

# Imperative Programming in Scheme

- Sequencing using `(begin arg1 ... argn)`
- Implicit sequencing

## Input-output in Scheme

The function `(display arg)` displays *arg* on the standard output stream. Example:

```
> (display "Hello Dan")
Hello Dan
> (display #\y)
y
> (display (list (+ 1 2 3) #\space '(a b c)))
(6 (a b c))
```

The function `(write arg)` is similar, except that *arg* is displayed *literally*.

```
> (write "Hello Dan")
"Hello Dan"
> (write #\y)
#\y
> (write (list (+ 1 2 3) #\space '(a b c)))
(6 #\space (a b c))
```

The function `(read)` reads an expression from standard input.

```
(define add3
  (lambda ()
    (display "x? ")
    (+ 3 (read))))
```

```
> (add3)
```

```
5
```

```
8
```

The function `(read-char)` reads a single character; `(eof-object? arg)` returns `#t` if *arg* is an end of file object.

These input/output functions can also access files:

```
> (define out (open-output-file "my-output"))  
out  
> (display "Hello world" out)  
> (display #\newline out)  
> (newline out)  
> (close-output-port out)
```

## Mutating data structures

`(set-car! obj new-val)` and `(set-cdr! obj new-val)` change the values of the car and cdr of *obj*, respectively.

```
> (define ls '(a b c))
(a b c)
> (set-cdr! ls '(y z))
(a y z)
> ls
(a y z)
> (set-car! ls 'x)
(x y z)
```

`(vector-set! v o value)` assigns a new value to the element of vector *v* that appears in offset *o*:

```
> (define v (vector 1 2 3))
v
> (vector-set! v 1 9)
#(1 9 3)
> v
#(1 9 3)
```

`(set! var exp)` assigns the value of *exp* to variable *var*.

```
> (define x 10)
```

```
x
```

```
> (set! x 20)
```

```
> x
```

```
20
```

## Sharing Local Variables

```
(define add1 '*)
(define sub1 '*)
(define reset '*)

(let ((count 0))
  (set! add1 (lambda ()
               (set! count (+ count 1)) count))
  (set! sub1 (lambda ()
               (set! count (- count 1)) count))
  (set! reset (lambda ()
                 (set! count 0) count))))
```

```
> (add1)
1
> (add1)
2
> (sub1)
1
> (reset)
0
```

# Streams

A *stream* is a sequence of values that can be accessed before every element in the sequence has been defined. Common examples are “standard input” and “standard output.”

Let  $s$  denote a non-empty stream, let  $v$  denote a value, and let  $p$  denote a procedure that accepts no arguments, and returns a new stream. Then, Appendix F in *EOPL* implements the stream ADT:

`(stream-car  $s$ )` returns the next element in  $s$ .

`(stream-cdr  $s$ )` returns the stream that remains after the first element in  $s$  has been removed.

`(make-stream  $v$   $p$ )` returns a stream that contains  $v$  for its next value, and the return value of  $p$  for its remaining values.

`the-null-stream` returns the empty stream.

`(stream-null?  $arg$ )` returns `#t` if  $arg$  is an empty stream.

# Thunks

A *thunk* is a procedure that accepts no arguments, and returns a data structure.

Why is this useful?

## Stream Implementation

Represent the stream with a cons cell. Let the car contain the next value, let the cdr contain a thunk that generates the rest of the stream.

```
(define stream-car car)
```

```
(define stream-cdr  
  (lambda (stream)  
    ((cdr stream))))
```

```
(define make-stream  
  (lambda (value thunk)  
    (cons value thunk)))
```

```
(define the-null-stream  
  (make-stream "end-of-stream"  
              (lambda () the-null-stream)))
```

# Memoization

It is usually efficient to evaluate functions no more than once.

```
(define stream-cdr
  (lambda (stream)
    (if (pair? (cdr stream))
        (cdr stream)
        (let ((s ((cdr stream))))
            (set-cdr! stream s)
            s))))
```