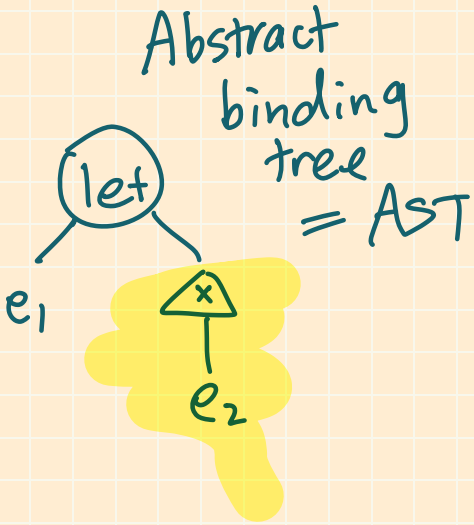
Announcements

- HW2 due Thursday
- Quiz 1 next Tuesday in class (closed-book)

HW 2 Walkthrough

- Lamp language manual
- Starter code
- CSIL reference interpreter

let $x = e_1$ in e_2 $\approx\!\!\Rightarrow$



My Original Plan

Date	Topic
Week 1	**How to design a programming language?**
06/24	Why study programming languages? + Python
06/25	Syntax
06/26	Inference rules
Week 2	**What makes a programming language?**
07/01	Semantics
07/02	Names
07/03	Types
Week 3	**How to abstract *data*?**
07/08	Finite and recursive types
07/09	Pattern-matching
07/10	*Quiz 1 (tentative)*
Week 4	**How to abstract *computation*?**
07/15	Lambda calculus
07/16	Polymorphism, type inference
07/17	Defunctionalization, continuation-passing
Week 5	**How to change the world?**
07/22	Mutable states
07/23	Effect handlers
07/24	*Quiz 2 (tentative)*
Week 6	**What is the future of programming like?**
07/29	Advanced topic, TBD
07/30	Advanced topic, TBD
07/31	Advanced topic, TBD
08/02	**(End of summer session A)**

Reality

Topic
How to design a programming language?
Intro, Syntax I
Syntax II, Inference Rules
Operational Semantics
Section: Python Tutorial
What makes a programming language?
Operational Semantics Practicum
Variables
Lambda Calculus
No class (Independence Day)
How to abstract *data*?
Types
Finite and recursive types
Pattern-matching
How to abstract *computation*?
Quiz 1 (scheduled)
Polymorphism, type inference
Defunctionalization, continuation-passing
How to change the world?
Mutable states
Effect handlers
Quiz 2 (tentative)
What is the future of programming like?
Advanced topic, TBD
Advanced topic, TBD
Advanced topic, TBD
(End of summer session A)

Winter 2025

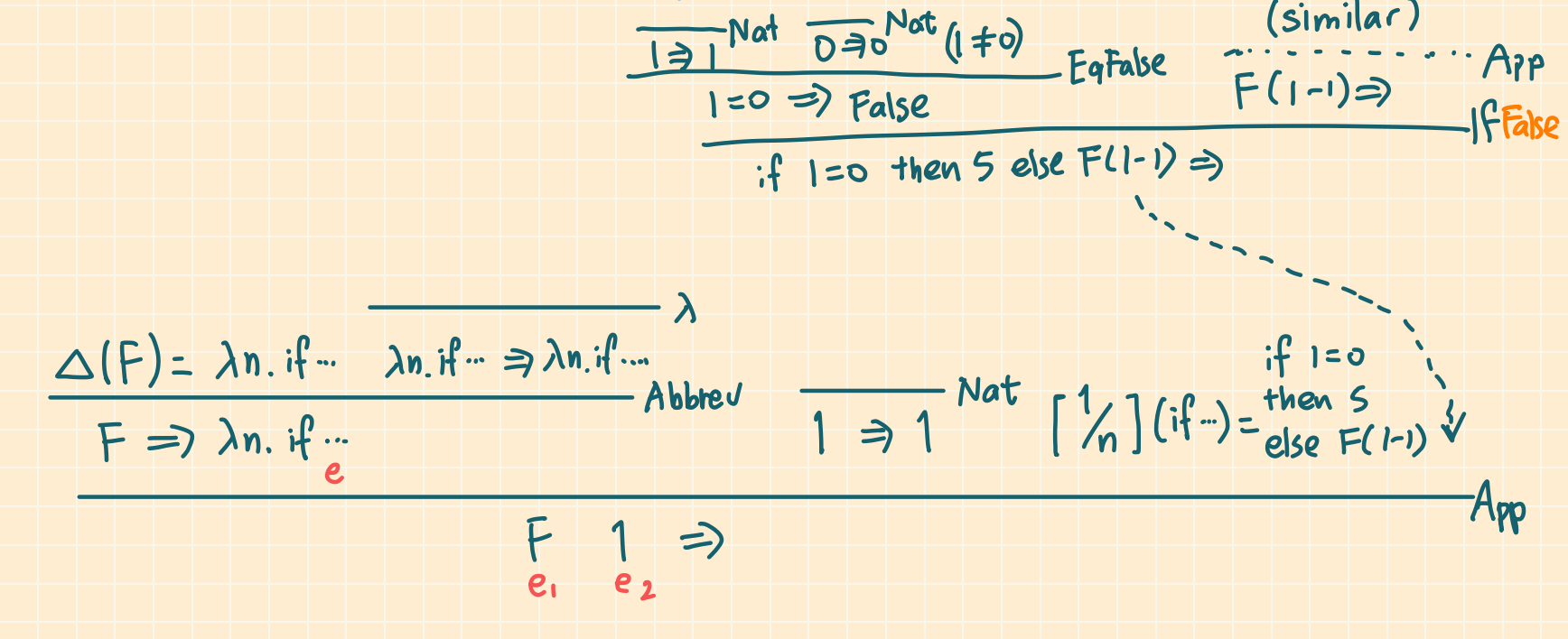
Date	Topic
1/6	Hello, World!
1/8	OCaml crash course I
1/13	OCaml crash course II
1/15	OCaml crash course III
1/20 (MLK)	No class
1/22	Lambda Calculus I
1/27	Lambda Calculus II
1/29	(Cancelled)
2/3	λ^+
2/5	Operational Semantics I
2/10	Operational Semantics II
2/12	Type Checking
2/17 (President)	No class
2/19	Type Checking (continued)
2/24	Type Inference
2/26	Type Inference (continued)
3/3	Polymorphism
3/5	Polymorphism (continued)
3/10	Curry Howard isomorphism

Exercise:

```
def F = \n. if n = 0 then 5 else F (n-1)
main = F 1
```

Draw the derivation tree for the evaluation of the main expression.
(Basically simulate how the interpreter would run the program)

$\Delta = F \mapsto \lambda n.\ \text{if } n=0 \text{ then } 5 \text{ else } F(n-1)$



Problem 4 (20 points)

Consider the following (untyped) λ^+ expression:

```
(fix r is lambda n.
    if n = 0 then 5 else n + r (n - 1)) 1
```

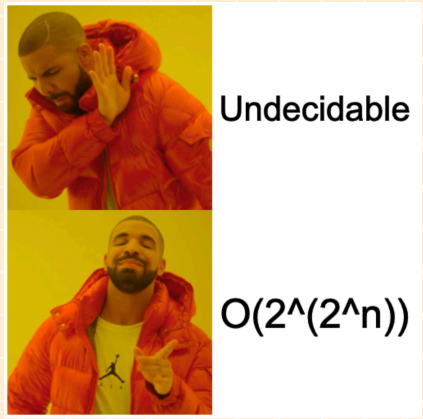
Some stats

- This was a problem from CS 162 final exam (winter' 24 & winter 25)!
- <30% students fully solved this problem
- Y'all are doing fantastic

Tomorrow, we'll talk about "abstraction" and "static analysis", the most challenging concepts in CS 162

Today, I want to prepare you as much as possible

- evaluation order (leftover from last time, covered by 1st quiz)
- lazy substitution (aka environments / closures)
- why almost every PL problem is undecidable.
- tomorrow: how to defeat undecidability.



Recall the let-rule

$$\frac{e1 \Rightarrow v1 \quad ([v1/x]\ e2 = e2') \quad e2' \Rightarrow v2}{\texttt{let } x = e1 \texttt{ in } e2 \Rightarrow v2}\ \text{Let}$$

Q: why do we need to eval e1 before substituting x?

$$\frac{e1 \Rightarrow v1 \quad ([v1/x]\ e2 = e2') \qquad e2 \Rightarrow v2}{\texttt{let x = e1 in e2} \Rightarrow v2}\ \text{Let}$$

$$\frac{([e1/x]\ e2 = e2') \qquad e2 \Rightarrow v2}{\texttt{let x = e1 in e2} \Rightarrow v2}\ \text{Let*}$$

Q: Suppose you're given two interpreters to play with,
one of which implements Let, the other implements Let*.
Can you tell which interpreter implements which rule?

Example: let x = 1+2 in x * x using Let

$$\frac{e1 \Rightarrow v1 \quad ([v1/x]\ e2 = e2') \quad e2 \Rightarrow v2}{\texttt{let x = e1 in e2} \Rightarrow v2}\ \text{Let}$$

Example: let x = 1+2 in x * x using Let*

$$\frac{([e1/x]\ e2 = e2') \qquad e2 \Rightarrow v2}{\texttt{let x = e1 in e2} \Rightarrow v2}\ \text{Let*}$$

How do the derivation trees differ?

Example: let x = 1+2 in 3 using Let

$$\frac{e1 \Rightarrow v1 \quad ([v1/x]\ e2 = e2') \quad e2 \Rightarrow v2}{\texttt{let x = e1 in e2} \Rightarrow v2}\ \text{Let}$$

Example: let x = 1+2 in 3 using Let*

$$\frac{([e1/x]\ e2 = e2') \quad e2 \Rightarrow v2}{\texttt{let x = e1 in e2} \Rightarrow v2}\ \text{Let*}$$

How do the derivation trees differ?

Observation:

- In Let, a variable is substituted with _____ (expr/value) "Call-by-value"
- In Let*, a variable is substituted with _____ (expr/value) "Call-by-name"

Call-by-value semantics = variables are substituted with values
Call-by-name semantics = variables are substituted with (unevaluated) programs

- If a variable is used >1 time, evaluation using _____ (CBV/CBN) requires fewer steps
- If a variable is used <1 time, evaluation using _____ (CBV/CBN) requires fewer steps
- What if a variable is used exactly 1 once?

Example: let x = 1 + True in x using CBV stuck

Example: let x = 1 + True in x using CBN stuck

Example: let x = 1 + True in 1 using CBV stuck

Example: let x = 1 + True in 1 using CBN ✓

Observation:

- CBV/CBN? If a variable is substituted with something with side effect, you'll always observe a bad behavior
- CBV/CBN? If a variable is substituted with side effect, you'll only observe a bad behavior if the variable is used

Every language feature that has variables requires a choice between CBV or CBN.

$$\frac{e_1 \Rightarrow \lambda x.e \quad e_2 \Rightarrow v_2 \quad [v_2/x]e = e' \quad e' \Rightarrow v'}{e_1 \; e_2 \Rightarrow v'} \text{App}$$

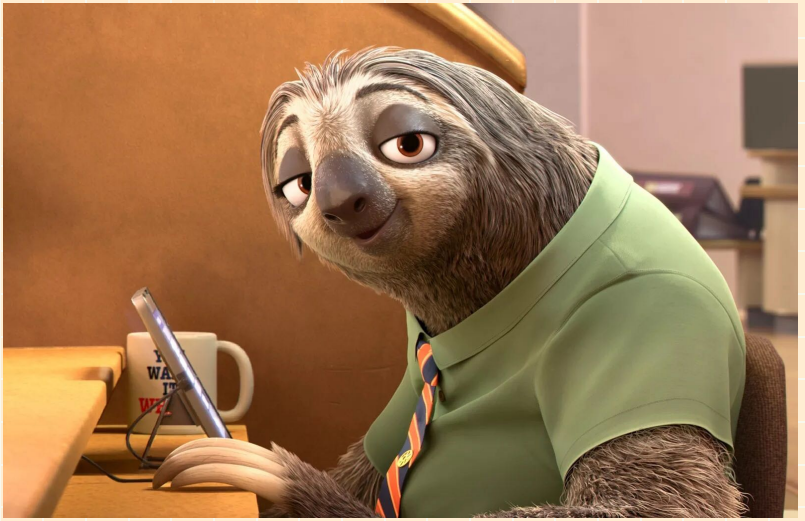
Q1: Is this rule CBV or CBN?

Usually, the entire language makes a single commitment: all of the features involving variables follow either CBV or CBN.

- CBV languages: basically all lang
- CBN languages: the OG lambda calculus (1930s)
- Neither CBV nor CBN languages: Haskell

Lazy Substitution

The art of getting faster
by being lazier



Eager substitution semantics (no matter CBV/CBN)

- conceptually simple
- extremely slow

$$\frac{e1 \Rightarrow v1 \quad ([v1/x]\ e2 = e2') \quad e2 \Rightarrow v2}{\text{let } x = e1 \text{ in } e2 \Rightarrow v2}\ \text{Let}$$

$$\frac{([e1/x]\ e2 = e2') \quad e2 \Rightarrow v2}{\text{let } x = e1 \text{ in } e2 \Rightarrow v2}\ \text{Let*}$$

A typical Python program:

```python
def main():
    teacherFirstName = userInput()
    print(teacherFirstName)

# create a function to
# get user input, return it to the calling function
def userInput():
    firstName = input('What is your first name? ')
    lastName = input('What is your last name? ')
    return firstName, lastName

# Create a function that prints names
def printNames(name1,name2):
    print('in print names')
```

```
let x = v1 in
let y = f(x) in
let z = g(x, y) in
…
```

Substitution is O(size of AST)

The above program requires O(n^2) substitutions

Ideally, we only need O(n)

Eager substitution semantics

$e \Rightarrow v$

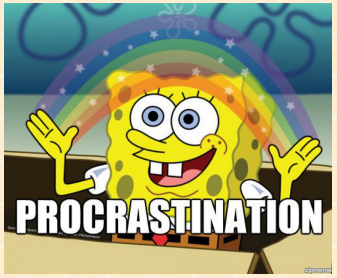
Lazy substitution semantics

$\gamma \vdash e \Rightarrow v$ where

$\gamma = \cdot$

$\quad | \ \gamma, x : v$ if CBV

(OR $\gamma, x : e$ if CBN)



γ remembers the sequence of substitutions $[v/x]$ that should have been performed, but delays them until it can no longer delay

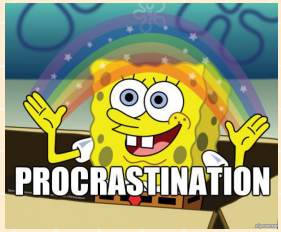
Eager substitution semantics

$$\frac{e_1 \Rightarrow v_1 \quad ([v_1/x]e_2 = e_2') \quad e_2 \Rightarrow v_2}{\texttt{let } x = e_1 \texttt{ in } e_2 \Rightarrow v_2}$$

(No rule for free variables)

Lazy substitution semantics

$$\frac{\gamma \vdash e_1 \Rightarrow v_1 \quad \gamma, x : v_1 \vdash e_2 \Rightarrow v_2}{\gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : v_2}$$



$$\frac{\gamma(x) = v}{\gamma \vdash x \Rightarrow v}$$