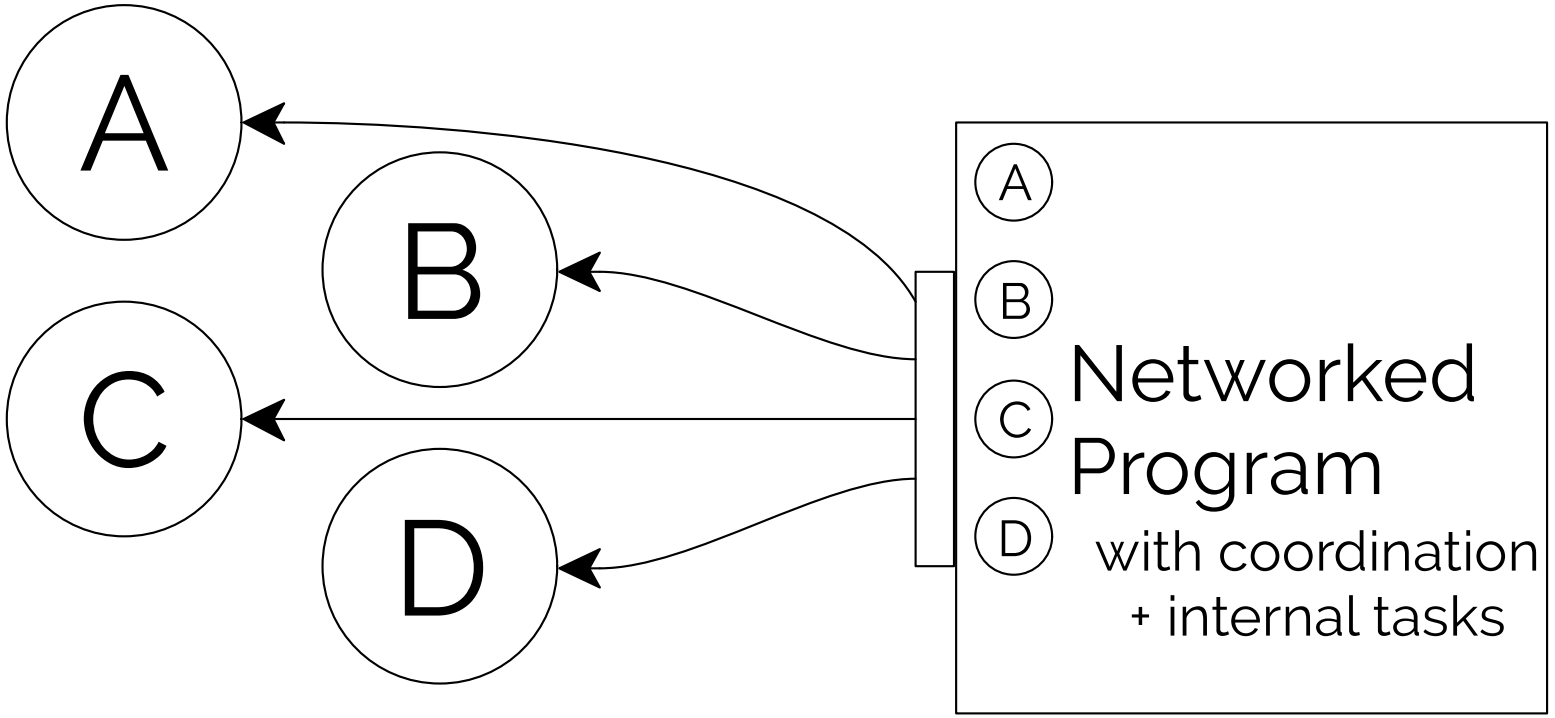
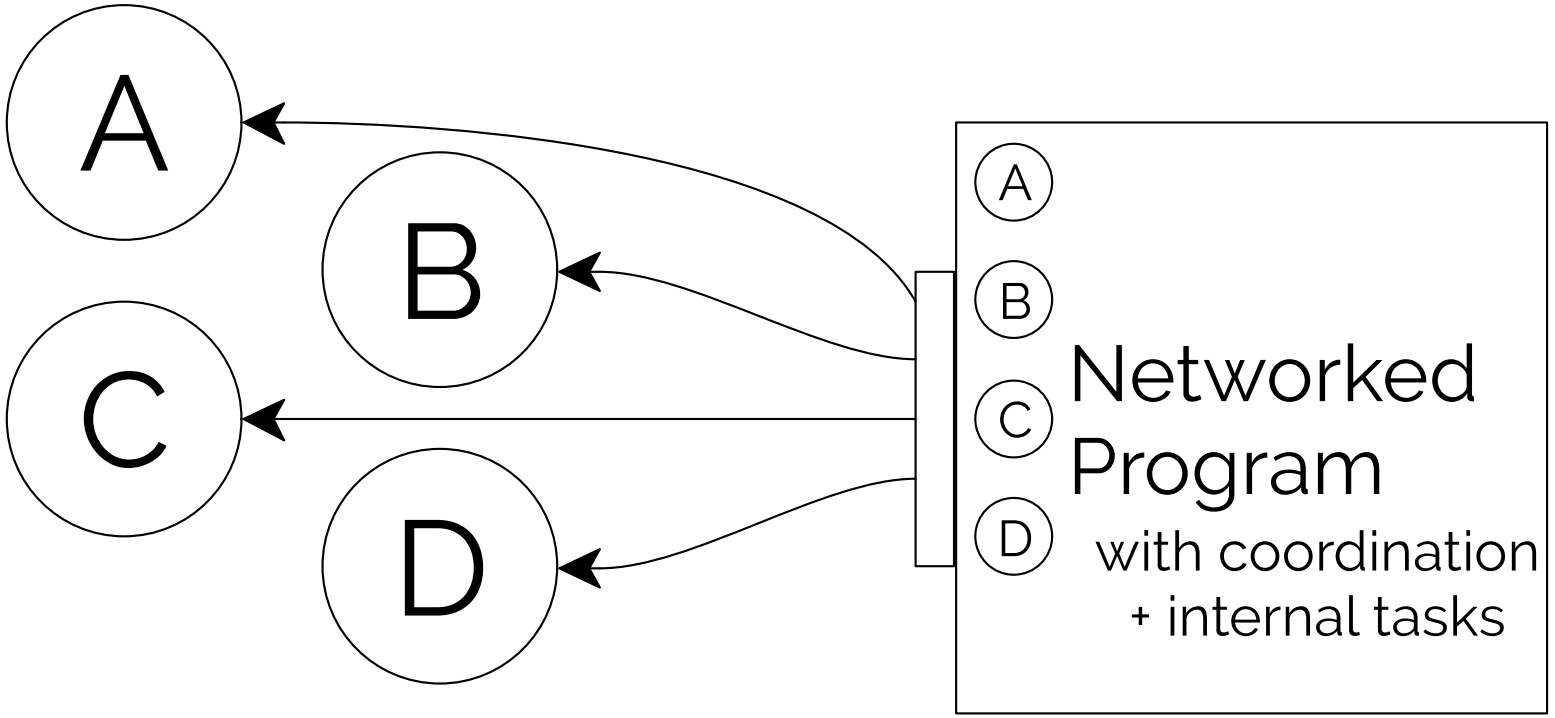
From Events to Reactions: A Progress Report

Tony Garnock-Jones
tonyg@ccs.neu.edu
Northeastern University

Joint work with Matthias Felleisen and Sam Caldwell

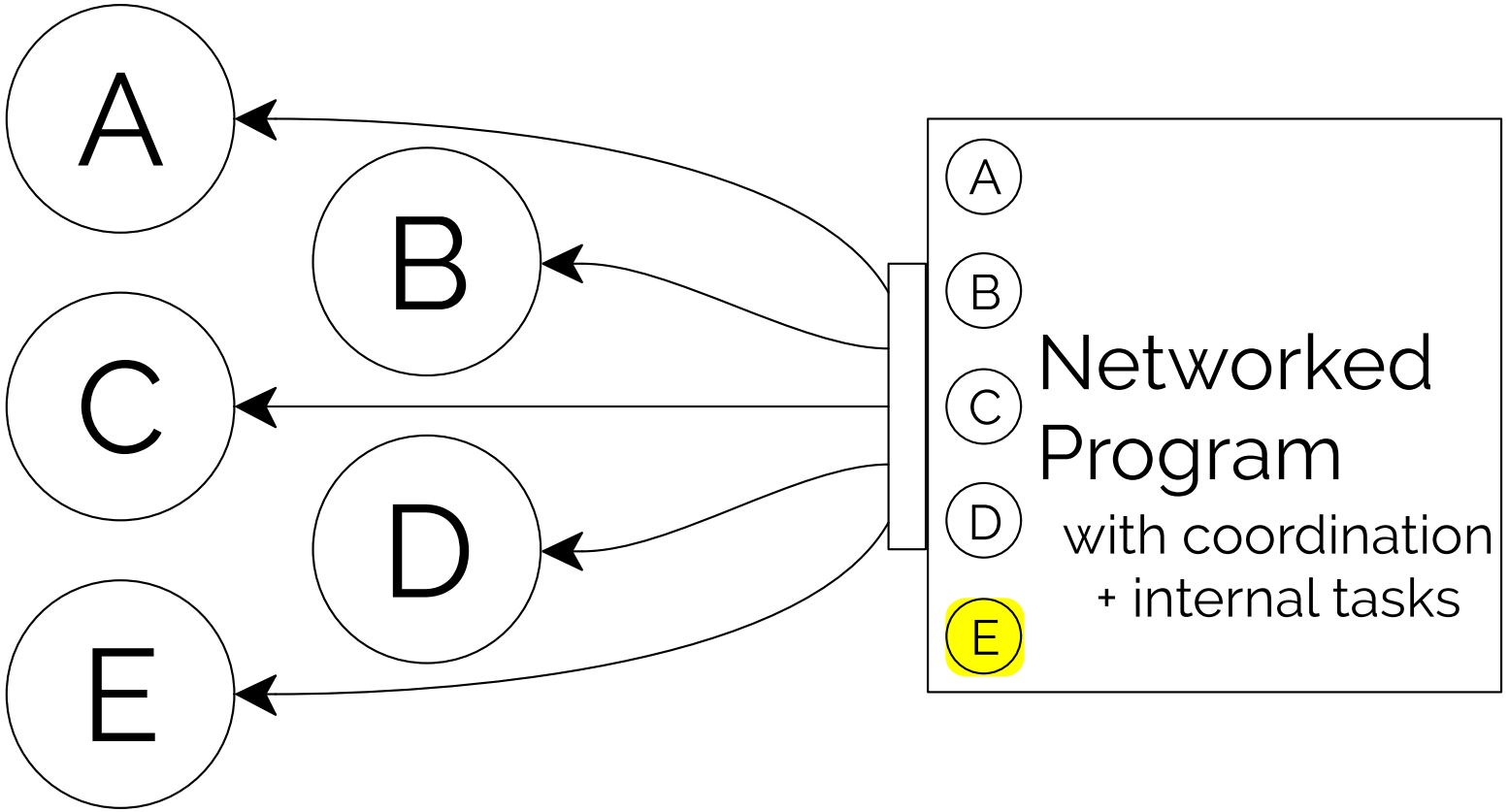


Interactivity ⇒ External Concurrency



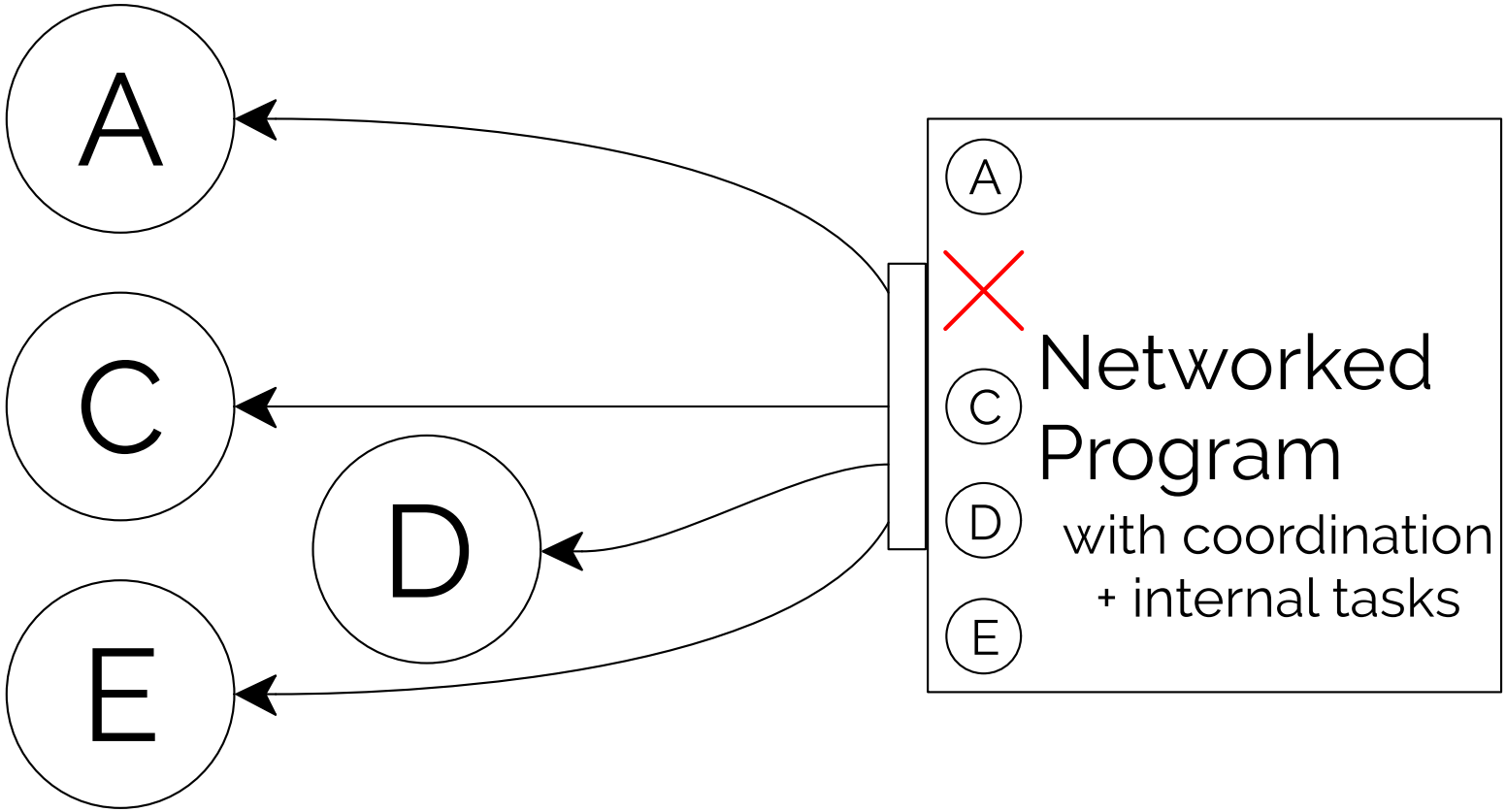
Interactivity ⇒ External Concurrency

Internal Organisation Reflects External Concurrency



Interactivity ⇒ External Concurrency

Component startup → interaction → shutdown/failure

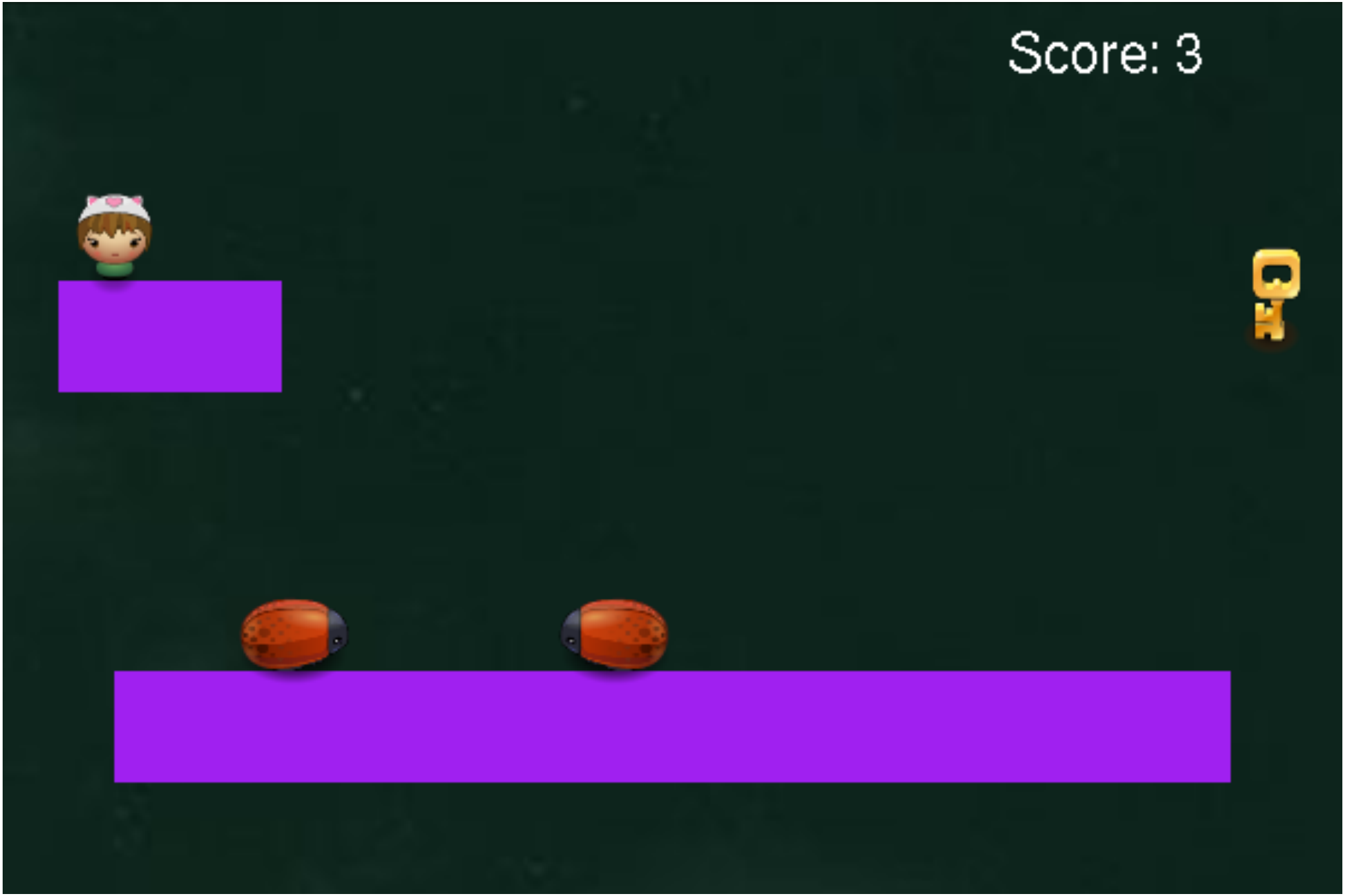


Interactivity ⇒ External Concurrency

Component startup → interaction → shutdown/failure

Syndicate DSL by example

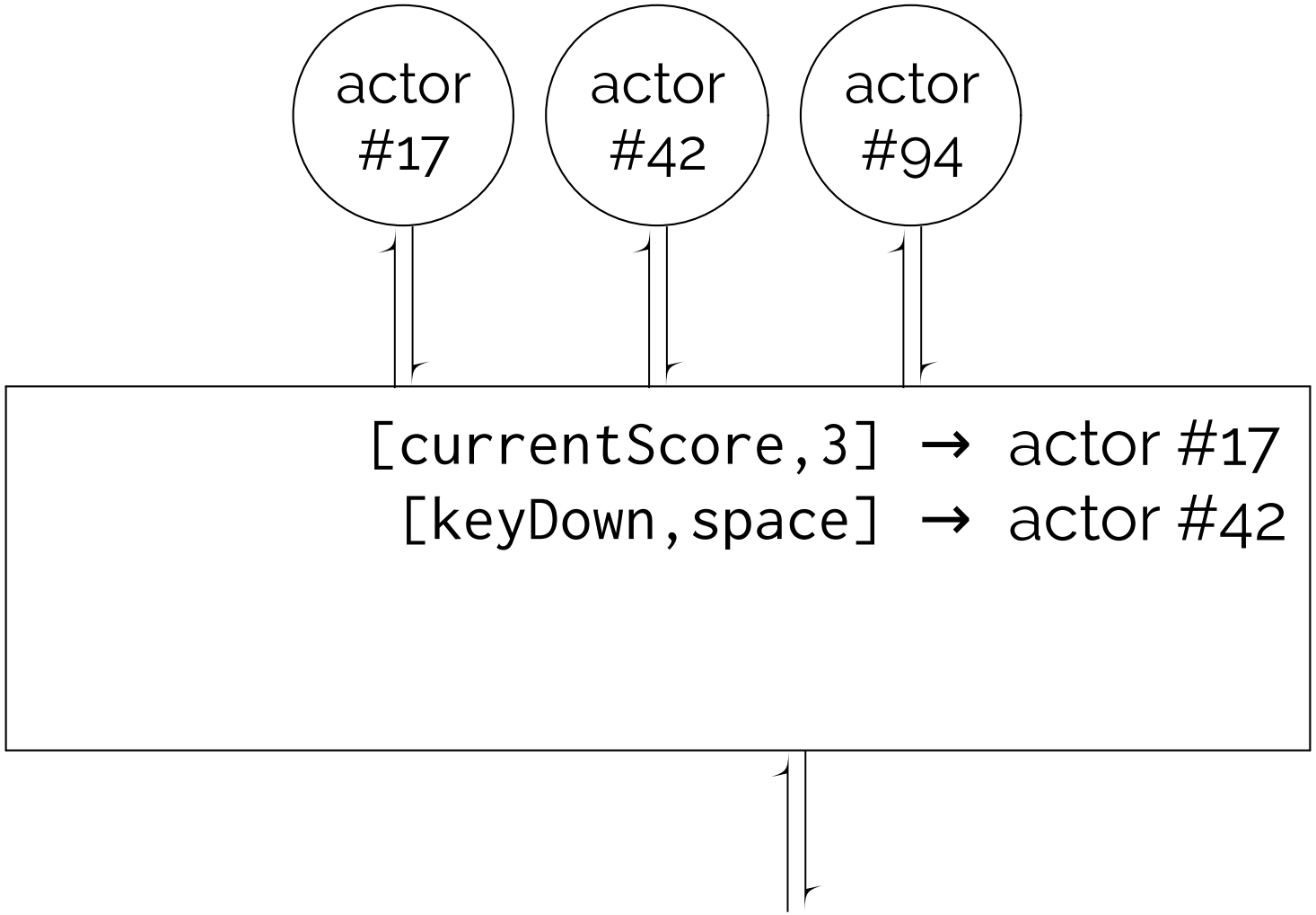
- Mapping events to components
- Managing conversational state
- Monitoring changes in shared state


Score: 3

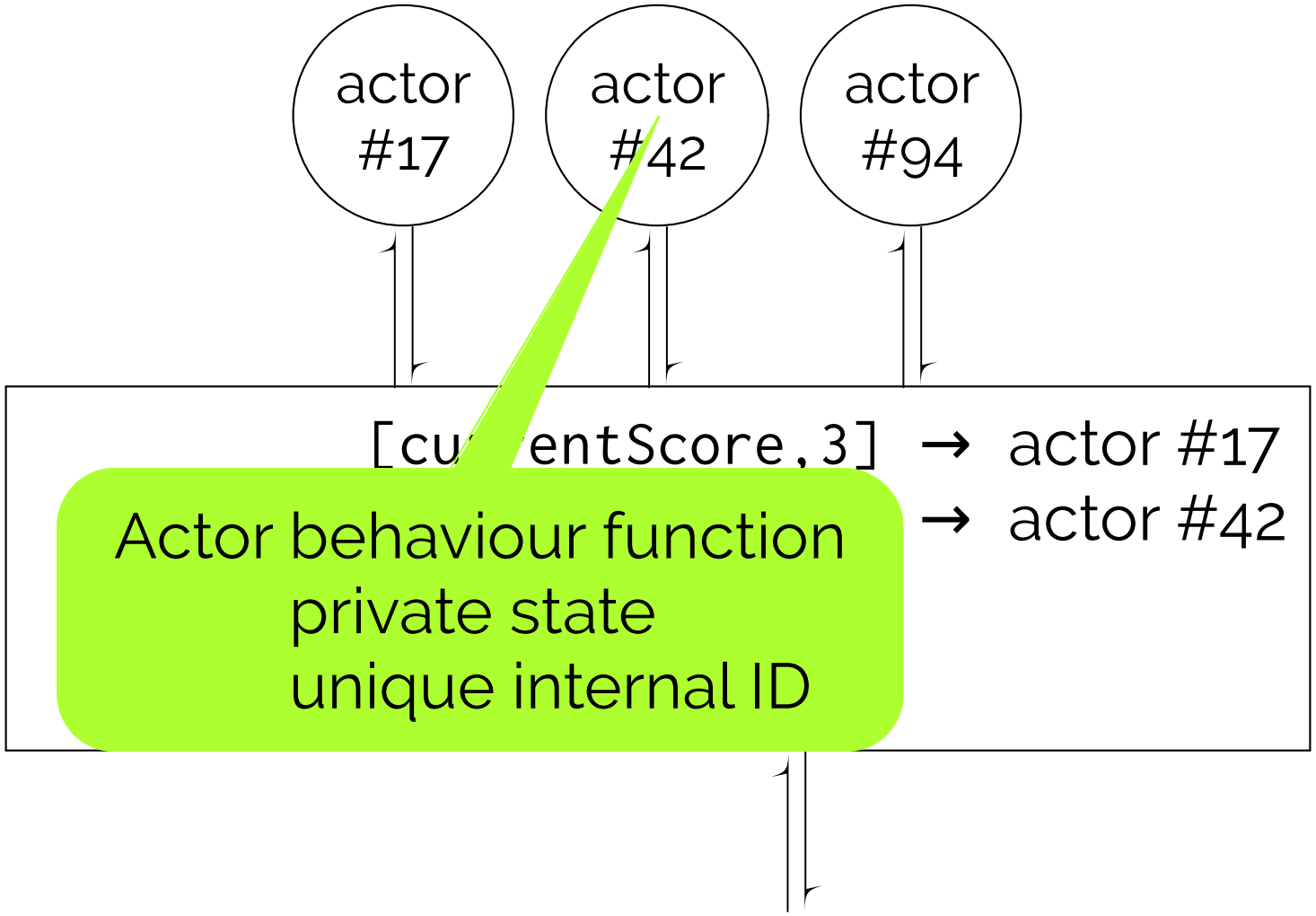
Syndicate

$$\text{event} \times \text{state} \to [\text{action}] \times \text{state}$$

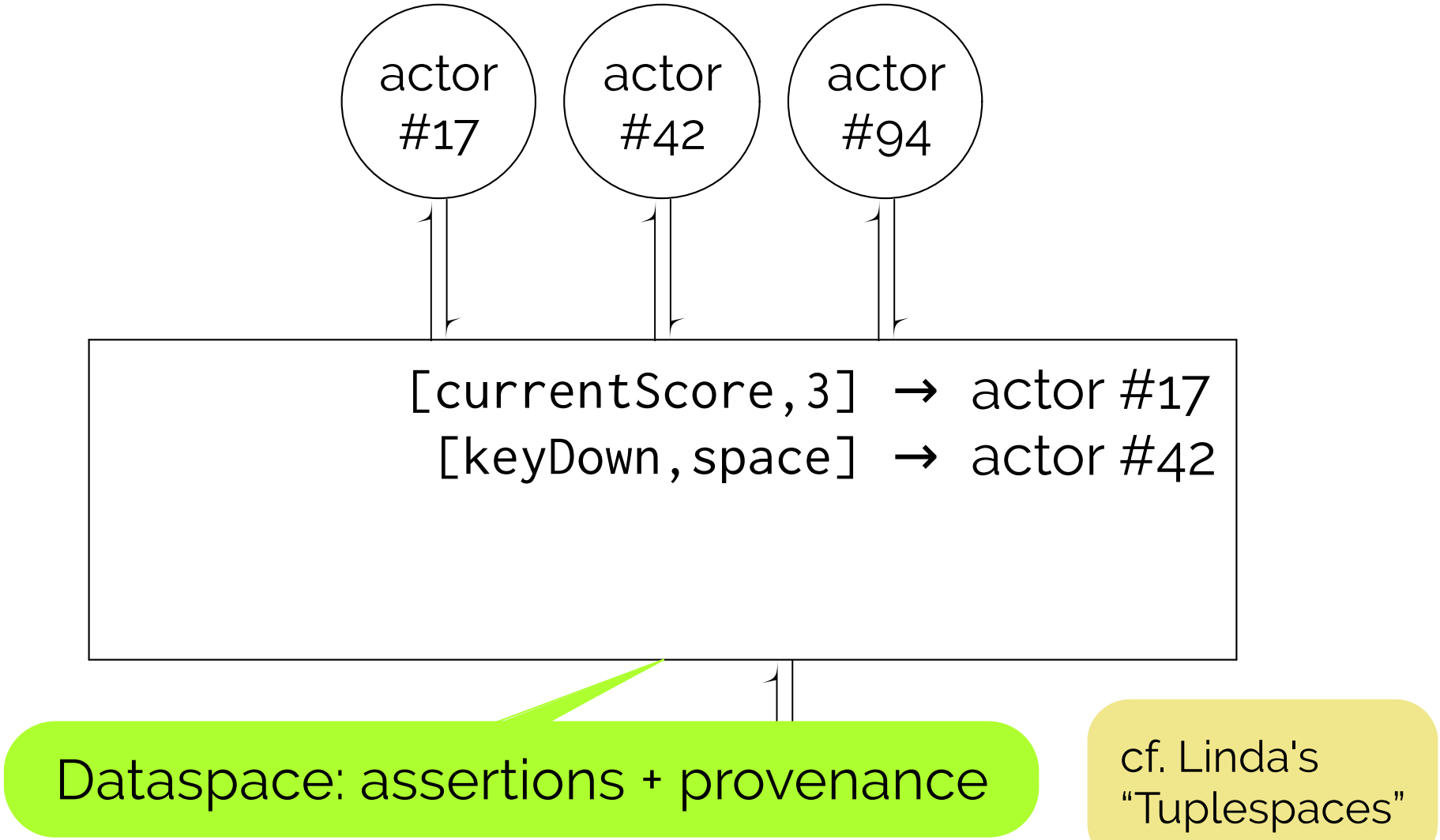
event × state → [action] × state



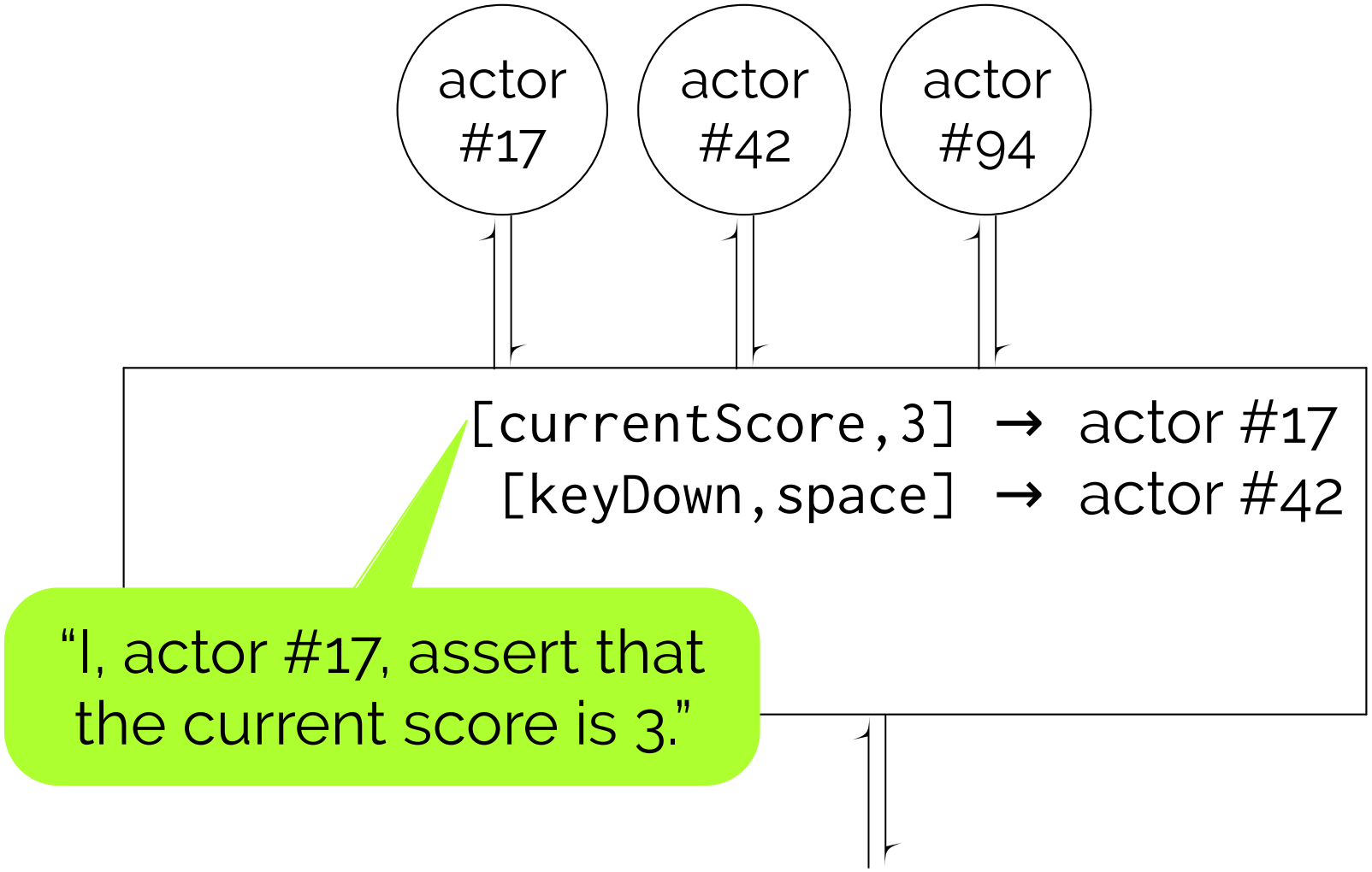
```
event × state → [action] × state
```



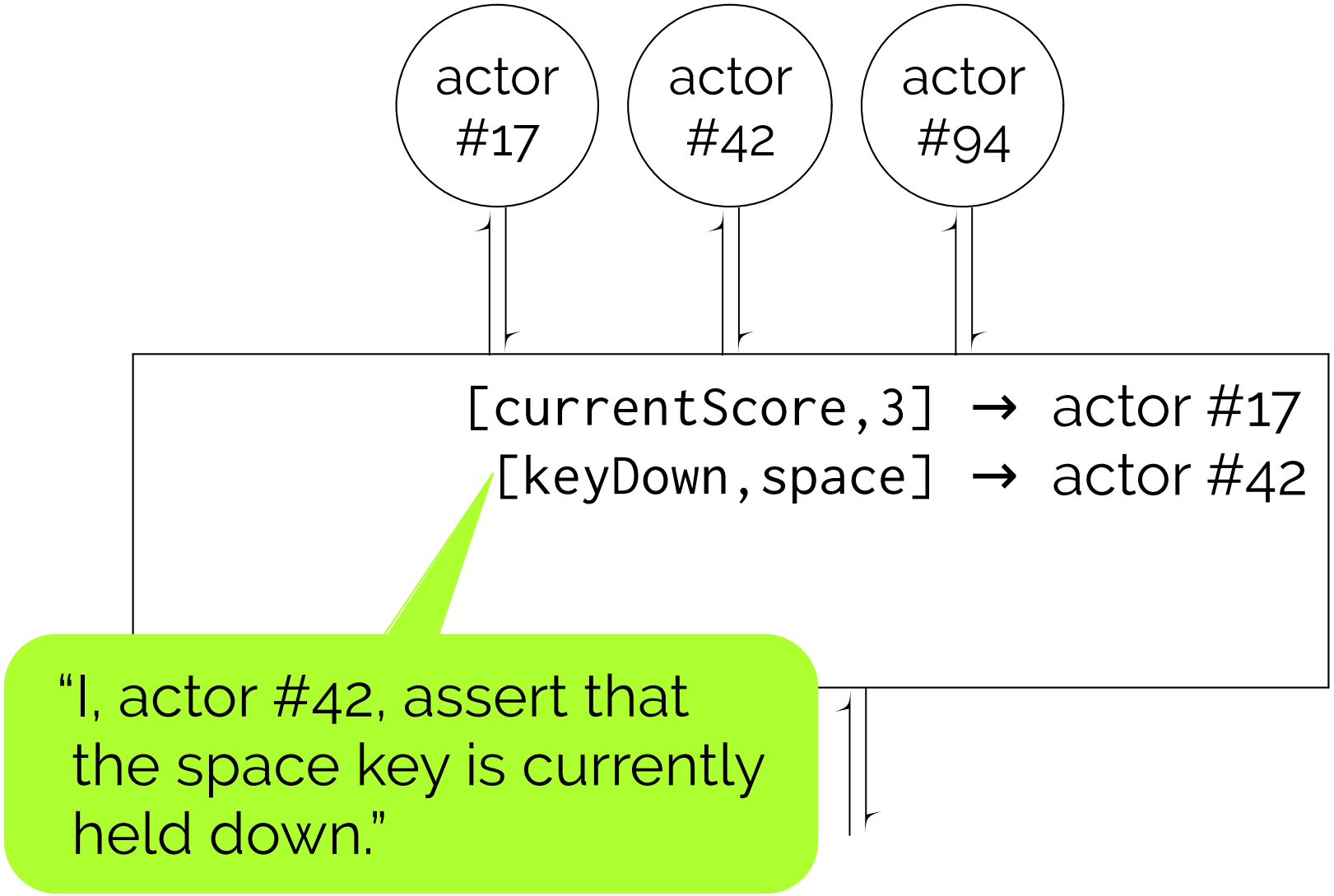
```
event × state → [action] × state
```



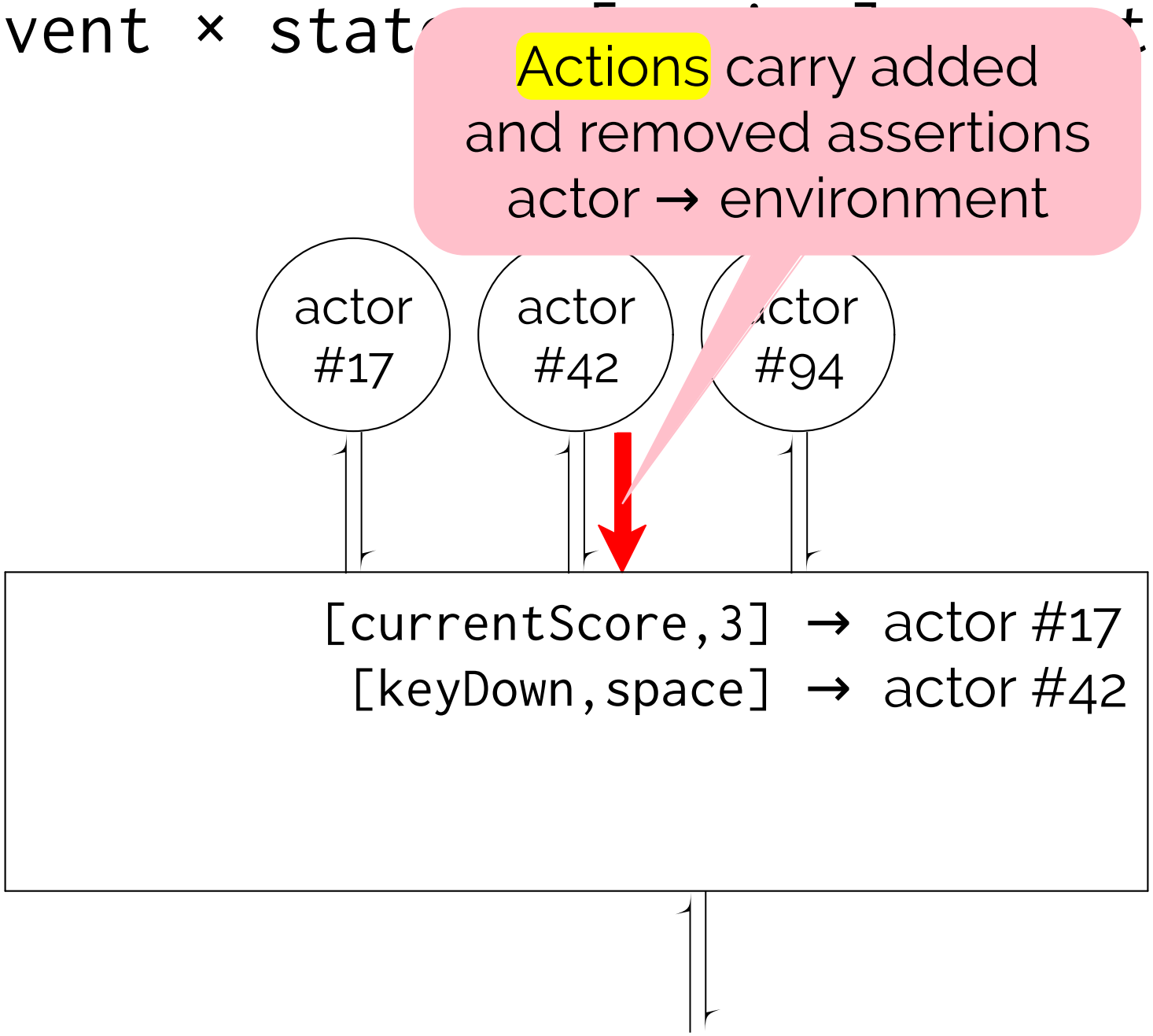
```
event × state → [action] × state
```



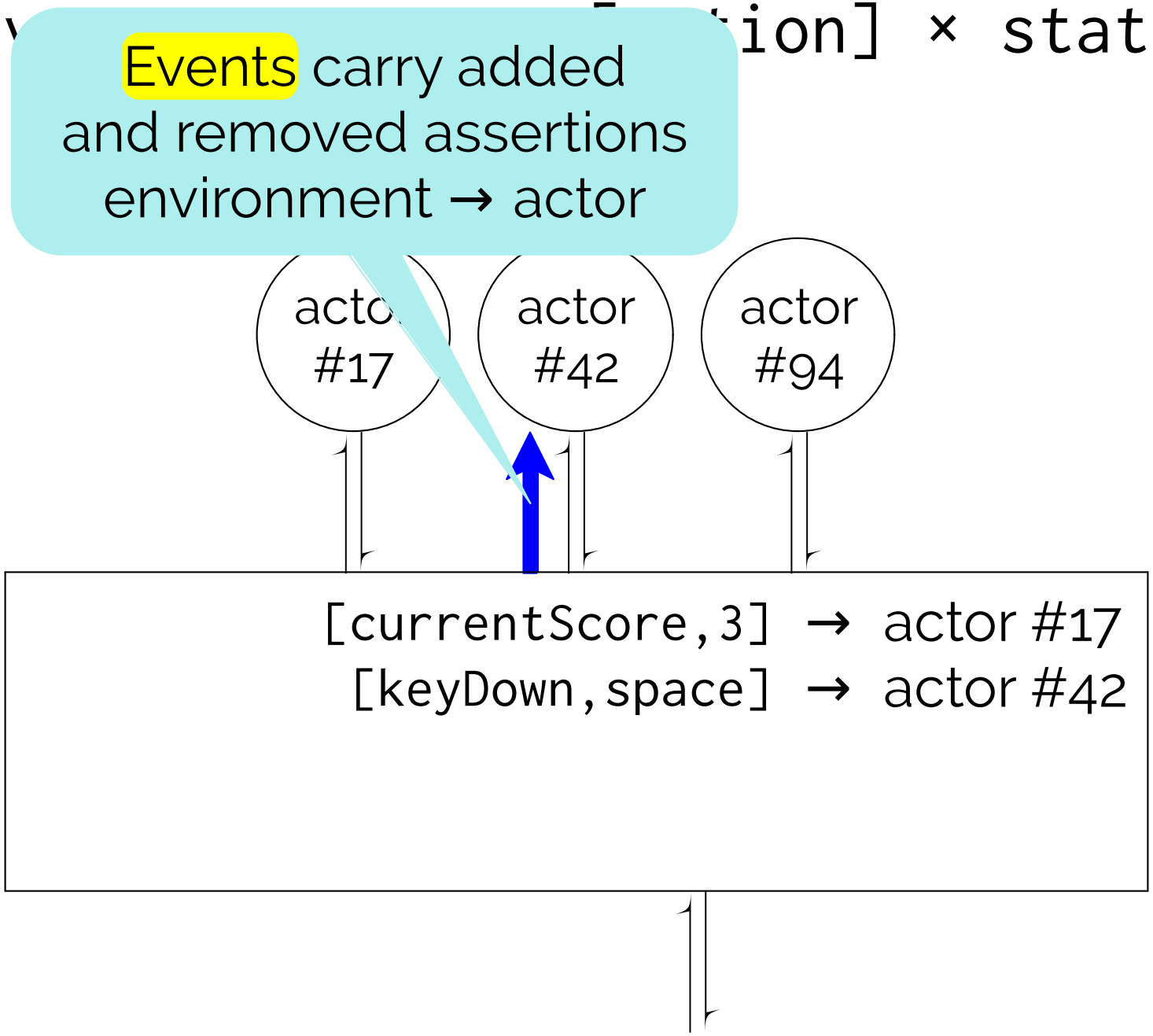
`event × state → [action] × state`



event × stat…te

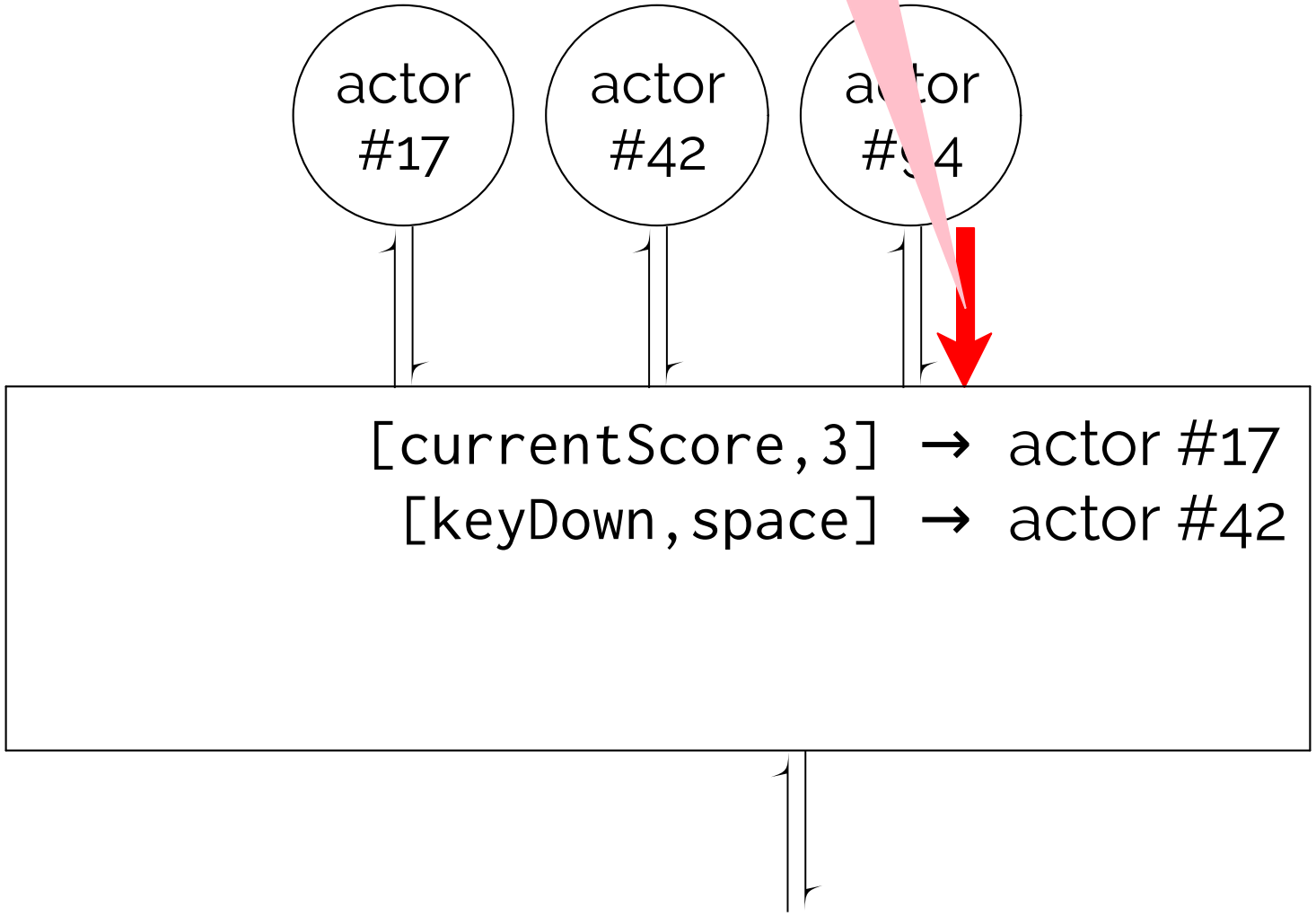


ev[illegible]ion] × state


Events carry added
and removed assertions
environment → actor
actor
#17
actor
#42
actor
#94
[currentScore,3] → actor #17
[keyDown,space] → actor #42

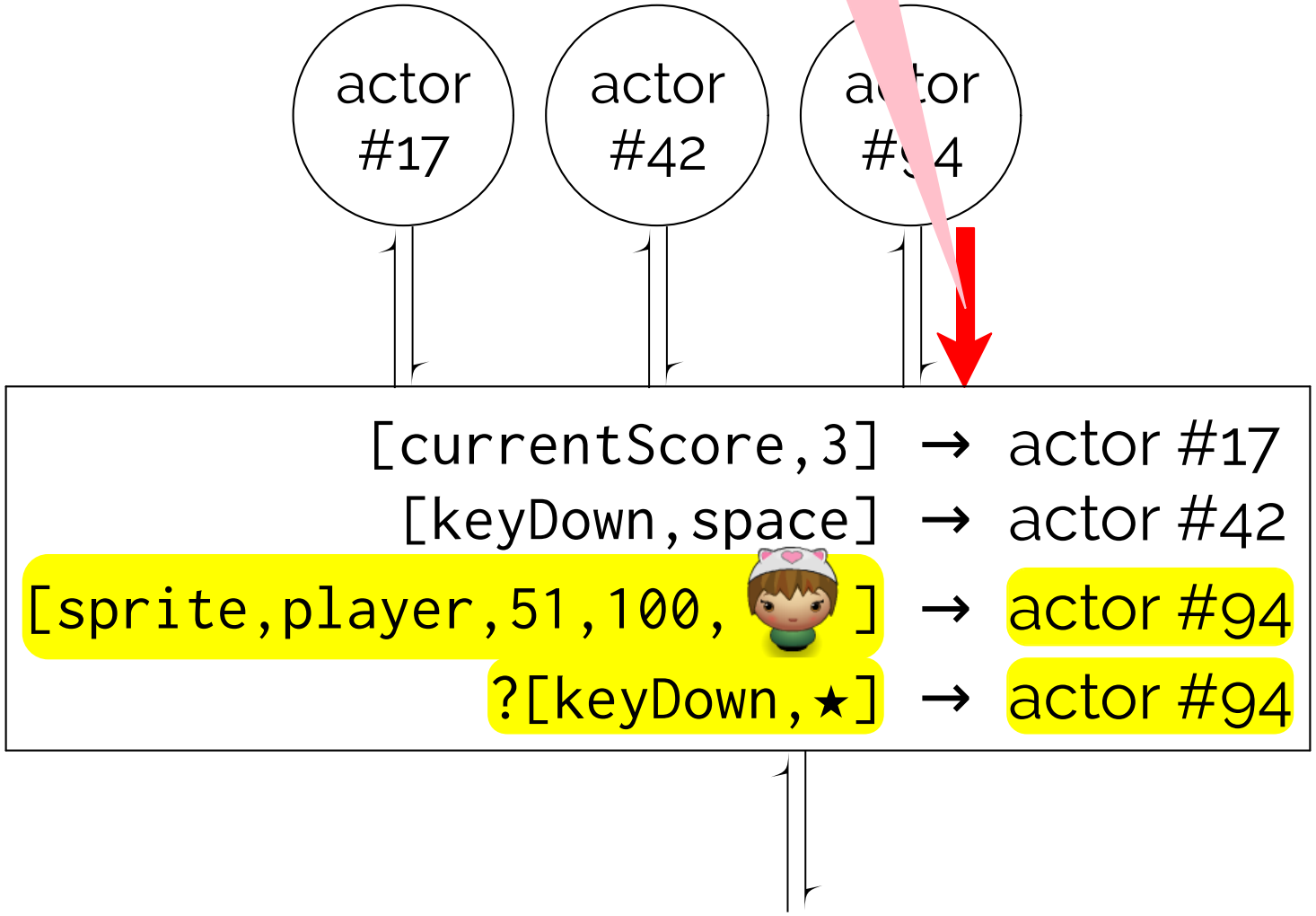
event

```
assert( [sprite,player,51,100, ] ),
assert( ?[keyDown,★] )
```

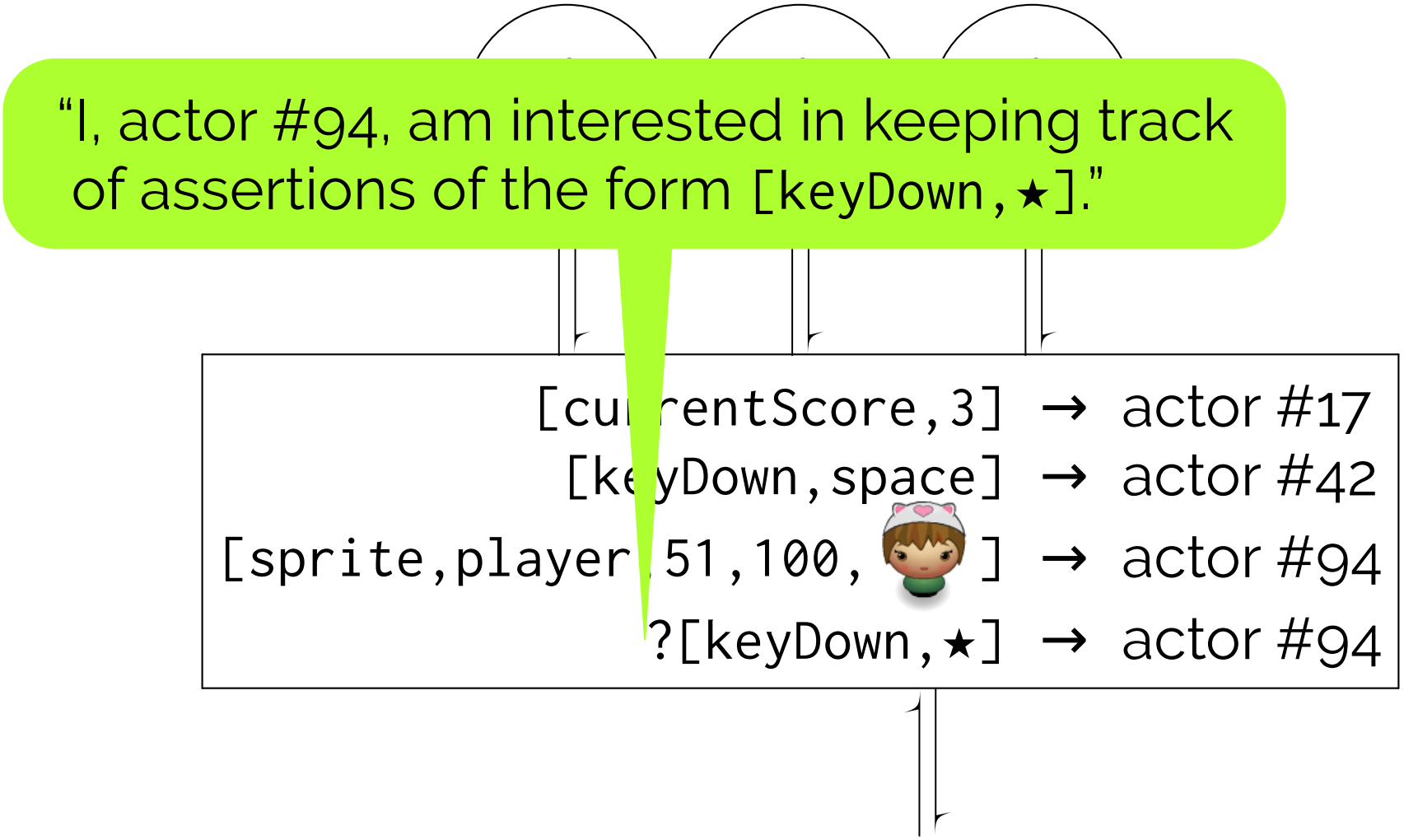


event

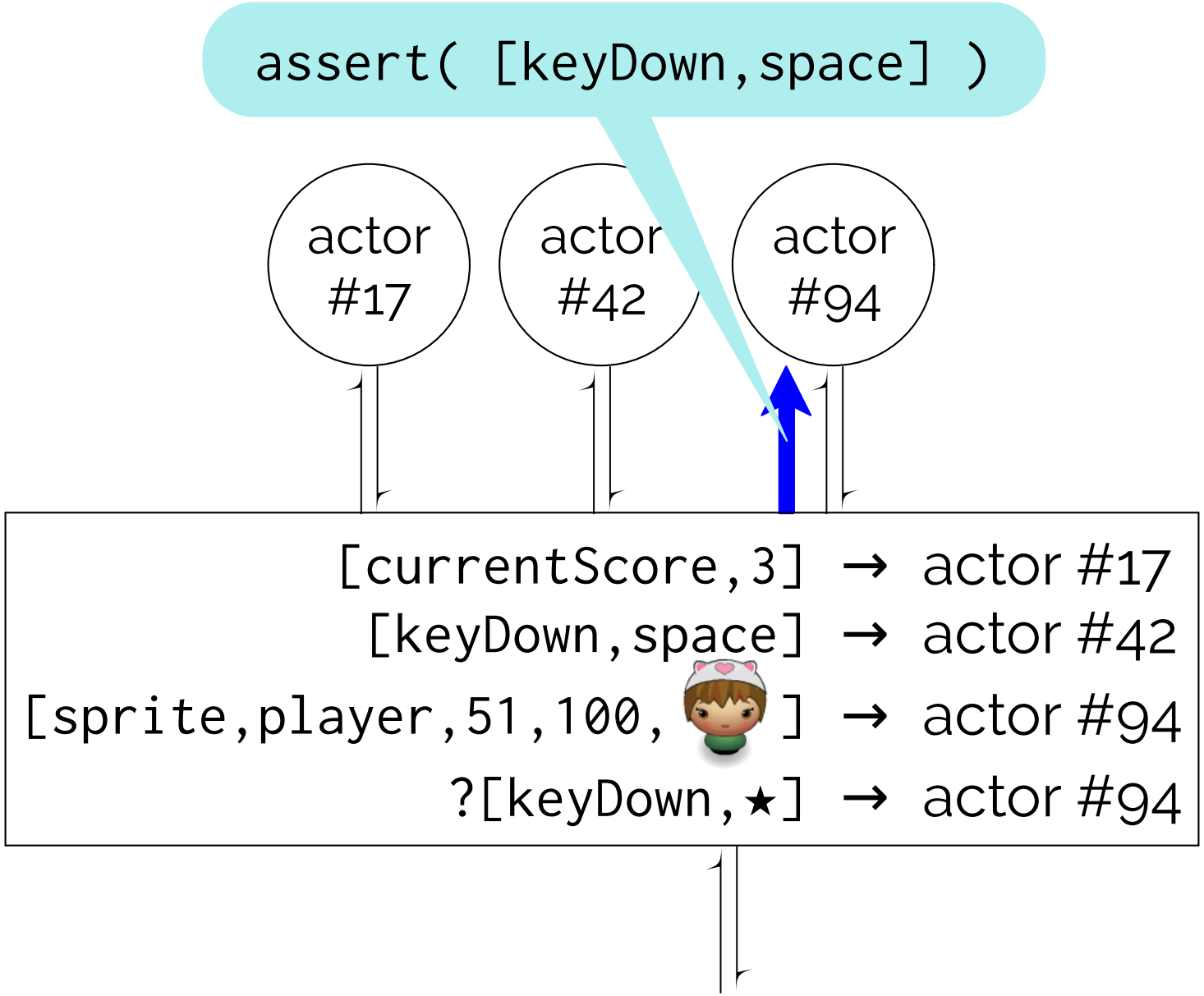
```
assert( [sprite,player,51,100, ] ),
assert( ?[keyDown,★] )
```



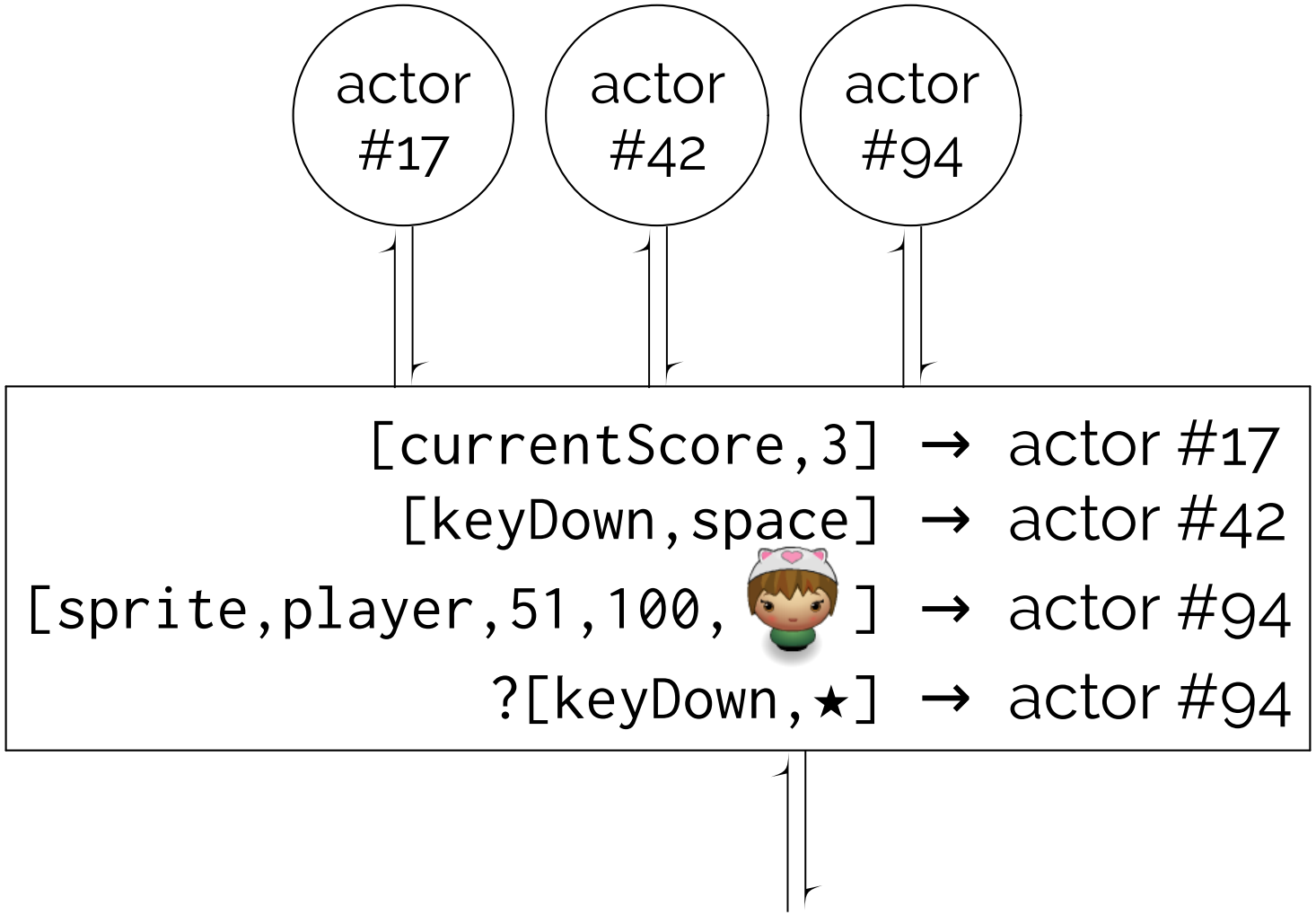
`event × state → [action] × state`



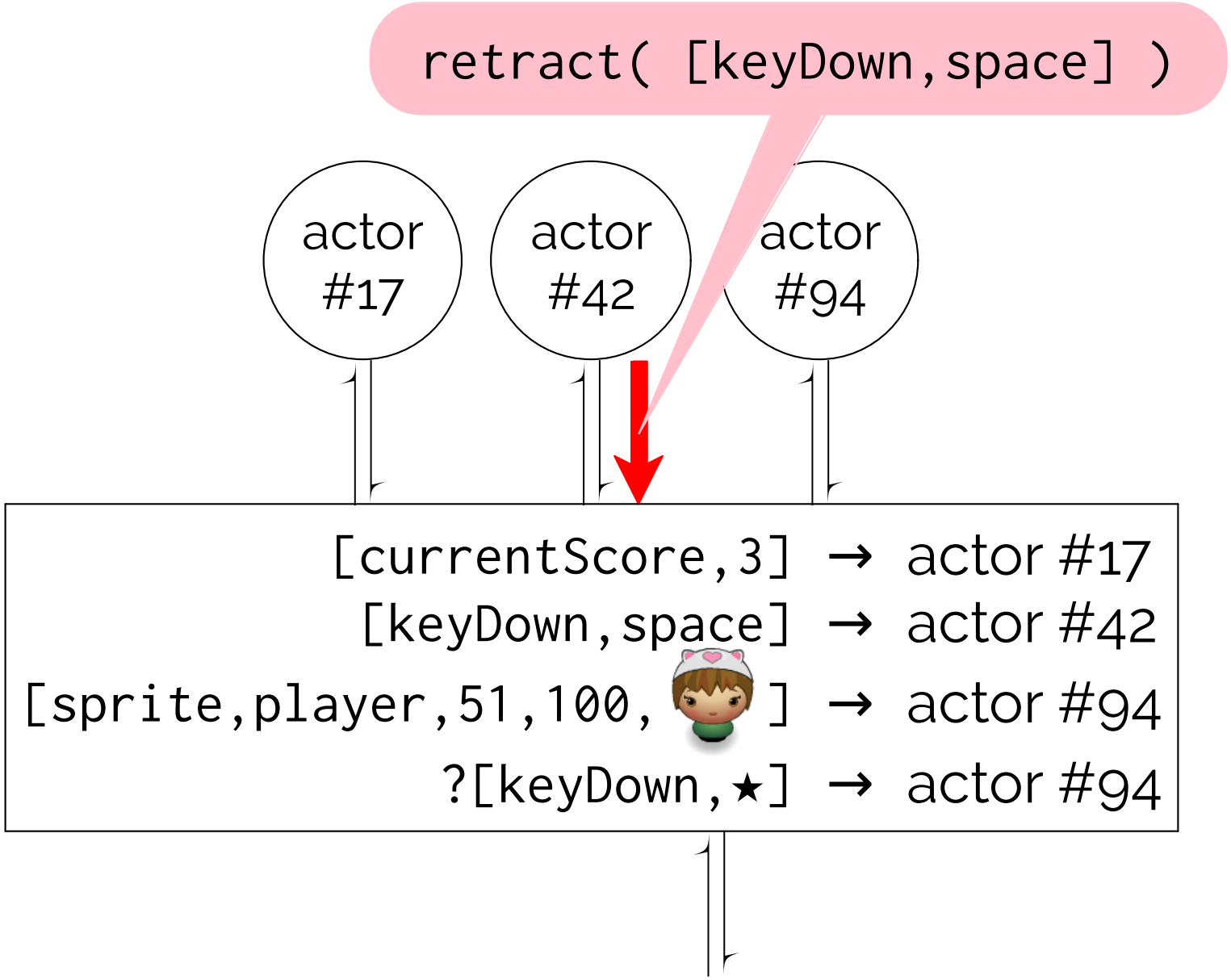
event × state → [action] × state



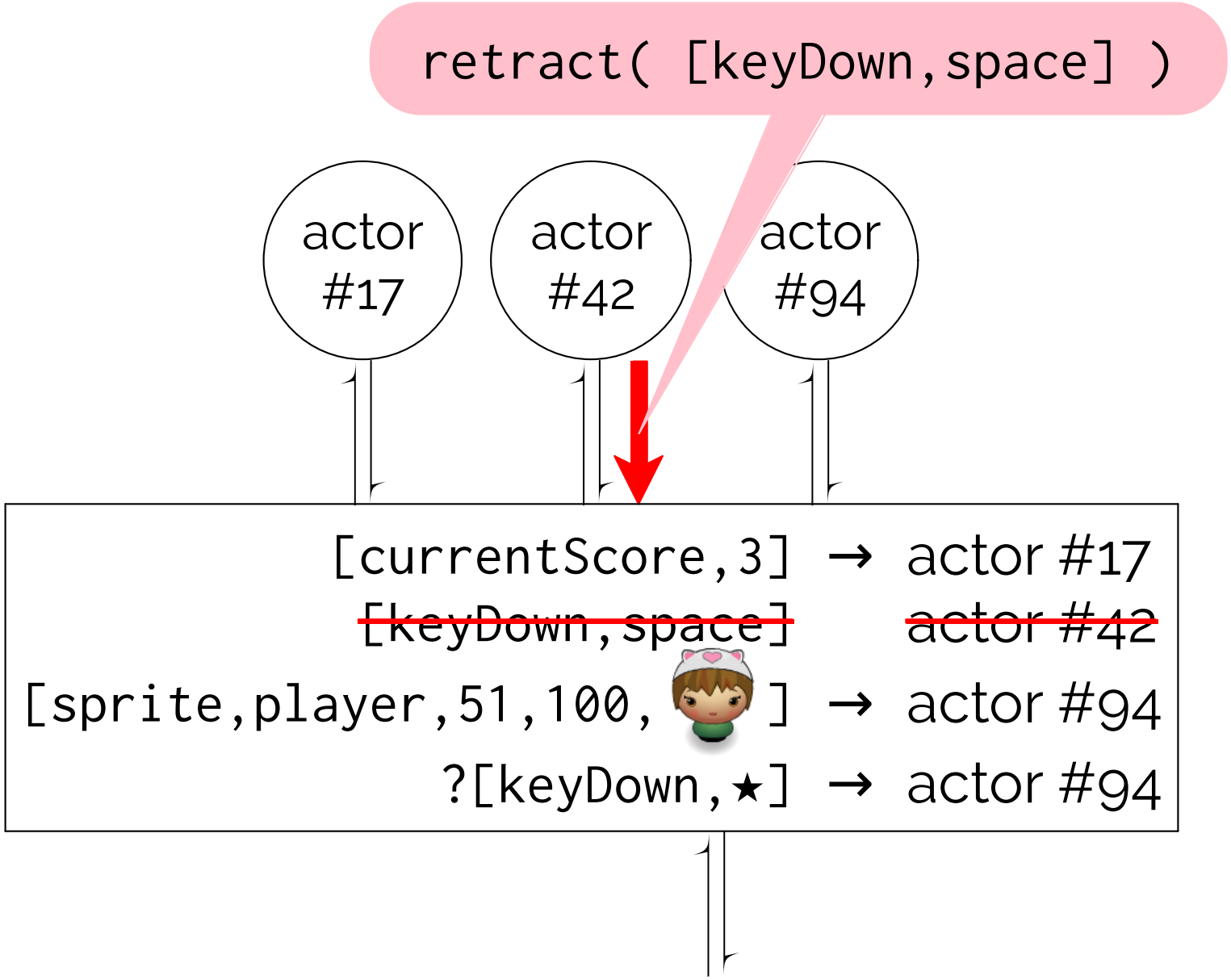
event × state → [action] × state



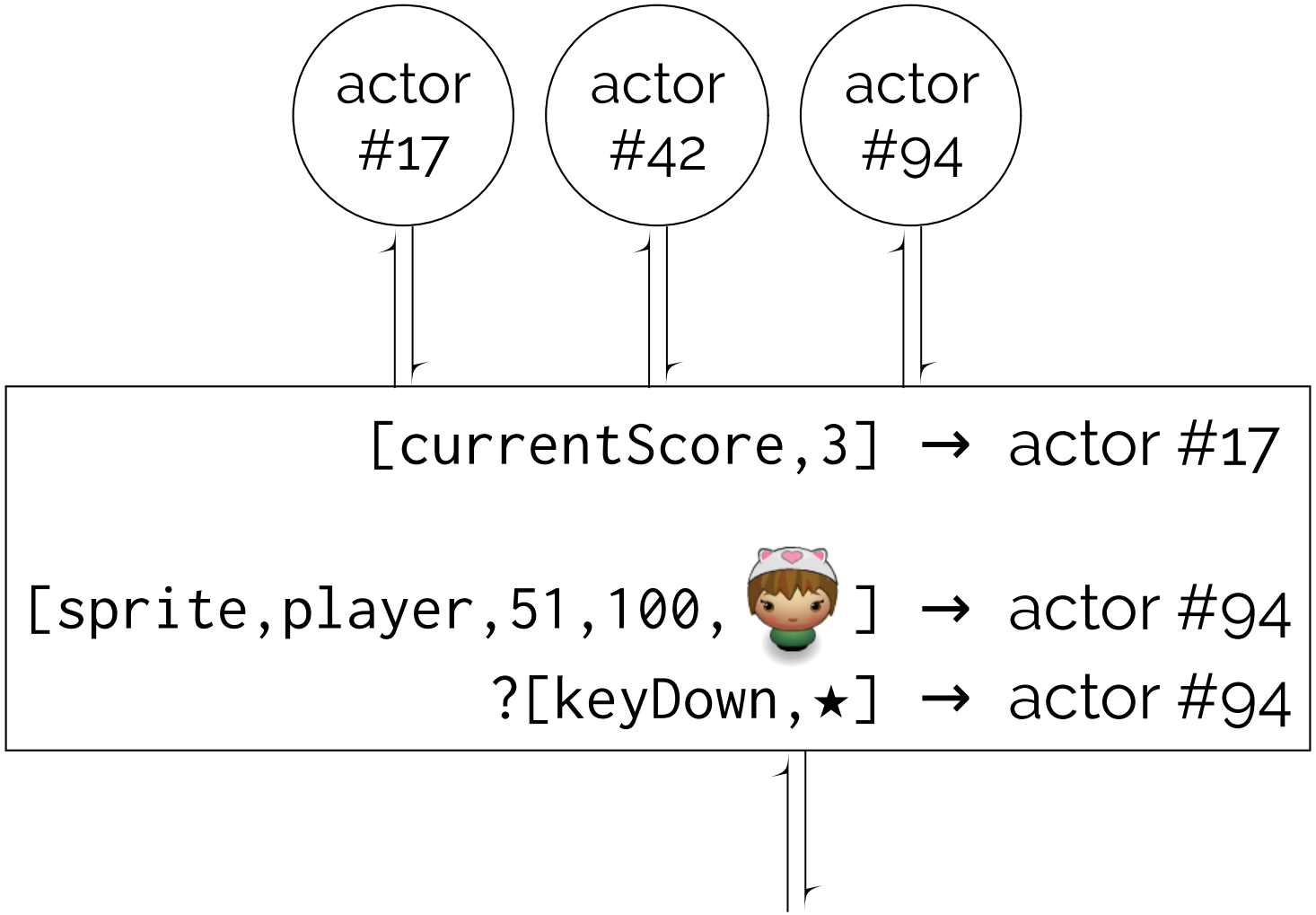
event × state → [action] × state



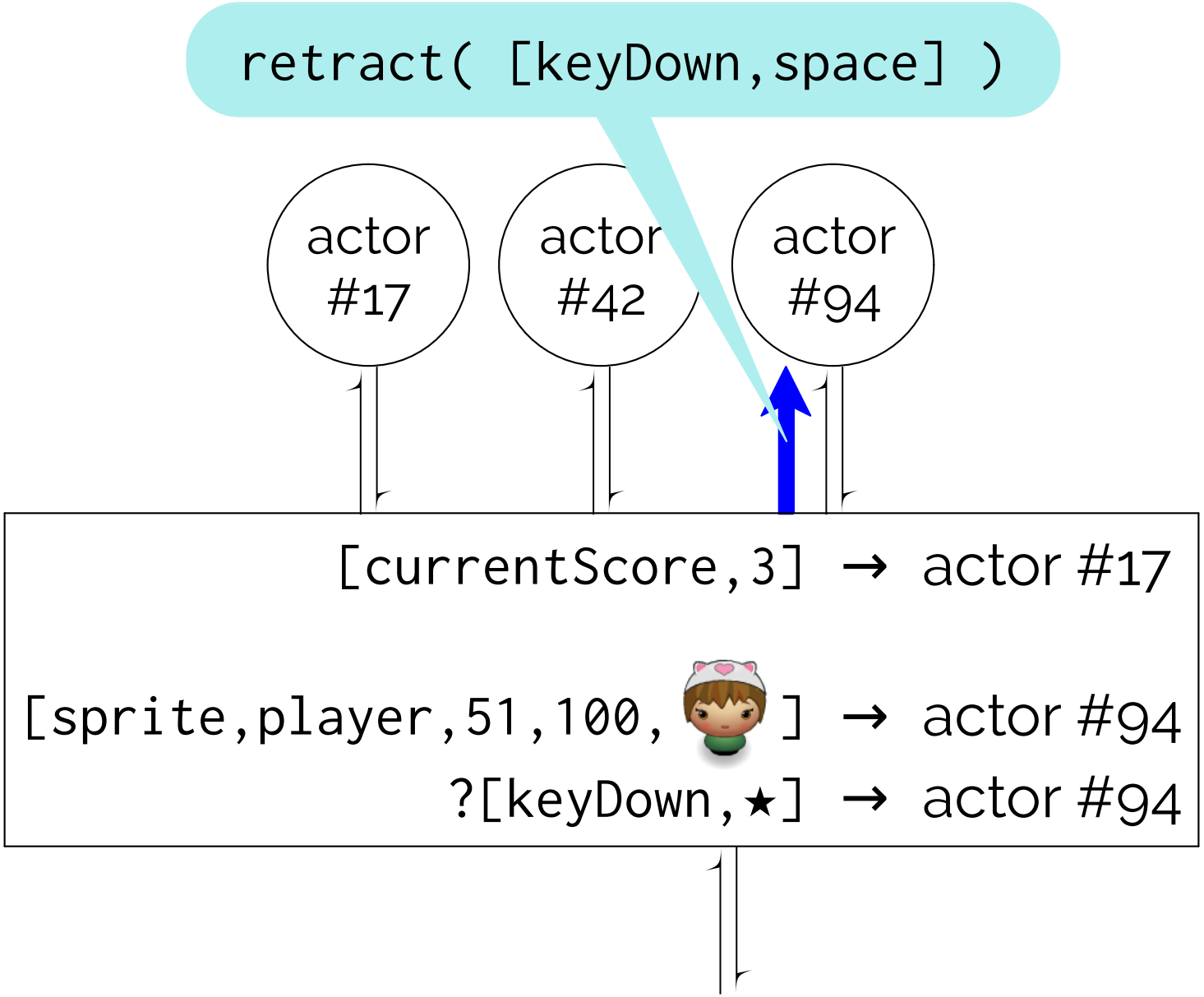
event × state → [action] × state



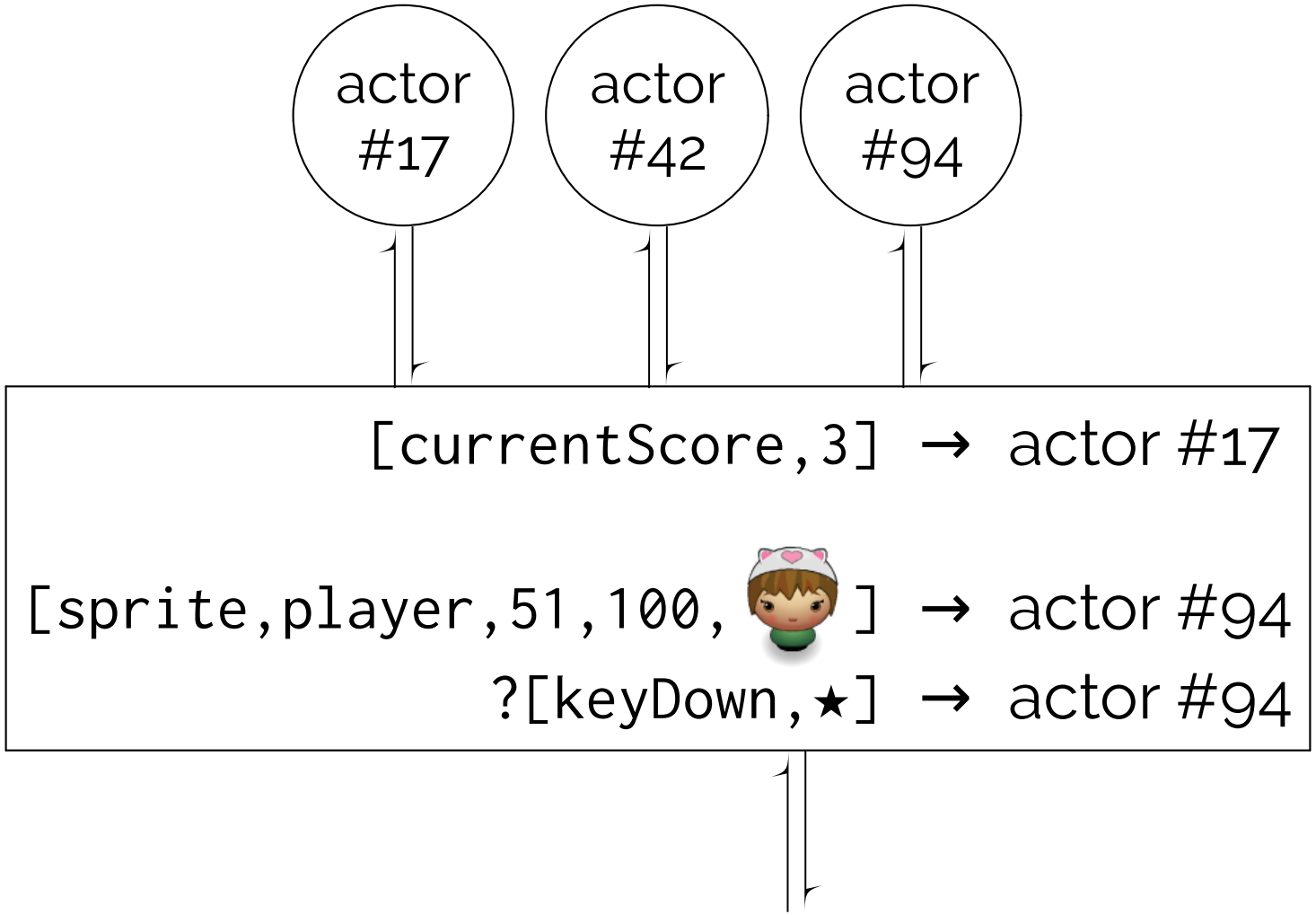
event × state → [action] × state



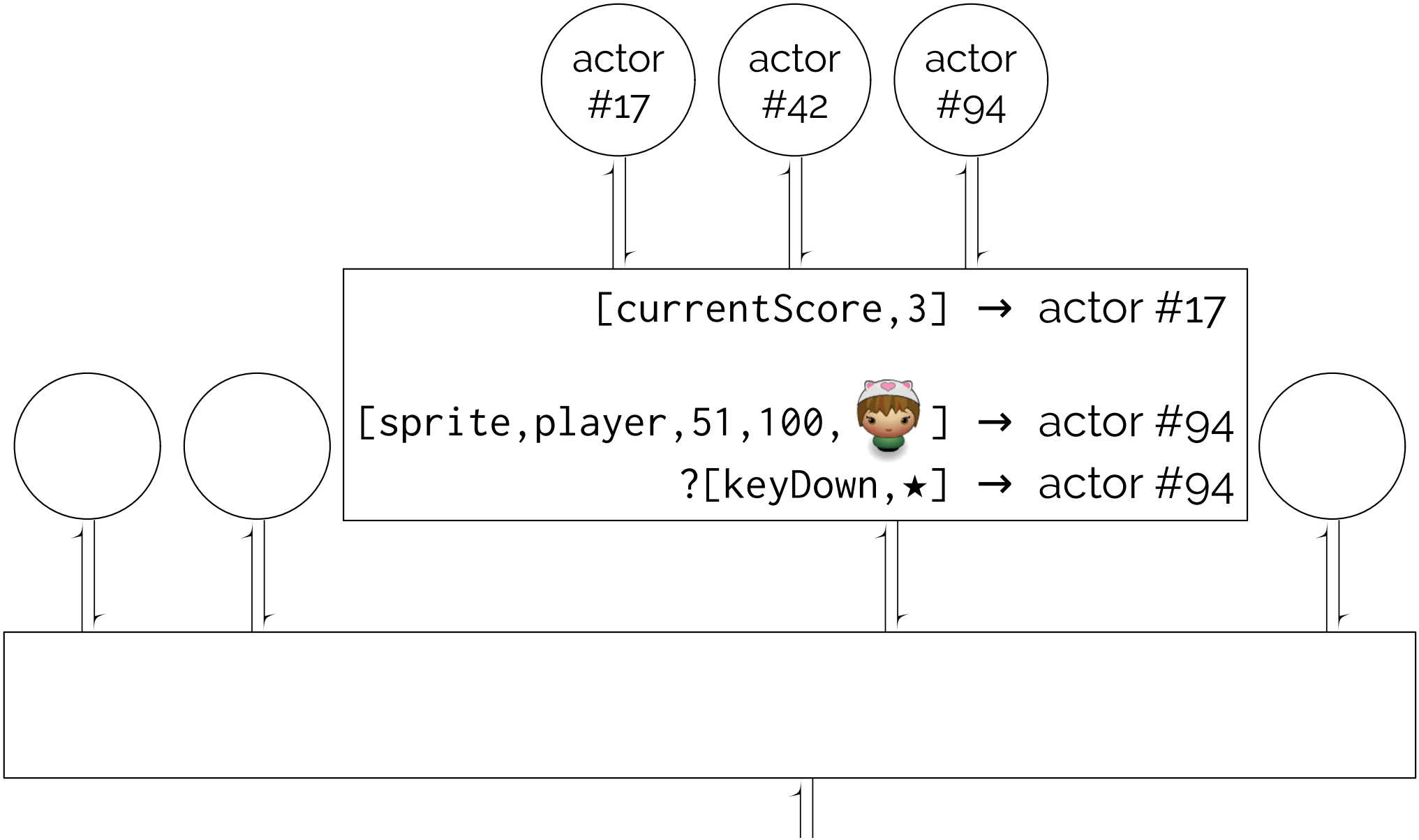
event × state → [action] × state



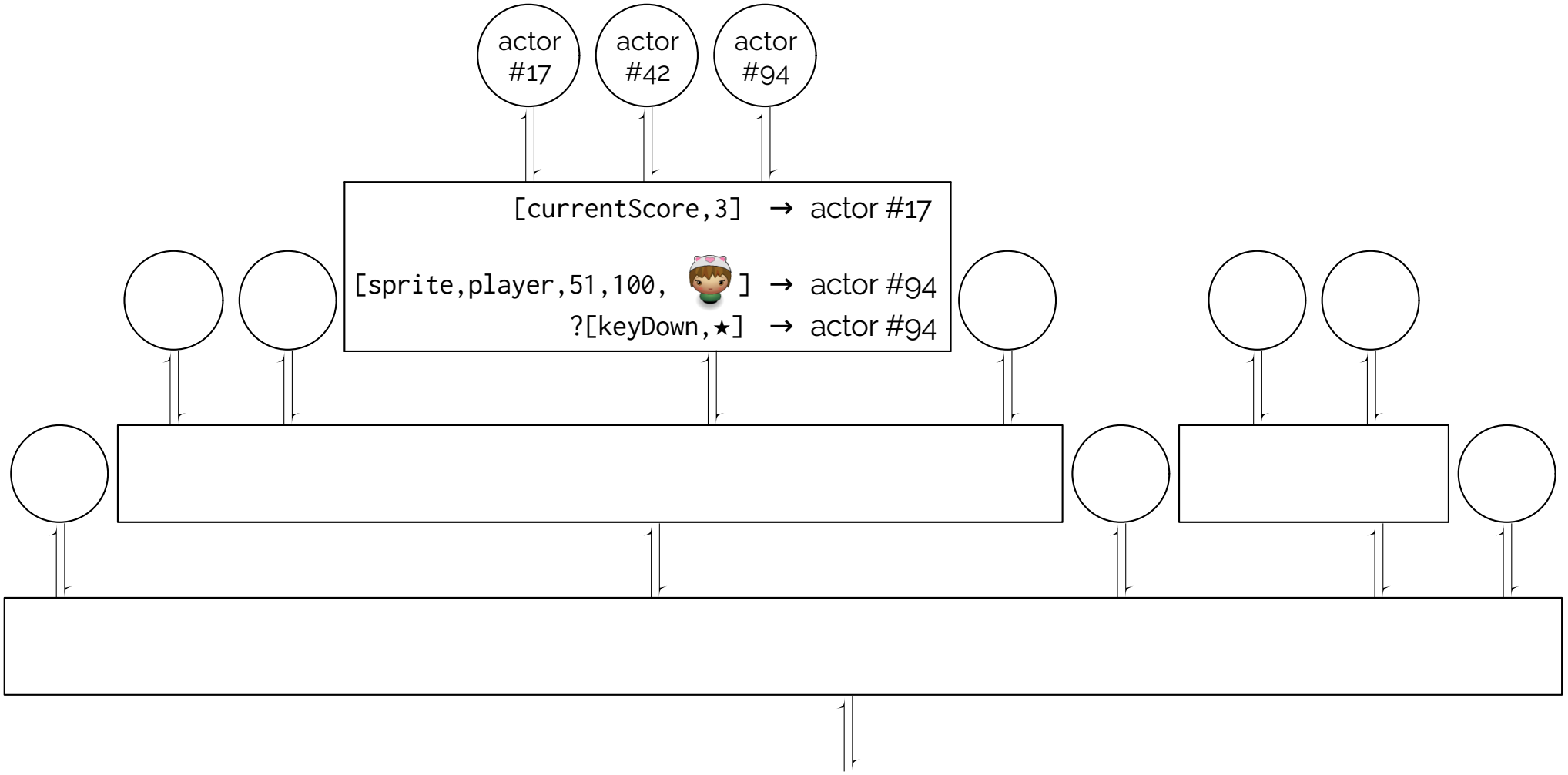
event × state → [action] × state



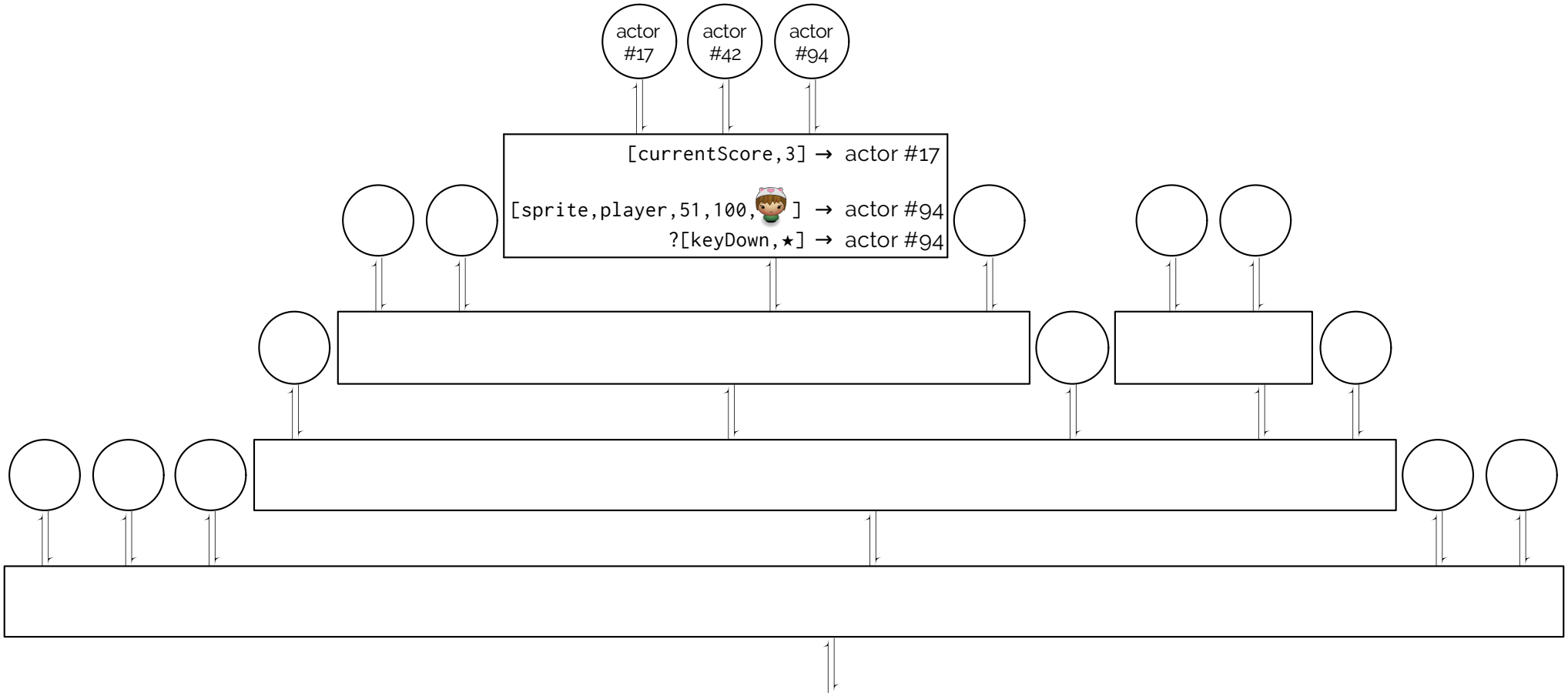
event × state → [action] × state



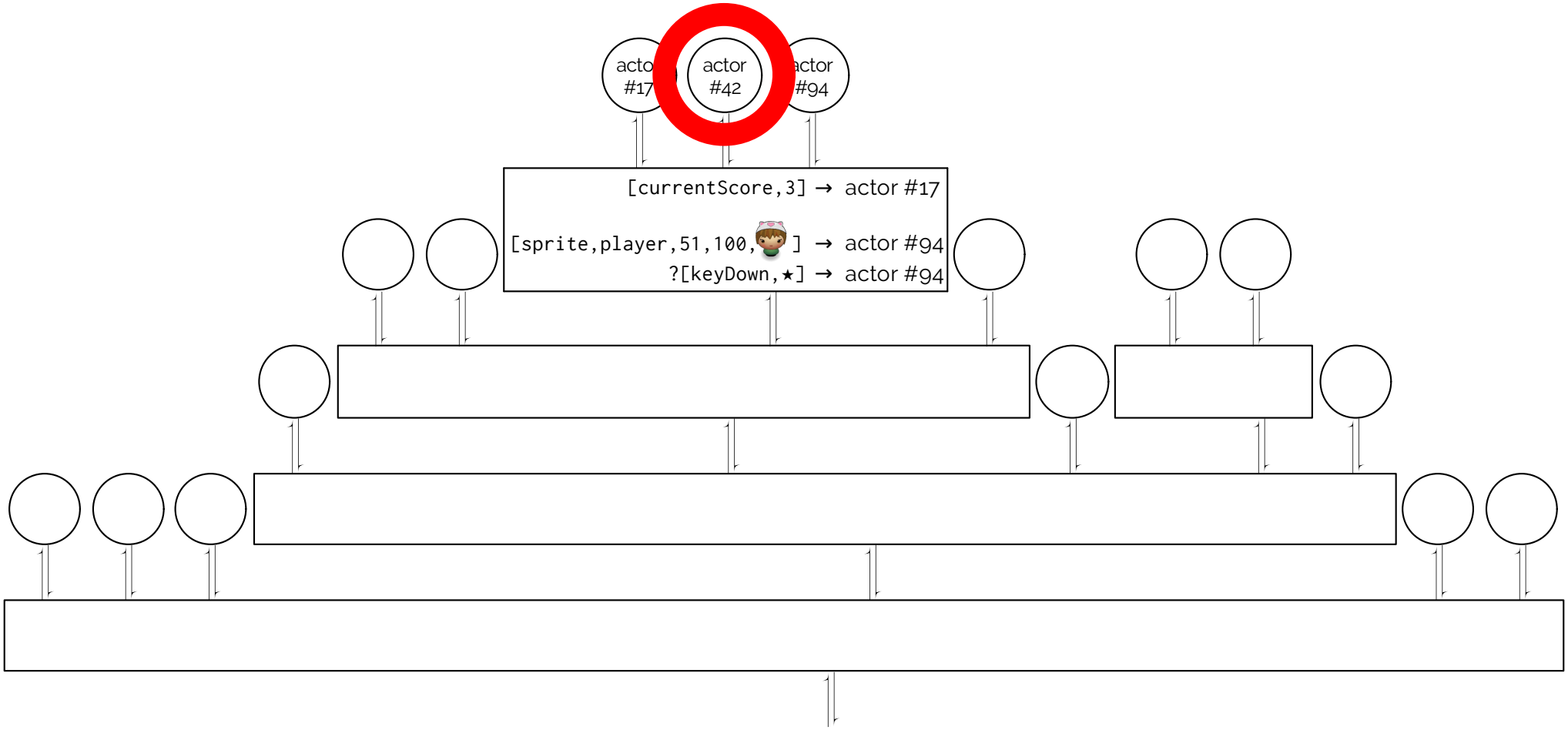
event × state → [action] × state



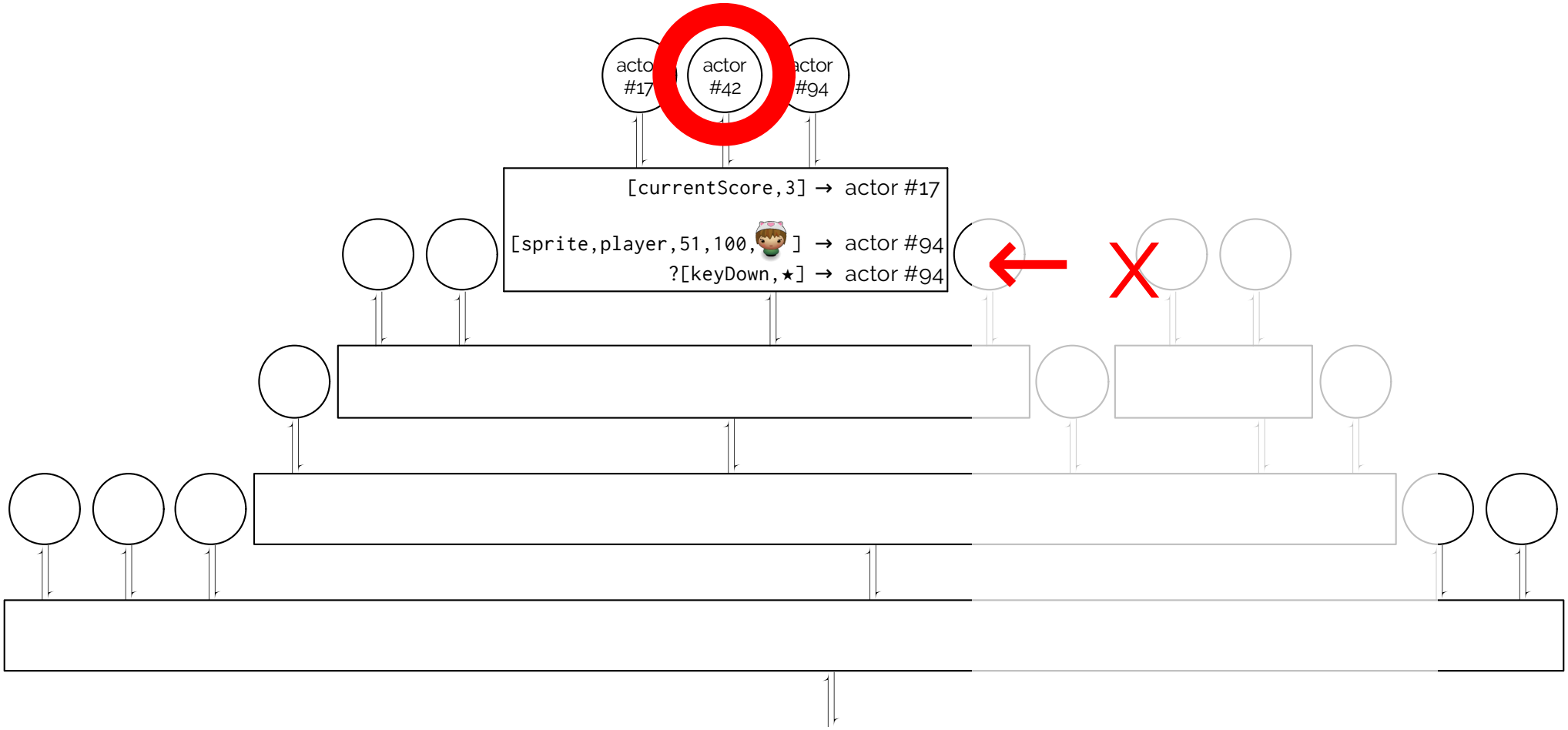
event × state → [action] × state



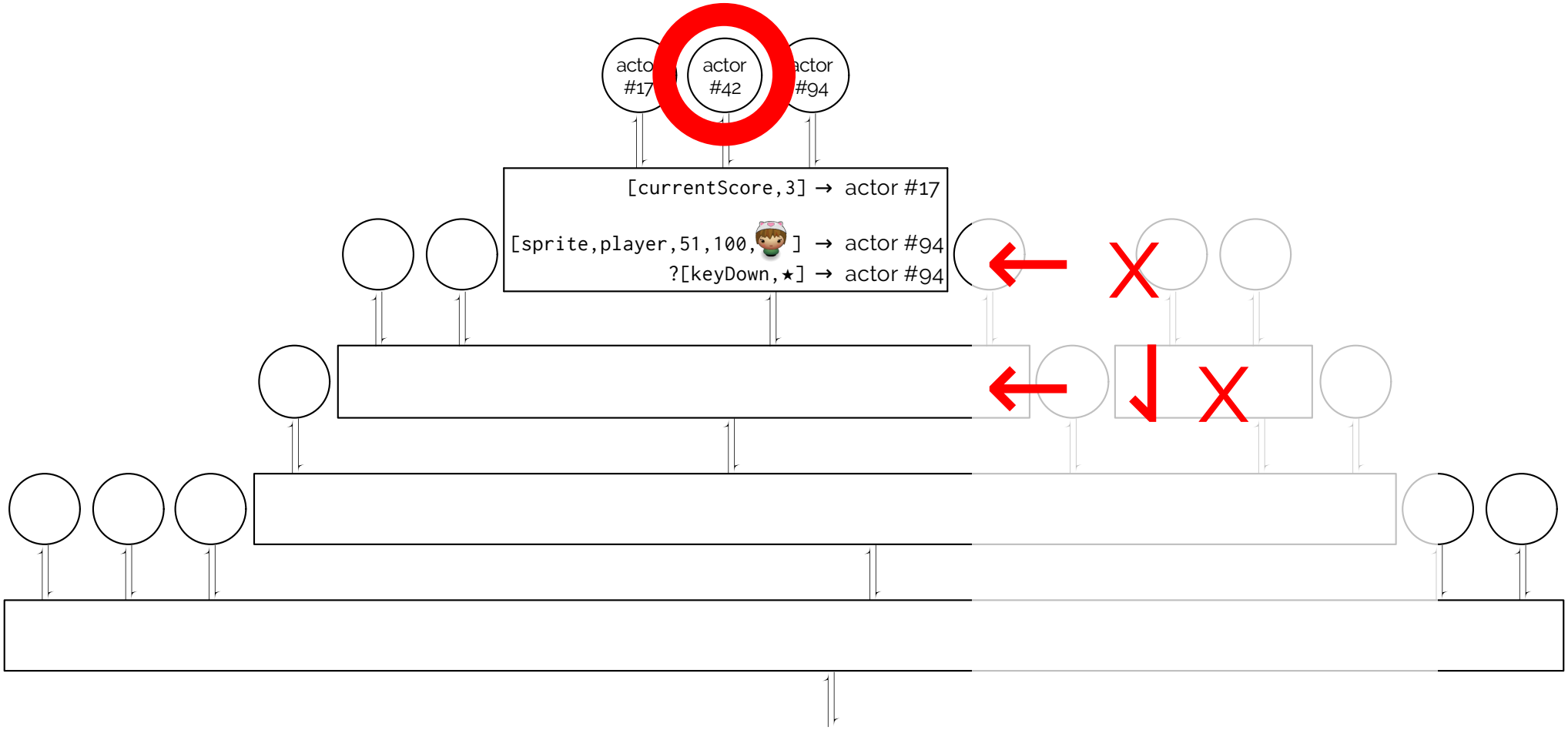
event × state → [action] × state



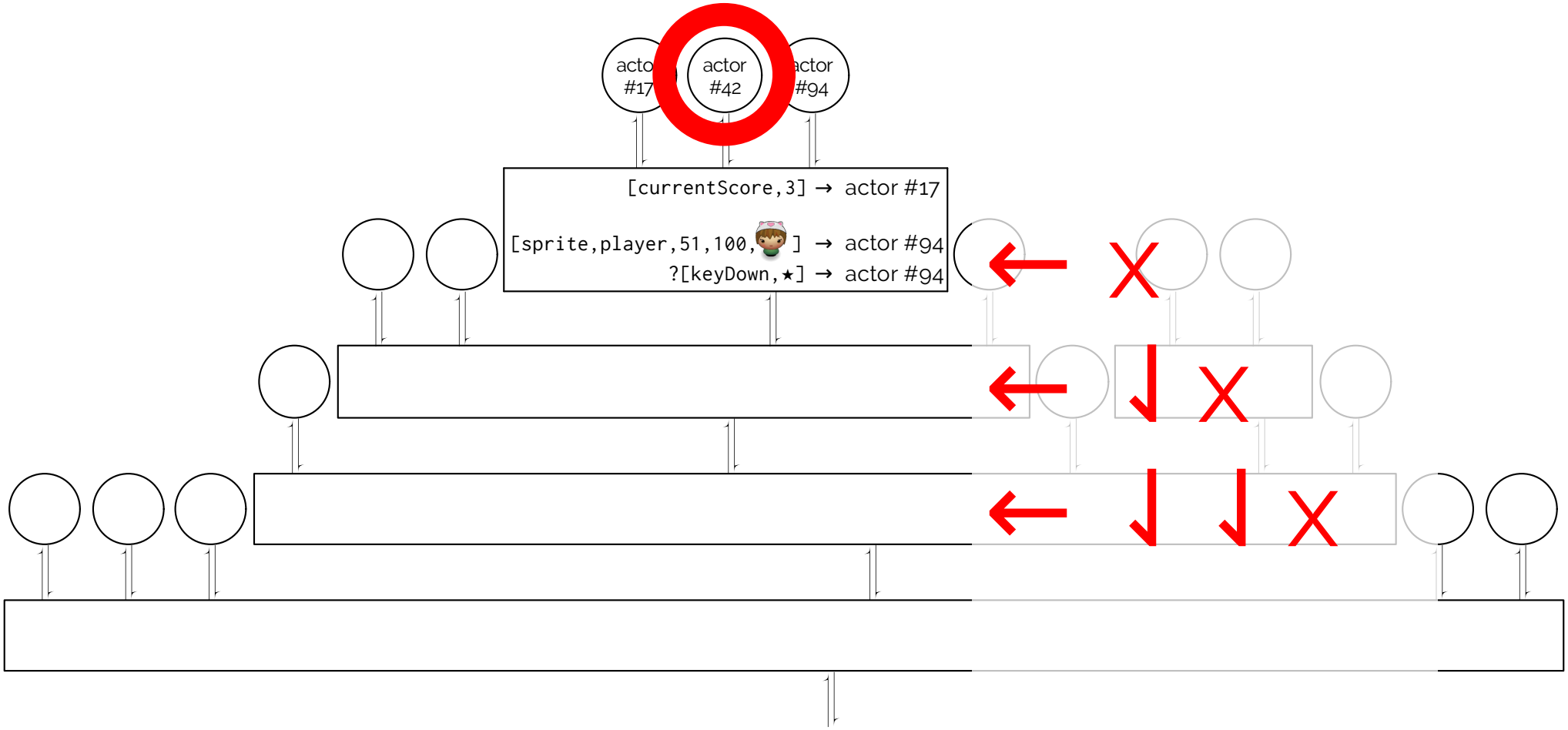
event × state → [action] × state



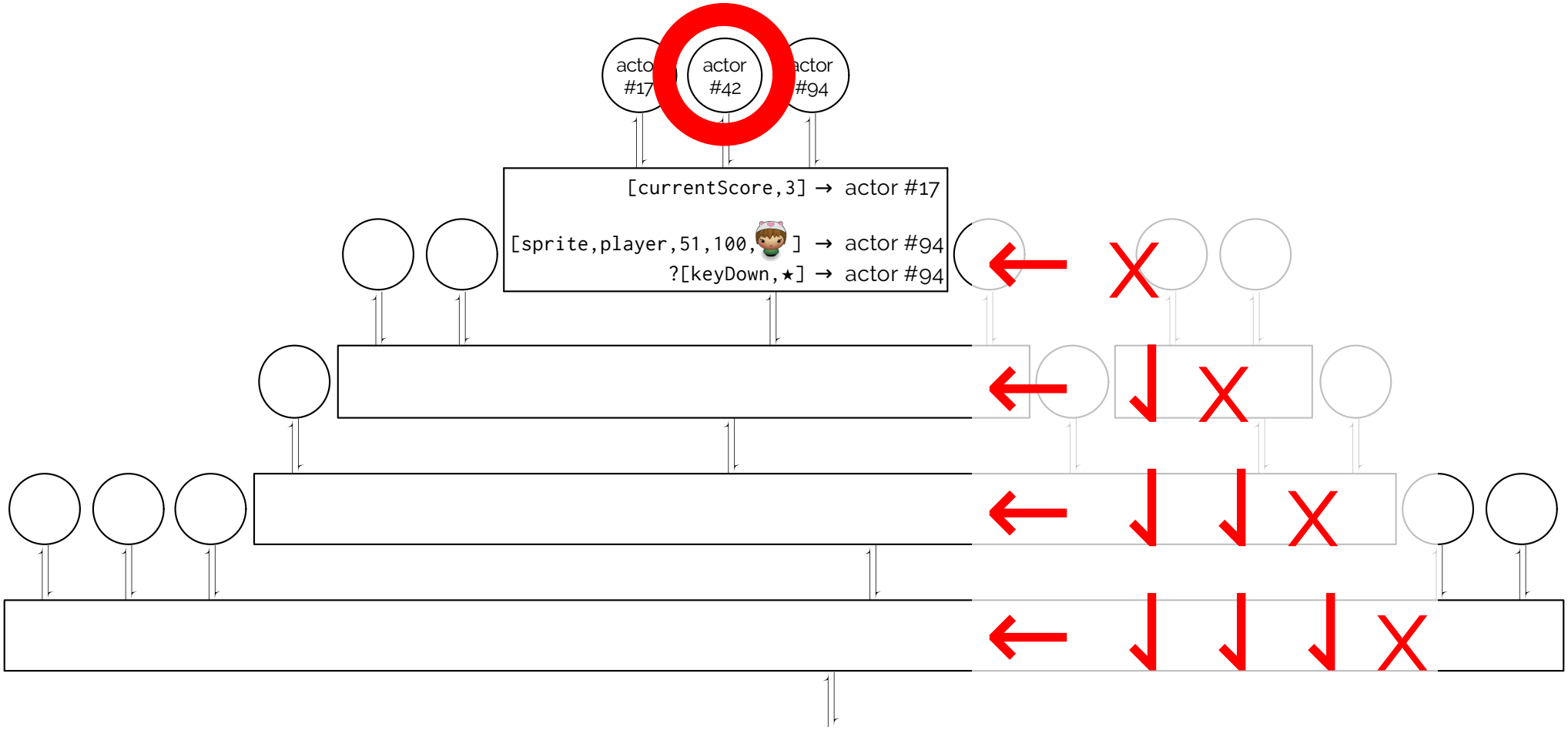
event × state → [action] × state



event × state → [action] × state



event × state → [action] × state



Messages are transient assertions

```
< [incrementScoreBy,3] >
```

~

```
assert( [incrementScoreBy,3] )
```

followed by

```
retract( [incrementScoreBy,3] )
```

(See “Coordinated Concurrent Programming in Syndicate” (ESOP 2016) for full detail of the semantics)

Syndicate Implementations

Racket macros & support library

`#lang syndicate`

Ohm-based translation to ECMAScript 5

Browser & node

Syndicate Implementations

Racket macros & support library

`#lang syndicate`

Ohm-based translation to ECMAScript 5

Browser & node

Syndicate DSL by example

- Mapping events to components
- Managing conversational state
- Monitoring changes in shared state

Mapping events to components

"start" checker

player

collision detector

...

```
↓?(controller-event 'start #t)  →  "start" checker
  ↓?(controller-event 'left ★)  →  player
              ↓?(clock-tick)  →  player
        ↓(sprite 5 'player)  →  player
 (game-piece-state 'player 5)  →  player
     ?(game-piece-state ★ ★)  →  collision detector
```

↑ game

controller driver

```
?(controller-event 'start #t)  →  game
  ?(controller-event 'left ★)  →  game
              ?(clock-tick)  →  game
        (sprite 5 'player)  →  game
```

Mapping events to components

"start" checker

player

collision detector

...

```
↓?(controller-event 'start #t)  →  "start" checker
  ↓?(controller-event 'left ★)  →  player
                                →  player
                                →  player
(game-piece-state 'player 5)    →  player
     ?(game-piece-state ★ ★)    →  collision detector
```

↑ game

Interest in START presses at next outer dataspace

```
?(controller-event 'start #t)  →  game
 ?(controller-event 'left ★)   →  game
              ?(clock-tick)    →  game
       (sprite 5 'player)      →  game
```

Mapping events to components

"start" checker

player

collision detector

...

```
↓?(controller-event 'start #t)  →  "start" checker
 ↓?(controller-event 'left ★)   →  player
                ↓?(clock-tick)  →  player
                                →
                ?(game-piece-state ★ ★)  →  collision detector
```

↑ game

controller

Interest in LEFT presses/releases at next dataspace

```
?(controller-event 'start #t)  →  game
 ?(controller-event 'left ★)   →  game
               ?(clock-tick)   →  game
        (sprite 5 'player)     →  game
```

Mapping events to components

"start" checker

player

collision detector

...

```
↓?(controller-event 'start #t)  →  "start" checker
  ↓?(controller-event 'left ★)  →  player
              ↓?(clock-tick)  →  player
        ↓(sprite 5 'player)  →  player
```

Interest in clock ticks at next outer dataspace

ector

↑ game

controller driver

```
?(controller-event 'start #t)  →  game
  ?(controller-event 'left ★)  →  game
              ?(clock-tick)  →  game
        (sprite 5 'player)  →  game
```

Mapping events to components

"start" checker

player

collision detector

...

```
↓?(controller-event 'start #t)  →  "start" checker
  ↓?(controller-event 'left ★)  →  player
              ↓?(clock-tick)  →  player
        ↓(sprite 5 'player)  →  player
  (game-piece-state  player 5)  →  player
```

Sprite published to next outer dataspace

↑game

controller driver

```
?(controller-event 'start #t)  →  game
  ?(controller-event 'left ★)  →  game
              ?(clock-tick)  →  game
        (sprite 5 'player)  →  game
```

Mapping events to components

"start" checker

player

collision detector

...

```
↓?(controller-event 'start #t)  →  "start" checker
  ↓?(controller-event 'left ★)  →  player
              ↓?(clock-tick)  →  player
        ↓(sprite 5 'player)  →  player
(game-piece-state 'player 5)  →  player
    ?(game-pi ce-state ★ ★)  →  collision detector
```

↑ game

controller driver

Game-piece state published locally

```
?(controller-event 'start #t)  →  game
  ?(controller-event 'left ★)  →  game
              ?(clock-tick)  →  game
        (sprite 5 'player)  →  game
```

Mapping events to components

"start" checker

player

collision detector

...

```
↓?(controller-event 'start #t)  →  "start" checker
  ↓?(controller-event 'left ★)  →  player
              ↓?(clock-tick)  →  player
        ↓(sprite 5 'player)  →  player
  (game-piece-state 'player 5)  →  player
      ?(game-piece-state ★ ★)  →  collision detector
```

↑ game

controller driver

Subscription to game-piece states

```
?(controller-event 'start #t)  →  game
  ?(controller-event 'left ★)  →  game
              ?(clock-tick)  →  game
        (sprite 5 'player)  →  game
```

Mapping events to components

"start" checker

player

collision detector

...

controller driver

```
↓?(controller-event 'start #t)  →  "start" checker
  ↓?(controller-event 'left ★)  →  player
              ↓?(clock-tick)    →  player
       ↓(sprite 5 'player)      →  player
(game-piece-state 'player 5)    →  player
    ?(game-piece-state ★ ★)     →  collision detector
```

↑ game

Interest in START presses at local dataspace

```
?(controller-event 'start #t)  →  game
  ?(controller-event 'left ★)  →  game
              ?(clock-tick)    →  game
       (sprite 5 'player)      →  game
```

Mapping events to components

"start" checker

player

collision detector

...

```
↓?(controller-event 'start #t)  →  "start" checker
  ↓?(controller-event 'left ★)  →  player
              ↓?(clock-tick)  →  player
       ↓(sprite 5 'player)  →  player
(game-piece-state 'player 5)  →  player
    ?(game-piece-state ★ ★)  →  collision detector
```

↑ game

controller driver

Interest in LEFT presses/releases at local dataspace

```
?(controller-event 'start #t)  →  game
  ?(controller-event 'left ★)  →  game
              ?(clock-tick)  →  game
       (sprite 5 'player)  →  game
```

Mapping events to components

"start" checker

player

collision detector

...

controller driver

```
↓?(controller-event 'start #t)  →  "start" checker
  ↓?(controller-event 'left ★)  →  player
              ↓?(clock-tick)    →  player
        ↓(sprite 5 'player)     →  player
(game-piece-state 'player 5)    →  player
    ?(game-piece-state ★ ★)     →  collision detector
```

↑ game

Interest in clock ticks at local dataspace

```
?(controller-event 'left ★)  →  game
          ?(clock-tick)      →  game
    (sprite 5 'player)       →  game
```

Mapping events to components

"start" checker

player

collision detector

...

```
↓?(controller-event 'start #t)  →  "start" checker
  ↓?(controller-event 'left ★)  →  player
              ↓?(clock-tick)  →  player
        ↓(sprite 5 'player)  →  player
  (game-piece-state 'player 5)  →  player
      ?(game-piece-state ★ ★)  →  collision detector
```

↑ game

controller driver

Assertion of sprite position at local dataspace

```
              ?(clock-tick)  →  game
        (sprite 5 'player)  →  game
```

Mapping events to components

"start" checker

player

collision detector

...

```
↓?(controller-event 'start #t)  →  "start" checker
  ↓?(controller-event 'left ★)  →  player
              ↓?(clock-tick)  →  player
        ↓(sprite 5 'player)  →  player
 (game-piece-state 'player 5)  →  player
     ?(game-piece-state ★ ★)  →  collision detector
```

↑ game

controller driver

```
?(controller-event 'start #t)  →  game
  ?(controller-event 'left ★)  →  game
              ?(clock-tick)  →  game
        (sprite 5 'player)  →  game
```

Mapping events to components

"start" checker

player

collision detector

...

```
↓?(controller-event 'start #t)  →  "start" checker
  ↓?(controller-event 'left ★)  →  player
              ↓?(clock-tick)  →  player
        ↓(sprite 5 'player)  →  player
  (game-piece-state 'player 5)  →  player
      ?(game-piece-state ★ ★)  →  collision detector
```

↑ game

controller driver

Pressing the START key should terminate the game

```
(sprite 5 'player)  →  game
```

Mapping events to components

< (controller-event 'start #t) >

"start" checker

player

collision detector

...

```
↓?(controller-event 'start #t)  →  "start" checker
  ↓(controller-event 'left ★)   →  player
          ↓?(clock-tick)        →  player
      ↓(sprite 5 'player)       →  player
(game-piece-state 'player 5)    →  player
    ?(game-piece-state ★ ★)     →  collision detector
```

↑ game

controller driver

```
?(controller-event 'start #t)  →  game
  ?(controller-event 'left ★)  →  game
          ?(clock-tick)        →  game
      (sprite 5 'player)       →  game
```

Mapping events to components

< (controller-event 'start #t) >

"start" checker

player

collision detector

...

controller driver

```
↓?(controller-event 'start  t)  →  "start" checker
  ↓?(controller-event 'left ★)  →  player
          ↓?(clock-tic k)       →  player
      ↓(sprite 5 'playe )       →  player
 (game-piece-state 'player  )   →  player
     ?(game-piece-state ★ ★)    →  collision detector
```

↑ game

```
?(controller-event 'start #t)  →  game
  ?(controller-event 'left ★)  →  game
          ?(clock-tick)        →  game
      (sprite 5 'player)       →  game
```

< ↓(controller-event 'start #t) >

"start" checker

player

collision detector

...

```
↓?(controller-event 'start #t)  →  "start" checker
  ↓?(controller-event 'left ★)  →  player
              ↓?(clock-tick)  →  player
        ↓(sprite 5 'player)  →  player
  (game-piece-state 'player 5)  →  player
      ?(game-piece-state ★ ★)  →  collision detector
```

↑ game

controller driver

```
?(controller-event 'start #t)  →  game
  ?(controller-event 'left ★)  →  game
              ?(clock-tick)  →  game
        (sprite 5 'player)  →  game
```

Mapp ments

quit-dataspace!

"start" checker

player

collision detector

...

```
↓?(controller-event 'start #t)  →  "start" checker
  ↓?(controller-event 'left ★)  →  player
              ↓?(clock-tick)  →  player
       ↓(sprite 5 'player)  →  player
(game-piece-state 'player 5)  →  player
    ?(game-piece-state ★ ★)  →  collision detector
```

↑ game

controller driver

```
?(controller-event 'start #t)  →  game
  ?(controller-event 'left ★)  →  game
              ?(clock-tick)  →  game
       (sprite 5 'player)  →  game
```

Mapping events to components

"start" checker

player

collision detector

...

```
↓?(controller-event 'start #t)  →  "start" checker
  ↓?(controller-event 'left ★)  →  player
              ↓?(clock-tick)    →  player
        ↓(sprite 5 'player)     →  player
 (game-piece-state 'player 5)   →  player
     ?(game-piece-state ★ ★)    →  collision detector
```

↑ game

controller driver

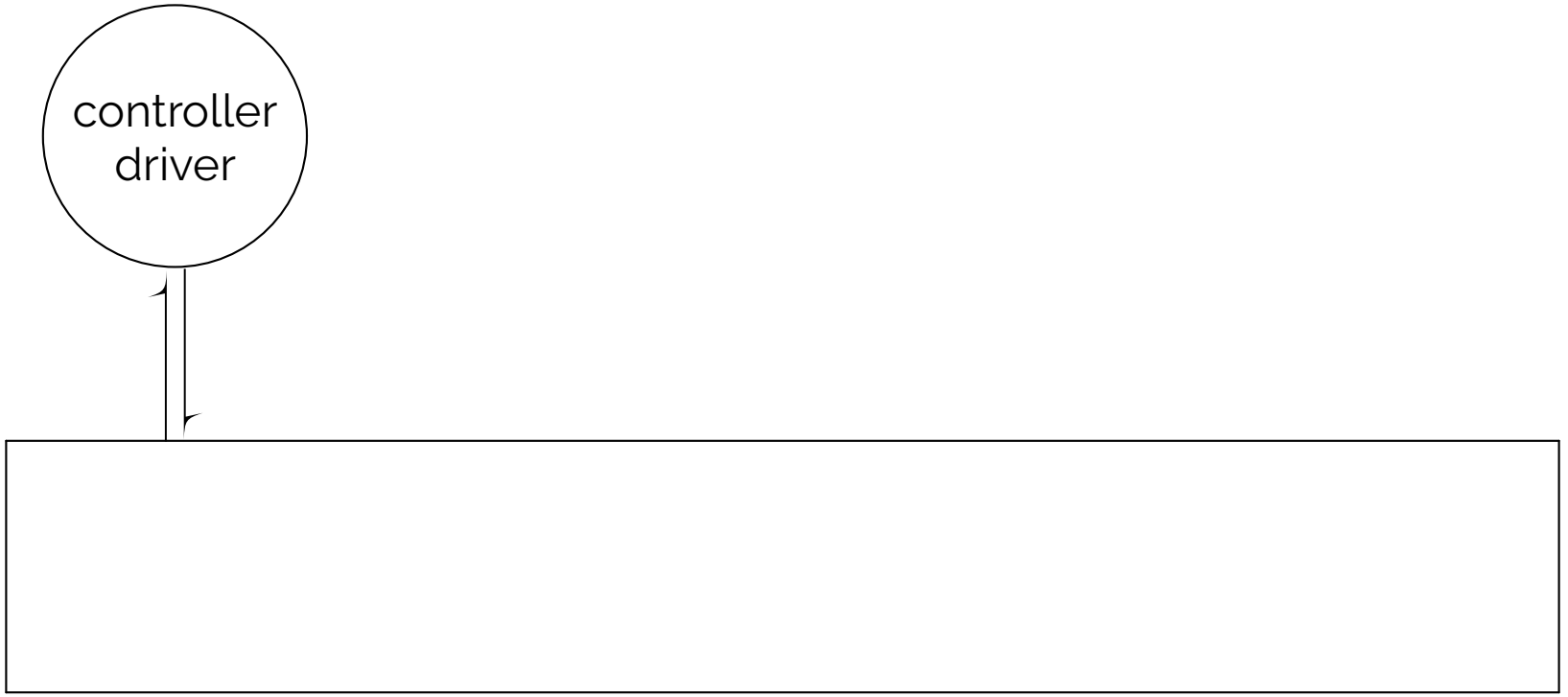
```
?(controller-event 'start #t)  →  game
  ?(controller-event 'left ★)  →  game
              ?(clock-tick)    →  game
        (sprite 5 'player)     →  game
```

Mapping events to components

controller driver

```
?(controller-event 'start #t)      game
?(controller-event 'left ★)        game
?(clock-tick)                      game
(sprite 5 'player)                 game
```

Mapping events to components



```
(spawn-dataspace (spawn-start-button-monitor)
                 (spawn-player)
                 (spawn-collision-detection)
                 ...)

(define (spawn-start-button-monitor)
  (spawn (lambda (evt state)
           (match-event evt
             [(message (at-meta
                        (controller-event 'start #t)))
              (transition state (quit-dataspace))]))
         (void)
         (sub (controller-event 'start #t)
              #:meta-level 1)                    ))
```

```
(spawn-dataspace (spawn-start-button-monitor)
                 (spawn-player)
                 (spawn-collision-detection)
                 ...)
```

```
(define (spawn-start-button-monitor)
  (spawn (lambda (evt state)
           (match-event evt
             [(message (at-meta
                        (controller-event 'start #t)))
              (transition state (quit-dataspace))]))
         (void)
         (sub (controller-event 'start #t)
              #:meta-level 1)                    ))
```

```
(spawn-dataspace (spawn-start-button-monitor)
                 (spawn-player)
                 (spawn-collision-detection)
                 ...)

(define (spawn-start-button-monitor)
  (spawn (lambda (evt state)
           (match-event evt
             [(message (at-meta
                        (controller-event 'start #t)))
              (transition state (quit-dataspace))]))
         (void)
         (sub (controller-event 'start #t)
              #:meta-level 1)                    ))
```

```
(spawn-dataspace (spawn-start-button-monitor)
                 (spawn-player)
                 (spawn-collision-detection)
                 ...)

(define (spawn-start-button-monitor)
  (spawn (lambda (evt state)
           (match-event evt
             [(message (at-meta
                        (controller-event 'start #t)))
              (transition state (quit-dataspace))]))
         (void)
         (sub (controller-event 'start #t)
              #:meta-level 1)                    ))
```

```
(spawn-dataspace (spawn-start-button-monitor)
                 (spawn-player)
                 (spawn-collision-detection)
                 ...)

(define (spawn-start-button-monitor)
  (spawn (lambda (evt state)
           (match-event evt
             [(message (at-meta
                        (controller-event 'start #t)))
              (transition state (quit-dataspace))]))
         (void)
         (sub (controller-event 'start #t)
              #:meta-level 1)                    ))
```

```
(spawn-dataspace (spawn-start-button-monitor)
                 (spawn-player)
                 (spawn-collision-detection)
                 ...)

(define (spawn-start-button-monitor)
  (spawn (lambda (evt state)
           (match-event evt
             [(message (at-meta
                        (controller-event 'start #t)))
              (transition state (quit-dataspace))]))
         (void)
         (sub (controller-event 'start #t)
              #:meta-level 1)))
```

```
(spawn-dataspace (spawn-start-button-monitor)
                 (spawn-player)
                 (spawn-collision-detection)
                 ...)

(define (spawn-start-button-monitor)
  (spawn (lambda (evt state)
           (match-event evt
             [(message (at-meta
                        (controller-event 'start #t)))
              (transition state (quit-dataspace))]))
         (void)
         (sub (controller-event 'start #t)
              #:meta-level 1)                    ))
```

```
(spawn-dataspace (spawn-start-button-monitor)
                 (spawn-player)
                 (spawn-collision-detection)
                 ...)

(define (spawn-start-button-monitor)
  (spawn (lambda (evt state)
           (match-event evt
             [(message (at-meta
                        (controller-event 'start #t)))
              (transition state (quit-dataspace))]))
         (void)
         (sub (controller-event 'start #t)
              #:meta-level 1)                    ))
```

```
(spawn-dataspace (spawn-start-button-monitor)
                 (spawn-player)
                 (spawn-collision-detection)
                 ...)

(define (spawn-start-button-monitor)
  (spawn (lambda (evt state)
           (match-event evt
             [(message (at-meta
                        (controller-event 'start #t)))
              (transition state (quit-dataspace))]))
         (void)
         (sub (controller-event 'start #t)
              #:meta-level 1)                    ))
```

```
(spawn-dataspace (spawn-start-button-monitor)
                 (spawn-player)
                 (spawn-collision-detection)
                 ...)

(define (spawn-start-button-monitor)
  (spawn (lambda (evt state)
           (match-event evt
             [(message (at-meta
                        (controller-event 'start #t)))
              (transition state (quit-dataspace))]))
         (void)
         (sub (controller-event 'start #t)
              #:meta-level 1)                    ))
```

Dataspace lifetime not syntactically apparent

```
(spawn-dataspace (spawn-start-button-monitor)
                 (spawn-player)
                 (spawn-collision-detection)
                 ...)

(define (spawn-start-button-monitor)
  (spawn (lambda (evt state)
           (match-event evt
             [(message (at-meta
                        (controller-event 'start #t)))
              (transition state (quit-dataspace))]))
         (void)
         (sub (controller-event 'start #t)
              #:meta-level 1)                    ))
```

```
(spawn-dataspace (spawn-start-button-monitor)
                 (spawn-player)
                 (spawn-collision-detection)
                 ...)

(define (spawn-start-button-monitor)
  (spawn (lambda (evt state)
           (match-event evt
             [(message (at-meta
                        (controller-event 'start #t)))
              (transition state (quit-dataspace))]))
         (void)
         (sub (controller-event 'start #t)
              #:meta-level 1)))
```

2× repetition of pattern

```
(spawn-dataspace (spawn-start-button-monitor)
                 (spawn-player)
                 (spawn-collision-detection)
                 ...)

(define (spawn-start-button-monitor)
  (spawn (lambda (evt state)
           (match-event evt
             [(message (at-meta
                        (controller-event 'start #t)))
              (transition state (quit-dataspace))]))
         (void)
         (sub (controller-event 'start #t)
              #:meta-level 1)                    ))
```

```
(spawn-dataspace (spawn-start-button-monitor)
                 (spawn-player)
                 (spawn-collision-detection)
                 ...)

(define (spawn-start-button-monitor)
  (spawn (lambda (evt state)
           (match-event evt
             [(message (at-meta
                        (controller-event 'start #t)))
              (transition state (quit-dataspace))]))
         (void)
         (sub (controller-event 'start #t)
              #:meta-level 1)                    ))
```

2× repetition of metalevel, in two styles

```
(dataspace (spawn-player)
           (spawn-collision-detection)
           ...
           (until (message (controller-event 'start #t)
                           #:meta-level 1)))
```

dataspace termination near dataspace startup

```
(dataspace (spawn-player)
           (spawn-collision-detection)
           ...
           (until (message (controller-event 'start #t)
                           #:meta-level 1)))
```

```
(dataspace (spawn-player)
           (spawn-collision-detection)
           ...
           (until (message (controller-event 'start #t)
                           #:meta-level 1)))
```

subscription/message pattern written *once*

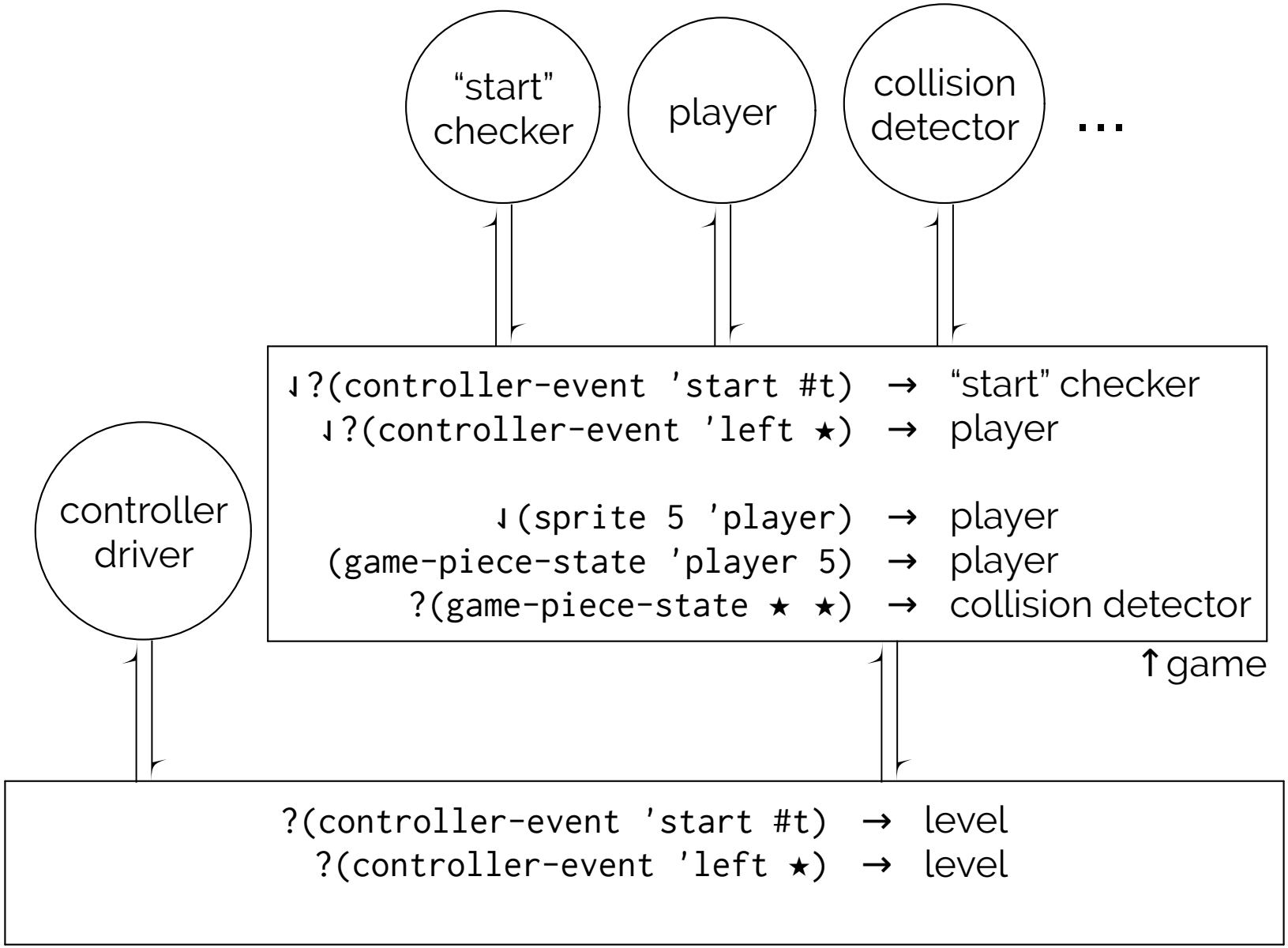
```
(dataspace (spawn-player)
           (spawn-collision-detection)
           ...
           (until (message (controller-event 'start #t)
                           #:meta-level 1)))
```

metalevel number written *once*, in one style

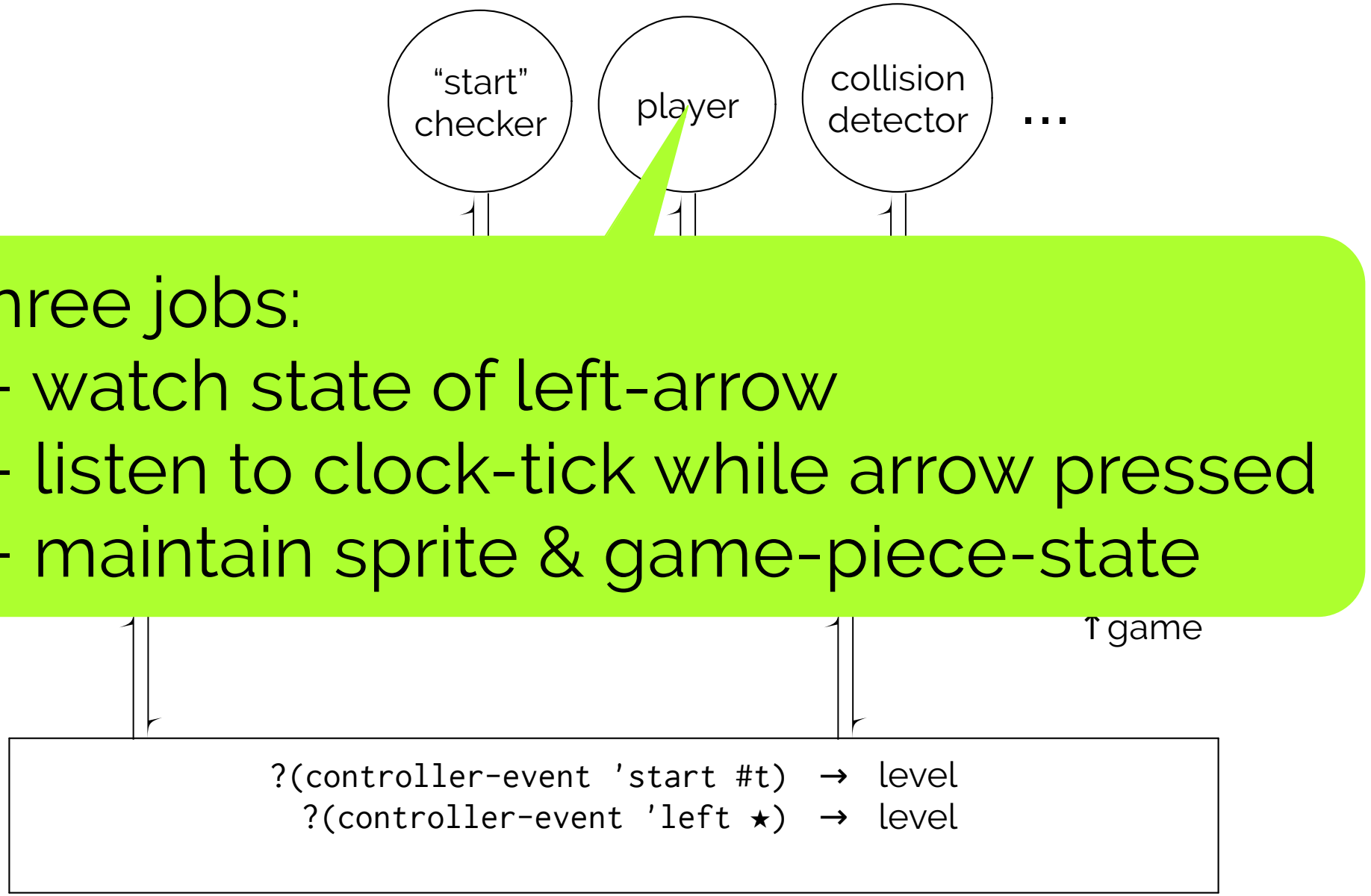
Syndicate DSL by example

- ✓ Mapping events to components
- • Managing conversational state
- • Monitoring changes in shared state

Managing conversational state

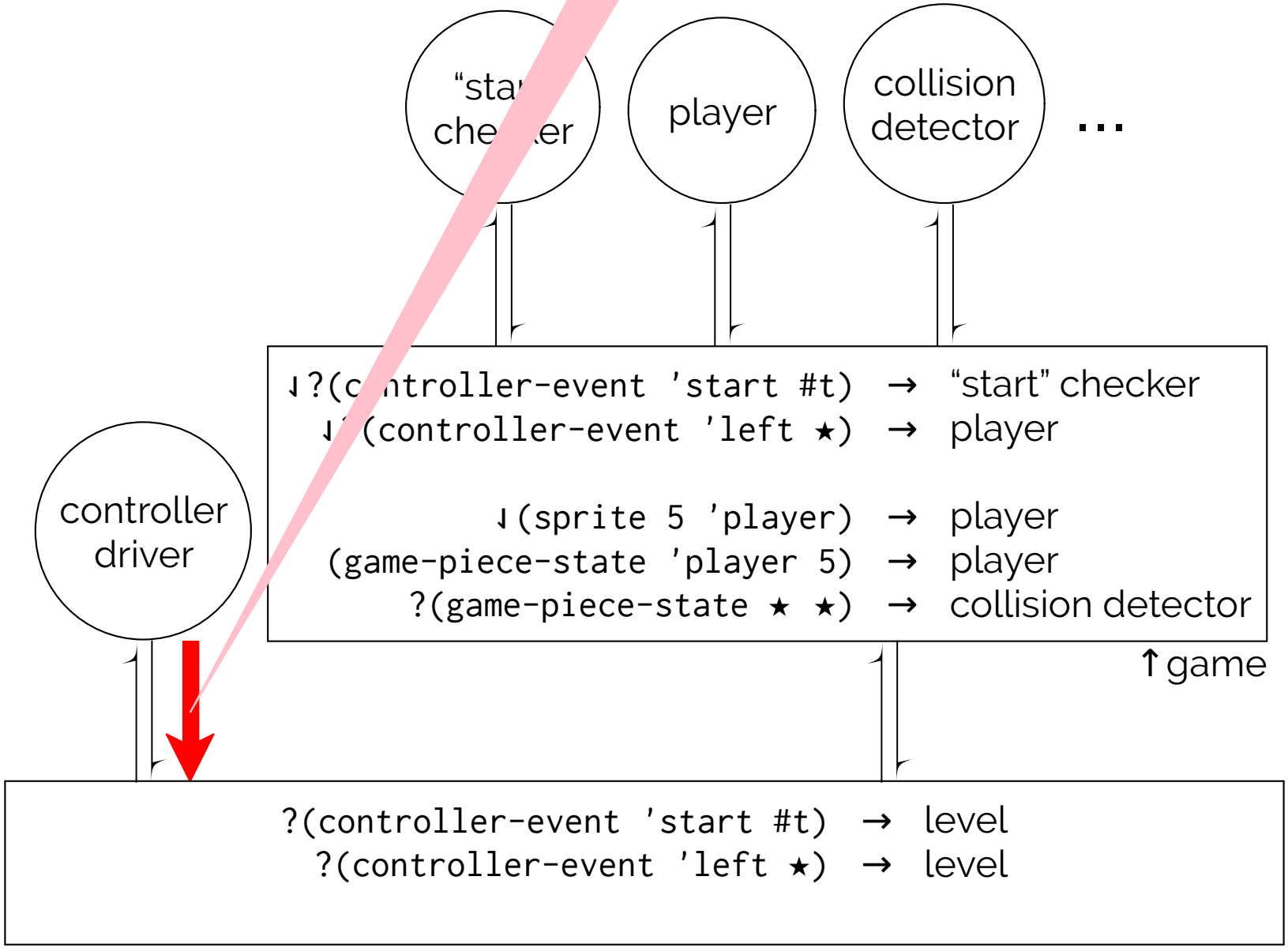


Managing conversational state

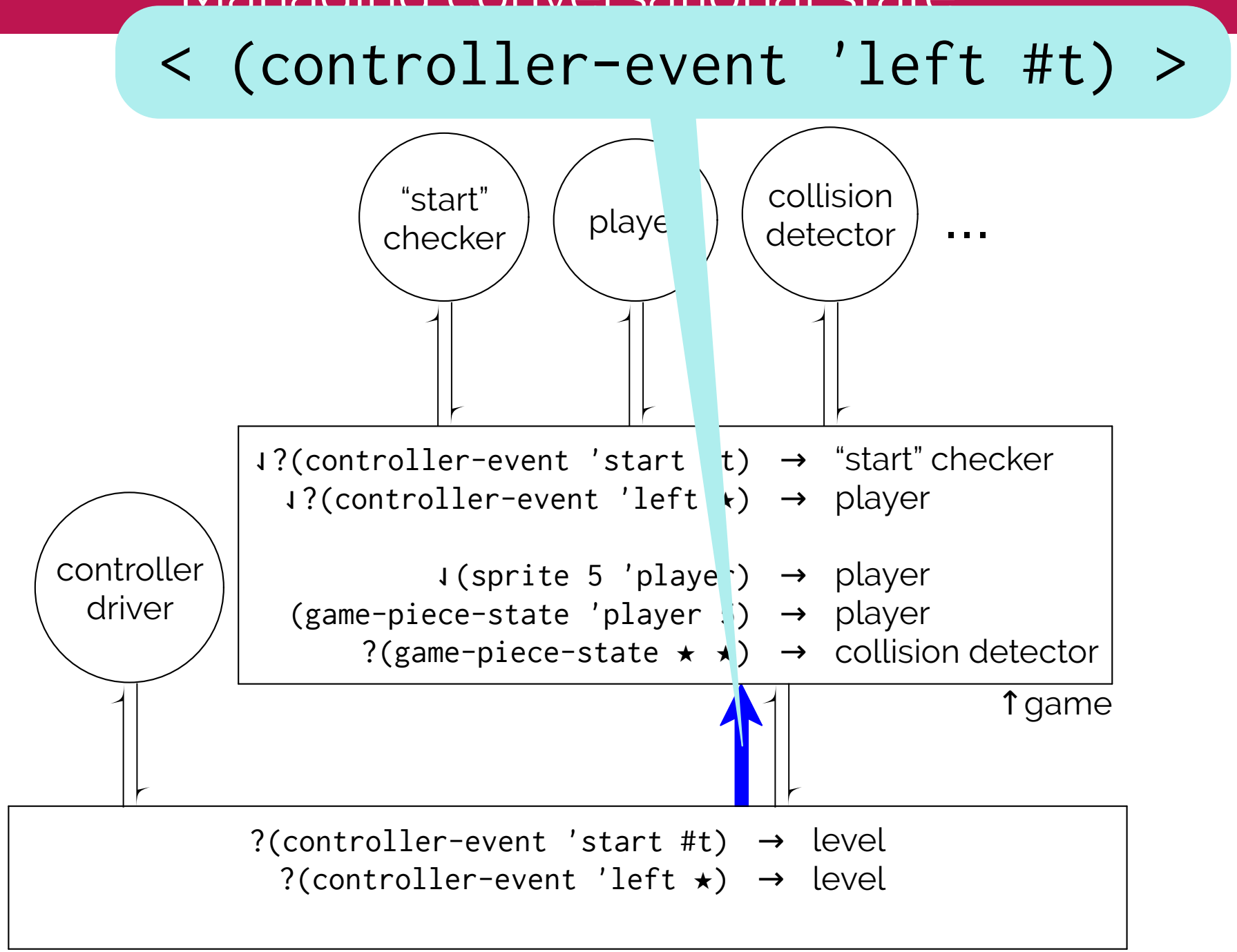


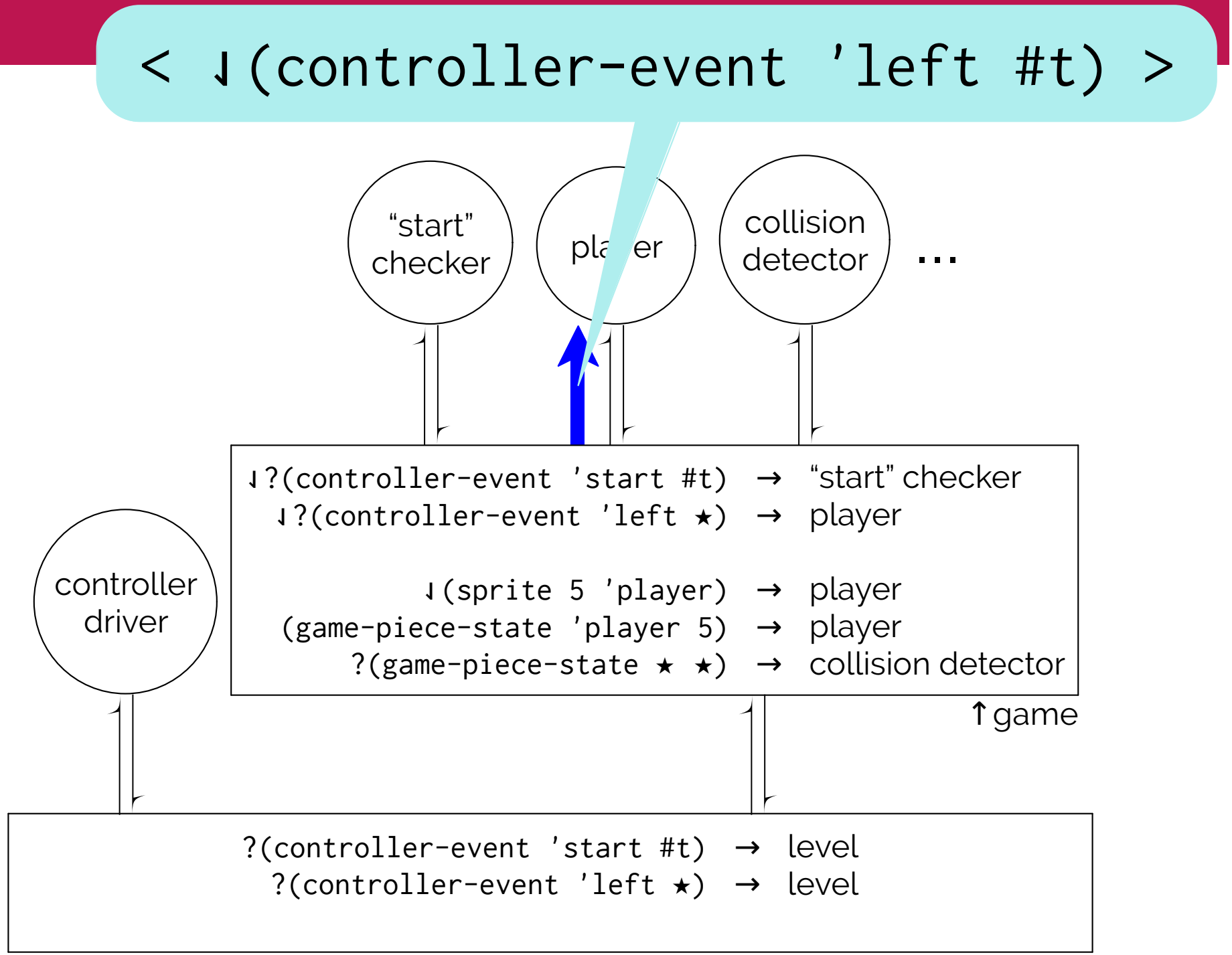
Managing conversational state

```
< (controller-event 'left #t) >
```

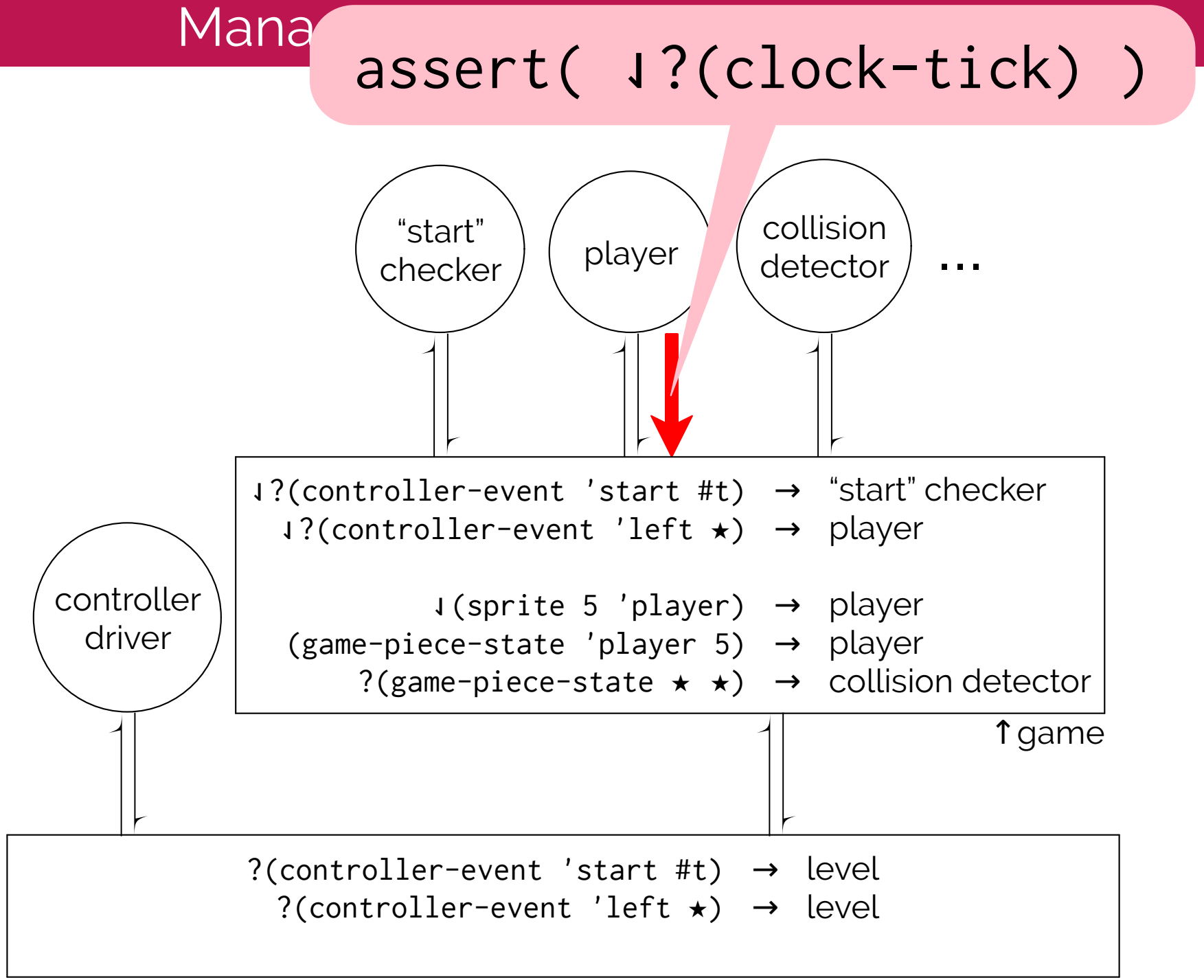


Managing conversational state

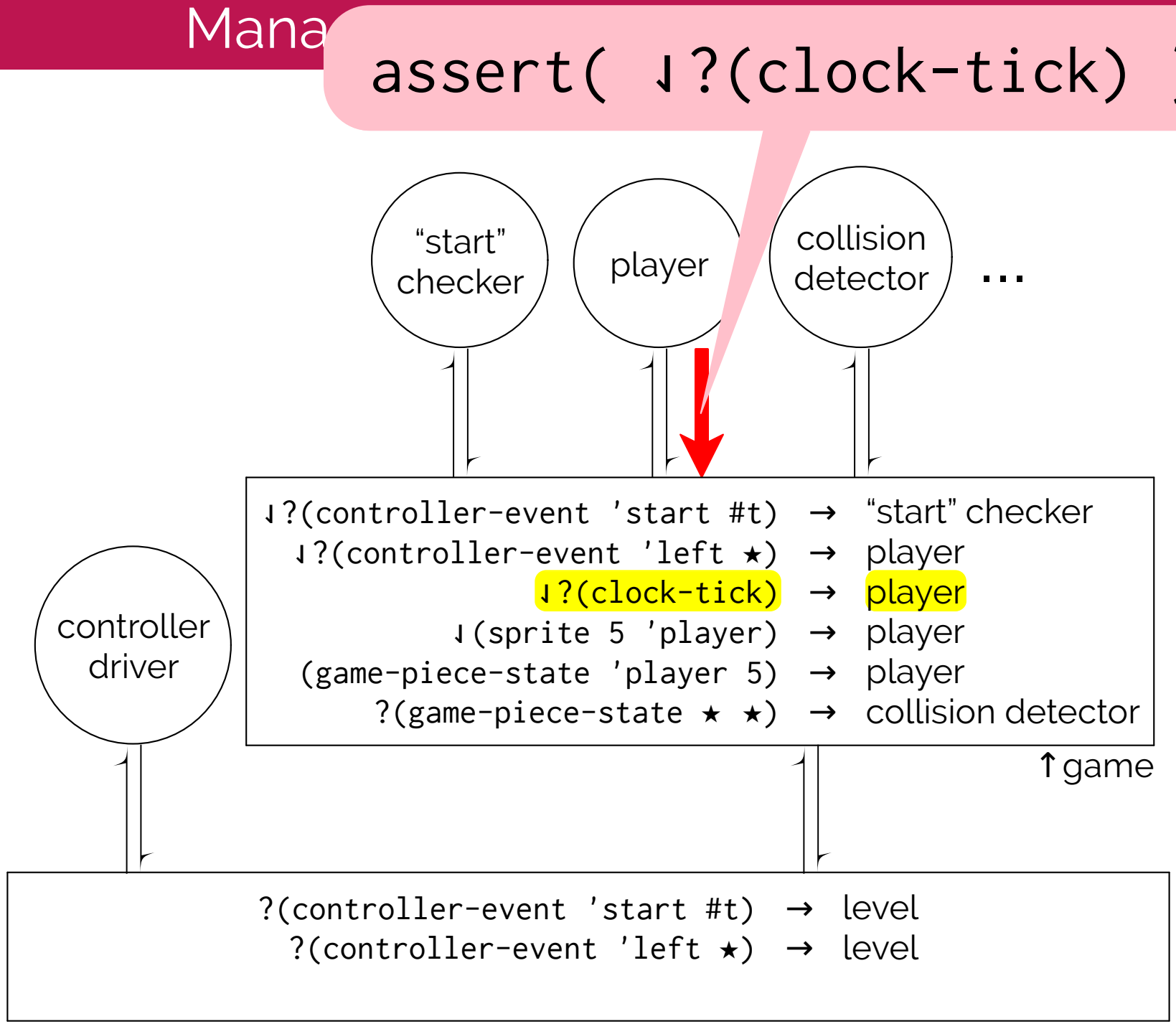



< ↓(controller-event 'left #t) >
"start" checker
player
collision detector
...
↓?(controller-event 'start #t) → "start" checker
↓?(controller-event 'left ★) → player
↓(sprite 5 'player) → player
(game-piece-state 'player 5) → player
?(game-piece-state ★ ★) → collision detector
↑game
controller driver
?(controller-event 'start #t) → level
?(controller-event 'left ★) → level

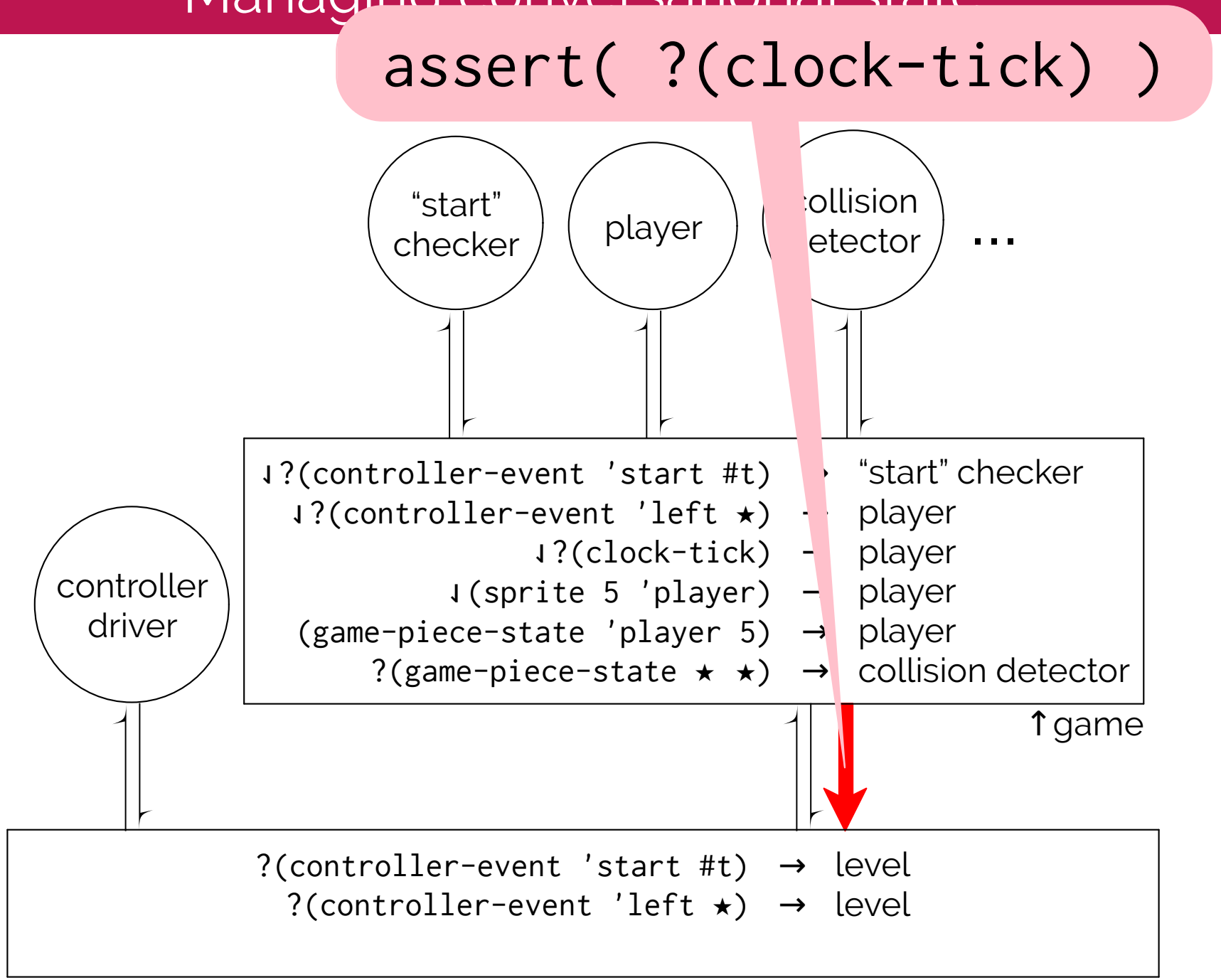
Mana


assert(↓?(clock-tick))
"start" checker
player
collision detector
...
↓?(controller-event 'start #t) → "start" checker
↓?(controller-event 'left ★) → player
↓(sprite 5 'player) → player
(game-piece-state 'player 5) → player
?(game-piece-state ★ ★) → collision detector
↑game
controller driver
?(controller-event 'start #t) → level
?(controller-event 'left ★) → level

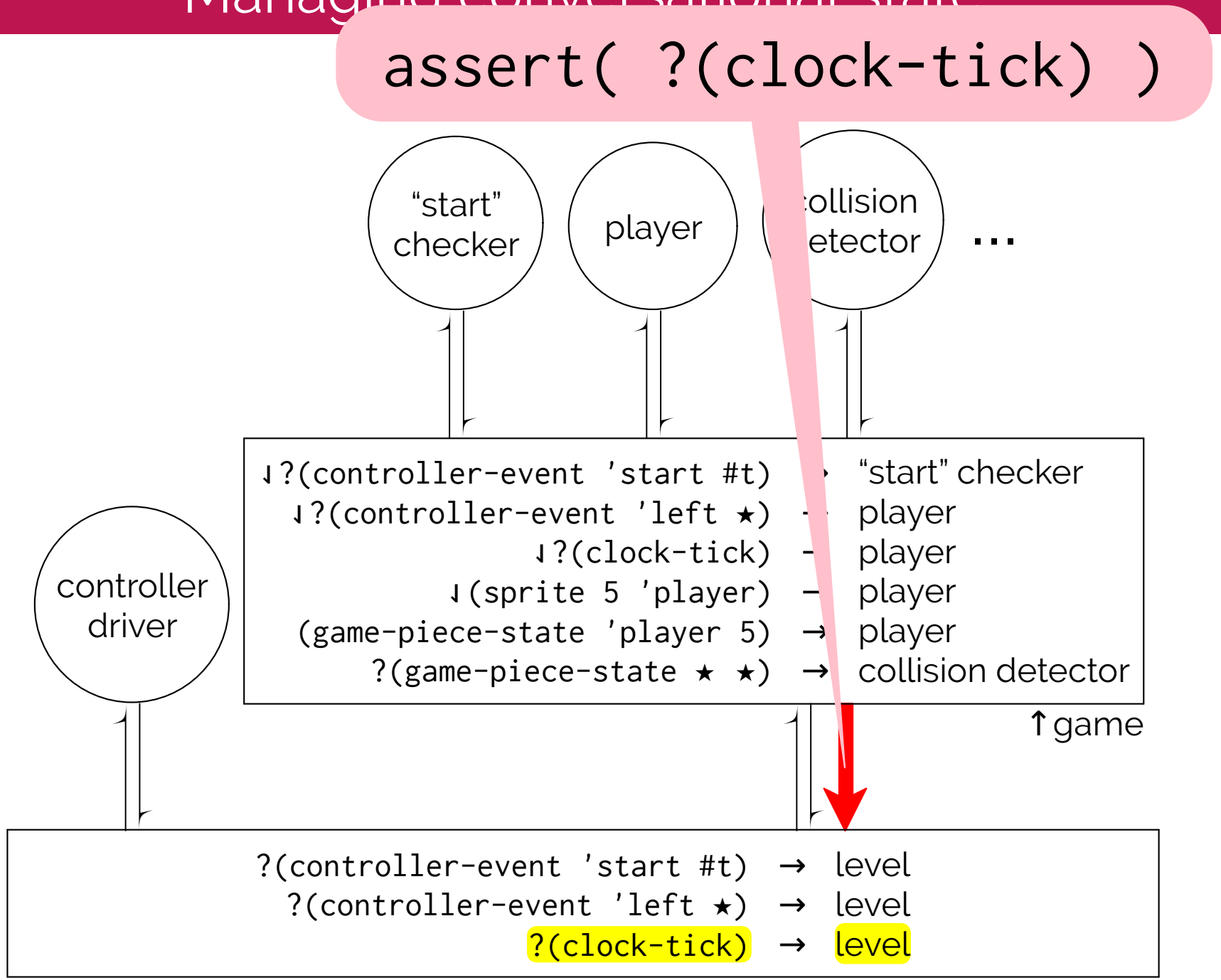
Mana


assert(↓?(clock-tick))
"start" checker
player
collision detector
...
↓?(controller-event 'start #t) → "start" checker
↓?(controller-event 'left ★) → player
↓?(clock-tick) → player
↓(sprite 5 'player) → player
(game-piece-state 'player 5) → player
?(game-piece-state ★ ★) → collision detector
↑game
controller driver
?(controller-event 'start #t) → level
?(controller-event 'left ★) → level

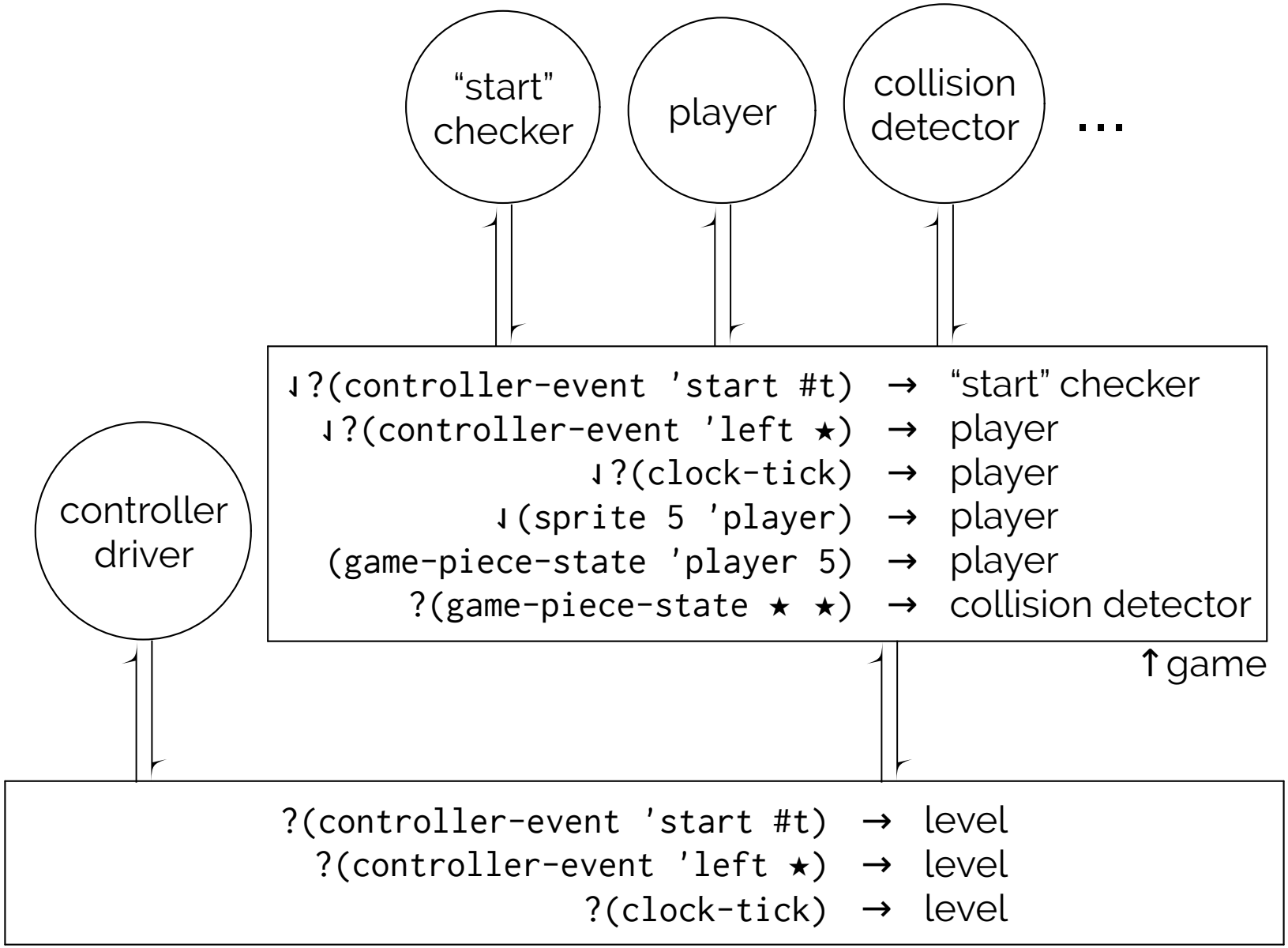
Managing conversational state



Managing conversational state

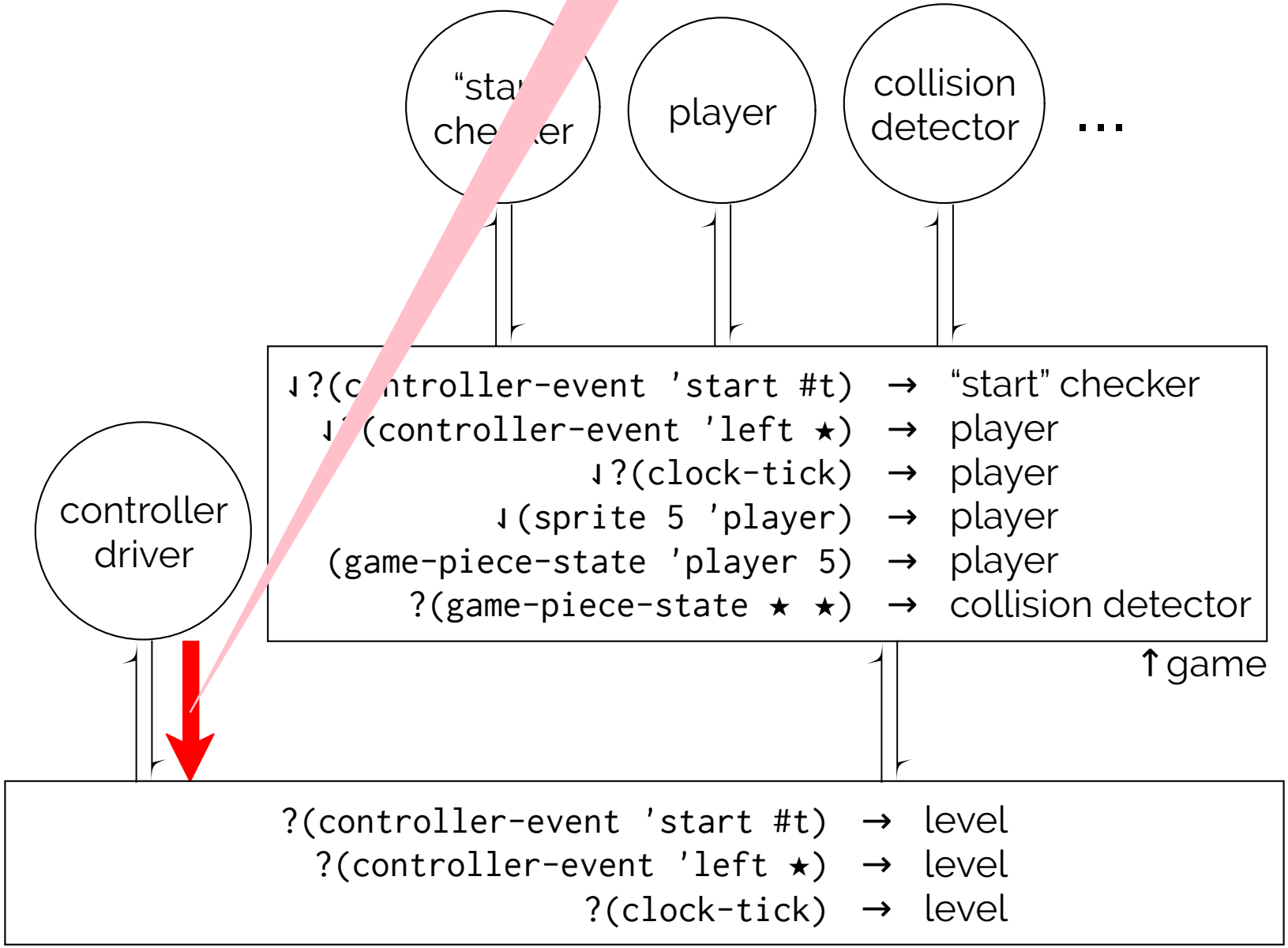


Managing conversational state



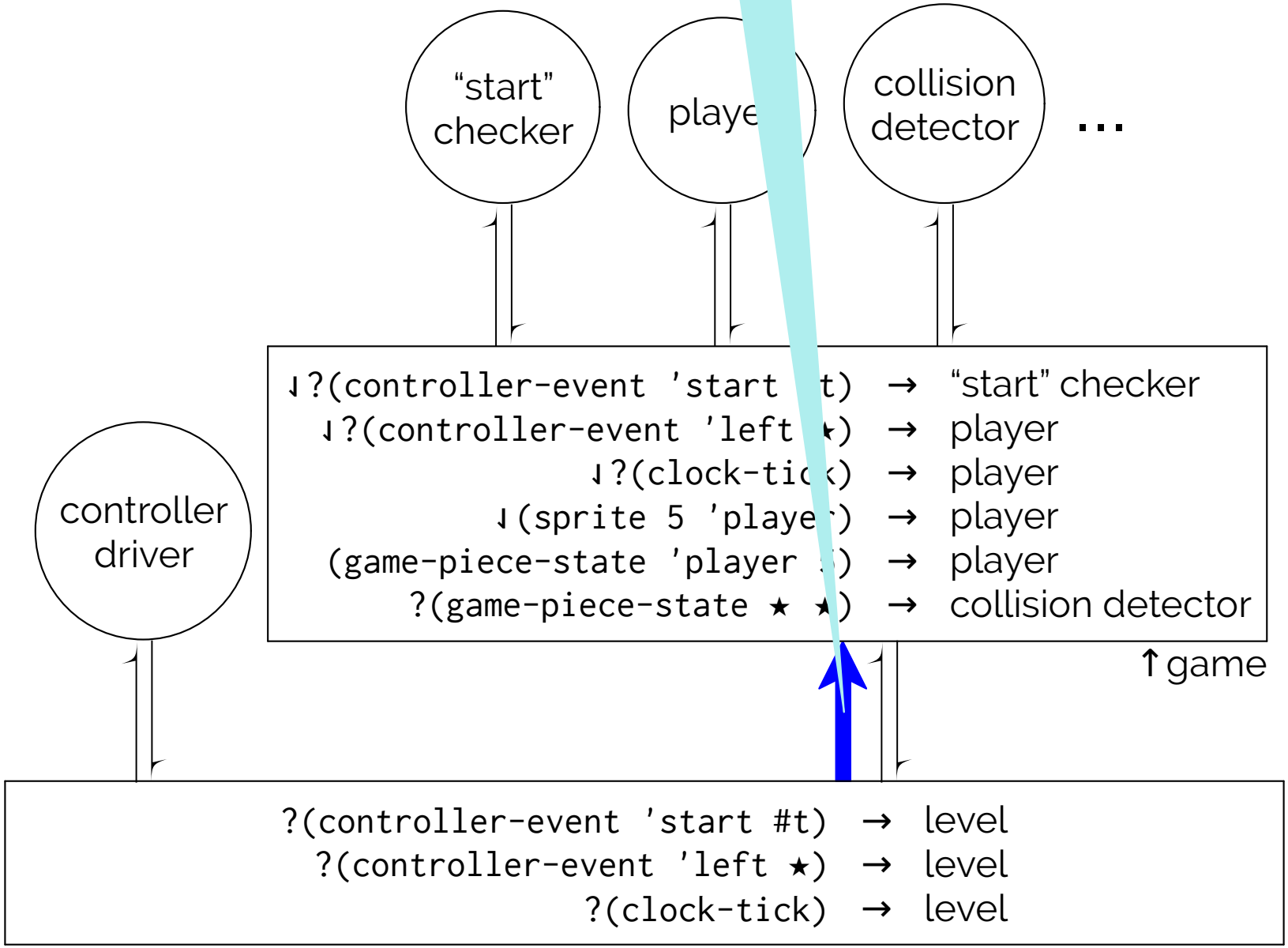
Managing conversational state

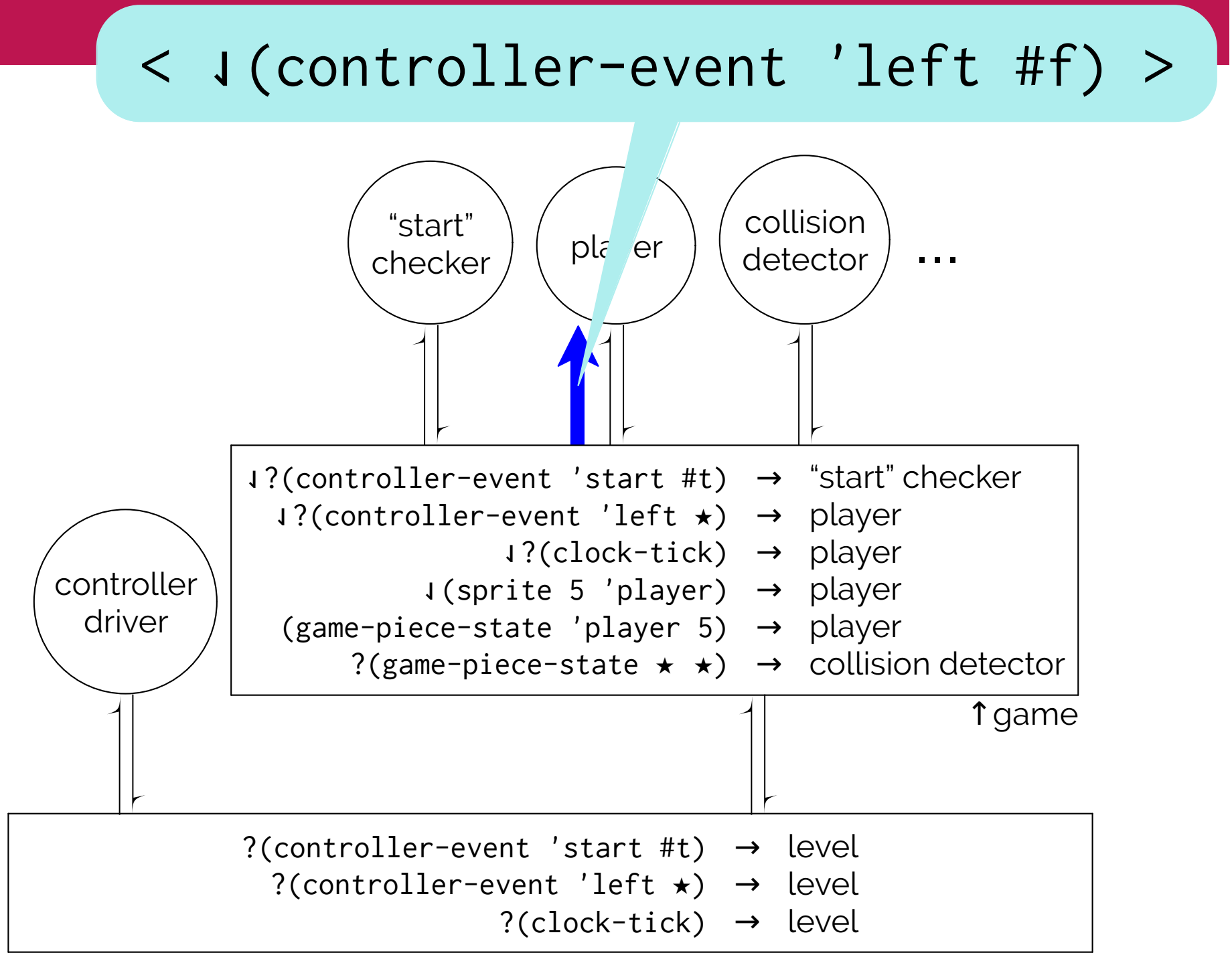
```
< (controller-event 'left #f) >
```



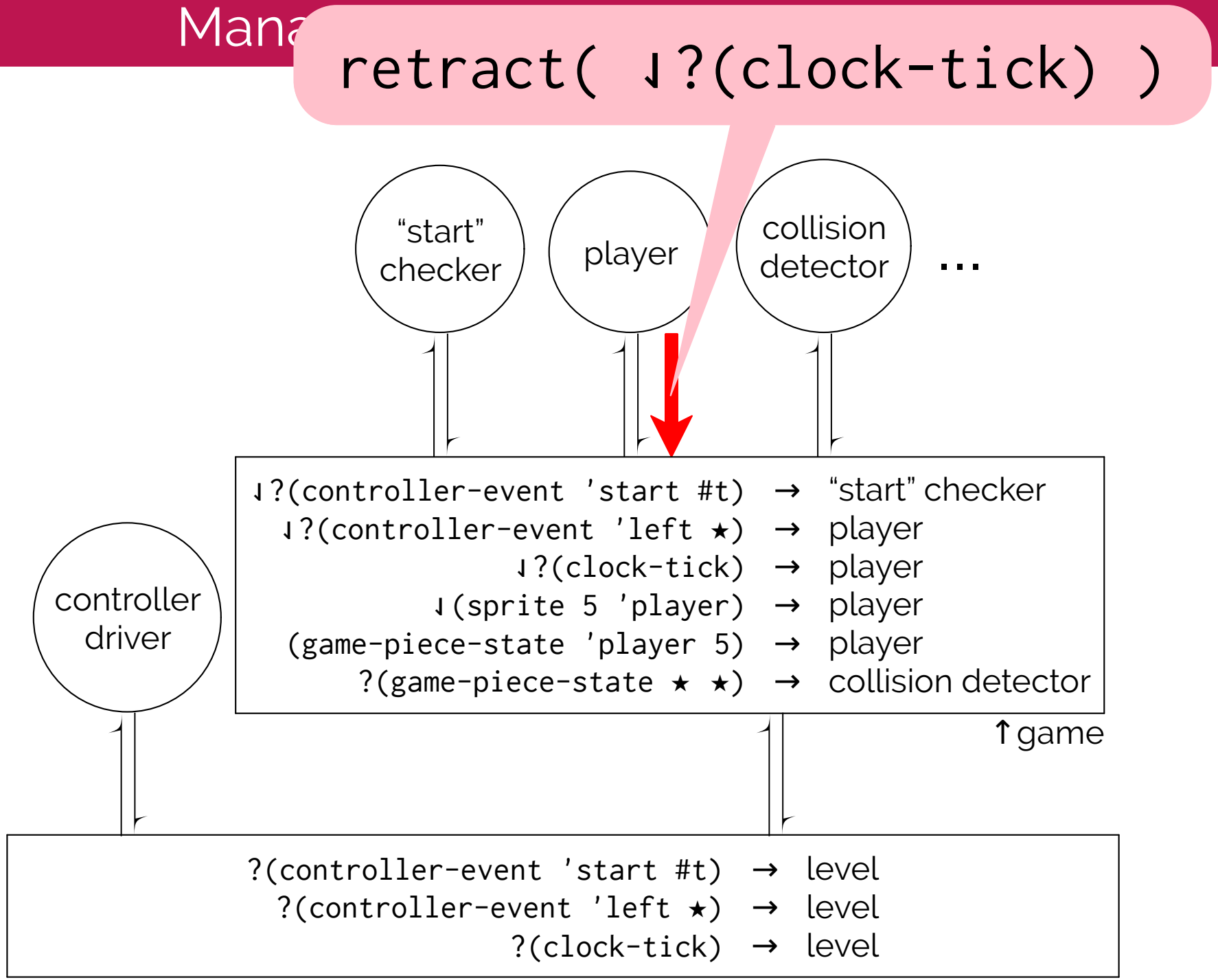
Managing conversational state

```
< (controller-event 'left #f) >
```

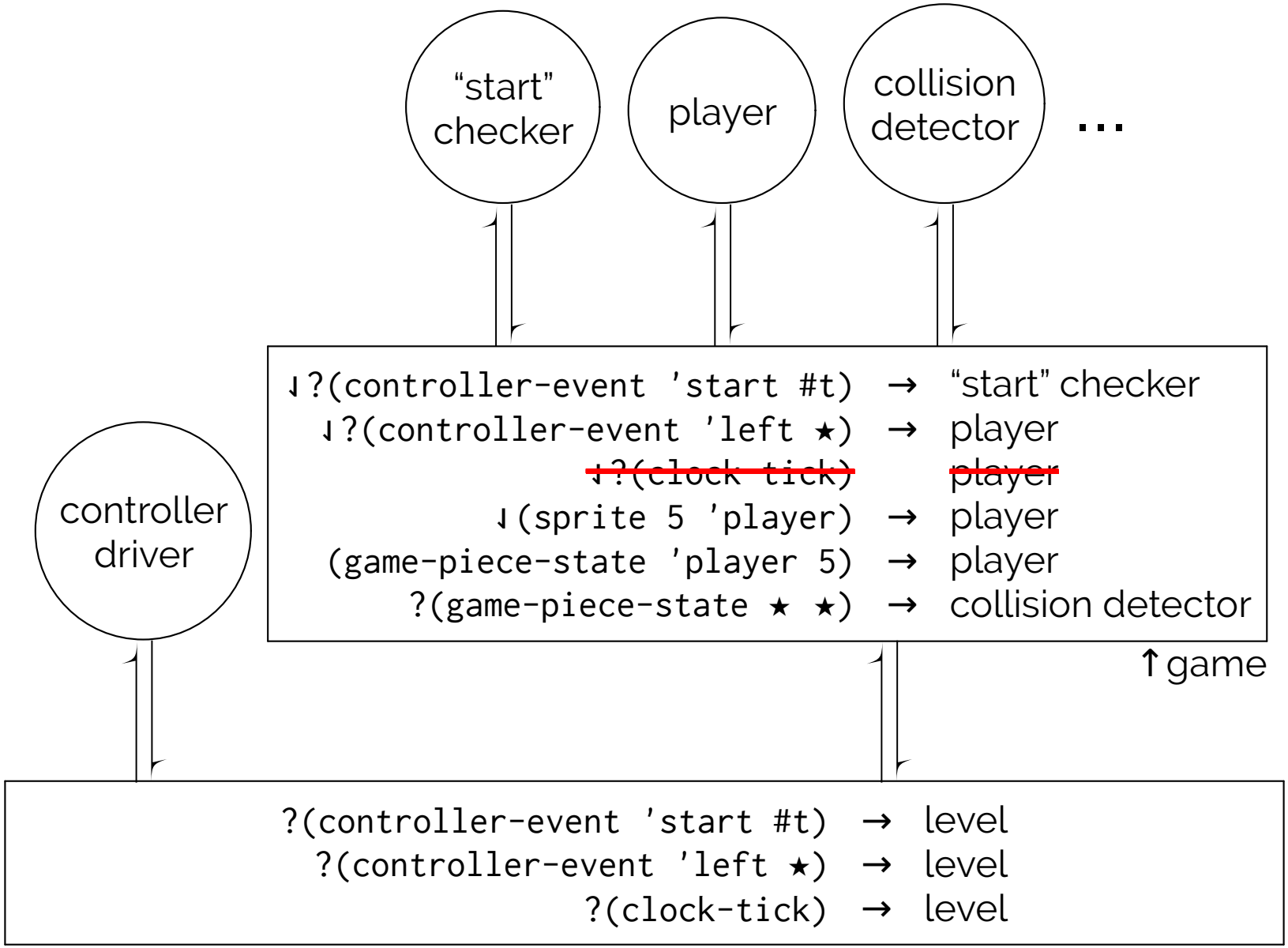



< ↓(controller-event 'left #f) >
"start" checker
player
collision detector
...
↓?(controller-event 'start #t) → "start" checker
↓?(controller-event 'left ★) → player
↓?(clock-tick) → player
↓(sprite 5 'player) → player
(game-piece-state 'player 5) → player
?(game-piece-state ★ ★) → collision detector
↑game
controller driver
?(controller-event 'start #t) → level
?(controller-event 'left ★) → level
?(clock-tick) → level

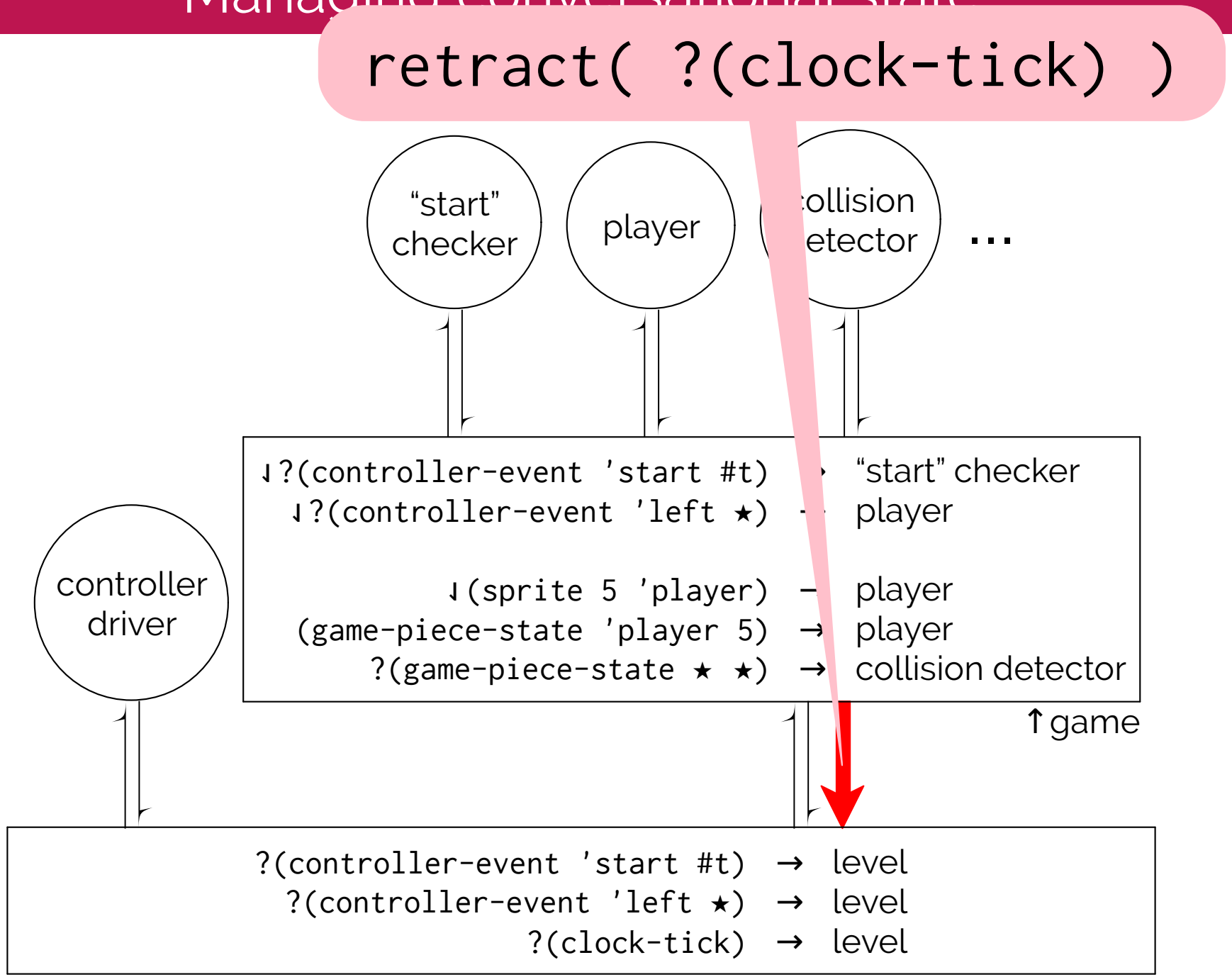
Mana


retract(↓?(clock-tick))
"start" checker
player
collision detector
...
↓?(controller-event 'start #t) → "start" checker
↓?(controller-event 'left ★) → player
↓?(clock-tick) → player
↓(sprite 5 'player) → player
(game-piece-state 'player 5) → player
?(game-piece-state ★ ★) → collision detector
↑game
controller driver
?(controller-event 'start #t) → level
?(controller-event 'left ★) → level
?(clock-tick) → level

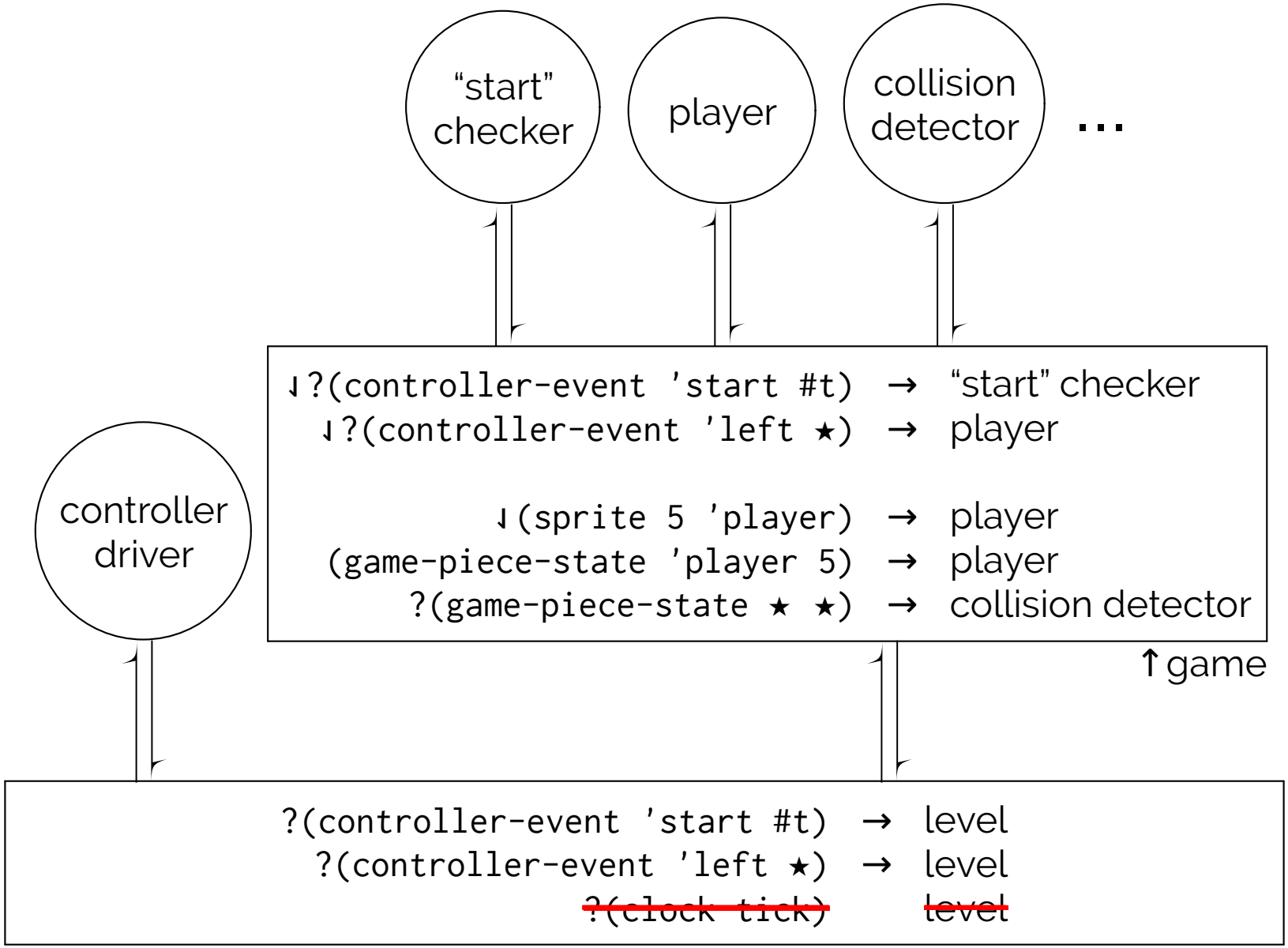
Managing conversational state



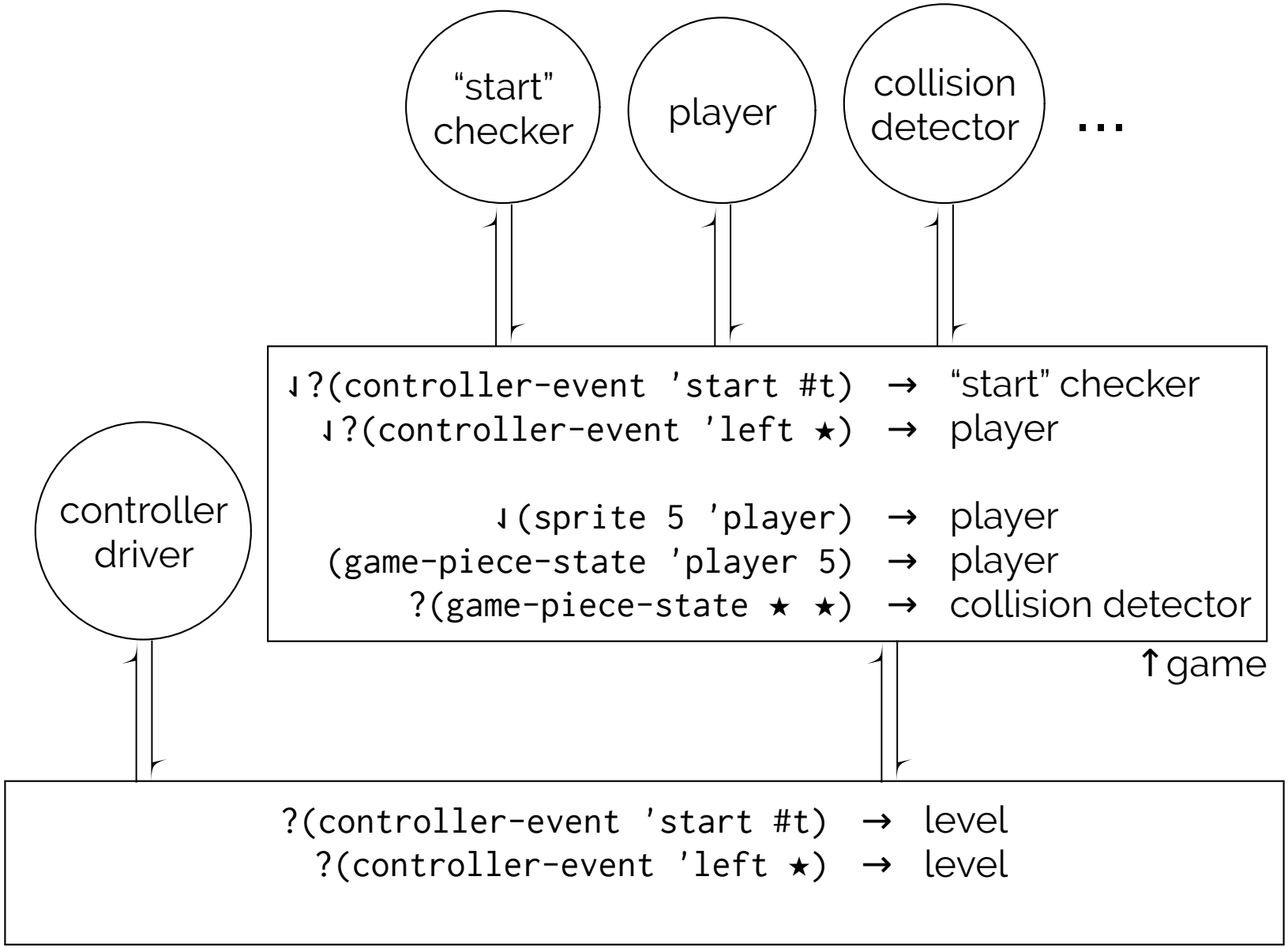
Managing conversational state



Managing conversational state



Managing conversational state



```
(struct player-state (position left-down?))

(define (spawn-player)
  (define initial-pos 5)
  (define initial-state (player-state initial-pos #f))
  (spawn (lambda (evt state)
           (match-event evt
            [(message (at-meta (controller-event 'left pressed?)))
             (transition (struct-copy player-state state
                                      [left-down? pressed?])
                         '())]
            [(message (at-meta (clock-tick)))
             (define new-state
               (if (player-state-left-down? state)
                   (struct-copy player-state state
                                [position (- (player-state-position state) 1)])
                   state))
             (define new-pos (player-state-position new-state))
             (transition new-state
                         (patch-seq (retract (sprite ? ?) #:meta-level 1)
                                    (assert (sprite new-pos 'player) #:meta-level 1)
                                    (retract (game-piece-state ? ?))
                                    (assert (game-piece-state 'player new-pos))))]
            )
           initial-state
           (patch-seq (sub (controller-event 'left ?) #:meta-level 1)
                      (sub (clock-tick) #:meta-level 1)
                      (assert (sprite initial-pos 'player) #:meta-level 1)
                      (assert (game-piece-state 'player initial-pos))))))
```

```
(struct player-state (position left-down?))

(define (spawn-player)
  (define initial-pos 5)
  (define initial-state (player-state initial-pos #f))
  (spawn (lambda (evt state)
           (match-event evt
            [(message (at-meta (controller-event 'left pressed?)))
             (transition (struct-copy player-state state
                                      [left-down? pressed?])
                         '())]
            [(message (at-meta (clock-tick)))
             (define new-state
               (if (player-state-left-down? state)
                   (struct-copy player-state state
                                [position (- (player-state-position state) 1)])
                   state))
             (define new-pos (player-state-position new-state))
             (transition new-state
                         (patch-seq (retract (sprite ? ?) #:meta-level 1)
                                    (assert (sprite new-pos 'player) #:meta-level 1)
                                    (retract (game-piece-state ? ?))
                                    (assert (game-piece-state 'player new-pos))))]
            )
           initial-state
           (patch-seq (sub (controller-event 'left ?) #:meta-level 1)
                      (sub (clock-tick) #:meta-level 1)
                      (assert (sprite initial-pos 'player) #:meta-level 1)
                      (assert (game-piece-state 'player initial-pos))))))
```

```
(struct player-state (position left-down?))

(define (spawn-player)
  (define initial-pos 5)
  (define initial-state (player-state initial-pos #f))
  (spawn (lambda (evt state)
           (match-event evt
            [(message (at-meta (controller-event 'left pressed?)))
             (transition (struct-copy player-state state
                                      [left-down? pressed?])
                         '())]
            [(message (at-meta (clock-tick)))
             (define new-state
               (if (player-state-left-down? state)
                   (struct-copy player-state state
                                [position (- (player-state-position state) 1)])
                   state))
             (define new-pos (player-state-position new-state))
             (transition new-state
                         (patch-seq (retract (sprite ? ?) #:meta-level 1)
                                    (assert (sprite new-pos 'player) #:meta-level 1)
                                    (retract (game-piece-state ? ?))
                                    (assert (game-piece-state 'player new-pos))))]))
         initial-state
         (patch-seq (sub (controller-event 'left ?) #:meta-level 1)
                    (sub (clock-tick) #:meta-level 1)
                    (assert (sprite initial-pos 'player) #:meta-level 1)
                    (assert (game-piece-state 'player initial-pos)))))
```

```
(struct player-state (position left-down?))

(define (spawn-player)
  (define initial-pos 5)
  (define initial-state (player-state initial-pos #f))
  (spawn (lambda (evt state)
           (match-event evt
            [(message (at-meta (controller-event 'left pressed?)))
             (transition (struct-copy player-state state
                                      [left-down? pressed?])
                         '())]
            [(message (at-meta (clock-tick)))
             (define new-state
               (if (player-state-left-down? state)
                   (struct-copy player-state state
                                [position (- (player-state-position state) 1)])
                   state))
             (define new-pos (player-state-position new-state))
             (transition new-state
                         (patch-seq (retract (sprite ? ?) #:meta-level 1)
                                    (assert (sprite new-pos 'player) #:meta-level 1)
                                    (retract (game-piece-state ? ?))
                                    (assert (game-piece-state 'player new-pos))))]))
         initial-state
         (patch-seq (sub (controller-event 'left ?) #:meta-level 1)
                    (sub (clock-tick) #:meta-level 1)
                    (assert (sprite initial-pos 'player) #:meta-level 1)
                    (assert (game-piece-state 'player initial-pos)))))
```

```
(struct player-state (position left-down?))

(define (spawn-player)
  (define initial-pos 5)
  (define initial-state (player-state initial-pos #f))
  (spawn (lambda (evt state)
           (match-event evt
            [(message (at-meta (controller-event 'left pressed?)))
             (transition (struct-copy player-state state
                                      [left-down? pressed?])
                         '())]
            [(message (at-meta (clock-tick)))
             (define new-state
               (if (player-state-left-down? state)
                   (struct-copy player-state state
                                [position (- (player-state-position state) 1)])
                   state))
             (define new-pos (player-state-position new-state))
             (transition new-state
                         (patch-seq (retract (sprite ? ?) #:meta-level 1)
                                    (assert (sprite new-pos 'player) #:meta-level 1)
                                    (retract (game-piece-state ? ?))
                                    (assert (game-piece-state 'player new-pos))))]))
         initial-state
         (patch-seq (sub (controller-event 'left ?) #:meta-level 1)
                    (sub (clock-tick) #:meta-level 1)
                    (assert (sprite initial-pos 'player) #:meta-level 1)
                    (assert (game-piece-state 'player initial-pos)))))
```

```
(struct player-state (position left-down?))

(define (spawn-player)
  (define initial-pos 5)
  (define initial-state (player-state initial-pos #f))
  (spawn (lambda (evt state)
           (match-event evt
            [(message (at-meta (controller-event 'left pressed?)))
             (transition (struct-copy player-state state
                                      [left-down? pressed?])
                         '())]
            [(message (at-meta (clock-tick)))
             (define new-state
               (if (player-state-left-down? state)
                   (struct-copy player-state state
                                [position (- (player-state-position state) 1)])
                   state))
             (define new-pos (player-state-position new-state))
             (transition new-state
                         (patch-seq (retract (sprite ? ?) #:meta-level 1)
                                    (assert (sprite new-pos 'player) #:meta-level 1)
                                    (retract (game-piece-state ? ?))
                                    (assert (game-piece-state 'player new-pos))))]
            )
           initial-state
           (patch-seq (sub (controller-event 'left ?) #:meta-level 1)
                      (sub (clock-tick) #:meta-level 1)
                      (assert (sprite initial-pos 'player) #:meta-level 1)
                      (assert (game-piece-state 'player initial-pos))))))
```

```
(struct player-state (position left-down?))

(define (spawn-player)
  (define initial-pos 5)
  (define initial-state (player-state initial-pos #f))
  (spawn (lambda (evt state)
           (match-event evt
            [(message (at-meta (controller-event 'left pressed?)))
             (transition (struct-copy player-state state
                                      [left-down? pressed?])
                         '())]
            [(message (at-meta (clock-tick)))
             (define new-state
               (if (player-state-left-down? state)
                   (struct-copy player-state state
                                [position (- (player-state-position state) 1)])
                   state))
             (define new-pos (player-state-position new-state))
             (transition new-state
                         (patch-seq (retract (sprite ? ?) #:meta-level 1)
                                    (assert (sprite new-pos 'player) #:meta-level 1)
                                    (retract (game-piece-state ? ?))
                                    (assert (game-piece-state 'player new-pos))))]))
         initial-state
         (patch-seq (sub (controller-event 'left ?) #:meta-level 1)
                    (sub (clock-tick) #:meta-level 1)
                    (assert (sprite initial-pos 'player) #:meta-level 1)
                    (assert (game-piece-state 'player initial-pos)))))
```

```
(struct player-state (position left-down?))

(define (spawn-player)
  (define initial-pos 5)
  (define initial-state (player-state initial-pos #f))
  (spawn (lambda (evt state)
           (match-event evt
            [(message (at-meta (controller-event 'left pressed?)))
             (transition (struct-copy player-state state
                                      [left-down? pressed?])
                         '())]
            [(message (at-meta (clock-tick)))
             (define new-state
               (if (player-state-left-down? state)
                   (struct-copy player-state state
                                [position (- (player-state-position state) 1)])
                   state))
             (define new-pos (player-state-position new-state))
             (transition new-state
                         (patch-seq (retract (sprite ? ?) #:meta-level 1)
                                    (assert (sprite new-pos 'player) #:meta-level 1)
                                    (retract (game-piece-state ? ?))
                                    (assert (game-piece-state 'player new-pos)))
                         )]
           initial-state
           (patch-seq (sub (controller-event 'left ?) #:meta-level 1)
                      (sub (clock-tick) #:meta-level 1)
                      (assert (sprite initial-pos 'player) #:meta-level 1)
                      (assert (game-piece-state 'player initial-pos))))))
```

```
(define (spawn-player)
  (define move-left (gensym))
  (actor (forever #:collect [(position 5)]
           (assert (sprite position 'player) #:meta-level 1)
           (assert (game-piece-state 'player position))
           (on (message (controller-event 'left #t)
                        #:meta-level 1)
               (until (message (controller-event 'left #f)
                               #:meta-level 1)
                 (on (message (clock-tick) #:meta-level 1)
                     (send! move-left))))
           (on (message move-left)
               (- position 1))    )))
```

```
(define (spawn-player)
  (define move-left (gensym))
  (actor (forever #:collect [(position 5)]
           (assert (sprite position 'player) #:meta-level 1)
           (assert (game-piece-state 'player position))
           (on (message (controller-event 'left #t)
                        #:meta-level 1)
               (until (message (controller-event 'left #f)
                               #:meta-level 1)
                 (on (message (clock-tick) #:meta-level 1)
                     (send! move-left))))
           (on (message move-left)
               (- position 1))    )))
```

```
(define (spawn-player)
  (define move-left (gensym))
  (actor (forever #:collect [(position 5)]
           (assert (sprite position 'player) #:meta-level 1)
           (assert (game-piece-state 'player position))
           (on (message (controller-event 'left #t)
                        #:meta-level 1)
               (until (message (controller-event 'left #f)
                               #:meta-level 1)
                 (on (message (clock-tick) #:meta-level 1)
                     (send! move-left))))
           (on (message move-left)
               (- position 1))    )))
```

Substate continues to apply until termination event triggered

```
(define (spawn-player)
  (define move-left (gensym))
  (actor (forever #:collect [(position 5)]
           (assert (sprite position 'player) #:meta-level 1)
           (assert (game-piece-state 'player position))
           (on (message (controller-event 'left #t)
                        #:meta-level 1)
               (until (message (controller-event 'left #f)
                               #:meta-level 1)
                 (on (message (clock-tick) #:meta-level 1)
                     (send! move-left))))
           (on (message move-left)
               (- position 1))    )))
```

Substate continues to apply until termination event triggered

```
(define (spawn-player)
  (define move-left (gensym))
  (actor (forever #:collect [(position 5)]
           (assert (sprite position 'player) #:meta-level 1)
           (assert (game-piece-state 'player position))
           (on (message (controller-event 'left #t)
                        #:meta-level 1)
               (until (message (controller-event 'left #f)
                               #:meta-level 1)
                 (on (message (clock-tick) #:meta-level 1)
                     (send! move-left))))
           (on (message move-left)
               (- position 1))    )))
```

Substate continues to apply until termination event triggered

```
(define (spawn-player)
  (define move-left (gensym))
  (actor (forever #:collect [(position 5)]
           (assert (sprite position 'player) #:meta-level 1)
           (assert (game-piece-state 'player position))
           (on (message (controller-event 'left #t)
                        #:meta-level 1)
               (until (message (controller-event 'left #f)
                               #:meta-level 1)
                 (on (message (clock-tick) #:meta-level 1)
                     (send! move-left))))
           (on (message move-left)
               (- position 1))    )))
```

Substate continues to apply until termination event triggered

```
(define (spawn-player)
  (define move-left (gensym))
  (actor (forever #:collect [(position 5)]
           (assert (sprite position 'player) #:meta-level 1)
           (assert (game-piece-state 'player position))
           (on (message (controller-event 'left #t)
                        #:meta-level 1)
               (until (message (controller-event 'left #f)
                               #:meta-level 1)
                 (on (message (clock-tick) #:meta-level 1)
                     (send! move-left))))
           (on (message move-left)
               (- position 1)))))
```

```
(define (spawn-player)
  (define move-left (gensym))
  (actor (forever #:collect [(position 5)]
           (assert (sprite position 'player) #:meta-level 1)
           (assert (game-piece-state 'player position))
           (on (message (controller-event 'left #t)
                        #:meta-level 1)
               (until (message (controller-event 'left #f)
                               #:meta-level 1)
                 (on (message (clock-tick) #:meta-level 1)
                     (send! move-left))))
           (on (message move-left)
               (- position 1))    )))
```

```
(define (spawn-player)
  (define move-left (gensym))
  (actor (forever #:collect [(position 5)]
           (assert (sprite position 'player) #:meta-level 1)
           (assert (game-piece-state 'player position))
           (on (message (controller-event 'left #t)
                        #:meta-level 1)
               (until (message (controller-event 'left #f)
                               #:meta-level 1)
                 (on (message (clock-tick) #:meta-level 1)
                     (send! move-left))))
           (on (message move-left)
               (- position 1))    )))
```

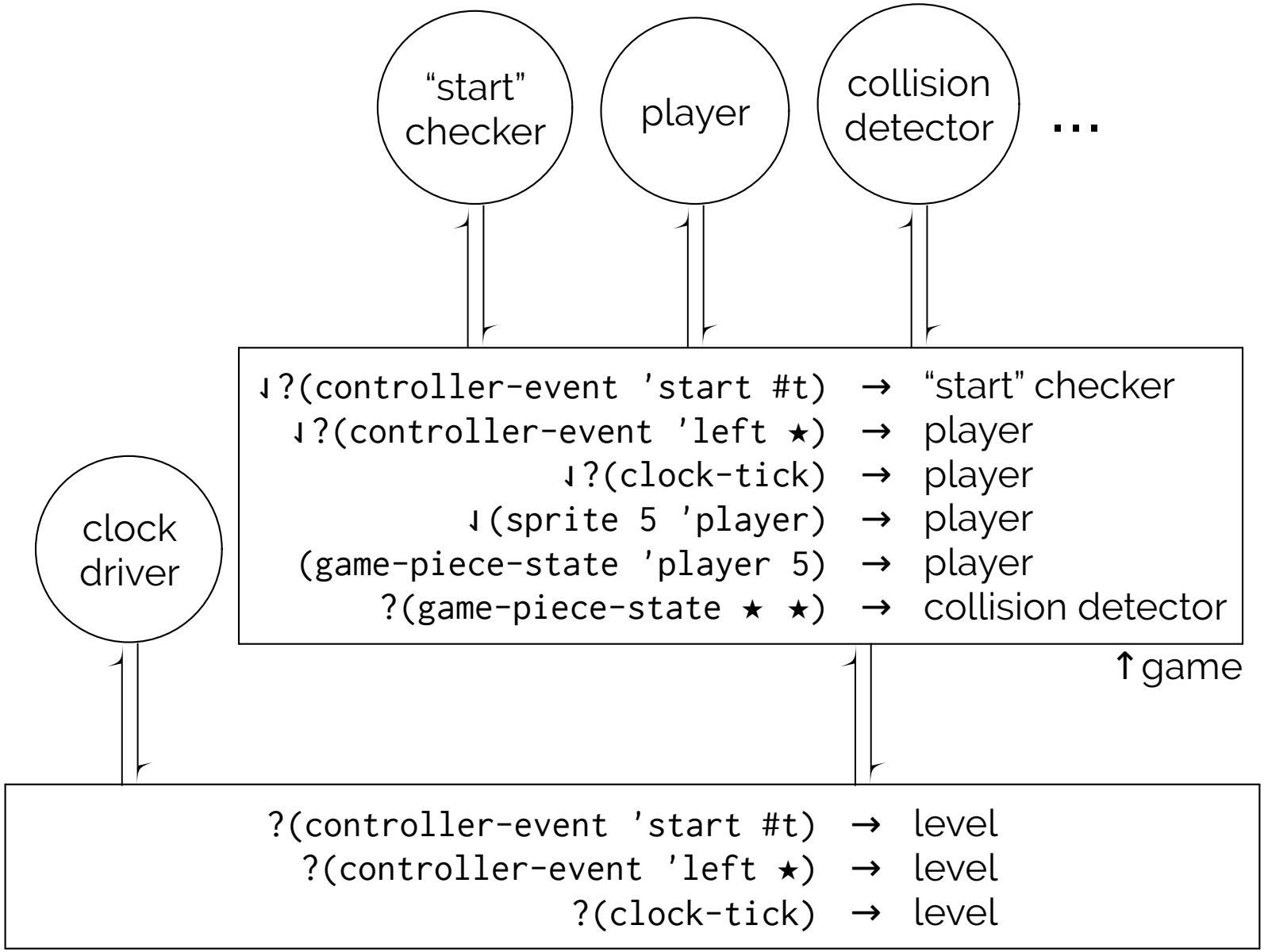
```
(define (spawn-player)
  (define position 5)
  (actor (forever
           (assert (sprite position 'player) #:meta-level 1)
           (assert (game-piece-state 'player position))
           (on (message (controller-event 'left #t)
                        #:meta-level 1)
               (until (message (controller-event 'left #f)
                               #:meta-level 1)
                 (on (message (clock-tick) #:meta-level 1)
                     (set! position (- position 1)))))))))
```

```
(define (spawn-player)
  (define position 5)
  (actor (forever
           (assert (sprite position 'player) #:meta-level 1)
           (assert (game-piece-state 'player position))
           (on (message (controller-event 'left #t)
                        #:meta-level 1)
               (until (message (controller-event 'left #f)
                               #:meta-level 1)
                 (on (message (clock-tick) #:meta-level 1)
                     (set! position (- position 1))))))))
```

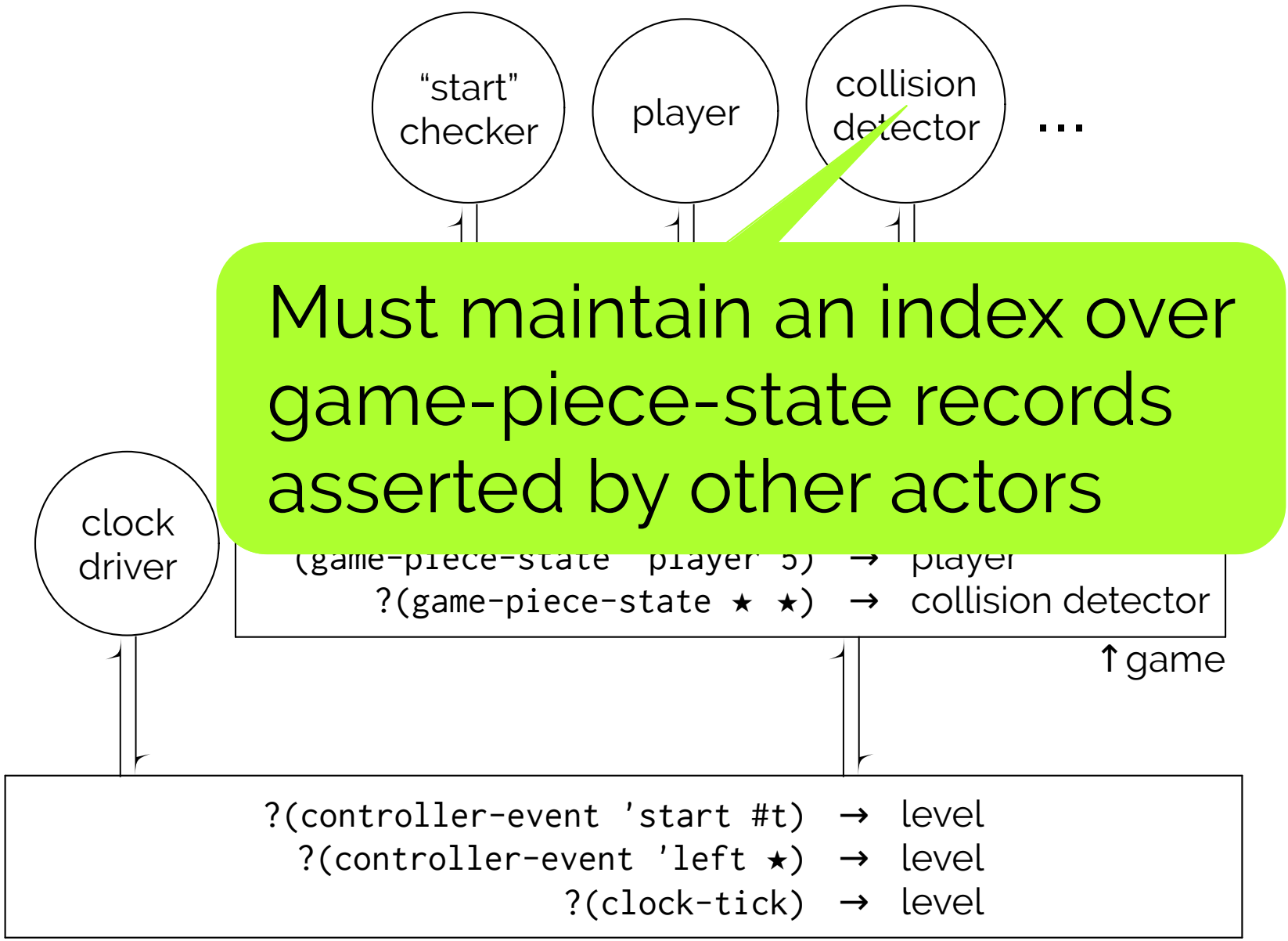
Syndicate DSL by example

- ✓ Mapping events to components
- ✓ Managing conversational state
- • Monitoring changes in shared state

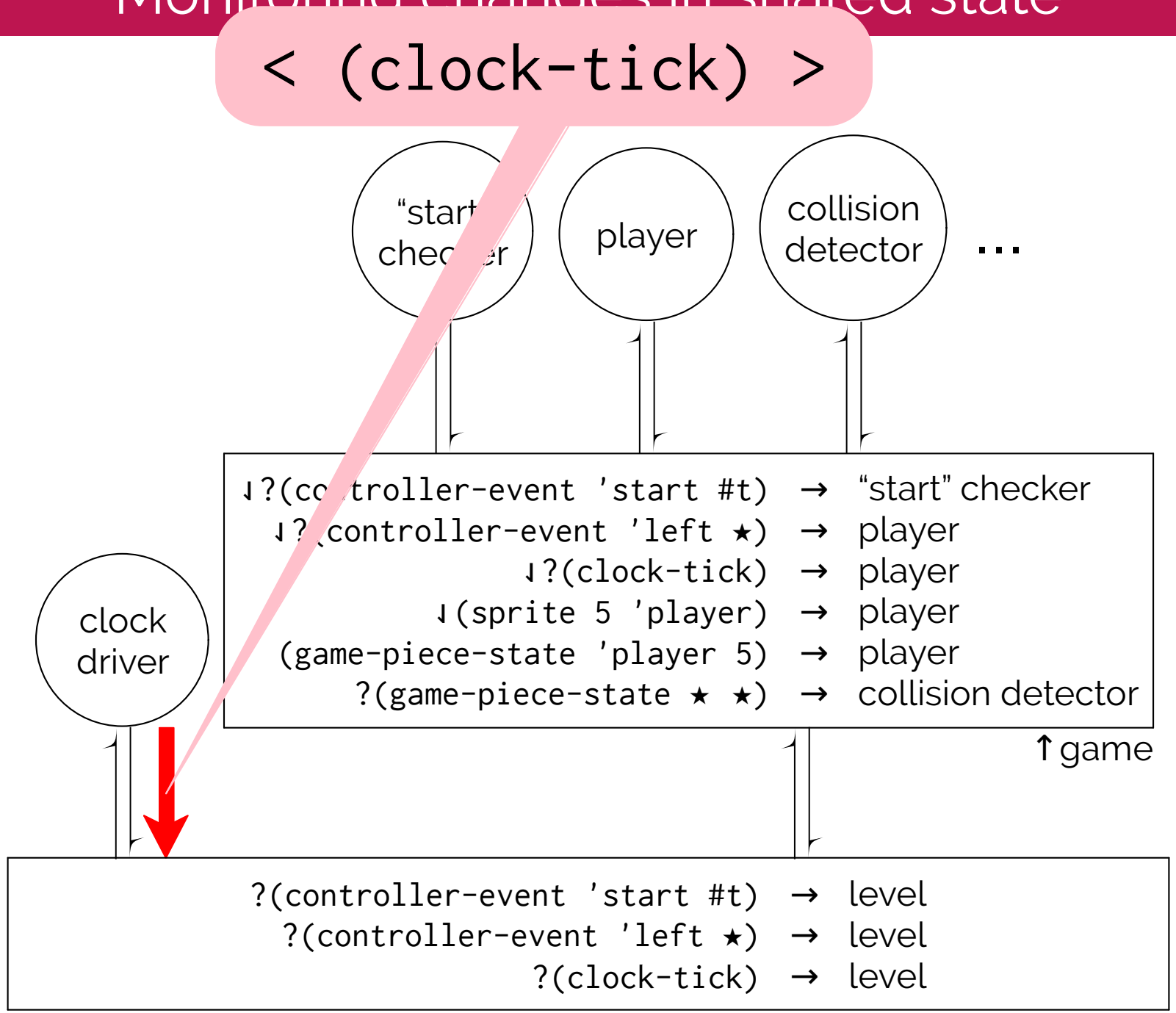
Monitoring changes in shared state



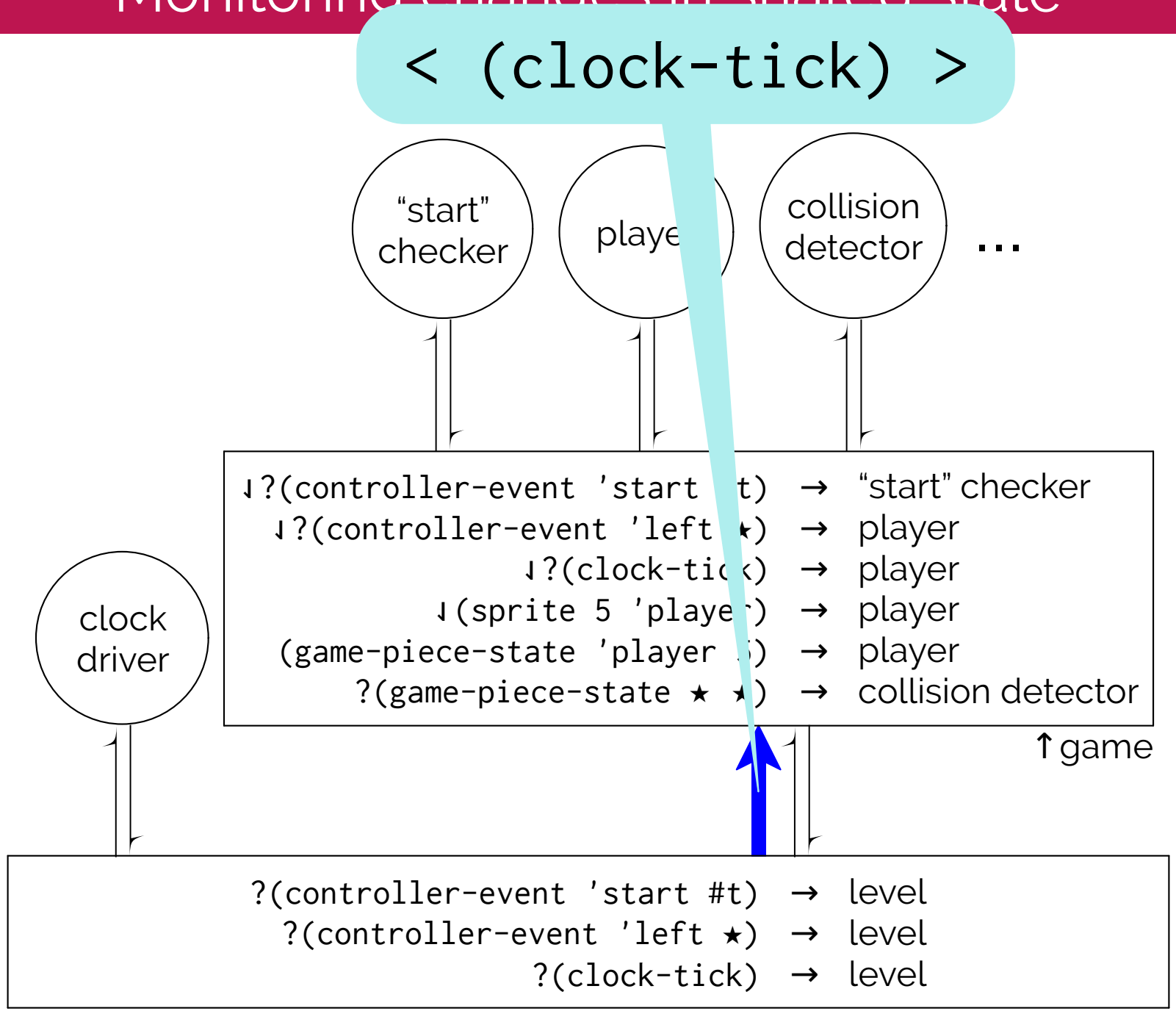
Monitoring changes in shared state



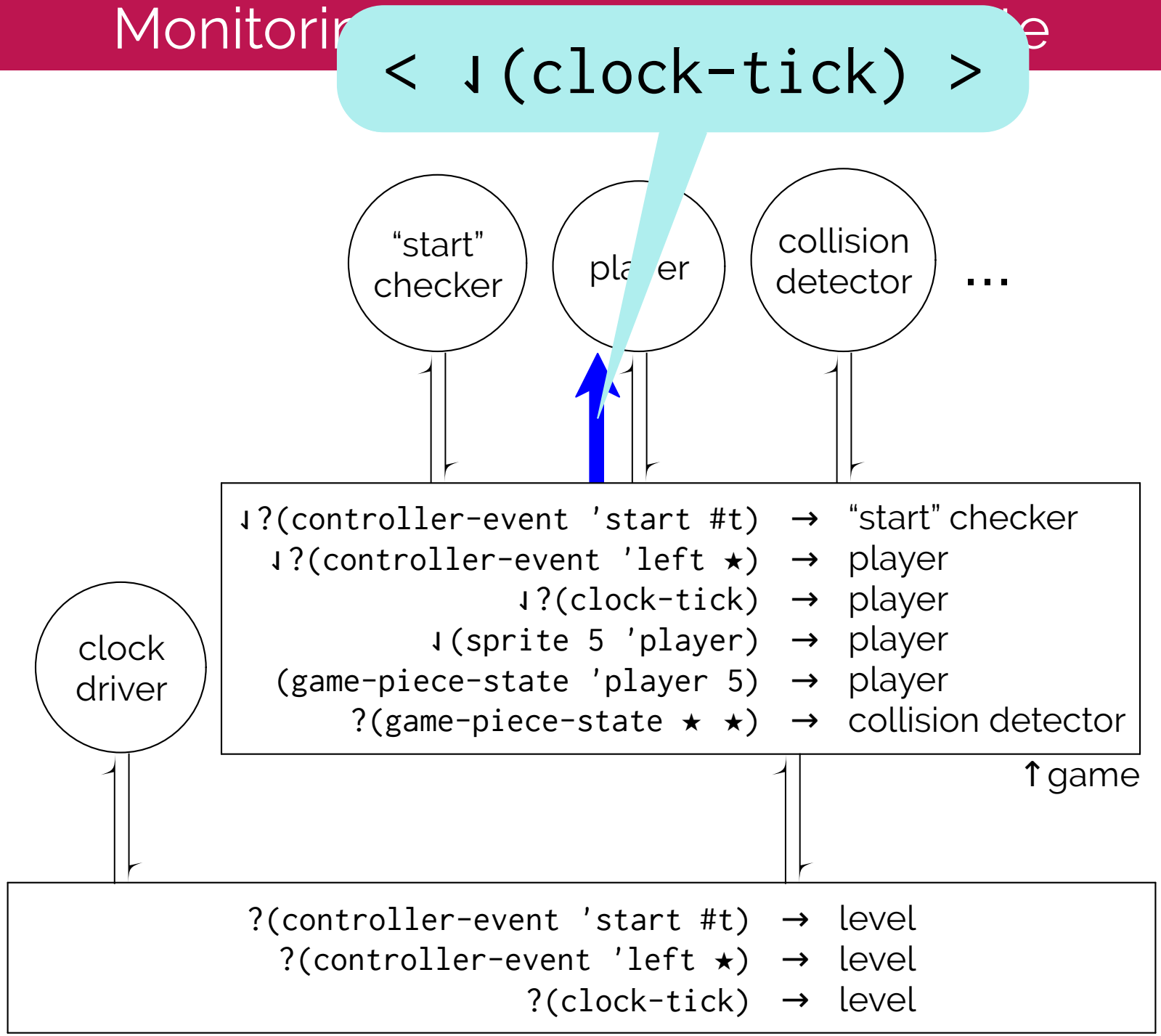
Monitoring changes in shared state



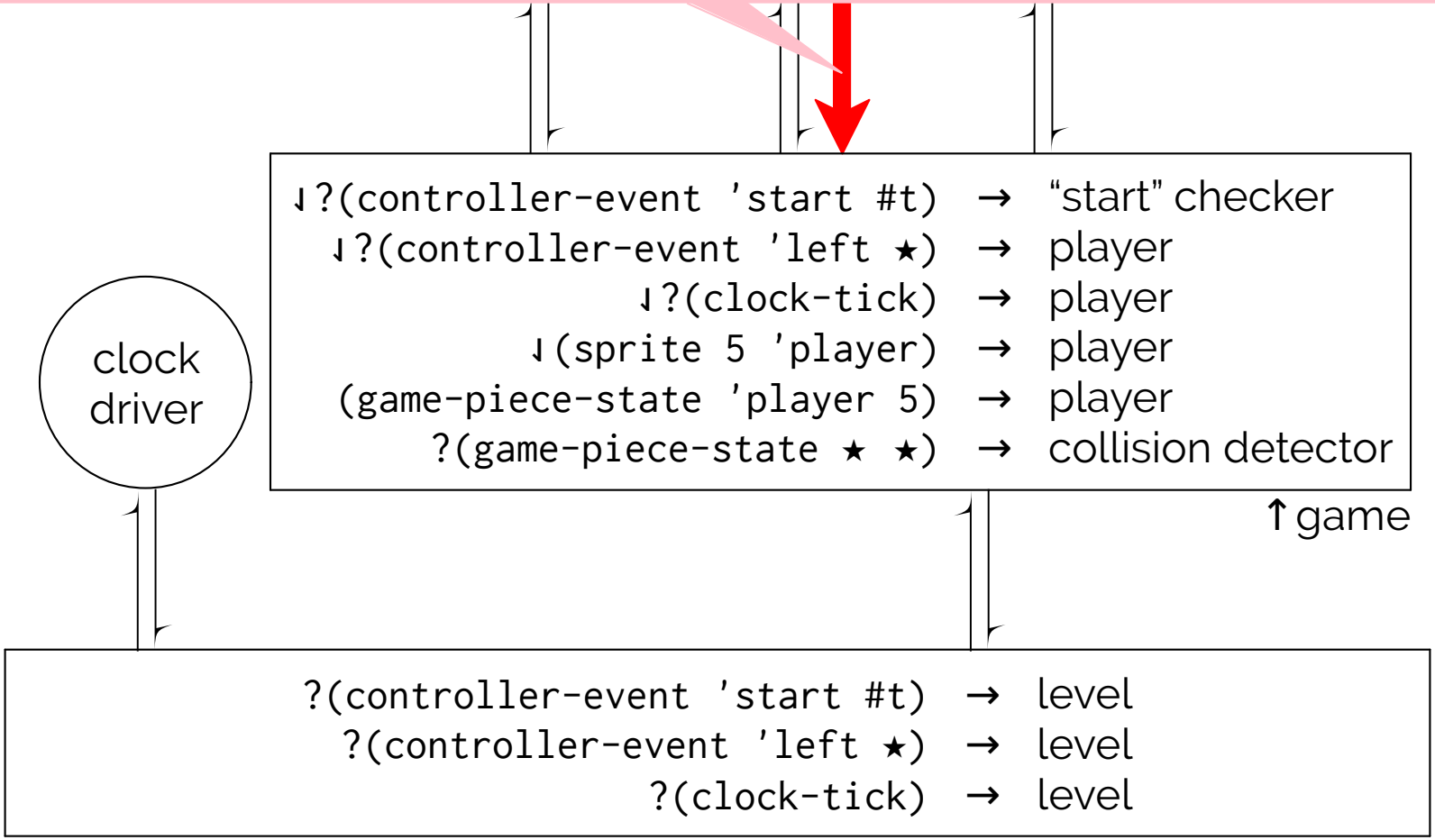
Monitoring changes in shared state



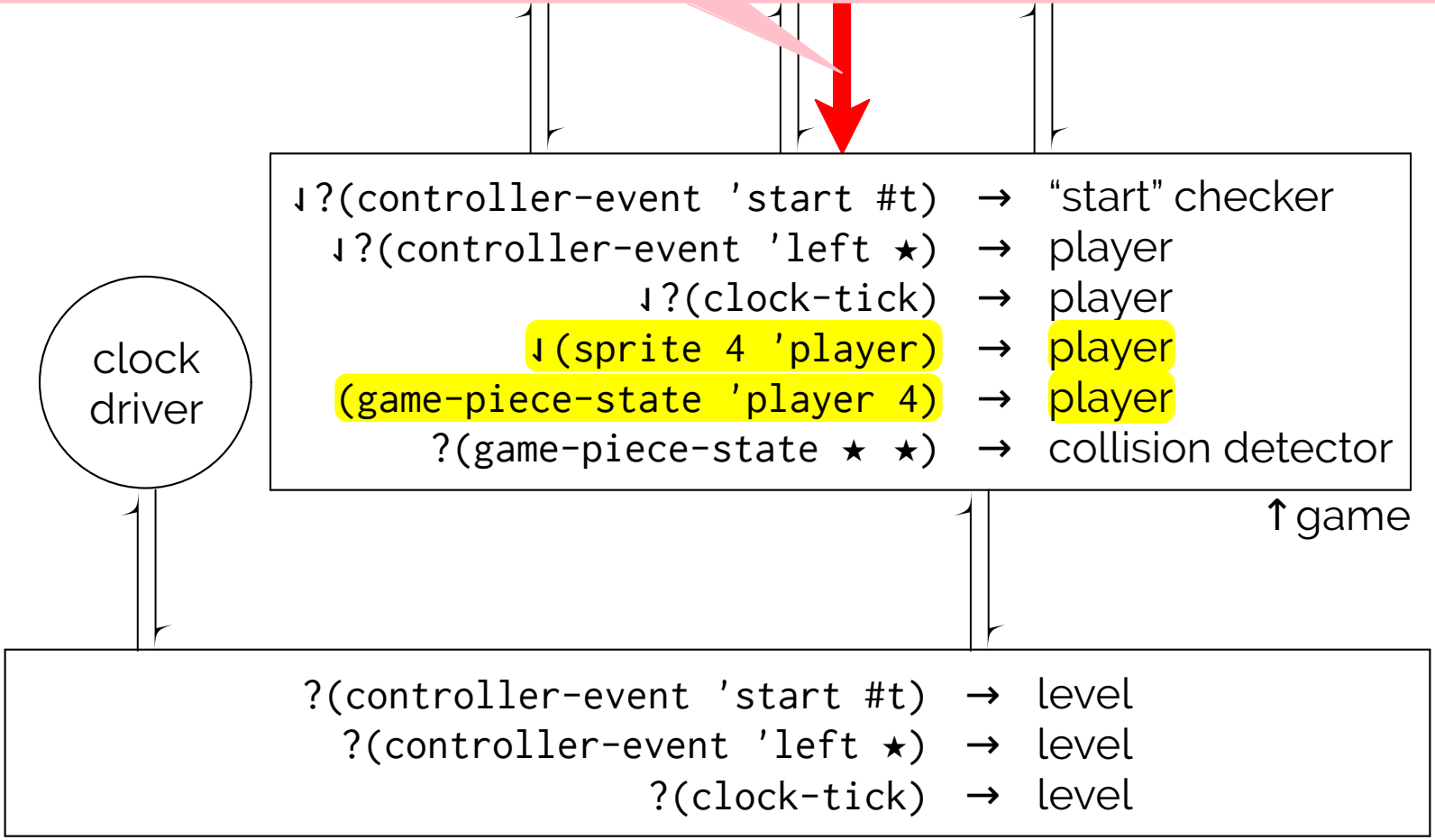
Monitorin e


< ↓(clock-tick) >
"start" checker
pla er
collision detector
...
↓?(controller-event 'start #t) → "start" checker
↓?(controller-event 'left ★) → player
↓?(clock-tick) → player
↓(sprite 5 'player) → player
(game-piece-state 'player 5) → player
?(game-piece-state ★ ★) → collision detector
↑ game
clock driver
?(controller-event 'start #t) → level
?(controller-event 'left ★) → level
?(clock-tick) → level

```
retract( (sprite ★ ★) ),
assert( (sprite 4 'player) ),
retract( (game-piece-state ★ ★) ),
assert( (game-piece-state 'player 4) )
```

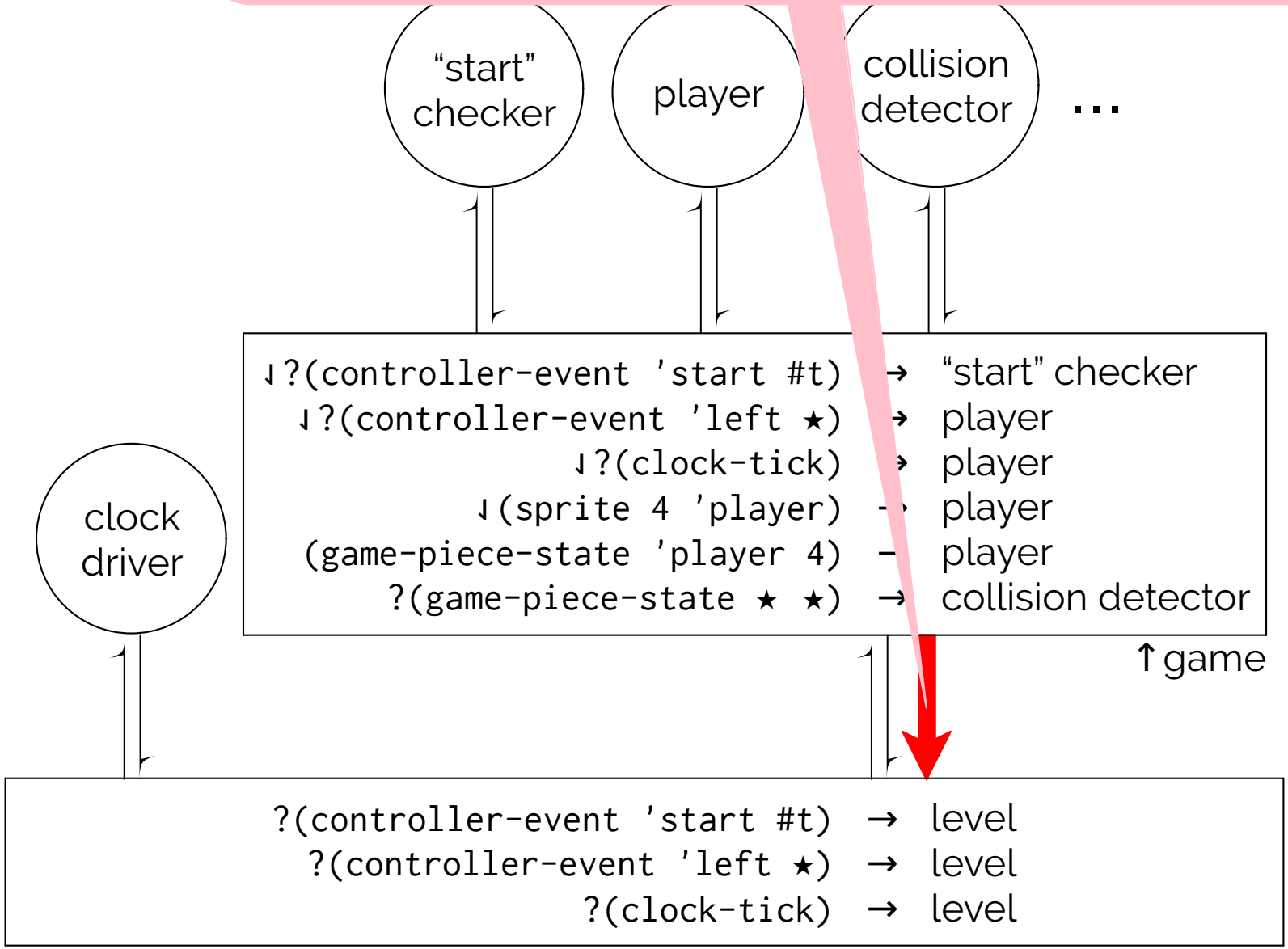


```
retract( (sprite ★ ★) ),
assert( (sprite 4 'player) ),
retract( (game-piece-state ★ ★) ),
assert( (game-piece-state 'player 4) )
```



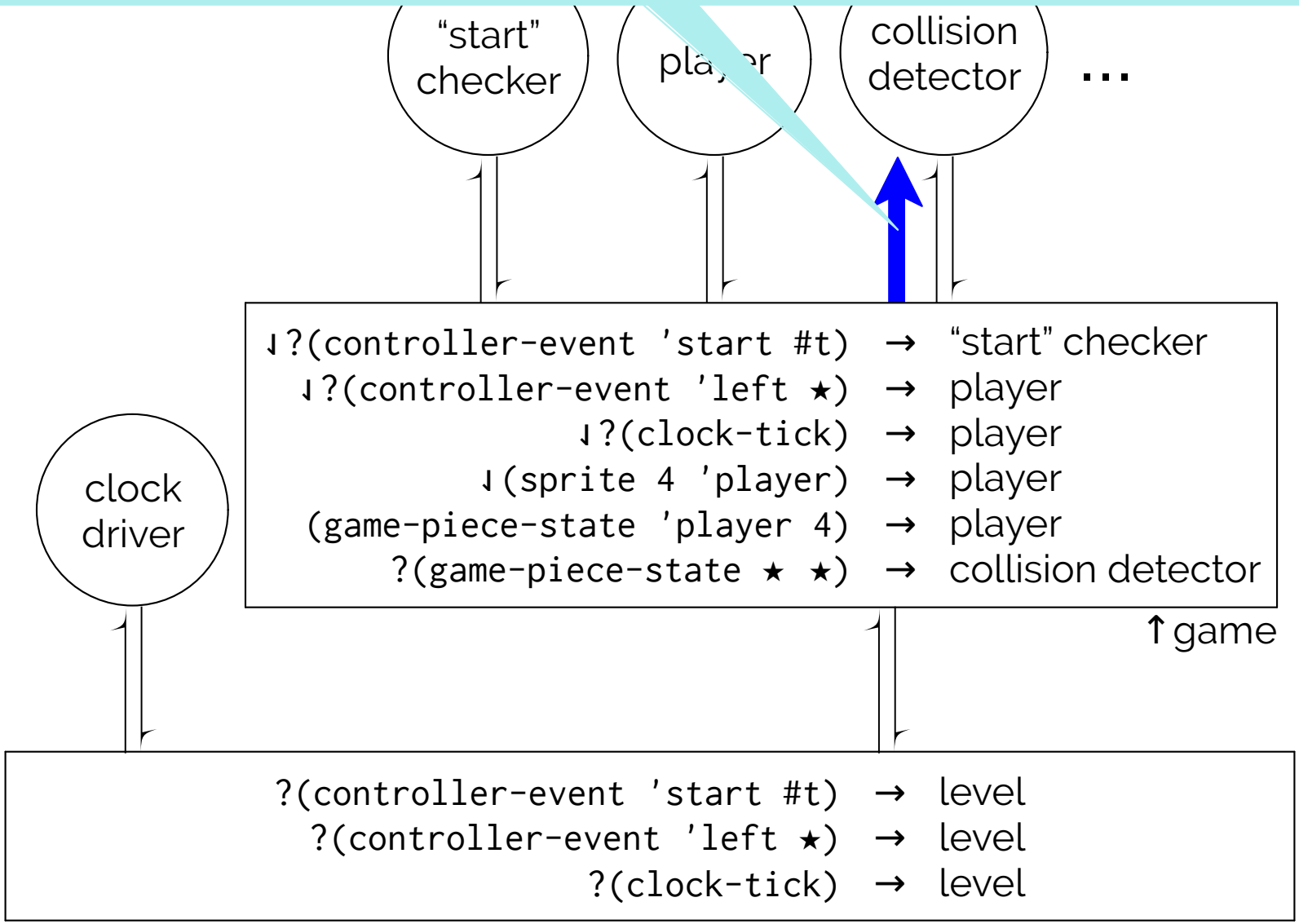
Mo

```
retract( (sprite 5 'player) )
assert( (sprite 4 'player) )
```

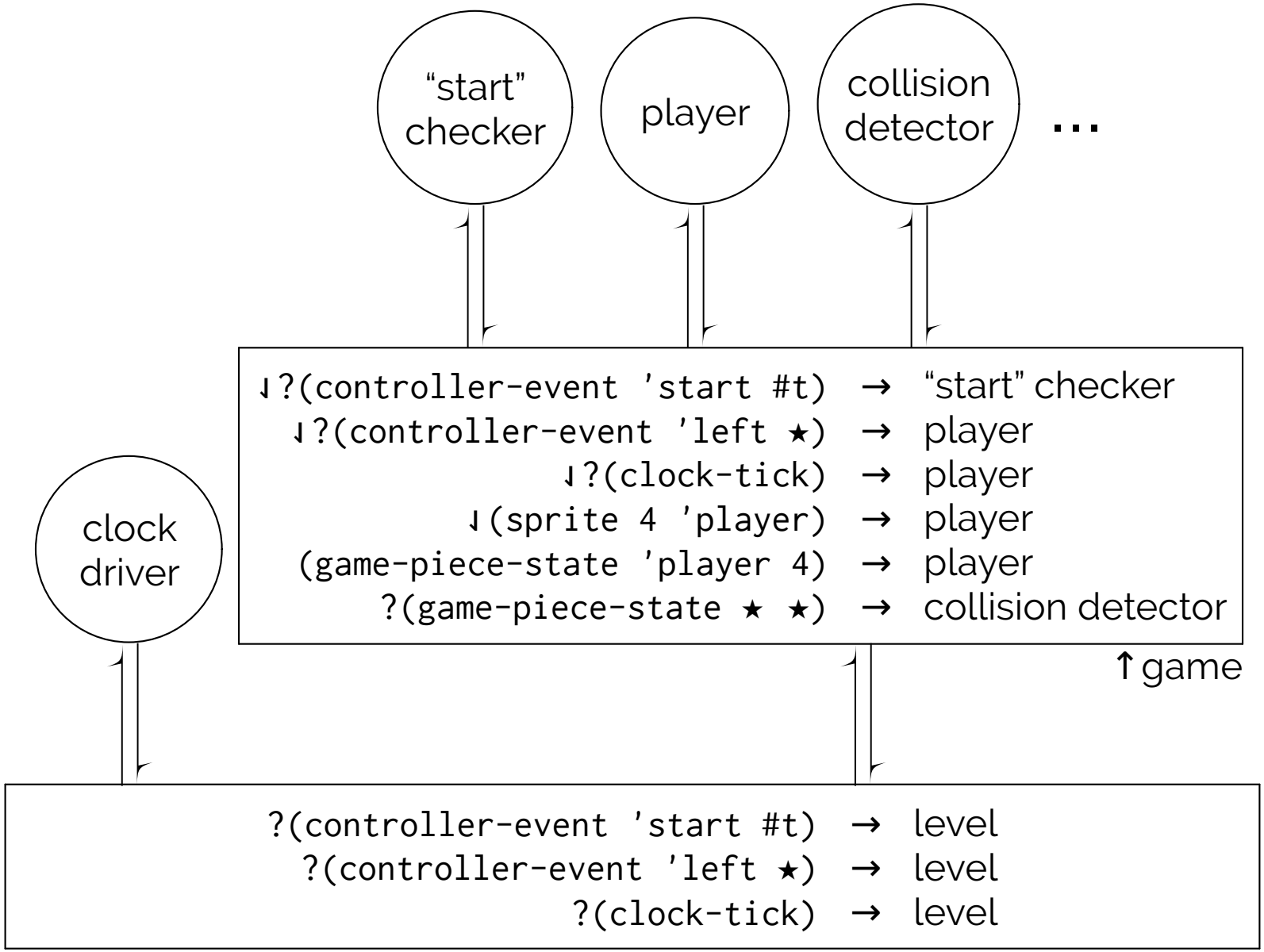


Monitoring changes in shared state

```
retract( (game-piece-state 'player 5) ),
assert( (game-piece-state 'player 4) )
```



Monitoring changes in shared state



```
(struct collision-detection-state (pieces))

(define (spawn-collision-detection)
  (spawn (lambda (evt state)
           (match-event evt
             [(? patch? p)
              (define p0 (collision-detection-state-pieces state))
              (define p1 (for-trie/fold
                             [(pieces p0)]
                             [((game-piece-state $id _) (patch-removed p))]
                           (hash-remove pieces id)))
              (define p2 (for-trie/fold [(pieces p1)]
                             [(($ piece (game-piece-state _ _)) (patch-added p))]
                           (hash-set pieces (game-piece-state-id piece) piece)))
              (transition (struct-copy collision-detection-state state
                                       [pieces p2])
                          '())]))
         (collision-detection-state (hash))
         (sub (game-piece-state ? ?))))
```

```
(struct collision-detection-state (pieces))

(define (spawn-collision-detection)
  (spawn (lambda (evt state)
           (match-event evt
             [(? patch? p)
              (define p0 (collision-detection-state-pieces state))
              (define p1 (for-trie/fold
                          [(pieces p0)]
                          [((game-piece-state $id _) (patch-removed p))]
                          (hash-remove pieces id)))
              (define p2 (for-trie/fold [(pieces p1)]
                          [(($ piece (game-piece-state _ _)) (patch-added p))]
                          (hash-set pieces (game-piece-state-id piece) piece)))
              (transition (struct-copy collision-detection-state state
                                       [pieces p2])
                          '())]))
         (collision-detection-state (hash))
         (sub (game-piece-state ? ?))))
```

```
(struct collision-detection-state (pieces))

(define (spawn-collision-detection)
  (spawn (lambda (evt state)
           (match-event evt
             [(? patch? p)
              (define p0 (collision-detection-state-pieces state))
              (define p1 (for-trie/fold
                          [(pieces p0)]
                          [((game-piece-state $id _) (patch-removed p))]
                          (hash-remove pieces id)))
              (define p2 (for-trie/fold [(pieces p1)]
                          [(($ piece (game-piece-state _ _)) (patch-added p))]
                          (hash-set pieces (game-piece-state-id piece) piece)))
              (transition (struct-copy collision-detection-state state
                                       [pieces p2])
                          '())]))
         (collision-detection-state (hash))
         (sub (game-piece-state ? ?))))
```

```
(struct collision-detection-state (pieces))

(define (spawn-collision-detection)
  (spawn (lambda (evt state)
           (match-event evt
             [(? patch? p)
              (define p0 (collision-detection-state-pieces state))
              (define p1 (for-trie/fold
                                                          p))]
                                                          e)))
           (collision-detection-state (hash))
           (sub (game-piece-state ? ?))))
```

Patch events describe whole *sets* of added and removed assertions, but programmers think about *single* assertions.

```
(struct collision-detection-state (pieces))

(define (spawn-collision-detection)
  (spawn (lambda (evt state)
           (match-event evt
             [(? patch? p)
              (define p0 (collision-detection-state-pieces state))
              (define p1 (for-trie/fold
                             [(pieces p0)]
                             [((game-piece-state $id _) (patch-removed p))]
                           (hash-remove pieces id)))
              (define p2 (for-trie/fold [(pieces p1)]
                             [(($ piece (game-piece-state _ _)) (patch-added p))]
                           (hash-set pieces (game-piece-state-id piece) piece)))
              (transition (struct-copy collision-detection-state state
                                       [pieces p2])
                          '())]))
         (collision-detection-state (hash))
         (sub (game-piece-state ? ?))))
```

```
(struct collision-detection-state (pieces))

(define (spawn-collision-detection)
  (spawn (lambda (evt state)
           (match-event evt
             [(? patch? p)
              (define p0 (collision-detection-state-pieces state))
              (define p1 (for-trie/fold
                             [(pieces p0)]
                             [((game-piece-state $id _) (patch-removed p))]
                           (hash-remove pieces id)))
              (define p2 (for-trie/fold [(pieces p1)]
                             [(($ piece (game-piece-state _ _)) (patch-added p))]
                           (hash-set pieces (game-piece-state-id piece) piece)))
              (transition (struct-copy collision-detection-state state
                                       [pieces p2])
                          '())]))
         (collision-detection-state (hash))
         (sub (game-piece-state ? ?))))
```

```
(struct collision-detection-state (pieces))

(define (spawn-collision-detection)
  (spawn (lambda (evt state)
           (match-event evt
             [(? patch? p)
              (define p0 (collision-detection-state-pieces state))
              (define p1 (for-trie/fold
                          [(pieces p0)]
                          [((game-piece-state $id _) (patch-removed p))]
                          (hash-remove pieces id)))
              (define p2 (for-trie/fold [(pieces p1)]
                          [(($ piece (game-piece-state _ _)) (patch-added p))]
                          (hash-set pieces (game-piece-state-id piece) piece)))
              (transition (struct-copy collision-detection-state state
                                       [pieces p2])
                          '())]))
         (collision-detection-state (hash))
         (sub (game-piece-state ? ?))))
```

```
(struct collision-detection-state (pieces))

(define (spawn-collision-detection)
  (spawn (lambda (evt state)
           (match-event evt
             [(? patch? p)
              (define p0 (collision-detection-state-pieces state))
              (define p1 (for-trie/fold
                          [(pieces p0)]
                          [((game-piece-state $id _) (patch-removed p))]
                          (hash-remove pieces id)))
              (define p2 (for-trie/fold [(pieces p1)]
                          [(($ piece (game-piece-state _ _)) (patch-added p))]
                          (hash-set pieces (game-piece-state-id piece) piece)))
              (transition (struct-copy collision-detection-state state
                                       [pieces p2])
                          '())]))
         (collision-detection-state (hash))
         (sub (game-piece-state ? ?))))
```

```
(struct collision-detection-state (pieces))

(define (spawn-collision-detection)
  (spawn (lambda (evt state)
           (match-event evt
             [(? patch? p)
              (define p0 (collision-detection-state-pieces state))
              (define p1 (for-trie/fold
                             [(pieces p0)]
                             [((game-piece-state $id _) (patch-removed p))]
                           (hash-remove pieces id)))
              (define p2 (for-trie/fold [(pieces p1)]
                             [(($ piece (game-piece-state _ _)) (patch-added p))]
                           (hash-set pieces (game-piece-state-id piece) piece)))
              (transition (struct-copy collision-detection-state state
                                       [pieces p2])
                          '())]))
         (collision-detection-state (hash))
         (sub (game-piece-state ? ?))))
```

3× repetition of pattern

```
(struct collision-detection-state (pieces))

(define (spawn-collision-detection)
  (spawn (lambda (evt state)
           (match-event evt
             [(? patch? p)
              (define p0 (collision-detection-state-pieces state))
              (define p1 (for-trie/fold
                          [(pieces p0)]
                          [((game-piece-state $id _) (patch-removed p))]
                        (hash-remove pieces id)))
              (define p2 (for-trie/fold [(pieces p1)]
                          [(($ piece (game-piece-state _ _)) (patch-added p))]
                        (hash-set pieces (game-piece-state-id piece) piece)))
              (transition (struct-copy collision-detection-state state
                                       [pieces p2])
                          '())]))
         (collision-detection-state (hash))
         (sub (game-piece-state ? ?))))
```

```
(define (spawn-collision-detection)
  (actor
   (forever #:collect [(pieces (hash))]
    (on (retracted (game-piece-state $id _))
        (hash-remove pieces id))
    (on (asserted ($ piece (game-piece-state _ _)))
        (hash-set pieces (game-piece-state-id piece) piece)))))
```

```
(define (spawn-collision-detection)
  (actor
   (forever #:collect [(pieces (hash))]
    (on (retracted (game-piece-state $id _))
        (hash-remove pieces id))
    (on (asserted ($ piece (game-piece-state _ _)))
        (hash-set pieces (game-piece-state-id piece) piece)))))
```

```
(define (spawn-collision-detection)
  (actor
   (forever #:collect [(pieces (hash))]
    (on (retracted (game-piece-state $id _))
        (hash-remove pieces id))
    (on (asserted ($ piece (game-piece-state _ _)))
        (hash-set pieces (game-piece-state-id piece) piece)))))
```

```
(define (spawn-collision-detection)
  (actor
   (forever #:collect [(pieces (hash))]
    (on (retracted (game-piece-state $id _))
        (hash-remove pieces id))
    (on (asserted ($ piece (game-piece-state _ _)))
        (hash-set pieces (game-piece-state-id piece) piece)))))
```

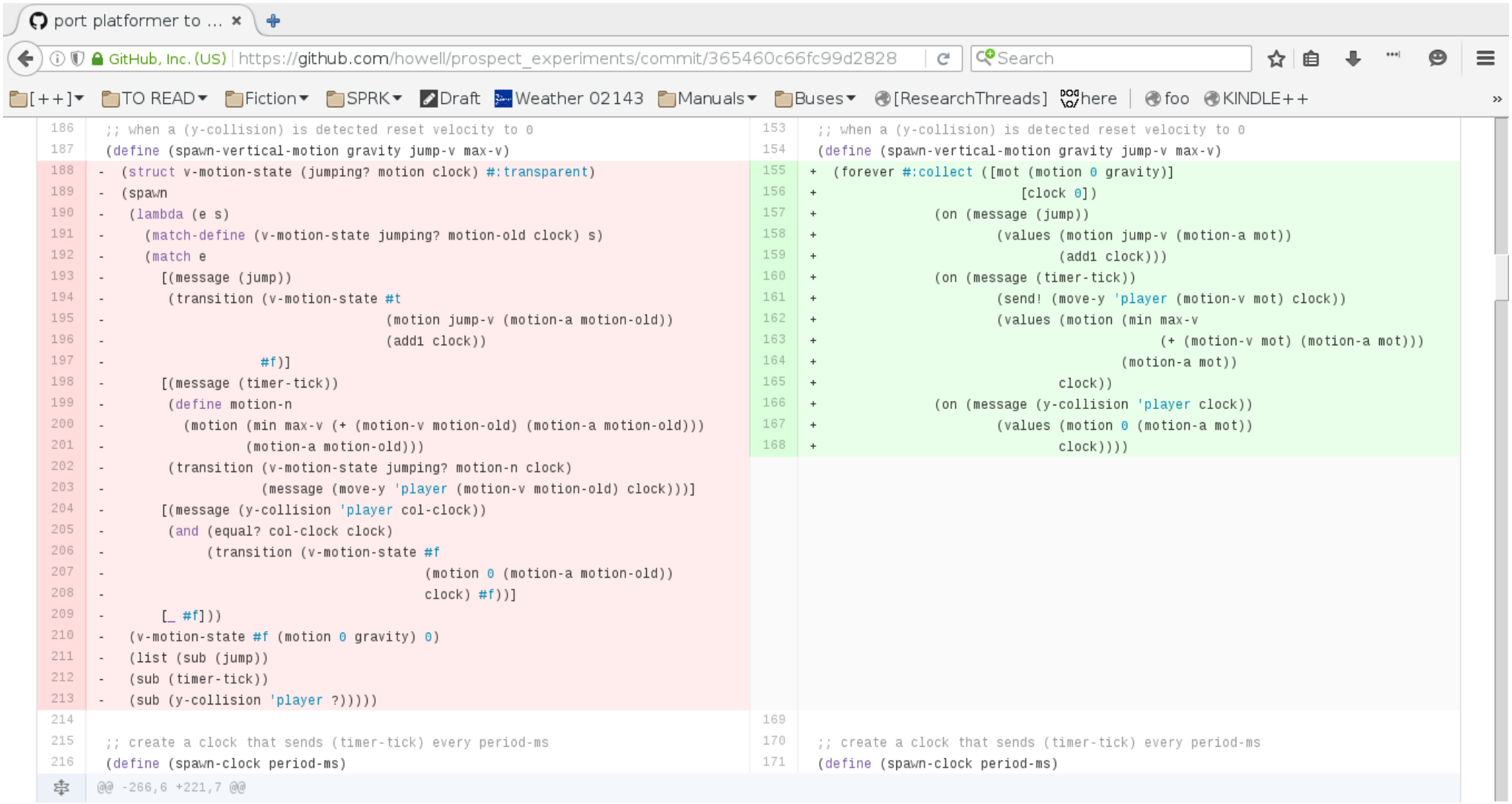
```
(define (spawn-collision-detection)
  (actor
   (forever #:collect [(pieces (hash))]
    (on (retracted (game-piece-state $id _))
        (hash-remove pieces id))
    (on (asserted ($ piece (game-piece-state _ _)))
        (hash-set pieces (game-piece-state-id piece) piece)))))
```

```
(actor
 (forever
  (query [pieces (hash id piece) ; “group-by”
                 ($ piece (game-piece-state $id _))])
  (on (changed pieces)
      ...)))
```

Syndicate DSL by example

- ✓ Mapping events to components
- ✓ Managing conversational state
- ✓ Monitoring changes in shared state

Status



Strong space savings in most places

syn·di·cate

a language for interactive programs

Progress Report on DSL Design

Repeated idioms ⟶ Language features

Future work:
- Improved state sharing with substates
- "Queries" (e.g. "group-by")
- Non-naive compilation strategy
- Better technique for naming metalevels
- More evaluations & case studies

`http://syndicate-lang.org/`