

CS 311O

Functions

Prof. Clarkson

Fall 2019

Today's music: Function by E-40 (Clean remix)

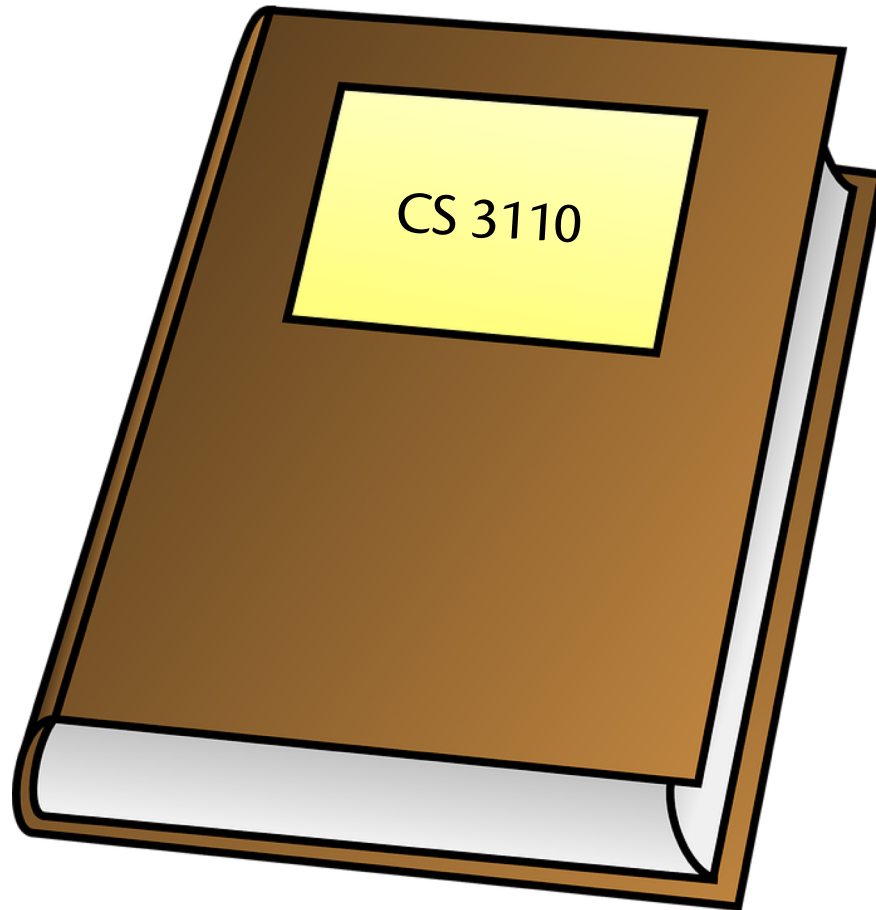
CLICKER QUESTION 1

Search

- Website
- Textbook
- Campuswire



Textbook vs. lecture



PRIMARY



summary

Review

Previously in 3110:

- **Syntax and semantics**
- **Expressions: if, let**
- **Definitions: let**

Today:

- **Functions**

ANONYMOUS FUNCTION EXPRESSIONS & FUNCTION APPLICATION EXPRESSIONS

Anonymous function expression

Syntax: **fun** **x1** ... **xn** \rightarrow **e**

fun is a keyword



Evaluation:

- A function is a value: no further computation to do
- In particular, body **e** is not evaluated until function is applied



Lambda

- Anonymous functions a.k.a. *lambda expressions*
- Math notation: $\lambda x . e$
- The lambda means “what follows is an anonymous function”



Lambda

- [Python](#)
- [Java 8](#)
- A popular [PL blog](#)
- [Lambda style](#)

Functions are values

Can use them anywhere we use values:

- Functions can **take** functions as arguments
- Functions can **return** functions as results

This is an incredibly powerful language feature!

Function application

Syntax: **$e_0 \ e_1 \ \dots \ e_n$**

No parentheses required!

(unless you need to force particular order of evaluation)

Function application

What is the evaluation rule for

$e_0 \ e_1 \ \dots \ e_n \quad ?$

Challenge: invent it right now!

Function application

Evaluation of $e_0 \ e_1 \ \dots \ e_n$:

1. Evaluate subexpressions:

$e_0 \implies v_0, \dots, e_n \implies v_n$

Note that v_0 is guaranteed to be a function:

fun $x_1 \ \dots \ x_n \rightarrow e$

2. Substitute v_i for x_i in e yielding new expression e' . Evaluate it: $e' \implies v$
3. Result is v

Example

Let vs. function

These two expressions are syntactically different but **semantically equivalent**:

```
let x = 2 in x+1
```

```
(fun x -> x+1) 2
```

FUNCTION DEFINITIONS

Two syntaxes to define functions

These definitions are syntactically different but semantically equivalent:

```
let inc = fun x -> x+1
```

```
let inc x = x + 1
```

- First is fundamentally no different from **let** definitions we saw last lecture
- Second is syntactic sugar: not necessary, makes language “sweeter”

Recursive function definition

Must explicitly state that function is recursive:

```
let rec f ...
```

FUNCTIONS AND TYPES

Function types

Type $\mathbf{t} \rightarrow \mathbf{u}$ is the type of a function that takes input of type \mathbf{t} and returns output of type \mathbf{u}

Type $\mathbf{t1} \rightarrow \mathbf{t2} \rightarrow \mathbf{u}$ is the type of a function that takes input of type $\mathbf{t1}$ and another input of type $\mathbf{t2}$ and returns output of type \mathbf{u}

etc.

Note dual purpose for \rightarrow syntax:

- Function types
- Function values

Function application

Type checking:

If $e_0 : t_1 \rightarrow \dots \rightarrow t_n \rightarrow u$

And $e_1 : t_1,$

$\dots,$

$e_n : t_n$

Then $e_0 e_1 \dots e_n : u$

Anonymous function expression

Type checking:

If $x_1:t_1, \dots, x_n:t_n$

And $e:u$

Then $(\text{fun } x_1 \dots x_n \rightarrow e) :$
 $t_1 \rightarrow \dots \rightarrow t_n \rightarrow u$

PARTIAL APPLICATION

More syntactic sugar

Multi-argument functions do not exist

fun x y -> e

is syntactic sugar for

fun x -> (fun y -> e)

More syntactic sugar

Multi-argument functions do not exist

```
let add x y = x + y
```

is syntactic sugar for

```
let add = fun x ->  
            fun y ->  
                x + y
```


CLICKER QUESTION 2

More syntactic sugar

Multi-argument functions do not exist

fun x y z -> e

is syntactic sugar for

fun x -> (fun y -> (fun z -> e))

Again: **Functions are values**

Can use them anywhere we use values:

- Functions can **take** functions as arguments
- Functions can **return** functions as results

This is an incredibly powerful language feature!

Upcoming events

- [today] A0 released, R1 released
- [Monday] R1 due
- [Wednesday] A0 due

This is fun!

THIS IS 3110