Pure Functional Programming

Two characteristics of functional programming:

- **Referential Transparency**

- **Functions are first class**

Procedures are First-Class

What does it mean for procedures to have *first-class status*?

- They can be given names.
- They can be arguments to procedures.
- They can be returned by procedures.
- They can be stored in data structures (e.g. lists).

Procedures returning procedures

Example: Get the predicate for the type of a sample input

```
(define (test-my-type something)
  (cond [(number? something) number?]
        [(symbol? something) symbol?]
        [(list? something)   list?  ]))
```

Useful when combined with higher-order procedures:

```
(define (like-the-first L)
  (filter (test-my-type (car L)) L))
```

Storing procedures in a list

Maybe we want to apply different functions to the same data:

```
(define (apply-all alof alon)
  (if (null? alof)
      '()
      (cons ((car alof) alon)
            (apply-all (cdr alof) alon))))
```

Then we can get statistics on a list of numbers:

`(apply-all (list length mean stdev) (list 2.4 5 3.2 3 8))`

Interruption: History Class

- The **lambda calculus** is a way of expressing computation
- Developed by Alonzo Church (left) in the 1930s
- Believed to cover everything that is computable (**Church-Turing thesis**)
- *Everything* is a function: numbers, points, booleans, . . .
- Functions are just a kind of data!

Anonymous functions in Scheme

`lambda` is a special form in Scheme that creates a nameless (or "anonymous") function:

```
  (lambda (arg1 arg2 ...)
    expr-using-args)
```

It's a special kind of *function-that-returns-a-function*.

`(lambda (x) (+ x 5))` ⇒ `#<procedure>`

`((lambda (x) (+ x 5)) 8)` ⇒ 13

Behind the curtain

You have already been using `lambda`!

- `(define (f x1 x2 ... xn) exp-using-xs)`
  is the same as:

- `(let ((x1 e1) (x2 e2) ... (xn en)) exp-using-xs)`
  is the same as:

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?

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.

- An `if` with no "else" clause, or a `cond` where all the tests return false, might return *nothing*, or `void`. Functions with side effects like `newline` also return `void`.

- `(begin exp1 exp2 ...)`
  This evaluates all the given expressions, sequentially, and *only returns the value of the last expression*.
  Notice how long it took us to need this!

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.

Save yourself a lot of trouble:
The changing "state" variable (i.e., the `let`) must be
inside the function (i.e., the `define`), but
outside the lambda.

Objects in Scheme

We can use closures and mutation to do OOP in Scheme!

More sophisticated counter:

```
(define (make-counter-obj)
  (let ((count 0))
    (lambda (command)
      (cond [(symbol=? command 'get) count]
            [(symbol=? command 'inc)
             (set! count (+ 1 count))]
            [(symbol=? command 'reset)
             (set! count 0)]))))
```

The object now has three methods: `get`, `inc`, and `reset`.

Built-in Data Structures

Scheme has some useful built-in data structures:

- Arrays (called "vectors").

```
(define A (make-vector 5))
(vector-set! A 3 'something)
(vector-ref A 3) ; produces 'something
(vector-ref A 5) ; error:  out of bounds
```

- Hash tables

```
(define H (make-eqv-hashtable))
(hashtable-set! H 2 'something)
(hashtable-set! H 'another-key 'crazy!)
(hashtable-contains? H 2)                ;true
(hashtable-ref H 'another-key 'default)  ;'crazy!
(hashtable-ref H 1234 'default)          ;'default
```

Inefficiency in Scheme

Recall the problem of computing Fibonacci numbers from lab 1.

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

Why is this function so slow?

Memoization in Scheme

Recall: *Memoization* is remembering the results of previous function calls, and never repeating the same computation.

Why is functional programming *perfect* for memoization?

Scheme's built-in hashes can be used to memoize.

Memoizing Fibonacci

Here's how we might memoize the Fibonacci function:

```
(define fib-memo
  (let ((memo (make-eqv-hashtable)))
    (define (fib-internal n)
      (cond [(<= n 1) n]
            [(hashtable-contains? memo n)
             (hashtable-ref memo n '())]
            [else
             (let ((val (+ (fib-internal (- n 1))
                           (fib-internal (- n 2)))))
               (hashtable-set! memo n val)
               val)]))
    fib-internal))
```

Stack space in recursive calls

Recursive calls can use a lot of memory, even when the results are puny.

```
;; Sum of squares from 1 to n
(define (ssq n)
    (if (= n 0)
        0
        (+ (sqr n) (ssq (- n 1)))))
```

Why does `(ssq 4000000)` run out of memory?

Stack space in recursive calls

This function does the same thing, but takes an *extra argument* that serves as an accumulator.

```
;; Sum of squares using tail recursion
(define (ssq-better n accum)
  (if (= n 0)
      accum
      (ssq-better (- n 1)
                  (+ (sqr n) accum))))
```

Now `(ssq-better 4000000 0)` actually works!

Tail recursion

The second version worked because there was no need to make a stack of recursive calls.

A function is *tail recursive* if its output expression in every recursive case is only the recursive call.

In Scheme, this means the recursive call is **outermost**
in the returned expression.

`ssq-better` is better because it is tail recursive!

Tail recursion for Fibonacci

To implement tail recursion we usually make a helper function:

```
(define (fib-helper n i fib-of-i fib-of-i+1)
  (if (= i n)
      fib-of-i
      (fib-helper
       n
       (+ i 1)
       fib-of-i+1
       (+ fib-of-i
          fib-of-i+1))))
```

The main function then becomes:

```
(define (fib-tail n) (fib-helper n 0 0 1))
```