Exercise 3.31.

The internal procedure accept-action-procedure! defined in make-wire specifies that when a new action procedure is added to a wire, the procedure is immediately run. Explain why this initialization is necessary. In particular, trace through the half-adder example in the paragraphs above and say how the system's response would differ if we had defined accept-action-procedure! as

```
(define (accept-action-procedure! proc)
  (set! action-procedures (cons proc action-procedures)))
```

Answer.

Running the action procedure as soon as it is added to a wire makes sure that the output signal of the function box the wire related get initialized in term of all its input signal.

Let's try to trace through the half-adder example in the paragraph above in the environment model to see how the system's response would differ if we had eliminated (proc) from the body of acceptaction-procedure.

Figure 1 shows the procedure objects of make-wire, propagate and probe created in the global

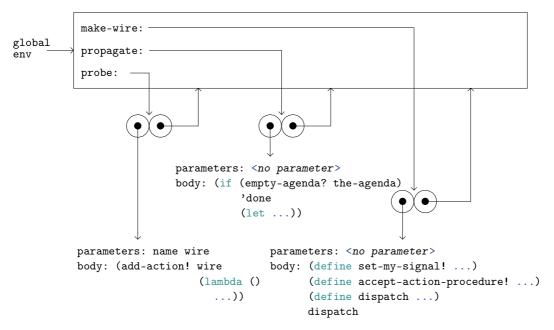


Figure 1. Procedure objects of make-wire, propagate and probe in the global environment.

frame. Using these we see how the four wires: input-1, input-2, sum and carry were defined, as figure 2 shows.

As figure 3 shows, to place a probe on sum, we first bound the arguments 'sum and sum onto the formal parameters name and wire respectively in E2 whose enclosing environment is E1, which was set up in evaluating the procedure sum, and then evaluated the body of probe. The procedure probe accomplished its job by calling to add-action!, this was in turns done in a new environment E3 whose enclosing environment is E2. This was followed by calling another procedure accept-action-procedure! in a newly constructed environment E4, which had its enclosing environment E3. Since we have erase (proc), the step of running a procedure whenever it is added to a wire. Hence, in evaluating (accept-action-procedure! (lambda () (newline)...)) the lambda expression was simpliy consing onto the list of procedure in a wire, without running. So what returned by (accept-action-procedure! (lambda () (newline)...)) was the side effect of expression

^{*.} Creative Commons © 000 2013, Lawrence X. Amlord (颜世敏, aka 颜序).

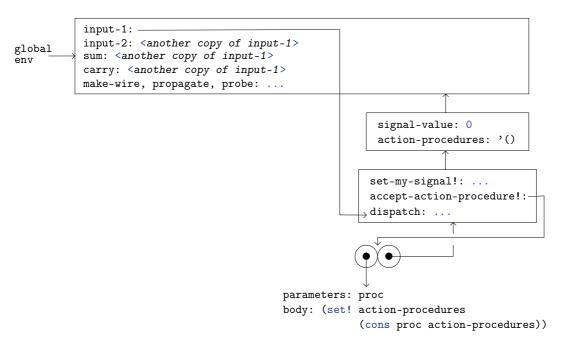


Figure 2. Environments created by defining input-1, input-2, sum and carry.

```
make-wire, propagate, probe: ...
           input-1: ...
global
           input-2: ...
env
           sum:
           carry: <akin to sum>
                                                         signal-value: 0
    E4
                                                         action-procedures: '()
           proc: (lambda ()
                    (newline)
                    ...)
                                                       set-my-signal!: ...
                                                       accept-action-procedure!: ...
     call to accept-action-procedure!
                                                       dispatch: ...
                                            E2
E3 -
                                                 name: 'sum
         wire: sum
                                                 wire: sum
         action-procedure: (lambda ()
                               (newline)
                                                call to probe.
                               . . . )
             call to add-action!
```

Figure 3. Environment created by evaluating (probe 'sum sum).

```
action-procedures))
```

which was the value of (display (get-signal sum)), that is, 0. This in turns gave rise to the situation:

```
(probe 'sum sum)
;Value: ()
```

while one interacting with the interpreter. Another expression for placing probes—(probe 'carry carry)—was evaluated in the same way and the interpreter responsed with the same thing:

```
(probe 'carry carry)
;Value: ()
```

Figure 4 shows the resulting environment structure in the completion of evaluating (probe 'sum sum) and (probe 'carry carry).

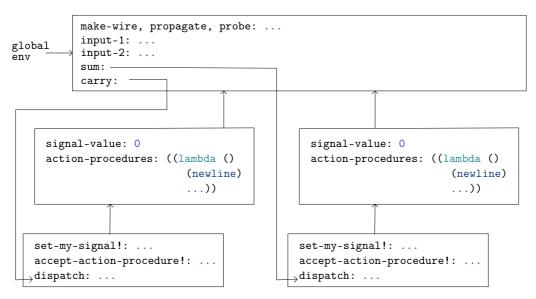


Figure 4. The resulting environment structure in the completion of evaluating (probe 'sum sum) and (probe 'carry carry).

Next we connect the wires in a half-adder circuit, by evaluating the expression (half-adder input-1 input-2 sum carry). As shown in figure 5, this created the procedure object half-adder whose environment was E5 where the local variable d and e are initialized. Then the arguments input-1, input-2, sum and carry were bound onto the formal parameters of half-adder in E6 and evaluated the body of half-adder, that is, things like (or-gate input-1 input-2 d) etc.

In applying or-gate to input-1, input-2 and d, we set up a new environment E7 whose enclosing environment was E6 and evaluated the body of or-gate in E7. To get its job done, or-gate in turns called to add-action! twice with input-1 and input-2 passed as their arguments respectively. The add-action! procedure at this point simply added the or-action-procedure to the action-procedures of input-1 and input-2 without running them, thus nothing yet has been added into the agenda. In other words, the or-gate was plugged into the half-adder without initializing its output. Figure 6 presents us the environement structure in evaluating (or-gate input-1 input-2 d). The

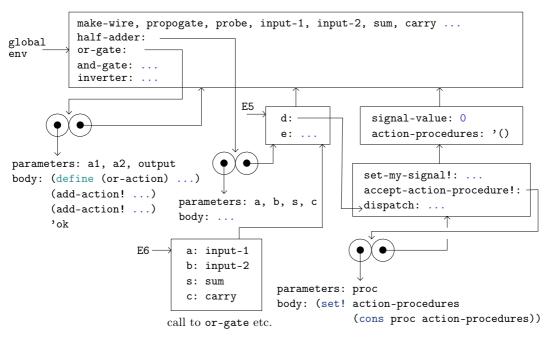


Figure 5. Environments created by applying half-adder to input-1, input-2, sum and carry.

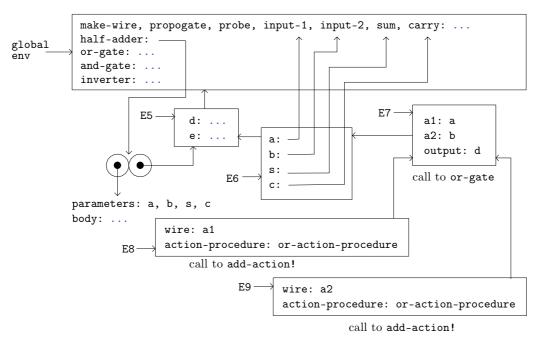


Figure 6. Environments built in evaluating (or-gate input-1 input-2 d).

subsequent expression inside half-adder was evaluated similarly and figure 7 together with figure 8 shows the situation after the call to half-adder.

Notice that the signal on the wire e had gone wrong to stay at 0, which was supposed to be 1 at this point. Because when the inverter was connected to the carry, it simply added the invert-input procedure to the action-procedures of the latter one without running. This was precisely the same case we encountered in hooking the or-gate up to the half-adder. We shall see the problems arised by this non-initialization wiring soon, although nothing exceptional appeals when we interact with the interpreter at present:

```
(half-adder input-1 input-2 sum carry)
; Value: ok
```

So far, the agenda still remains empty.

As we change the signal on input-1 in figure 7 from 0 to 1:

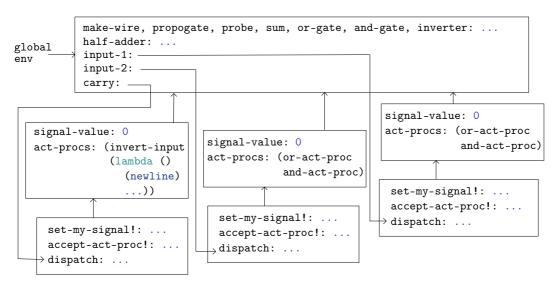


Figure 7. Environments after the call to half-adder. (continued on next figure)

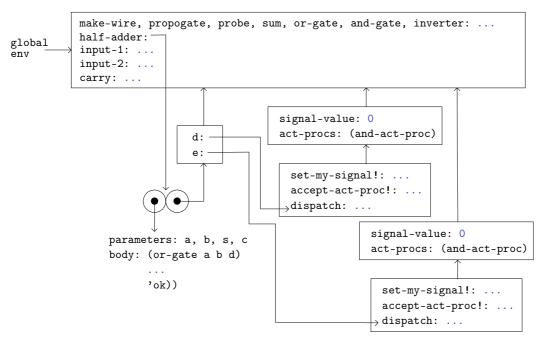


Figure 8. Environments after the call to half-adder. (continued)

```
(set-signal! input-1 1)
;Value: done
```

All the action-procedures it possessed were run, dued to what stated by set-my-signal!, one of the internal procedures of make-wire. Note that it was at this juncture that the actions of setting the signal on d and carry were added into the agenda. The signal on carry would stay at 0 even its signal was reset by the and-gate after an and-gate-delay, for the value of input-2 remained to be 0. Hence, the inverter would not be triggered and the signal on e stayed to be 0. However, D would change its signal from 0 to 1 at time 5 for the sake of the or-gate. This mutation further triggered sum to reset its signal at one and-gate-delay later, that is, at time 8, since the signal on e remained 0, the signal on sum stayed at 0. Figure 9 shows the states of wires after evaluating (set-signal! input-1 1) and figure 10 shows contents of agenda at this point. Since neither the sum nor the carry changes its value during the propagation, nothing more than 'done is prompted when we run the simulation:

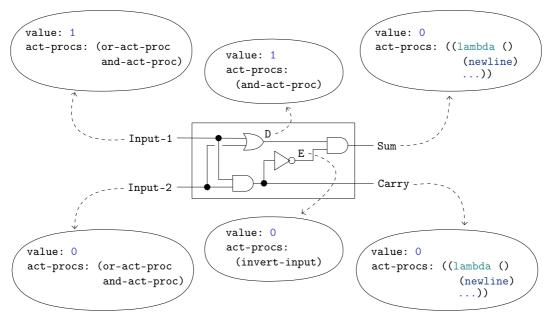


Figure 9. States of wires after evaluating (set-signal! input-1 1).

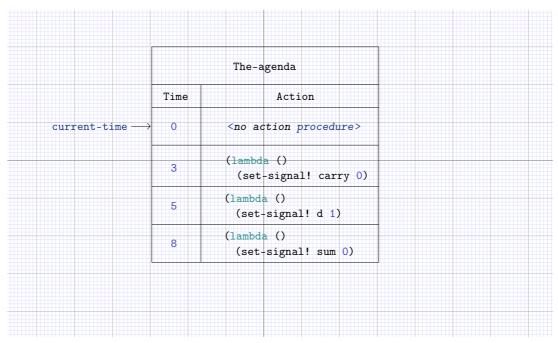


Figure 10. Contents of the-agenda after evaluating (set-signal! input-1 1).

(propagate)
;Value: done

Currently, we are 8 time units from the begining of the simulation.

Finally, let's come to analyse how the process evolves when we set the signal on input-2 to 1. Figure 11 gives us a clear view of the evolution of process after we changed the signal on input-2.

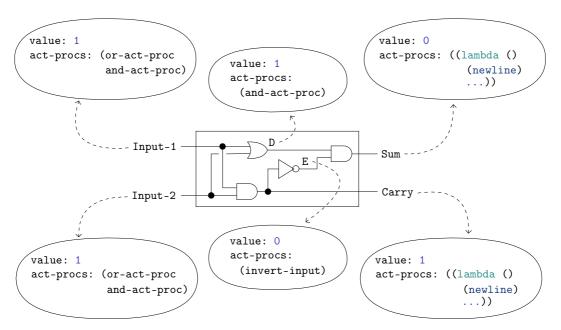


Figure 11. States of wires after evaluating (set-signal! input-2 1)

See that when the signal on input-2 was altered, the action-procedures it contained would be run immediately. Evaluating or-action-procedure added (lambda () (set-signal d 1)) into the agenda, and by scheduling, it set the signal on d to be 1 after an or-gate-delay. Likewise, evaluating and-action-procedure caused the agenda to extend with an action procedure (lambda ()

(set-signal carry 1)) which changed the signal on carry from 0 to 1 after an and-gate-delay. In other words, carry changed its signal to 1 at time 11 and d regenerate the signal of 1 at time 13. The mutation of the signal on carry further trigger the inverter to reset its output, e, to be 0 also at time 13. Since neither of d and e changes its value in this process, the and-gate connected to sum won't be triggered. Figure 12 shows the contents of the agenda just before we run the simulation. To confirm

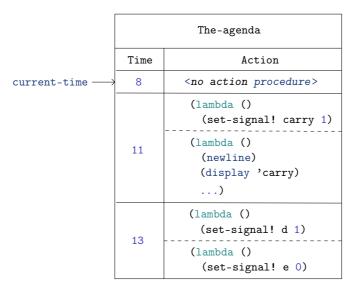


Figure 12. Contents of the-agenda before running the simulation for a second time.

our prediction, we just set the signal of input-2 and allow the value to propagate:

```
(set-signal! input-2 1)
;Value: done
(propagate)
carry 11 New-value = 1
;Value: done
```