Review: What is a static analysis?
- program (meta language) Ocaml
- A tool/algorithim that it
takes as input a program P
  (object language)
  JavaScript
  and derives facts about P
  without running P

This Week:
- Read syllabus
- Install Ocaml
  and get familiar with it
- Introductions
  Next Week
- Evan is traveling
- Arlen leads guest discussions

Interpreter: Program \times Data \rightarrow Data
  (Test Case)

Verify: Program \rightarrow Boolean
Verify: Program \times Specification \rightarrow Boolean

Optimize: Program \rightarrow Program

Semantics = meaning

def f(x):
    return x + 1
Programming Language

Syntax = Strings/Terms = "What you can write in the language"
Semantics = Meaning of a program in the language

Concrete Syntax = Strings = what you the developer in the PL write
- keywords, separators
- important in practice
- well understood principles - lexers/parsers

Abstract Syntax = terms/trees in the language
we'll ignore parsing issues

Higher-Order Abstract Syntax

ASTs with binding + scope explicit variables

\[(\text{let } x=1 \text{ in } x+1) + x\]

\[\text{III} x\]

\[(\text{let } y=1 \text{ in } y+1) + x\]

"abstract binding trees"

\[
\text{type } \text{expr} = \begin{cases} 
1 \text{ Let of string } \times \text{ expr } \times \text{ expr} \\
1 \text{ Let of } \text{ expr } \rightarrow \text{ expr} 
\end{cases}
\]
\( (\text{let } y = 1 \text{ in } 1 + y) + x \)

- **\( y \)** is a **bound variable**
- **\( x \)** is a **free variable**

---

**Syntactic Entities**

- **Set**
  - **Description**: Meta-Variable

- **\( \mathbb{Z} \)**
  - Integers
  - **\( n \)**

- **\( \mathbb{B} \)**
  - Booleans
  - **\( b \)**
  - **\( B = \{ \text{true, false} \} \)**

- **Var**
  - Program variables
  - **\( x, y \)** [Italic font]

- **Expr**
  - Program expressions
  - **\( c \)**
e ::= x
    | n
    | b
    | vop e₁
    | e₁ bop e₂
    | if (e₁) e₂ e₃

type e = Var of strng
    | Num of int

Now what?
- What is the "meaning" of a given expression
- I want to write an interpreter so that run my programs
- How do we evaluate expressions?
Operational Semantics

specifies how expressions should be evaluated.

Some expressions we will consider "final results" — values

\[ \text{Val} \ v \ ::= \ n \mid \text{b} \quad \text{Val} = \mathbb{Z} \cup \text{IB} \]

operational semantics is

a concrete interpreter

\[ e \Downarrow v \quad (\text{define this relation}) \]

\[ \text{eval}(e, v) \quad e: \text{Expr} \quad v: \text{Val} \]

Define this relation inductively
Judgment is statement holding or not
Judgment form is a "template" for statements

\[ e \lor \neg e \quad \text{judgment form} \]

1 + 1 \neq 2 \quad \text{judgment (we want to hold)}

1 + 1 \neq 42 \quad \text{"provable"}

\[ \mathcal{L} \text{ "ab"} \lor 6 \quad \text{not provable} \]

1 + 1 \neq 7 \quad \text{not e}

Set of inference rules defines a judgment form
gives meaning to the judgment form
Eval Int

\[ n \downarrow n \]

axioms

Eval Plus

\[ e_1 \downarrow n_1 \quad e_2 \downarrow n_2 \]

\[ e_1 + e_2 \downarrow n_1 + n_2 \]

\[ \uparrow \quad \text{syntactic plus} \quad \uparrow \quad \text{"mathy" plus} \]

Top Down Reading

Premises \[ J_1 \ldots J_n \]

\[ \frac{\text{conclusion}}{J} \]

"show judgement holds"

Bottom-Up View: Interpretation View / Algorithm View

Derivation = Execution of interpreter

\[ \frac{\text{Eval Num}}{1 \downarrow 1} \quad \frac{\text{Eval Num}}{1 \downarrow 1} \]

\[ \frac{\text{Eval Plus}}{(1+1) \downarrow 2} \]

\[ \frac{\text{Eval Num}}{(1+1)+1 \downarrow 3} \]

\[ \text{Eval Plus} \]

\[ \text{Eval Num} \]
\[
\frac{e_1 \downarrow n_1 \quad e_2 \downarrow n_2}{e_1 + e_2 \downarrow n_1 + n_2}
\]

\[
\frac{e_1 \downarrow n_1}{e_1 \downarrow n_1 + 0}
\]

\underline{not \ syntax-directed}