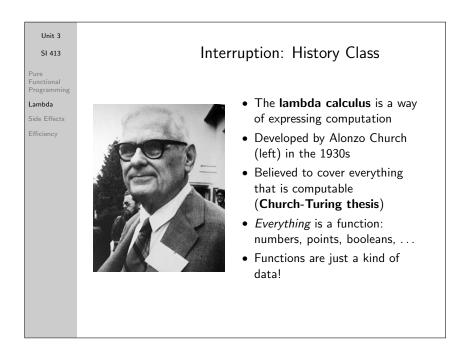


Unit 3 SI 413 Pure Functional Programming Lambda Side Effects Efficiency Efficiency Unit 3 Procedures returning procedures Example: Get the predicate for the type of a sample input (define (test-my-type something) (cond [(number? something) number?] [(symbol? something) number?] [(list? something) list? ])) 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:

Then we can get statistics on a list of numbers: (apply-all (list length mean stdev) (list 2.4 5 3.2 3 8))



Unit 3 SI 413 Anonymous functions in Scheme Pure Functional Programming Lambda Side Effects Efficiency I ambda 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.

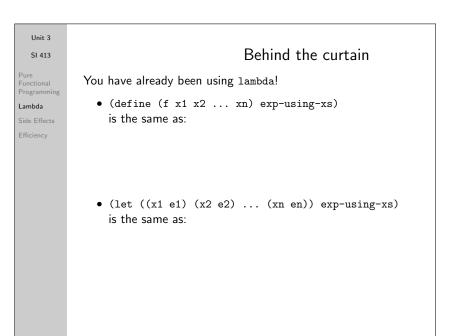
(lambda (x) (+ x 5))  $\Rightarrow$  #<procedure>

((lambda (x) (+ x 5)) 8)  $\Rightarrow$  13

Unit 3 SI 413

Pure Functional Programming

Side Effects



Unit 3	
SI 413	Side Effects
re nctional ogramming	Remember the intro to the Scheme standard:
mbda le Effects iciency	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, <b>imperative</b> , and message passing styles, find convenient expression in Scheme. What do we have to give up to get side effects?

Pur Fun Pro Lan Side

> Unit SI 41

Pure Function Program

Side Effe

t 3	
13	Controlling Output
nal nming	
ects	Displaying text to the screen is a kind of side effect.
ey.	Here are some useful functions for screen output:
	• (display X)
	• (newline)

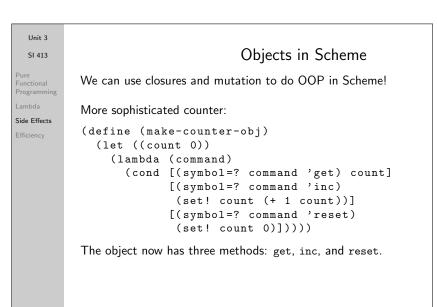
• (printf format args...) The catch-all format flag is ~a.

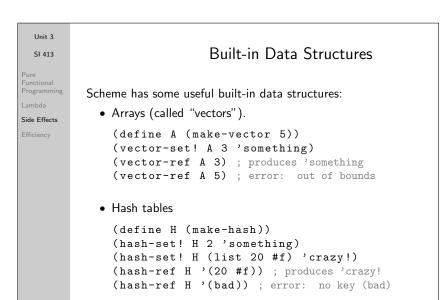
(Note: Strings in Scheme are made using double quotes, like "This\_is\_a\_string".)

Unit 3	
SI 413	Structuring code with side-effects
Pure Functional Programming	
Lambda Side Effects Efficiency	With side effects, we have to violate the one-expression-per-function rule.
	• (void) is a Scheme function that returns <i>nothing</i> . Functions like newline return this type.
	<ul> <li>(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!</li> </ul>

Unit 3 SI 413 Pure Functional	Mutation!
Programming Lambda Side Effects Efficiency	The built-in special form (set!\ x val) changes the value of x to be val.
	<pre>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))))</pre>

Unit 3 SI 413	Closures
Pure Functional Programming	
Lambda	
Side Effects	Notice that make-counter makes a different count variable
Efficiency	each time it is called.
	This is because each lambda call produces a <i>closure</i> — 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.





Unit 3 SI 413 Inefficiency in Scheme Pure Functional Programming Lambda Side Effects Efficiency 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?

Unit 3	
SI 413	Memoization in Scheme
Pure Functional Programming	
Lambda	
Side Effects	
Efficiency	Recall: <i>Memoization</i> is remembering the results of previous function calls, and never repeating the same computation.
	Why is functional programming <i>perfect</i> for memoization?
	Scheme's built-in hashes can be used to memoize.

Unit 3 SI 413	Memoizing Fibonacci
Pure Functional Programming	
Lambda	Here's how we might memoize the Fibonacci function:
Side Effects	(define fib-hash (make-hash))
Efficiency	
	<pre>(define (fib-memo n) (cond [(not (hash-has-key? fib-hash n)) (hash-set! fib-hash n (if (&lt;= n 1)</pre>

Unit 3
Si 413
Si 413
Ctarter Stack space in recursive calls
Furgramming
Lambda
Side Effects
Efficiency
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?

Unit 3	
SI 413	Stack space in recursive calls
Pure Functional Programming	
Lambda	
Side Effects	This function does the same thing, but takes an
Efficiency	extra argument that serves as an accumulator.
	<pre>;; Sum of squares using tail recursion (define (ssq-better n accum) (if (= n 0)</pre>

Unit 3	
SI 413	Tail recursion
Pure Functional Programming	
Lambda	
Side Effects	The second version worked because there was no need to make
Efficiency	a stack of recursive calls.
	A function is <i>tail recursive</i> if its output expression in every recursive case is only the recursive call.
	In Scheme, this means the recursive call is <b>outermost</b> in the returned expression.
	ssq-better is better because it is tail recursive!

Unit 3         SI 413         Tail recursion for Fibonacci	i
Pure Functional Programming Lambda Side Effects Efficiency To implement tail recursion we usually make a helper (define (fib-helper n i fib-of-i fib-of (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	-i+1)