Side-Effects!

Side-effects are:

- Doing I/O (read or write).
- Changing a variable value.
- Crashing: `(/ 1 0)`.

So far we have only seen the purely functional subset of Scheme.

Input and Output

write/display: outputs a scheme object.
E.g. `(display "hello world")`.

newline: outputs a newline.

write-char: outputs a character.

read: reads in a scheme object.

read-char: reads in a character.

readline: (Guile only) reads in a line as a string.

Setting Variable Values

In general commands that change values end in `!`:

set! sets variable value: `(set! x 10)`

set-car! sets car of existing `cons` cell.

set-cdr! sets cdr of existing `cons` cell.

Use these sparingly!

Sequencing Operations

Use `begin` to sequence expressions:

```
(define x 0)
(define y 0)
(begin
   (display "enter two numbers: ")
   (set! x (read))
   (set! y (read))
   (display "the sum is ")
   (display (+ x y))
   (newline)
)
```

Return value is value of last expression.

With no side-effects sequencing expressions is not useful:

```
(begin
   (+ 1 2 3)
   (car '(1 2 3))
)
```

Sequencing Operations (cont)

Special forms `lambda`, `let`, and `cond` allow for sequencing operations:

```
(define (welcome)
   (display "hello world")
   (newline)
)

(cond (#f (display "never") 0)
      (#t (display "always") 1)
)

(let ((x 5))
     (display "inside let\n")
     (+ x 10)
)
```

Value of sequence is value of last expression.

Examples of `set!`

```
(define x 10)
(define (x-plus y) (+ x y)

(x-plus 2)

(set! x 100)

(x-plus 2)
```


`fib` Example

Ever wonder how many times recursive `(fib 30)` calls `+`?

```
(define (fib n)
    (if (<= n 1)
        n
        (+ (fib (- n 1)) (fib (- n 2)))
    )
)

(define plus +)
(define count 0)
(define (+ . args)
   (set! count (plus 1 count))
   (apply plus args)
)

(fib 30)                 => 832040
count                    => 1346268
```


List Surgery

Can set `car` and `cdr` of a `cons` cell with `set-car!` and `set-cdr!`.

```
(define x '(1 2 3))
(define y '(a b c))
(define z (cdr x))
(define w (append x y))

(set-cdr! x '(200 300))
(set-cdr! (cdr y) '(ccc))
(set-car! z 99)

x
y
z
w
```

Note: cannot `set-c[ad]r!` the empty list. Why?

Circular Lists

```
(define x '(1 2 3))
(set-cdr! (cddr x) x)

x
```


`set!` vs `set-car!`

`set!` sets binding in environment.

`set-car!` sets binding in heap.

Compare:

```
(define (fix-car1 lst)
   (set! lst (cons 10 (cdr lst)))
)

(define (fix-car2 lst)
   (set-car! lst 10)
)

(define x '(1 2 3))
(define y '(1 2 3))

(fix-car1 x)
(fix-car2 y)

x
y
```