Pure Functional Programming			
Pure Functional Programming			
Two characteristics of functional prog	ramming:		
Referential Transparency			
• Functions are first class			
SI 413 (USNA) Unit	3	Fall 2021	1 / 25
Pure Functional Programming			
Procedures are First-Class			
What does it mean for procedures to have <i>first-class status</i> ?			
They can be given names.			
• They can be arguments to procee	lures.		
 They can be returned by procedu 	res		
• They can be retained by procedu			
. They are be should in data stored	uuraa (a. a. liata)		
• They can be stored in data struct	tures (e.g. lists).		
SI 413 (USNA) Unit	3	Fall 2021	2 / 25

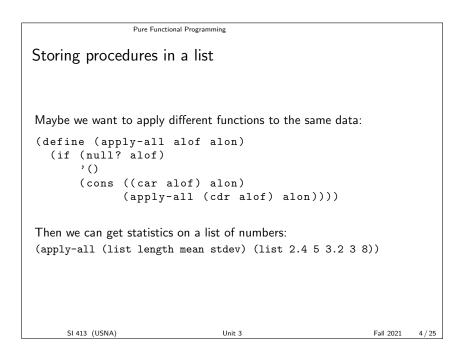
Pure Functional Programming

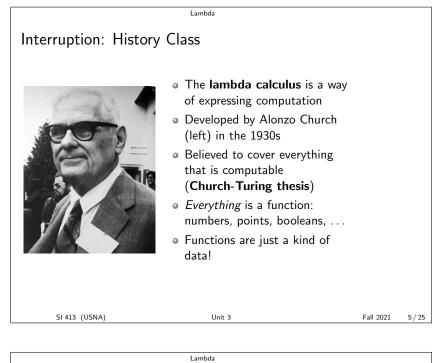
SI 413 (USNA)

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))

Unit 3

Fall 2021 3 / 25





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)) \Rightarrow #<procedure> ((lambda (x) (+ x 5)) 8) \Rightarrow 13 Unit 3 Fall 2021

6 / 25

SI 413 (USNA)

Lambda		
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:		
SI 413 (USNA) Unit 3	Fall 2021	8/25

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?

SI 413 (USNA)

Unit 3

Fall 2021 9 / 25

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 t	to violate the one-expr	ession-per-function ru	ıle.			
 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!						
SI 413 (USNA)	Unit 3	Fall 2021	11/25			
	Side Effects					
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:						
<pre>(define (make-counte (let ((count 0)) (lambda () (set! count (+</pre>						
(display count (newline)))))					
SI 413 (USNA)	Unit 3	Fall 2021	13 / 25			
Closures	Side Effects					
Notice that make-counter makes a different count variable 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

Side Effects **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. Fall 2021 15 / 25 SI 413 (USNA) Unit 3 Side Effects

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
```

Unit 3

SI 413 (USNA)

Efficiency

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?

Unit 3

Fall 2021 17 / 25

Efficiency 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. St 413 (USNA) Unit 3 Efficiency Memoizing Fibonacci Here's how we might memoize the Fibonacci function: (define fib-memo (let ((memo (make-eqv-hashtable))))

Efficiency

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?

Unit 3

SI 413 (USNA)

Fall 2021

21 / 25

	5 <i>4</i> . •			
	Efficiency			
Stack space in recurs	ive calls			
This function does the sam serves as an accumulator.	ne thing, but takes an <i>extr</i>	a argument that		
;; Sum of squares using (define (ssq-better (if (= n 0) accum				
(ssq-better (- n 1) + (sqr n) accum))))			
Now (ssq-better 4000000	0) actually works!			
		E # 0001		
SI 413 (USNA)	Unit 3	Fall 2021 2	22 / 25	
	Efficiency			
Tail recursion				
The second version worked recursive calls.	because there was no nee	ed to make a stack c	of	
A function is <i>tail recursive</i> only the recursive call.	if its output expression in	every recursive case	is	
In Scheme, this means the in the returned expression.	recursive call is outermos	st		
ssq-better is better becau	use it is tail recursive!			
SI 413 (USNA)	Unit 3	Fall 2021 2	23 / 25	
	Efficiency			
Tail recursion for Fibonacci				
rall recursion for FID	UIIACCI			

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))
SI 413 (USNA) Unit 3 Fall 2021 24/25
```