CS2135: Macros for Monitors

Kathi Fisler, WPI

February 11, 2003

1 A Non-Graphical Notation for Monitors

In the previous lecture, we developed two data definitions for monitors, hand-translated the traffic light example into both of those definitions, and wrote the code necessary to run the monitors on inputs. As with our slideshow and animation language examples, we would like to have a nicer notation for writing down monitors and use macros to convert that notation into our desired data definitions.

Here's a possible ascii notation for monitors. We want to use $-$ $>$ to depict the arrows and : to separate a state name from its transitions.

```
(define TL-monitor
  (monitor see-red
           (see-red : (green -> see-green)
                      (red -> see-red))
           (see-green : (yellow -> see-yellow)
                        (green -> see-green))
           (see-yellow : (red -> see-red)
                         (yellow -> see-yellow))))
```

Our goal is to write the **define-syntax** expression that will let us use this notation. We will do this exercise twice, once for each data definition we developed for monitors.

Converting to the Structure-Based Language

Recall that our representation of *TL-monitor* in the structures-based language looked as follows [note: I've changed the name of the structure to *monitor-struct* from *monitor* to avoid a naming conflict between the structure and the macro in Scheme.]

```
(define TL-monitor
  (make-monitor-struct
    'see-red
    (list (make-state 'see-red (list (make-trans 'red 'see-red)
                                     (make-trans 'green 'see-green)))
          (make-state 'see-green (list (make-trans 'green 'see-green)
                                       (make-trans 'yellow 'see-yellow)))
          (make-state 'see-yellow (list (make-trans 'yellow 'see-yellow)
                                        (make-trans 'red 'see-red))))))
```

We want to write the monitor macro that to produce this code.

First, let's write the macro up to, but not including, the input pattern:

```
(define-syntax monitor
  (syntax-rules (-> :)
    ))
```

Recall that the items in parentheses after **syntax-rules** are the keywords for the macro. We treat both -¿ and : as keywords because we expect them to appear literally in the input pattern (rather than wanting to treat them as variable names).

Next, let's add the input pattern:

```
(define-syntax monitor
  (syntax-rules (-> :)
    [(monitor initname
              (curr-state : (label -> next-state) ...)
               ...)
     <FILL IN>]))
```

Notice that this much of the macro definition will be the same across both language data definitions; only the output patterns will change between the two macros.

In this case, the output pattern seems pretty easy based on the target definition of *TL-monitor*:

```
(define-syntax monitor
  (syntax-rules (-> :)
    [(monitor initname
              (curr-state : (label -> next-state) ...)
               ...)
     (make-monitor-struct
      initname
      (list (make-state curr-state (list (make-trans label next-state)
                                         ...))
            ...))]))
```

If we put both the macro definition and the macro use in a file and try to run it, what happens? We get an error that *see-red* is undefined. What do you think happened?

Notice that when we wrote the definition of *TL-monitor* by hand (without the macro), we used symbols for *see-red*, etc. Nowhere in the macro do we see the tick-marks that denote symbols. As a result, Scheme is trying to treat *see-red* as an identifier rather than a symbol, hence the error.

We could fix this by using the tick-marks when we use the macro, as follows:

```
(define TL-monitor
  (monitor 'see-red
           ('see-red : ('green -> 'see-green)
                       ('red -> 'see-red))
           ('see-green : ('yellow -> 'see-yellow)
                         ('green -> 'see-green))
           ('see-yellow : ('red -> 'see-red))))
```

That's pretty ugly though, certainly not as nice as the version without the tick-marks (and it exposes Scheme features in the syntax, which we want to avoid).

Alternatively, we can put the tick-marks into the macro definition:

```
(define-syntax monitor
  (syntax-rules (-> :)
    [(monitor initname
              (curr-state : (label -> next-state) ...)
               ...)
     (make-monitor-struct
      'initname
      (list (make-state 'curr-state (list (make-trans 'label 'next-state)
                                          ...))
            ...))]))
```

Note that we don't put the tick-marks into the input pattern (since they aren't in the input), only in the output pattern.

If you're thinking carefully about this, you're probably wondering how on earth this could work – wouldn't the tick-marks cause the macro to put the symbols like 'initname and 'curr-state into the output? After all, in our work with Scheme to date, the tick-marks prevented something from being evaluated! The real answer is that the tick-mark is actually an interesting Scheme operator named **quote** that can do a lot more than what we've done with it in this course. As always, stop by my office sometime if you want to know more, or look up **quote** in the DrScheme HelpDesk.

Converting to the Function-Based Language

Now let's write the macro to create the function-based language for automata. We start with the same header information and input pattern:

```
(define-syntax monitor
  (syntax-rules (-> :)
    [(monitor initname
              (curr-state : (label -> next-state) ...)
              ...)
     <FILL IN>]))
```

How do we fill in the output pattern? As a reminder, here's the version we created by hand:

```
(define see-red
  (lambda (samples)
    (cond [(empty? samples) 'okay]
          [(cons? samples)
           (cond [(symbol=? (first samples) 'red) (see-red (rest samples))]
                 [(symbol=? (first samples) 'green) (see-green (rest samples))]
                 [else 'error])])))

(define see-yellow
  (lambda (samples)
    (cond [(empty? samples) 'okay]
          [(cons? samples)
           (cond [(symbol=? (first samples) 'yellow) (see-yellow (rest samples))]
                 [(symbol=? (first samples) 'red) (see-red (rest samples))]
                 [else 'error])])))

(define see-green
  (lambda (samples)
    (cond [(empty? samples) 'okay]
          [(cons? samples)
           (cond [(symbol=? (first samples) 'green) (see-green (rest samples))]
                 [(symbol=? (first samples) 'yellow) (see-yellow (rest samples))]
                 [else 'error])])))

(define TL-monitor see-red)
```

We have a problem. The output pattern can contain only one expression, but this version has four! Before we can fill in the output pattern in the macro, we need to figure out how to merge all of the code we see here into a single expression.

Recall that when we designed the slide language, we used **let** to gather all of the slide definitions into a single expression. Perhaps we can do the same thing here:

```
(define TL-monitor
  (let ([see-red
```

```
    (lambda (samples)
      (cond [(empty? samples) 'okay]
            [(cons? samples)
             (cond [(symbol=? (first samples) 'red) (see-red (rest samples))]
                   [(symbol=? (first samples) 'green) (see-green (rest samples))]
                   [else 'error])]))]
   [see-yellow
    (lambda (samples)
      (cond [(empty? samples) 'okay]
            [(cons? samples)
             (cond [(symbol=? (first samples) 'yellow) (see-yellow (rest samples))]
                   [(symbol=? (first samples) 'red) (see-red (rest samples))]
                   [else 'error])]))]
   [see-green
    (lambda (samples)
      (cond [(empty? samples) 'okay]
            [(cons? samples)
             (cond [(symbol=? (first samples) 'green) (see-green (rest samples))]
                   [(symbol=? (first samples) 'yellow) (see-yellow (rest samples))]
                   [else 'error])]))])
  see-red))
```

If we do this and execute (*TL-monitor* (*list* 'red 'green 'red)) at the prompt, we get an error that the use of *see-red* in the sixth line (the (*symbol=?* (*first samples*) 'red) line) is undefined. Is this a quote/tick-mark problem again? No, because in this version we *want see-red* to be an identifier. What's going on?

This example introduces a property of **let**, and a general topic in programming languages called *scoping*. The short answer is that you can't define recursive functions using **let** (this explanation is a bit too specific – we'll get to the more precise definition of the problem in a moment). However, Scheme gives you a similar construct for defining local variables called **letrec** that does let you define recursive procedurs (the "rec" part stands for "recursive"). Changing the **let** to **letrec** (and leaving everything else alone) yields the following code:

```
(define TL-monitor
  (letrec ([see-red
            (lambda (samples)
              (cond [(empty? samples) 'okay]
                    [(cons? samples)
                     (cond [(symbol=? (first samples) 'red) (see-red (rest samples))]
                           [(symbol=? (first samples) 'green) (see-green (rest samples))]
                           [else 'error])]))]
           [see-yellow
            (lambda (samples)
              (cond [(empty? samples) 'okay]
                    [(cons? samples)
                     (cond [(symbol=? (first samples) 'yellow) (see-yellow (rest samples))]
                           [(symbol=? (first samples) 'red) (see-red (rest samples))]
                           [else 'error])]))]
           [see-green
            (lambda (samples)
              (cond [(empty? samples) 'okay]
                    [(cons? samples)
                     (cond [(symbol=? (first samples) 'green) (see-green (rest samples))]
                           [(symbol=? (first samples) 'yellow) (see-yellow (rest samples))]
                           [else 'error])]))])
    see-red))
```

This definition of *TL-monitor* runs just as the original did.

Now that we have all of *TL-monitor* in a single expression, we can develop the output pattern in the macro to produce this expression:

```
(define-syntax monitor
  (syntax-rules (-> :)
    [(monitor initname
              (curr-state : (label -> next-state) ...)
              ...)
     (letrec ([curr-state
               (lambda (samples)
                 (cond [(empty? samples) 'okay]
                       [else
                        (cond [(symbol=? (first samples) 'label) (next-state (rest samples))]
                              ...
                              [else 'error])]))]
              ...)
       initname)]))
```

Look at where we used **quote** (the tick-marks) in this macro as compared to the macro in the structure version. In the structure version, we wanted to turn the uses of *curr-state*, *label*, and *next-state* in the output pattern into symbols. In this macro, only the *label* uses get quoted; *curr-state* and *next-state* are left as identifiers. Why? Because now *curr-state* and *next-state are* identifiers – they are variable names defined in the **letrec** statement. What do you think would happen if we did put the quote on them? We'd get an error that we tried to use a symbol as a function (when we call *next-state* as a function).

With this example, we're getting into some pretty neat uses of macros. We're able to use non-alpha-numeric characters as keywords in macros, which can help a lot with defining custom notations. Our notations are looking less and less Scheme-like (are you *really* still seeing the parens as much as you were when we started with Scheme?), and we're writing more sophisticated output patterns. Once you see how to write macros like the **monitor** macro, you're well on your way to really using Scheme to help you define custom languages.