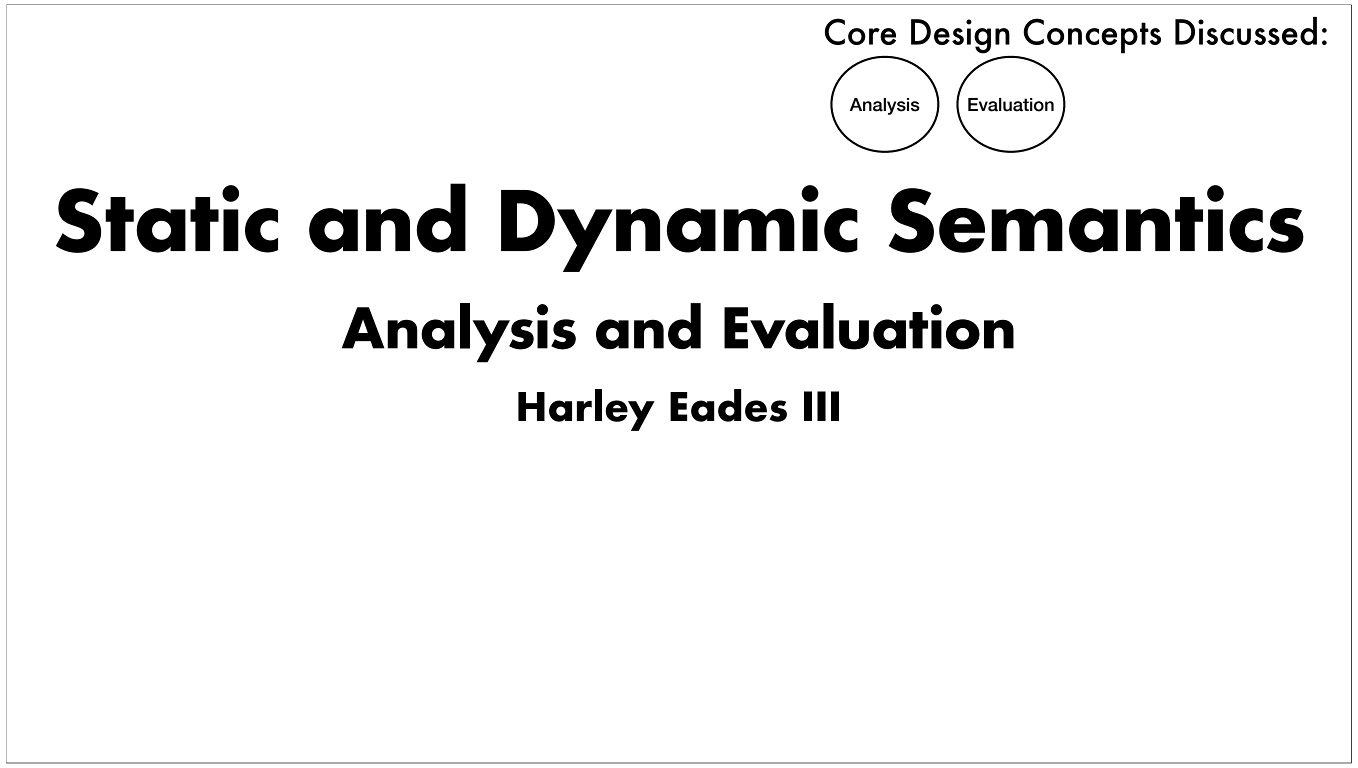


Core Design Concepts Discussed:

Analysis Evaluation

Static and Dynamic Semantics

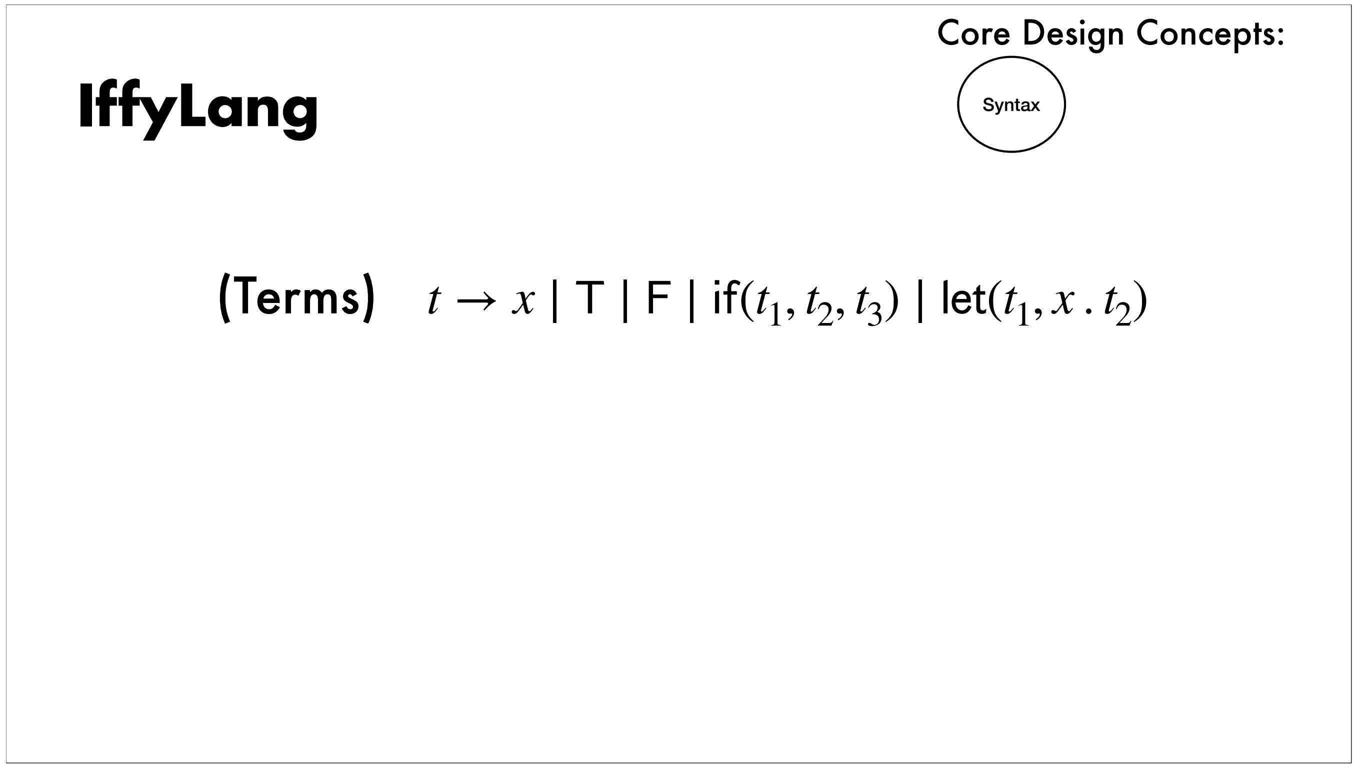
Analysis and Evaluation

Harley Eades III

In this lecture we take things a bit further and formally define a lot of the structures we have been using in class. Our ultimate goal is to take a look at two new important concepts to the design and implementation of programming languages: their static semantics which is called type checking and their dynamic semantics which is called evaluation of programs. In this lecture we will use a very simply language to introduce the concepts.

Outline:

- Binding:
    - substitution
    - alpha-eq terms
    - alpha-eq classes
    - capture-avoiding substitution
- Typing:
    - hypothetical judgments
    - judgment reading
    - typing rules
    - closed typing has no free variables (the point of typing in simple lang)
- Evaluation:
    - hypothetical environment form
    - alternate substitution version with non-hypothetical judgment
    - Canonical forms lemma using environment



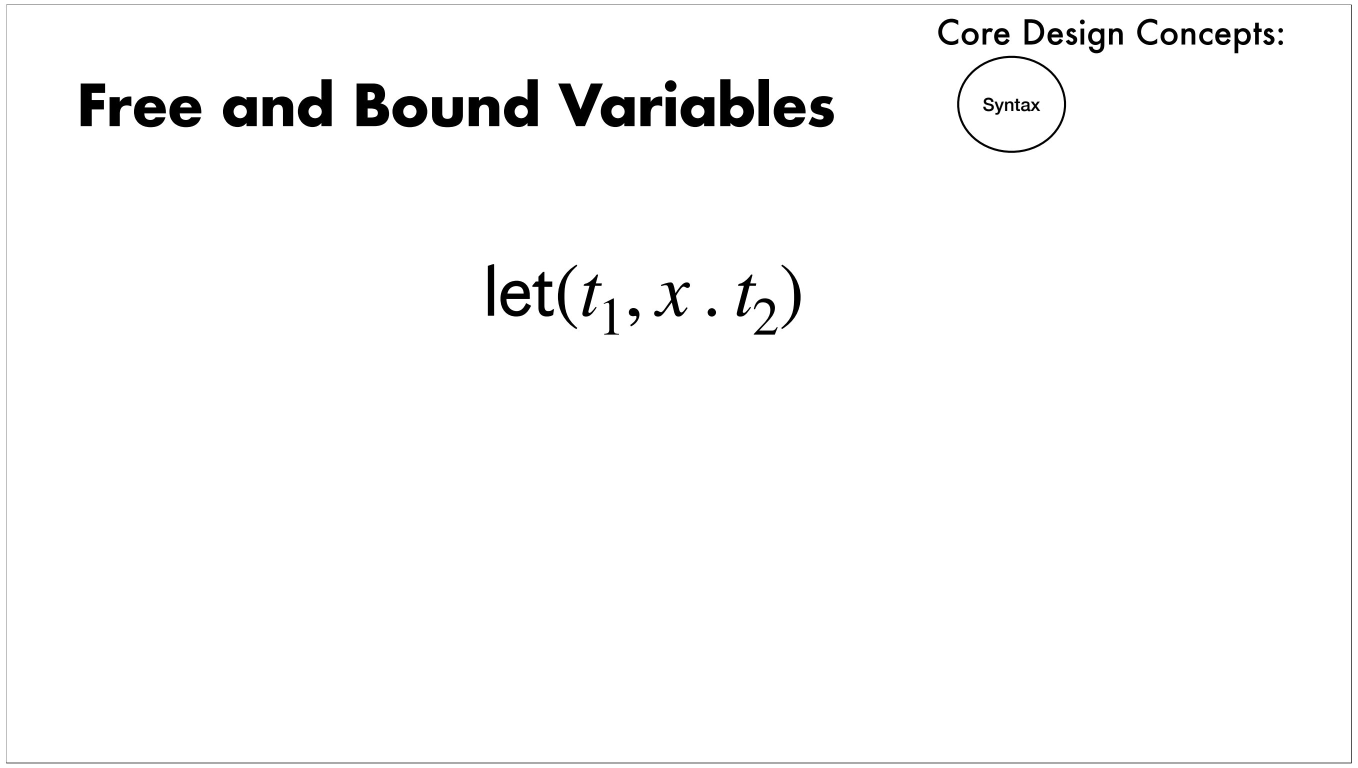
Core Design Concepts:

Syntax

IffyLang

(Terms) $t \to x \mid \mathsf{T} \mid \mathsf{F} \mid \mathsf{if}(t_1, t_2, t_3) \mid \mathsf{let}(t_1, x\,.\,t_2)$

The language we will be using we call IffyLang which consists of variables, true, false, if-then-else, and definitions. Notice that we are using abstract syntax, because the concepts we are introducing are internal notions of a programming language and are always implemented using the abstract syntax. First, let's consider the notion of...



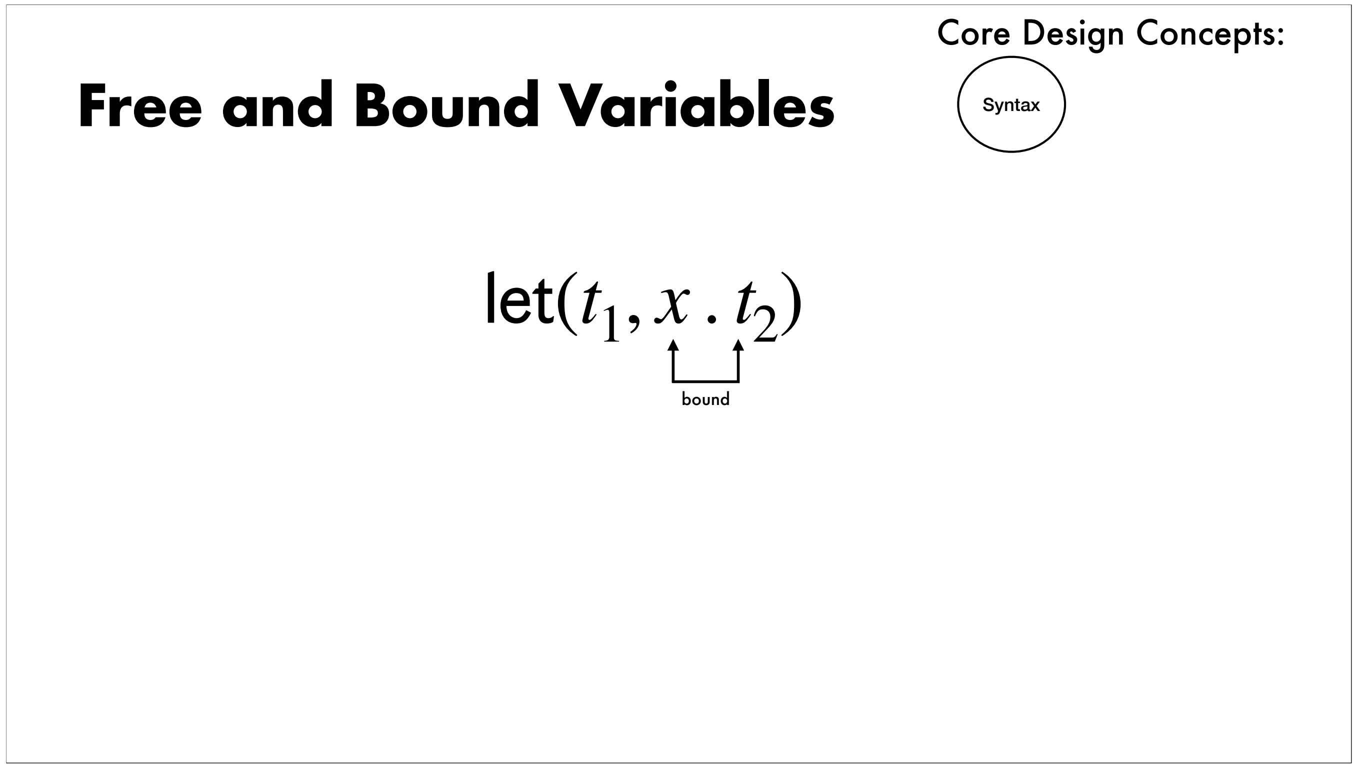
Core Design Concepts:

Syntax

Free and Bound Variables

$$\text{let}(t_1, x \,.\, t_2)$$

free and bound variables. In the let-expression used for definitions, we say that...



Free and Bound Variables

Core Design Concepts:

Syntax

$$\text{let}(t_1, \underbrace{x \,.\, t_2}_{\text{bound}})$$

x, is bound in t_2. Thus, t_2 is the scope of x. We call anything that binds a variable to it's scope a binder; so let is a binder and it's bound variable is x. This is the same for examples like parameters of a method, type variables in class definitions, parameters to functions, and any other structure that binds a variable. Now we are allowed to have a variable that is not associated with a binder, because of...

Core Design Concepts:

Syntax

Free and Bound Variables

(Terms) $t \rightarrow x$ | T | F | if(t_1, t_2, t_3) | let$(t_1, x \,.\, t_2)$

the x in the grammar. For example, ...

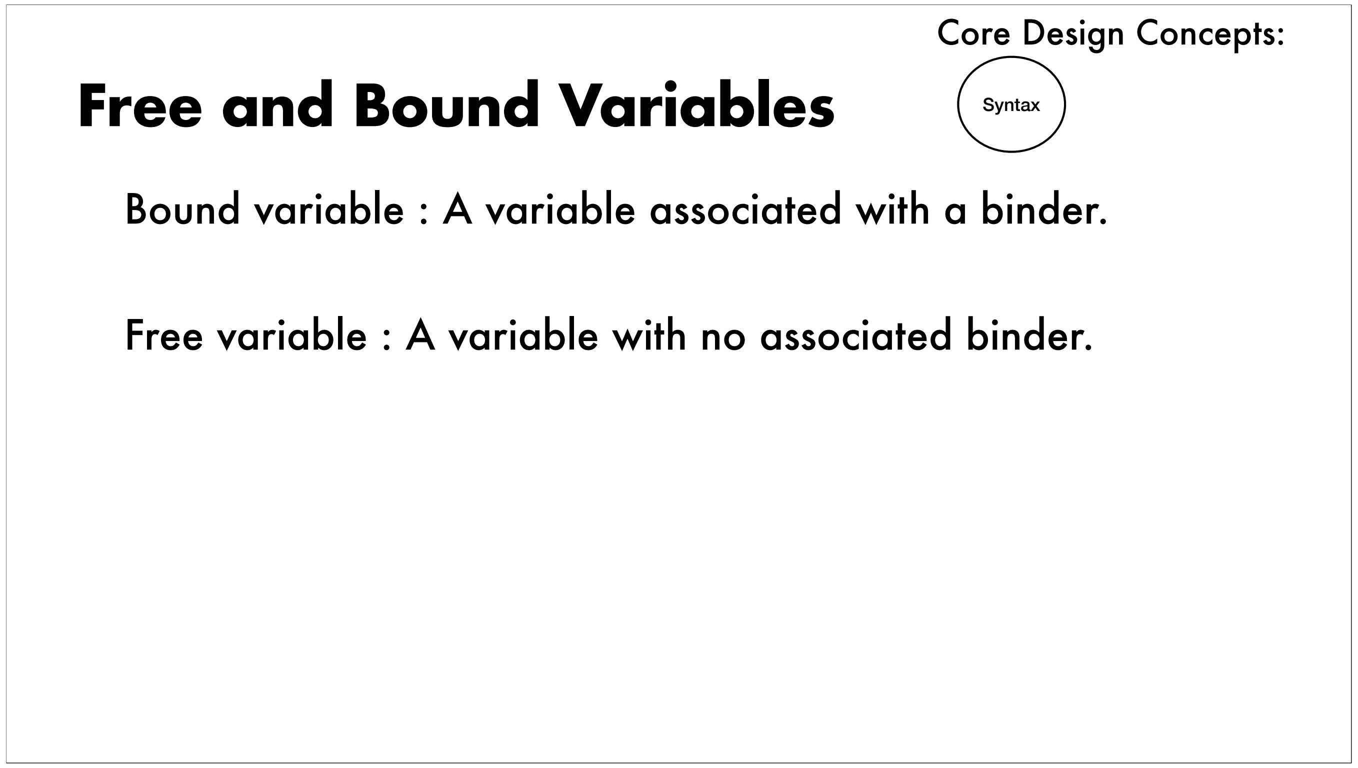
Core Design Concepts:

Free and Bound Variables

Syntax

$$\mathrm{if}(y, \mathsf{T}, \mathsf{F})$$

y has no associated binder. A variable like y with no associated binder is called a free variable....



Core Design Concepts:

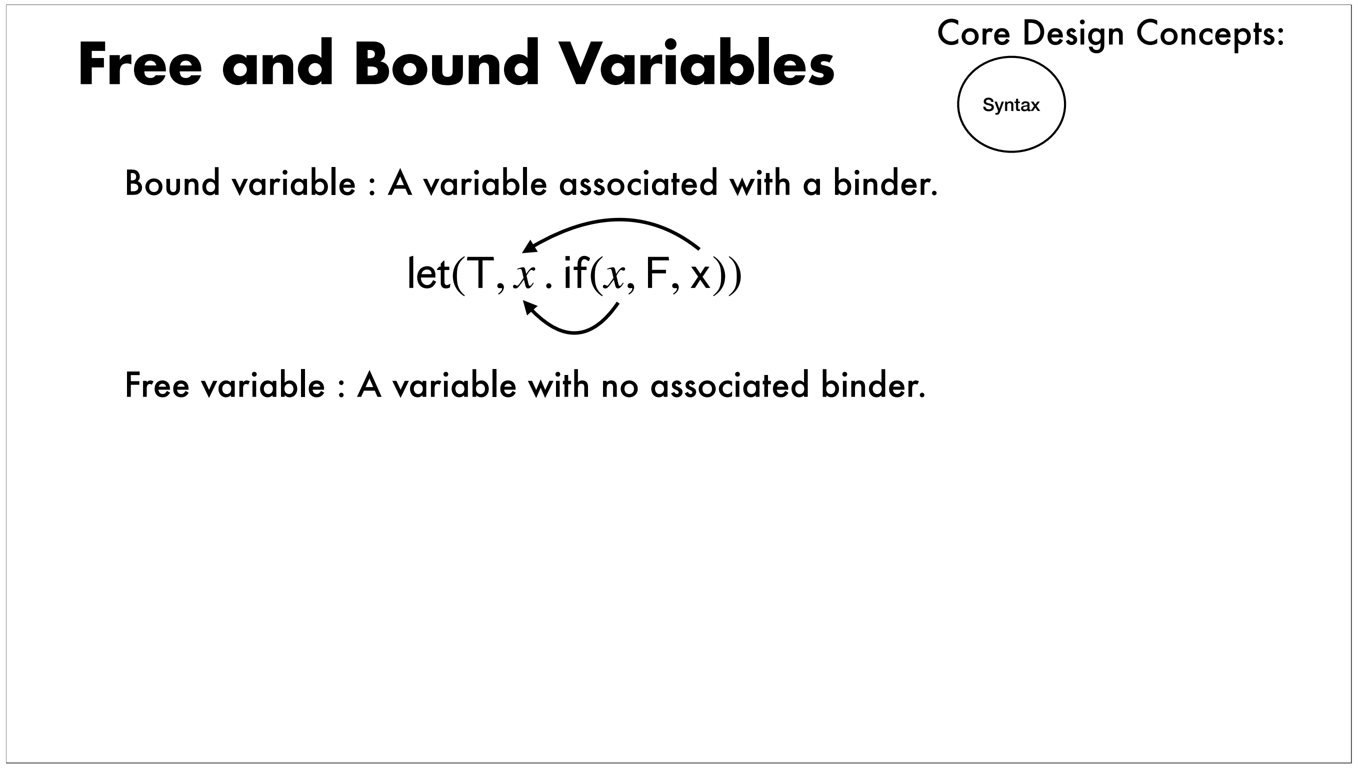
Free and Bound Variables

Syntax

Bound variable : A variable associated with a binder.

Free variable : A variable with no associated binder.

So as definitions a bound variable has an associated binder, but a free variable has no associated binder. Here are two more examples...



Free and Bound Variables

Core Design Concepts:

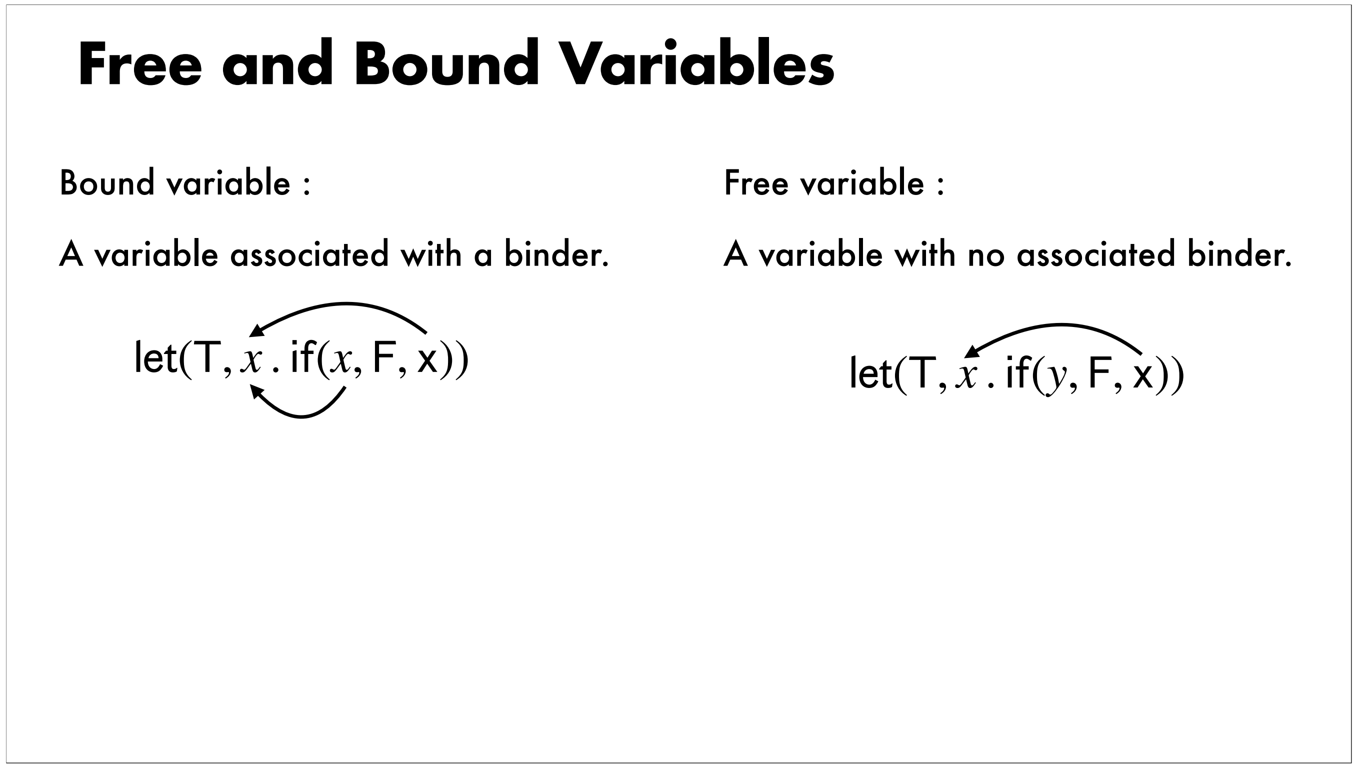
Syntax

Bound variable : A variable associated with a binder.

$$\begin{tikzcd}[column sep=tiny] \mathrm{let}(\mathsf{T}, & x & .\,\mathrm{if}(& x \arrow[ll, bend left] & ,\mathsf{F}, & \mathrm{x} \arrow[llll, bend right] &)) \end{tikzcd}$$

Free variable : A variable with no associated binder.

the first has two occurrences of the bound variable x and arrows pointing to their binder. The second...



Free and Bound Variables

Bound variable :

A variable associated with a binder.

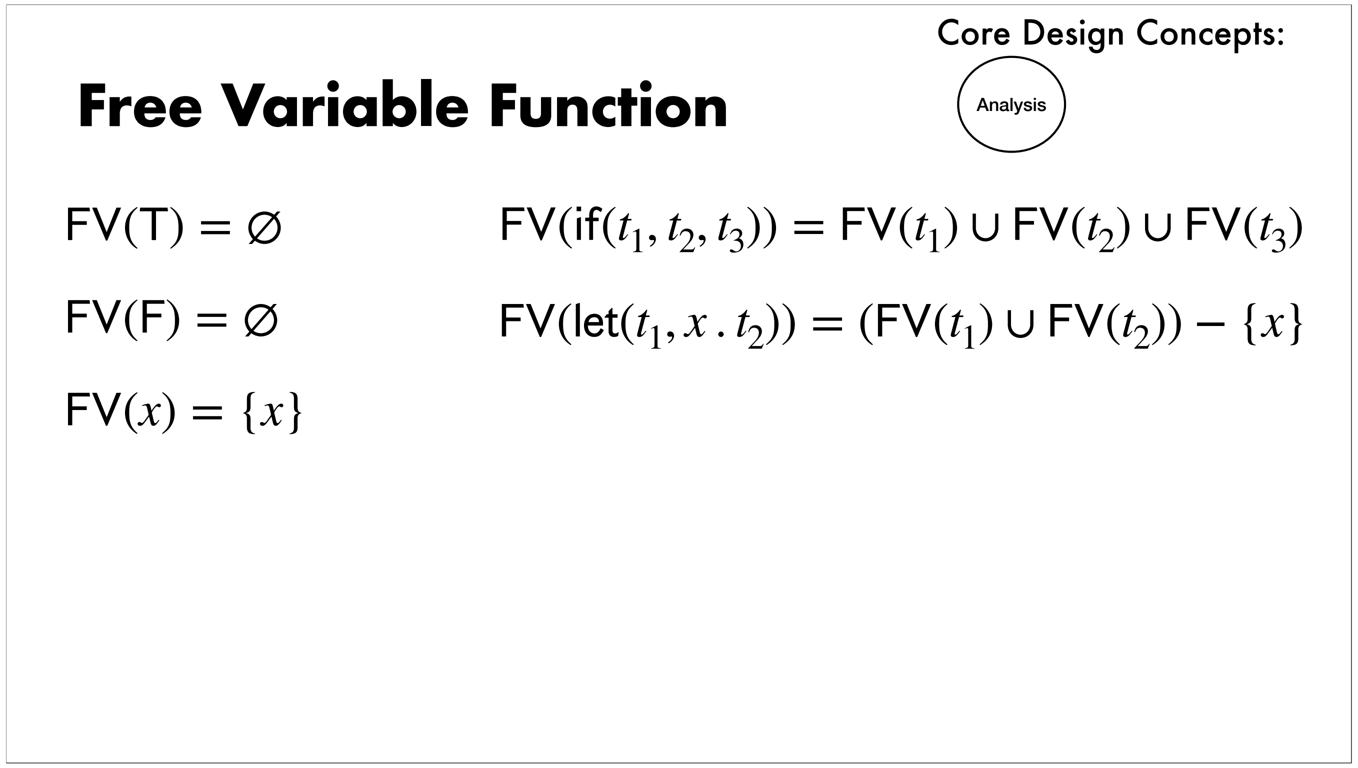
$$\text{let}(\mathsf{T}, x\,.\,\text{if}(x, \mathsf{F}, \mathsf{x}))$$

Free variable :

A variable with no associated binder.

$$\text{let}(\mathsf{T}, x\,.\,\text{if}(y, \mathsf{F}, \mathsf{x}))$$

has a free variable y, but also a bound variable x. The set of free variables in term can be computed by...



Core Design Concepts:

Analysis

Free Variable Function

$\mathsf{FV}(\mathsf{T}) = \varnothing$

$\mathsf{FV}(\mathsf{F}) = \varnothing$

$\mathsf{FV}(x) = \{x\}$

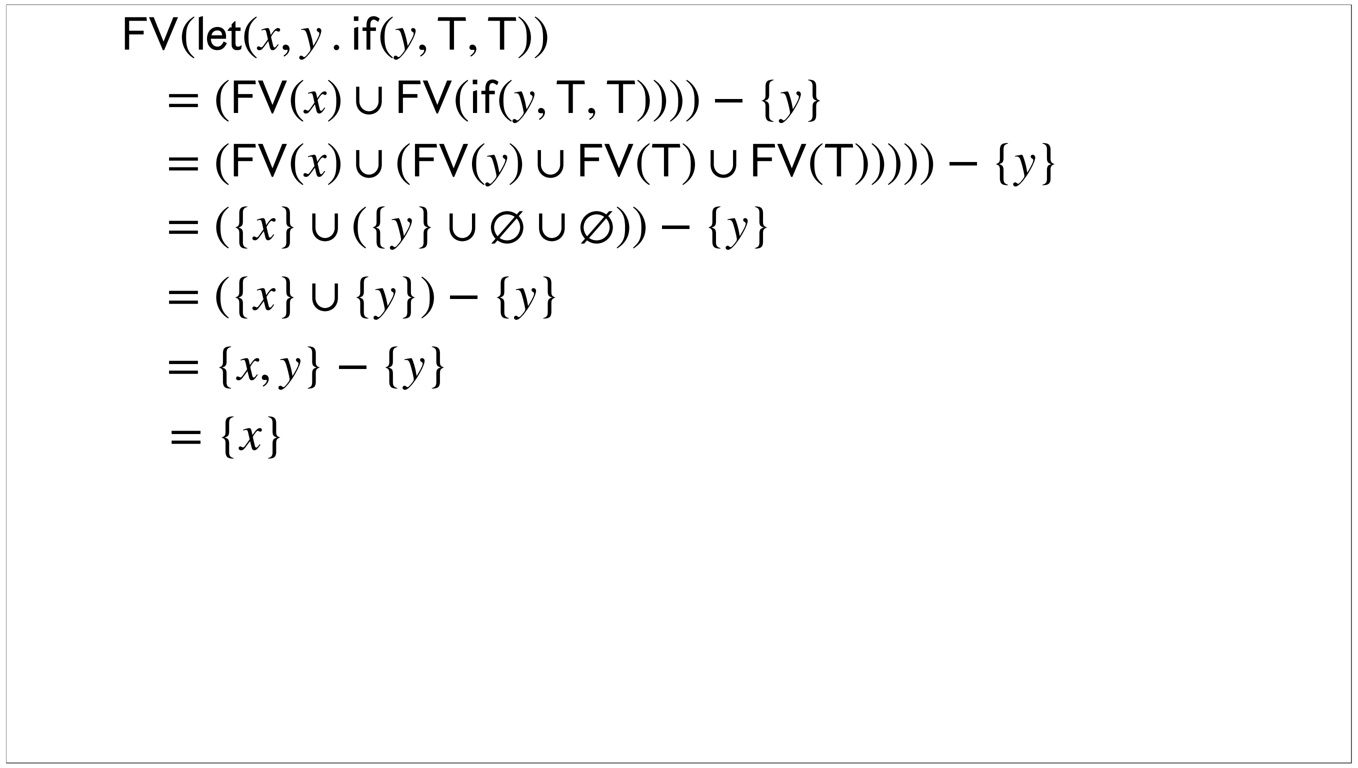
$\mathsf{FV}(\mathsf{if}(t_1, t_2, t_3)) = \mathsf{FV}(t_1) \cup \mathsf{FV}(t_2) \cup \mathsf{FV}(t_3)$

$\mathsf{FV}(\mathsf{let}(t_1, x \,.\, t_2)) = (\mathsf{FV}(t_1) \cup \mathsf{FV}(t_2)) - \{x\}$

The free function is defined by recursion the structure of abstract terms. It simply moves down the parse tree until it hits the leaves and collects all the variables without a binder in a set. Note that in the case of let we must remove the bound variable from the set of variables collected in t_1 union t_2, because t_2 may depend on x. However, this definition introduces a problem...

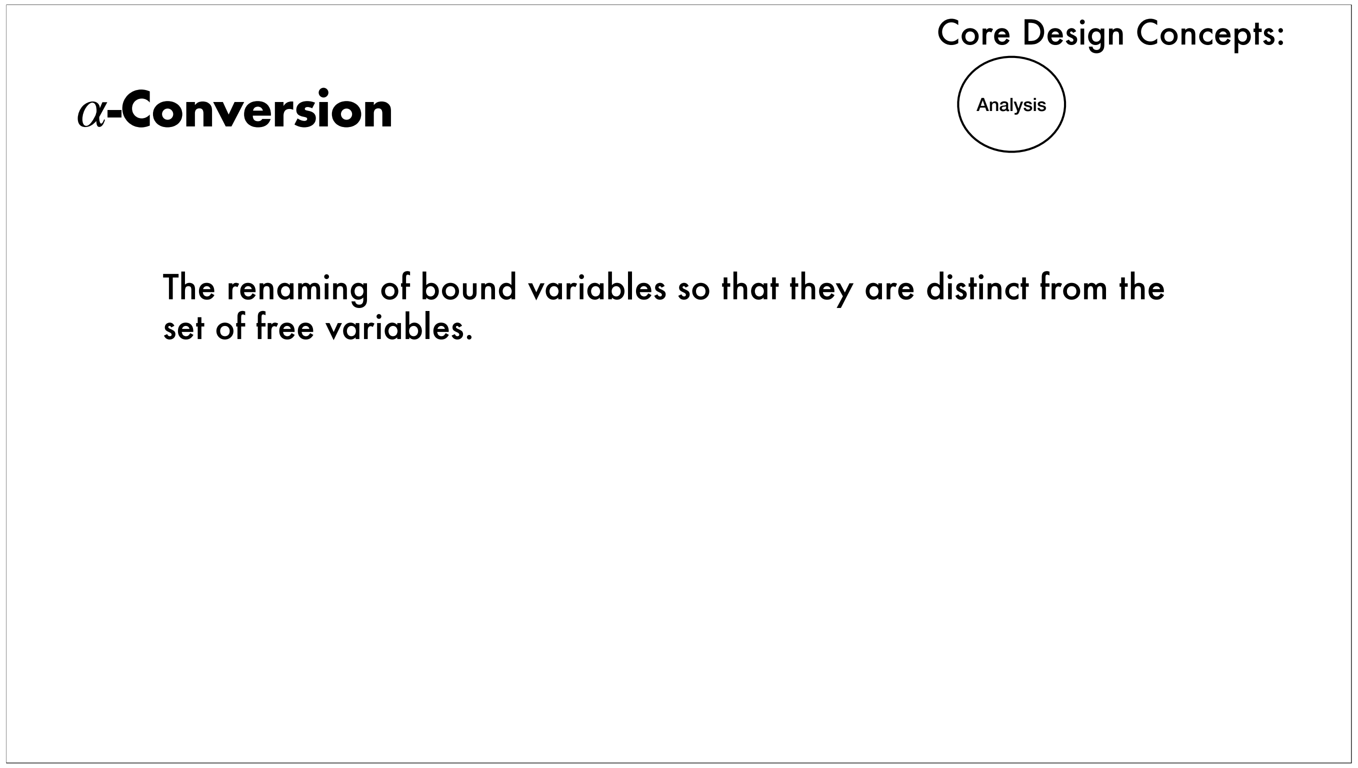
$$
\begin{aligned}
&\mathsf{FV}(\mathsf{let}(x, x\,.\,\mathsf{if}(x, \mathsf{T}, \mathsf{T})) \\
&\quad = (\mathsf{FV}(\{x\}) \cup \mathsf{FV}(\mathsf{if}(x, \mathsf{T}, \mathsf{T})))) - \{x\} \\
&\quad = (\mathsf{FV}(\{x\}) \cup (\mathsf{FV}(x) \cup \mathsf{FV}(\mathsf{T}) \cup \mathsf{FV}(\mathsf{T}))))) - \{x\} \\
&\quad = (\{x\} \cup (\{x\} \cup \varnothing \cup \varnothing)) - \{x\} \\
&\quad = (\{x\} \cup \{x\}) - \{x\} \\
&\quad = \{x\} - \{x\} \\
&\quad = \varnothing
\end{aligned}
$$

Here we compute the free variables in a term that has both a bound variable x and a free variable x. The definition of the free variable function results in the empty set, but it should be a singleton set with x in it. The problem is that the free variable is shadowing the bound variable. We can fix this problem by renaming the bound variable to a unique name....



$$
\begin{aligned}
&\mathsf{FV}(\mathsf{let}(x, y\,.\,\mathsf{if}(y, \mathsf{T}, \mathsf{T}))\\
&\quad = (\mathsf{FV}(x) \cup \mathsf{FV}(\mathsf{if}(y, \mathsf{T}, \mathsf{T})))) - \{y\}\\
&\quad = (\mathsf{FV}(x) \cup (\mathsf{FV}(y) \cup \mathsf{FV}(\mathsf{T}) \cup \mathsf{FV}(\mathsf{T}))))) - \{y\}\\
&\quad = (\{x\} \cup (\{y\} \cup \varnothing \cup \varnothing)) - \{y\}\\
&\quad = (\{x\} \cup \{y\}) - \{y\}\\
&\quad = \{x, y\} - \{y\}\\
&\quad = \{x\}
\end{aligned}
$$

After renaming the variable we see that the free variable function computes the right set of free variables. This operation of renaming variables is called...



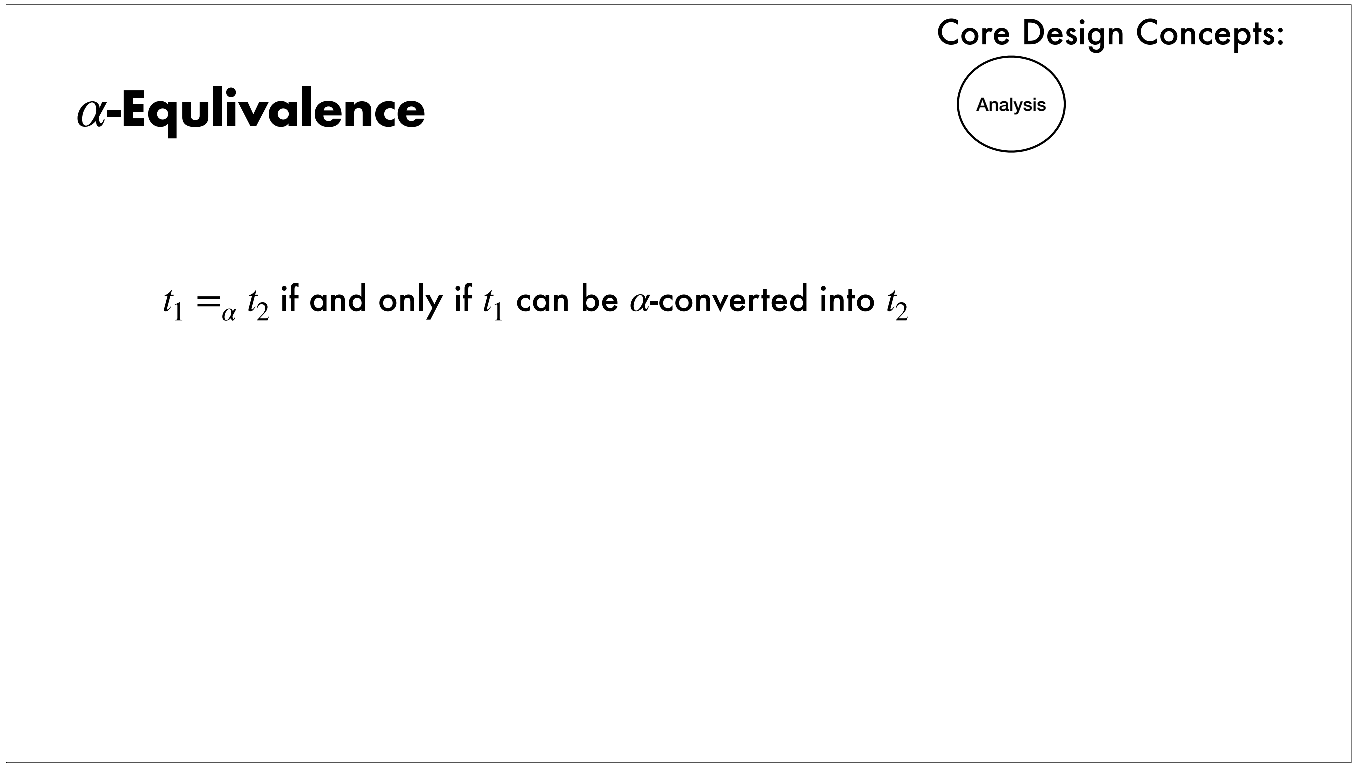
Core Design Concepts:

Analysis

α-Conversion

The renaming of bound variables so that they are distinct from the set of free variables.

alpha-conversion. We will use alpha-conversion to define an equivalence relation between terms...



Core Design Concepts:

Analysis

α-Equlivalence

$t_1 =_\alpha t_2$ if and only if t_1 can be α-converted into t_2

called alpha-equivalence which considers two terms equal if and only if they can be alpha-converted, that is, their variables can be renamed, so that they are the same term. This then gives rise to the notion of...

α-Equlivalence Classes

Core Design Concepts:

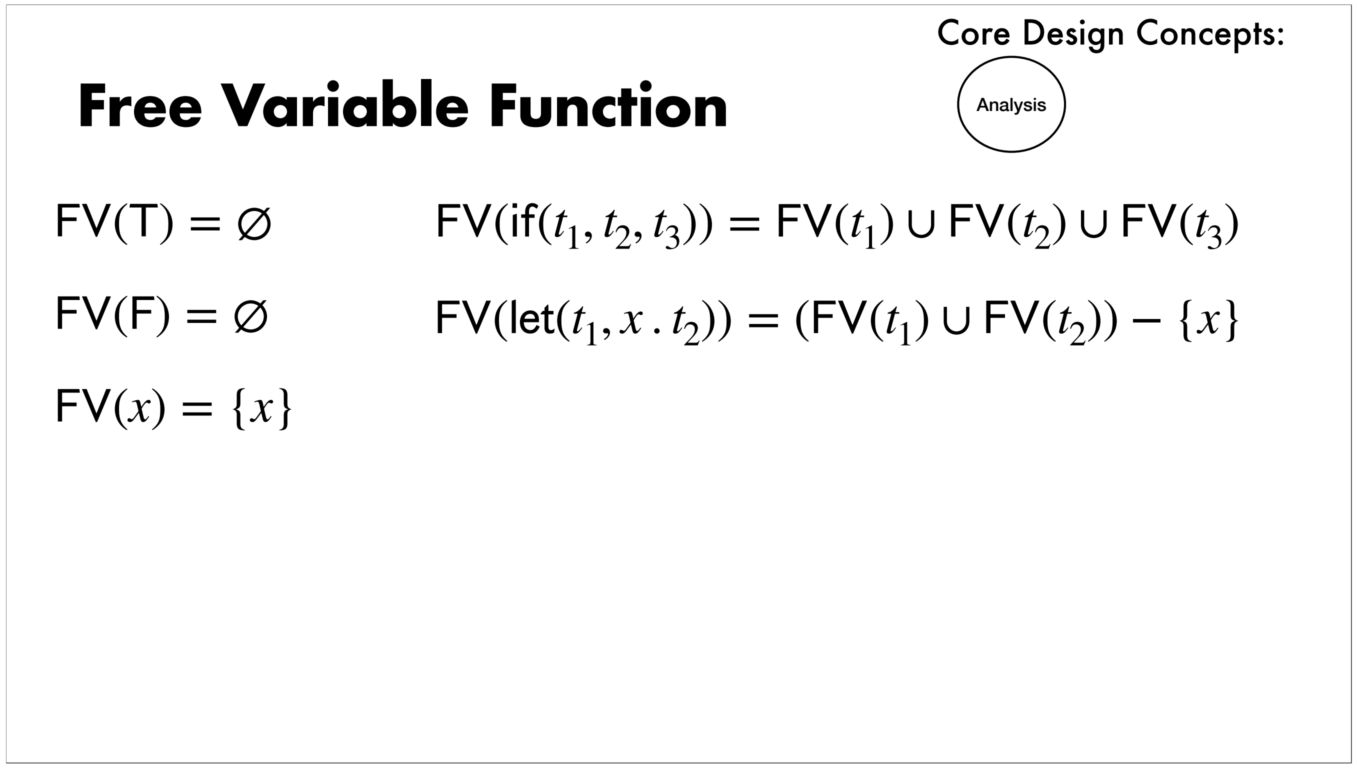
Analysis

$$[t] = \{t' \mid t =_\alpha t'\}$$

We will now be working up to alpha-equivalence.

All operations, judgments, etc will be defined on α-equivalence classes.

alpha-equivalence classes which are the set of alpha-equivalent terms. This is very important, because we will now be working up to alpha-equivalence, that is, we will only consider two terms different if they differ by more than alpha-eqiivance. So when we speak of a particular term then we are speaking of it's alpha-equivalence. Then we will define all our operations and judgments on terms to be on alpha-equivalence classes. Now to make syntax easy to read and write we will not write the square brackets around terms, but we will be implicitly using alpha-equivalence and conversion. Now going back to the free-variable function...



Core Design Concepts:

Analysis

Free Variable Function

$\mathsf{FV}(\mathsf{T}) = \varnothing$

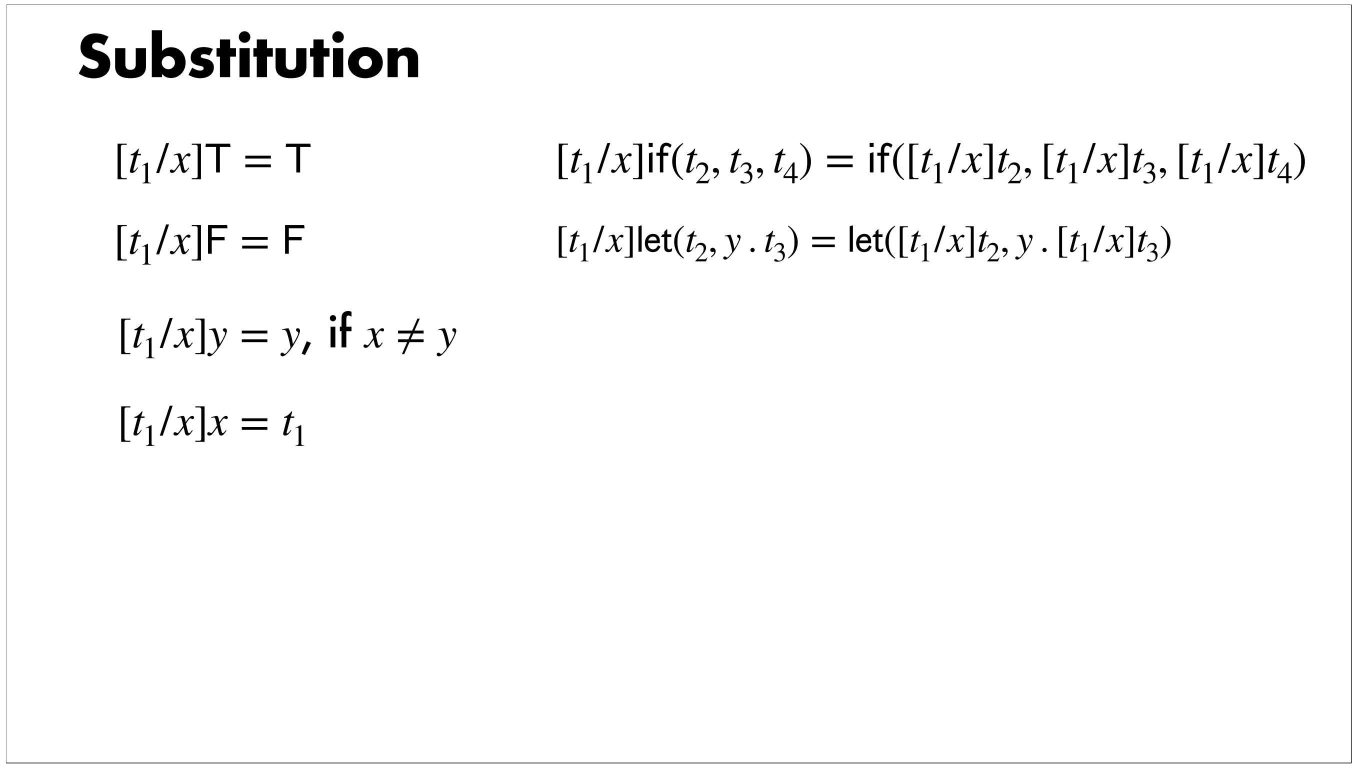
$\mathsf{FV}(\mathsf{F}) = \varnothing$

$\mathsf{FV}(x) = \{x\}$

$\mathsf{FV}(\mathsf{if}(t_1, t_2, t_3)) = \mathsf{FV}(t_1) \cup \mathsf{FV}(t_2) \cup \mathsf{FV}(t_3)$

$\mathsf{FV}(\mathsf{let}(t_1, x \,.\, t_2)) = (\mathsf{FV}(t_1) \cup \mathsf{FV}(t_2)) - \{x\}$

Viewing the input as an alpha-equivalence class we can see that we will never run into the problem we did before, because in the final case of let-expressions x cannot appear free in t_1, because t_1 and t_2 are assumed to be alpha-distinct terms, and thus, are members of different alpha-equivalence classes, and so have distinctly named variables. A similar problem called...



Substitution

$[t_1/x]\mathsf{T} = \mathsf{T}$

$[t_1/x]\mathsf{F} = \mathsf{F}$

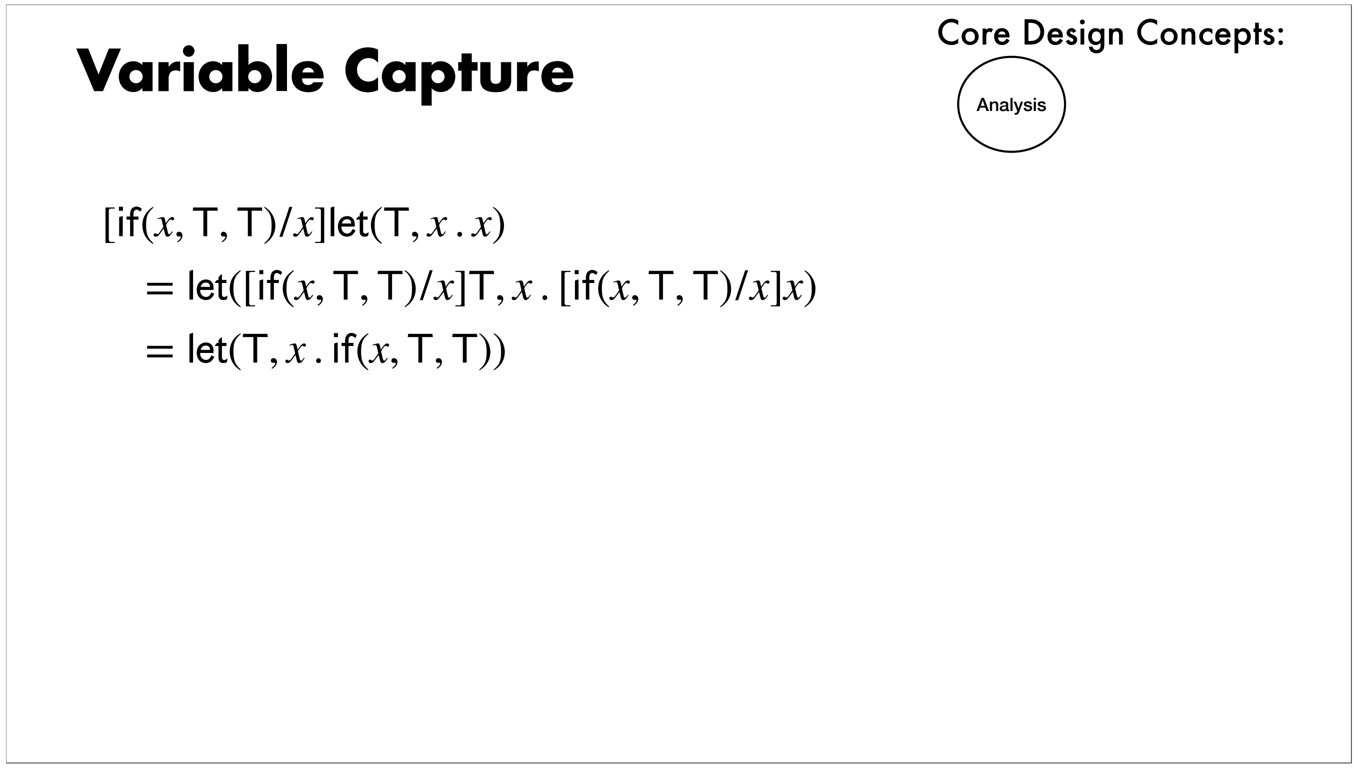
$[t_1/x]y = y$, if $x \neq y$

$[t_1/x]x = t_1$

$[t_1/x]\mathsf{if}(t_2, t_3, t_4) = \mathsf{if}([t_1/x]t_2, [t_1/x]t_3, [t_1/x]t_4)$

$[t_1/x]\mathsf{let}(t_2, y\,.\,t_3) = \mathsf{let}([t_1/x]t_2, y\,.\,[t_1/x]t_3)$

comes up as a result of substitution defined here. Substitution takes as input a term t1, a variable x, and a term t2, then replaces every occurrence of x in t2 with t1. If you look at the case for let, we can see that, if t1 has a free variable y, then it becomes bound when x is replaced in t3 by t1. Let's consider an example...



Variable Capture

Core Design Concepts:

Analysis

$$
\begin{aligned}
&[\mathsf{if}(x, \mathsf{T}, \mathsf{T})/x]\mathsf{let}(\mathsf{T}, x\,.\,x) \\
&\quad = \mathsf{let}([\mathsf{if}(x, \mathsf{T}, \mathsf{T})/x]\mathsf{T}, x\,.\,[\mathsf{if}(x, \mathsf{T}, \mathsf{T})/x]x) \\
&\quad = \mathsf{let}(\mathsf{T}, x\,.\,\mathsf{if}(x, \mathsf{T}, \mathsf{T}))
\end{aligned}
$$

Oh, no, if we look at the term we are using to replace x with, the if-expression, we see that x is a free variable, and so after substitution, it better still be a free variable or we have changed the meaning of the program. But, after substitution, we can see that it is indeed a bound variable! We can prevent this by modifying the definition of substitution slightly...

Substitution

$[t_1/x]\mathsf{T} = \mathsf{T}$

$[t_1/x]\mathsf{F} = \mathsf{F}$

$[t_1/x]y = y$, if $x \neq y$

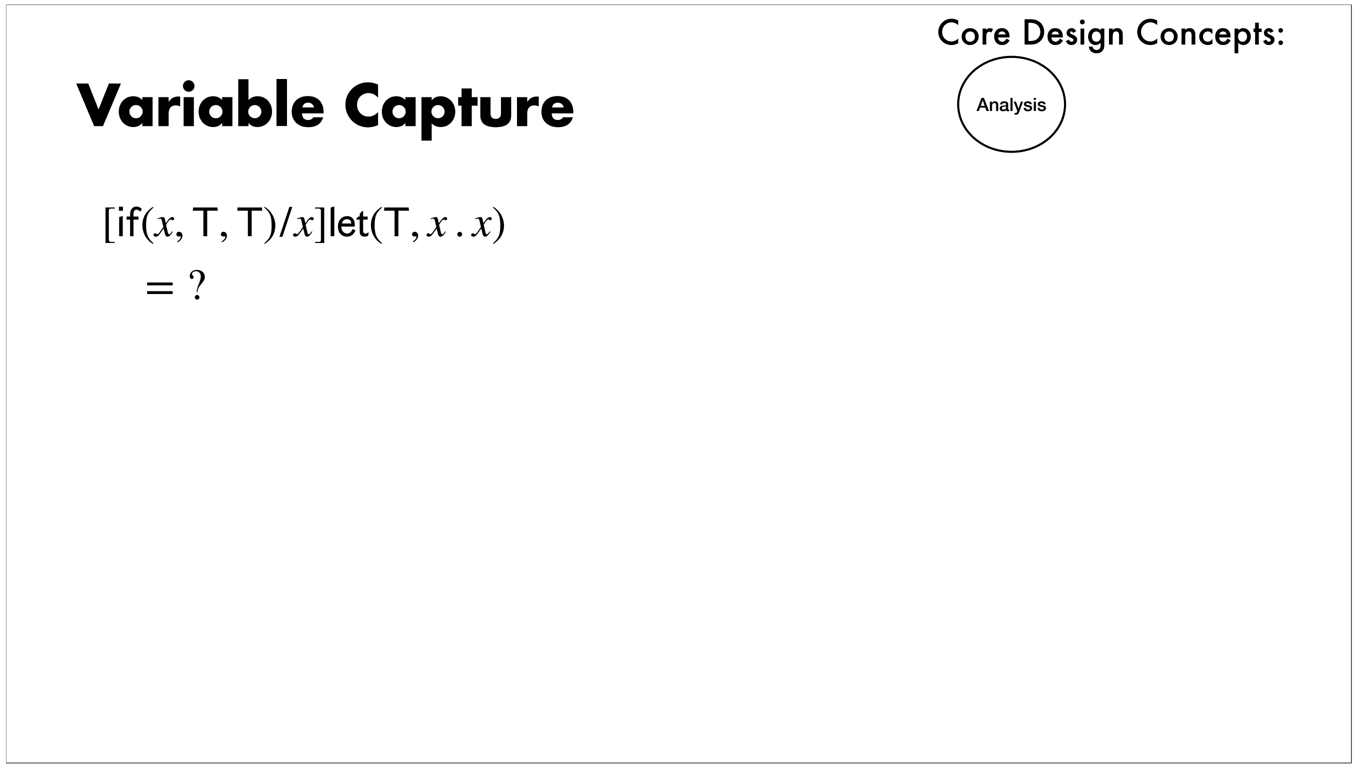
$[t_1/x]x = t_1$

$[t_1/x]\mathsf{if}(t_2, t_3, t_4) = \mathsf{if}([t_1/x]t_2, [t_1/x]t_3, [t_1/x]t_4)$

$[t_1/x]\mathsf{let}(t_2, y\,.\,t_3) = \mathsf{let}([t_1/x]t_2, y\,.\,t_3)$, if $x = y$

$[t_1/x]\mathsf{let}(t_2, y\,.\,t_3) = \mathsf{let}([t_1/x]t_2, y\,.\,[t_1/x]t_3)$, if $x \neq y$

We add that the variable x cannot be the same name as the bound variable y in the case for let-expressions. But, now our example...



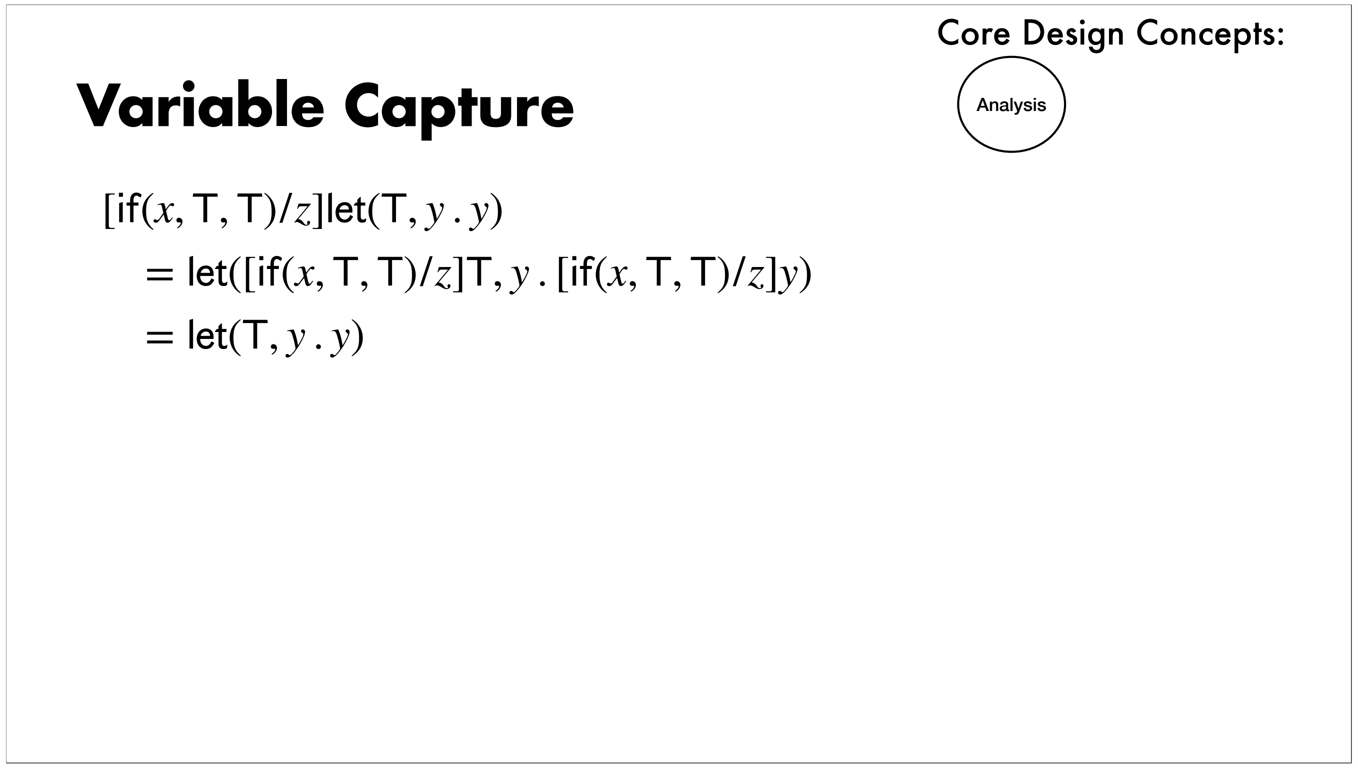
Core Design Concepts:

Analysis

Variable Capture

$$[\mathsf{if}(x, \mathsf{T}, \mathsf{T})/x]\mathsf{let}(\mathsf{T}, x\,.\,x)$$
$$= ?$$

fails, because x matches the bound variable. This implies that substitution is a partial function on arbitrary terms, but the function becomes a total function on alpha-equivalent terms which changes our example to...



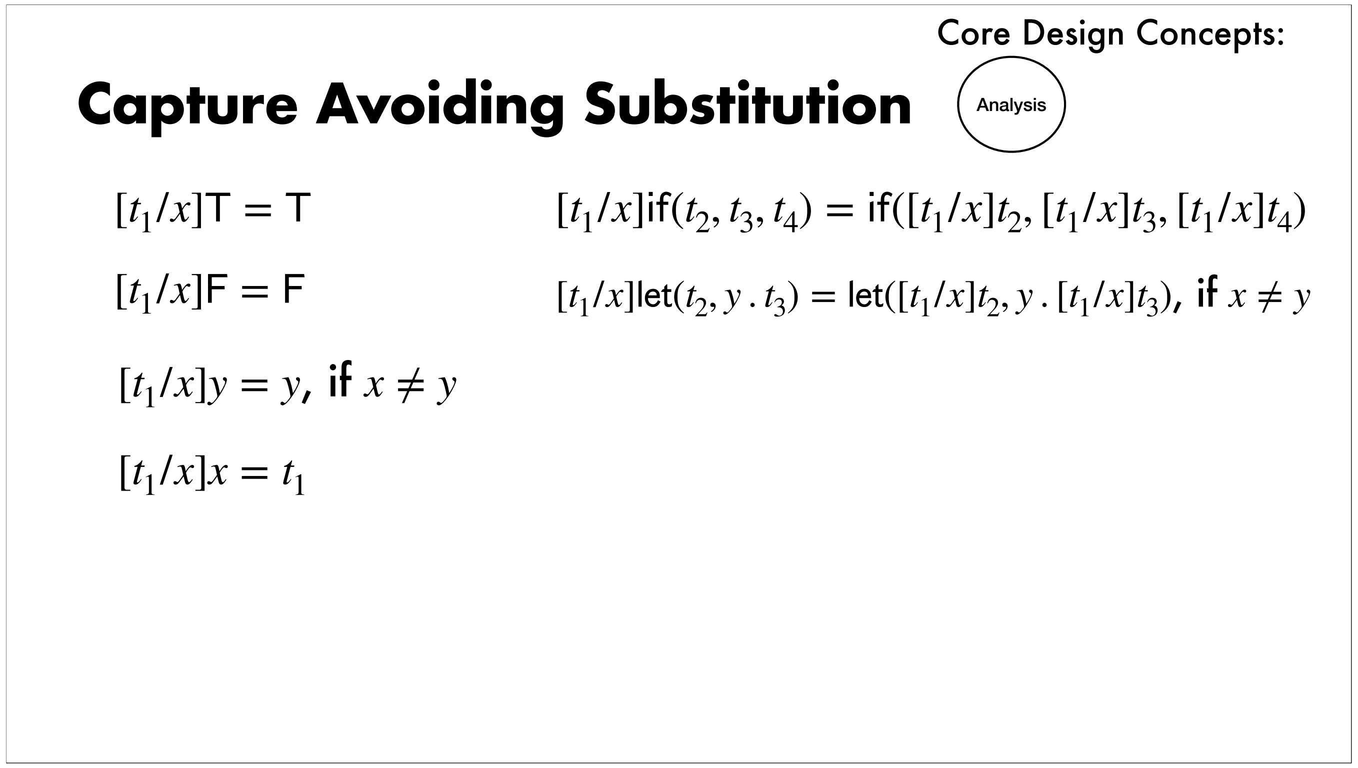
Core Design Concepts:

Analysis

Variable Capture

$$[\mathsf{if}(x, \mathsf{T}, \mathsf{T})/z]\mathsf{let}(\mathsf{T}, y\,.\,y)$$
$$= \mathsf{let}([\mathsf{if}(x, \mathsf{T}, \mathsf{T})/z]\mathsf{T}, y\,.\,[\mathsf{if}(x, \mathsf{T}, \mathsf{T})/z]y)$$
$$= \mathsf{let}(\mathsf{T}, y\,.\,y)$$

now alpha-equivalence allows us to choose unique names in each term. Which produces the desired result...



Core Design Concepts: Analysis

Capture Avoiding Substitution

$[t_1/x]\mathsf{T} = \mathsf{T}$

$[t_1/x]\mathsf{F} = \mathsf{F}$

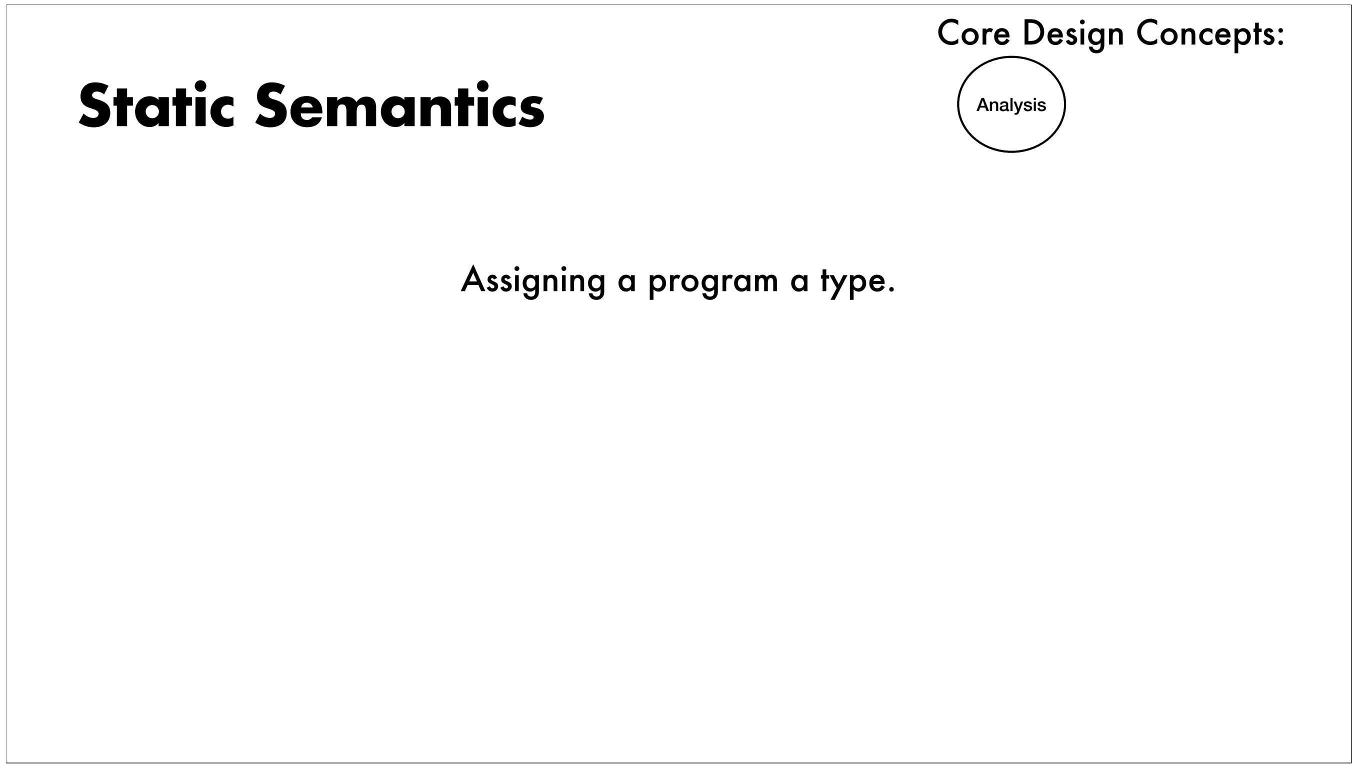
$[t_1/x]y = y$, if $x \neq y$

$[t_1/x]x = t_1$

$[t_1/x]\mathsf{if}(t_2, t_3, t_4) = \mathsf{if}([t_1/x]t_2, [t_1/x]t_3, [t_1/x]t_4)$

$[t_1/x]\mathsf{let}(t_2, y\,.\,t_3) = \mathsf{let}([t_1/x]t_2, y\,.\,[t_1/x]t_3)$, if $x \neq y$

This notion of substitution is called capture avoiding substitution. We now have all that is required to define static and dynamic semantics...



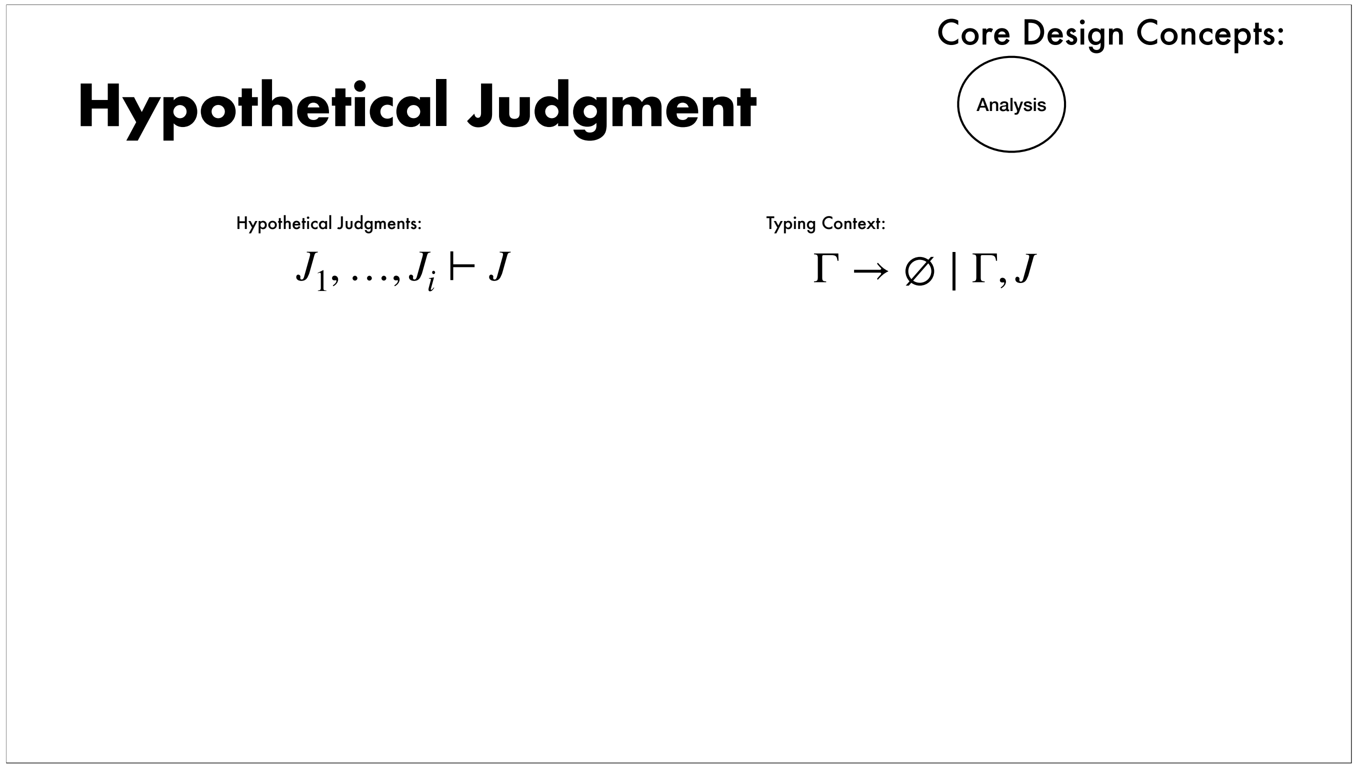
Core Design Concepts:

Analysis

Static Semantics

Assigning a program a type.

The static semantics corresponds to assigning a program a type. The type characterizes the kind of computation the program is. We assign programs types using a typing judgment, but this typing judgment is a bit more general than the judgments we have seen and are called hypothetical judgements..



Hypothetical Judgment

Core Design Concepts:

Analysis

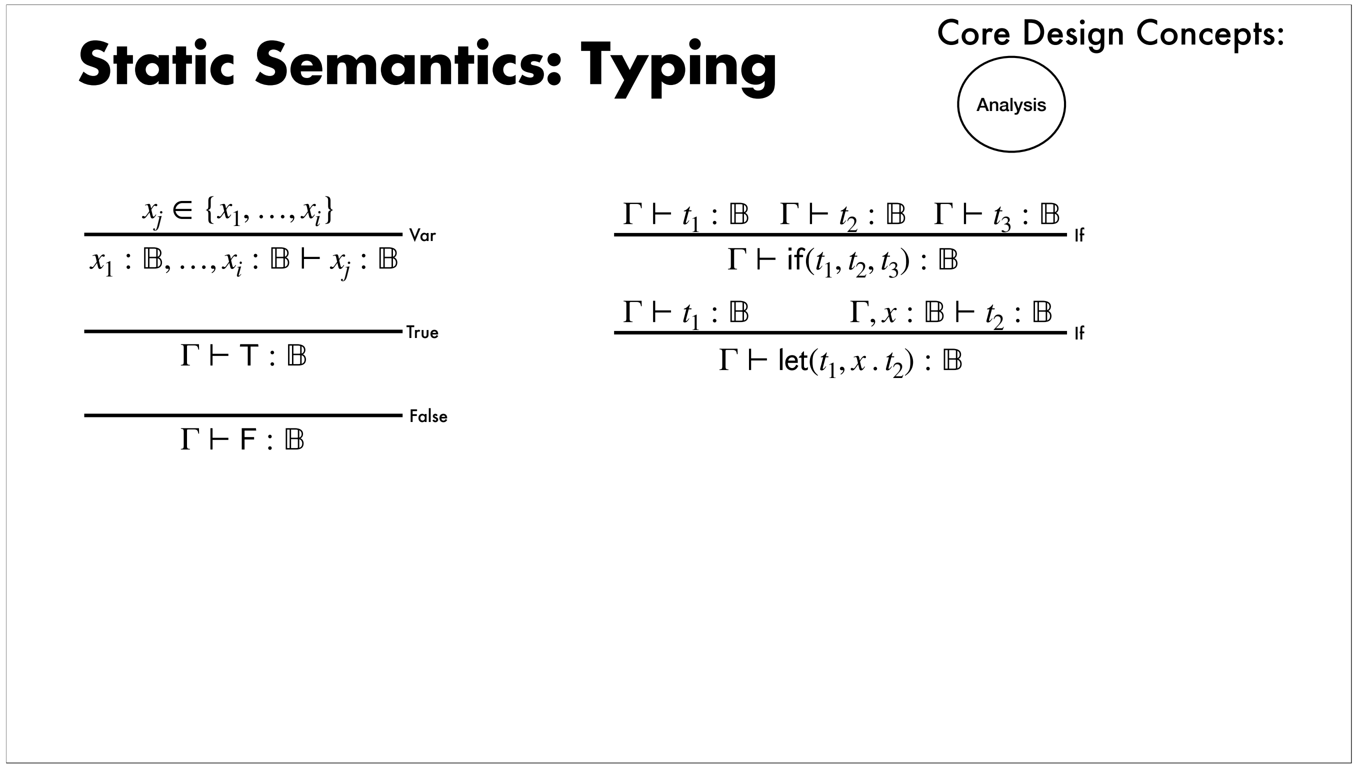
Hypothetical Judgments:

$$J_1, \ldots, J_i \vdash J$$

Typing Context:

$$\Gamma \rightarrow \varnothing \mid \Gamma, J$$

hypothetical judgments have zero or more hypotheses here denoted by J_1 through J_i. Then assuming these hypotheses are true, we verify using the inference rules that J is true. We will denote sequences of hypotheses using typing contexts which are defined on the right. Let's use this new notion to define the typing rules for our small language...



Static Semantics: Typing

Core Design Concepts:

Analysis

$$\frac{x_j \in \{x_1, \ldots, x_i\}}{x_1 : \mathbb{B}, \ldots, x_i : \mathbb{B} \vdash x_j : \mathbb{B}}\ \text{Var}$$

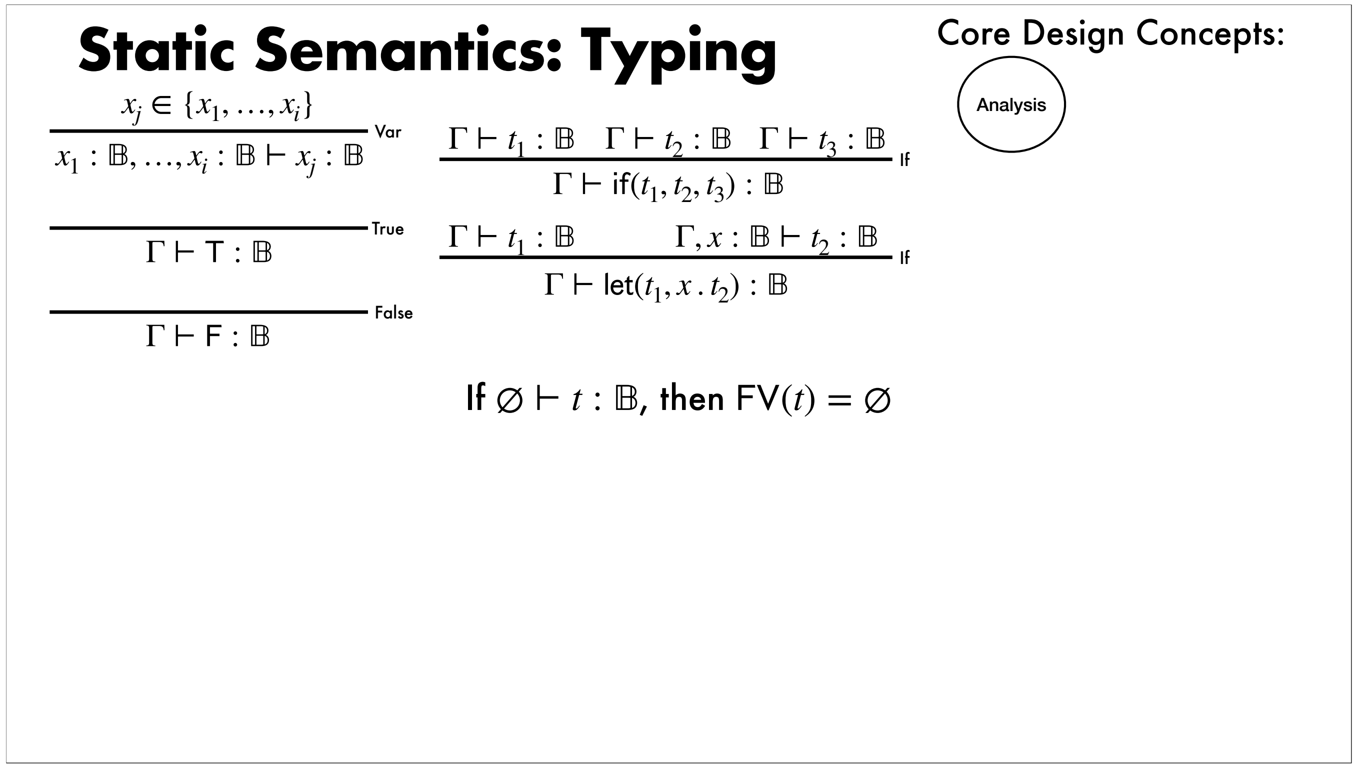
$$\frac{\Gamma \vdash t_1 : \mathbb{B} \quad \Gamma \vdash t_2 : \mathbb{B} \quad \Gamma \vdash t_3 : \mathbb{B}}{\Gamma \vdash \mathsf{if}(t_1, t_2, t_3) : \mathbb{B}}\ \text{If}$$

$$\frac{}{\Gamma \vdash \mathsf{T} : \mathbb{B}}\ \text{True}$$

$$\frac{\Gamma \vdash t_1 : \mathbb{B} \qquad \Gamma, x : \mathbb{B} \vdash t_2 : \mathbb{B}}{\Gamma \vdash \mathsf{let}(t_1, x . t_2) : \mathbb{B}}\ \text{If}$$

$$\frac{}{\Gamma \vdash \mathsf{F} : \mathbb{B}}\ \text{False}$$

This language only has a single type called Boolean. There is a typing rule for each piece of abstract syntax. A variable has type bool only if it is a member of the context. It is left implicit that j is in the range from 1 to i. True and False are pretty easy they are both values of type Bool. Then if is of type Bool if each one of its pieces is of type bool. Finally, let has type bool only if t1 has type bool, and t2 has type bool, but we have to add the bound variable, x, to the context. The static semantics enforces properties on programs, and not every program will have a type. The typing given here enforces only one property...



Static Semantics: Typing

Core Design Concepts:

Analysis

$$\frac{x_j \in \{x_1, \ldots, x_i\}}{x_1 : \mathbb{B}, \ldots, x_i : \mathbb{B} \vdash x_j : \mathbb{B}}\ \text{Var}$$

$$\frac{}{\Gamma \vdash \mathsf{T} : \mathbb{B}}\ \text{True}$$

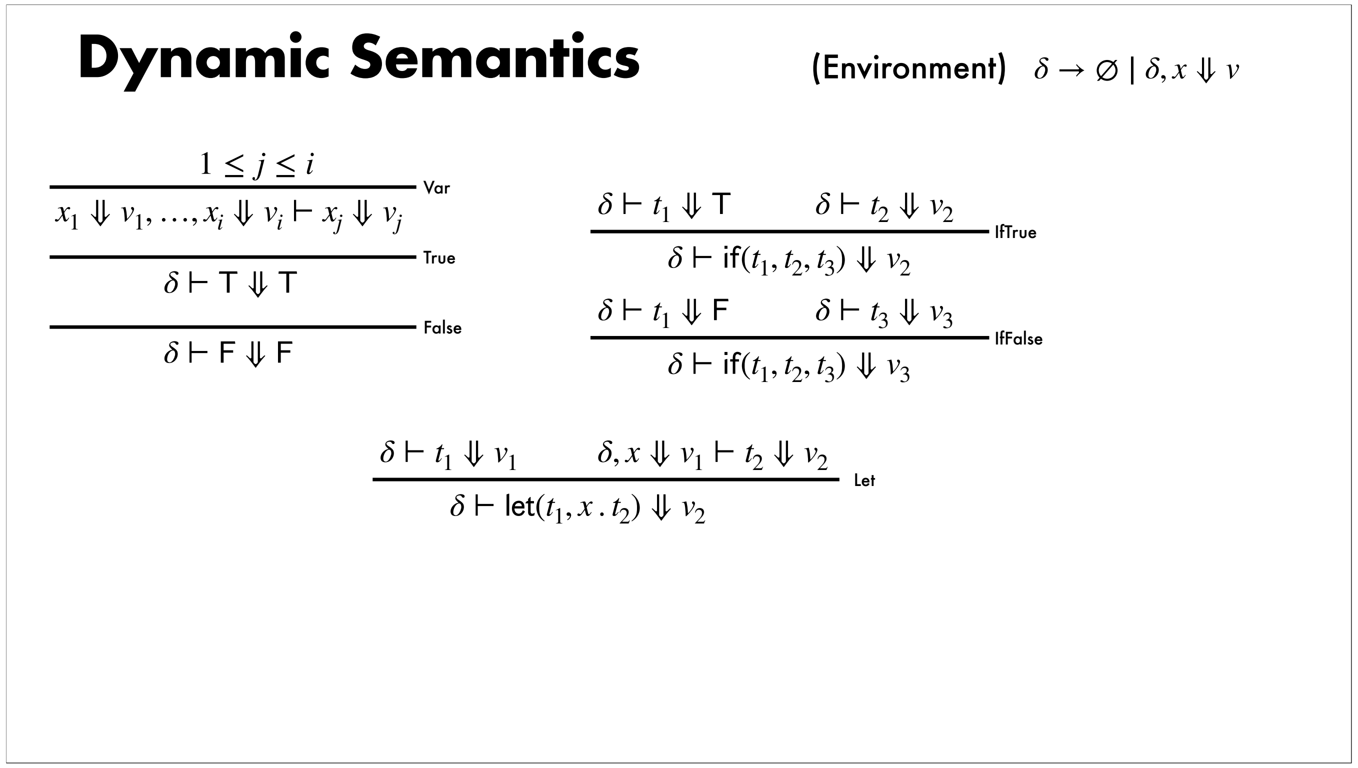
$$\frac{}{\Gamma \vdash \mathsf{F} : \mathbb{B}}\ \text{False}$$

$$\frac{\Gamma \vdash t_1 : \mathbb{B} \quad \Gamma \vdash t_2 : \mathbb{B} \quad \Gamma \vdash t_3 : \mathbb{B}}{\Gamma \vdash \mathsf{if}(t_1, t_2, t_3) : \mathbb{B}}\ \text{If}$$

$$\frac{\Gamma \vdash t_1 : \mathbb{B} \qquad \Gamma, x : \mathbb{B} \vdash t_2 : \mathbb{B}}{\Gamma \vdash \mathsf{let}(t_1, x \,.\, t_2) : \mathbb{B}}\ \text{If}$$

If $\varnothing \vdash t : \mathbb{B}$, then $\mathsf{FV}(t) = \varnothing$

that a program typeable in the empty context has no free variables. We call such programs closed programs. The programs programmers write are always closed. Closed typeable programs are the most important set of programs. Let's now consider the dynamic semantics...



Dynamic Semantics

(Environment) $\delta \to \varnothing \mid \delta, x \Downarrow v$

$$\frac{1 \le j \le i}{x_1 \Downarrow v_1, \ldots, x_i \Downarrow v_i \vdash x_j \Downarrow v_j}\ \text{Var}$$

$$\frac{}{\delta \vdash \mathsf{T} \Downarrow \mathsf{T}}\ \text{True}$$

$$\frac{}{\delta \vdash \mathsf{F} \Downarrow \mathsf{F}}\ \text{False}$$

$$\frac{\delta \vdash t_1 \Downarrow \mathsf{T} \qquad \delta \vdash t_2 \Downarrow v_2}{\delta \vdash \mathsf{if}(t_1, t_2, t_3) \Downarrow v_2}\ \text{IfTrue}$$

$$\frac{\delta \vdash t_1 \Downarrow \mathsf{F} \qquad \delta \vdash t_3 \Downarrow v_3}{\delta \vdash \mathsf{if}(t_1, t_2, t_3) \Downarrow v_3}\ \text{IfFalse}$$

$$\frac{\delta \vdash t_1 \Downarrow v_1 \qquad \delta, x \Downarrow v_1 \vdash t_2 \Downarrow v_2}{\delta \vdash \mathsf{let}(t_1, x \,.\, t_2) \Downarrow v_2}\ \text{Let}$$

The dynamic semantics corresponds to evaluation of programs to a value. We use a notion of an environment denoted by delta to collect the values that will replace the free variables in a program. The variable rule is much like the one for typing, we can see that the True and False rules show that T and F are both values, that is, they compute to themselves. We have two if rules, one when t1 is true and and one when t2 is false. Then it evaluates to the value of the corresponding branch. Finally, the let rule evaluates t1 to a value v1, and then adds to the environment that the bound variable x evaluates to t1's value, v1, and then evaluates t2 to v2 in the extended environment. Finally, the entire let evaluates then to v2. We will discuss this and do lots of examples in class.