Class 5: Breaking the Functional Paradigm

SI 413 - Programming Languages and Implementation

Dr. Daniel S. Roche

United States Naval Academy

Fall 2011

Side Effects

Remember the intro to the Scheme standard:

> Scheme is a statically scoped and properly tail-recursive dialect of the Lisp programming language invented by Guy Lewis Steele Jr. and Gerald Jay Sussman. It was designed to have an exceptionally clear and simple semantics and few different ways to form expressions. A wide variety of programming paradigms, including functional, imperative, and message passing styles, find convenient expression in Scheme.

What do we have to give up to get side effects?

Side Effects

Remember the intro to the Scheme standard:

> Scheme is a statically scoped and properly tail-recursive dialect of the Lisp programming language invented by Guy Lewis Steele Jr. and Gerald Jay Sussman. It was designed to have an exceptionally clear and simple semantics and few different ways to form expressions. A wide variety of programming paradigms, including functional, imperative, and message passing styles, find convenient expression in Scheme.

What do we have to give up to get side effects? *referential transparency*

Controlling Output

Displaying text to the screen is a kind of side effect.

Here are some useful functions for screen output:

- `(display X)`
- `(newline)`
- `(printf format args...)`
  The catch-all format flag is `~a`.

(Note: Strings in Scheme are made using double quotes, like `"This is a string"`.)

Structuring code with side-effects

With side effects, we have to violate the one-expression-per-function rule.

- `(void)` Is a special construct in Scheme that returns *nothing*.
  This is actually what gets returned by things like `(newline)`.

- `(begin exp1 exp2 ...)`
  This evaluates all the given expressions, sequentially, and
  only returns the value of the last expression.
  So all expressions but the last are only there for side effects.

Exercises

1. Write a function `(print-height inches)` that takes a number of inches and prints the feet and inches.
   For instance, calling `(print-height 73)` should print
   `6 feet 1 inches`

2. Write a function `(print-reverse L)` that takes a list `L` and prints its elements in reverse, one per line.
   For instance, calling `(print-reverse '(1 2 3))` should print
   `3`
   `2`
   `1`

Exercises

1.
```
(define (ph2 n)
  (printf "~a feet ~a inches\n)"
          (quotient n 12)
          (remainder n 12)))
```

Exercises

1.
```
(define (ph2 n)
  (printf "~a feet ~a inches\n)"
          (quotient n 12)
          (remainder n 12)))
```

2.
```
(define (print-reverse L)
  (if (null? L)
      (void)
      (begin (print-reverse (cdr L))
             (display (car L))
             (newline))))
```

Mutation!

The built-in special form `(set! x val)`
changes the value of `x` to be `val`.

Mutation!

The built-in special form `(set! x val)`
changes the value of `x` to be `val`.

Say we want a function that will print out how many times it's been called. The following factory produces one of those:

```
(define (make-counter)
  (let ((count 0))
    (lambda ()
      (set! count (+ 1 count))
      (display count)
      (newline))))
```

Closures

Notice that `make-counter` makes a different `count` variable each time it is called.

This is because each `lambda` call produces a *closure* — the function along with its referencing environment.

Closures

Notice that `make-counter` makes a different `count` variable each time it is called.

This is because each `lambda` call produces a *closure* — the function along with its referencing environment.

Closures can also be used to do object-oriented programming in Scheme.

```
(define (make-stack)
  (let ((stack '()))
    (lambda (arg)
      (cond [(equal? arg 'pop) ...]
            [else ...push arg...])))))
```