Generating Extra Output

**Summary:** We consider ways in which to have Scheme write some output (in addition to generating responses).

**Contents:**
- Introduction
- write
- display
- newline
- Detour: Sequencing Multiple Things
- Displaying Values vs. Returning Values
- Output vs. Return, Revisited: When To Display Values

**Introduction**

The Scheme model we have worked with so far is both simple and straightforward: The user types a Scheme expression, the computer thinks for awhile, and then prints the value of the expression. However, some programs may benefit from additional output printed while the program is computing. For example, one helpful technique for understanding recursive procedures is to print out the current call at each step. More importantly, output procedures (along with corresponding input procedures) permit programmers to write programs that interact with the user.

Scheme provides four basic output operations: write, display, newline, and write-char. We discuss all but write-char below.

**write**

The `write` procedure takes one argument and prints out a representation of that argument. The nature of the value that it *returns* is unspecified (under DrScheme, for instance, it’s the special “void” value) -- the printing is a side effect of the evaluation of the call to `write`, not its result.

DrScheme also encloses the material that `write` prints out inside an interaction box. You can distinguish user input from program output in an interaction box by its color: User input is displayed in green, program output in purple. Both are distinguished from DrScheme’s usual way of exhibiting the value of an expression, which is to print it in dark blue without drawing an interaction box.

```scheme
> (define my-input (read))
116
> my-input
116
> (write my-input)
116
```
**display**

The display procedure also takes one argument and prints out a representation of it, but it differs from write in that it does not enclose the representations of strings in double quotation marks and does not print the mesh-backslash combination when displaying a character:

> (display "sample string")
> sample string
> (write "sample string")
> "sample string"
> (DISPLAY #\A)
> A
> (write #\A)
> #\A

**newline**

The newline procedure takes no arguments and returns an unspecified value; as a side effect, it terminates the current output line. Successive calls to write and display normally produce output that is all strung together on one line. Calls to newline are used to break up such output into separate lines.

> (begin
    (display "all-")
    (display "on-")
    (display "one-")
    (display "line")
    (newline)
    (display "This is on a ")
    (display "separate line.")
    (newline))

```
all-on-one-line
This is on a separate line.
```

The call (newline) has exactly the same effect as (display #\newline), for which you can consider it a convenient shorthand.

**Detour: Sequencing Multiple Things**

In most of the code we’ve written to date, each expression does one thing (most typically, computes a value). Once we start using output, we often need to sequence operations. For example, we might display a value and then follow that value with a newline.

```
(display "Hello.")
(newline)
```

In many contexts, you can simply put the instructions in sequence. For example, if the body of a procedure contains multiple expressions, Scheme knows to execute them in turn.
You can also have multiple instructions in the consequent position of a \texttt{cond}.

\begin{alltt}
(define greet
  (lambda (name)
    (cond
      ((member name (list "Janet" "Sam"))
        (display "Go away, ")
        (display name)
        (display ".")
        (newline)
        (display "CS teachers not allowed.")
        (newline))
      ((member name (list "Emily" "Eryn" "Ian"))
        (display "So, how many typing errors did you observe today?")
        (newline))
      (else
        (display "Hello, ")
        (display name)
        (display ".")
        (newline)))))
\end{alltt}

Here are some examples that show the three cases.

\begin{verbatim}
> (greet "Sam")
Go away, Sam.
CS teachers not allowed.
> (greet "Joe")
Hello, Joe.
> (greet "Ian")
So, how many typing errors did you observe today?
\end{verbatim}

However, you can’t put multiple actions in the consequent or alternate positions in an \texttt{if} statement, since there would then be no way for Scheme to know which actions belonged to the consequent and which to the alternate. What do we do instead? We use the \texttt{begin} structure, whose primary intent is to group and sequence actions.

The form of \texttt{begin} is fairly simple:

\begin{verbatim}
(begin exp1 exp2 ... expn)
\end{verbatim}

This statement evaluates each expression in turn (generating output when the statement generates output) and then returns the value of the last expression.

I use \texttt{begin} most frequently when writing output in an \texttt{if} statement, as in the following.
(define goodbye
  (lambda (name)
    (display "Goodbye, ")
    (display name)
    (display ".")
    (newline)
    (if (equal? name "Sam")
        (begin
            (display "Good luck getting your work done.")
            (newline))
        (begin
            (display "I hope to see you soon.")
            (newline))))

## Displaying Values vs. Returning Values

As you may have noted, when we run a procedure, we can now see its results in two ways. The results can be displayed (with the `display` or `write` command) or the results can appear simply because the procedure returns them. Consider, for example, the following pair of similar procedures.

```scheme
(define square1
  (lambda (x)
    (* x x)))

(define square2
  (lambda (x)
    (display (* x x))
    (newline)))
```

When we run them on the same value, we seem to get the same behavior in terms of what we see on the screen.

```
> (square1 5)
25
> (square2 5)
25
```

However, the behavior is actually quite different. In the first case, we are returning a value, a value that can be used by another procedure. Since it’s not used by another procedure, Scheme simply displays it. In the second case, we are explicitly displaying something, and no other procedure can access that value.

To see the difference, let’s try adding one to the result.

```
> (+ 1 (square1 5))
26
> (+ 1 (square2 5))
>25
+: expects type <number> as 2nd argument, given: #<void>; other arguments were: 1
```

In general, unless your goal is to print something to the screen, you should compute the value, rather than using the additional `display`.
Output vs. Return, Revisited: When To Display Values

The previous section seems to suggest that you should often simply return a value, rather than using display. If your main goal is computation, that’s certainly the case. However, there are also many times that your goal is output.

For example, consider the problem of making a simple children’s story that reads Daniel likes ____ for a variety of things. Here’s how we might write a procedure to write such stories using display.

```scheme
(define story1
  (lambda (lst)
    (if (null? lst)
      (begin
        (display "Daniel likes other things, too.")
        (newline))
      (begin
        (display "Daniel likes ")
        (display (car lst))
        (display ".")
        (newline)
        (story1 (cdr lst))))))
```

We can then use it to tell simple stories.

```scheme
> (story1 (list "mom" "dad" "jazz that’s trad" "brother" "sister" "playing twister"))
Daniel likes mom.
Daniel likes dad.
Daniel likes jazz that’s trad.
Daniel likes brother.
Daniel likes sister.
Daniel likes playing twister.
Daniel likes other things, too.
```

Now, if we just used the strings, we won’t get what we expect.

```scheme
(define story2
  (lambda (lst)
    (if (null? lst)
      "Daniel likes other things, too.\n"
      (begin
        (string-append "Daniel likes 

" (car lst)

".\n")
        (story2 (cdr lst))))))
```

```scheme
> (story2 (list "mom" "dad" "jazz that’s trad" "brother" "sister" "playing twister"))
"Daniel likes other things, too.\n"
```

Why don’t we get the intermediate strings? Because they never get displayed, and Scheme only uses the last value in a begin block.
You’ll note another problem, too: We see the quotation marks and the backslash \n. Let’s fix the first problem and then come back to the second one. We can fix the first problem by appending the “Daniel likes ...” sentence to the result of the recursive call.

(define story3
  (lambda (lst)
    (if (null? lst)
      "Daniel likes other things, too.\n"
      (begin
        (string-append "Daniel likes "
                      (car lst)
                      ".\n"
                      (story3 (cdr lst)))))))

> (story3 (list "mom" "dad" "jazz that's trad" "brother" "sister" "playing twister"))
"Daniel likes mom.\nDaniel likes dad.\nDaniel likes jazz that's trad.\nDaniel likes brother.\nDaniel likes sister.\nDaniel likes playing twister.\nDaniel likes other things, too.\n"

Better, but not very readable. And that suggests the other reason we might use display. If we’ve built a long and complicated string (with \n representing carriage return), we can make it look nice using display.

> (display (story3 (list "mom" "dad" "jazz that's trad" "brother" "sister" "playing twister")))
Daniel likes mom.
Daniel likes dad.
Daniel likes jazz that’s trad.
Daniel likes brother.
Daniel likes sister.
Daniel likes playing twister.
Daniel likes other things, too.