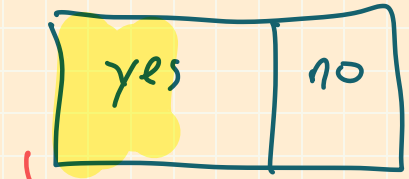
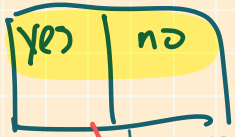
Match each item on the left with at least 1 description on the right.



Sound

Complete

Unsound



Incomplete

Actual no always -> predicted no

Actual yes always -> predicted yes

Predicted yes always -> actual yes

Predicted no always -> actual no

Sometimes, predict yes but actually no

Sometimes, predict no but actually yes

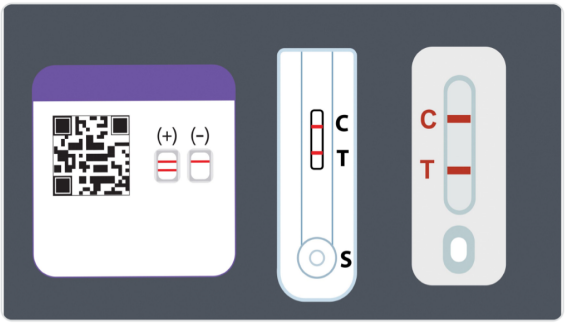
Actual question: do I have Covid?

Prediction: does the antigen test show 2 lines?

2 lines = positive = yes

1 line = negative = no

Antigen Tests



Antigen Test results reflecting a positive result.

Antigen tests* are rapid tests that usually produce results in 15-30 minutes. Positive results are accurate and reliable. However, in general, antigen tests are less likely to detect the virus than NAAT tests, especially when symptoms are not present. Therefore, a single negative antigen test cannot rule out infection.

sound

incomplete

Is this prediction method sound / complete / unsound / incomplete?

Actual question: do I have Covid?

Prediction: do I have a fever (is my body temp above 100F = 38C)?

Assumption: every Covid case is symptoptic
= if you have Covid, then you'll get a fever

fever yes ⇒ actual no

Is this prediction method sound / complete / unsound / incomplete?

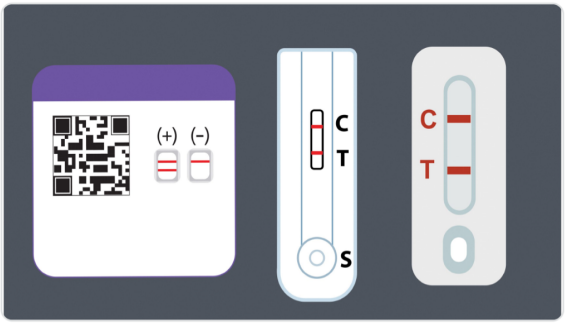
Actual question: am I free of Covid = bad?

Prediction: does the antigen test show 1 line?

1 line = yes = free of Covid

2 lines = no = have covid

Antigen Tests



Antigen Test results reflecting a positive result.

Antigen tests* are rapid tests that usually produce results in 15-30 minutes. Positive results are accurate and reliable. However, in general, antigen tests are less likely to detect the virus than NAAT tests, especially when symptoms are not present. Therefore, a single negative antigen test cannot rule out infection.

Is this prediction method sound / complete / unsound / incomplete?

Static analysis = automated/algorithmic prediction of actual runtime behaviors of programs

Can we have static analysis that's both sound and complete?

Static analysis:

Actual: is the program free of bad behaviors when I run it?

Prediction: does it look like free of bad behaviors?

Sound static analysis:

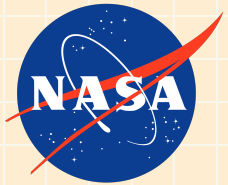
If the analysis says the aviation software is bugless, then your plane won't crash when you fly in it.

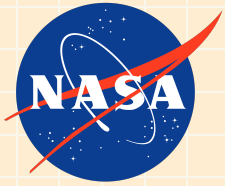
Complete static analysis

If the analysis found a bug, then the plane will definitely run into a bug when flown.

Would you prefer: (no wrong answer, always a tradeoff)

- sound but incomplete analysis
- complete but unsound analysis





Sound (but incomplete) analysis

- Sound: able to prove a program is safe once and for all
- Incomplete: false alarms require human intervention
- If all false alarms are resolved, the system is 100% safe

Complete (but unsound) analysis

- Complete: every bug is a bug caught
- Unsound: can't prove the absence of bugs
- Finds as many actual bugs as possible, but no cigar

Type systems (aka type checkers) are the canonical sound analysis

- Implemented by every compiler that generates machine code (C++, C, Java, Rust, OCaml, ...)
- Question predicted by type systems: does the program have undefined behavior = getting stuck during evaluation?

Almost every type checker aims to be sound. Why?

- If a program has undefined behavior, the compiler can't even generate the code.
- Thus, the compiler is free to generate arbitrary code.
- Your program may end up doing something CRAZY (lauching nukes).

Who should I blame for my programs that go CRAZY?

Type system sound => compiler guarantees the program is free of undefined behaviors.

- If you see crazy behaviors, blame the compiler!

Type system unsound => the programmer is responsible for remembering all possible undefined behaviors and avoid them.

- If you see crazy behaviors, yourself is to blame!

C compilers are unsound (C99)

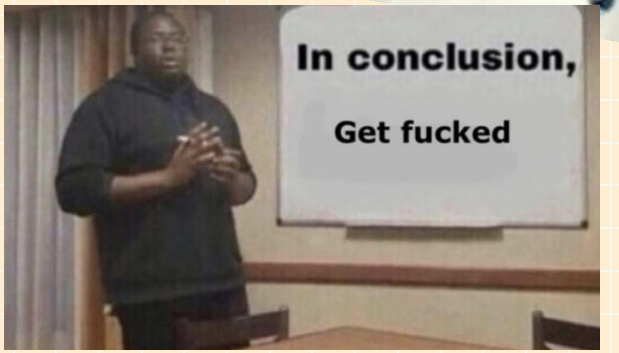
As a programmer, you're responsible for memorizing every one of the 199 undefined behaviors, and being careful to avoid them.

J.2 Undefined behavior

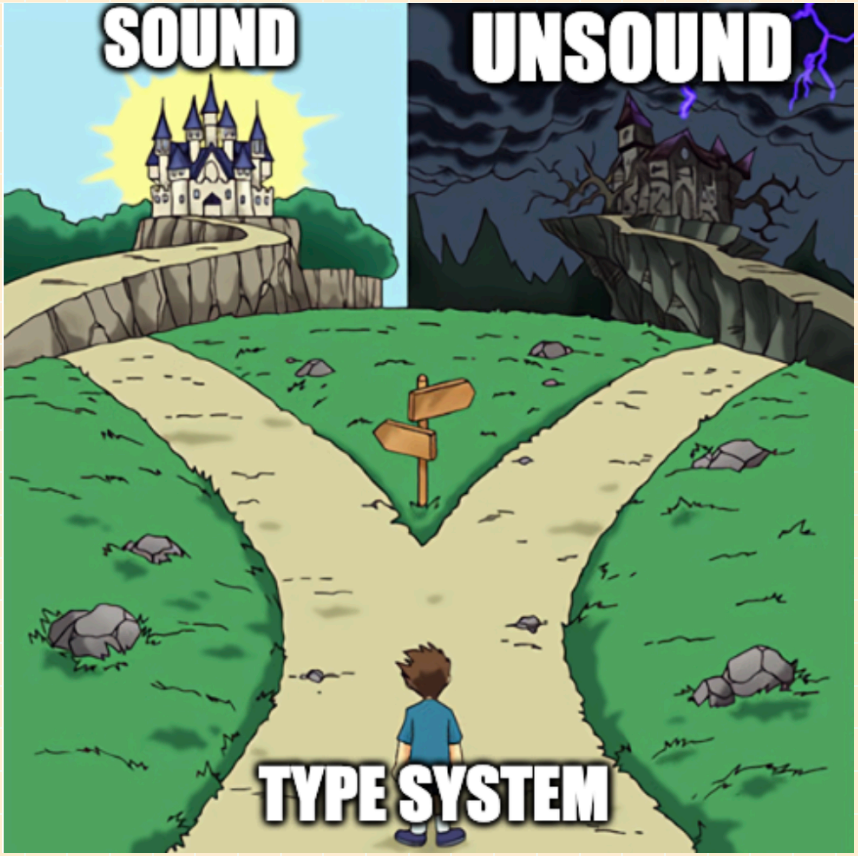
1 The behavior is undefined in the following circumstances:

- A "shall" or "shall not" requirement that appears outside of a constraint is violated (clause 4).
- A nonempty source file does not end in a new-line character which is not immediately preceded by a backslash character or ends in a partial preprocessing token or comment (5.1.1.2).
- Token concatenation produces a character sequence matching the syntax of a universal character name (5.1.1.2).
- A program in a hosted environment does not define a function named main using one of the specified forms (5.1.2.2.1).
- A character not in the basic source character set is encountered in a source file, except in an identifier, a character constant, a string literal, a header name, a comment, or a preprocessing token that is never converted to a token (5.2.1).
- An identifier, comment, string literal, character constant, or header name contains an invalid multibyte character or does not begin and end in the initial shift state (5.2.1.2).
- The same identifier has both internal and external linkage in the same translation unit (6.2.2).
- An object is referred to outside of its lifetime (6.2.4).
- The value of a pointer to an object whose lifetime has ended is used (6.2.4).
- The value of an object with automatic storage duration is used while it is indeterminate (6.2.4, 6.7.8, 6.8).
- A trap representation is read by an lvalue expression that does not have character type (6.2.6.1).
- A trap representation is produced by a side effect that modifies any part of the object using an lvalue expression that does not have character type (6.2.6.1).
- The arguments to certain operators are such that could produce a negative zero result, but the implementation does not support negative zeros (6.2.6.2).
- Two declarations of the same object or function specify types that are not compatible (6.2.7).
- Conversion to or from an integer type produces a value outside the range that can be represented (6.3.1.4).
- Demotion of one real floating type to another produces a value outside the range that can be represented (6.3.1.5).
- An lvalue does not designate an object when evaluated (6.3.2.1).
- A non-array lvalue with an incomplete type is used in a context that requires the value of the designated object (6.3.2.1).
- An lvalue having array type is converted to a pointer to the initial element of the array, and the array object has register storage class (6.3.2.1).
- An attempt is made to use the value of a void expression, or an implicit or explicit conversion (except to void) is applied to a void expression (6.3.2.2).
- Conversion of a pointer to an integer type produces a value outside the range that can be represented (6.3.2.3).
- Conversion between two pointer types produces a result that is incorrectly aligned (6.3.2.3).
- A pointer is used to call a function whose type is not compatible with the pointed-to type (6.3.2.3).
- An unmatched ' or " character is encountered on a logical source line during tokenization (6.4).
- A reserved keyword token is used in translation phase 7 or 8 for some purpose other than as a keyword (6.4.1).
- A universal character name in an identifier does not designate a character whose encoding falls into one of the specified ranges (6.4.2.1).
- The initial character of an identifier is a universal character name designating a digit (6.4.2.1).
- Two identifiers differ only in nonsignificant characters (6.4.2.1).
- The identifier __func__ is explicitly declared (6.4.2.2).
- The program attempts to modify a string literal (6.4.5).
- The characters ', \, ", //, or /* occur in the sequence between the < and > delimiters, or the characters ', \, //, or /* occur in the sequence between the " delimiters, in a header name preprocessing token (6.4.7).
- Between two sequence points, an object is modified more than once, or is modified and the prior value is read other than to determine the value to be stored (6.5).
- An exceptional condition occurs during the evaluation of an expression (6.5).
- An object has its stored value accessed other than by an lvalue of an allowable type (6.5).
- An attempt is made to modify the result of a function call, a conditional operator, an assignment operator, or a comma operator, or to access it after the next sequence point (6.5.2.2, 6.5.15, 6.5.16, 6.5.17).
- For a call to a function without a function prototype in scope, the number of arguments does not equal the number of parameters (6.5.2.2).
- For call to a function without a function prototype in scope where the function is defined with a function prototype, either the prototype ends with an ellipsis or the types of the arguments after promotion are not compatible with the types of the parameters (6.5.2.2).
- For a call to a function without a function prototype in scope where the function is not defined with a function prototype, the types of the arguments after promotion are not compatible with those of the parameters after promotion (with certain exceptions) (6.5.2.2).

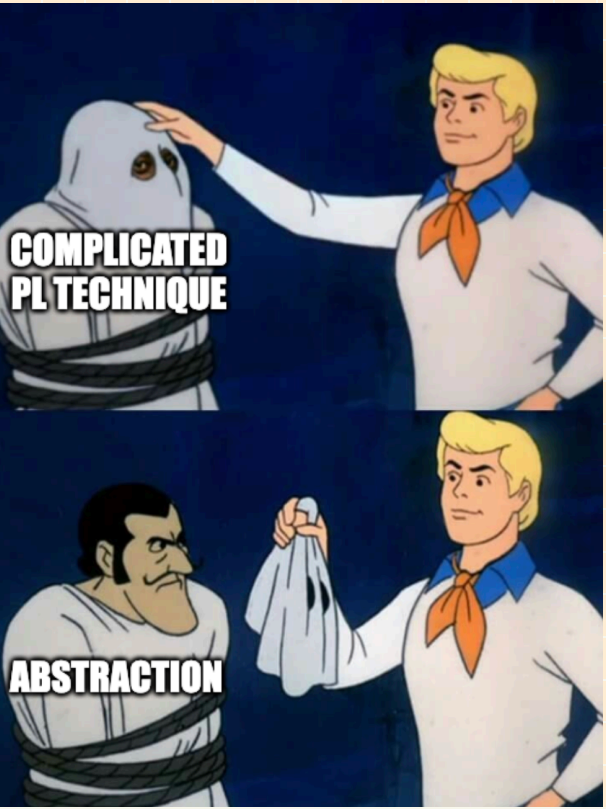
Programming in a language with unsound type system be like

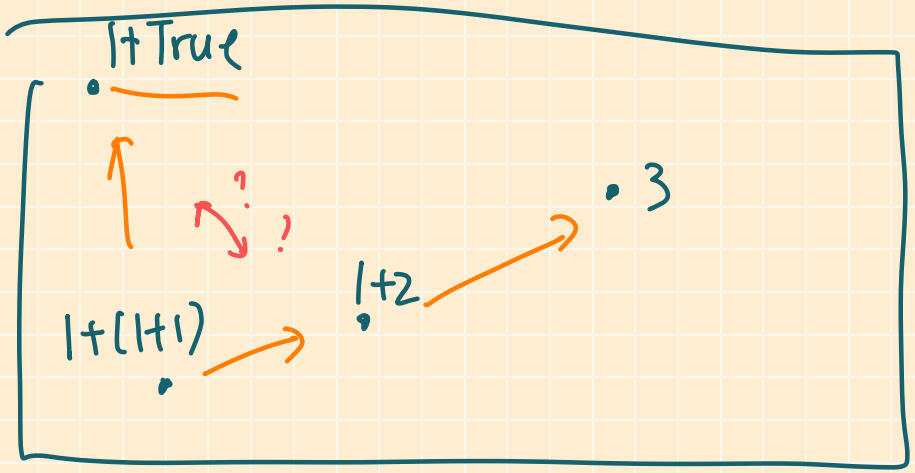


We want sound type systems. But How?



ABSTRACTION is the only trick for sound analysis known to mankind

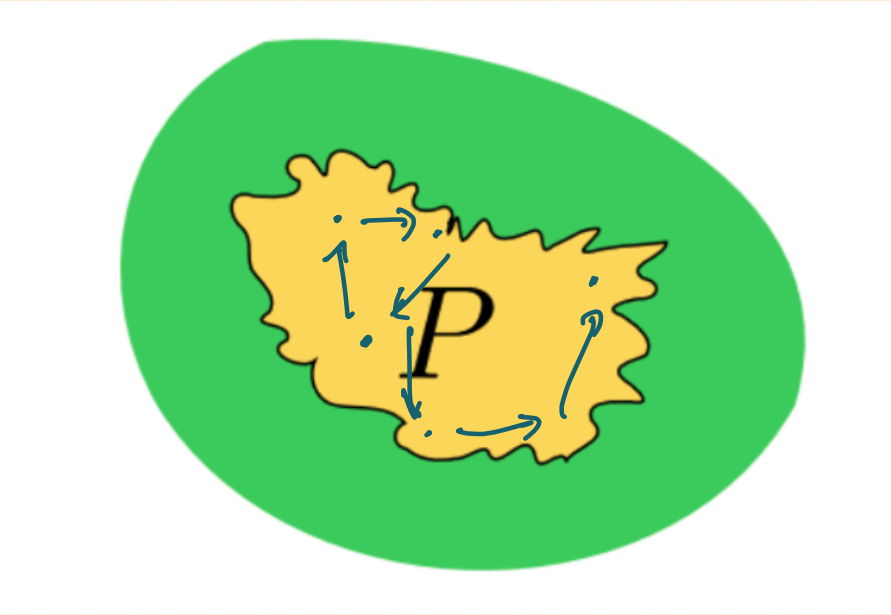




- A program traces through a trajectory as it's being run
- Question: is it gonna step into a bad program?
- Actual trajectory is unknownable in advance.

Solution: look at programs in an approximate/fuzzy way.

Solution: look at the "over-approximated" trajectory through foggy windows



Yellow:

- actual trajectory space of program P
- unknownable by any algorithm

Green:

- abstraction/over-approximation
- computable by design

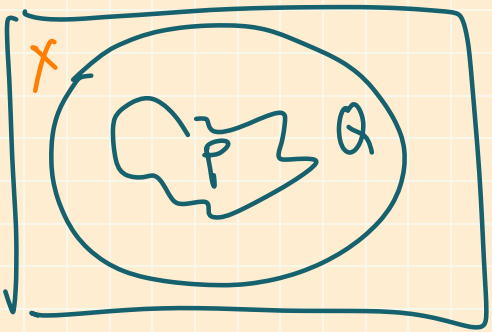
You've already seen this idea in CS 162

P

```
if collatz(n):
   print("happy")
else:
  print("sad")
```

Q

```
if (*):
  print("happy")
else:
  print("sad")
```



If the abstracted program doesn't print S, then neiver can the original program.

Why? Because Q prints more stuff than P. So if Q doesn't print something neither can P.

Let's abstraction the semantics (runtime behavior) of lamp

```
expr ::= <nat> | <bool>
       | e1 + e2 | ..
       | e2 == e2
       | let x = e1 in e2
       | if e1 then e2 else e3
<nat> in { 0, 1, 2, 3, ... }
<bool> in { True, False }
```

ops semantics

```
value ::= <nat>
        | <bool>
```

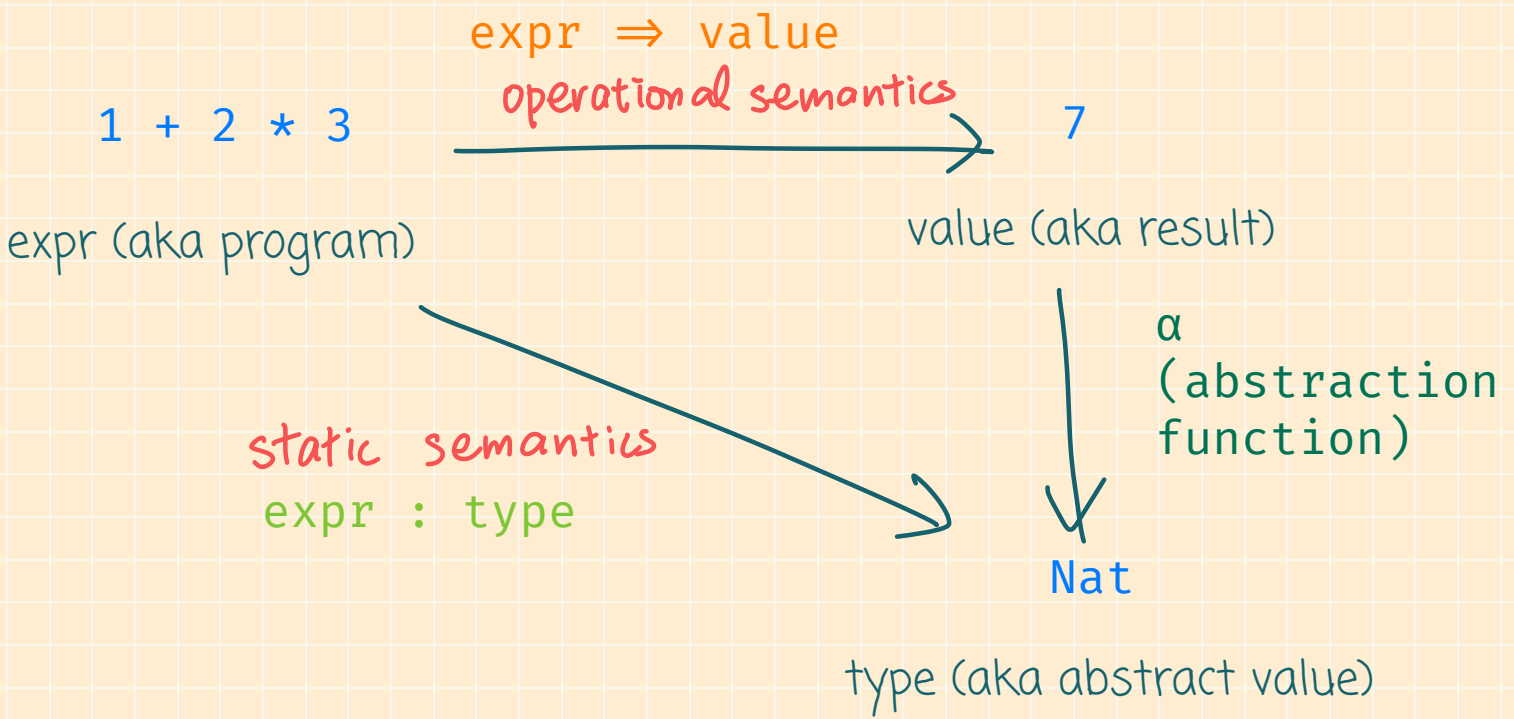
0, 1, 2, 3

True, False

static semantics

α abstraction function

```
type ::= Nat
       | Bool
```



Type soundness: "well-typed programs don't go wrong"

If an expression doesn't get stuck during type checking, then neither can it get stuck during runtime.

(We'll prove this next week!)

How to design a type system

Step 1: define abstraction function 👓:

```
α(n) = Nat, for n in {0, 1, 2, .. }
α(b) = Bool, for b in {True, False}
```

Step 2:

- go through each operational (dynamic) semantics rule
- look at it through the abstraction function 👓 to obtain static semantics

Step 3: next week

Prove "well-typed programs can't go wrong"

Operational semantics
(dynamic/runtime)

$e \Rightarrow v$

$$\frac{}{n \Rightarrow n}\ \textsc{Nat}$$

$$\frac{}{b \Rightarrow b}\ \textsc{Bool}$$

$$\frac{e_1 \Rightarrow n_1 \quad e_2 \Rightarrow n_2 \quad (\oplus \in \{+,-,*\}) \quad (n_1 \oplus n_2 = n_3)}{e_1 \oplus e_2 \Rightarrow n_3}\ \textsc{Arith}$$

$$\frac{e_1 \Rightarrow n_1 \quad e_2 \Rightarrow n_2 \quad (n_1 = n_2)}{e_1 == e_2 \Rightarrow \text{True}}\ \textsc{EqTrue}$$

$$\frac{e_1 \Rightarrow n_1 \quad e_2 \Rightarrow n_2 \quad (n_1 \neq n_2)}{e_1 == e_2 \Rightarrow \text{False}}\ \textsc{EqFalse}$$

if true then 1 else False
if False then 1 else True

$$\frac{e_1 \Rightarrow \text{True} \quad e_2 \Rightarrow v}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow v}\ \textsc{IfTrue}$$

$$\frac{e_1 \Rightarrow \text{False} \quad e_3 \Rightarrow v}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow v}\ \textsc{IfFalse}$$

Abstract semantics
(static/compile-time)

$e : t$

$$\frac{}{n : \text{Nat}}\ \text{TNat}$$

$$\frac{}{b : \text{Bool}}\ \text{TBool}$$

$$\frac{e_1 : \text{Nat} \quad e_2 : \text{Nat} \quad \oplus \in \{+,-,*\}}{e_1 \oplus e_2 : \text{Nat}}\ \text{TArith}$$

$$\frac{e_1 : \text{Nat} \quad e_2 : \text{Nat}}{e_1 = e_2 : \text{Bool}}\ \text{TEQ}$$

$$\frac{e_1 : \text{Bool} \quad e_2 : t_2 \quad e_3 : t_3 \quad (t_2 = t_3)}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t_2}\ \text{TIf}$$

Operational semantics
(dynamic/runtime)

Abstract semantics
(static/compile-time)

(No rule for variables)

$$\frac{e_1 \Rightarrow v_1 \qquad ([v_1/x]e_2 = e_2') \qquad e_2' \Rightarrow v_2}{\text{let } x = e_1 \text{ in } e_2 \Rightarrow v_2} \textsc{Let}$$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t} \text{TVar}$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma, x{:}t_1 \vdash e_2 : t_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : t_2} \text{TLet}$$

$$\frac{\dfrac{}{\cdot \vdash 1 : \text{Nat}}\text{TNat} \qquad \dfrac{\dfrac{}{x:\text{Nat} \vdash 1 : \text{Nat}}\text{TNat} \qquad \dfrac{(x:\text{Nat})(x) = \text{Nat}}{x:\text{Nat} \vdash x : \text{Nat}}}{X:\text{Nat} \vdash 1 + x : \text{Nat}}\text{TARITH}}{\cdot \vdash \text{let } x = 1 \text{ in } 1 + x :}$$

$\Gamma \vdash e : t$

Type environment (Γ) given by CFG:

```
Γ ::= (empty)
    | Γ, x: t
```

Exercise: let's add pairs to our language

```
expr ::= ... | (e1, e2)
       | let (x, y) = e1 in e2
```

pack

unpack

Example:

```
let (x, y) = (1+2, 3*4) in
x - y
```

```
value ::= ... | (v1, v2)
```

Operational Semantics $e \Rightarrow v$

$$\frac{e1 \Rightarrow v1 \qquad e2 \Rightarrow v2}{(e1, e2) \Rightarrow (v1, v2)}\ \text{Pack}$$

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

Exercise: let's design a type system with pairs

```
value ::= <nat>
        | <bool>
        | (v1, v2)
```

```
type ::= Nat
       | Bool
       | (t1, t2)
```

Step 1: define abstraction function 👓:

```
α(n) = Nat, for n in {0, 1, 2, .. }
α(b) = Bool, for b in {True, False}
α((v1,v2)) = (α(v1), α(v2))
```

Example: α(1, True) = (Nat, Bool)

Step 2:

- go through each operational (dynamic) semantics rule
- look at it through the abstraction function 👓 to obtain static semantics

$$\frac{e1 \Rightarrow v1 \qquad e2 \Rightarrow v2}{(e1,\ e2) \Rightarrow (v1,\ v2)}$$ Pack

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : (t_1, t_2)}$$

$$\frac{e1 \Rightarrow (v1,\ v2) \qquad [v1/x,\ v2/y]e2 = e2' \qquad e2' \Rightarrow v2}{\texttt{let}\ (x,\ y) = e1\ \texttt{in}\ e2 \Rightarrow v2}$$ Unpack

$$\frac{\Gamma \vdash e_1 : (t_1, t_2) \qquad \Gamma, x:t_1, y:t_2 \vdash e_2 : t_3}{\Gamma \vdash \text{let } (x,y) = e_1 \text{ in } e_2 : t_3}$$

Exercise: design a type system with functions

```
value ::= <nat>
        | <bool>
        | (v1, v2)
        | \x. e
```

```
type ::= Nat
       | Bool
       | (t1, t2)
       | t1 → t2
```

input type — output type

$$\frac{}{\lambda x.\ e \Rightarrow \lambda x.\ e}\ \text{Lambda}$$

$$\frac{\Gamma, x{:}t_1 \vdash e : t_2}{\Gamma \vdash \lambda x.\ e\ :\ t_1 \to t_2}\ \text{TLambda}$$

$t_1 \to t_2$: given by oracle

$$\frac{e_1 \Rightarrow \lambda x.\ e \qquad e_2 \Rightarrow v \qquad ([v/x]e = e') \qquad e' \Rightarrow v'}{e_1\ e_2 \Rightarrow v'}\ \text{App}$$

$$\frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_3 \qquad (t_1 = t_3)}{\Gamma \vdash e_1\ e_2 : t_2}\ \text{TApp}$$