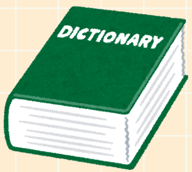
Inference rules are the common language spoken by PL designers.

わたしは　プログラミングげんごが　だいすきだ

watashi wa　puroguramingu gengo ga　daisuki da



mechanical process,
just like applying inference rules

I　programming　languages　love

🥳 🎉 Congratulations, you've graduated from having to model everything with inference rules.

😎 From this point on, we'll only model interesting language features using inference rules.

🤷 Non-interesting computation becomes a side condition

```
expr ::= nat
       | e1 + e2
       | e1 * e2
value ::= nat
```

$$
\begin{array}{rcll}
e \in expr & ::= & value & \text{value} \\
 & | & expr + expr & \text{addition} \\
 & | & expr \times expr & \text{multiplication} \\
 \\
v \in value & ::= & 0 & \text{zero} \\
 & | & S(value) & \text{successor}
\end{array}
$$

$$
\frac{}{v \Rightarrow v}\ \textsc{Val} \qquad \frac{e_1 \Rightarrow 0 \quad e_2 \Rightarrow v_2}{e_1 + e_2 \Rightarrow v_2}\ \textsc{Add0} \qquad \frac{e_1 \Rightarrow S(v_1) \quad v_1 + e_2 \Rightarrow v_2}{e_1 + e_2 \Rightarrow S(v_2)}\ \textsc{AddS}
$$

This is + in math (adding two nats)

$$
\frac{e_1 \Rightarrow n_1 \qquad e_2 \Rightarrow n_2 \qquad (n_1 + n_2 = n_3)}{e_1 + e_2 \Rightarrow n_3}\ \text{Add}
$$

This is + in language syntax

Name a concept that's common in all following expressions/programs

$$\int_0^{\pi} \int_x^{2x} \frac{x^2 + y^2}{\sqrt{x+y}} \, dy \, dx$$

```
def ipsum(x):
  def dolor(x):
    return x+1
  return x+2
```

$$f(x) = x^2 + 1$$

```
forall n, exists m, m > n + 1
```

```
template<typename T1, T2>
T1 max(T1 x, T1 y)
{ ... }
```

$$\sum_{i=0}^{\infty} \prod_{j=i}^{100} \sum_{i=j}^{100} (i+j)$$

VARIABLES

Activity: With your neighbor, come up with at least 4 categories.

Then for each row, sort each item into one of those categories.

1

```python
def lorem(x):
  return x + y
```

```python
def ipsum(x):
  def dolor(x):
    return x+1
  return x+2
```

```python
def sit(x):
  return 3
```

```python
def amet(x):
  return x
print(amet(3))
```

2

```cpp
template<class A> class LinkedList {
  template<class A> class Node
  { A value;
    Node* next; }
  Node<A>* head; }
```

```cpp
template<typename T1, T2>
T1 max(T1 x, T1 y)
{ ... }
```

```cpp
template<class Key>
class HashMap {
  void insert(Key, Val);
  Val* find(Key);
}
```

3

```
forall n, exists m, m > n + 1
```

```
exists u, u → (forall u, u ∨ -u)
```

```
forall x, exists y, x → true
```

```
exists f, f(x) > 0
```

4

$$f(x) = \int_0^x \sin 2x \, dx$$

$$\int_0^{\pi} \int_x^{2x} \frac{x^2 + y^2}{\sqrt{x+y}} \, dy \, dx$$

$$\int_0^{\pi} \int_x^{2x} \frac{x^2 + y^2}{\sqrt{x+y}} \, dx \, dy$$

$$\sum_{i=0}^{100} \prod_{j=i}^{100} \sum_{i=j}^{100} (i+j)$$

Concepts we discovered:

Variables are used / unused

Variables are undefined

Variables are shadowed

Count the number of variables

There are two kinds of variables

```
def lorem( x ):
    return x
```

What is the difference between x and x?

```
def lorem( x ):
    return 1
```

Is it okay to remove x?

Yes

```
def lorem(   ):
    return x
```

Is it okay to remove x?

No

There are two kinds of variables

```
def lorem( x ):

   return x
```

We call

- x a variable declaration
- x a variable reference

We learned:

- Every variable reference must be declared first.
- It's ok to declare a variable without referencing it.

For every variable, identify it as a declaration or a reference.

1

```python
def lorem(x):
  return x + y
```

```python
def ipsum(x):
  def dolor(x):
    return x+1
  return x+2
```

```python
def sit(x):
  return 3
```

```python
def amet(x):
  return x
print(amet(3))
```

2

```cpp
template<class A> class LinkedList {
  template<class A> class Node
  { A value;
    Node* next; }
  Node<A>* head; }
```

```cpp
template<typename T1, T2>
T1 max(T1 x, T1 y)
{ ... }
```

```cpp
template<class Key>
class HashMap {
  void insert(Key, Val);
  Val* find(Key);
}
```

3

```
forall n, exists m, m > n + 1
```

```
exists u, u → (forall u, u ∨ -u)
```

```
forall x, exists y, x → true
```

```
exists f, f(x) > 0
```

4

$$f(x) = \int_0^x \sin 2y \, dy$$

$$\int_0^{\pi} \int_x^{2x} \frac{x^2 + y^2}{\sqrt{x+y}} \, dy \, dx$$

$$\int_0^{\pi} \int_x^{2x} \frac{x^2 + y^2}{\sqrt{x+y}} \, dx \, dy$$

$$\sum_{i=0}^{100} \prod_{j=i}^{100} \sum_{i=j}^{100} (i+j)$$

```python
def lorem( x ):

    return x
```

- A reference points to declaration
- A declaration binds all references that point to it

If a reference:

- points to a declaration, we say it's bound
- doesn't point to a declaration, we say it's free (or unbound).

For each reference, determine whether it's bound or free.

1

```
def lorem(x):
  return x + y
```

```
def ipsum(x):
  def dolor(x):
    return x+1
  return x+2
```

```
def sit(x):
  return 3
```

```
def amet(x):
  return x
print(amet(3))
```

2

```
template<class A> class LinkedList {
  template<class A> class Node
  { A value;
    Node* next; }
  Node<A>* head; }
```

```
template<typename T1, T2>
T1 max(T1 x, T1 y)
{ ... }
```

```
template<class Key>
class HashMap {
  void insert(Key, Val);
  Val* find(Key);
}
```

3

forall n, exists m, m > n + 1

exists u, u → (forall u, u ∨ ¬u)

forall x, exists y, x → true

exists f, f(x) > 0

4

$$f(x) = \int_0^x \sin 2y \, dy$$

$$\int_0^{\pi} \int_x^{2x} \frac{x^2 + y^2}{\sqrt{x+y}} \, dy \, dx$$

$$\int_0^{\pi} \int_x^{2x} \frac{x^2 + y^2}{\sqrt{x+y}} \, dx \, dy$$

$$\sum_{i=0}^{100} \prod_{j=i}^{100} \sum_{i=j}^{100} (i+j)$$

How to know which declaration a reference points to?

"Binder"

Every declaration of x is associated with a scope:

- Scope = a subprogram or sub-expression (specified by the language designer as a "scoping rule")
- Every reference to x inside the scope points to the declaration with which the scope is associated

```
def lorem(x):
  return x + y
```

? bound free

```
(forall n, exists m,
m > n + 1) ∧ true
```

```
template<class Key>
class HashMap {
  void insert(Key, Val);
  Val* find(Key);
}
```

Draw an arrow from each reference to its declaration if it's bound

1

```
def lorem(x):
  return x + y
```

```
def ipsum(x):
  def dolor(x):
    return x+1
  return x+2
```

```
def sit(x):
  return 3
```

```
def amet(x):
  return x
print(amet(3))
```

2

```
template<class A> class LinkedList {
  template<class A> class Node
  { A value;
    Node* next; }
  Node<A>* head; }
```

```
template<typename T1, T2>
T1 max(T1 x, T1 y)
{ ... }
```

```
template<class Key>
class HashMap {
  void insert(Key, Val);
  Val* find(Key);
}
```

3

forall n, exists m, m > n + 1

forall x, exists y, x → true

exists u, u → (forall u, u ∨ ¬u)

exists f, f(x) > 0

4

$$f(x) = \int_0^x \sin 2y \, dy$$

$$\int_0^\pi \int_x^{2x} \frac{x^2 + y^2}{\sqrt{x+y}} \, dy \, dx$$

$$\int_0^\pi \int_x^{2x} \frac{x^2 + y^2}{\sqrt{x+y}} \, dx \, dy$$

$$\sum_{i=0}^{100} \prod_{j=i}^{100} \sum_{i=j}^{100} (i+j)$$

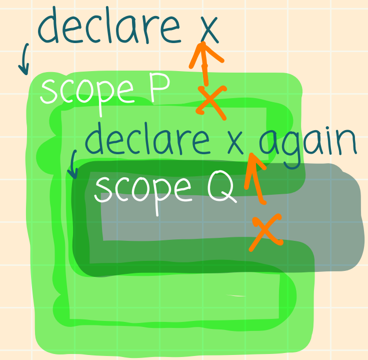
How to know which declaration a reference points to?

Every declaration of x is associated with a scope:

- Scope = a subprogram or sub-expression
- Every reference to x inside the scope points to the declaration with which the scope is associated

Caveat: scopes can be nested.

If you declare a variable x with scope P, and inside P, you declare variable x again with scope Q, then all references to x refers to the declaration associated with Q, not P.



Draw an arrow from each reference to its declaration if it's bound

1

```
def lorem(x):
  return x + y
```

```
def ipsum(x):
  def dolor(x):
    return x+1
  return x+2
```

```
def sit(x):
  return 3
```

```
def amet(x):
  return x
print(amet(3))
```

2

```
template<class A> class LinkedList {
  template<class A> class Node
  { A value;
    Node* next; }
  Node<A>* head; }
```

```
template<typename T1, T2>
T1 max(T1 x, T1 y)
{ ... }
```

```
template<class Key>
class HashMap {
  void insert(Key, Val);
  Val* find(Key);
}
```

3

forall n, exists m, m > n + 1

exists u, u → (forall u, u ∨ ¬u)

forall x, exists y, x → true

exists f, f(x) > 0

4

$f(x) = \int_0^x \sin 2y \, dy$

$\int_0^{\pi} \int_x^{2x} \frac{x^2 + y^2}{\sqrt{x+y}} \, dy \, dx$

$\int_0^{\pi} \int_x^{2x} \frac{x^2 + y^2}{\sqrt{x+y}} \, dx \, dy$

$\sum_{i=0}^{100} \prod_{j=i}^{100} \sum_{i=j}^{100} (i+j)$

I've been using natural language to explain variables...

- Fine if you're talking to human programmers who're trying to learn the language.

- NOT fine if you're designing the language

Let's make variables formal now!

Let's add local variables to the language of natural numbers

```
e2, e2 ∈ expr ::= nat
                | e1 + e2
                | e1 - e2
                | x
                | let x = e1 in e2
```

Informal semantics of let x = e1 in e2

1. First evaluate e1 to get a value v1
2. During the evaluation e2, any reference to x is v1
3. The overall let gets the value of e2

Exercise:

- which x in the grammar denotes declarations?
- which x denotes references?
- what is the scope of a declaration?

Let's add local variables to the language of natural numbers

```
e2, e2 ∈ expr ::= nat
               | e1 + e2
               | e1 - e2
               | x
               | let x = e1 in e2
```

Informal semantics of let x = e1 in e2

1. First evaluate e1 to get a value v1
2. During the evaluation e2, any reference to x is v1
3. The overall let gets the value of e2

For each expression, tell me what its value is based on the informal semantics.

1. let x = 1+2 in 2*x ⇒ 6
2. let x = 1+2 in (let y = 2*x in x*y) ⇒ 8

well-scoped

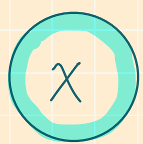
3. let x = x in 1 ill-scoped
4. (let x = 1 in x) + x ill-scoped
5. let x = (let x = 1 in x+1) in x ⇒ 2

Formally representing programs with variables and binders

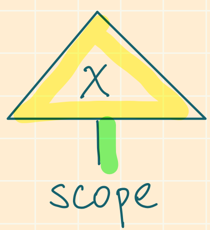
An abstract binding tree (ABT) is an AST with two new types of nodes

1. Variables: represent variable references (no children)
2. Binders represent declarations

A binder node is annotated with the declared variable, and a single subtree called the scope (where references are bound).



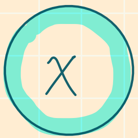
Variable node



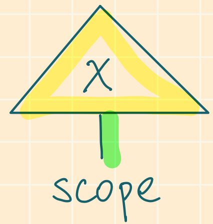
Binder node

Abstract binding trees (ABTs)

```
e2, e2 ∈ expr ::= 0 | S(0)
                | e1 + e2
                | e1 * e2
                | x
              ? | let x = e1 in e2
```



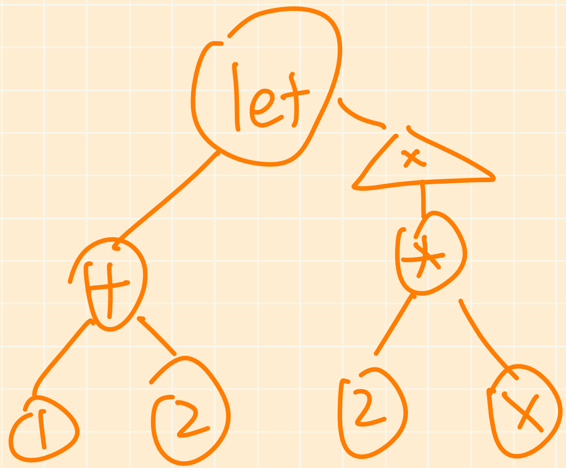
Variable node



Binder node

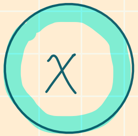
Examples:

let x = 1+2 in 2*x

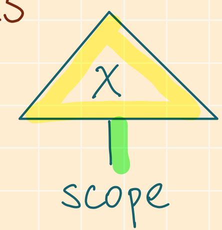


Abstract binding trees

```
e2, e2 ∈ expr ::= 0 | S(0)
                | e1 + e2
                | e1 - e2
                | x
                | let x = e1 in e2
```



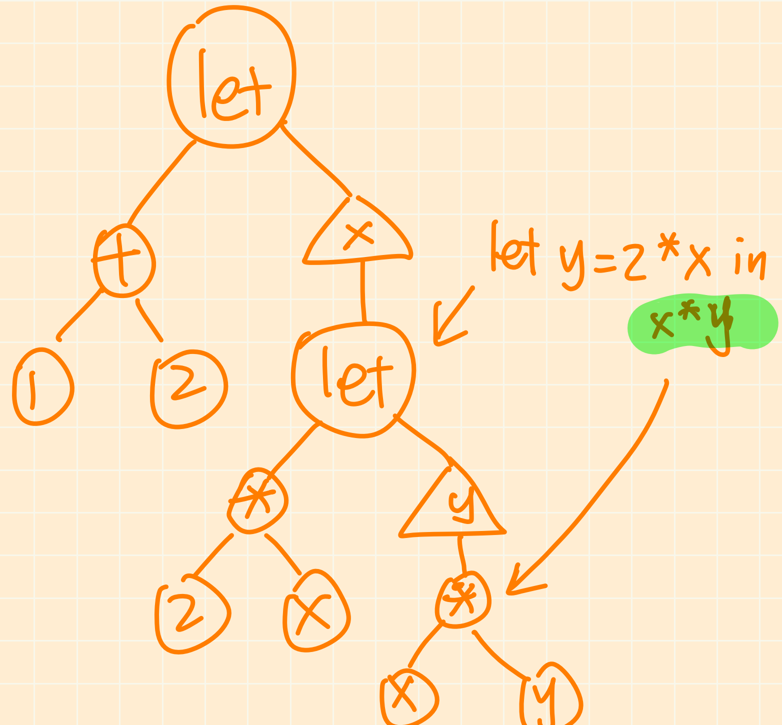
Variable node



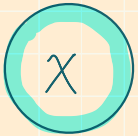
Binder node

Examples:

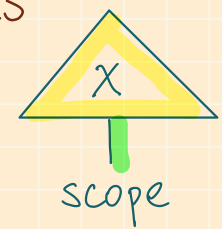
let x = 1+2 in let y = 2*x in x*y



Abstract binding trees



Variable node



Binder node

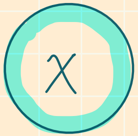
```
e2, e2 ∈ expr ::= 0 | S(0)
                | e1 + e2
                | e1 - e2
                | x
                | let x = e1 in e2
```

Examples:

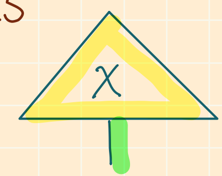
let x = x in 1

Abstract binding trees

```
e2, e2 ∈ expr ::= 0 | S(0)
                | e1 + e2
                | e1 - e2
                | x
                | let x = e1 in e2
```



Variable node



scope

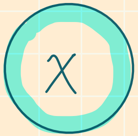
Binder node

Examples:

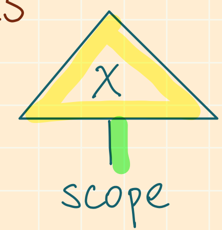
(let x = 1 in x) + x

```
e2, e2 ∈ expr ::= 0 | S(0)
                | e1 + e2
                | e1 - e2
                | x
                | let x = e1 in e2
```

Abstract binding trees



Variable node



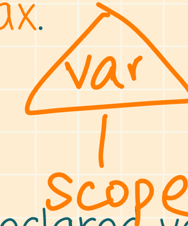
Binder node

Examples:

let x = (let x = 1 in x+1) in x

To specify binder nodes in CFGs, we use the binder syntax.

A binder syntax has the form "<var>. <scope>" =

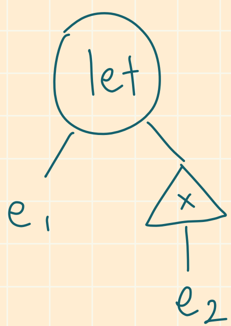


- <var> is the variable being declared
- <scope> is the subtree of the binder node where the declared variable is in scope

```
e2, e2 ∈ expr ::= nat | e1 + e2
                | x
                | let x = e1 in e2
```

"let" in binder syntax

→ `let(e1, x. e2)`



⇐

Meaning: "let" is an ABT node with 2 children:

- The left child is an expr ABT (e1)
- The right child is a binder node, where
- it declares variable x
- The scope of x is an expr ABT (e2)

Semantics of variables

```
e2, e2 ∈ expr ::= 0 | S(0)
                | e1 + e2
                | e1 - e2
                | x
                | let x = e1 in e2
```

Informal semantics of let x = e1 in e2

1. First evaluate e1 to get a value v1
2. During the evaluation e2, any reference to x is v1
3. The overall let gets the value of e2

$$\frac{e_1 \Rightarrow v_1 \qquad \underbrace{}_{?}}{\text{let } x = e_1 \text{ in } e_2 \Rightarrow ?}\ \text{Let}$$

We should be able to show `let x = 3 in 2*x` ⇒ 6

The meaning of variables is given by substitution

Context

We should:

```
let x = 3 in 2*x
```

Substitute x with 3 in 2*x

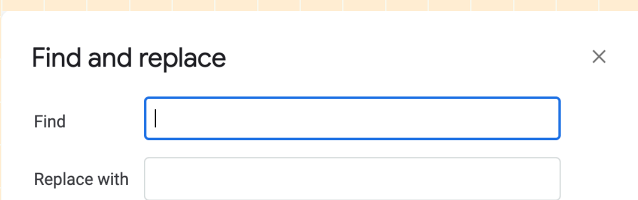
The substitution function `[r/x]e` substitutes variable x with replacement expression r in context expression e

---

Important: we'll use substitution to define operational semantics.

But substitution itself doesn't evaluate/run the program

The only thing it does:



The substitution function `[r/x]e` substitutes variable references x with replacement expression r in context expression e:

Defined recursively over the context expression e

```
[r/x]e =
  if e is a number: return x

  if e is e1 + e2 or e1 * e2:

  if e is variable y:
        if x = y: return r
        else    : return e
 (if e is let(e1, y. e2): )
```

$[3/x](4) = 4$

$[3/x](x) = 3$

$[3/x](y) = y$

```
[3/x]2

[3/x]x

[3/x]y

[3/x](2 + x)

[3/x](x * y)

[10/y](x + y * x)
```

```
[r/x] e =
  if e is Num(n):
      return e
  if e is Add(e1, e2):
      return Add(e1, e2)
  if e is Mul(e1, e2):
      return Mul(e1, e2)
  if e is Var(y):
    if x == y:
      return r
    else:
      return e
  if e is let(e1, y. e2):
    TODO
```

The meaning of variables is given by substitution

```
let x = 3 in 2*x
```

$[3/x]$(let x=x+1 in x*2)

```
let x = 3 in
(let x = x+1 in x*2)
```

3

4*2=8

We should:

Substitute x with ___ in ____

$=[3/x]$ let(~~x~~+1, x. x*2)

3

Substitute x with ___ in ____

so don't need to do anything

The substitution function `[r/x]c` substitutes variable x with replacement expression r in context expression e

Key: we should replace references to x that are free

The substitution function **[r/x]e** substitutes free references of x with replacement expression r in context expression e:

Defined recursively over the context expression e

```
[r/x] e =
  if e is a number: e
  if e is Add(e1, e2): Add([r/x]e1, [r/x]e2)
  if e is variable y:
    if x == y: return r
    else: return e
  if e is let(e1, y. e2):
    if x == y:
```

$\mathrm{let}([r/x]e_1, \ y.\ e_2)$

```
    else:
```

$\mathrm{let}([r/x]e_1, \ y.\ [r/x]e_2)$

[3/x] (x + 1) = 3 + 1

[3/x] let y = x in y   let y = 3 in y

[3/x] let y = x in x

[3/x] let x = y in x

[3/x] let x = x in x

```
[r/x] e =
if e is Num(n):
    return e
if e is Add(e1, e2):
    return Add(e1, e2)
if e is Var(y):
  if x == y:
    return r
  else:
    return e
if e is let(e1, y. e2):
  e1s = [r/x]e1
  if x == y:
    return let(e1s, y. e2)
  else:
    e2s = [r/x]e2
    return let(e1s, y. e2s)
```

[3/x] (let y = x in (let x = y in x + y)) = let y = 3 in $\boxed{\text{[3/x] let x = y in x + y}}$ = (let x = y in x + y)

Using substitution to define semantics of language features with binders:

$$\frac{e_1 \Rightarrow v_1}{\text{let } x = e_1 \text{ in } e_2 \Rightarrow v_2}\ \text{Let}$$

```
[r/x] e =
if e is Num(n):
    return e
if e is Add(e1, e2):
    return Add(e1, e2)
if e is Var(y):
  if x == y: return r
  else: return e
if e is let(e1, y. e2):
  e1s = [r/x]e1
  if x == y:
    return let(e1s, y. e2)
  else:
    e2s = [r/x]e2
    return let(e1s, y. e2s)
```

Examples

1. let x = 1+2 in 2*x
2. let x = 1+2 in let y = 2*x in x*y
3. let x = x in 1
4. (let x = 1 in x) + x
5. let x = (let x = 1 in x+1) in x