SICP section 3.3.4

October 08, 2007 at 06:31 **Tags** SICP

The code for this section is in Common Lisp.

As I mentioned in previous sections, while using a function that returns a `dispatch` procedure is probably accepted as a basic method of implementing objects with state in Scheme, it is not commonly used in Common Lisp. In fact, CL has a full fledged object oriented system called CLOS that contains many tools that help build extensible object systems. So I've decided to give it a try for this section, and coded `wire` as a CLOS class.

I will present the code here without explaning too much, because there are better sources for that. For example, the excellent Practical Common Lisp book[1] has a good explanation of the topic in chapters 16 and 17.

```
(defclass wire ()
  ( (signal-value
      :initform 0
      :accessor signal-value
      :documentation
        "The signal associated with the wire")
    (action-procedures
      :initform '()
      :accessor action-procedures
      :documentation
        "Procedures that are executed when there's
        an event on the wire")))

(defun make-wire ()
  (make-instance 'wire))

(defmethod (setf signal-value) :around (new-value (w wire))
  (let ((old-value (signal-value w)))
    (prog1 (call-next-method)
      (unless (= old-value new-value)
        (call-each (action-procedures w))))))

(defgeneric add-action! (w proc)
  (:documentation "Add a new action procedure"))

(defmethod add-action! ((w wire) proc)
  (push proc (action-procedures w))
  (funcall proc))

(defmethod print-object ((w wire) stream)
  (print-unreadable-object (w stream :type t)
    (with-slots ( (sv signal-value)
                  (ap action-procedures)) w
      (format stream "Sig: ~a, N procs: ~a"
        sv (length ap)))))

(defun call-each (procs)
  (dolist (proc procs)
    (funcall proc)))
```

Most of it is quite straightforward. The only non-trivial part is the definition of the setter for `signal-value`. I defined the slot `signal-value` with an accessor of a similar name[2]. This is in order to comply to the CL convention of accessing the slot as follows:

```
(defvar x (make-wire))

(setf (signal-value x) 1) ; writer
(print (signal-value x)) ; reader
```

However, the writer of `signal-value` is not trivial – it must check for a change in the value and notify the relevant action procedures. So I used the `:around` feature of CLOS to add custom code

on calls of the writer. Features like this one make CLOS a very flexible system, and you should definitely check it out for writing objects with state.

The code for `inverter` and `and-gate` is pretty much the same:

```
(defvar *inverter-delay* 2)

(defun inverter (input output)
  (flet ((invert-input ()
           (let ((new-value
                   (logical-not
                     (signal-value input))))
             (after-delay
               *inverter-delay*
               (lambda ()
                 (setf
                   (signal-value output)
                   new-value))))))
    (add-action! input #'invert-input)
    'ok))

(defun logical-not (s)
  (case s
    (0 1)
    (1 0)
    (otherwise (error "Invalid signal" s))))

(defvar *and-gate-delay* 3)

(defun and-gate (a1 a2 output)
  (flet ((and-action ()
           (let ((new-value
                   (logand
                     (signal-value a1)
                     (signal-value a2))))
             (after-delay
               *and-gate-delay*
               (lambda ()
                 (setf
                   (signal-value output)
                   new-value))))))
    (add-action! a1 #'and-action)
    (add-action! a2 #'and-action)
    'ok))
```

Exercise 3.28

```
(defvar *or-gate-delay* 3)

(defun or-gate (a1 a2 output)
  (flet ((or-action ()
           (let ((new-value
                   (logior
                     (signal-value a1)
                     (signal-value a2))))
             (after-delay
               *or-gate-delay*
               (lambda ()
                 (setf
                   (signal-value output)
                   new-value))))))
    (add-action! a1 #'or-action)
    (add-action! a2 #'or-action)
    'ok))
```

Exercise 3.29

Using De-Morgan's rules, we know that `A or B` is equivalent to `not ((not A) and (not B))`. So:

```
(defun or-gate-dm (a1 a2 output)
  (with-wires (na1 na2 and12)
    (inverter a1 na1)
    (inverter a2 na2)
    (and-gate na1 na2 and12)
    (inverter and12 output))
    'ok)
```

But wait a sec, what is `with-wires` ? Ah, that… it's really about time we started using some of the advanced features of Lisp. In this case the most powerful of them all – macros !

```
(defmacro with-wires ((&rest names) &body body)
  `(let ,(loop for n in names collect `(,n (make-wire)))
    ,@body))
```

This macro makes our code less repetitive, and hence much more pleasant to write. For example, `half-adder` is defined as follows (Compare it with the definition in the book, writing all those `make-wire` calls manually):

```
(defun half-adder (a b s c)
  (with-wires (d e)
    (or-gate a b d)
    (and-gate a b c)
    (inverter c e)
    (and-gate d e s)
    'ok))
```

The delay of this or-gate is the delay of the and-gate plus twice the delay of the inverter.

Exercise 3.30

```
(defun ripple-carry-adder (la lb ls c)
  (let ((n (length la)))
    (unless (= n (length lb) (length ls))
      (error "Expecting all lists of same length"))
    (labels (
      (ripple-build (la lb lcin ls lcout)
        (unless (null la)
          (full-adder (car la) (car lb) (car lcin)
                      (car ls) (car lcout))
          (ripple-build (cdr la) (cdr lb) (cdr lcin)
                        (cdr ls) (cdr lcout)))))

      (let ((lcin '()) (lcout '()))
        (dotimes (i n)
          (let ((ci (make-wire)))
            (push ci lcin)
            (push ci lcout)))
        (push c lcout)
        (ripple-build la lb lcin ls lcout)))))
```

Agenda

Here's the CL code for `agenda`. It begins by loading the code from exercise 3.21 where queue operations are implemented:

```
;;;;;;;;;;;;;;;;;;;; Agenda ;;;;;;;;;;;;;;;;;;;;
;;

(load "ex_3_21") ; queue

(defun make-time-segment (time queue)
  (cons time queue))

(defun segment-time (s) (car s))
(defun segment-queue (s) (cdr s))

(defun make-agenda () (list 0))
(defun cur-time (agenda) (car agenda))
(defun set-cur-time! (agenda time)
  (setf (car agenda) time))
(defun segments (agenda) (cdr agenda))
(defun set-segments! (agenda segments)
  (setf (cdr agenda) segments))
(defun first-segment (agenda)
  (car (segments agenda)))
(defun rest-segments (agenda)
  (cdr (segments agenda)))

(defun empty-agenda? (agenda)
  (null (segments agenda)))
```

```
(defun add-to-agenda! (time action agenda)
  (labels (
    (belongs-before? (segments)
      (or (null segments)
          (< time (segment-time (car segments)))))
    (make-new-time-segment (time action)
      (let ((q (make-queue)))
        (insert-queue! q action)
        (make-time-segment time q)))
    (add-to-segments! (segments)
      (if (= (segment-time (car segments)) time)
        (insert-queue!
          (segment-queue (car segments))
          action)
        (let ((rest (cdr segments)))
          (if (belongs-before? rest)
            (setf
              (cdr segments)
              (cons
                (make-new-time-segment time action)
                (cdr segments)))
            (add-to-segments! rest))))))

    (let ((segments (segments agenda)))
      (if (belongs-before? segments)
        (set-segments!
          agenda
          (cons
            (make-new-time-segment time action)
            segments))
        (add-to-segments! segments)))))

(defun remove-first-agenda-item! (agenda)
  (let ((q (segment-queue (first-segment agenda))))
    (delete-queue! q)
    (if (empty-queue? q)
      (set-segments! agenda (rest-segments agenda)))))

(defun first-agenda-item (agenda)
  (if (empty-agenda? agenda)
    (error "Agenda is empty")
    (let ((first-seg (first-segment agenda)))
      (set-cur-time! agenda (segment-time first-seg))
      (front-queue (segment-queue first-seg)))))

(defun print-agenda-details (agenda)
  (format t "--------~%Agenda:~%Cur time: ~a~%"
    (cur-time agenda))
  (dolist (seg (segments agenda))
    (format t "Seg. Time: ~a" (segment-time seg))
    (format t ", Queue: ~a~%" (segment-queue seg))
  (format t "--------~%")))
```

```
;;;;;;;;;;;;;;;;;; Simulation ;;;;;;;;;;;;;;;;;;
;;

(defvar *the-agenda* (make-agenda))

(defun after-delay (delay action)
  (add-to-agenda!
    (+ delay (cur-time *the-agenda*))
    action
    *the-agenda*))

(defun propagate ()
  (if (empty-agenda? *the-agenda*)
    'done
    (let ((first-item (first-agenda-item *the-agenda*)))
      (funcall first-item)
      (remove-first-agenda-item! *the-agenda*)
      (propagate))))

(defun probe (name wire)
  (add-action! wire
    (lambda ()
      (format t "~a ~a new-value = ~a~%"
        name
        (cur-time *the-agenda*)
        (signal-value wire)))))
```

Exercise 3.31

By calling the action procedure, it makes sure that the operation of the gate is actually defined. Consider the inverter gate again:

```
(defun inverter (input output)
  (flet ((invert-input ()
           (let ((new-value
                   (logical-not
                     (signal-value input))))
             (after-delay
               *inverter-delay*
               (lambda ()
                 (setf
                   (signal-value output)
                   new-value))))))
    (add-action! input #'invert-input)
    'ok))
```

When `(inverter x nx)` is executed, `add-action!` is called. It is expected to call `invert-input` once, because otherwise the `after-delay` call won't be executed and won't add the gate operation to the agenda.

What this means is that the *initial* values of the signals won't be propagated through the logic

gates.

Exercise 3.32

The first transition from 0 to 1 would bring the gate's inputs both to 1 momentarily, which would make it switch to 1. But at the same segment, the transition of the other signal from 1 to 0 would make it switch back to 0. It can also be simulated using the current queue implementation, by changing the order of signal assignments. Compare this:

```
(setq x (make-wire))
(setq y (make-wire))
(setq z (make-wire))

(and-gate y x z)

(setf (signal-value x) 1)
(setf (signal-value y) 0)

(propagate)

(setf (signal-value x) 0)
(setf (signal-value y) 1)

(propagate)

=> Z 0 new-value = 0
```

To this (the change is in the order between assignments to `x` and `y` before the 2nd `propagate`):

```
(setq x (make-wire))
(setq y (make-wire))
(setq z (make-wire))

(and-gate y x z)

(setf (signal-value x) 1)
(setf (signal-value y) 0)

(propagate)

(setf (signal-value y) 1)
(setf (signal-value x) 0)

(propagate)

=> Z 0 new-value = 0
=> Z 6 new-value = 1
=> Z 6 new-value = 0
```

In real electronic systems this is called a *hazard*, and can happen if the time difference between the change of two inputs is larger than the propagation delay of the gate, and the intermediate state is different from the final one.

[1] Available for free online.

[2] The final version of this code owes much to very helpful advice I received from the comp.lang.lisp newsgroup.

For comments, please send me an email.

Back to top