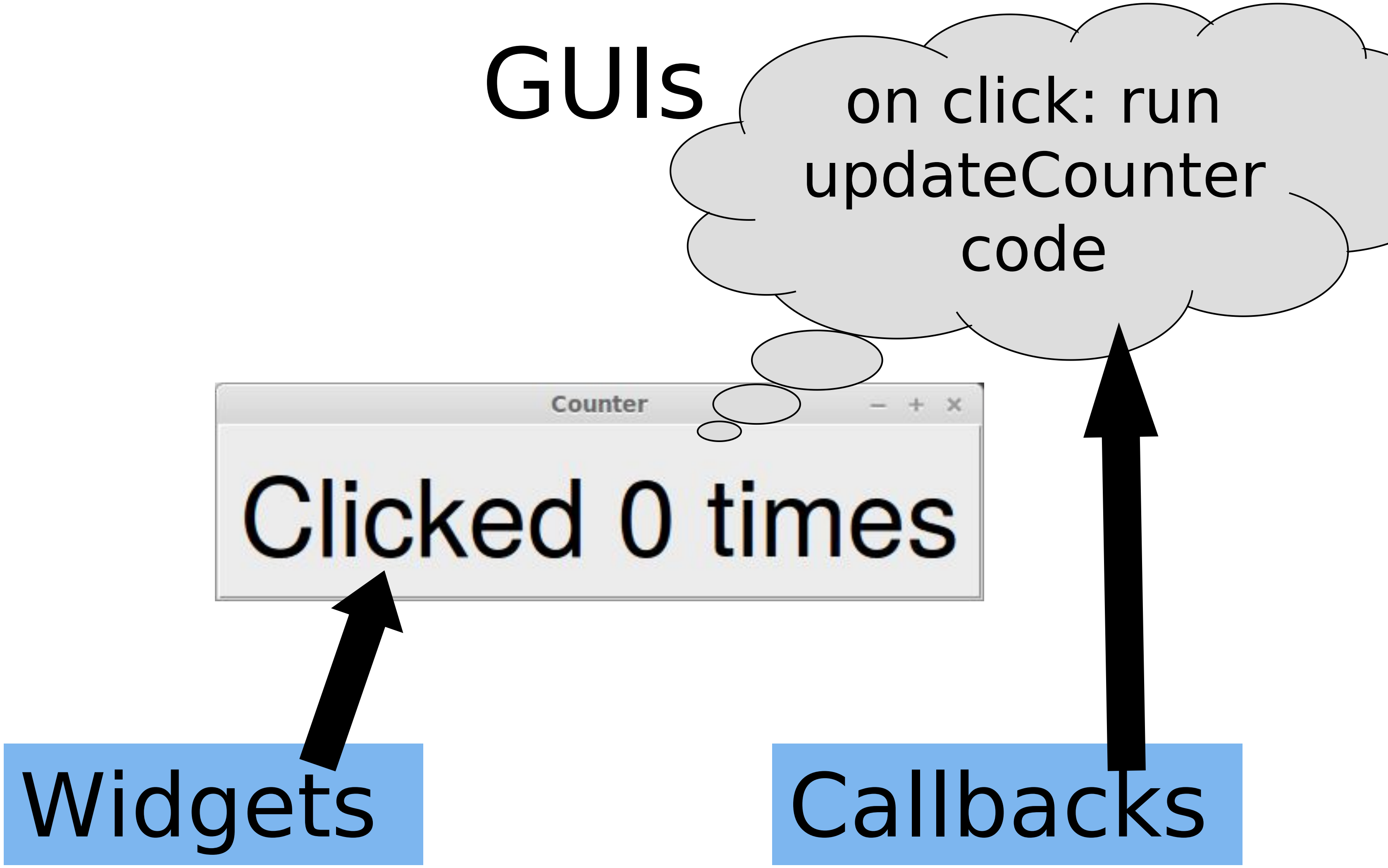
GUIs



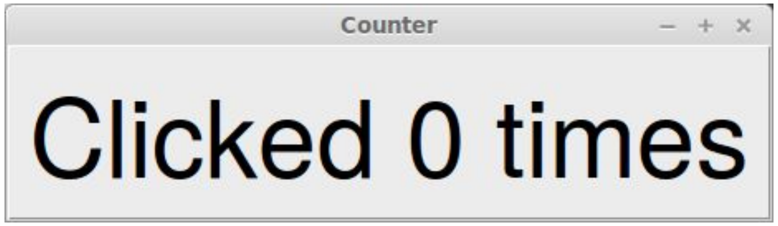
Widgets

Callbacks

GUIs


on click: run updateCounter code
Counter
Clicked 0 times
Widgets
Callbacks

A Simple GUI



Clicked 0 times

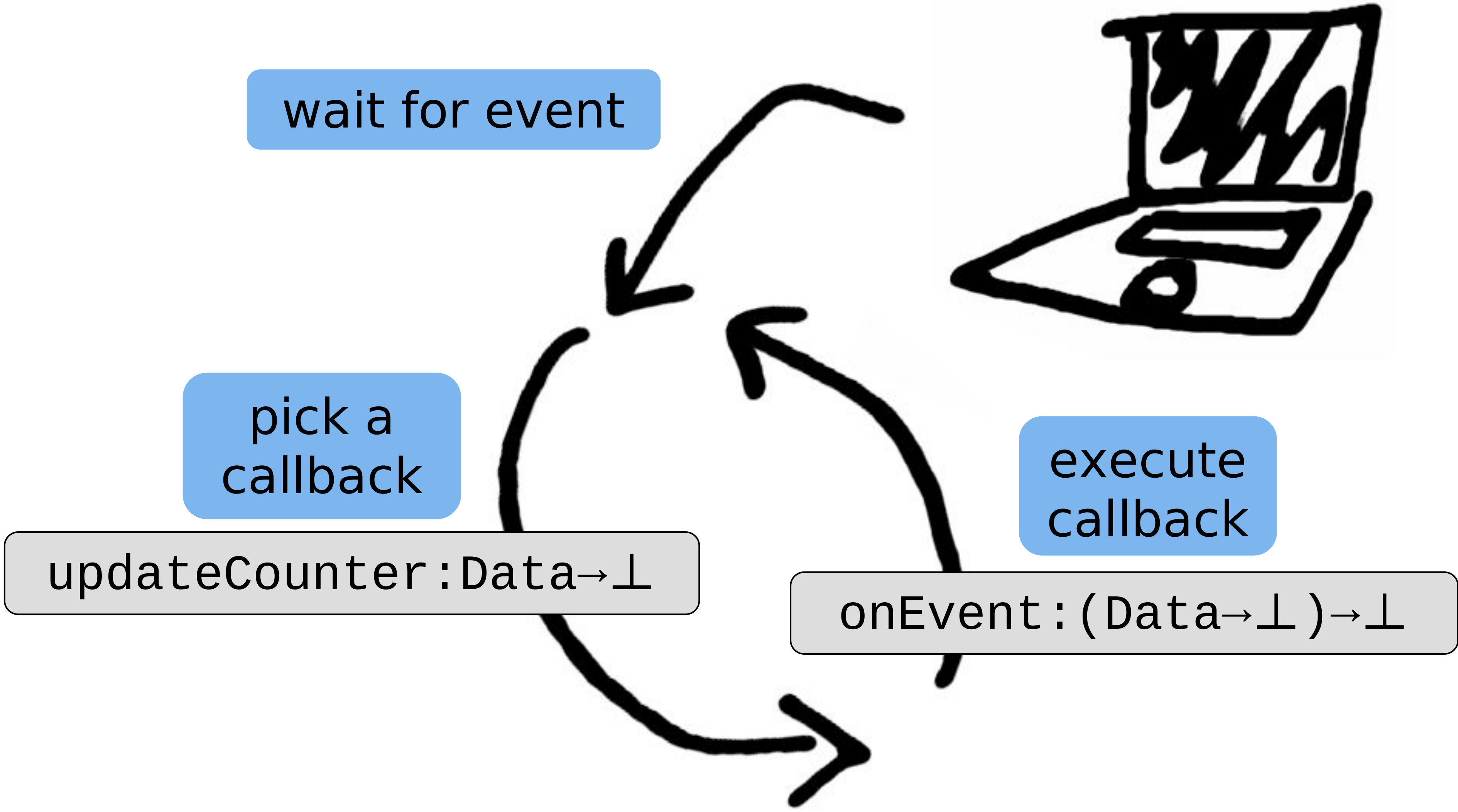
```
n    = 0
text = "Clicked " + str(n) + " times"

# button is a widget
button = Button(label  =text,
                command=updateCounter)

# updateCounter : Unit -> Void
def updateCounter():
  n.set(n.get()+1)
  text.set("Clicked" + str(n) + "times")

mainloop()
```

Event Loop



updateCounter:Data→⊥

onEvent:(Data→⊥)→⊥

Non-local Code

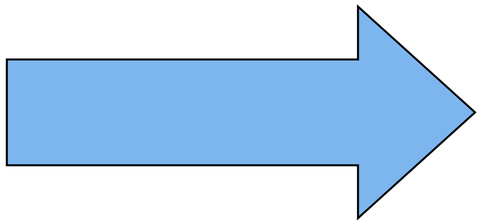
Three sections of code:

1. Define widgets
2. Define callbacks
3. Define event loop

Non-local Code

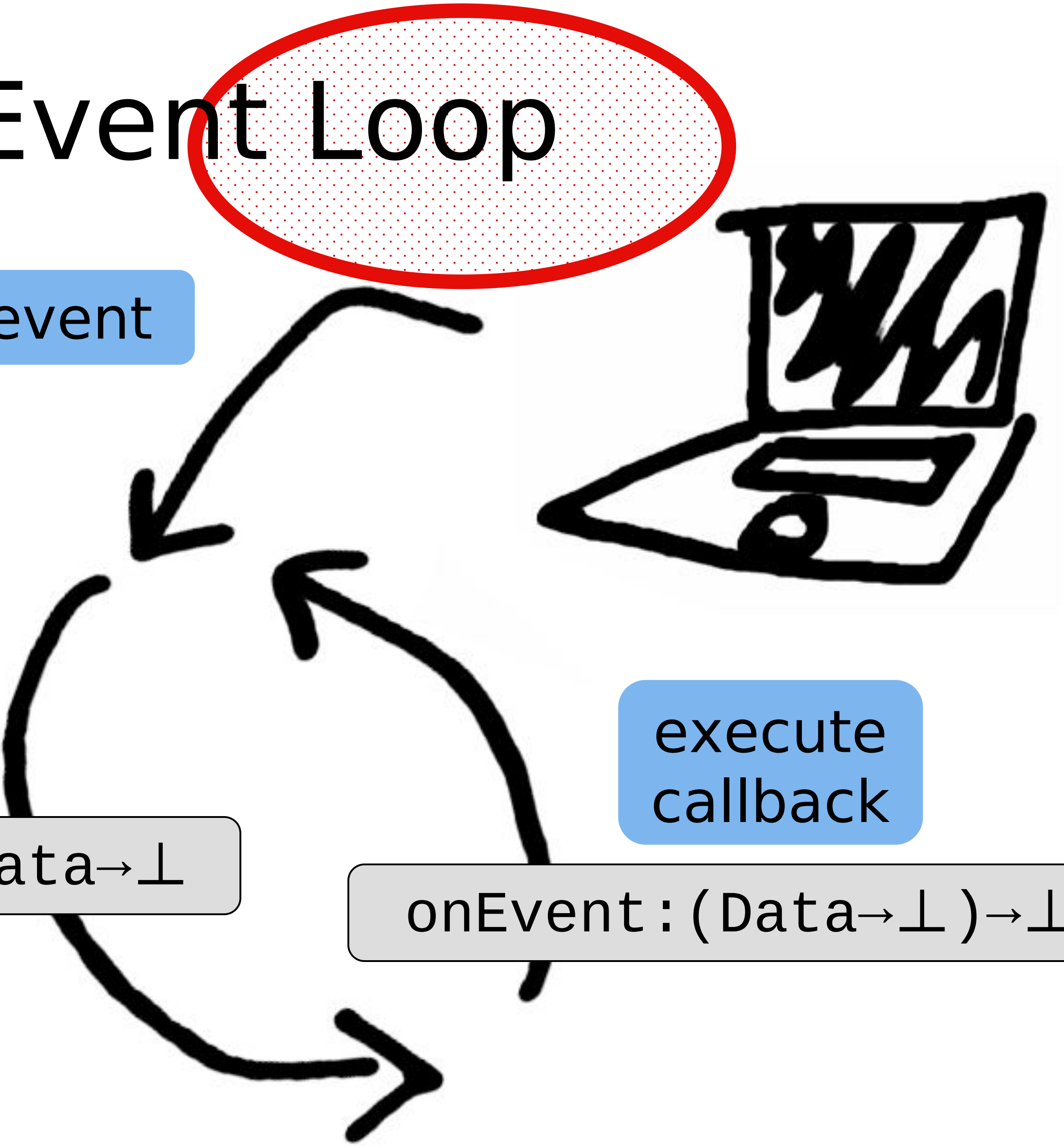
Three sections of code:

1. Define widgets
2. Define callbacks
3. Define event loop



Surface language with one section

Event Loop



wait for event

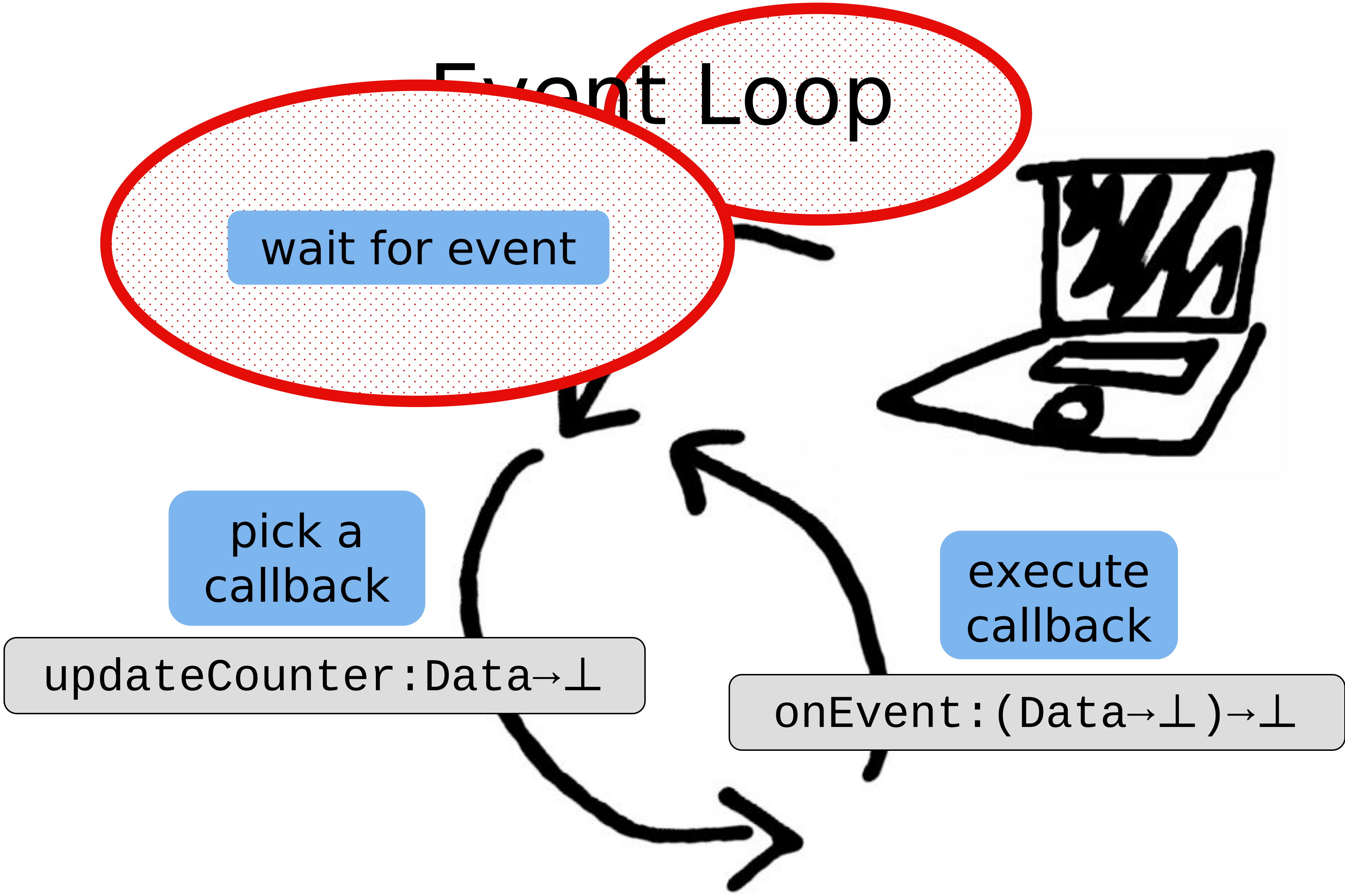
pick a callback

`updateCounter:Data→⊥`

execute callback

`onEvent:(Data→⊥)→⊥`

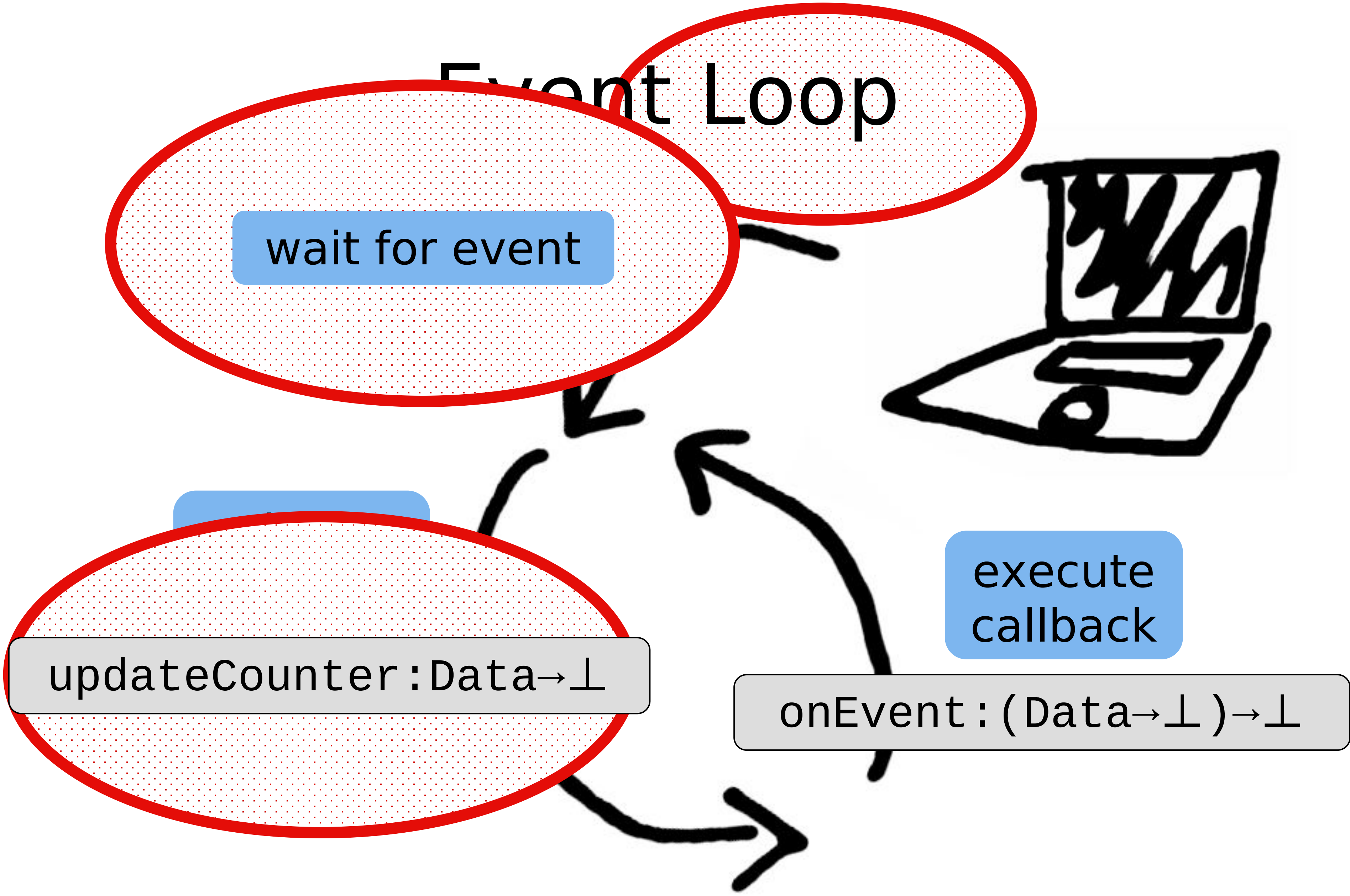
Event Loop



```
updateCounter:Data→⊥
```

```
onEvent:(Data→⊥)→⊥
```

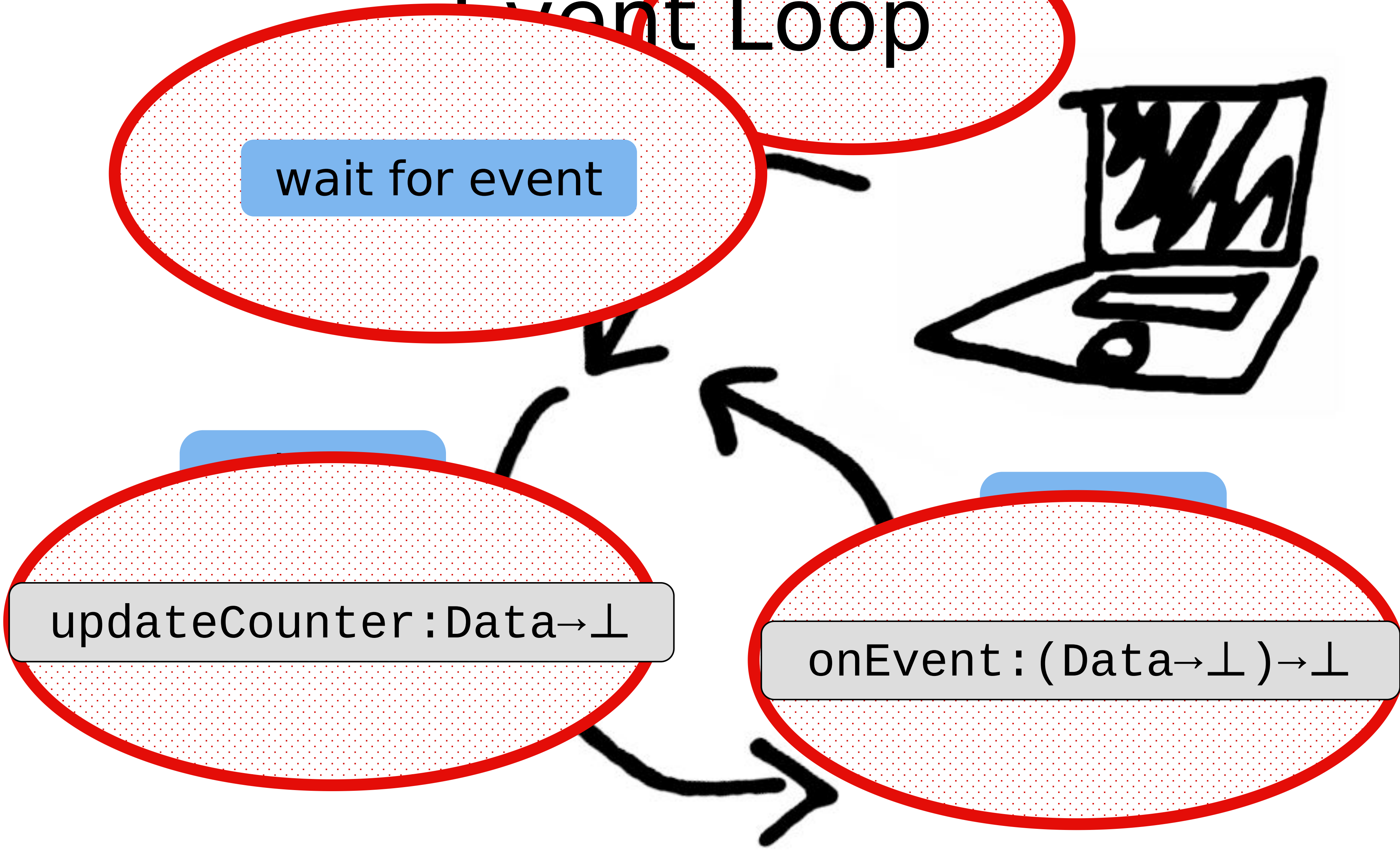
Event Loop



```
updateCounter:Data→⊥
```

```
onEvent:(Data→⊥)→⊥
```

Event Loop



Localized Code

```
let counter =
  let  n  = 0 in
  let  w  = newWidget() in
  wait () = onClick(w) in
  wait () = drawButton(w,n+1) in
  ()
```

Localized Code

```
letrec count (w:Widget) (n:Nat) =
  wait () = onClick(w) in
  wait () = drawButton(w,n+1) in
  count w (n+1)
```

```
let counter () =
  let n = 0 in
  let w = newWidget() in
  count w n
```

Localized Code

```
letrec count (w:Widget) (n:Nat) =
   ...
```

```
let counter () =
   ...
```

```
let 2counters =
  let w1 = counter() in
  let w2 = counter() in
  wait (_,_) = sync w1 w2 in
  ()
```

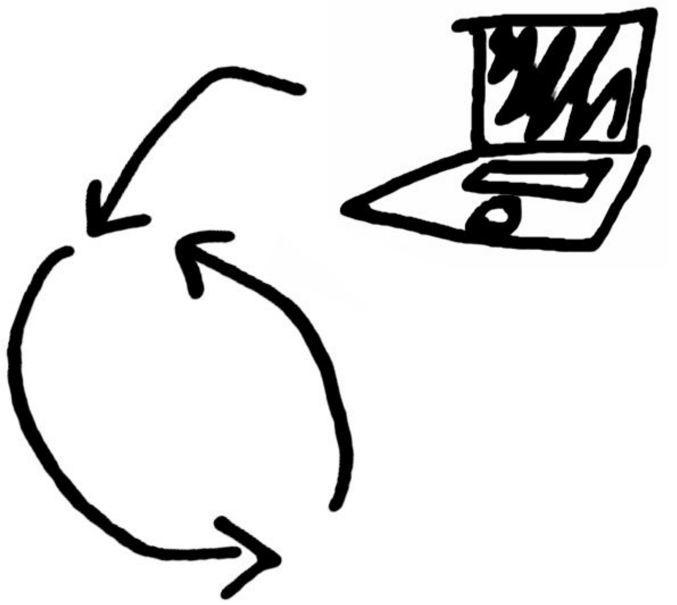
Goal

```
letrec count (w:Widget) (n:Nat) =
  wait () = onClick(w) in
  wait () = drawButton(w,n+1) in
  count w (n+1)
```

wait for event

pick a callback

execute callback



Goal

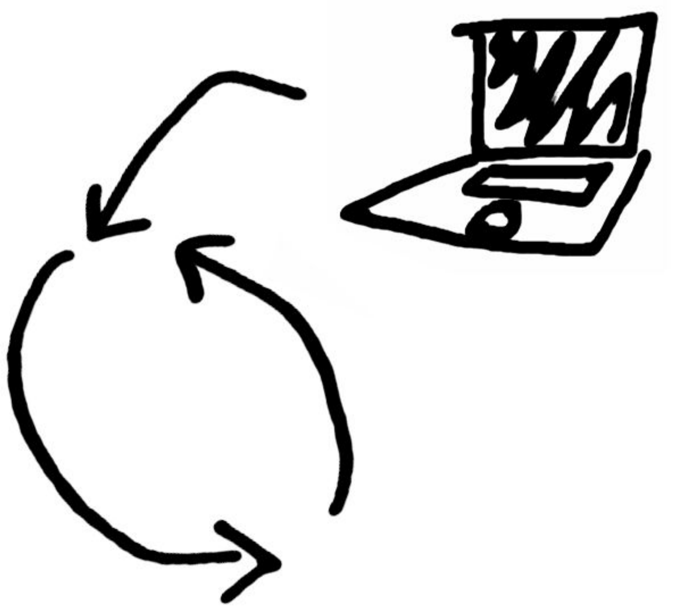
```
letrec count (w:Widget) (n:Nat) =
  wait () = onClick(w) in
  wait () = drawButton(w,n+1) in
  count w (n+1)
```

wait for event

`wait`

pick a callback

execute callback



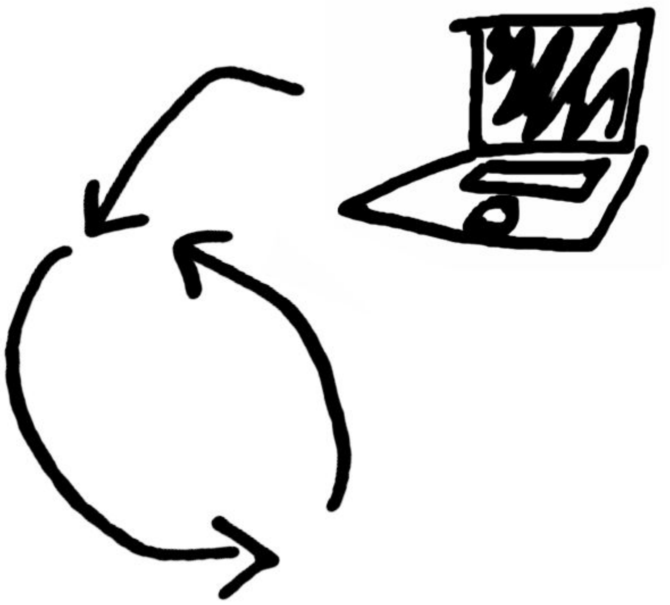
Goal

```
letrec count (w:Widget) (n:Nat) =
  wait () = onClick(w) in
k wait () = drawButton(w,n+1) in
  count w (n+1)
```

wait for event `wait`

pick a callback `λ().k:Data→⊥`

execute callback



Goal

```
letrec count (w:Widget) (n:Nat) =
  wait () = onClick(w) in
k wait () = drawButton(w,n+1) in
  count w (n+1)
```

wait for event

`wait`

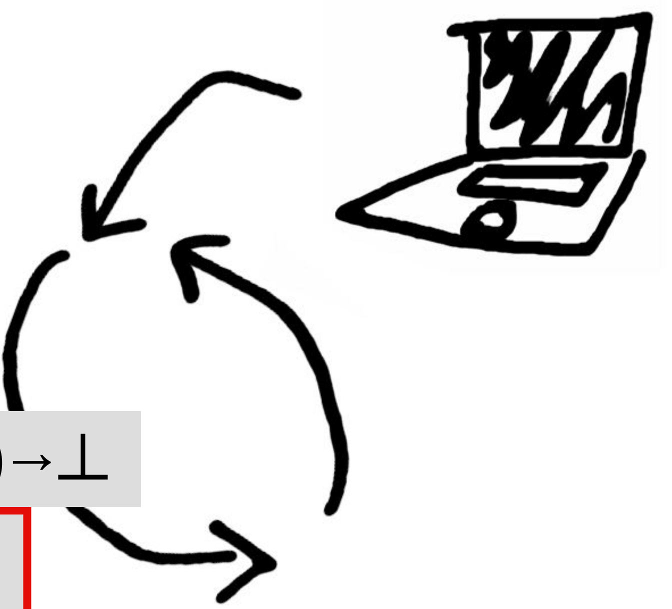
pick a callback

`λ().k:Data→⊥`

execute callback

`onEvent:(Data→⊥)→⊥`

`onEvent(λ().k)`



What have we done?

1. What is the langauge for?
2. What features do we need?
3. Define a language
4. Describe how it executes (semantics)

How do we do it?

Types/
Logic

1. What is the langauge for?
2. What features do we need?
3. Define a language
4. Describe how it executes (semantics)

How do we do it?

1. What is the langauge for?
2. What features do we need?
3. Define a language
4. Describe how it executes (semantics)

Types/ Logic

Combine Features

Curry-Howard Isomorphism

Type System	Type	Term
Logic	Proposition	Proof

Types vs Propositions

$$A \wedge B \Rightarrow B \wedge A$$

Proofs and Propositions

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge B \quad A \wedge B}}{A \wedge B \quad B} \qquad \dfrac{\dfrac{}{A \wedge B \quad A \wedge B}}{A \wedge B \quad A}}{A \wedge B \quad B \wedge A}}{A \wedge B \Rightarrow B \wedge A}$$

Types vs Propositions

$$A \land B \Rightarrow B \land A$$

$$\lambda x.(\pi_2 x, \pi_1 x) : A \times B \rightarrow B \times A$$

Terms and Types

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x:A\times B \vdash x:A\times B}}{x:A\times B \vdash \pi_2 x : B} \qquad \dfrac{\dfrac{}{x:A\times B \vdash x:A\times B}}{x:A\times B \vdash \pi_1 x : A}}{x:A\times B \vdash (\pi_2 x,\pi_1 x):B\times A}}{\vdash \lambda x.(\pi_2 x,\pi_1 x) : A\times B \to B\times A}$$

Curry-Howard Isomorphism

Logic $\simeq$ Type System

Properties of Logic

$\simeq$

Properties of Programming Languages

Logic	Feature
intuitionistic	pure functional
classical	callbacks
temporal	computations
linear	resource consciousness

Logic	Feature
intuitionistic	pure functional
classical	callbacks
temporal	computations
linear	resource consciousness

Classical Logic & Negation

$$\neg\neg A \simeq A$$

$$A \rightarrow B \simeq \neg A \vee B$$

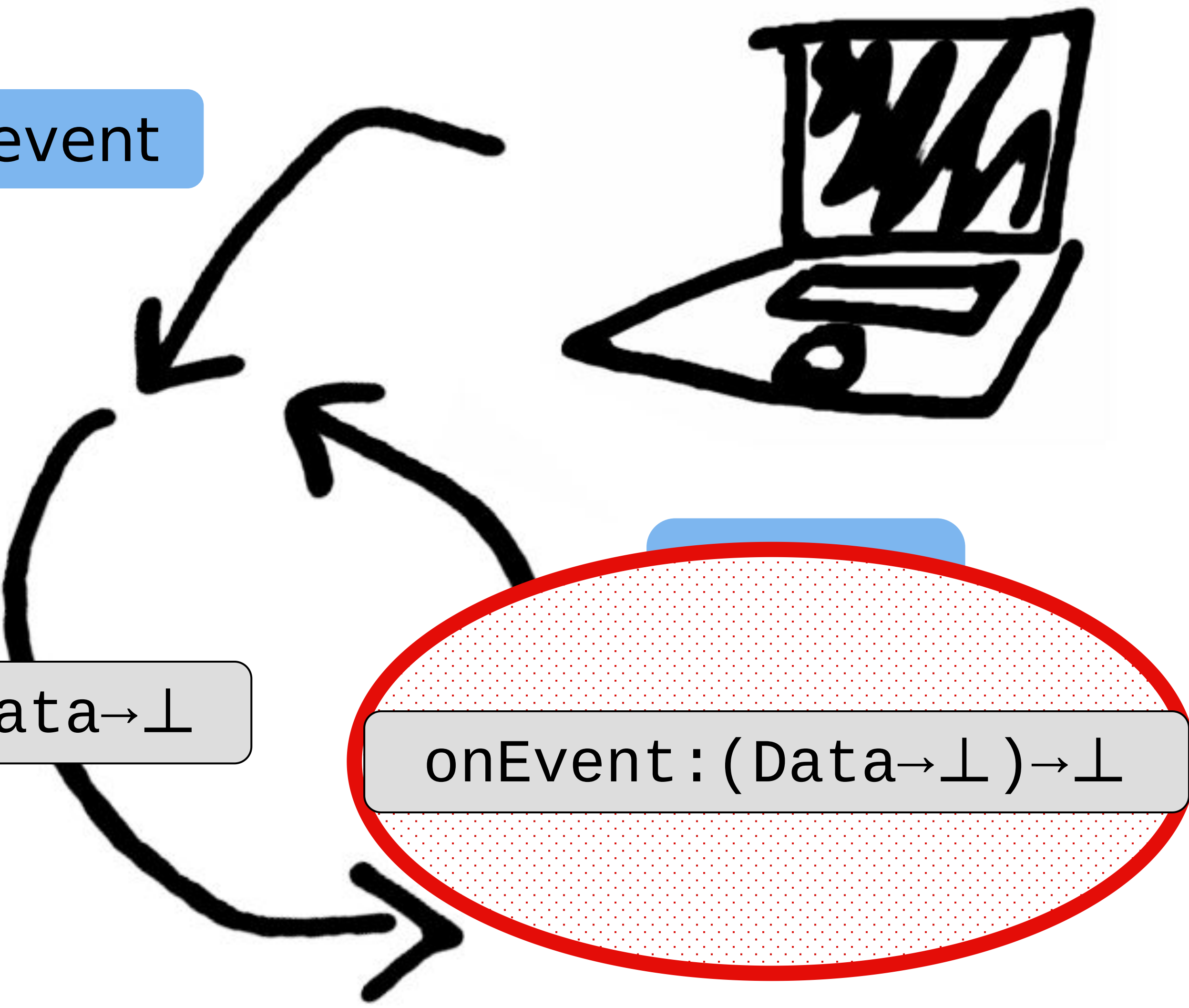
Classical Logic & Negation

$$\neg\neg A \simeq A$$

$$A \to B \simeq \neg A \vee B$$

$$A \to \bot \simeq \neg A \vee \bot \simeq \neg A$$

Event Loop



wait for event

pick a callback

`updateCounter:Data→⊥`

`onEvent:(Data→⊥)→⊥`

Double Negation

`onEvent:(Data→⊥)→⊥`

$A \to \bot \simeq \neg A$

`onEvent:¬¬Data`

$\neg\neg A \simeq A$

`onEvent:Data`

Double Negation Syntax

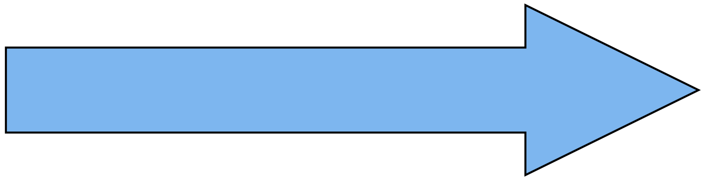
onEvent : Data

```
let x : Data = onEvent in t
```

Double Negation Syntax

```
onEvent:Event
```

```
let x : Data = onEvent in t
```



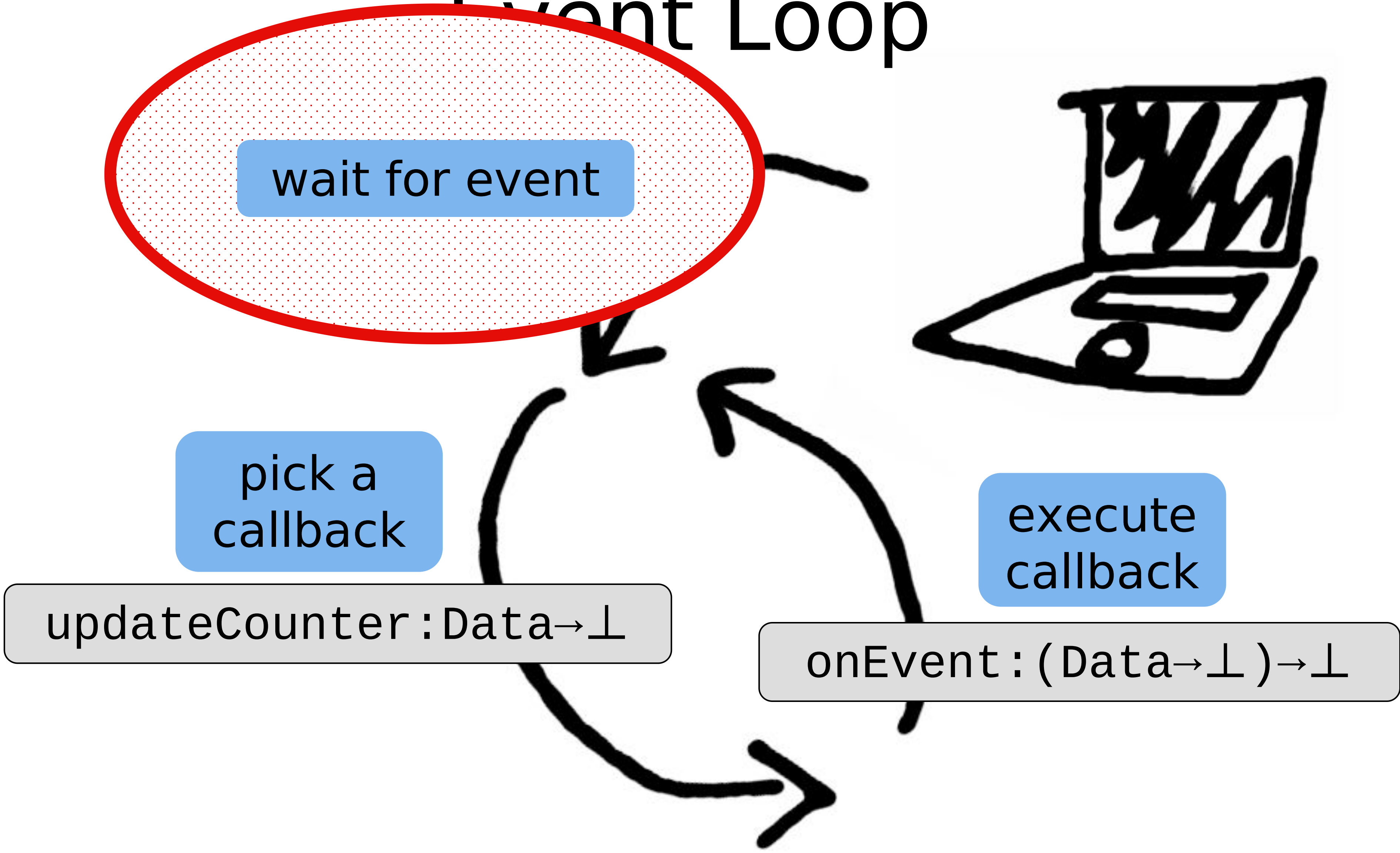
```
onEvent:(Data→⊥)→⊥
```

```
onEvent (λx:Data.t)
```

Wait?

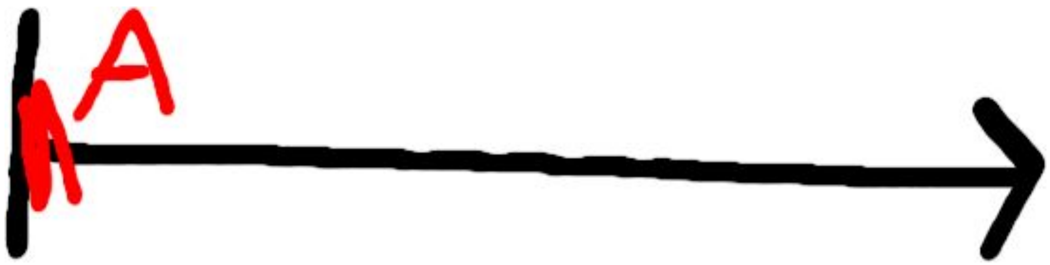
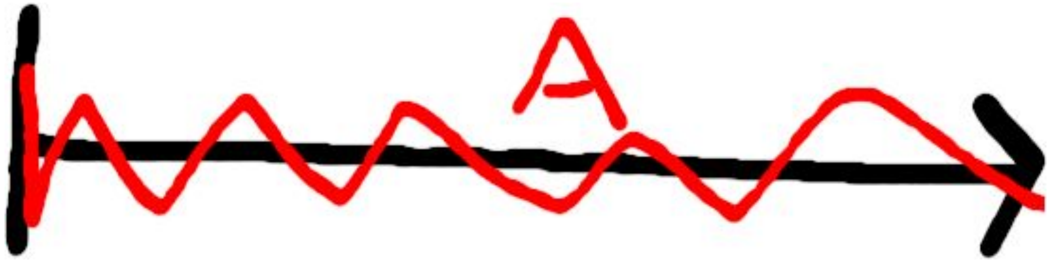
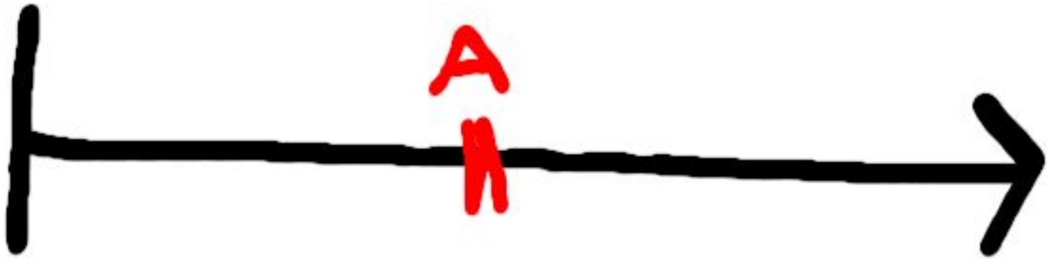
```
wait x : Data = onEvent in t
```

Event Loop


wait for event
pick a callback
updateCounter:Data→⊥
execute callback
onEvent:(Data→⊥)→⊥

Logic	Feature
intuitionistic	pure functional
classical	callbacks
temporal	computations
linear	resource consciousness

Temporal Logic

Now	A	
Always	□ A	
Eventually	◊ A	

Eventually as Computation

Eventually as Computation

$$\frac{\Gamma \quad \mathtt{t1} :\lozenge A \qquad \Gamma, x{:}A \quad \mathtt{t2} :\lozenge B}{\Gamma \quad \mathtt{wait\ x\ =\ t1\ in\ t2} :\lozenge B}$$

Double Negation + Time

```
onEvent:(Data→⊥)→⊥
```

```
onEvent:□(Data→⊥)→⊥
```

Double Negation + Time

```
onEvent:□(Data→⊥)→⊥
```

$$A \to \bot \simeq \neg A$$

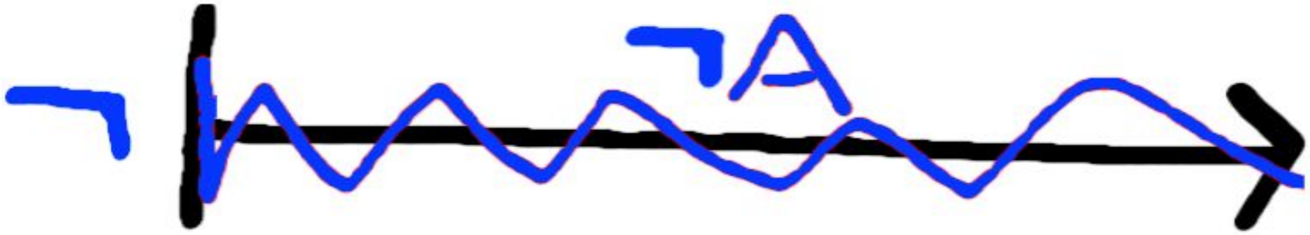
```
onEvent:¬□¬Data
```

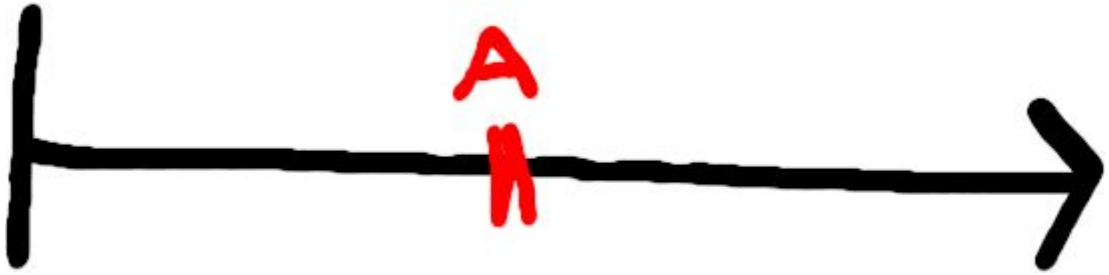
Classical Temporal Logic

$\neg\Box\neg A$

$\simeq$

$\Diamond A$





Double Negation + Time

```
onEvent:□(Data→⊥)→⊥
```

$A \to \bot \simeq \neg A$

```
onEvent:¬□¬Data
```

$\neg \Box \neg A \simeq \Diamond A$

```
onEvent:◊Data
```

Event Loop Syntax

```
wait x:Data = onEvent in t
```

wait for event

```
onEvent:◊Data
```

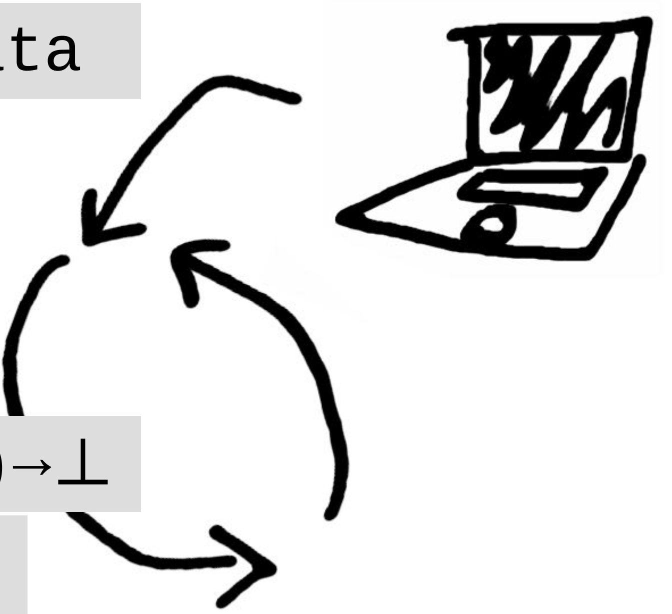
pick a callback

```
λx.t:Data→⊥
```

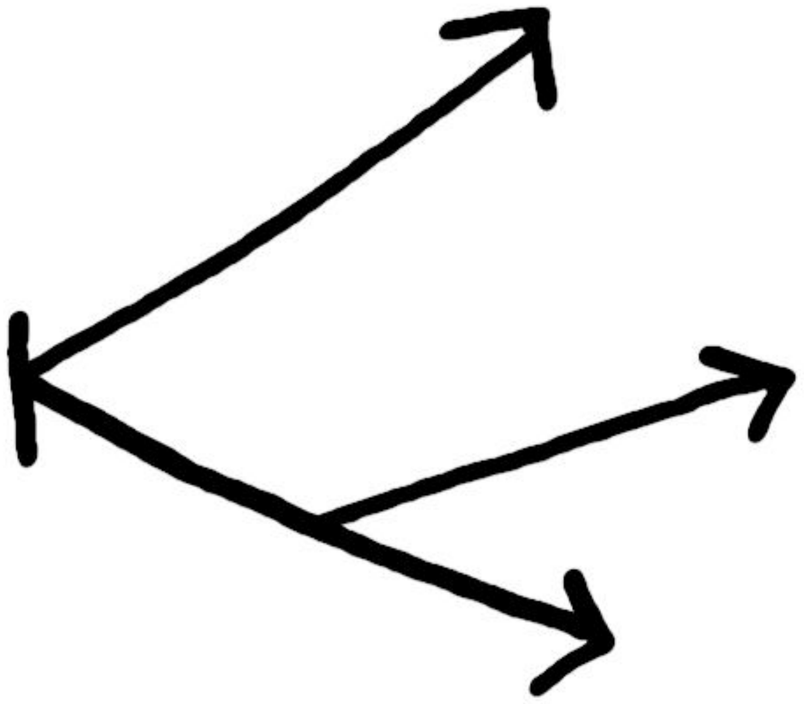
execute callback

```
onEvent:(Data→⊥)→⊥
```

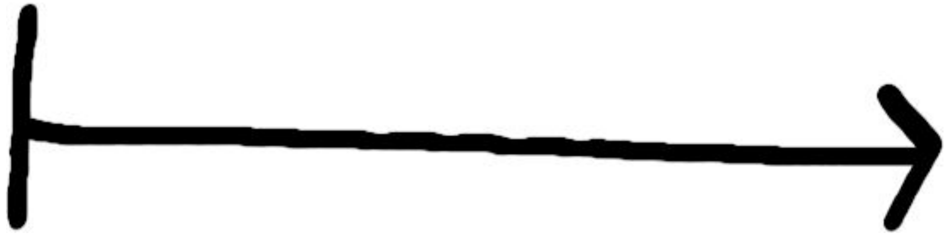
```
onEvent(λx.t)
```



Linear (Time) Temporal Logic

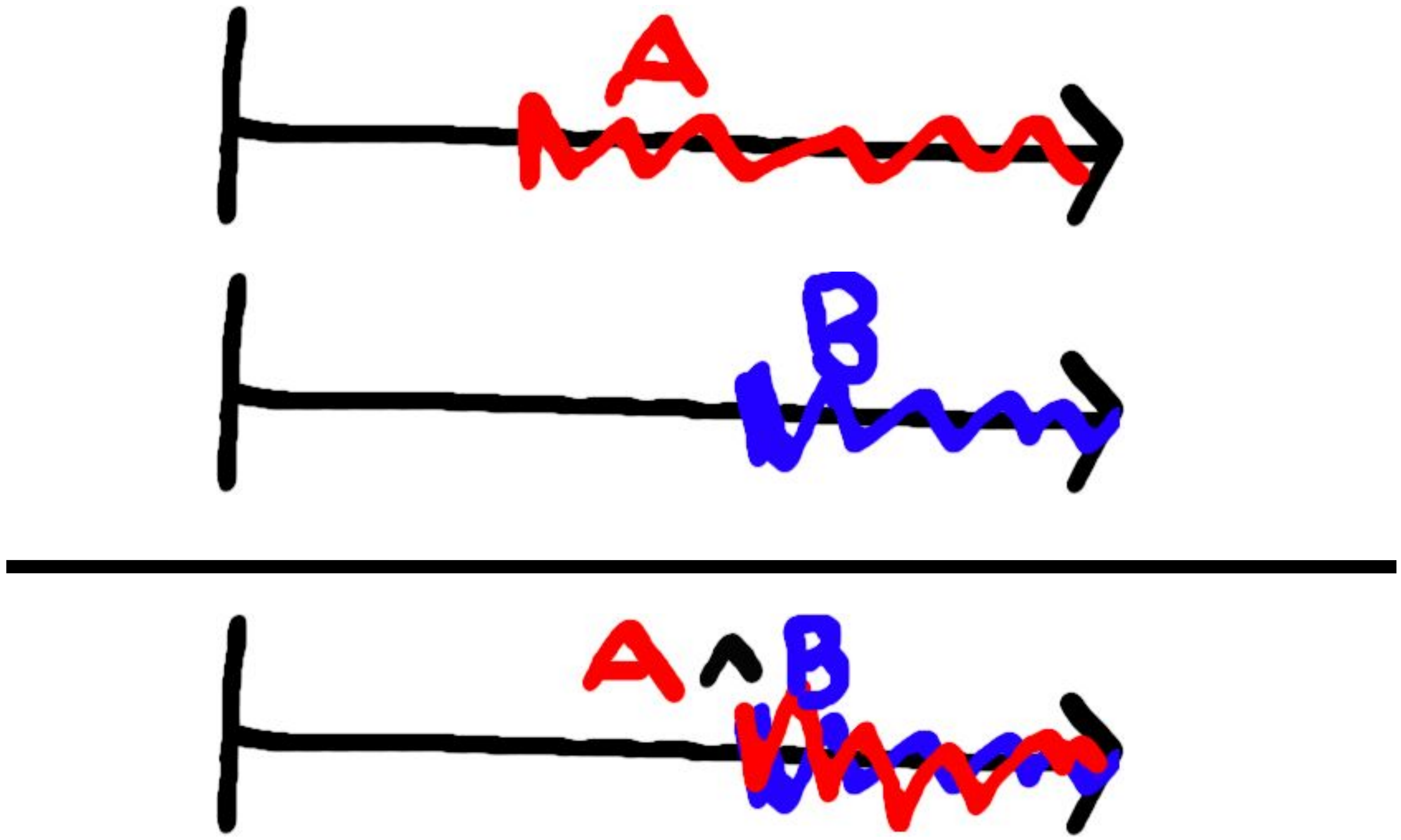


Branching Time



Linear Time

$\Diamond\Box A \to \Diamond\Box B \to \Diamond\Box(A \land B)$



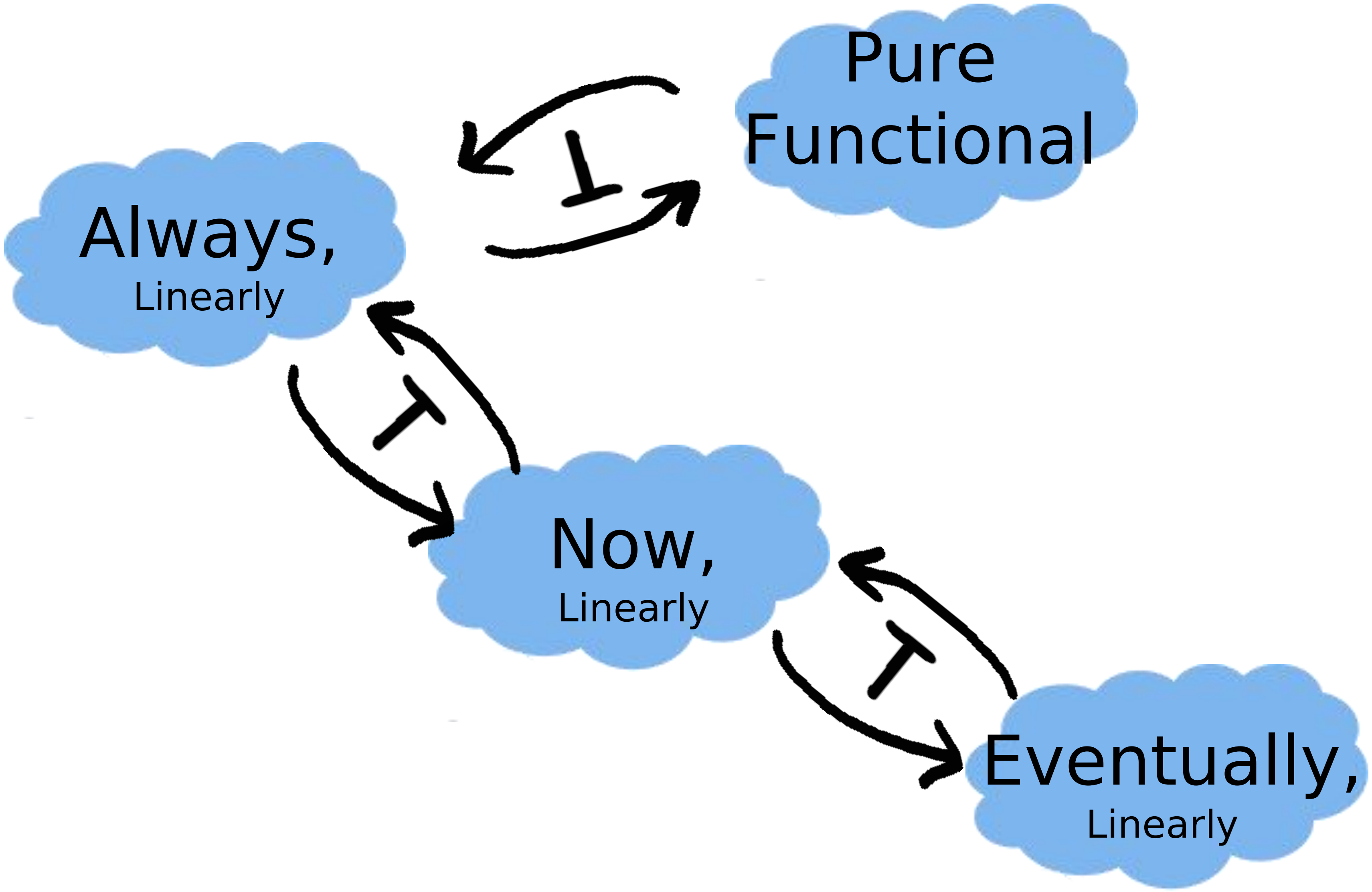
Synchronize

$$\frac{\Gamma \qquad t1\ :\lozenge\Box A \qquad\qquad \Gamma \qquad t2\ :\lozenge\Box B}{\Gamma \qquad \mathtt{sync}\ t1\ t2\ :\lozenge\Box(A\times B)}$$

```
let 2counters =
  let w1 = counter() in
  let w2 = counter() in
  wait (_,_) = sync w1 w2 in
  ()
```

Logic	Feature
intuitionistic	pure functional
classical	callbacks
temporal	computations
linear	resource consciousness

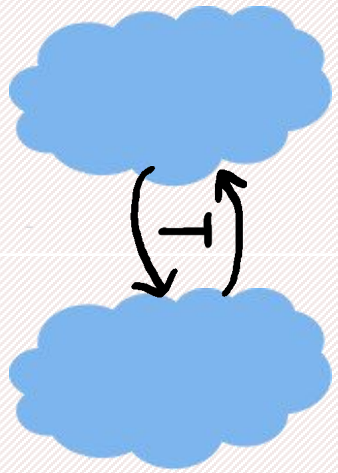
Features as worlds

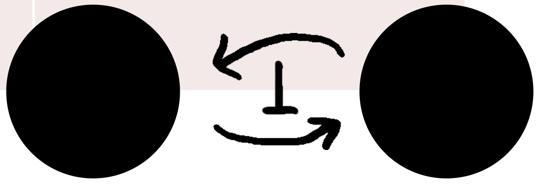

Pure Functional
Always, Linearly
Now, Linearly
Eventually, Linearly

Curry-Howard Isomorphism

Type System	Type	Term
Logic	Proposition	Proof
Category	Object	Morphism

"Adjoint functors arise everywhere..."

Type System	Type	Term	
Logic	Proposition	Proof	
Category	Object	Morphism	Adjunction



Current & Future Work

- GUI language
  - localized syntax
  - event loop semantics
- "features as worlds"
  - framework for relationships between worlds