WELLESLEY

# Programming Languages

**CS 251**

*Fall 2021*

*Carolyn Anderson*

# Interpreters

# Interpreters

An **interpreter** for a programming language is a procedure that, when applied to an expression of the language, performs the actions required to evaluate that expression.

# Metacircularity

A meta-circular interpreter is an interpreter that is written in the same language that is being interpreted.

# Substitution-based Model of Evaluation

**The Substitution Model of Variable Binding**:
When a value $v$ is bound to an expression $e$, substitute the value $v$ for every **unbound** occurrence of $e$ in the scope of the binder.

# Environments

```
(define x 10)
(define (foo x y)
    (let ((x (+ x 1)))
    (match y
      ((list 5 x) (print x))
      ((list 10) (print x)))))
(foo x (list x))
```

global environment

function environment

let environment

case environment

# Environments

An environment is a sequence of frames, each of which is a table of bindings that associate variables with values.

# Environments

Instead of a substitution model of evaluation, we'll move to an environment model of evaluation.

Our environment model of evaluation will have two parts: a rule for evaluating expressions, and a rule for evaluating functions.

# Evaluating Compound Expressions

## Rule for evaluating expressions:

To evaluate a compound expression (other than a special form), evaluate the subexpressions and then apply the value of the operator subexpression to the value of the argument subexpressions.

(This is call-by-value evaluation!)

# Evaluating Functions

## Rule for evaluating functions:

To apply a compound function to a set of arguments, evaluate the body of the function in a new environment. To construct this environment, extend the environment part of the function object by a frame in which the formal parameters of the function are bound to the actual arguments to which the function is applied.

# Interpreter Project Overview

## Week 1

- Variables and environments
- Evaluation

## Week 2

- Primitive functions
- Special forms

## Week 3

- Lambda and local binding

## Week 4

- Meta-circularity

# Interpreter Project Overview

- We'll be programming in class most days.
- Each week's component is due on Monday at 10pm.
- I will release my solutions so that you can use them for the next part.
- You will be required to write tests for your code. Your test cases will be graded.

# Writing Test Cases

Guidelines for the interpreter project

- Keep your tests in interpreter_tests.rkt.
- Import the test library using (require rackunit).
- You should write 2-3 tests per function, and more for functions that have particularly tricky edge cases.

# Practice: write test cases for lst-alphabetized?

```
(define (lst-alphabetized? lst)
  (cond ((< (length lst) 2) #t)
        ((string<=? (first lst)
                    (first (rest lst)))
             (lst-alphabetized? (rest lst)))
        (else #f)))
```

# Importing modules

In order to run your tests, you will need to import your definitions from your interpreter file. This requires two things.

First, make your definitions available to be imported using (provide (all-defined-out)) at the top of your interpreter file.

Second, import the definitions using (require “interpreter.rkt") at the top of your test suite file.

# Eval

Eval takes an expression and an environment as arguments. Based on what kind of expression it is, eval directs its evaluation.

```
> (eval '(+ 1 2))
3
```

# Eval

We will define our own version of eval in the interpreter project, but Racket also has a built-in eval function, which takes a quoted expression and evaluates it.

This only works in the REPL, not in the definitions window. This is because it needs an environment, which the REPL makes available, but the definitions window does not.

# Apply

Apply takes a function and a list of arguments to apply the function to. Primitive functions are applied directly, while compound procedures have their subexpressions evaluated in a new environment.

# Apply

We will define our own version of apply in the interpreter project, but Racket also has a built-in apply function.

```
> (apply + (3 1 2))
6
```