

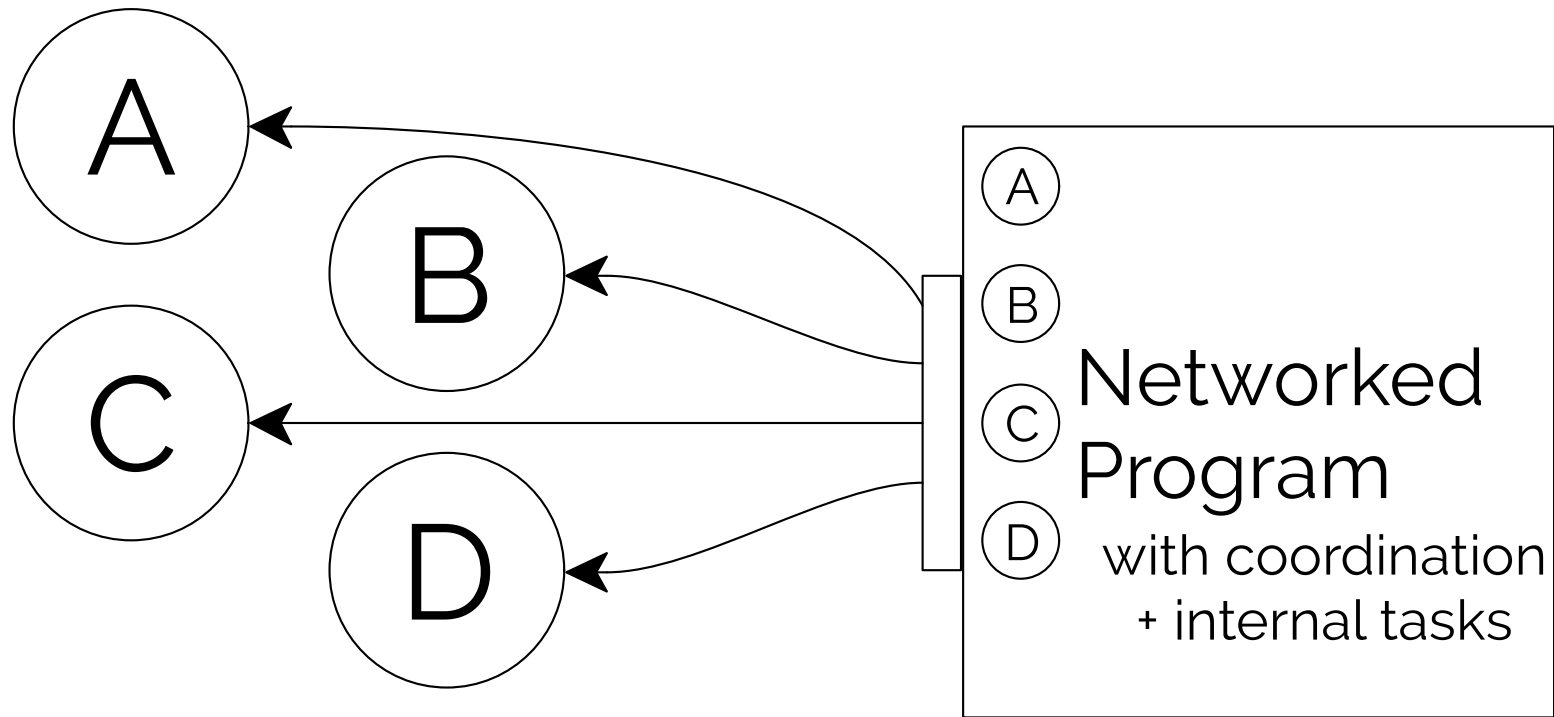
From Events to Reactions: A Progress Report

Tony Garnock-Jones

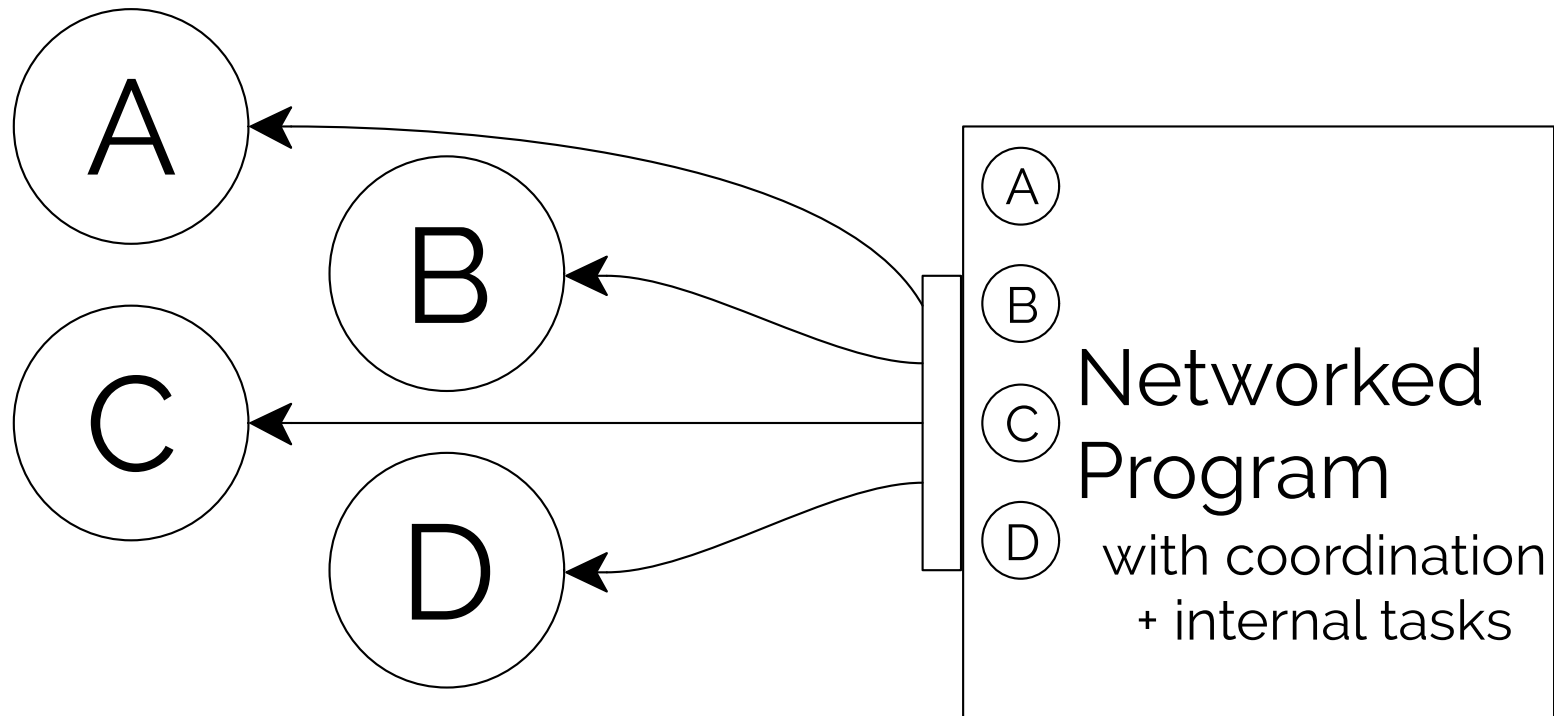
tonyg@ccs.neu.edu

Northeastern University

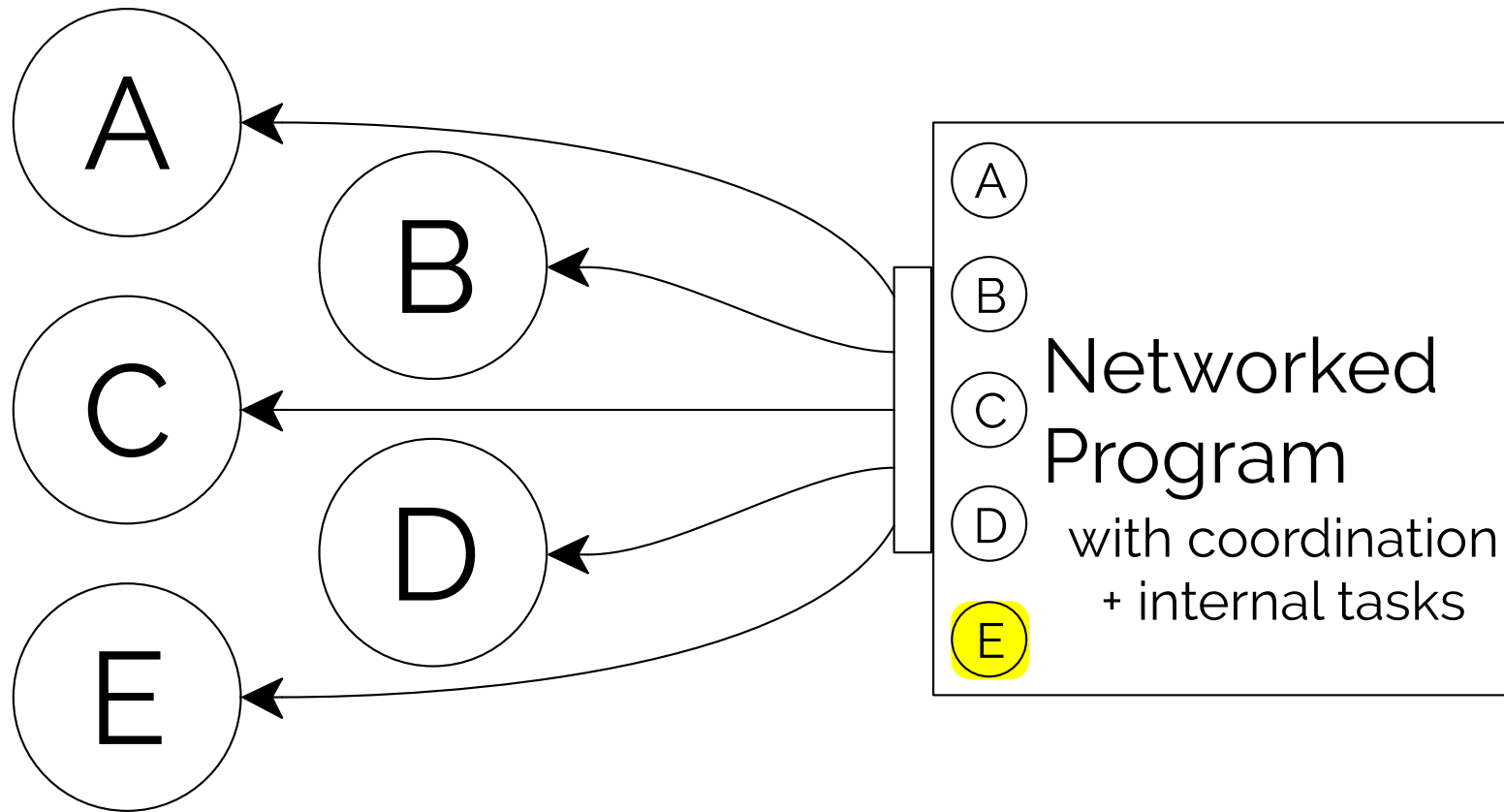
Joint work with Matthias Felleisen and Sam Caldwell



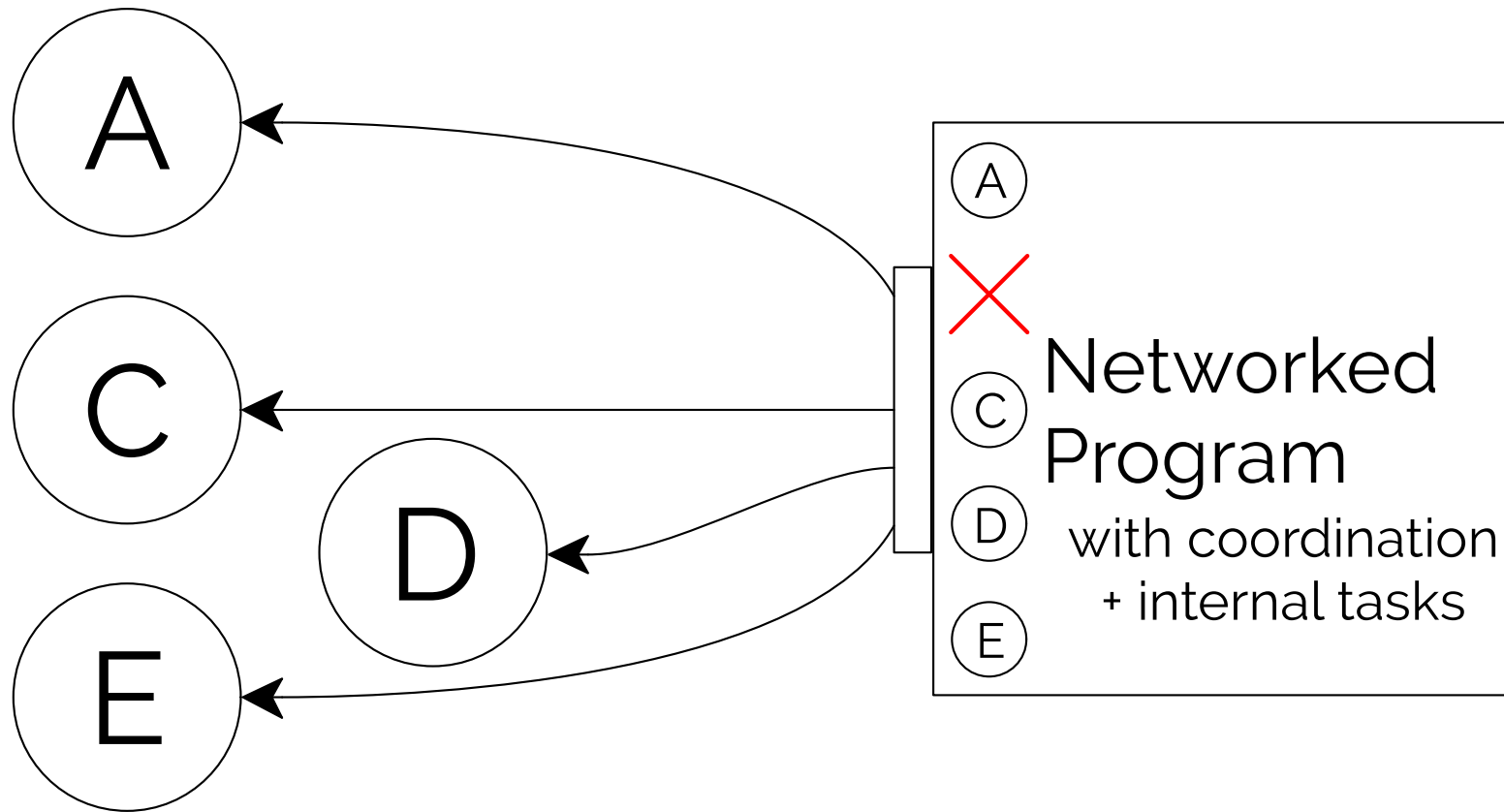
Interactivity \Rightarrow External Concurrency



Interactivity \Rightarrow External Concurrency
Internal Organisation Reflects External Concurrency



Interactivity \Rightarrow External Concurrency
Component startup \rightarrow interaction \rightarrow shutdown/failure



Interactivity \Rightarrow External Concurrency
Component startup \rightarrow interaction \rightarrow shutdown/failure

Syndicate DSL by example

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

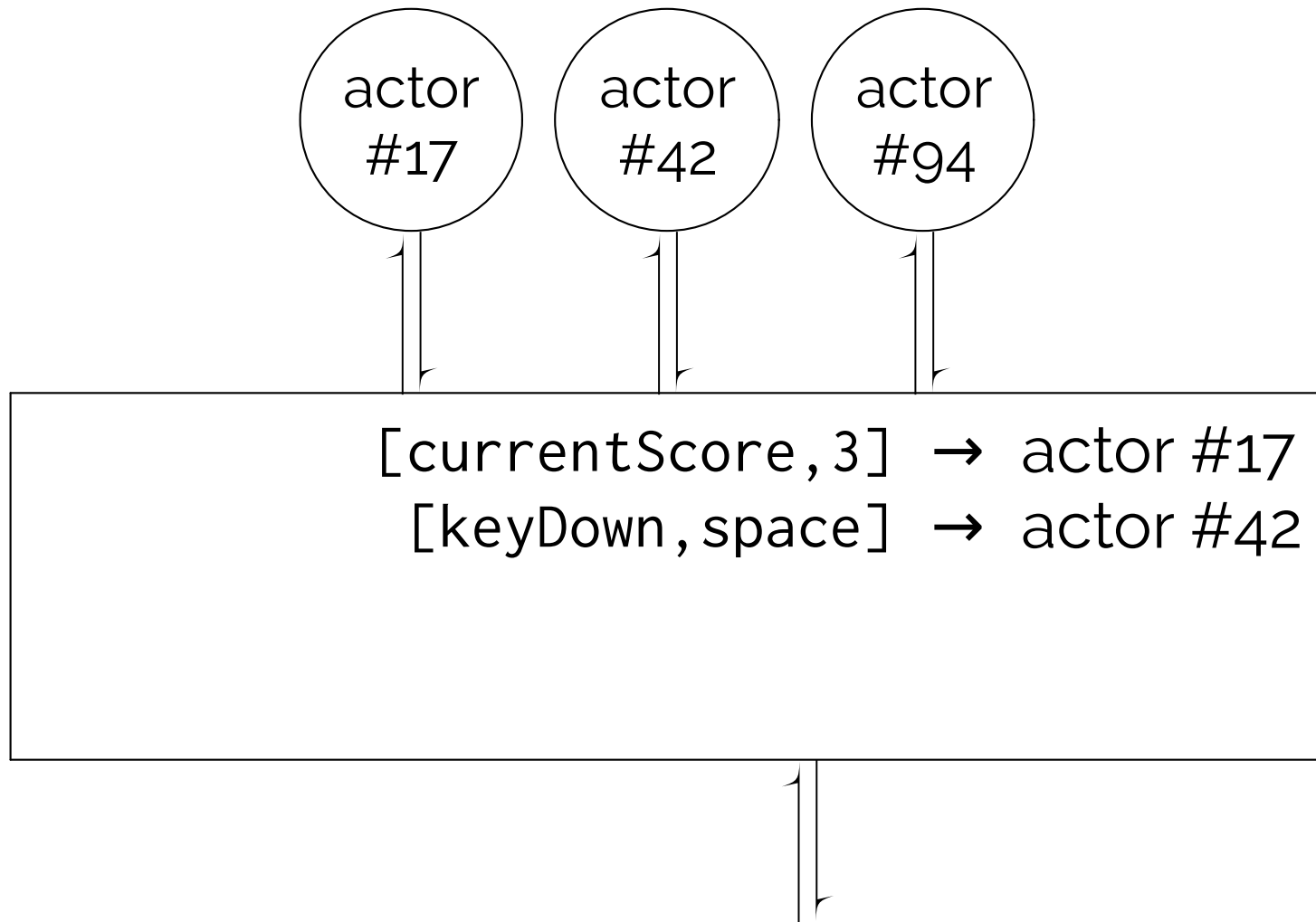
Score: 3



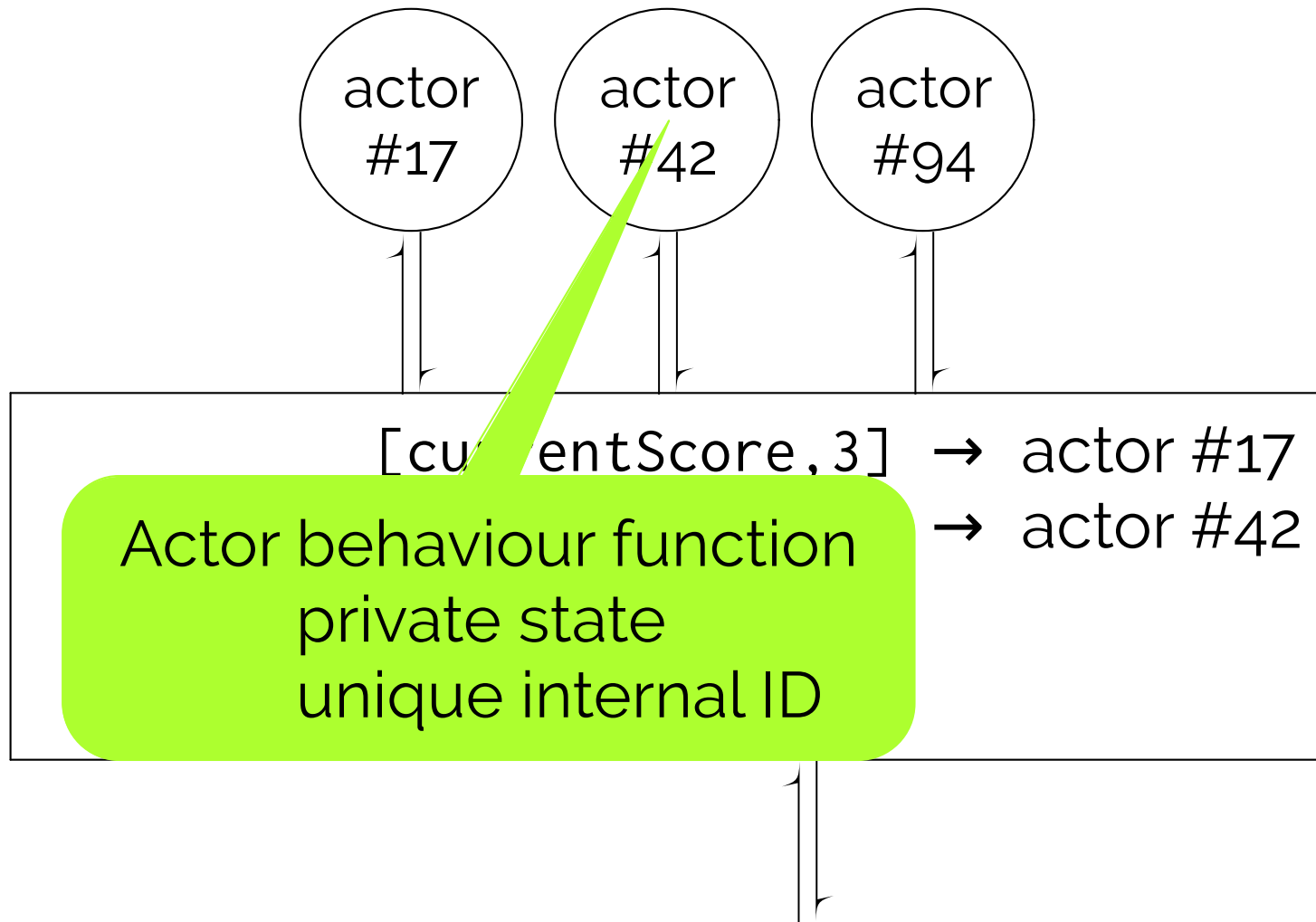
SYNDICATE

event × state → [action] × state

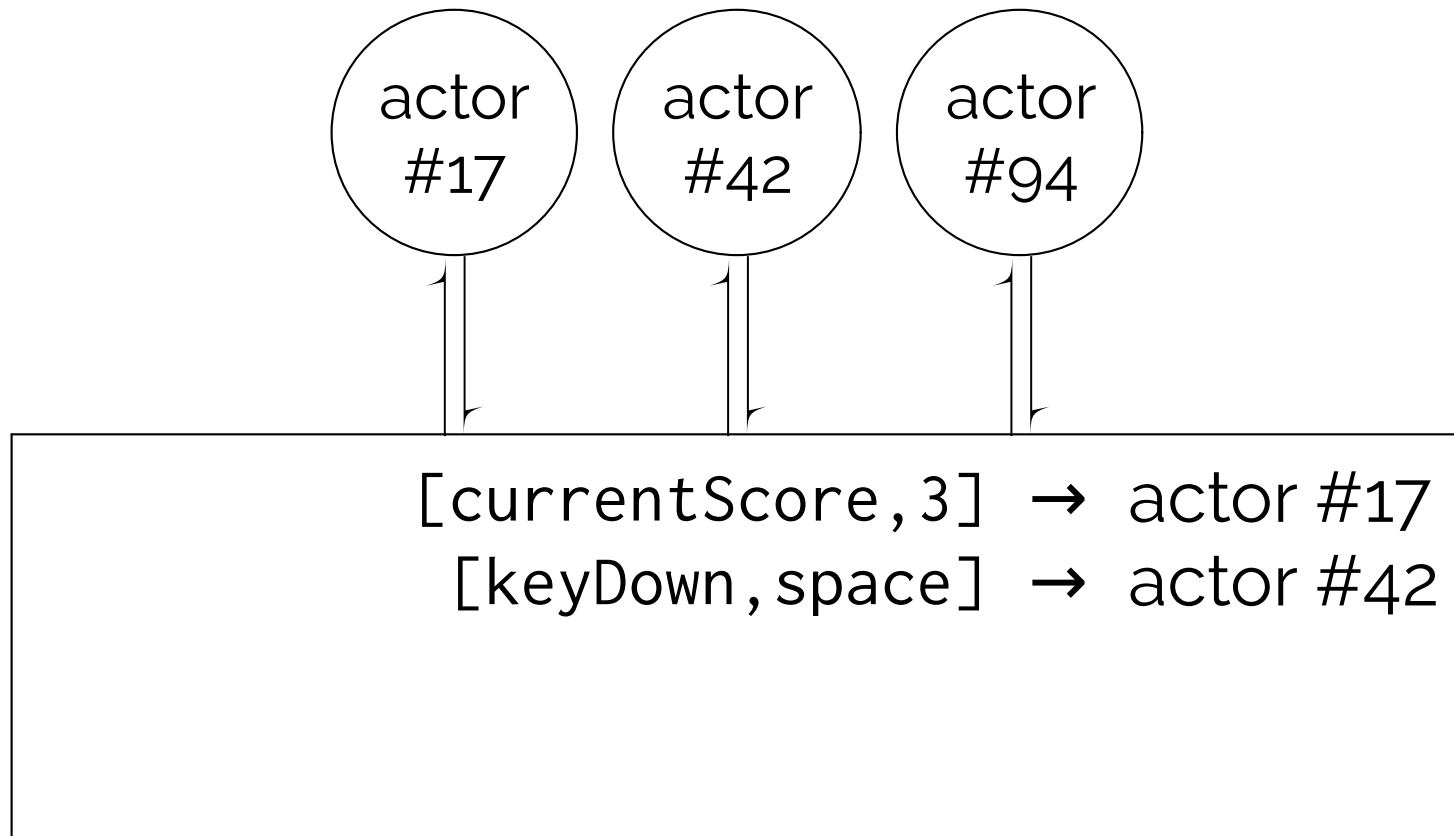
event × state → [action] × state



event × state → [action] × state



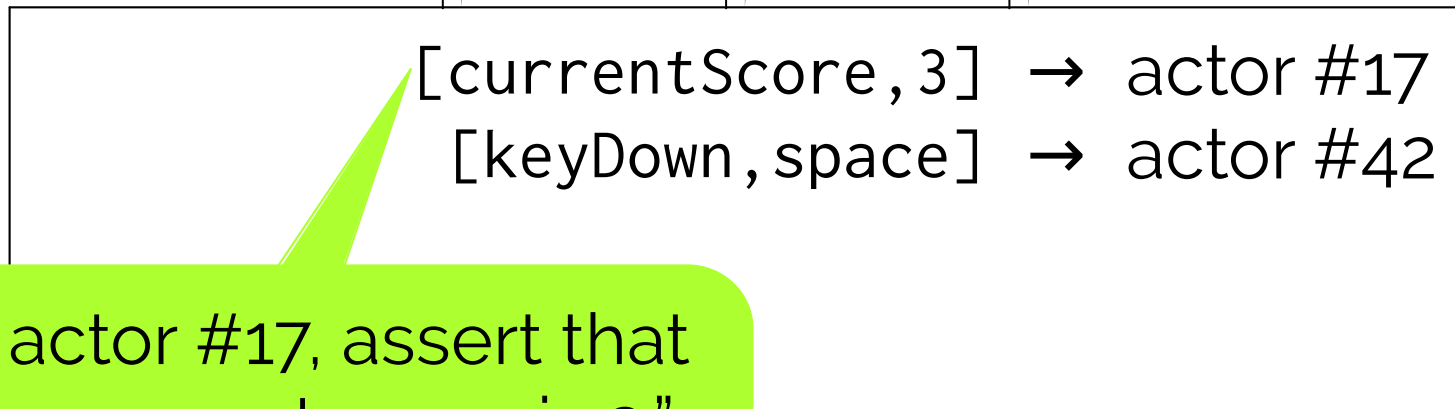
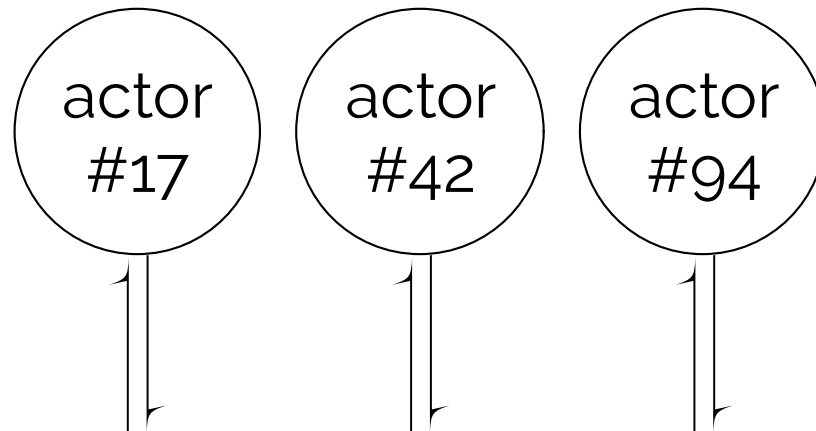
event × state → [action] × state



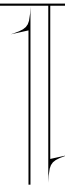
Dataspace: assertions + provenance

cf. Linda's
"Tuplespaces"

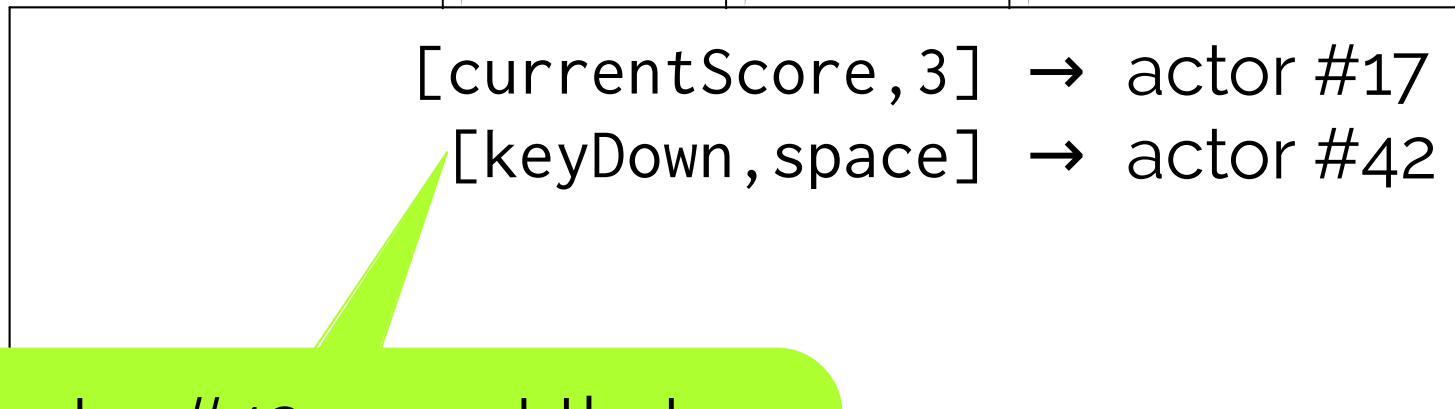
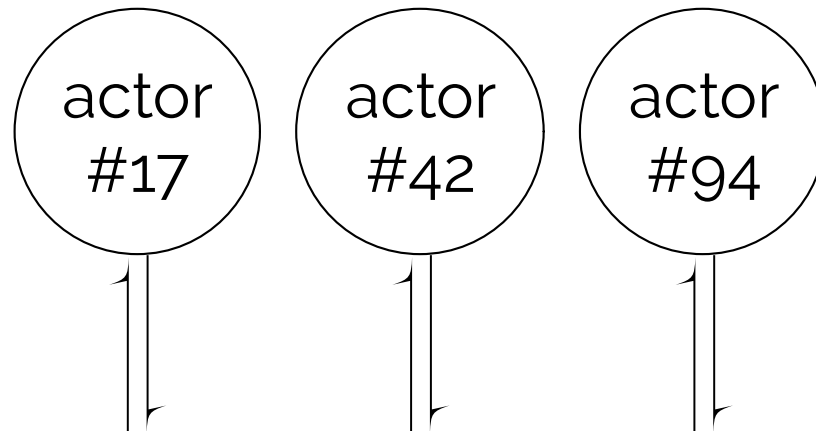
event × state → [action] × state



"I, actor #17, assert that the current score is 3."



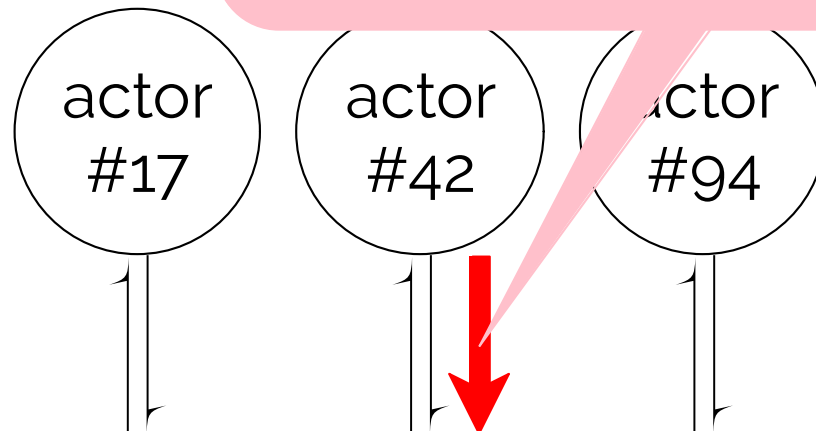
event × state → [action] × state



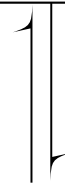
“I, actor #42, assert that the space key is currently held down.”

event × state

Actions carry added and removed assertions
actor → environment

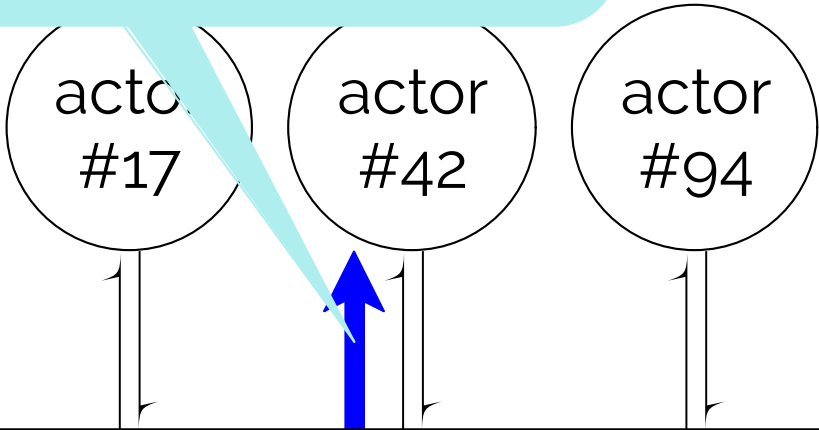


[currentScore, 3] → actor #17
[keyDown, space] → actor #42

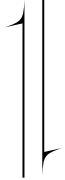


ev [currentScore, 3] × state


Events carry added and removed assertions environment → actor

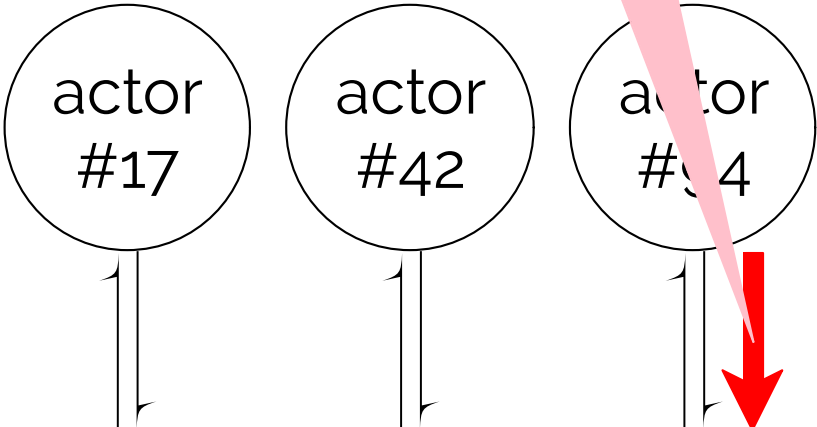


[currentScore, 3] → actor #17
[keyDown, space] → actor #42

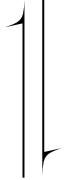


event


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

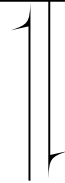
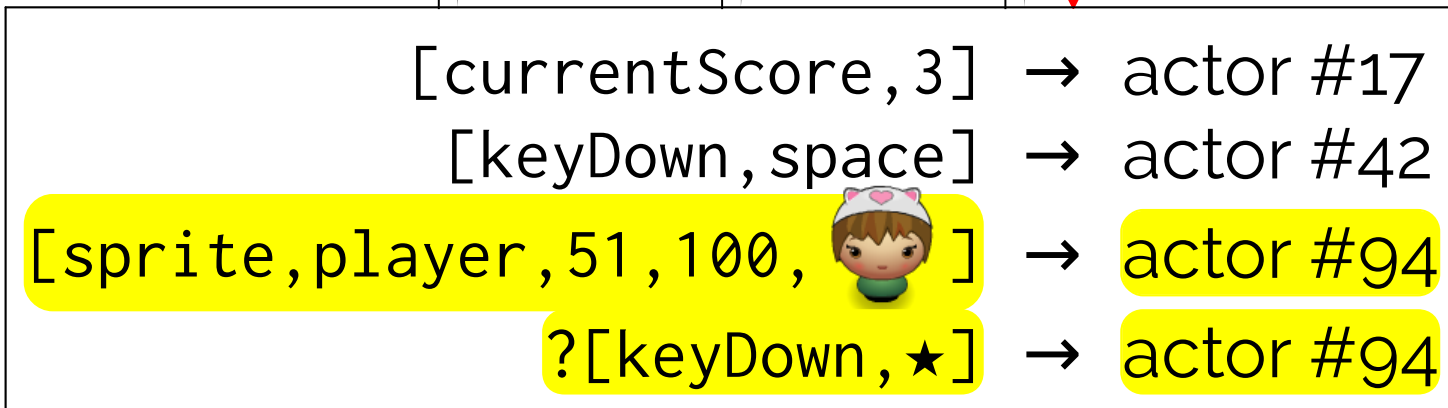
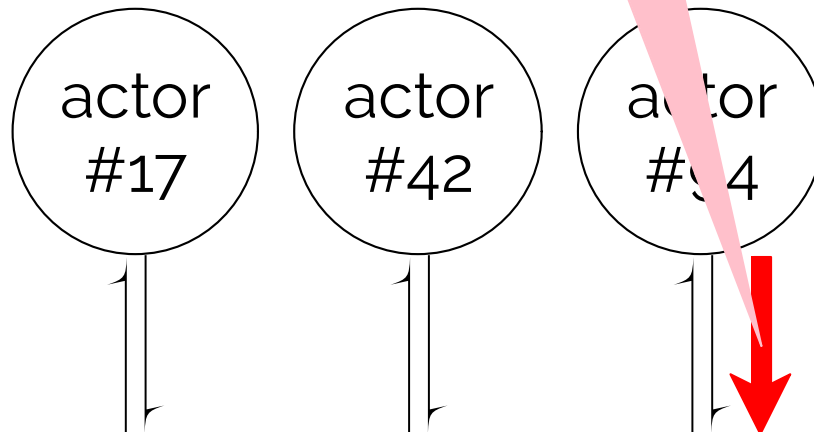


[currentScore, 3]	→	actor #17
[keyDown, space]	→	actor #42




event

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



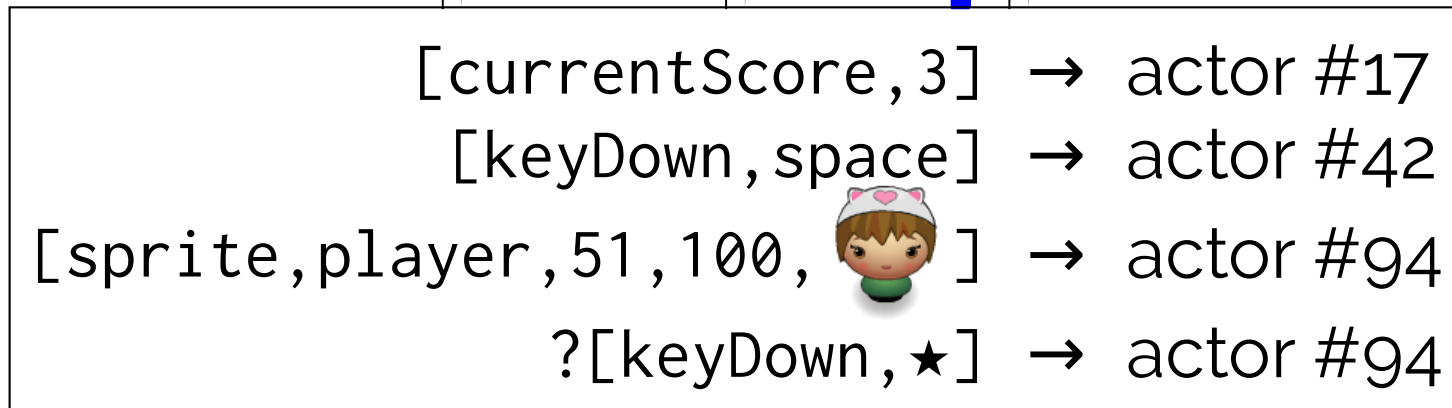
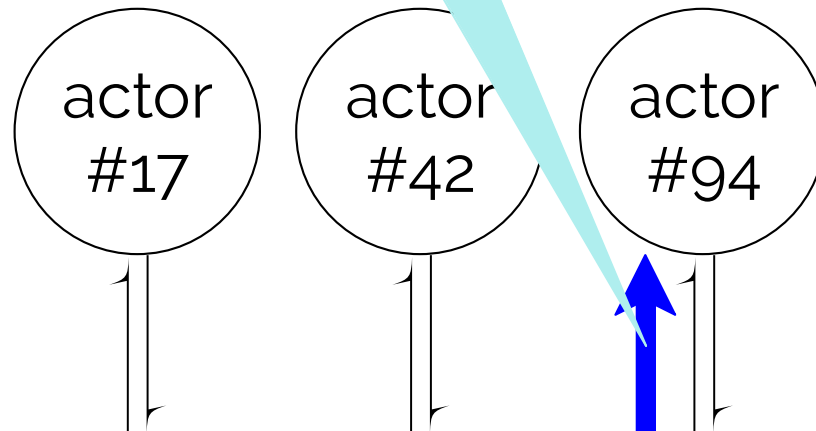
event × state → [action] × state

"I, actor #94, am interested in keeping track of assertions of the form [keyDown, ★]."

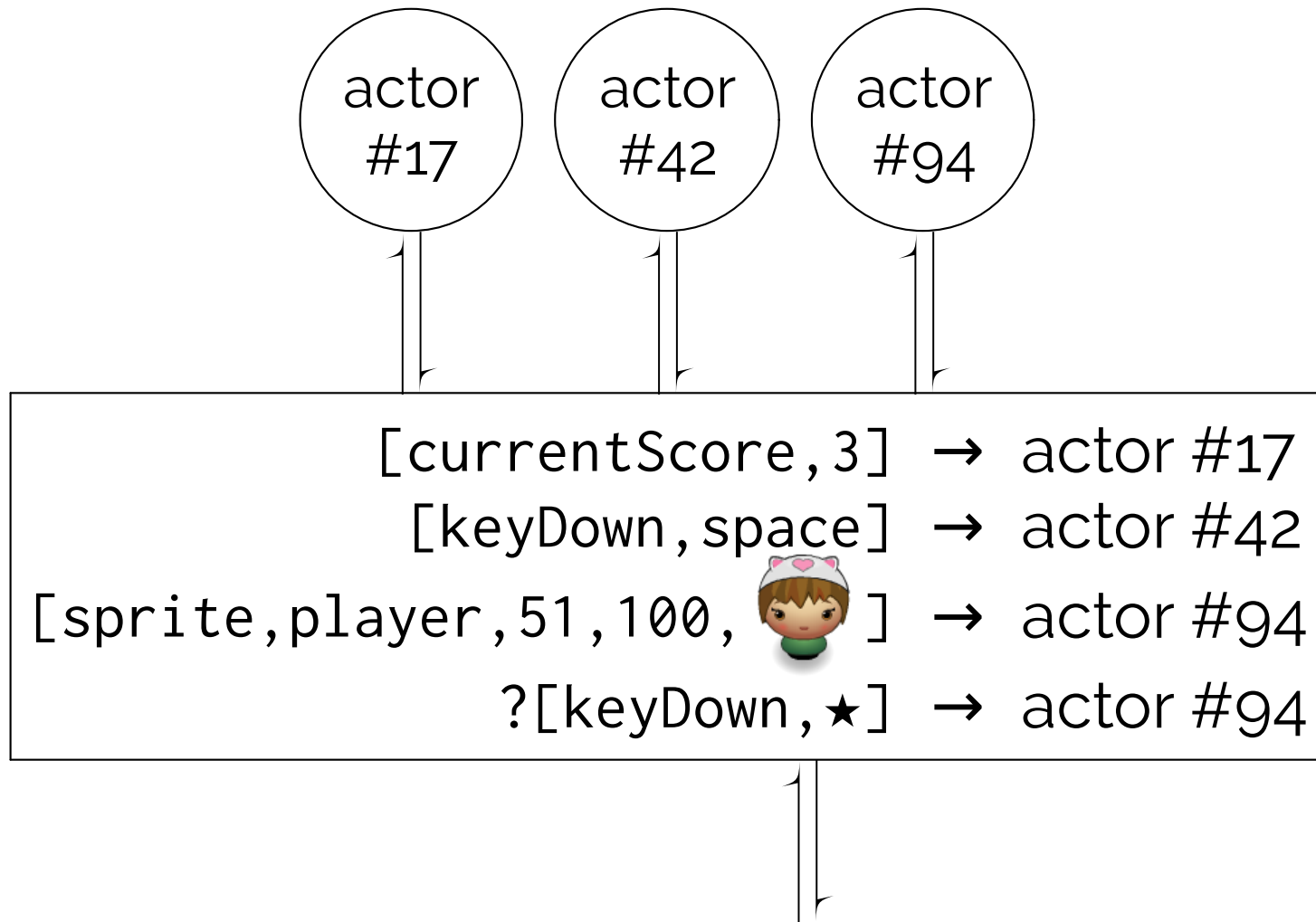
[currentScore, 3] → actor #17
[keyDown, space] → actor #42
[sprite, player, 51, 100, ] → actor #94
?[keyDown, ★] → actor #94

event × state → [action] × state

assert([keyDown, space])

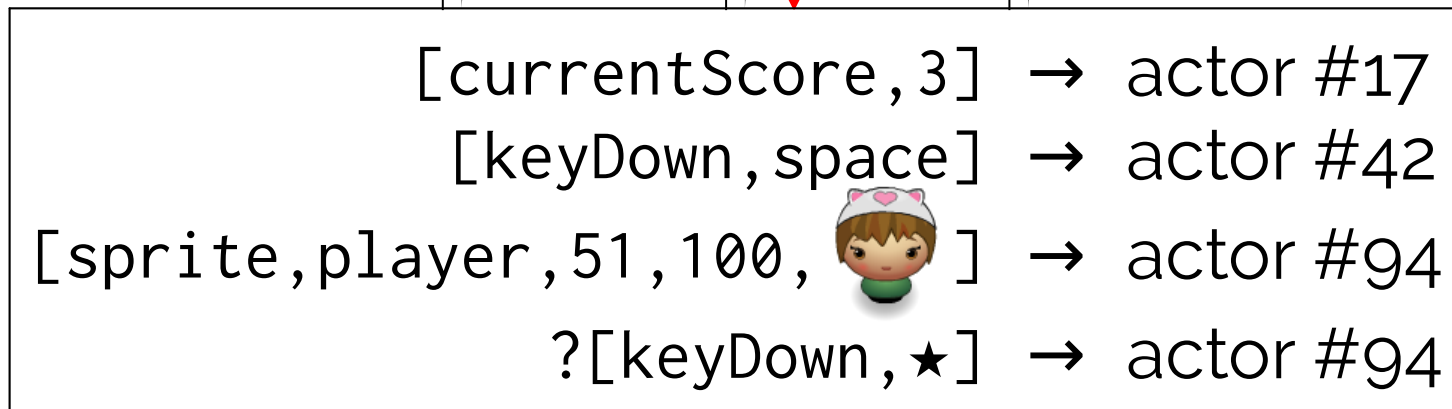
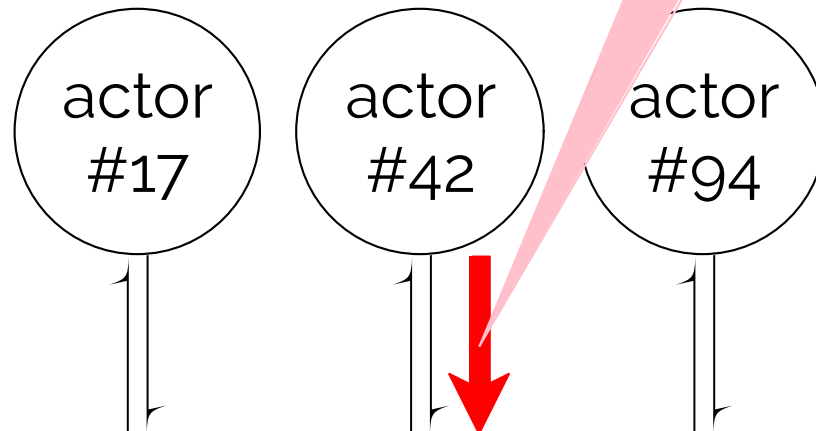


event × state → [action] × state



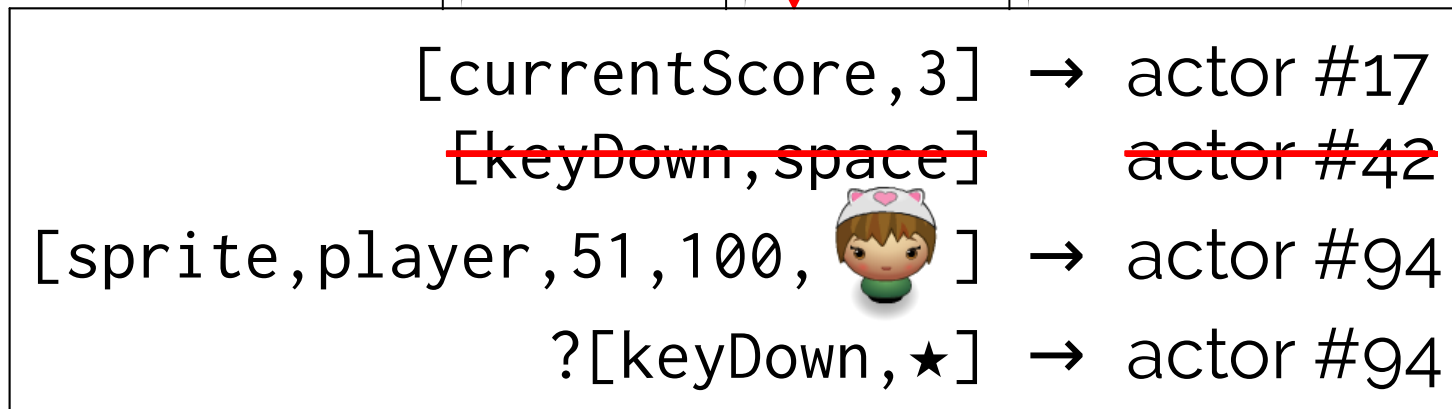
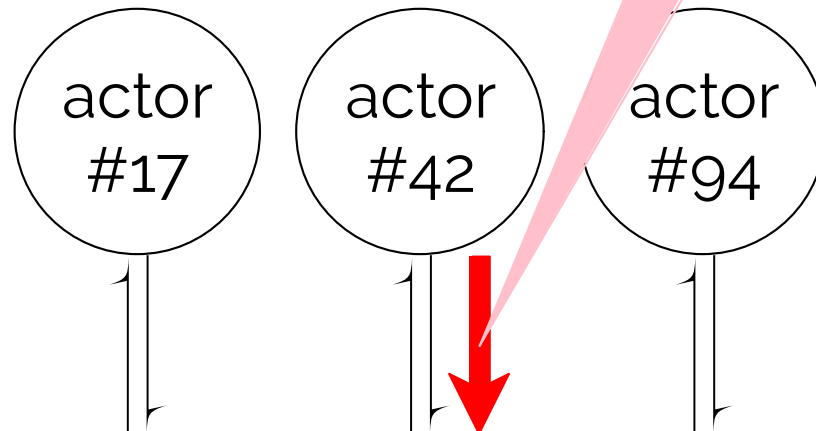
event × state → [action] × state

retract([keyDown, space])

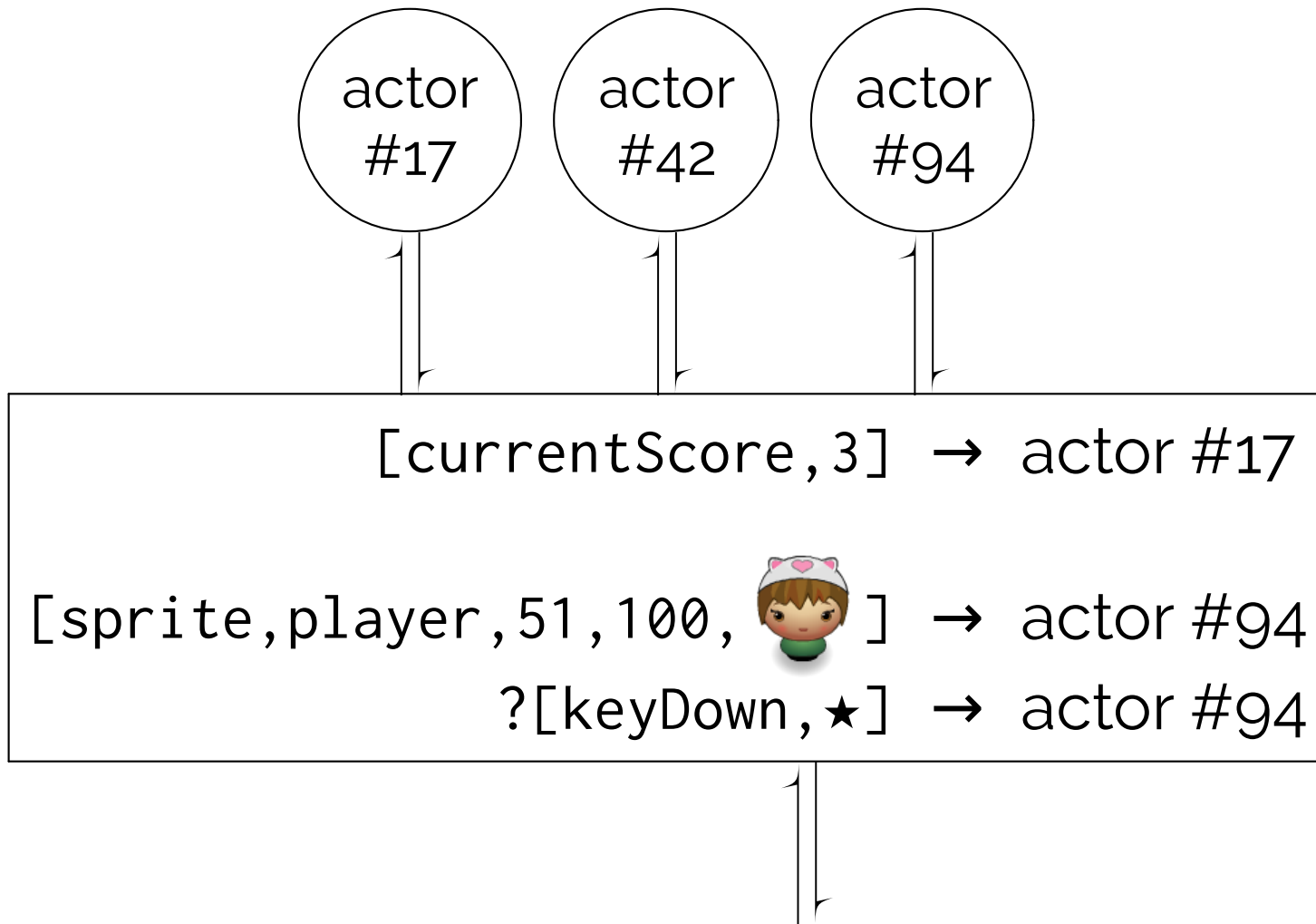


event × state → [action] × state

retract([keyDown, space])

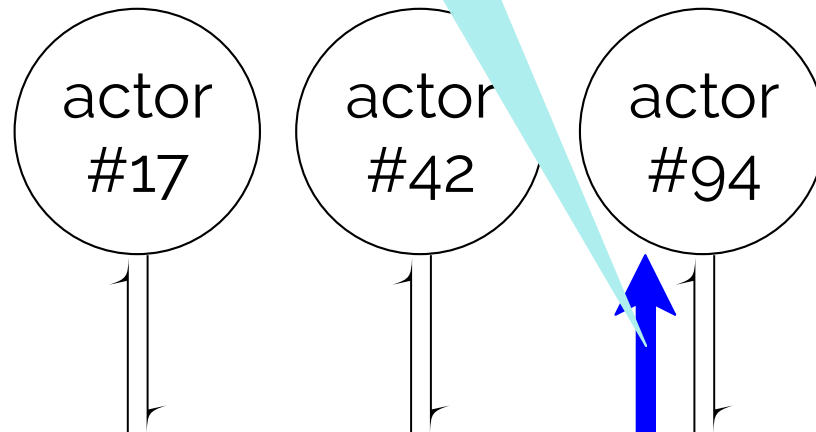


event × state → [action] × state




event × state → [action] × state

retract([keyDown, space])

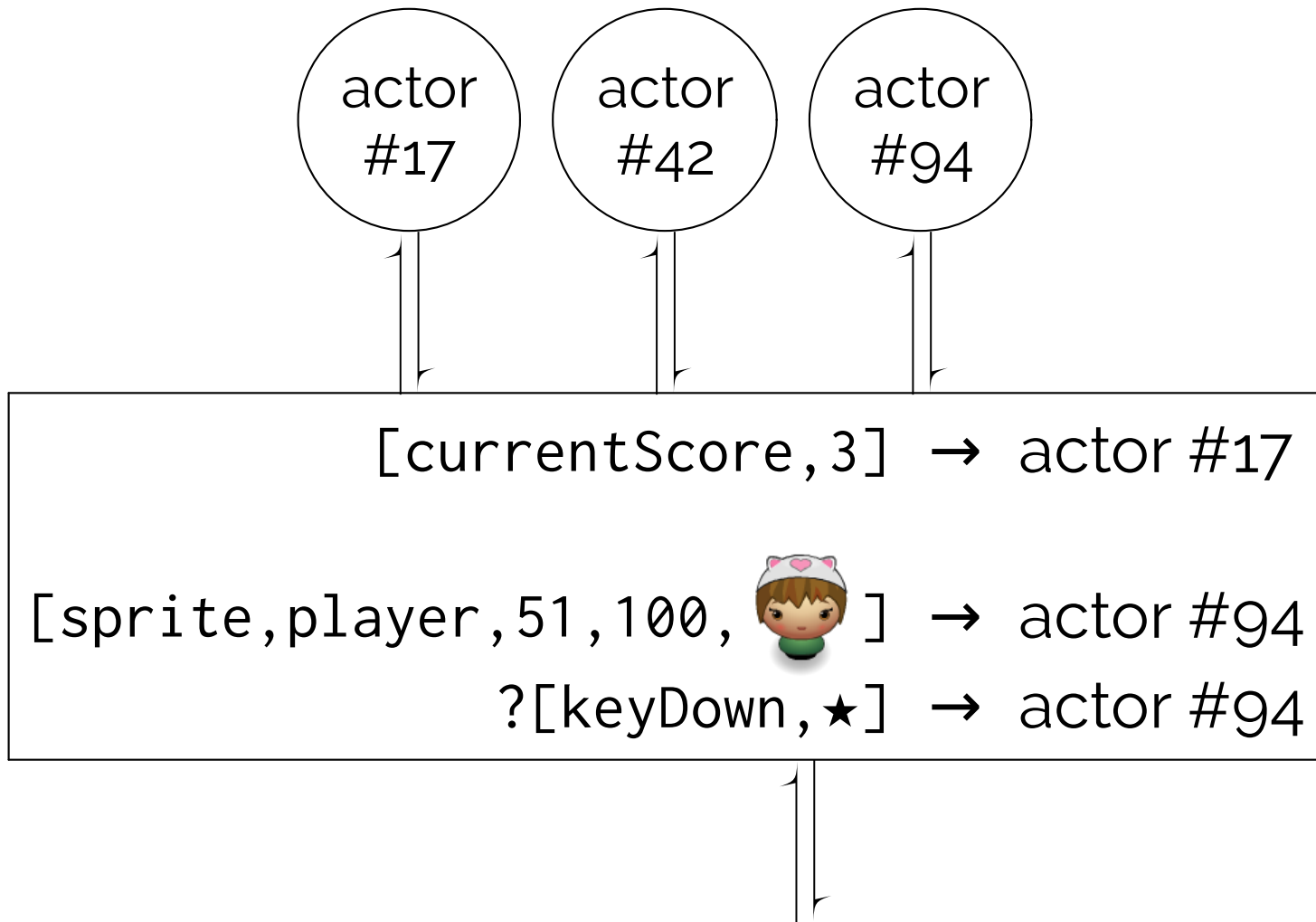


[currentScore, 3] → actor #17

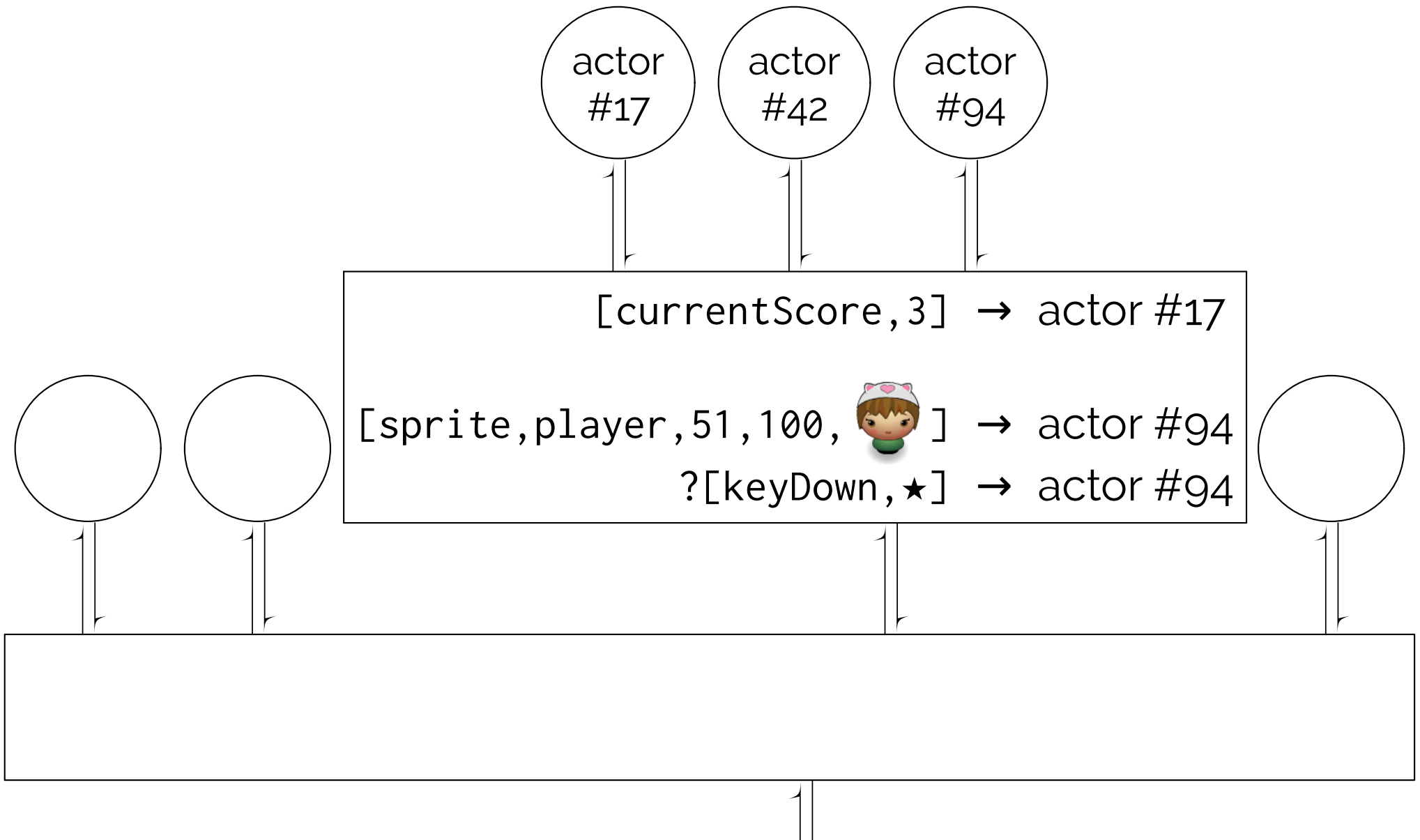
[sprite, player, 51, 100, ] → actor #94

?[keyDown, ★] → actor #94

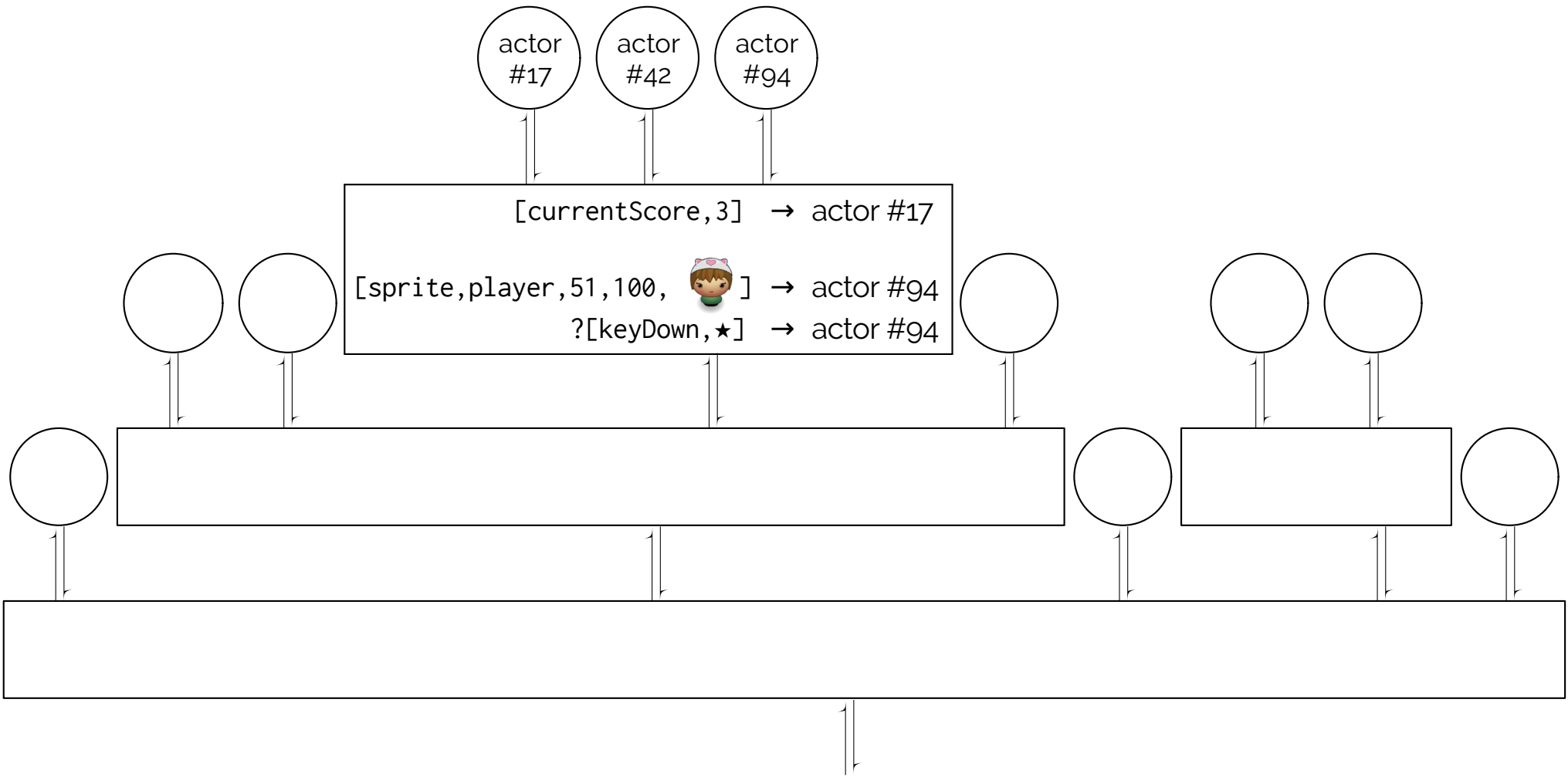
event × state → [action] × state



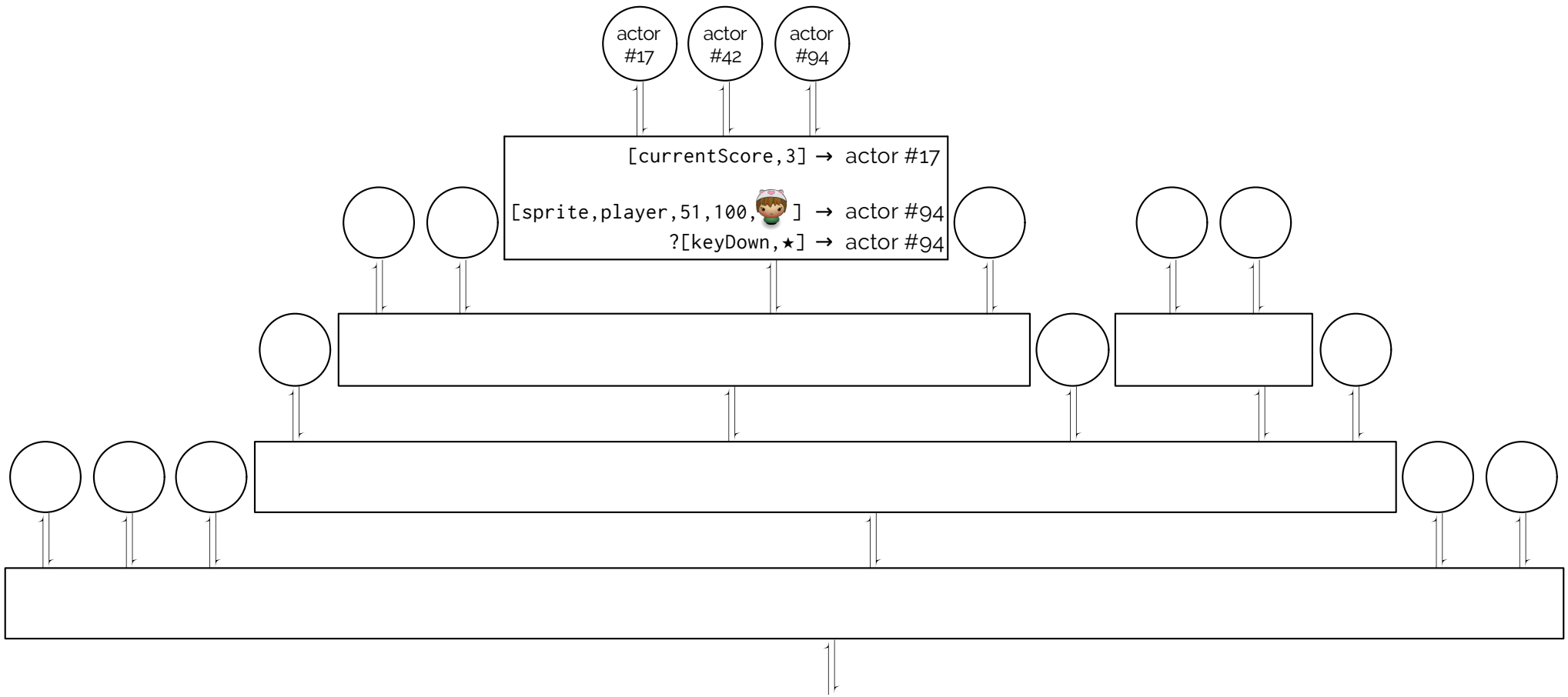
event × state → [action] × state



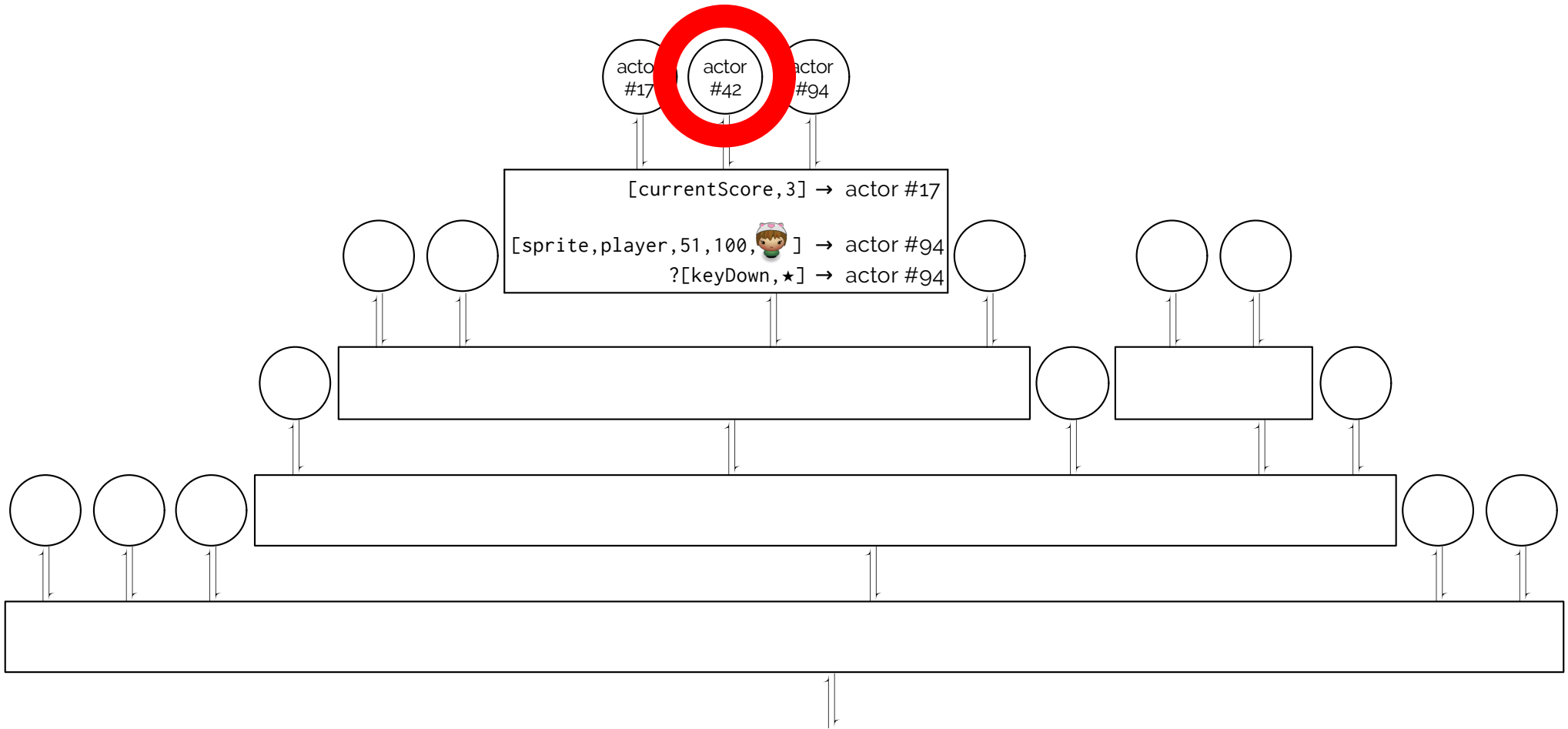
event × state → [action] × state



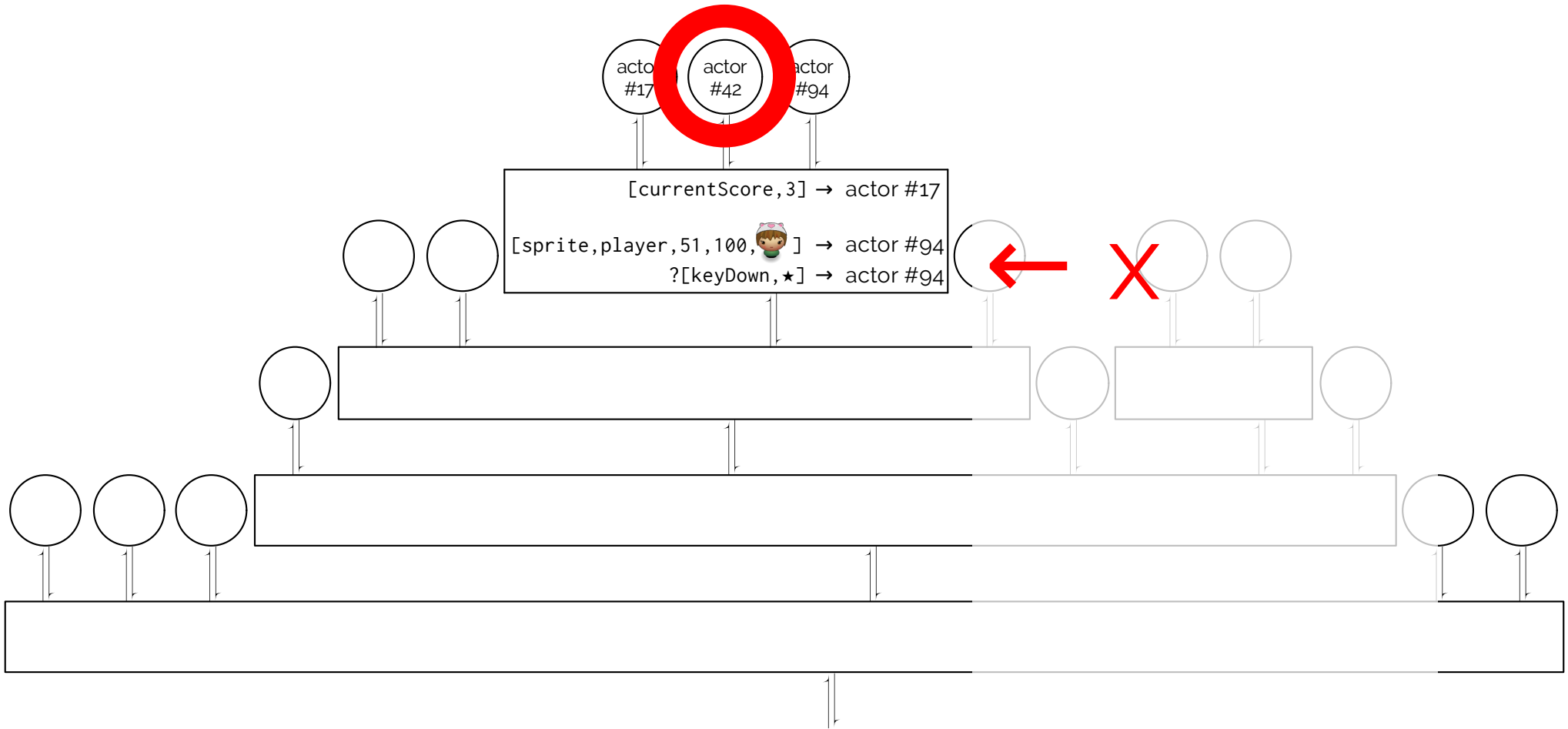
event × state → [action] × state



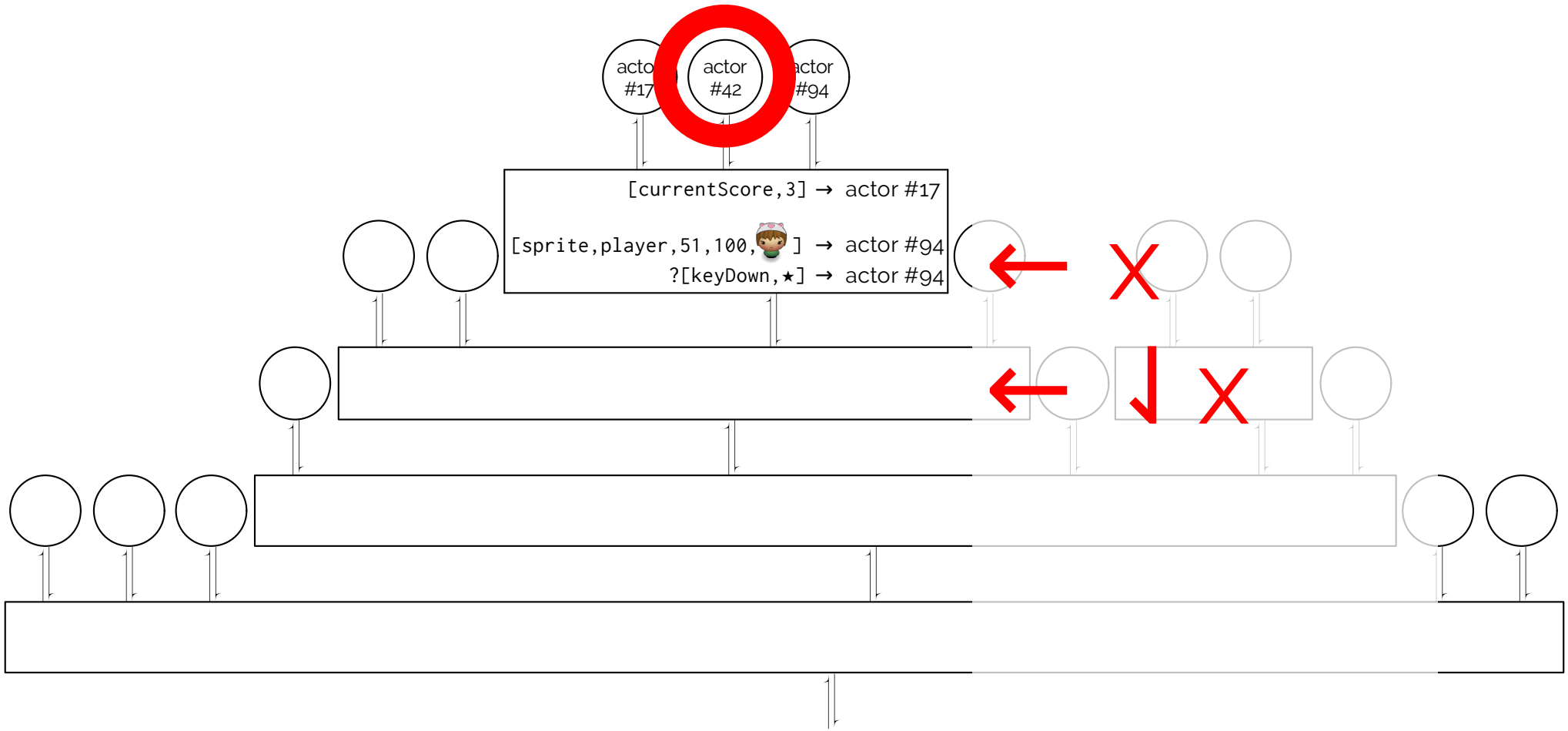
event × state → [action] × state



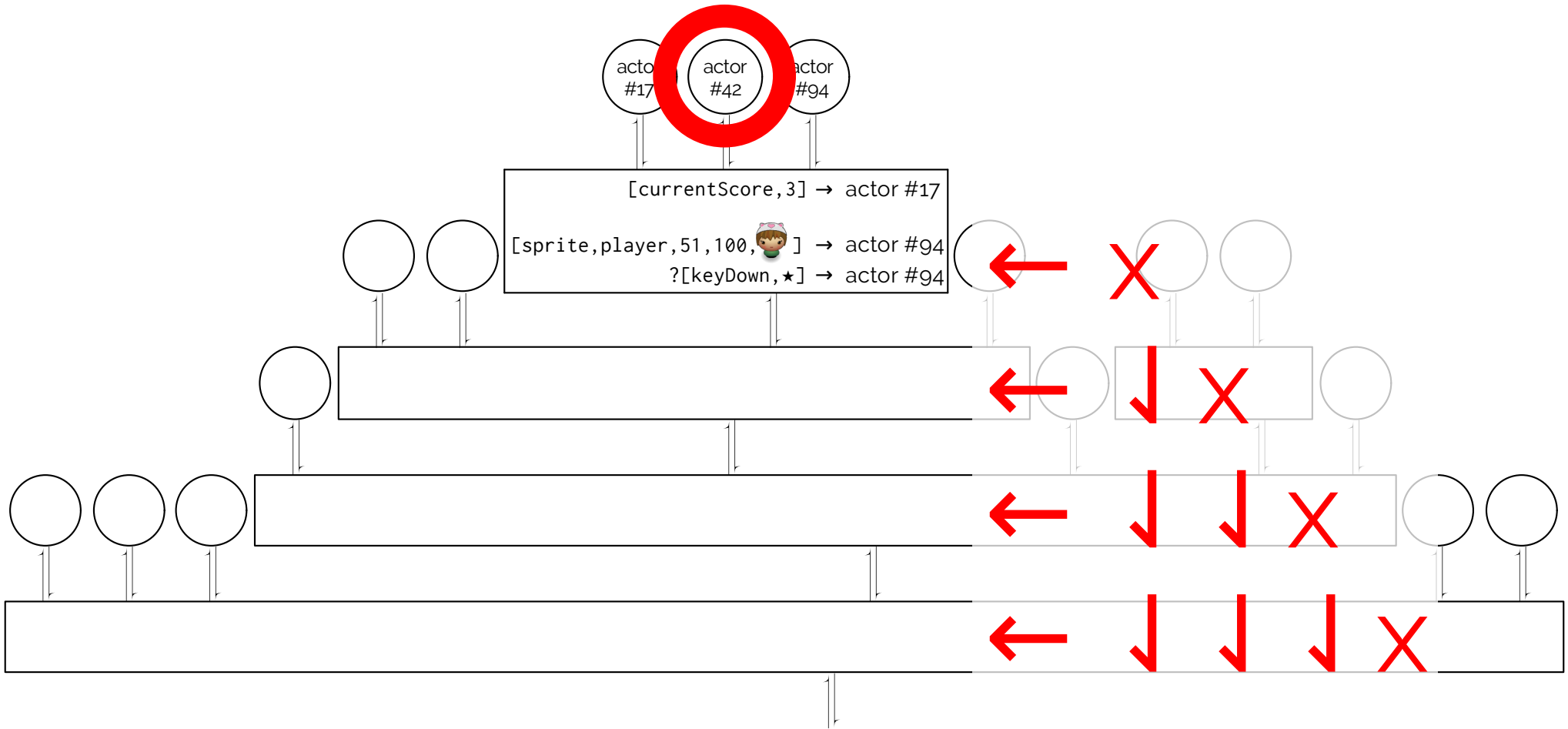
event × state → [action] × state



event × state → [action] × state



event × state → [action] × state



Messages are transient assertions

`< [incrementScoreBy, 3] >`

~

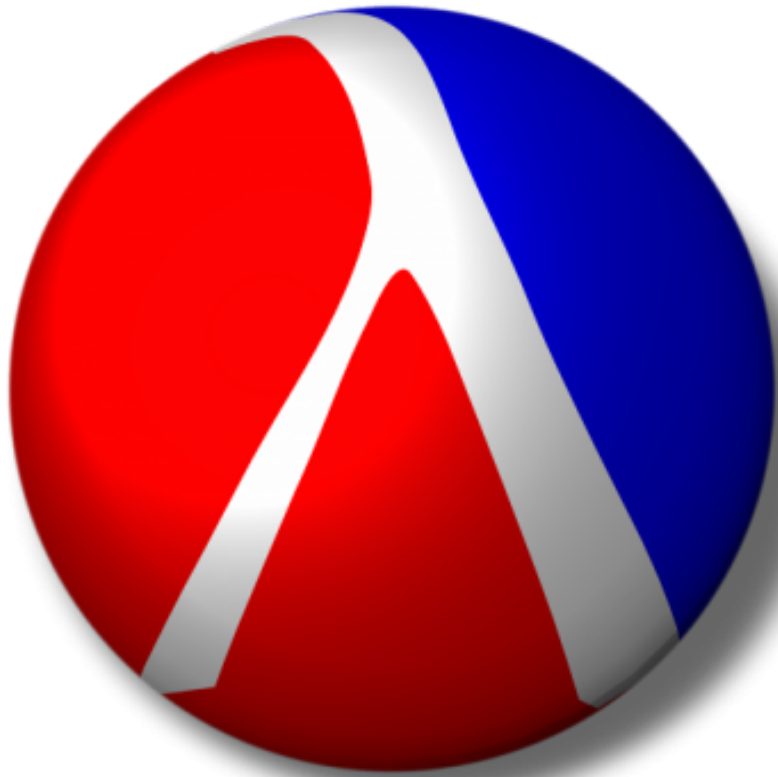
`assert([incrementScoreBy, 3])`

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`



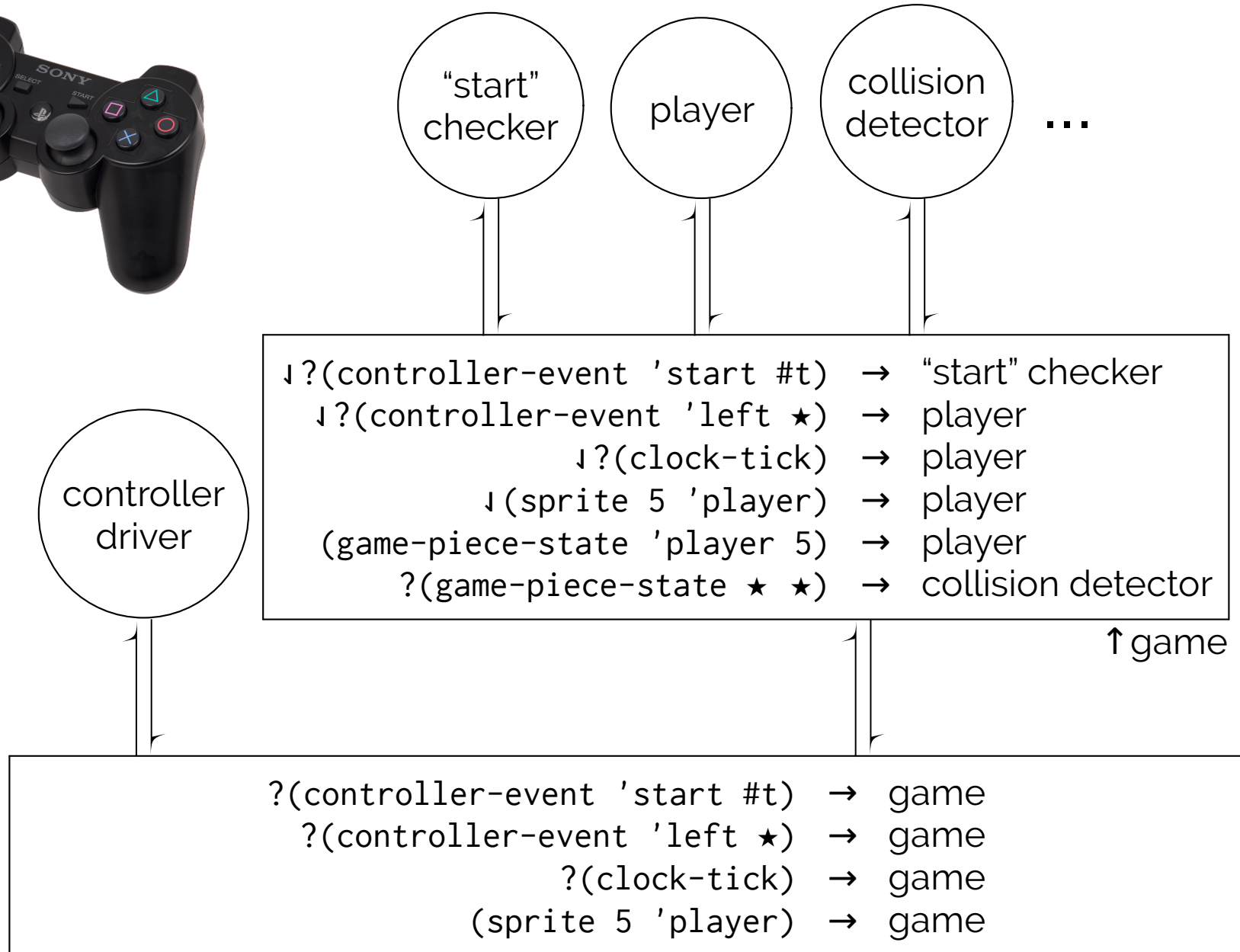
Ohm-based translation
to ECMAScript 5

Browser & node

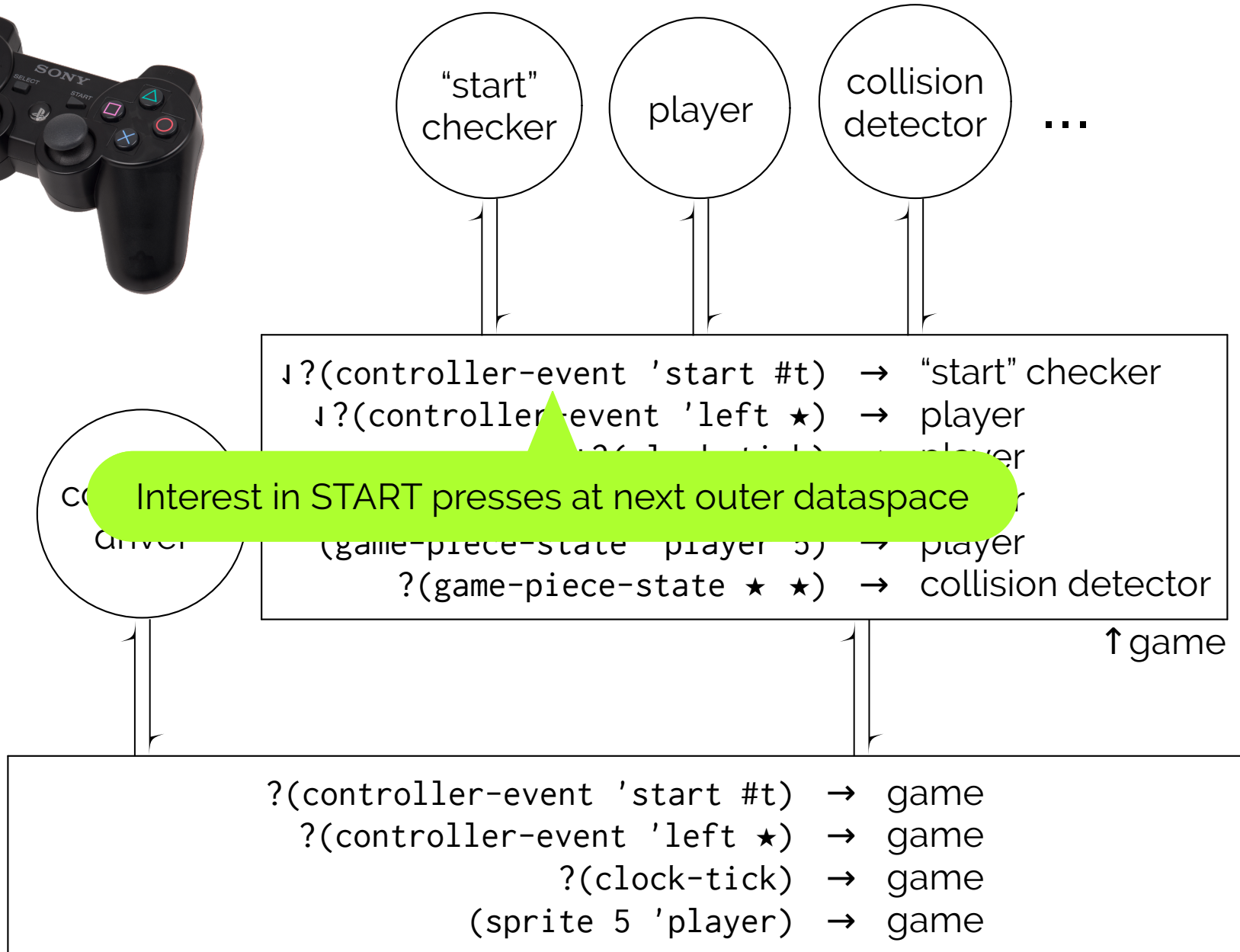
Syndicate DSL by example

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

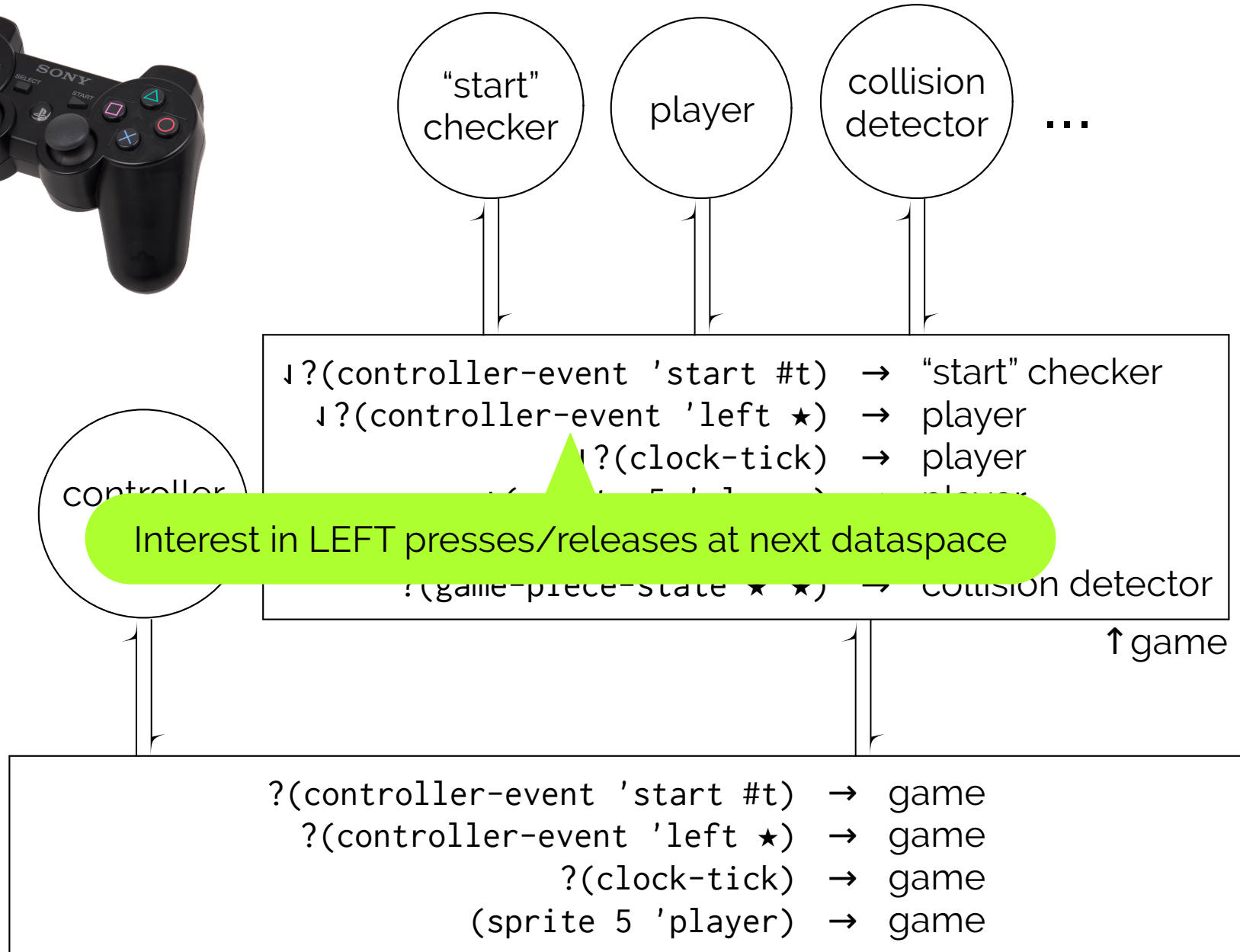
Mapping events to components



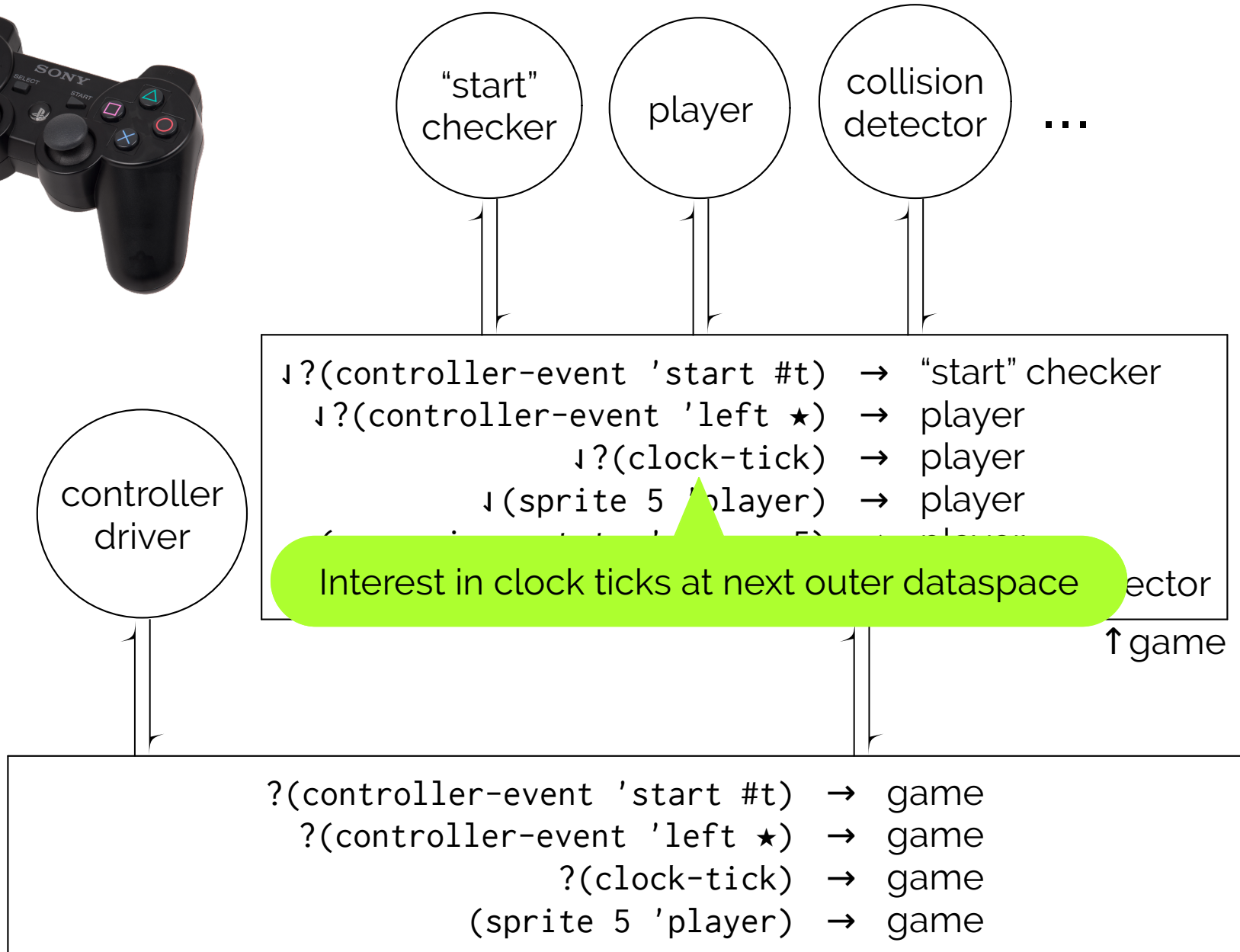
Mapping events to components



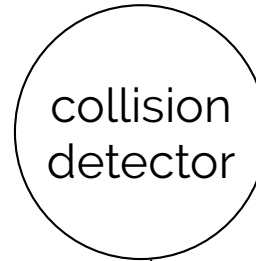
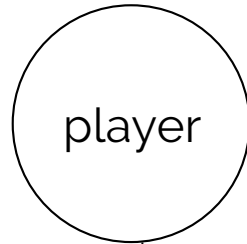
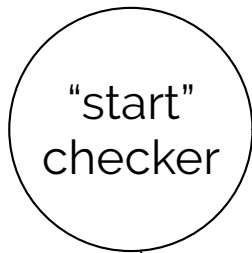
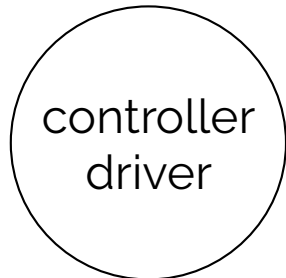
Mapping events to components



Mapping events to components



Mapping events to components



...

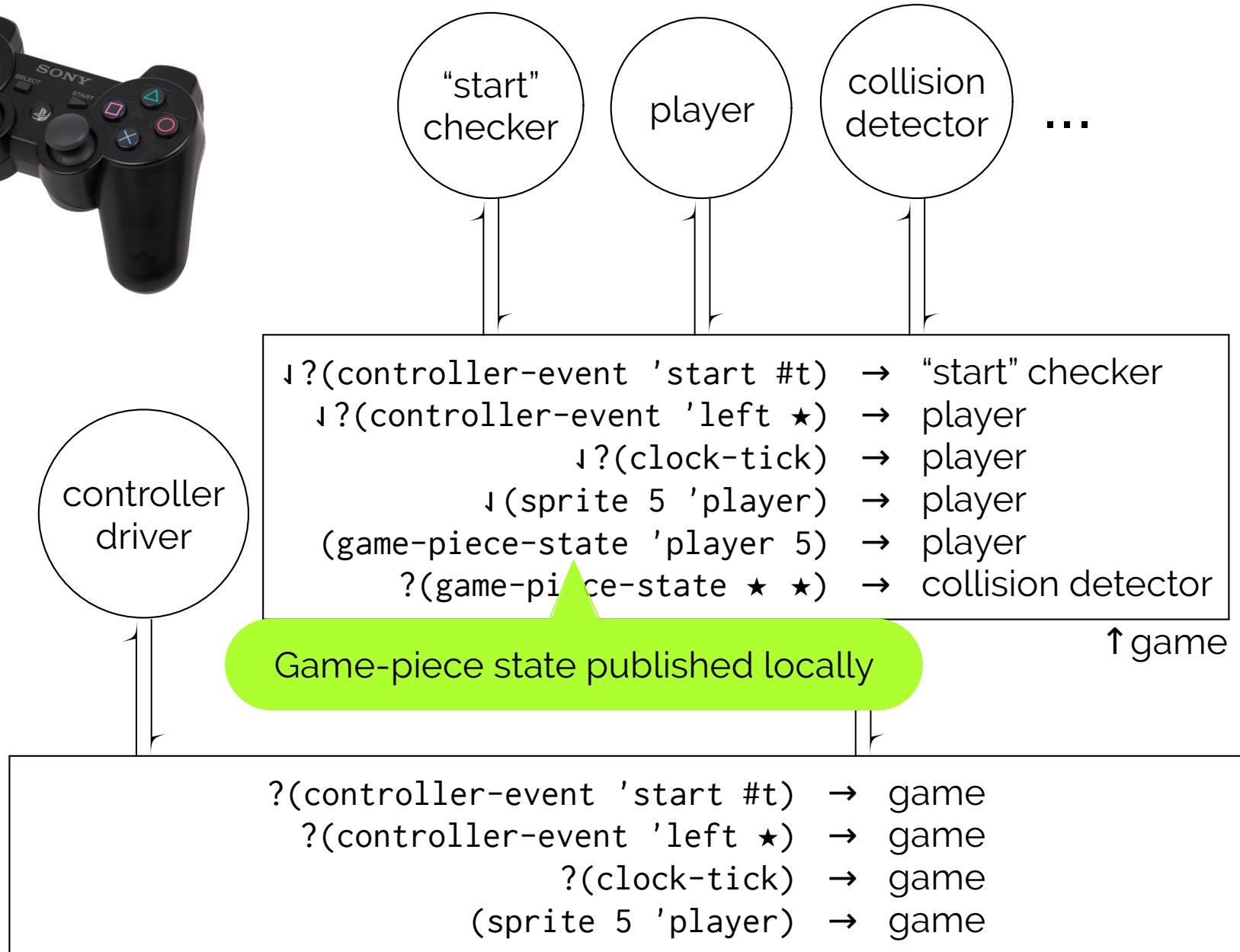
```
↓?(controller-event 'start #t) → "start" checker
↓?(controller-event 'left ★) → player
    ↓?(clock-tick) → player
        ↓(sprite 5 'player) → player
            (game-piece-state player 5) → player
                (collision-detector) → collision detector
```

Sprite published to next outer dataspace

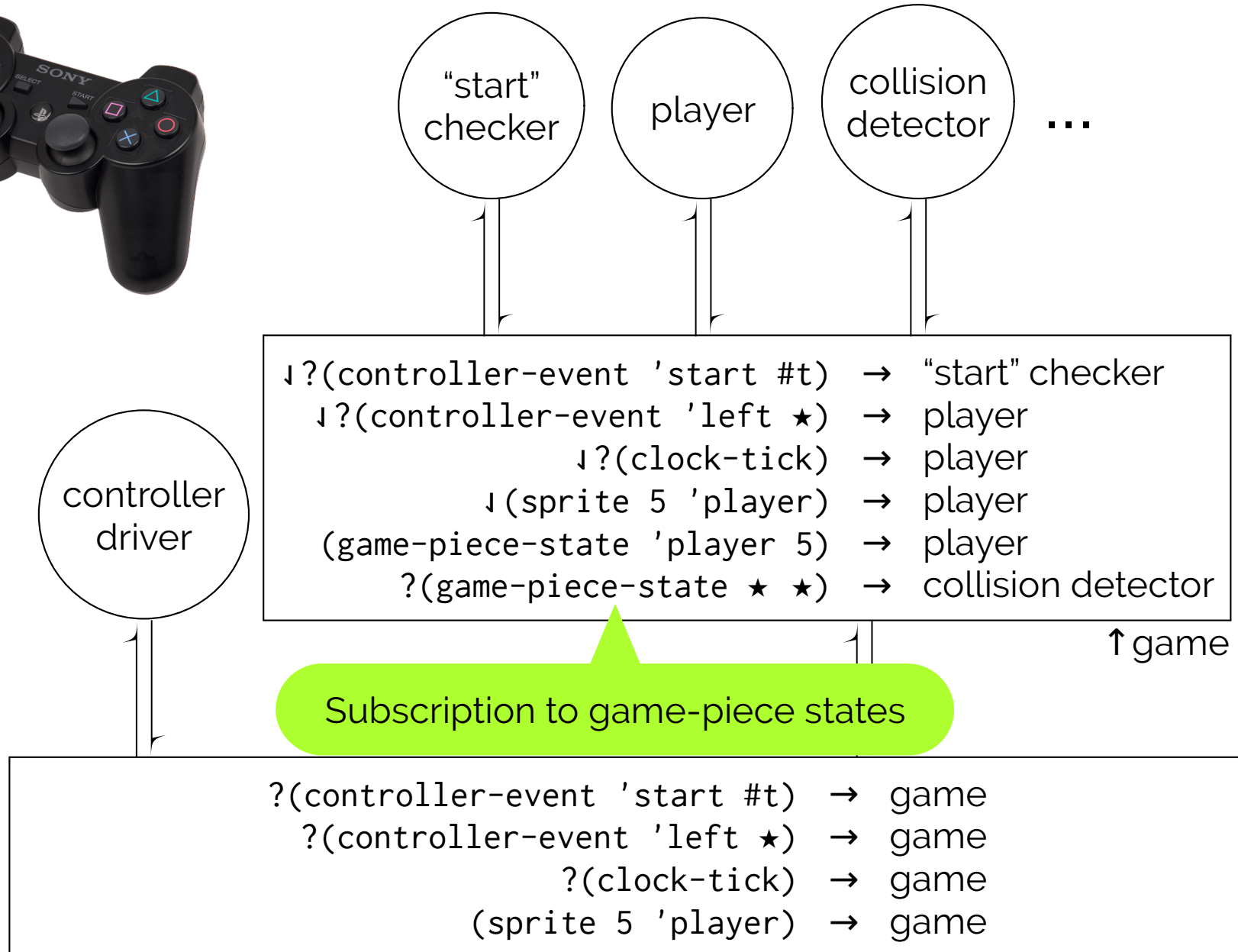
↑ game

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

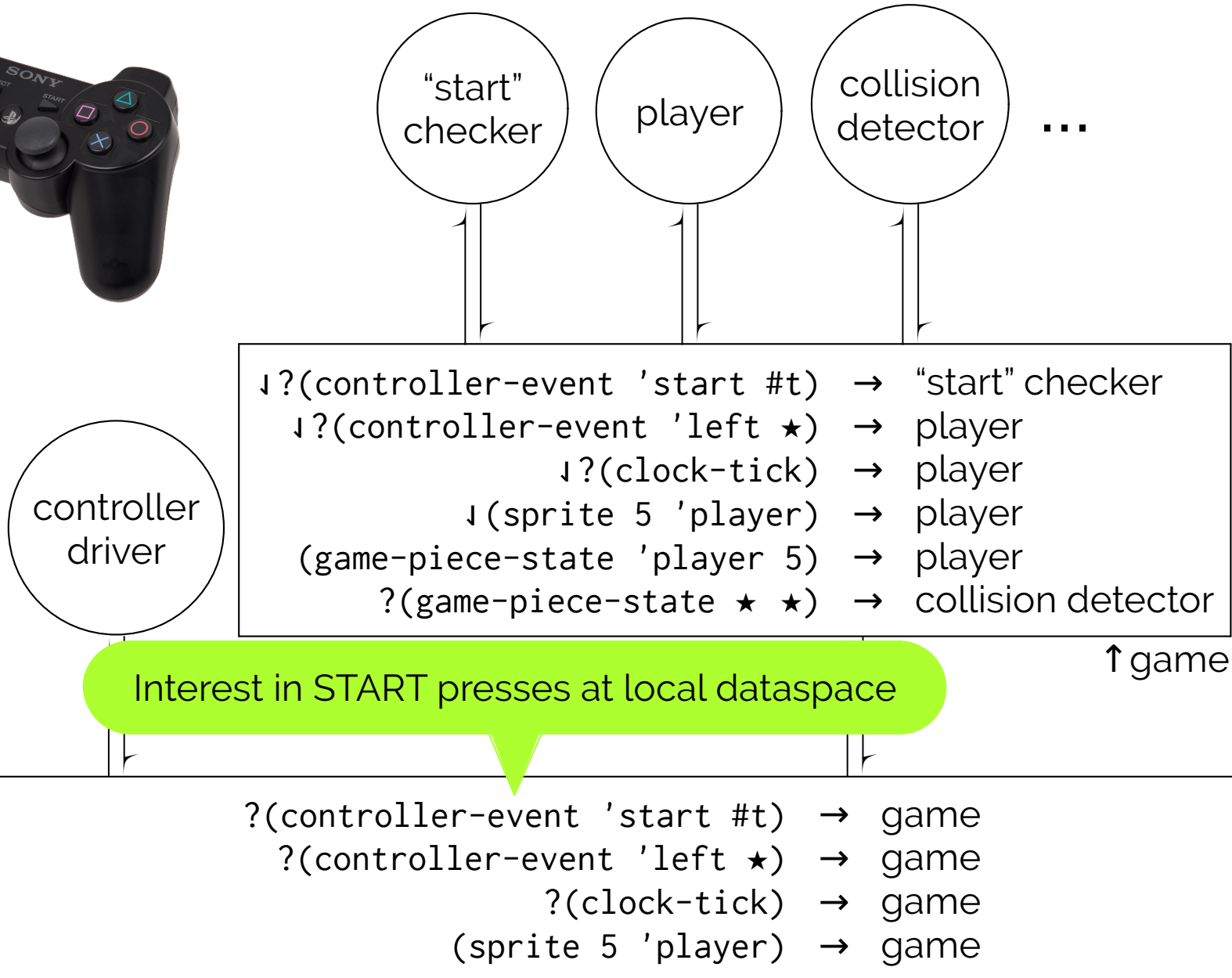
Mapping events to components



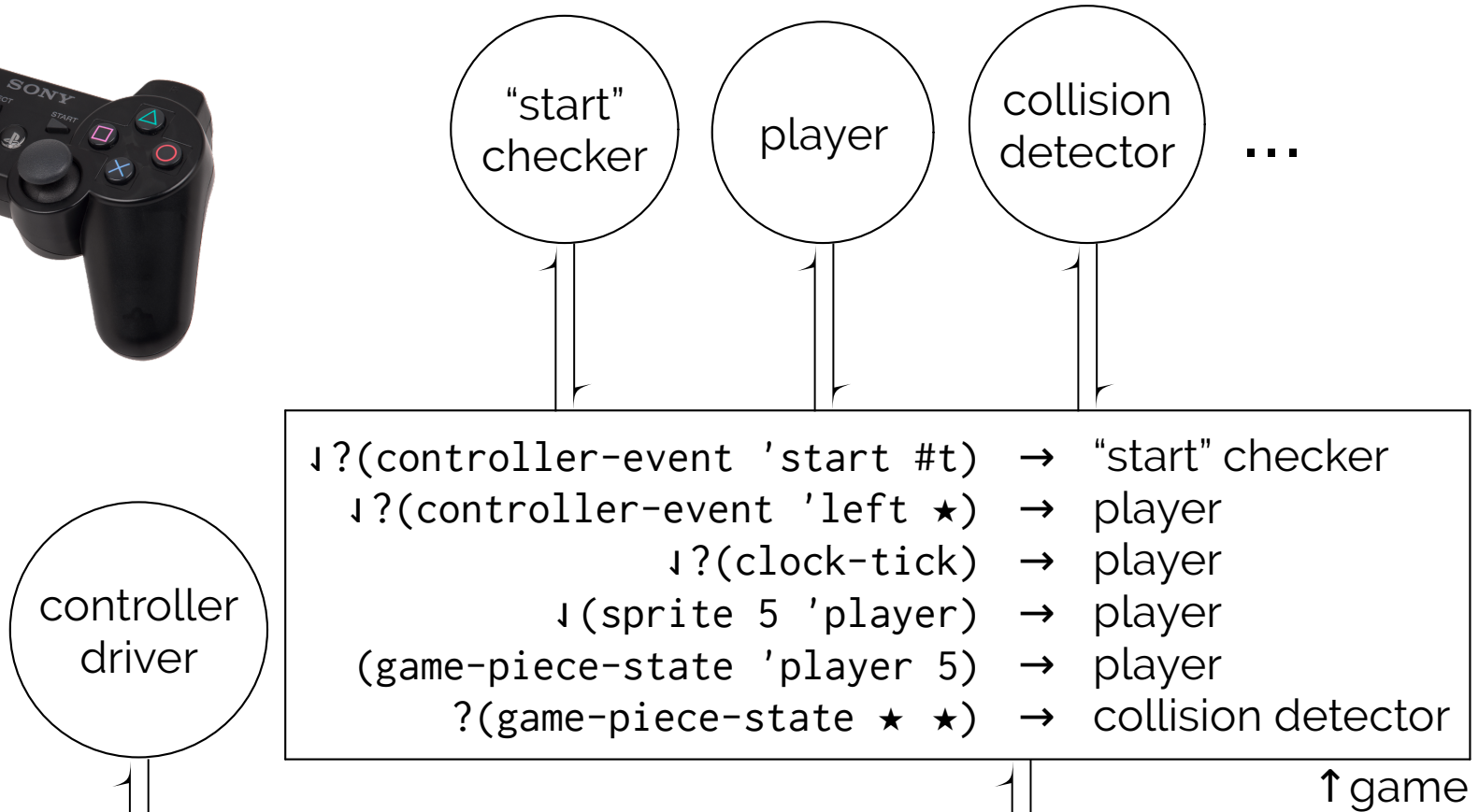
Mapping events to components



Mapping events to components



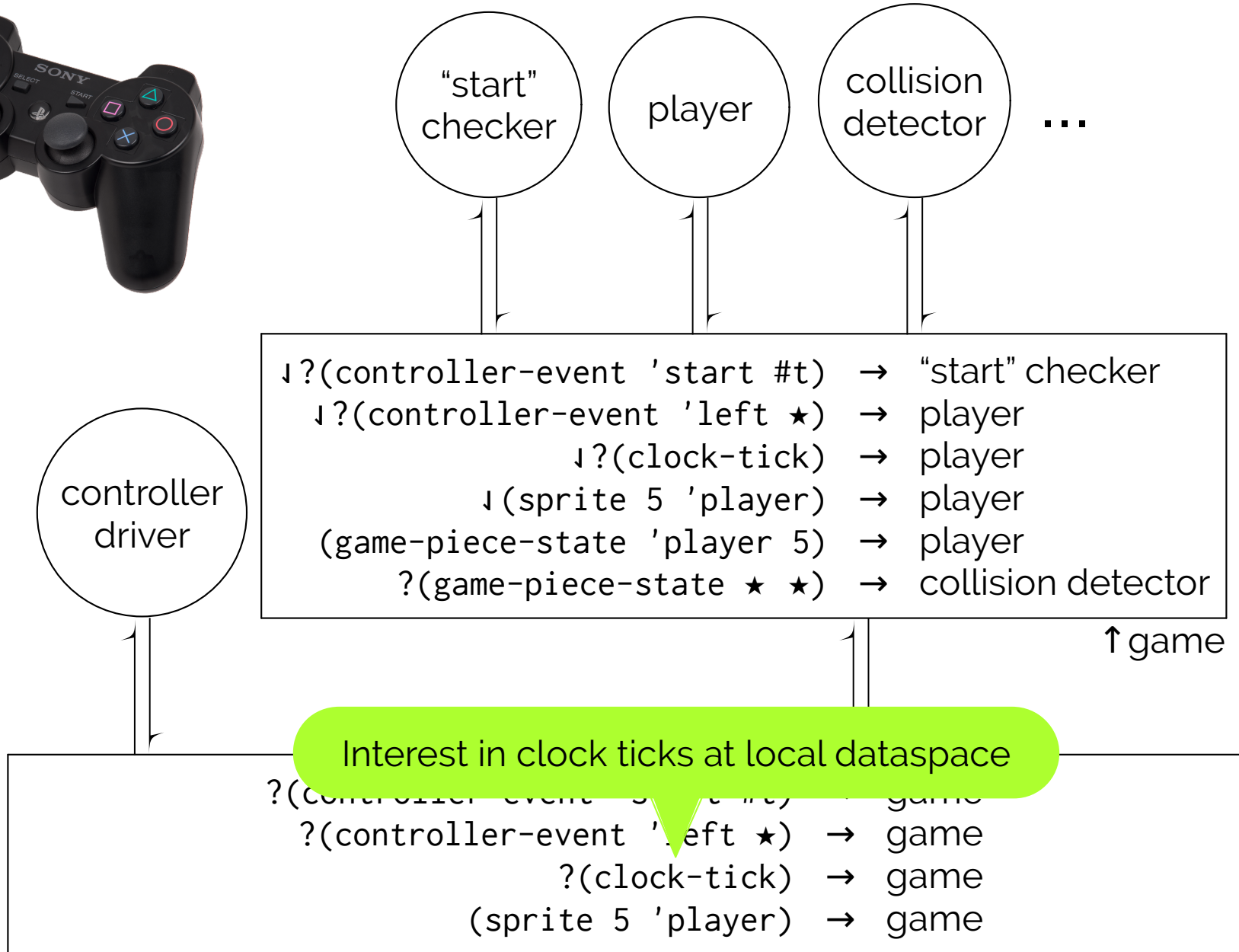
Mapping events to components



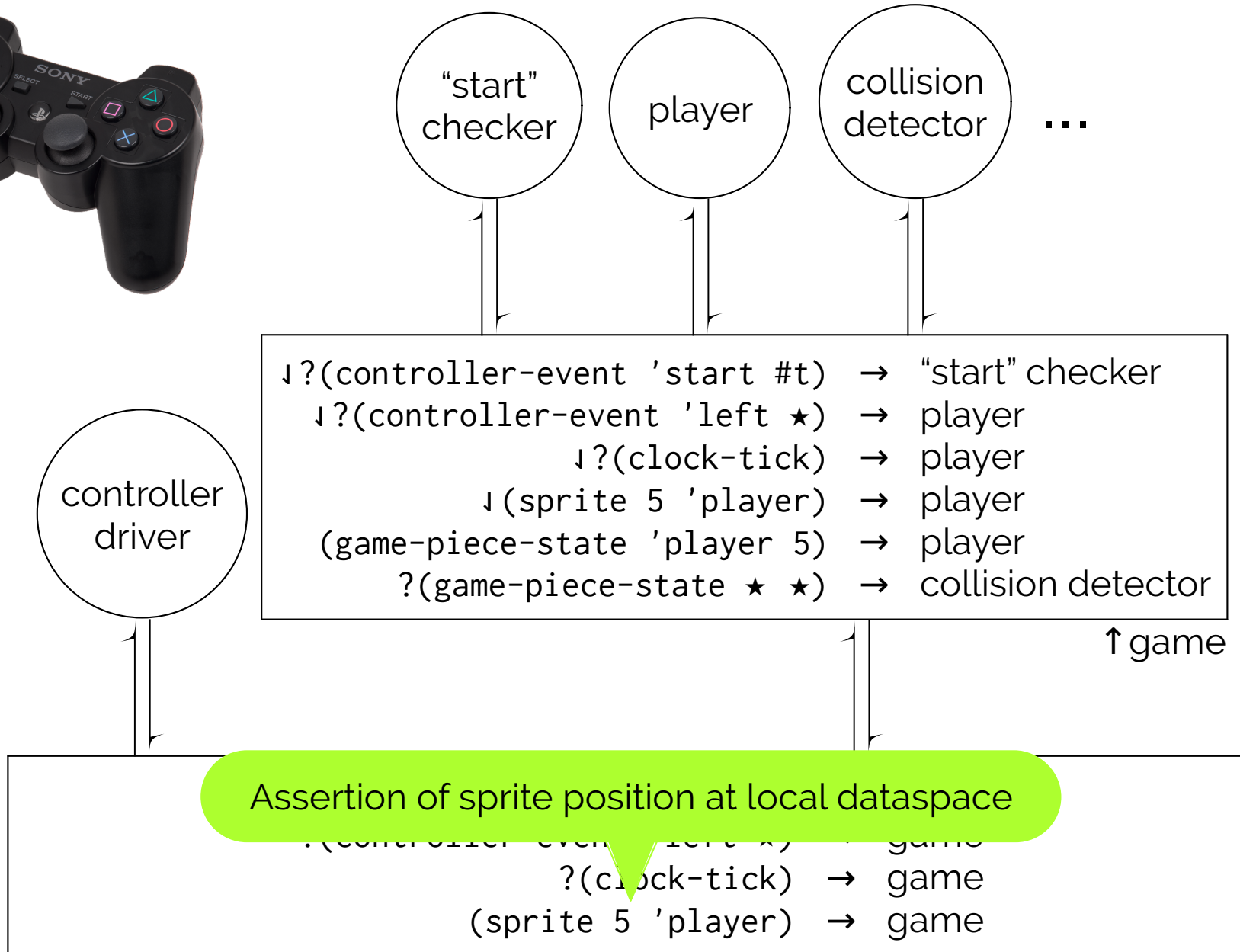
Interest in LEFT presses/releases at local dataspace

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

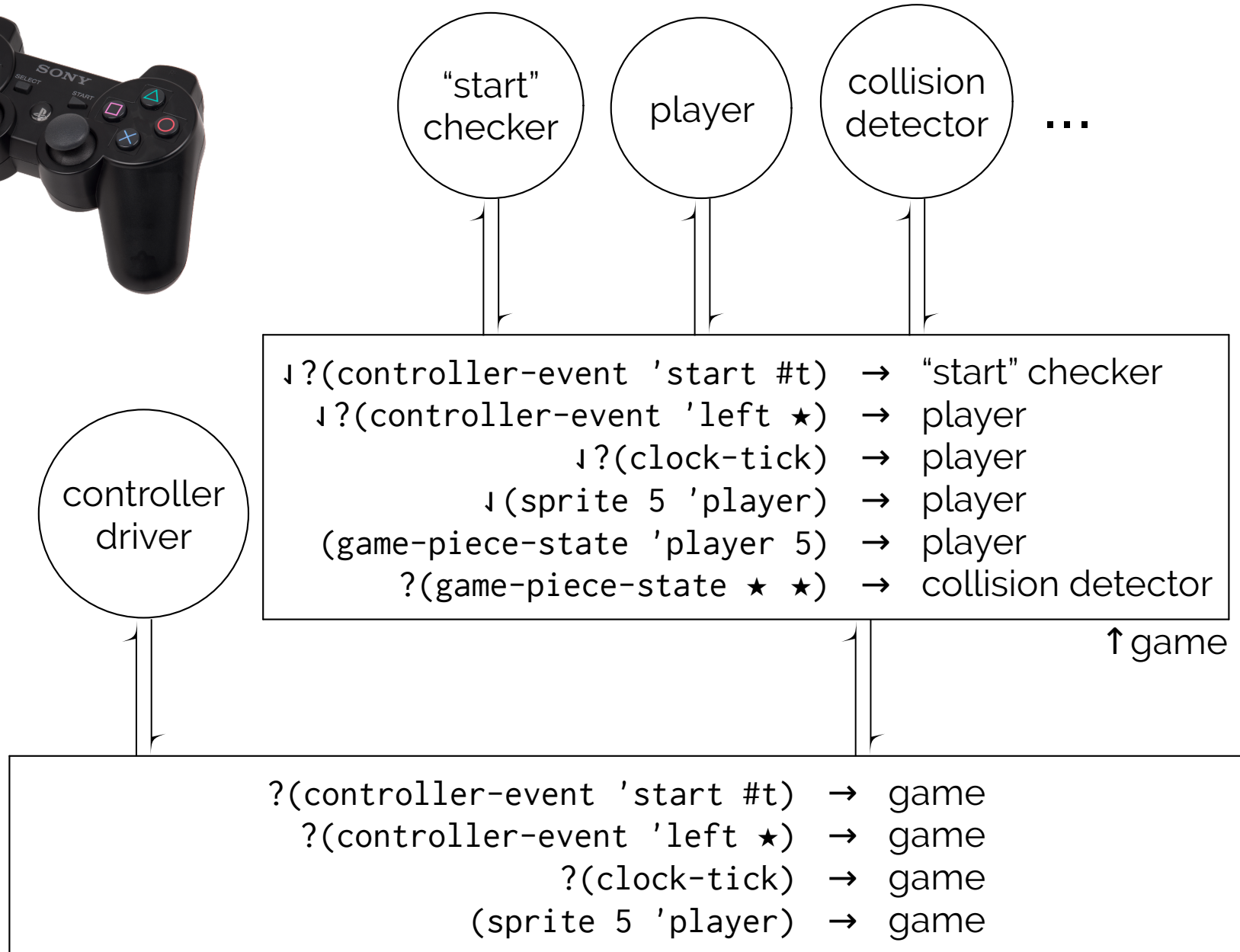
Mapping events to components



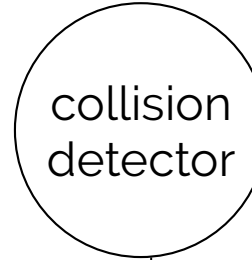
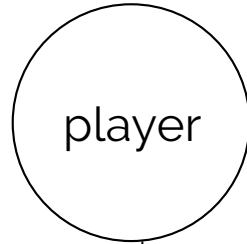
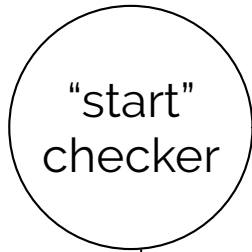
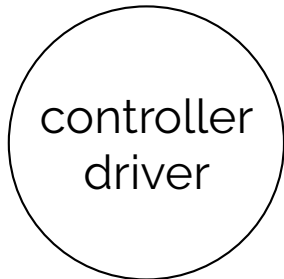
Mapping events to components



Mapping events to components



Mapping events to components

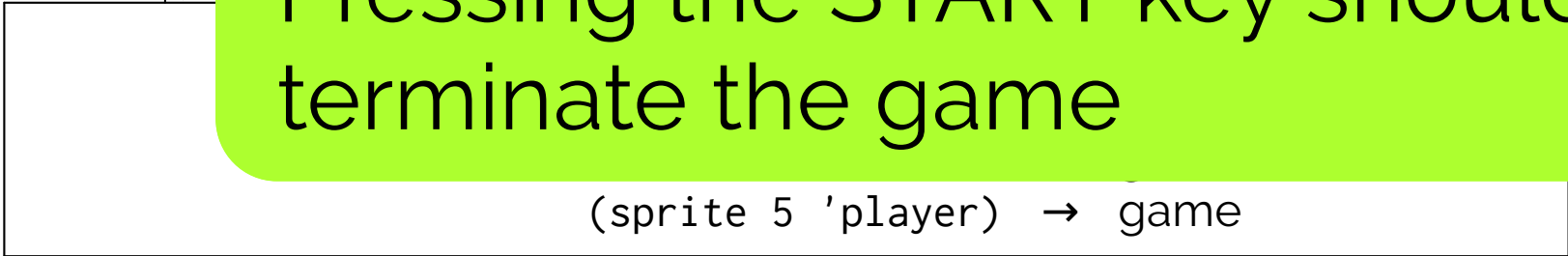


...



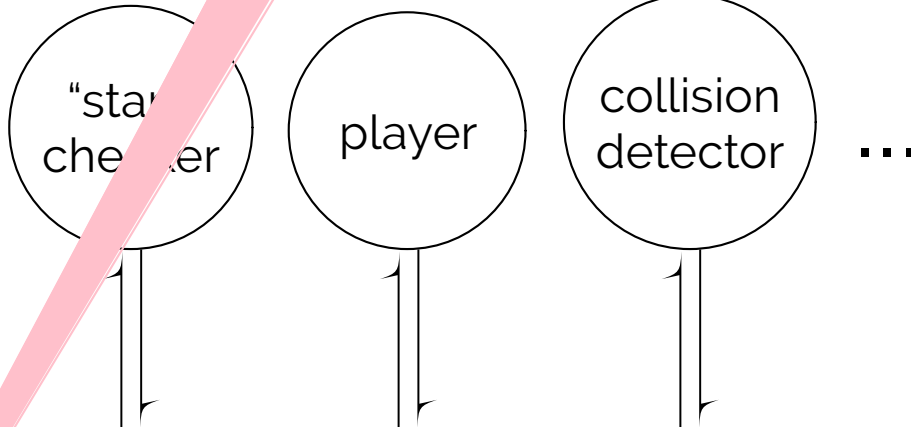
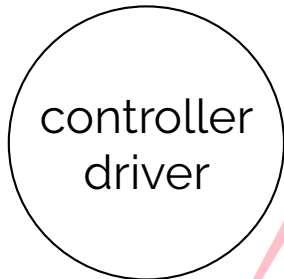
↑ game

Pressing the START key should terminate the game

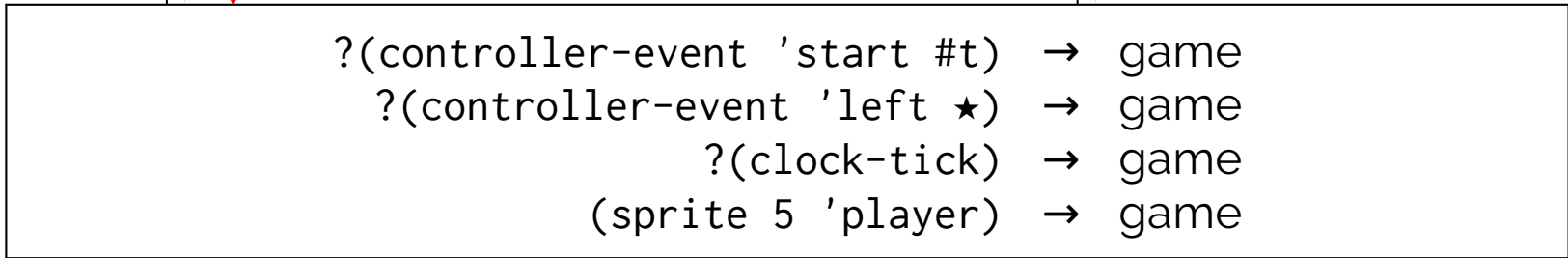


Mapping events to components

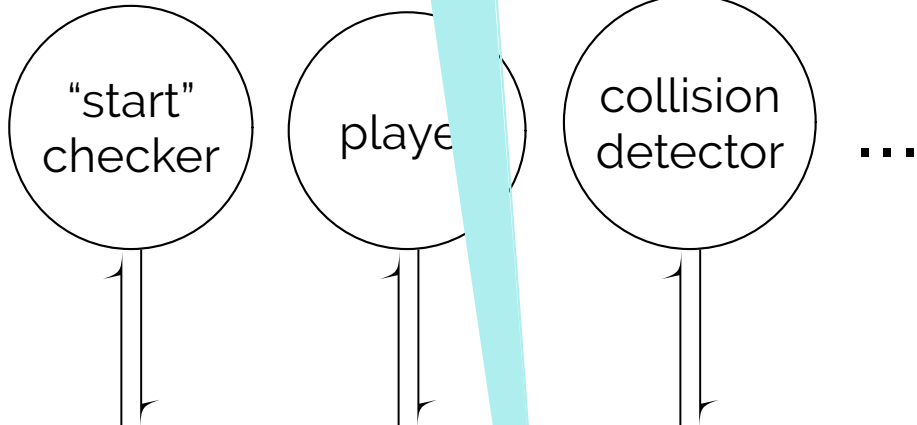
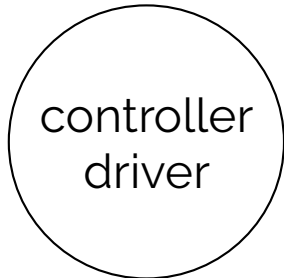
< (controller-event 'start #t) >



↑ game



< (controller-event 'start #t) >



```

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

```

↑ game

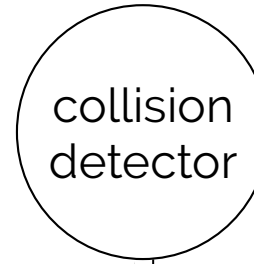
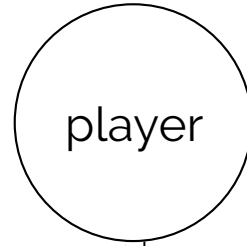
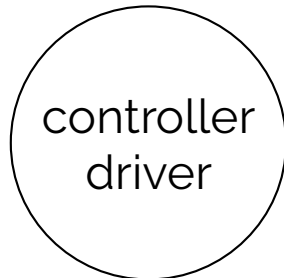
```

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

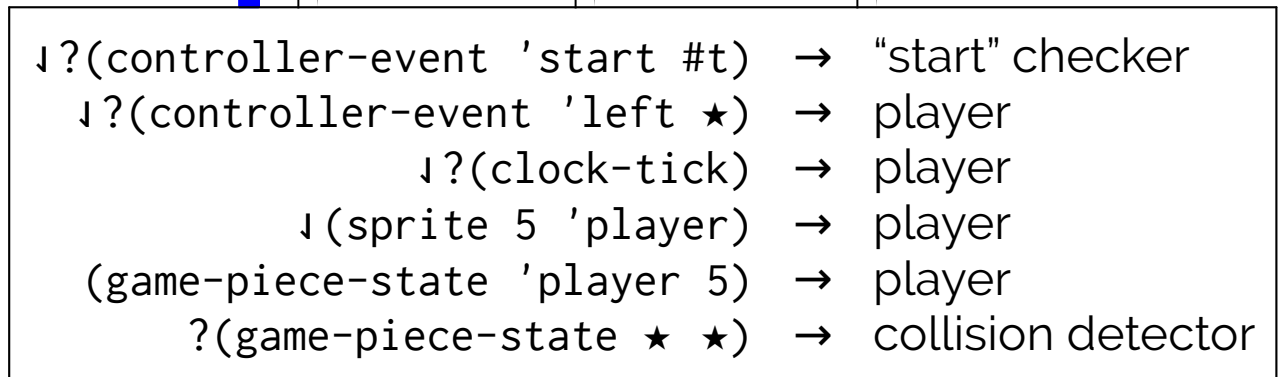
```



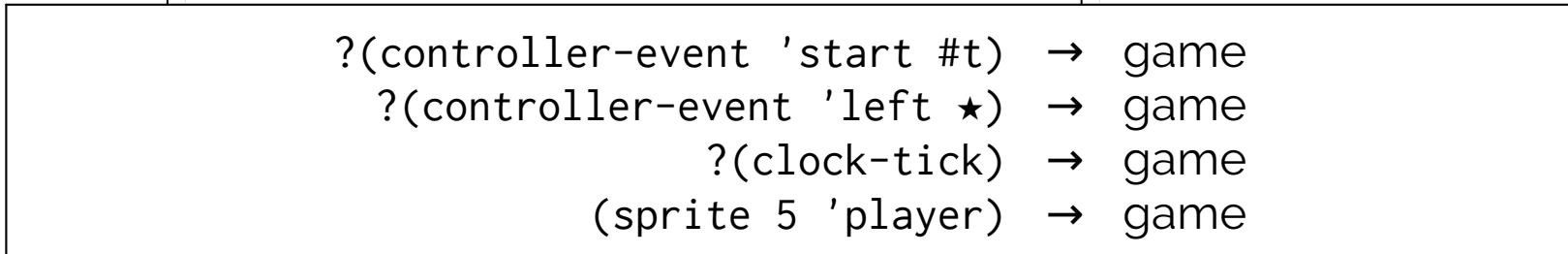
< ↓(controller-event 'start #t) >



...

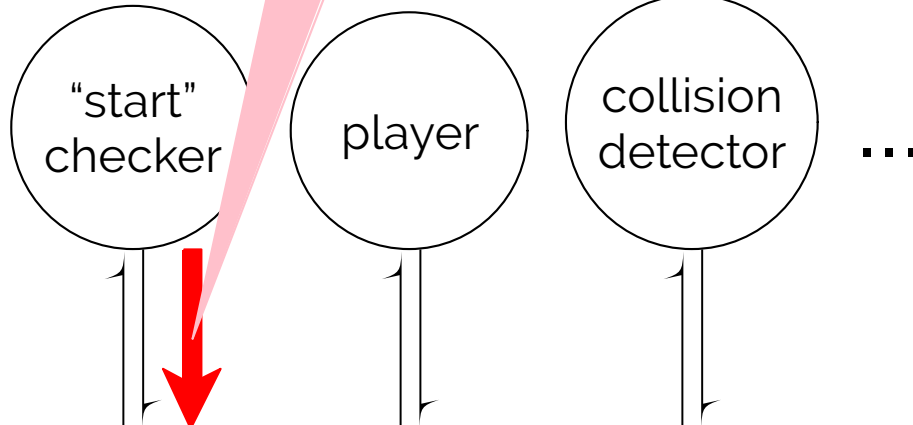
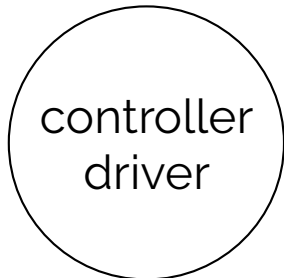


↑ game



Mapping to game elements

quit-dataspace!

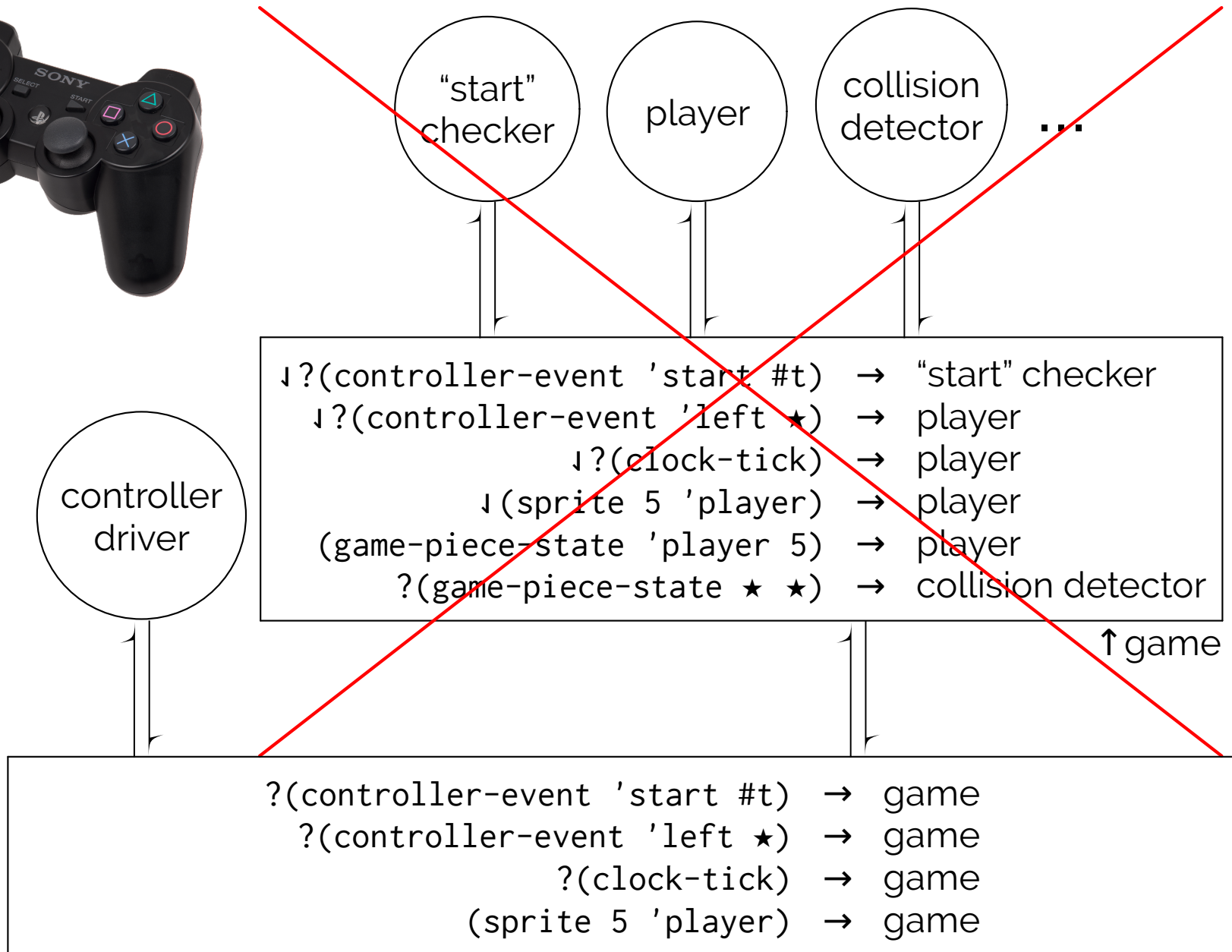


```
↓?(controller-event 'start #t) → "start" checker
↓?(controller-event 'left ★) → player
    ↓?(clock-tick) → player
        ↓(sprite 5 'player) → player
(game-piece-state 'player 5) → player
?(game-piece-state ★ ★) → collision detector
```

↑ game

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

Mapping events to components



Mapping events to components



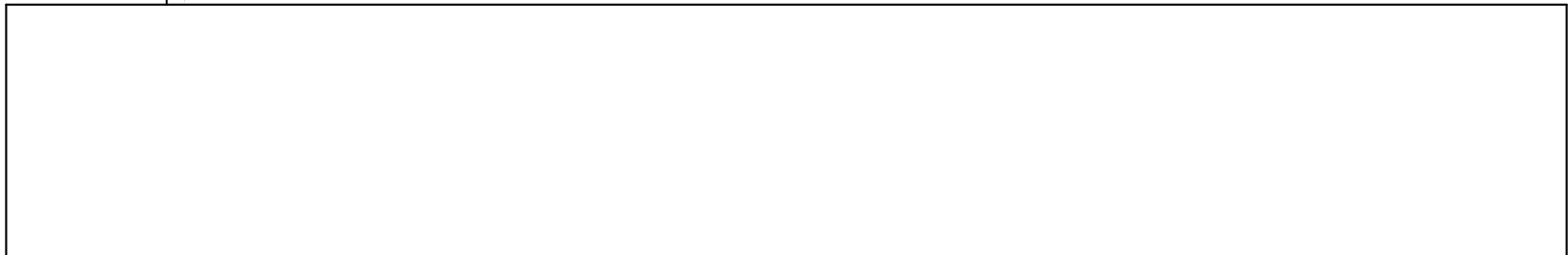
controller
driver

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

Mapping events to components



controller
driver



```
(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*

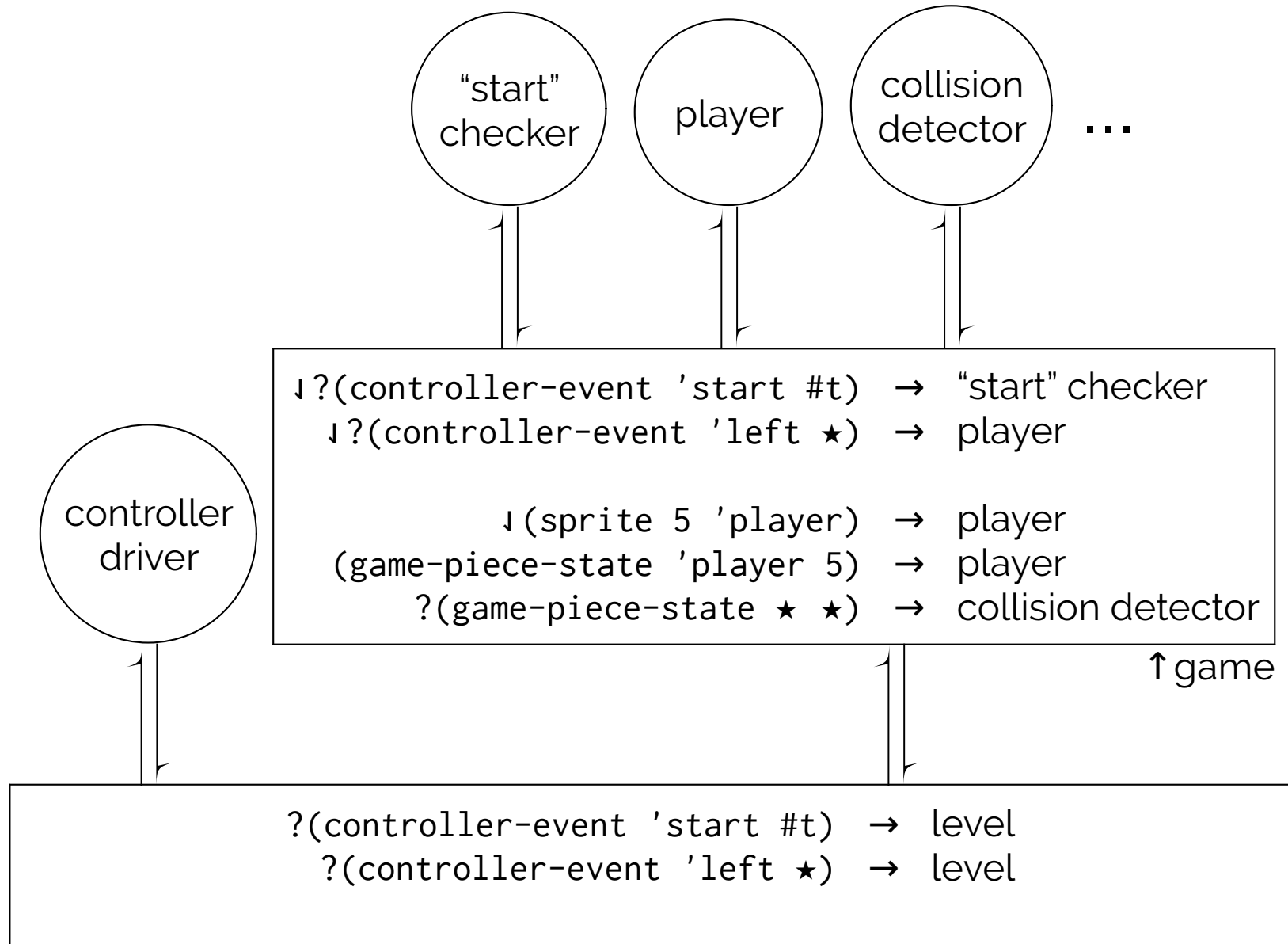
```
(dataspace (spawn-player)
  (spawn-collision-detection)
  ...
  (until (message (controller-event 'start #t)
    #:meta-level 1))))
```

metalevel number written *once*, in one style

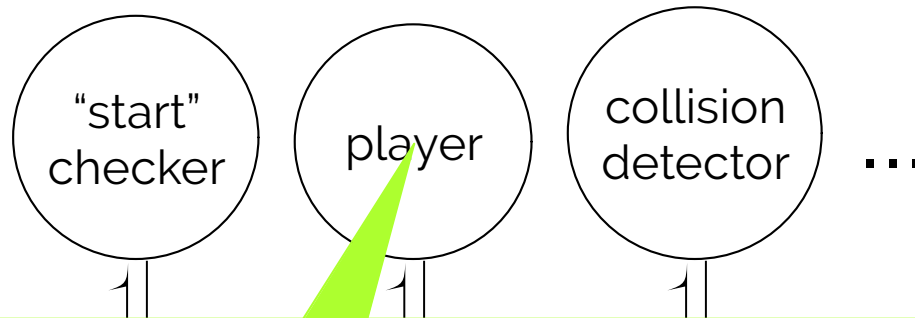
Syndicate DSL by example

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

Managing conversational state

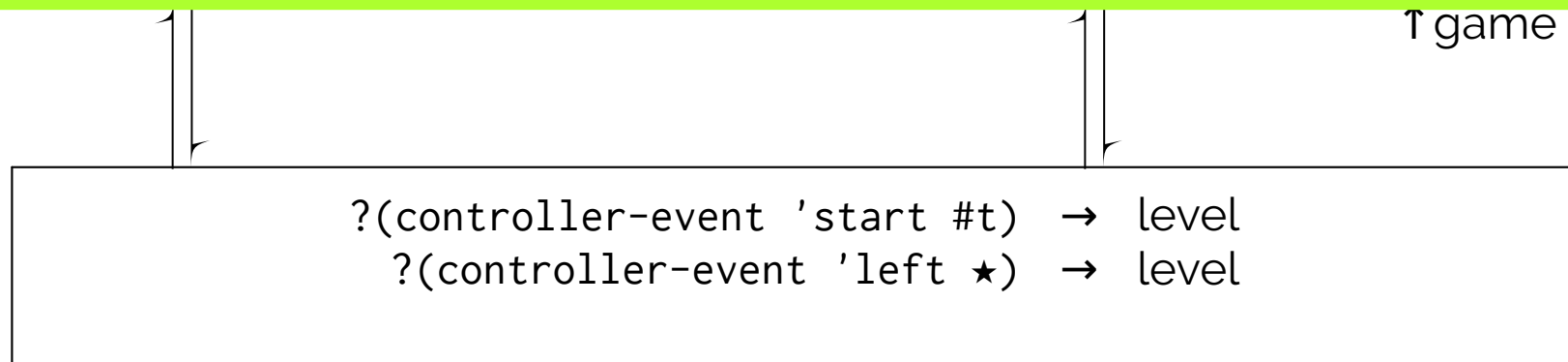


Managing conversational state



Three jobs:

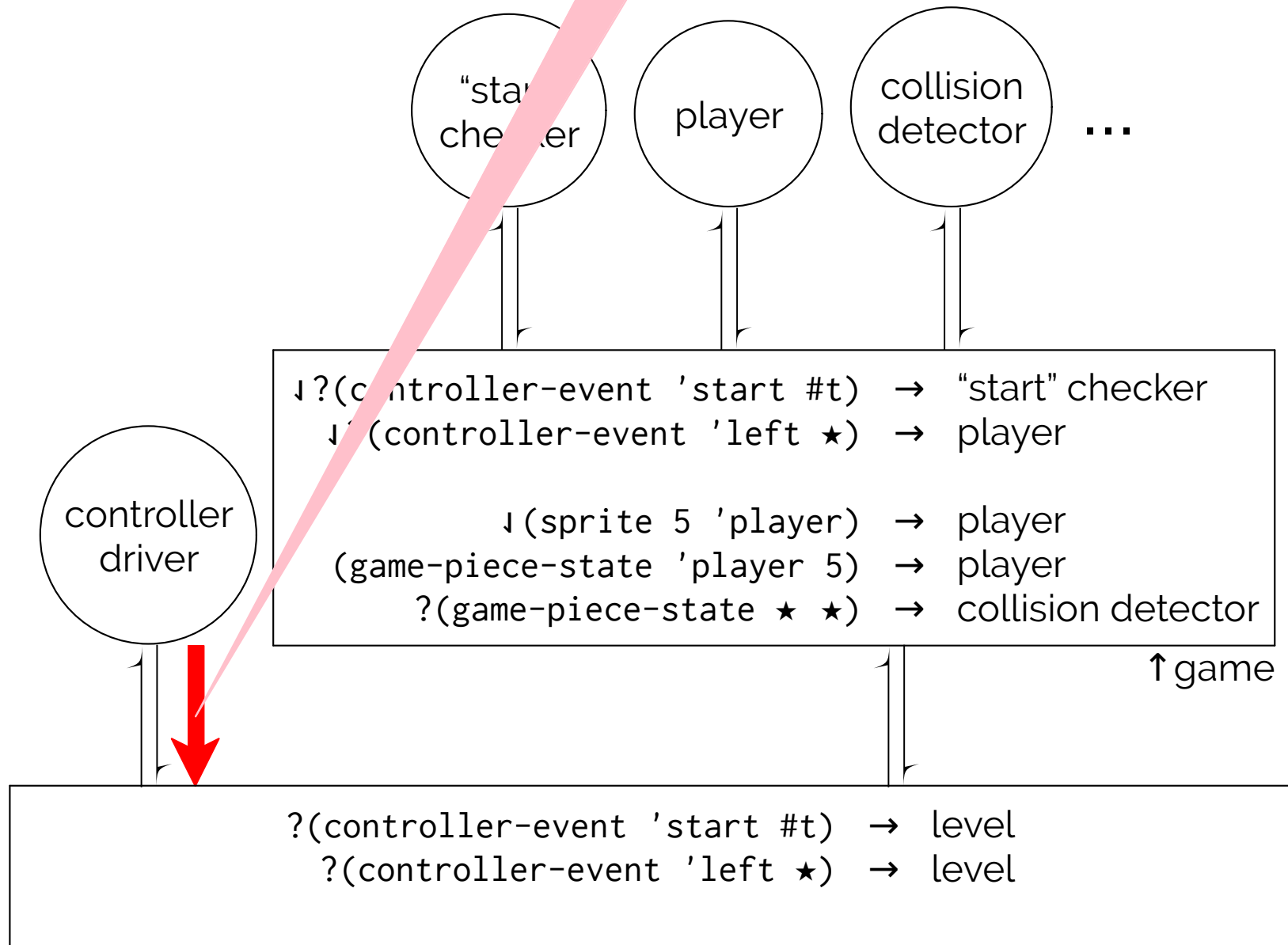
- watch state of left-arrow
- listen to clock-tick while arrow pressed
- maintain sprite & game-piece-state



↑ game

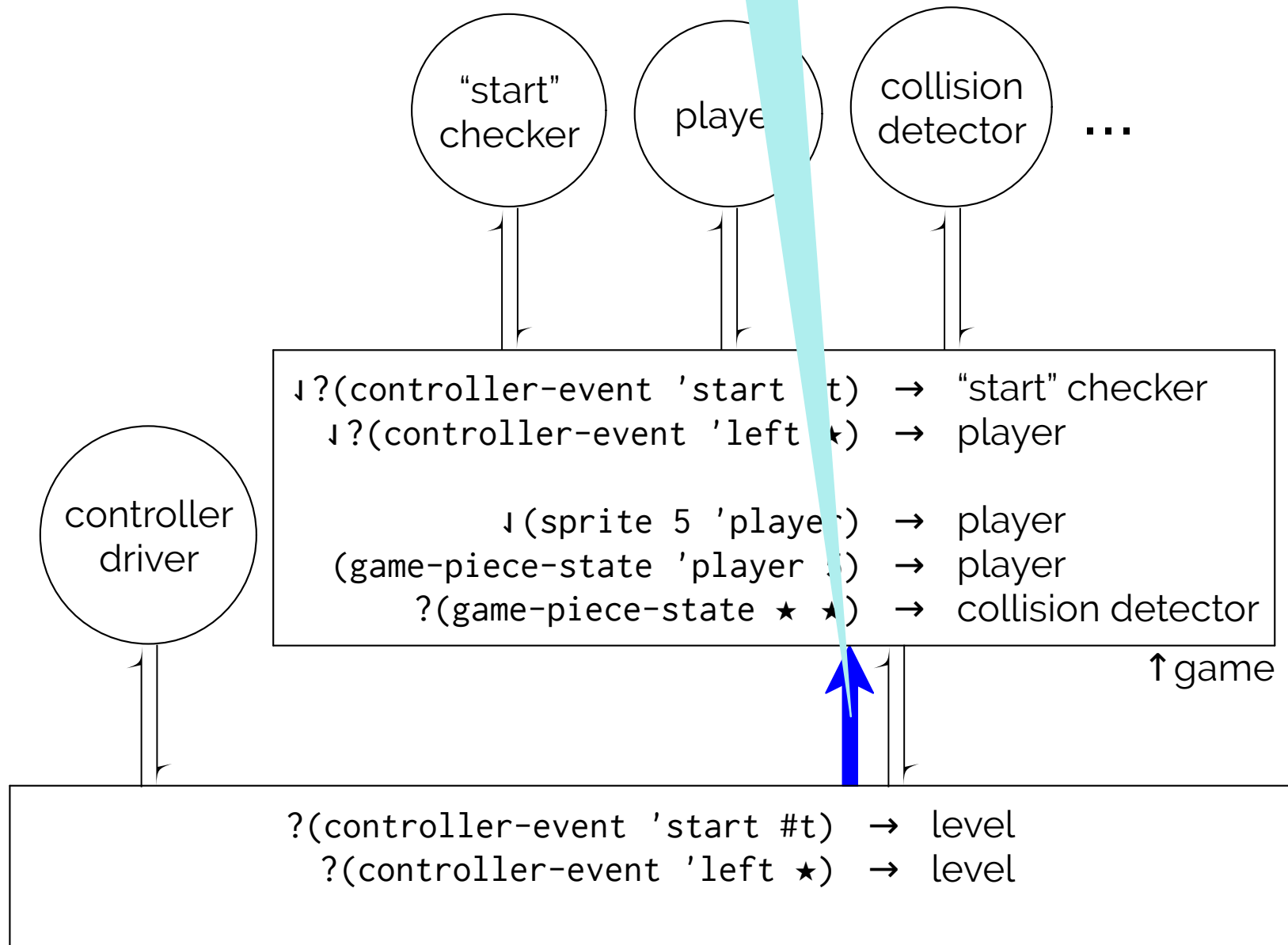
Managing conversational state

< (controller-event 'left #t) >

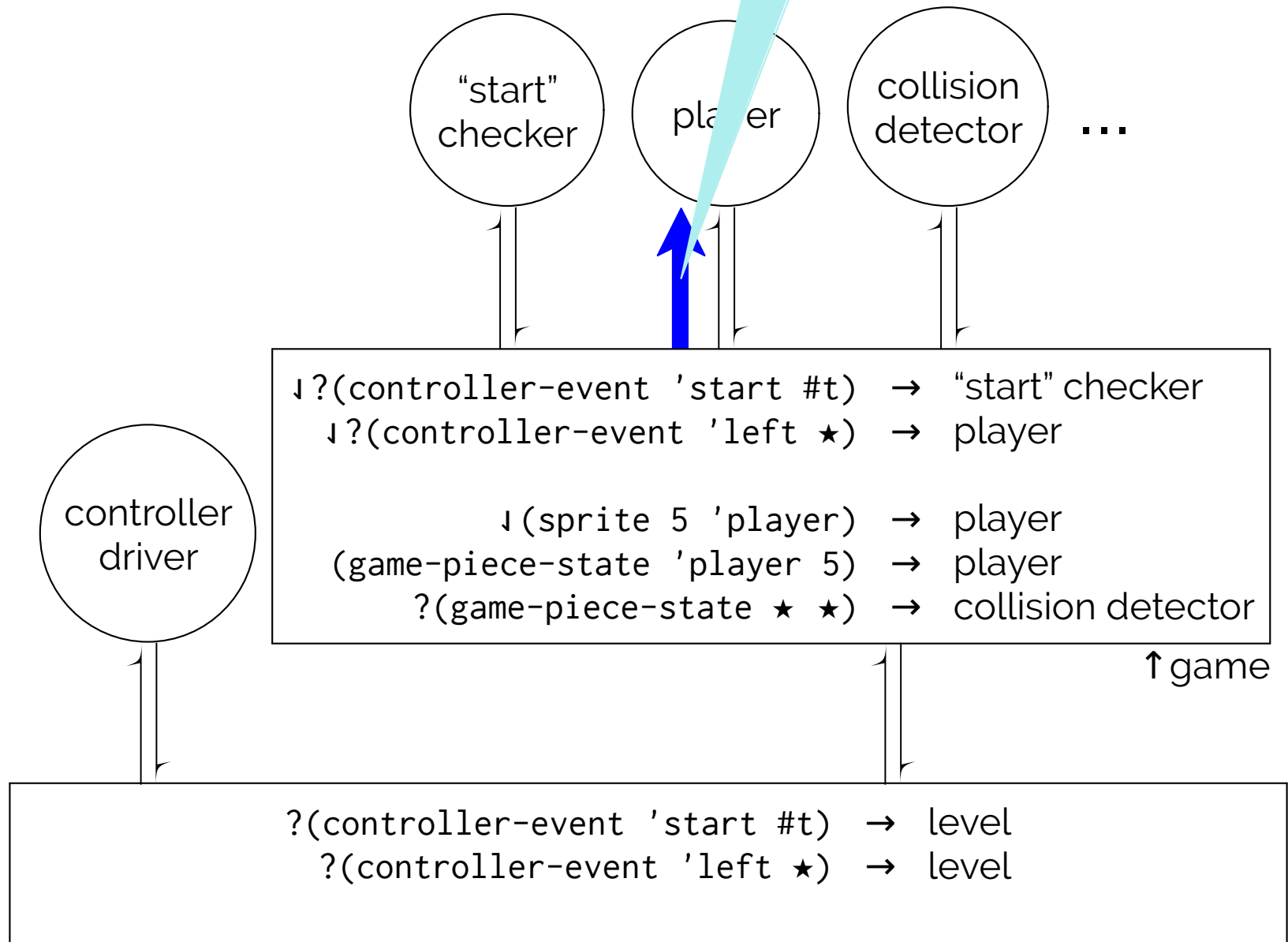


Managing conversational state

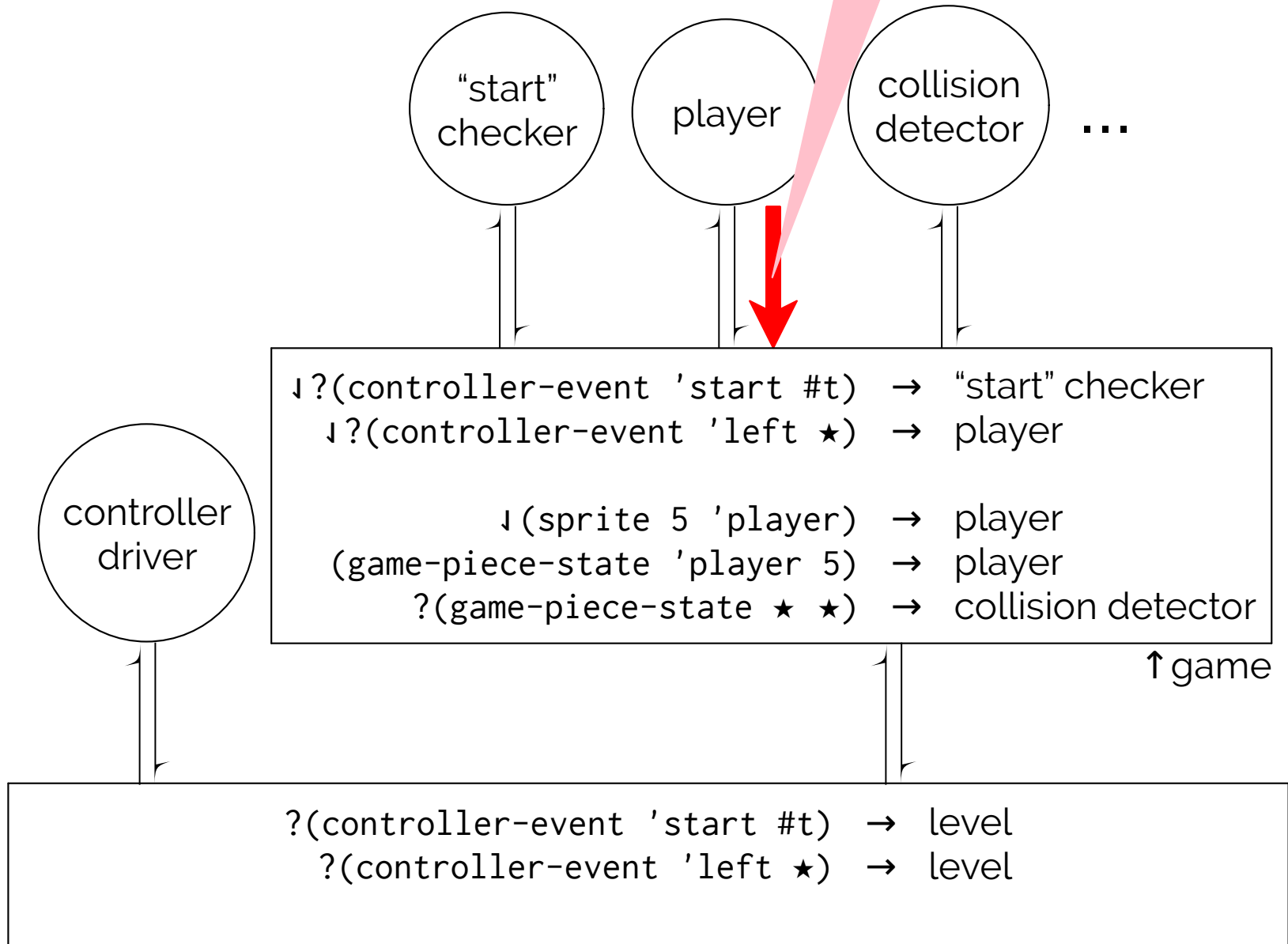
< (controller-event 'left #t) >



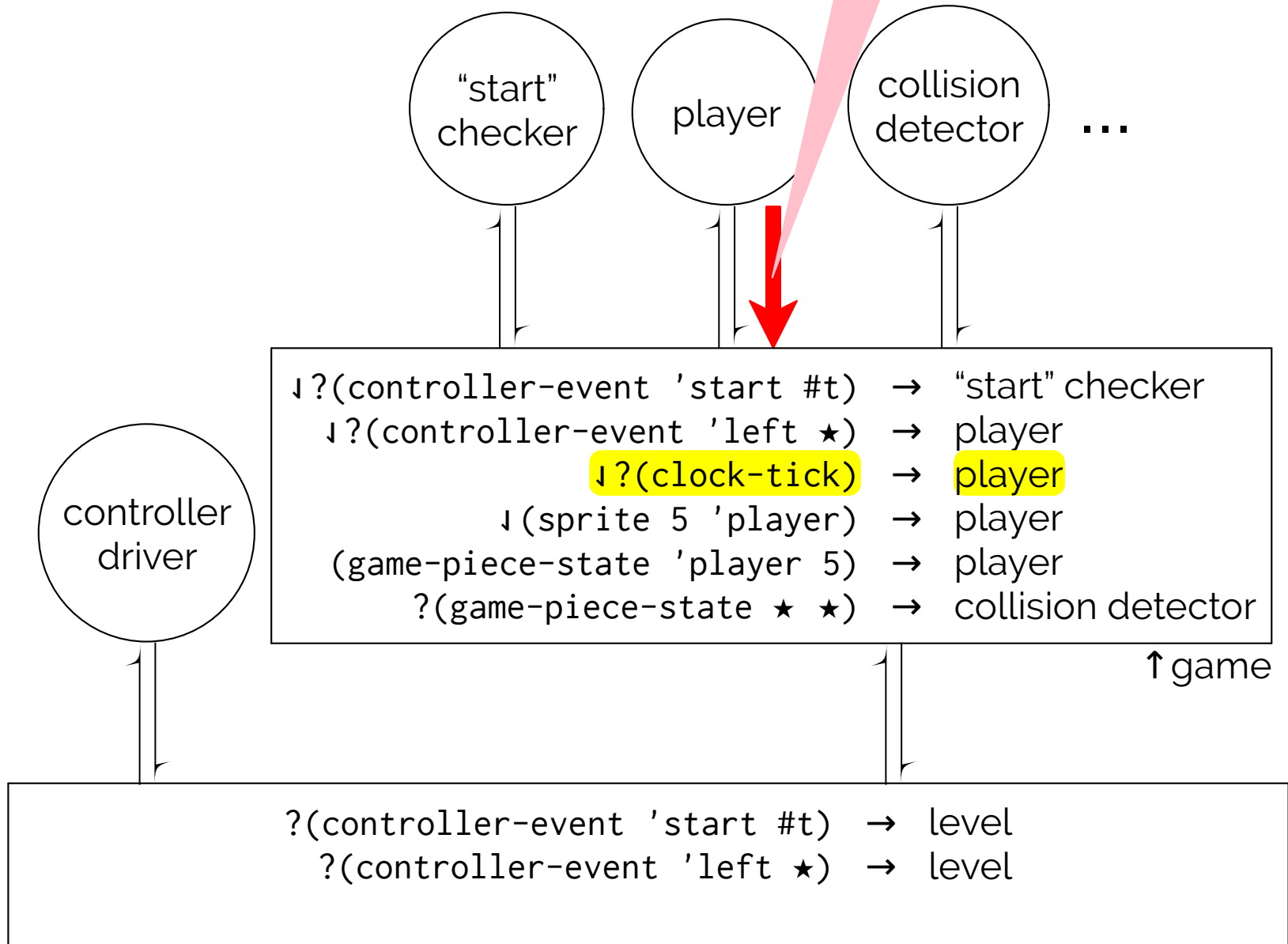
< ↓(controller-event 'left #t) >



assert(↓?(clock-tick))

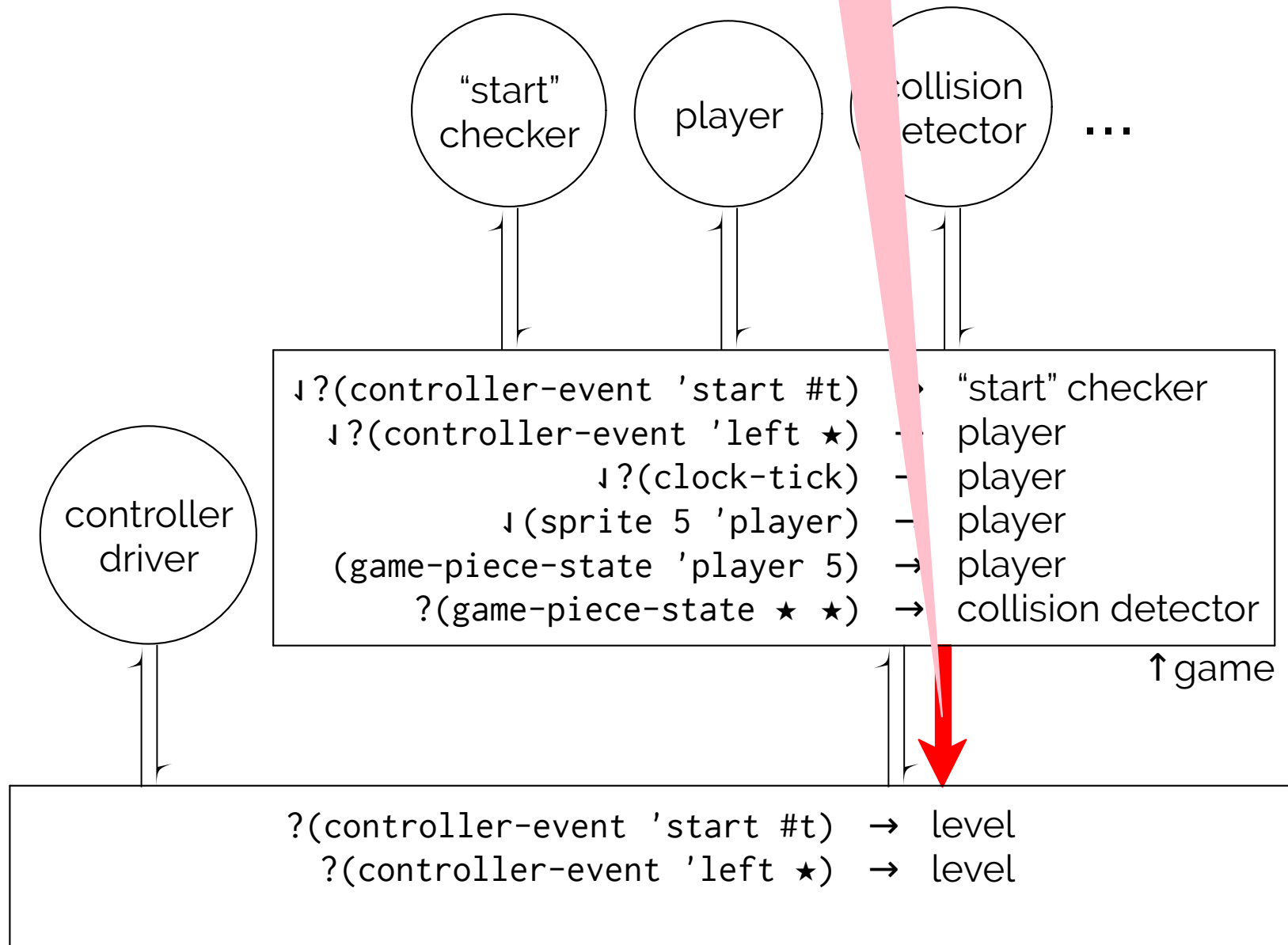


assert(↓?(clock-tick))



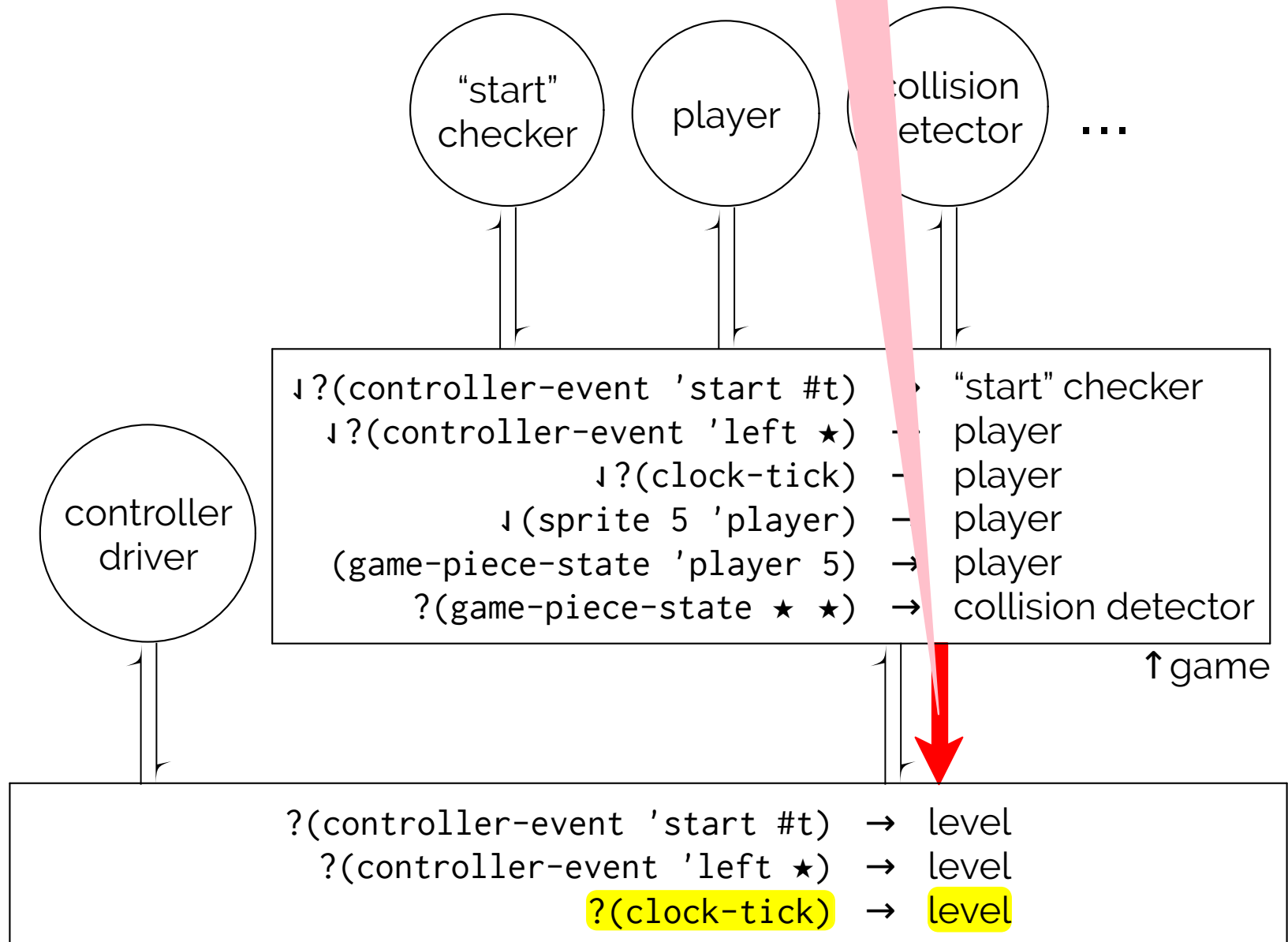
Managing conversational state

`assert(?(clock-tick))`

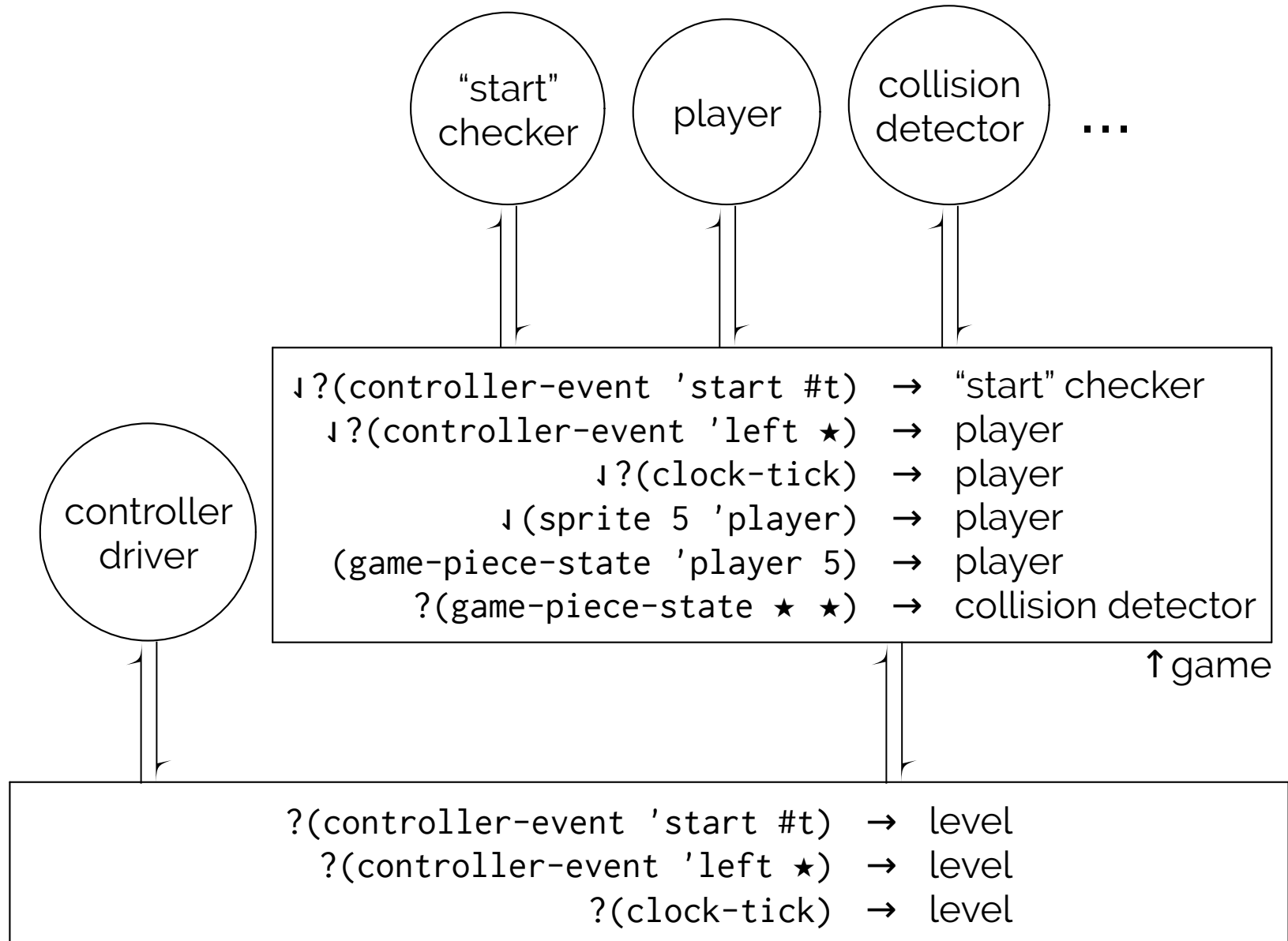


Managing conversational state

`assert(?(clock-tick))`

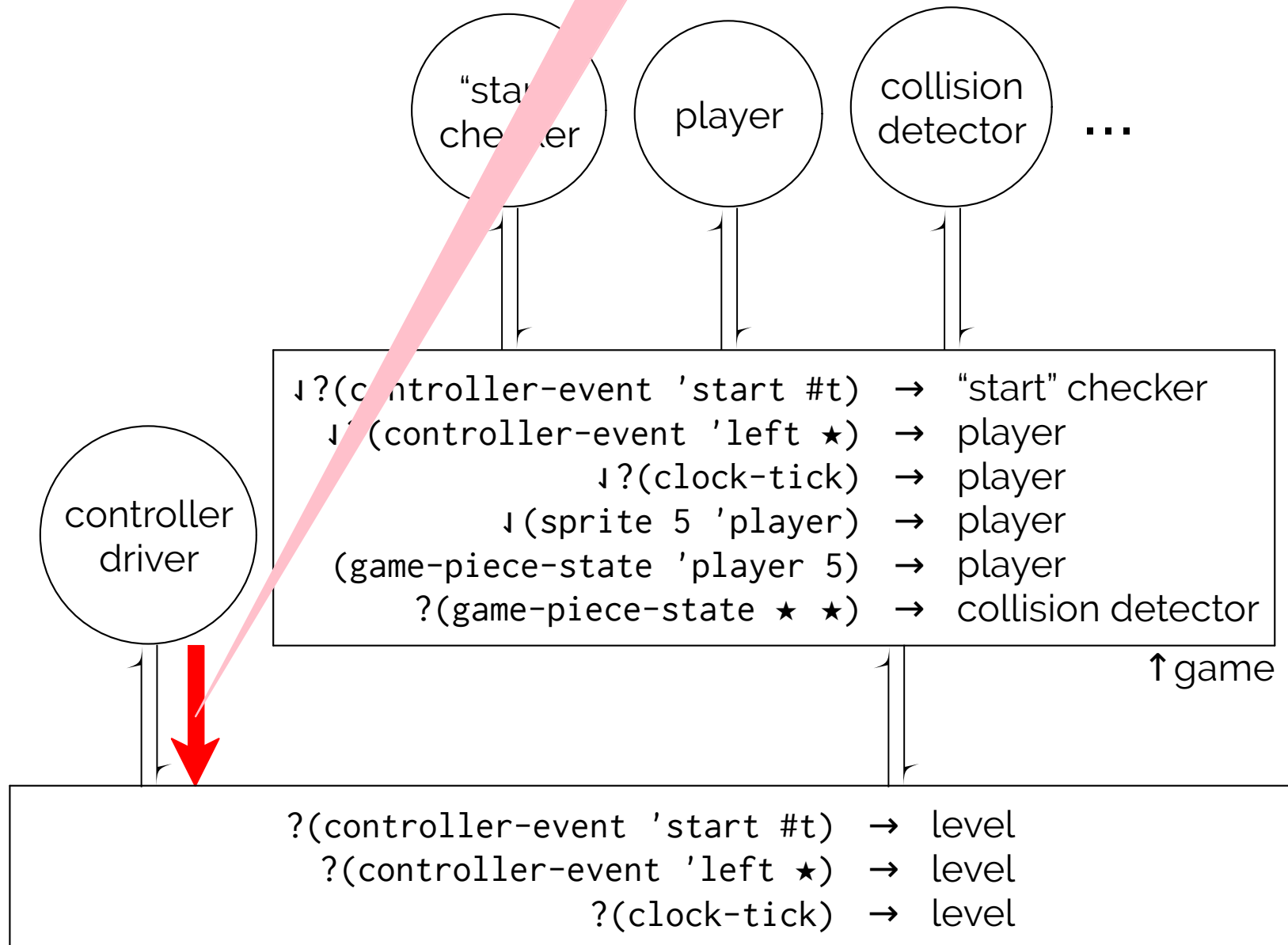


Managing conversational state



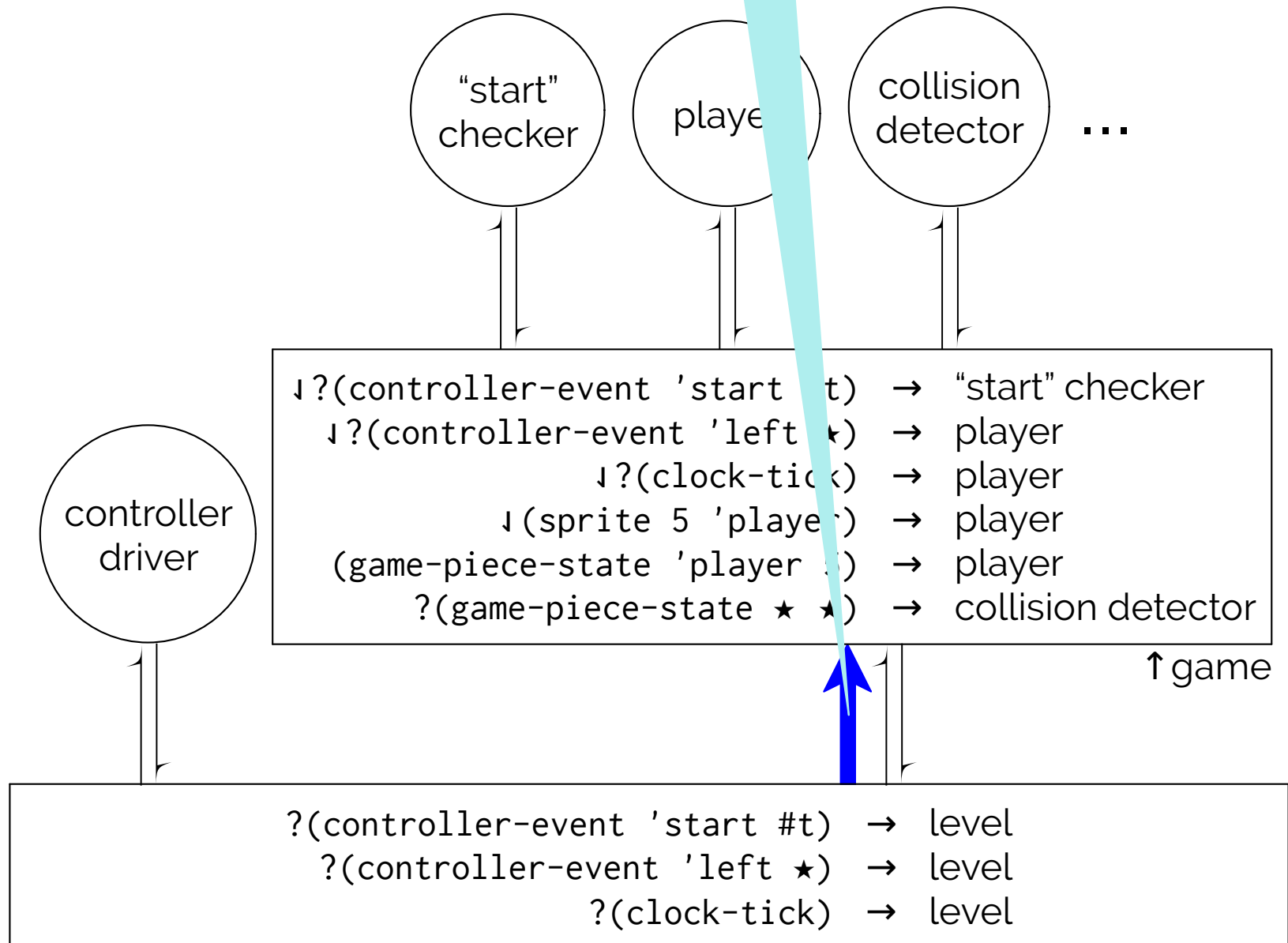
Managing conversational state

< (controller-event 'left #f) >

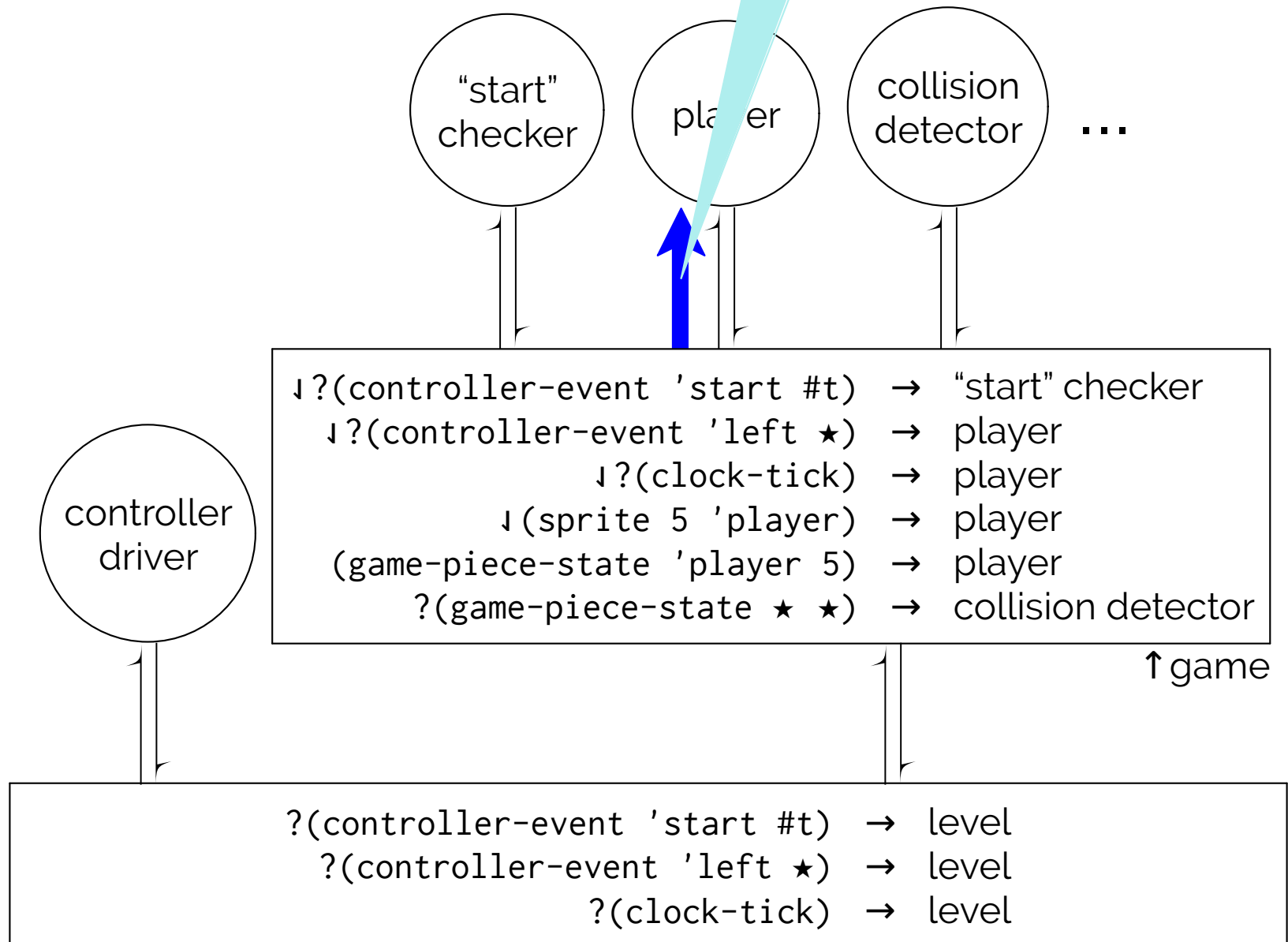


Managing conversational state

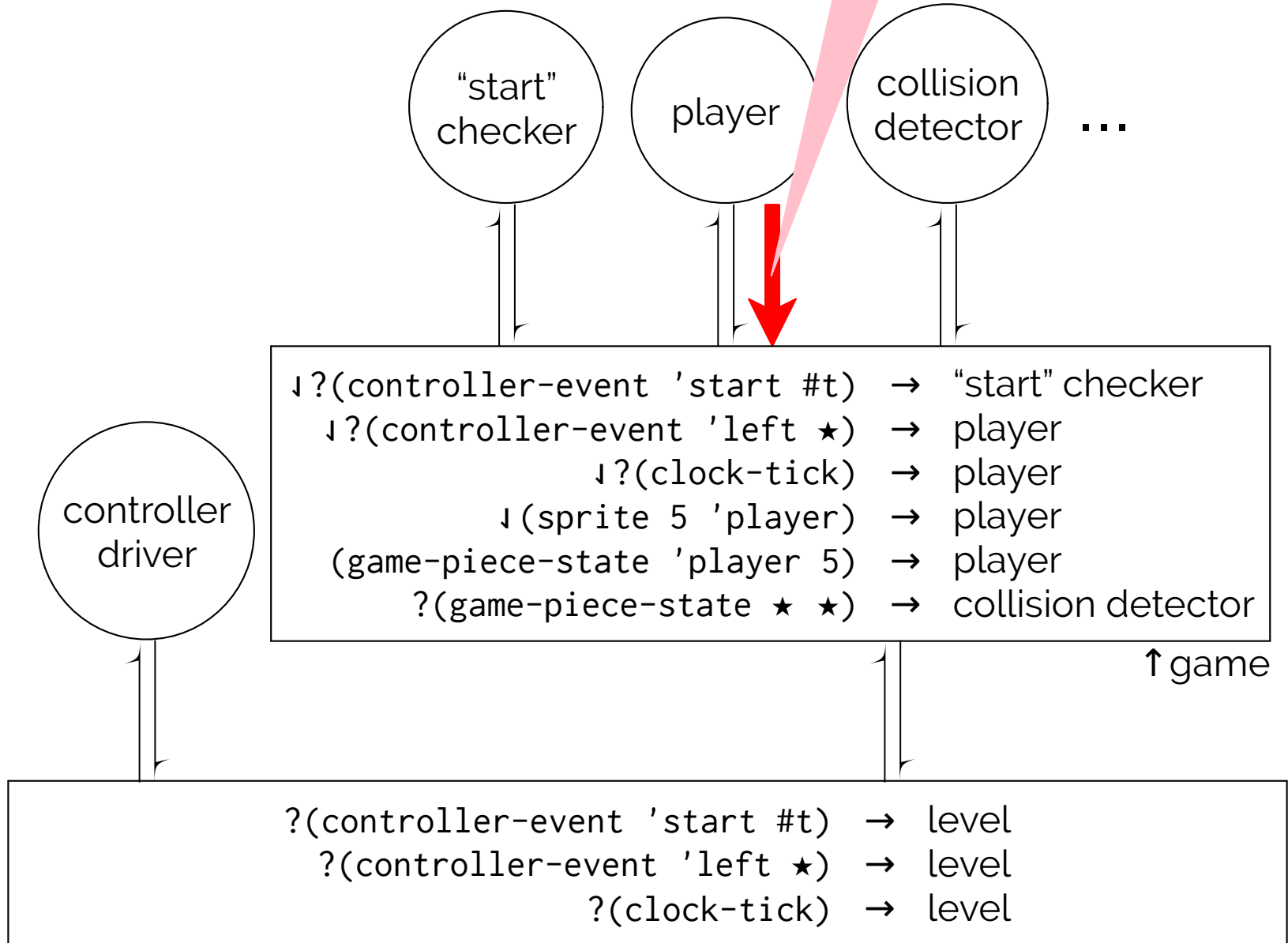
< (controller-event 'left #f) >



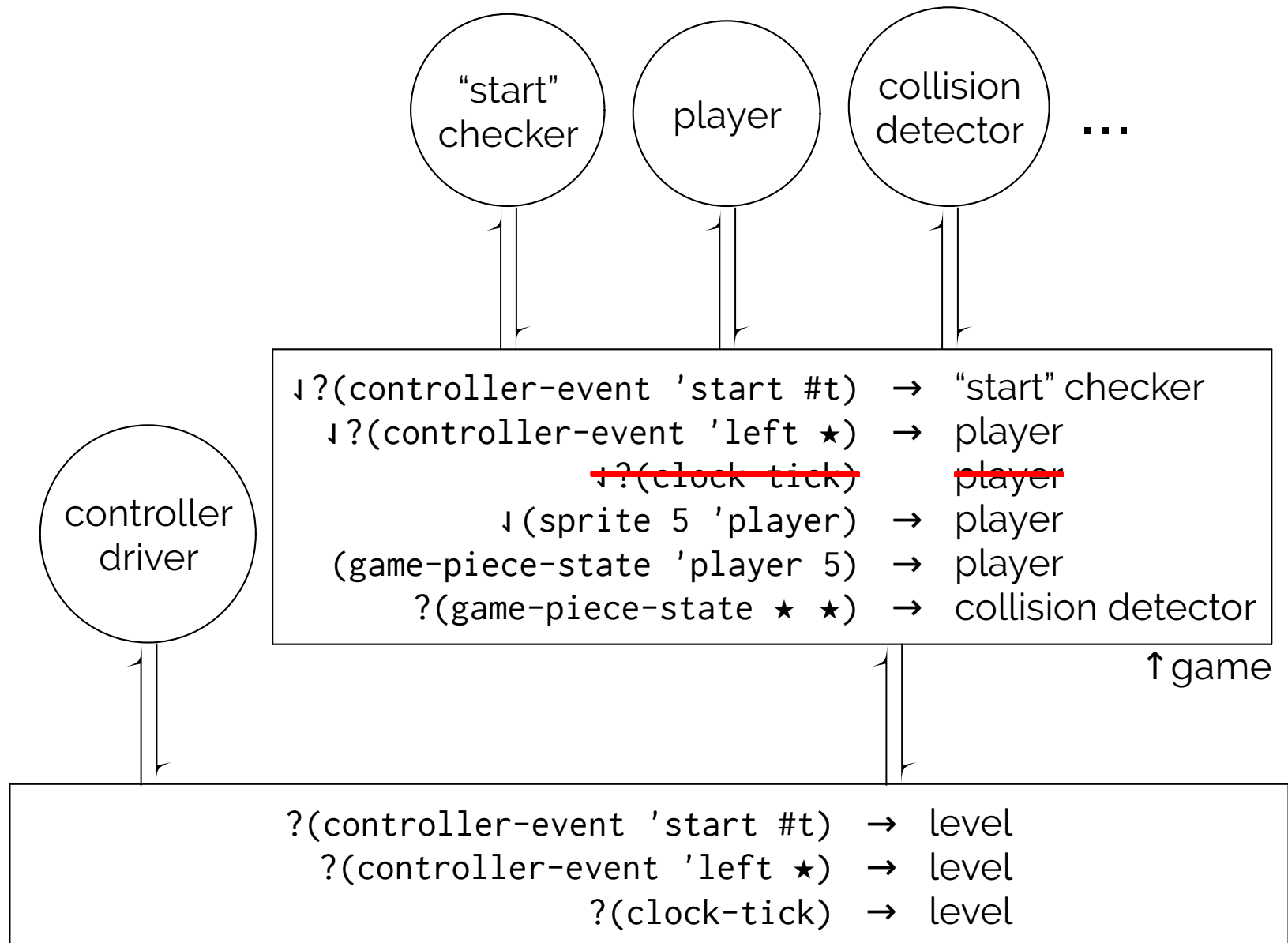
< ↓(controller-event 'left #f) >



retract(↓?(clock-tick))

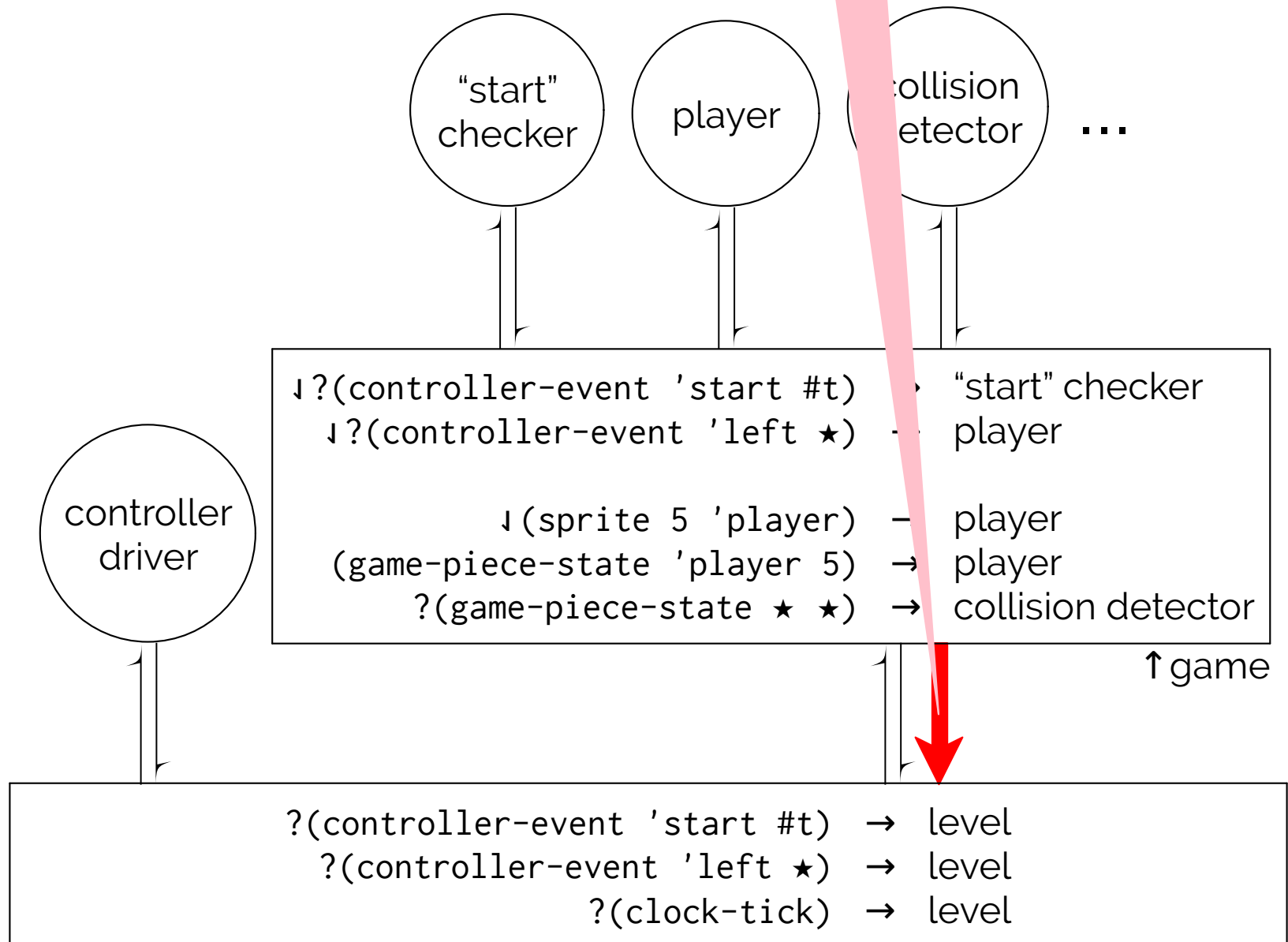


Managing conversational state

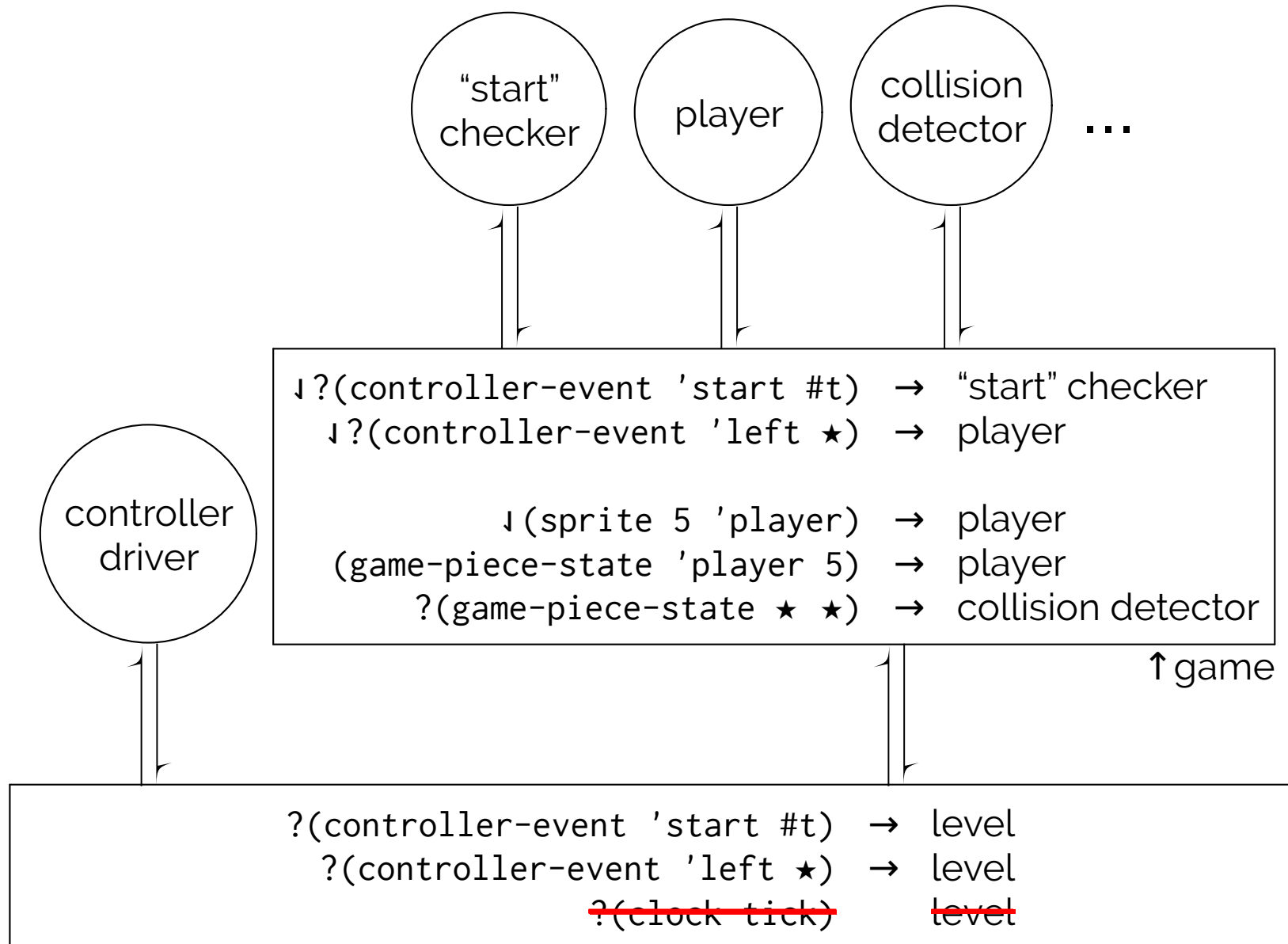


Managing conversational state

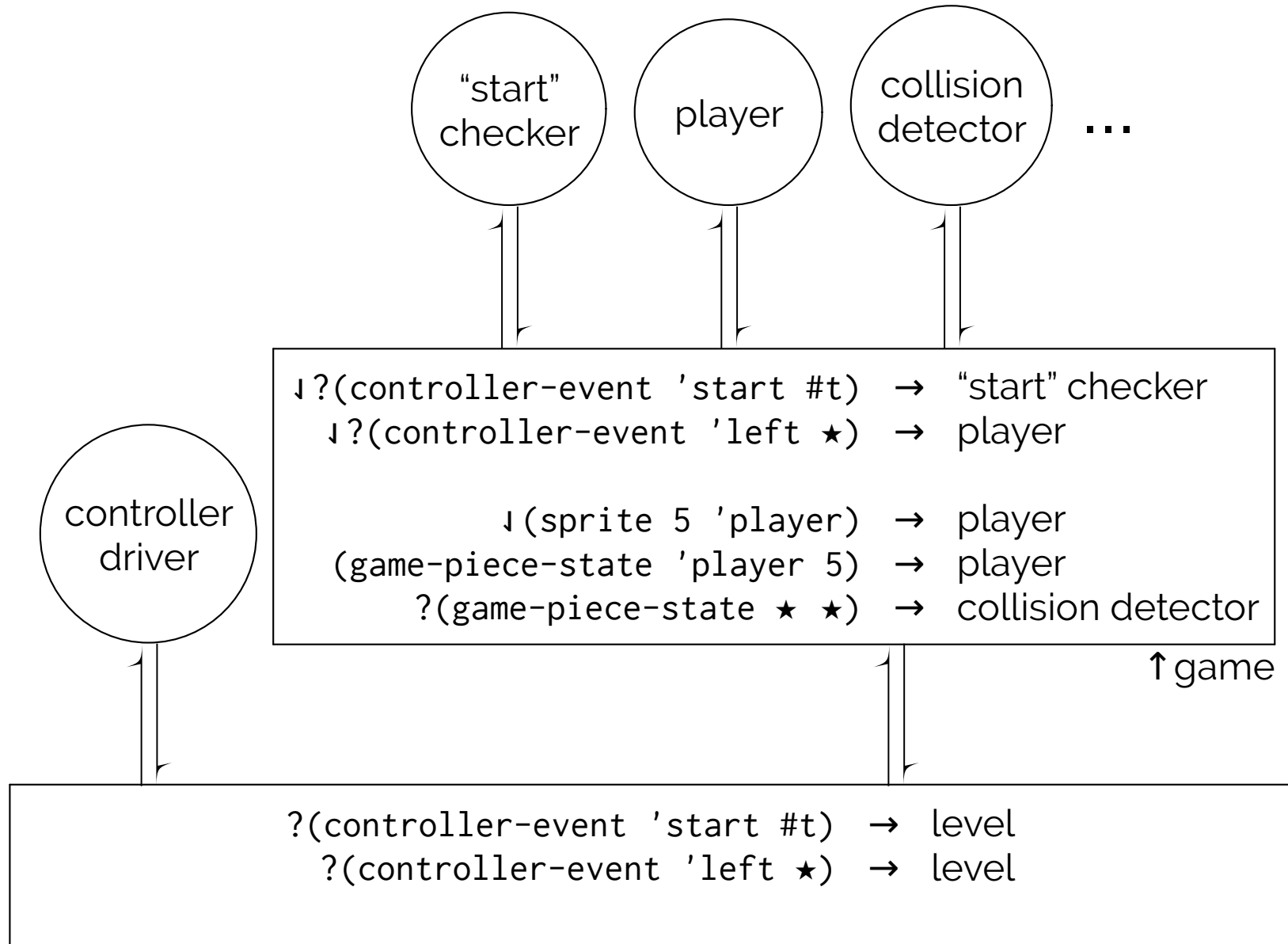
`retract(?(clock-tick))`



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))))))
```



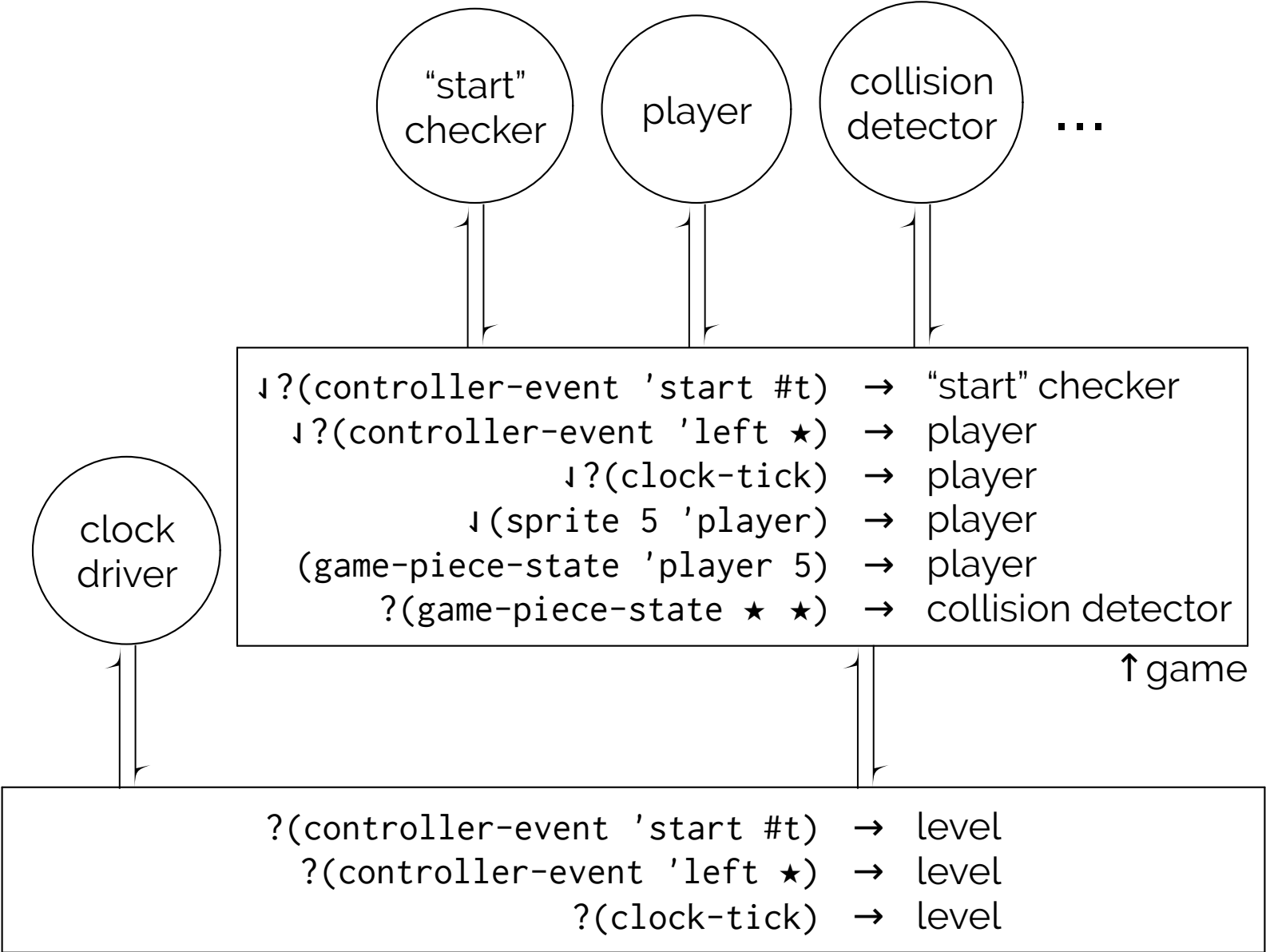
```
(define (spawn-player)
  (define position 5)
  (actor (forever
    (assert (sprite position 'player) #:meta-level 1)
    (assert (game-piece-state 'player position))
    (on (message (controller-event 'left #t)
      #:meta-level 1)
      (until (message (controller-event 'left #f)
        #:meta-level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

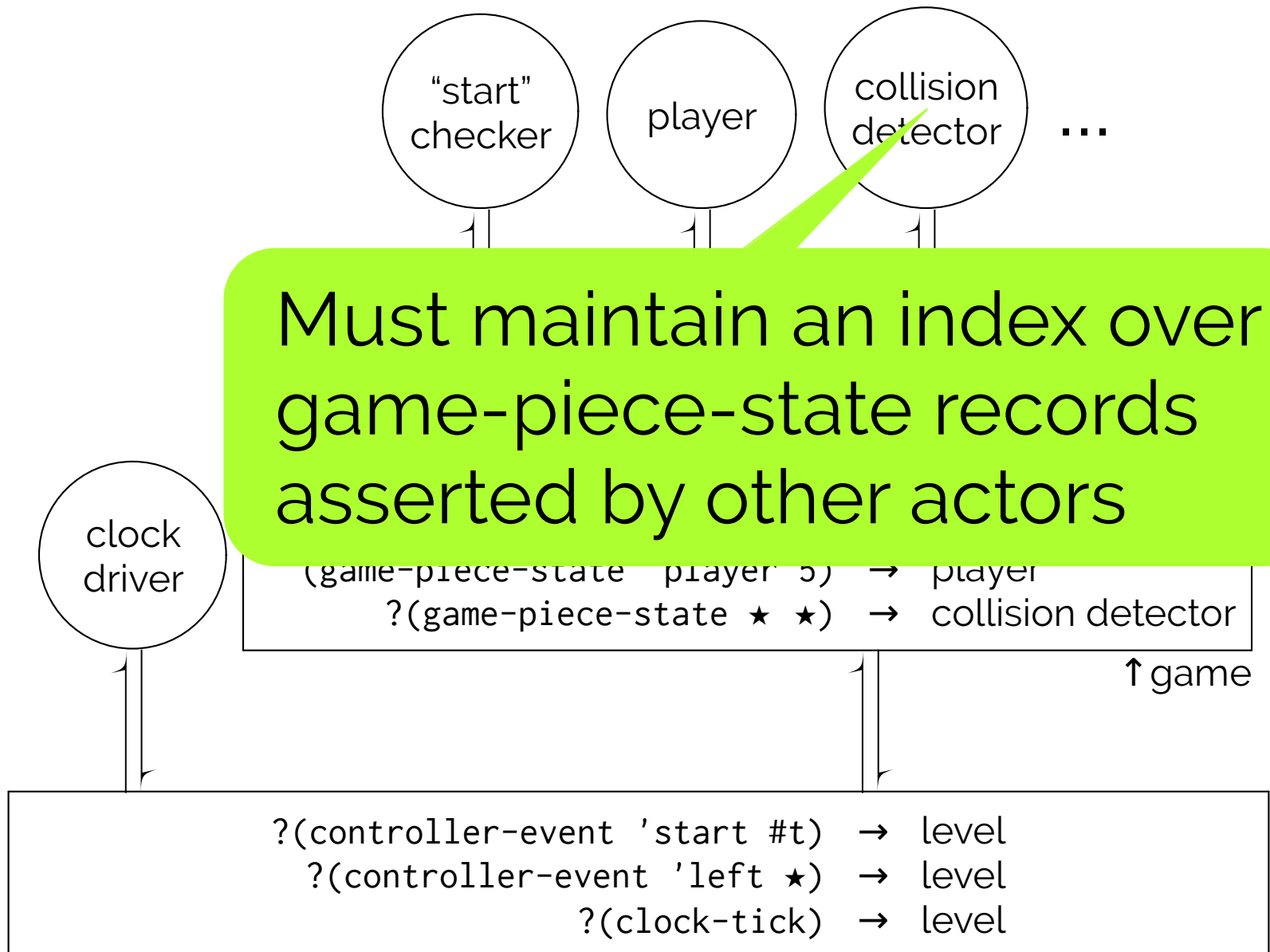
Syndicate DSL by example

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

Monitoring changes in shared state

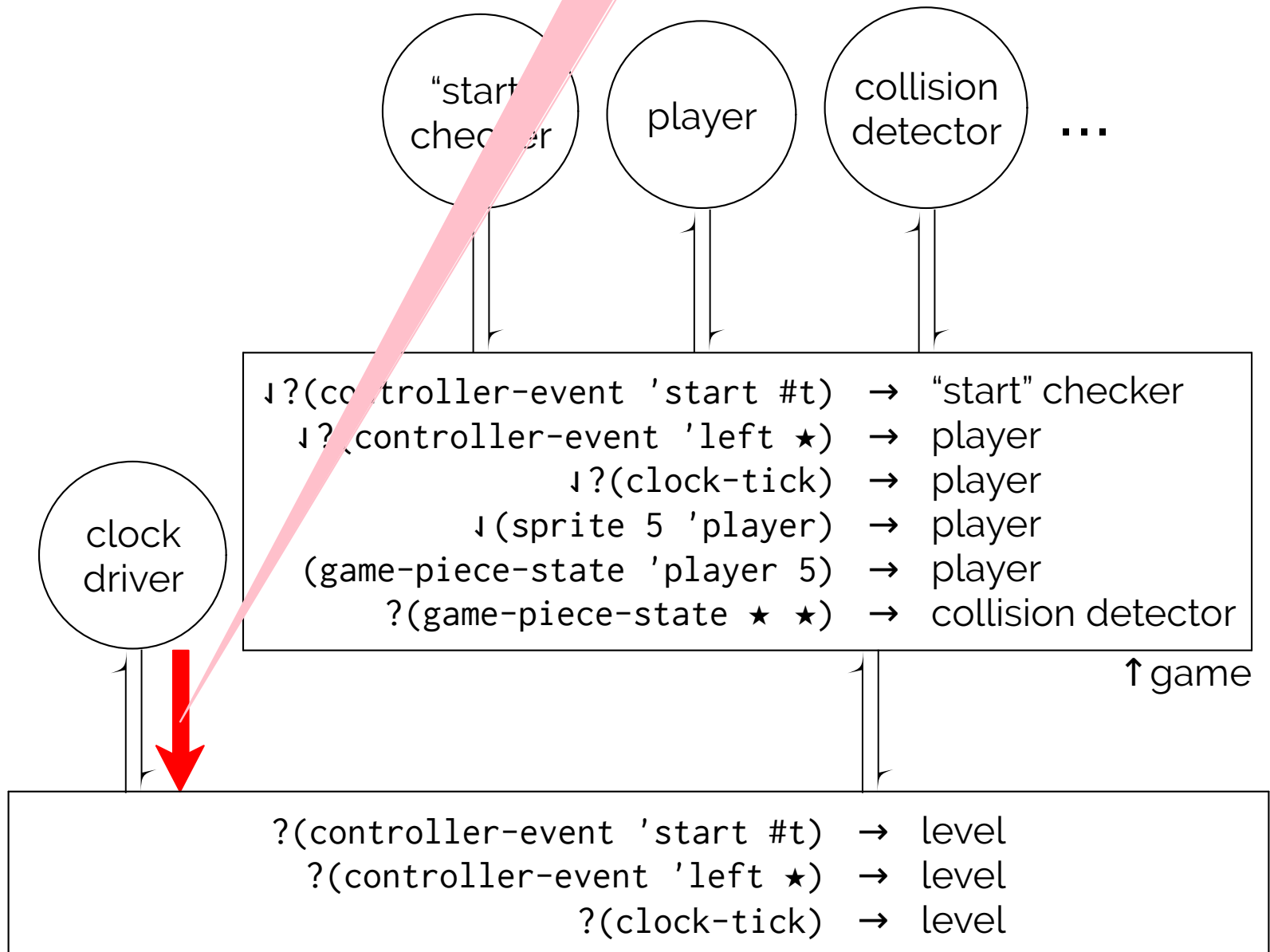


Monitoring changes in shared state



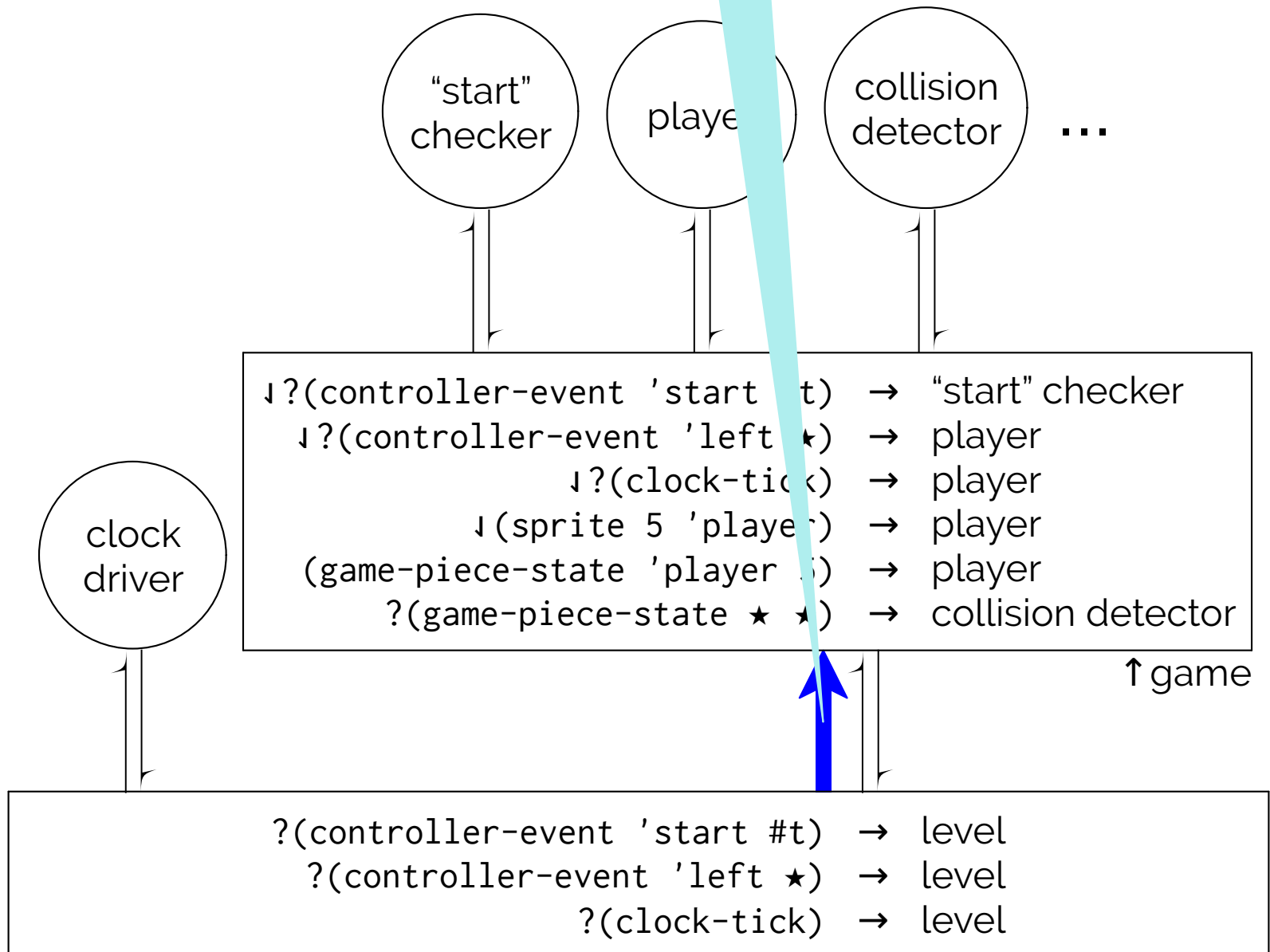
Monitoring changes in shared state

< (clock-tick) >

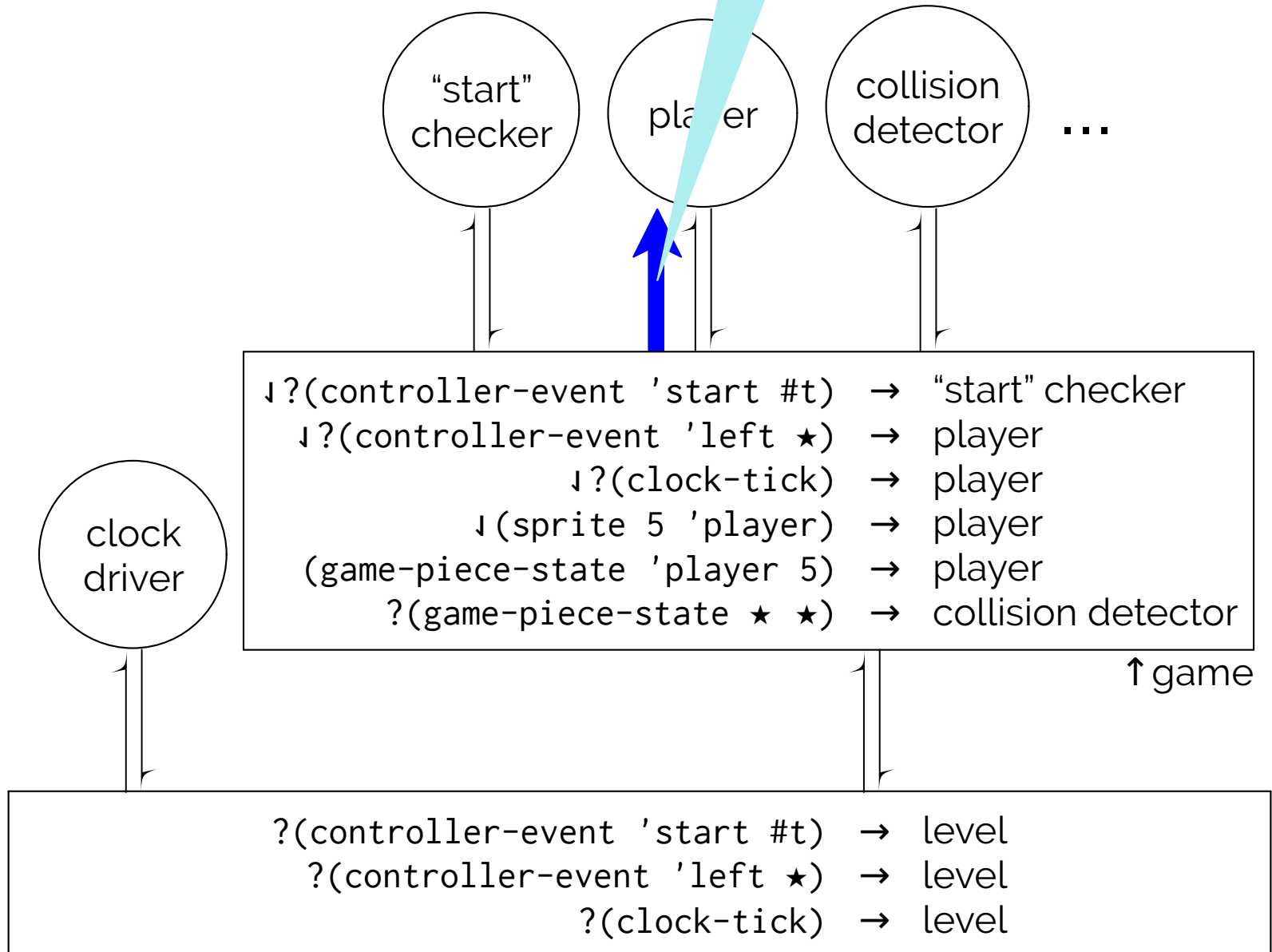


Monitoring changes in shared state

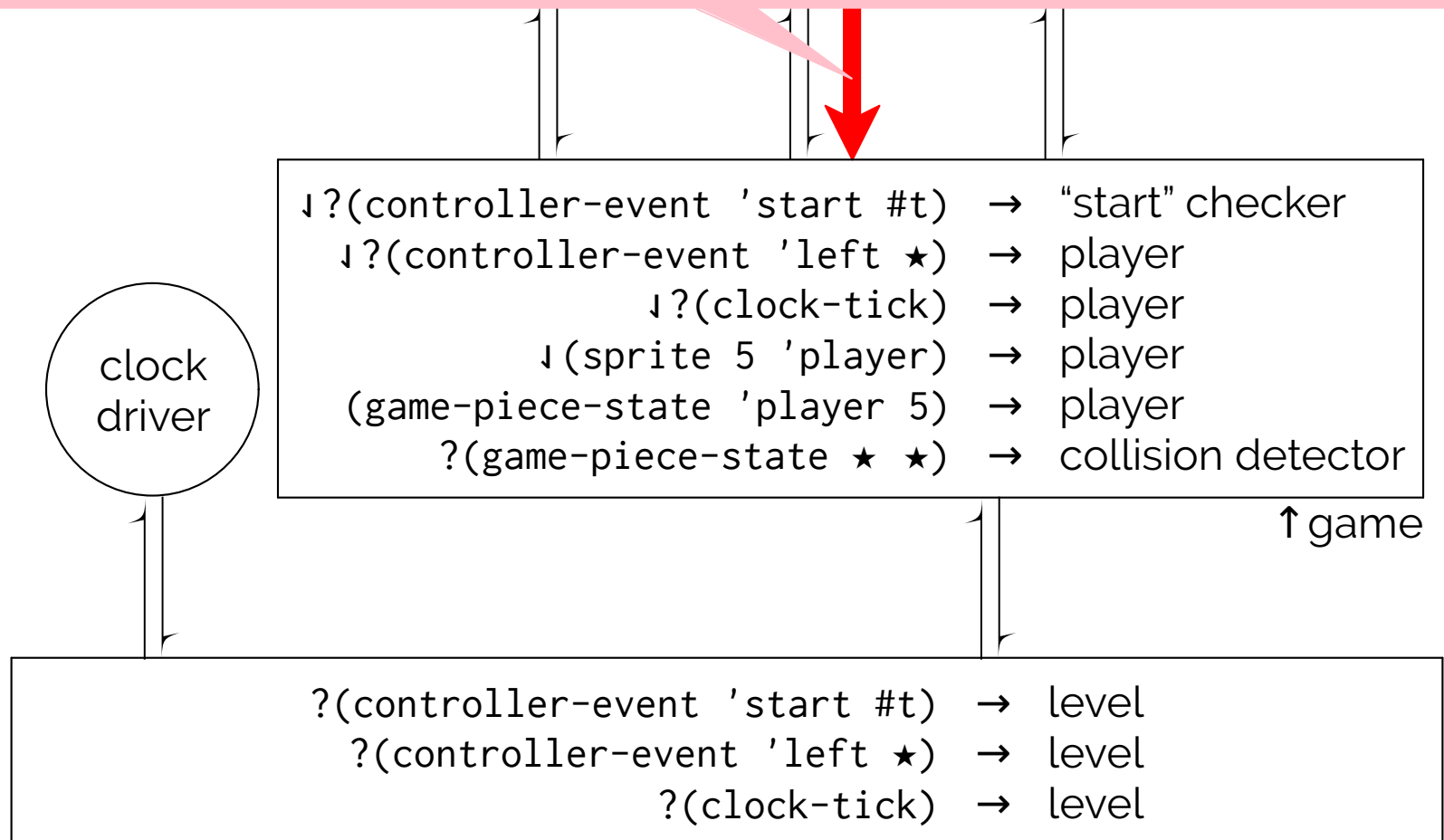
< (clock-tick) >



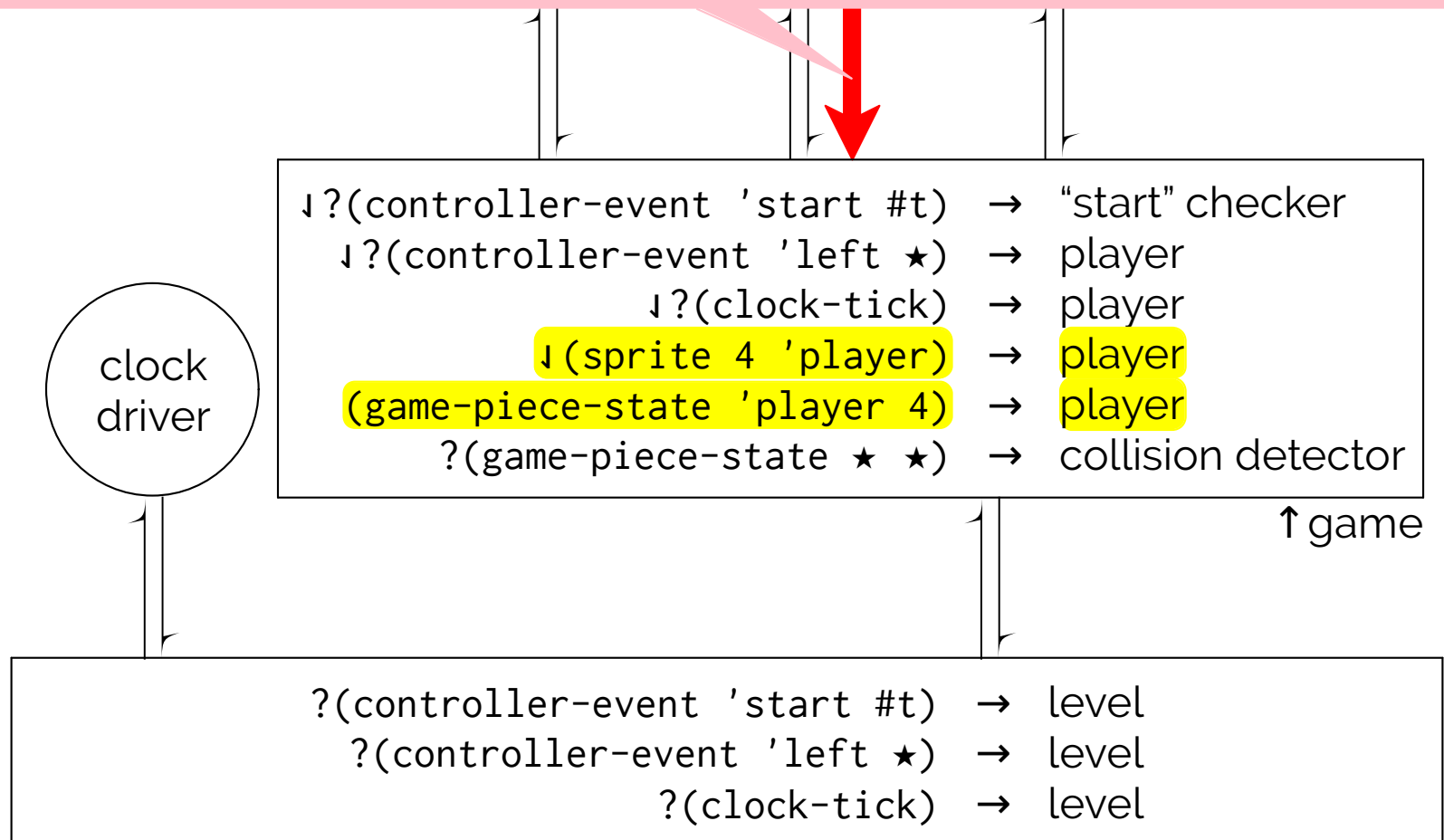
< ↓ (clock-tick) >




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

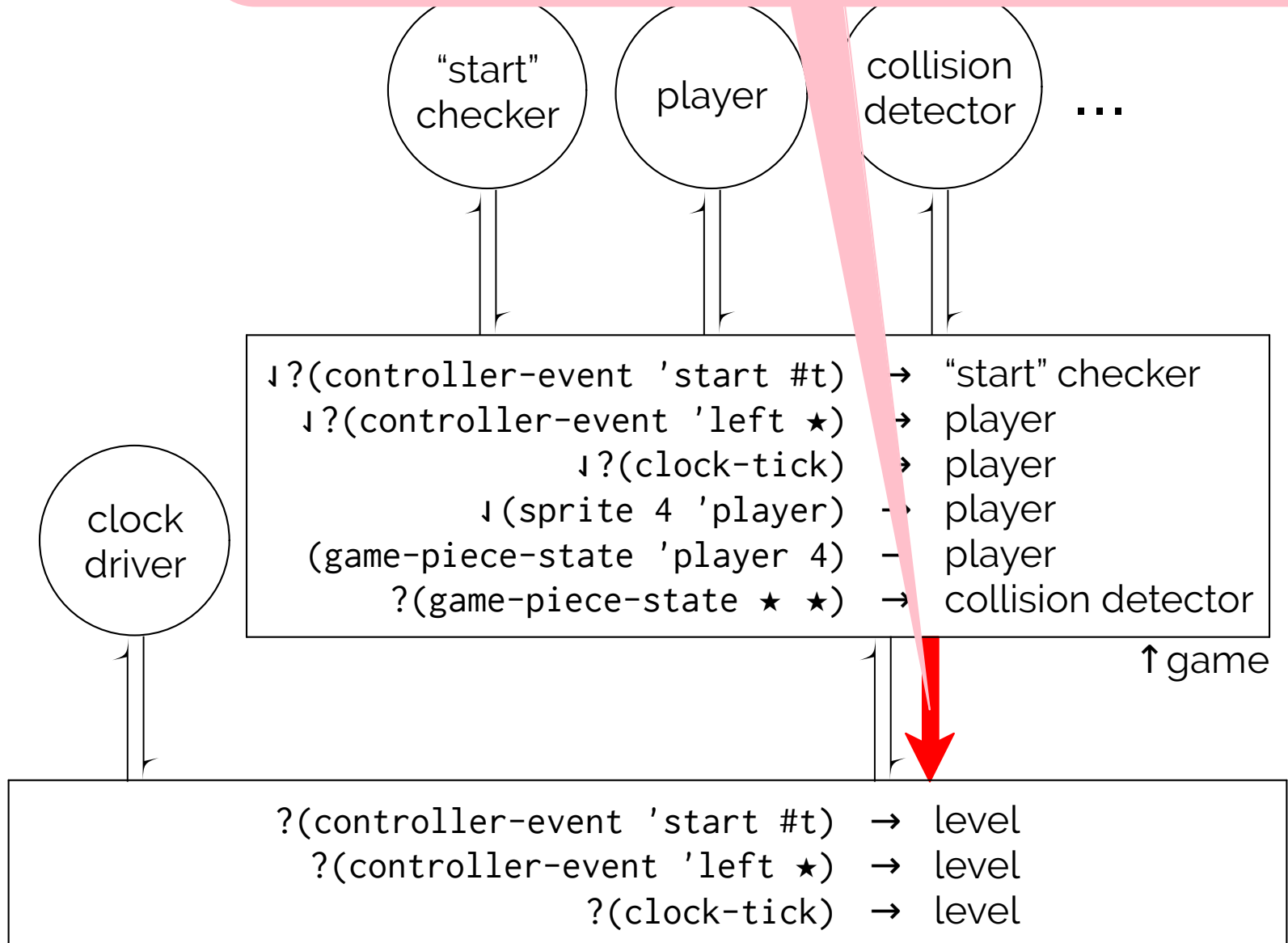


```
retract( (sprite ★ ★) ),
assert( (sprite 4 'player) ),
retract( (game-piece-state ★ ★) ),
assert( (game-piece-state 'player 4) )
```



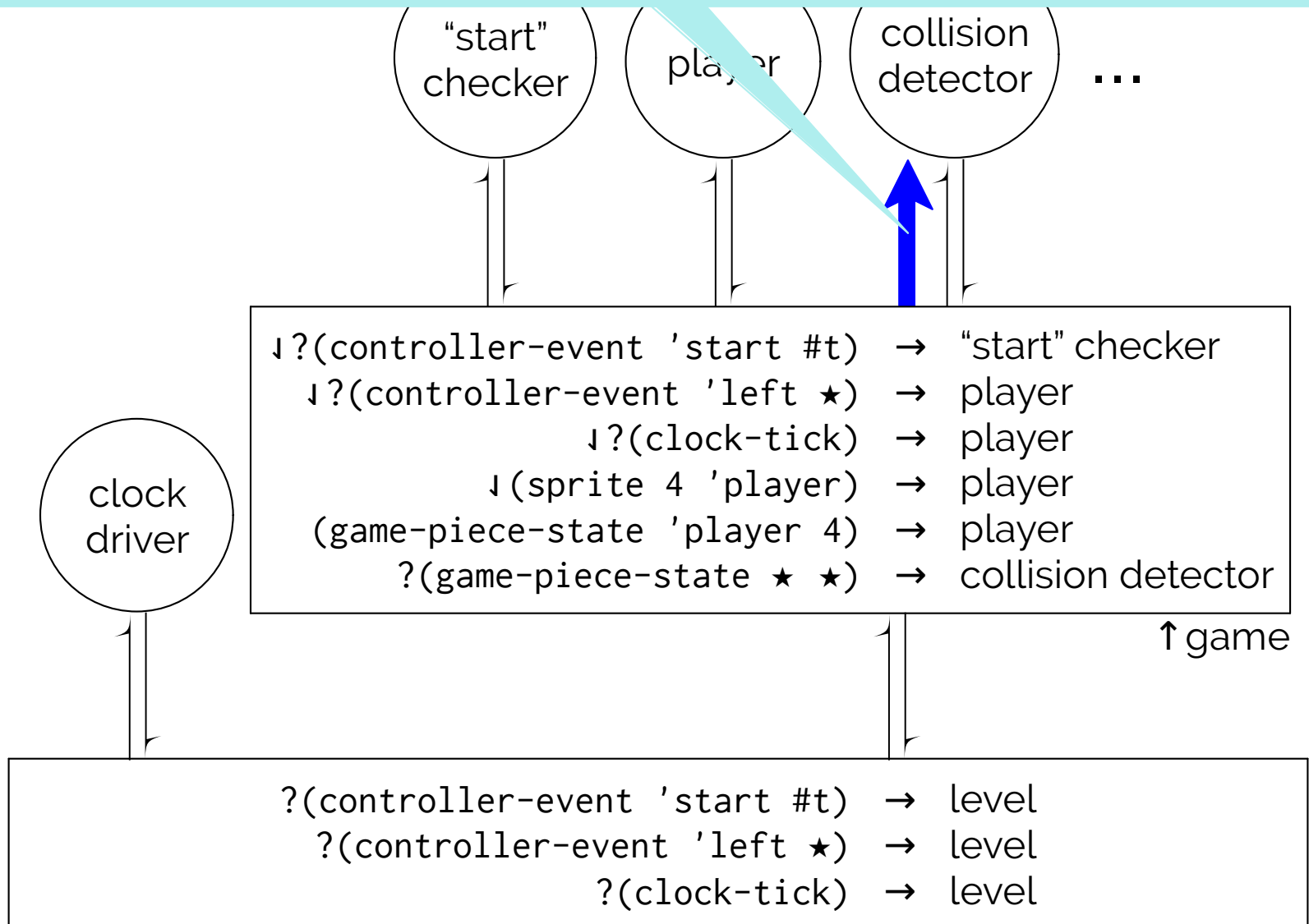
Mo

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

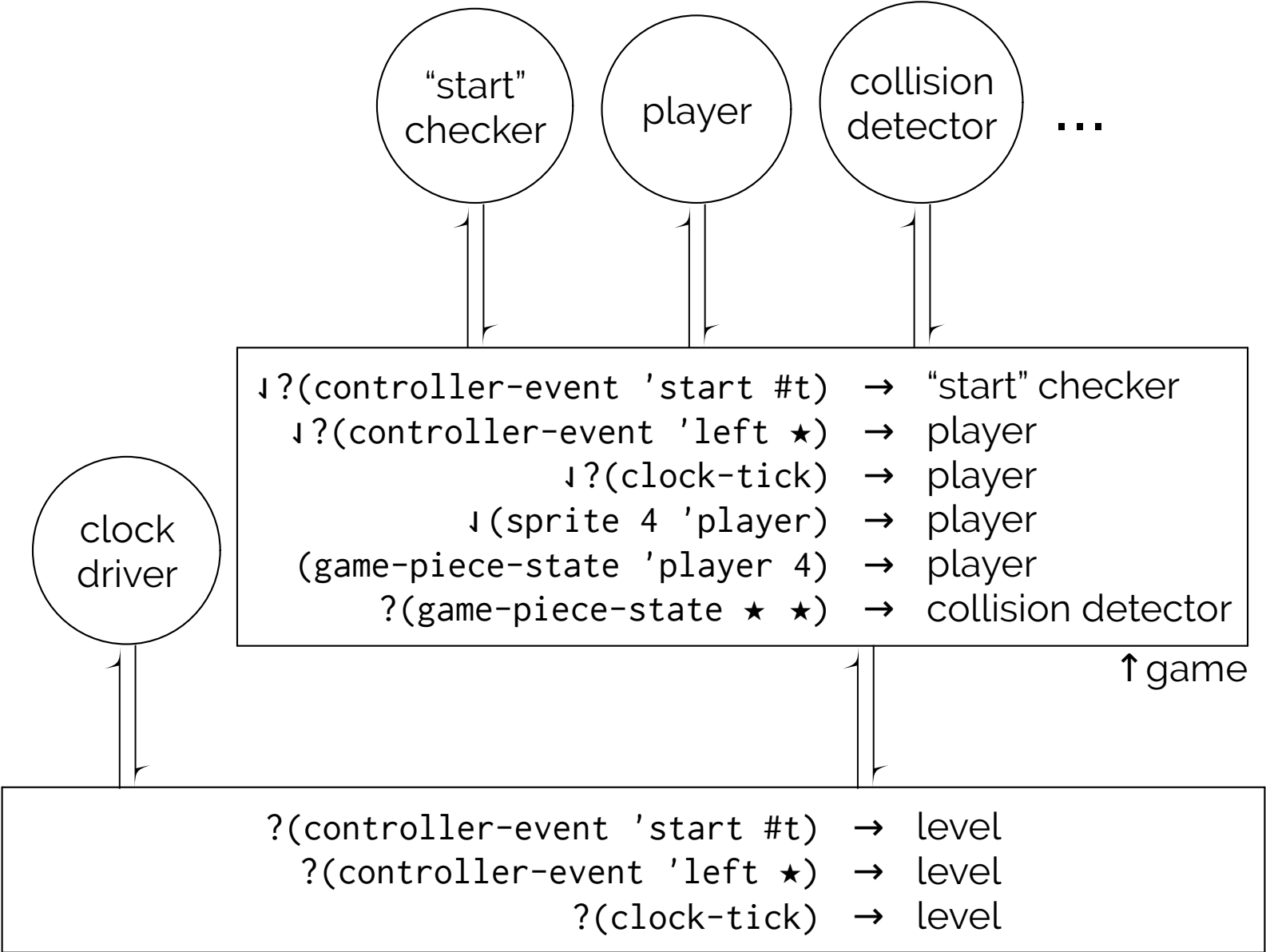


Monitoring changes in shared state

```
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
```

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

```
(collision-detection-state (hash))
```

```
(sub (game-piece-state ? ?))))
```

```
p)))]  
e)))
```

```

(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

```
186 ;; when a (y-collision) is detected reset velocity to 0
187 (define (spawn-vertical-motion gravity jump-v max-v)
188 - (struct v-motion-state (jumping? motion clock) #:transparent)
189 - (spawn
190 - (lambda (e s)
191 - (match-define (v-motion-state jumping? motion-old clock) s)
192 - (match e
193 - [(message (jump))
194 - (transition (v-motion-state #t
195 - (motion jump-v (motion-a motion-old))
196 - (add1 clock))
197 - #f])
198 - [(message (timer-tick))
199 - (define motion-n
200 - (motion (min max-v (+ (motion-v motion-old) (motion-a motion-old)))
201 - (motion-a motion-old)))
202 - (transition (v-motion-state jumping? motion-n clock)
203 - (message (move-y 'player (motion-v motion-old) clock)))]
204 - [(message (y-collision 'player col-clock))
205 - (and (equal? col-clock clock)
206 - (transition (v-motion-state #f
207 - (motion 0 (motion-a motion-old))
208 - clock) #f))]
209 - [_ #f]))
210 - (v-motion-state #f (motion 0 gravity) 0)
211 - (list (sub (jump))
212 - (sub (timer-tick))
213 - (sub (y-collision 'player ?))))

214
215 ;; create a clock that sends (timer-tick) every period-ms
216 (define (spawn-clock period-ms)

153 ;; when a (y-collision) is detected reset velocity to 0
154 (define (spawn-vertical-motion gravity jump-v max-v)
155 + (forever #:collect ([mot (motion 0 gravity)]
156 + [clock 0])
157 + (on (message (jump))
158 + (values (motion jump-v (motion-a mot))
159 + (add1 clock)))
160 + (on (message (timer-tick))
161 + (send! (move-y 'player (motion-v mot) clock))
162 + (values (motion (min max-v
163 + (+ (motion-v mot) (motion-a mot)))
164 + (motion-a mot))
165 + clock))
166 + (on (message (y-collision 'player clock))
167 + (values (motion 0 (motion-a mot))
168 + clock))))

169
170 ;; create a clock that sends (timer-tick) every period-ms
171 (define (spawn-clock period-ms)
```

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/>