Review

Design space of semantics for variables

	Substitution	Eager	Lazy
Evaluation order			
Call by value (CBV)		$\dfrac{e_1 \Rightarrow v_1 \quad \left([v_1/x]e_2 \Rightarrow e_2'\right) \quad e_2' \Rightarrow v_2}{\text{let } x = e_1 \text{ in } e_2 \Rightarrow v_2}$ (expensive)	$\dfrac{\gamma \vdash e_1 \Rightarrow v_1 \quad \gamma, x : v_1 \vdash e_2 \Rightarrow v_2}{\gamma \vdash \text{let } x = e_1 \text{ in } e_2 \Rightarrow v_2}$
Call by name (CBN)		$\dfrac{\left([e_1/x]e_2 \Rightarrow e_2'\right) \quad e_2' \Rightarrow v_2}{\text{let } x = e_1 \text{ in } e_2 \Rightarrow v_2}$	$\dfrac{\gamma(x) = v}{\gamma \vdash x \Rightarrow v}$

Lazy substitution

- Found in most language implementation (aka environments)
- More efficient: O(n) to eval programs with n variables
- Foreshadows what we'll see later today
- Extremely easy to get it wrong

Examples

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

Derive let x = 1 in x + 1

$$\frac{\dfrac{}{\cdot \vdash 1 \Rightarrow 1}\ \text{Nat} \qquad \dfrac{\dfrac{(\cdot, x{:}1)(x) = 1}{\cdot, x{:}1 \vdash x \Rightarrow 1}\ \text{Var} \qquad \dfrac{}{\cdot, x{:}1 \vdash 1 \Rightarrow 1}\ \text{Nat}}{\cdot, x{:}1 \vdash x + 1 \Rightarrow 2}\ \text{Arith}}{\cdot \vdash \text{let } x = 1 \text{ in } x + 1 \Rightarrow 2}\ \text{Let}$$

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

Derivate let x = 1 in let x = 2 in x

why?

$$\frac{\dfrac{}{\cdot \vdash 1 \Rightarrow 1}\ \text{Nat} \qquad \dfrac{\dfrac{}{x{:}1 \vdash 2 \Rightarrow 2}\ \text{Nat} \qquad \dfrac{(x{:}1, x{:}2)(x) = 2}{x{:}1, x{:}2 \vdash x \Rightarrow 2}\ \text{Var}}{x{:}1 \vdash \text{let } x = 2 \text{ in } x \Rightarrow 2}\ \text{Let}}{\cdot \vdash \text{let } x = 1 \text{ in let } x = 2 \text{ in } x \Rightarrow 2}\ \text{Let}$$

Formalizing variable lookups

$$\frac{}{(\gamma, x{:}v)(x) = v}$$

$$\frac{(x \neq y) \qquad \gamma(x) = v}{(\gamma, y{:}v)(x) = v}$$

$$\boxed{\gamma(x) = v}$$

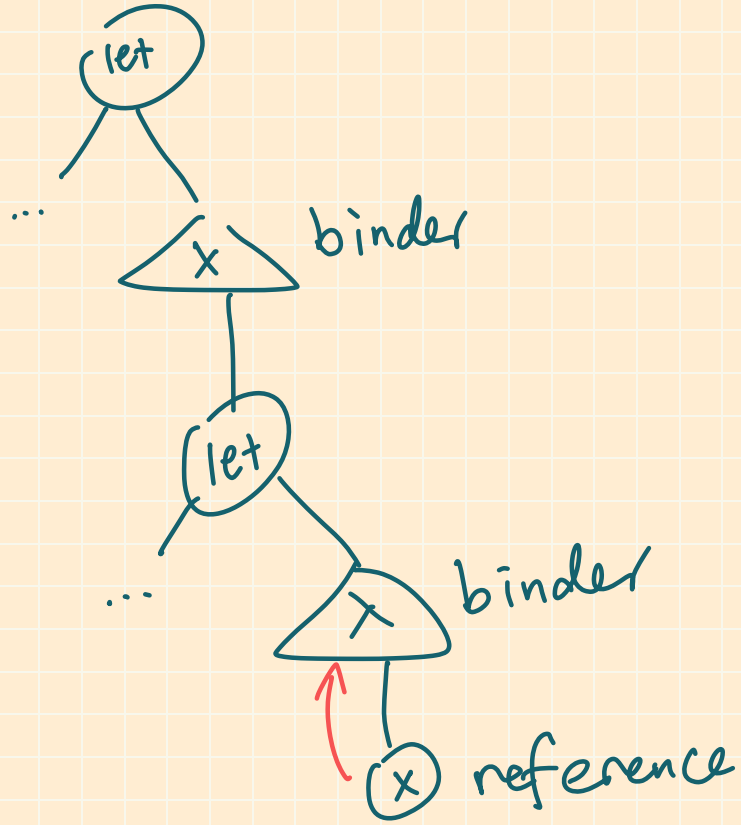
$$\frac{}{(x{:}v, \gamma)(x) = v}$$

$$\frac{x \neq y \qquad \gamma(x) = v}{(y{:}v, \gamma)(x) = v}$$

$$\gamma ::= \cdot \mid \gamma, x{:}v \quad (x \mapsto v)$$

Exercise: try both alternatives, and see which one works

Why should variable insertions & lookups follow FIFO (stack discipline)?



Practical implementation of association lists

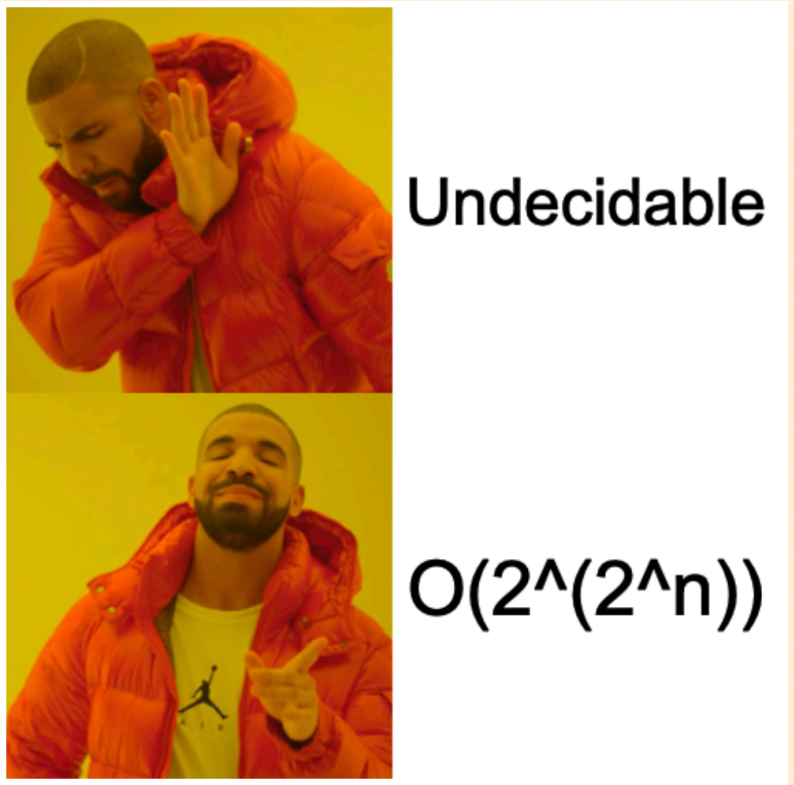
Any map that's persistent / immutable. Example:

- Using list in Python

- Using dictionary (hashmap) in Python

- Using functional binary search trees (see HW3 problem)

Why problems in PL are HARD



"Hey, just a heads up, the aviation software in this aircraft was vibe-coded by me 🙂."

We want to predict (& fix) bugs before our code is deployed.

Rice's Theorem: Correctly predicting any non-trivial property about program semantics is undecidable.

Corollary 1: no automated algorithm (including LLMs) can correctly predict and fix bugs.

I'll teach you how to get around Rice's Theorem

AI writes shit code

Code needs to be updated

You're hired to fix their shit

Bugs lead to disasters

Company gets sued

Rice's Theorem: Correctly predicting any non-trivial property about program semantics is undecidable.

def check(prog: str) -> bool

where check(p) returns True iff p satisfies some semantic property.

Example: check(p) = True iff prog computes the nth-Fibonacci number

Example: check(p) = True iff prog

Example: check(p) = True iff prog never raises an exception

Example: check(p) = True iff prog never dereferences null (None in Python)

Rice's Theorem: Correctly predicting any non-trivial property about program semantics is undecidable.

In other words, we claim the following Python function does not exist:

```
def check(prog: str) -> bool
```

where check(p) returns True iff p satisfies some semantic property.

Plan of attack:

- We know the Halting Problem is undecidable. I.e., the following program does not exist:

```
def halts(prog: str, inp: str) -> bool
```

- Assume for a contradiction that check did exist. We'll use it to implement halts.
- Since halts can't exist, check can't either.

What we have at our disposal

- `def check(prog: str) → bool: ...`
- at least 1 prog passes the check. Call it ref (for reference implementation). So `check(ref) = True`
- at least 1 prog fails the check

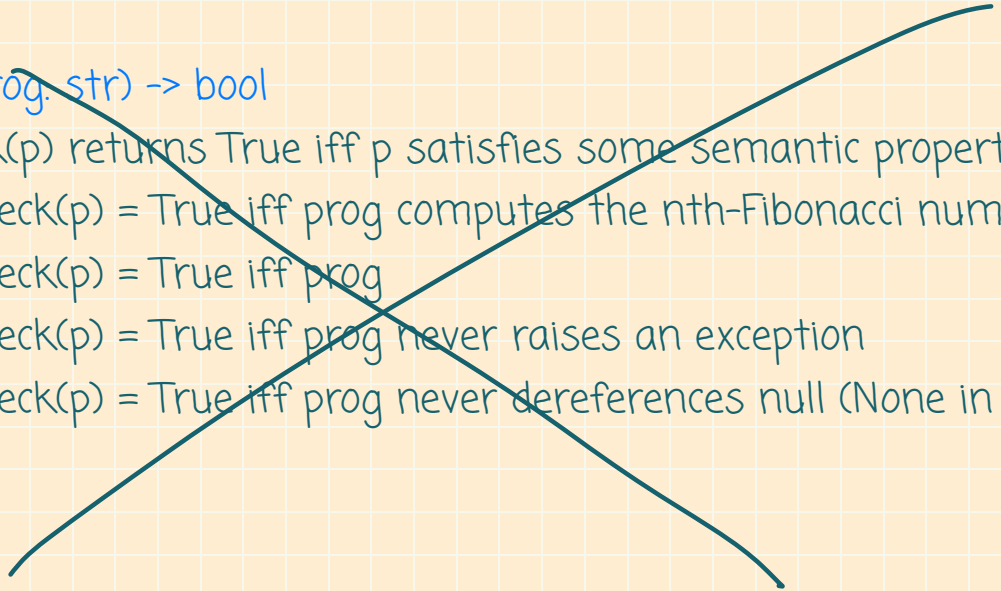
What we want:

```
def halts(prog: str, x: str)
  → bool:

  def almost_ref(i: str) → Any:
    eval(prog)(x)
    return ref(i)

  return check(almost_ref)
```

How `almost_ref` works:

- if prog halts on x, then **`almost_ref`** agrees with **`ref`** on all inputs.
- if prog doesn't halt on x, then **`almost_ref`** doesn't halt on any input at all.

Rice's Theorem: Correctly predicting any non-trivial property about program semantics is undecidable.



def check(prog: str) -> bool

where check(p) returns True iff p satisfies some semantic property.

Example: check(p) = True iff prog computes the nth-Fibonacci number

Example: check(p) = True iff prog

Example: check(p) = True iff prog never raises an exception

Example: check(p) = True iff prog never dereferences null (None in Python)

DO NOT EXIST!

Rice's Theorem: Correctly predicting any non-trivial property about program semantics is undecidable.

2 ways out

Sacrifice automation: rely on human cleverness to reason about program behaviors mentally.

- This is basically what we've been doing when we draw derivation trees
- 1 step further: use math to prove individual programs can't go wrong (week 5/6)

now until week 5

Sacrifice correctness*:

- accept occasional "incorrect" answers
- be very careful about how we define "correctness" so we still get some level of guarantees

How to sacrifice correctness while being "correct" in some sense?

Dynamic behavior:
Is a program free of any bad behavior when executed?

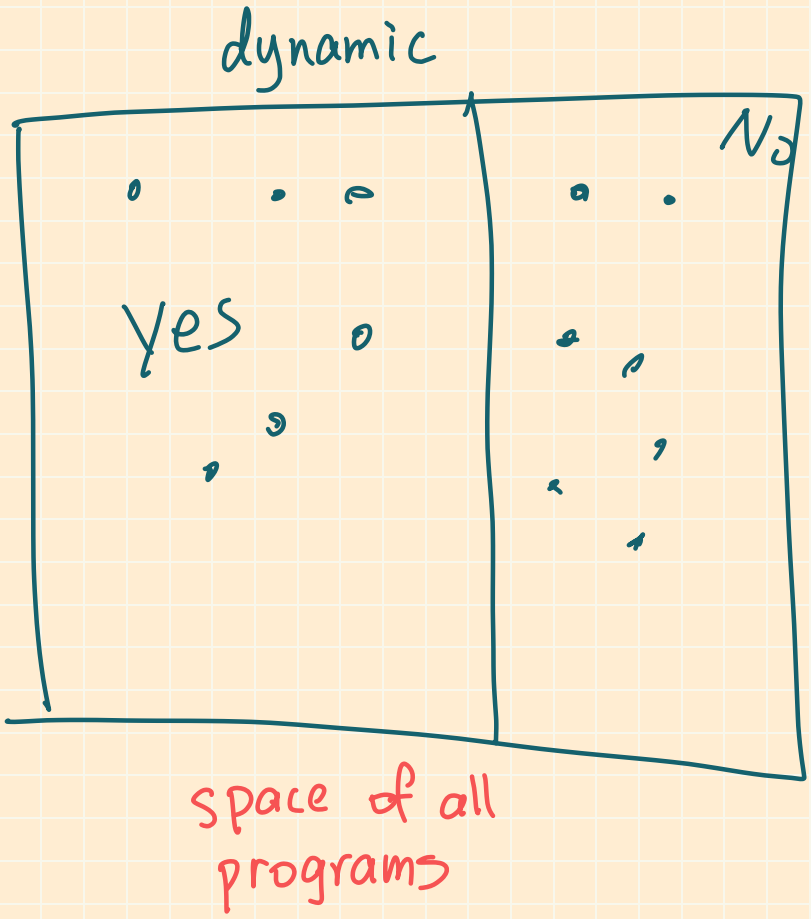
Yes

No

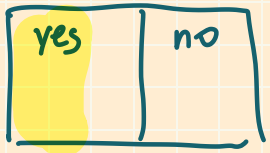
Static analysis:
Will it be free of any bad behavior if I only "look at its AST" before running it?

Yes

No

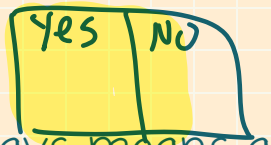


Categorizing static analyses



A static analysis is sound if a predicted "yes" always means an actual "yes".

- "predict free of bad behaviors" => "actually free of bad behaviors"
- In other words, you can always trust positive predictions.
- Sound but incomplete = false alarms: some programs are actually fine, but we predict they're not fine.



A static analysis is complete if a predicted "no" always means an actual "no".

- "predict this program is bad" => "actually bad"
- Complete but unsound = fake guarantees: if I say this program is fine, you can't trust me.

Examples of sound (but incomplete) analysis

Type system
(today + tomorrow)

Example: C++ compiler

Q: "Does a program call functions that are defined"

Note

Examples of complete (but unsound) analysis

every behavior you run into during testing is an actual behavior.

Testing

But you can never have enough tests to be sound

Sound!

incomplete:

```
if (true) { }
else { foo(); }
```

← actually fine, but C++ rejects this!

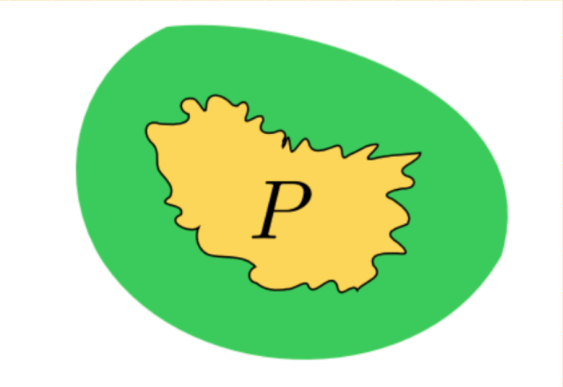
Soundness and completeness are flipped if you negate the question.

- In CS 162, our question is always: is a program free of bad things?
- If you negate the question, sound => complete, complete => sound
- In some research community, the question is negated:

does a program exhibit bad behaviors?

How to design sound analysis

ABSTRACTION !!!



Yellow:

- actual behavior of program P
- unknownable by any algorithm

Green:

- abstraction/over-approximation
- computable by design