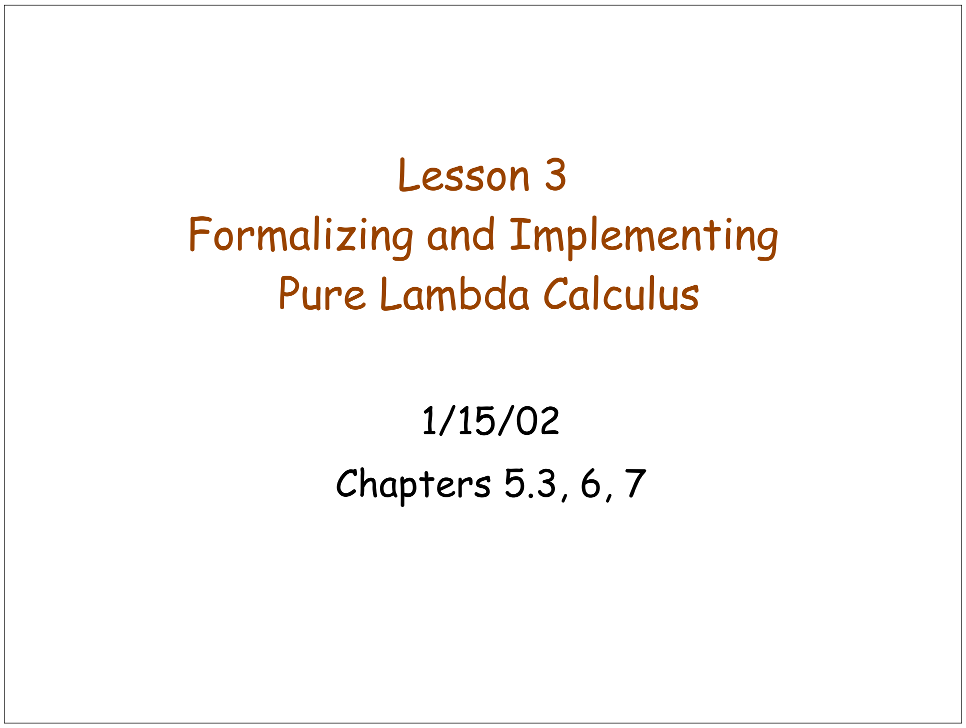
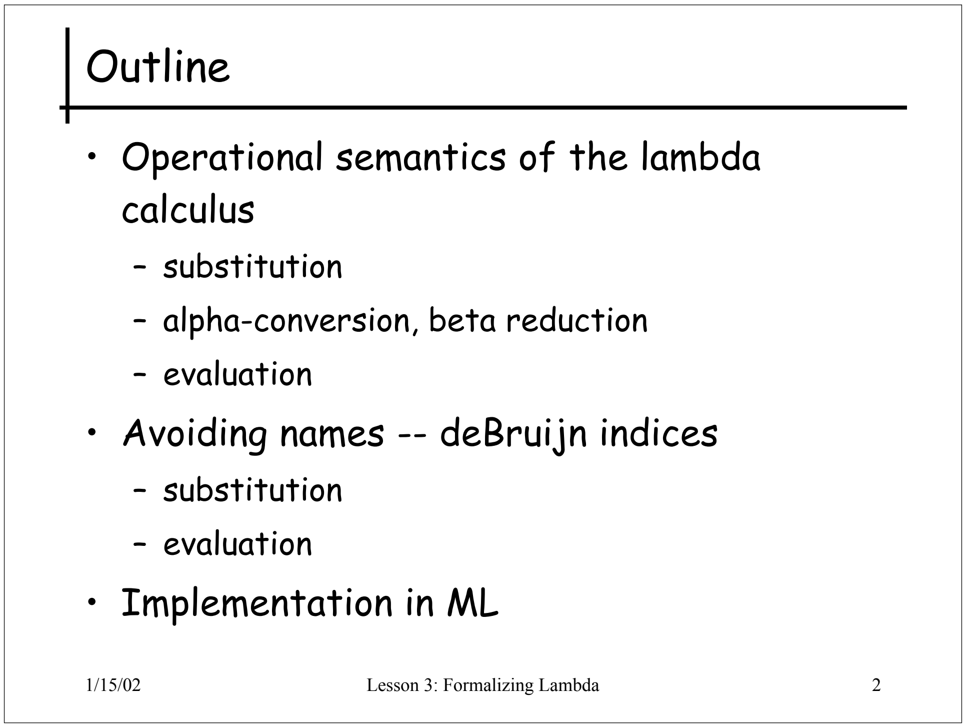


Lesson 3
Formalizing and Implementing Pure Lambda Calculus

1/15/02

Chapters 5.3, 6, 7



Outline

- Operational semantics of the lambda calculus
  - substitution
  - alpha-conversion, beta reduction
  - evaluation
- Avoiding names -- deBruijn indices
  - substitution
  - evaluation
- Implementation in ML

1/15/02 Lesson 3: Formalizing Lambda 2

Abstract Syntax

- $\mathcal{V}$ is a countable set of variables
- $\mathcal{T}$ is the set of terms defined by

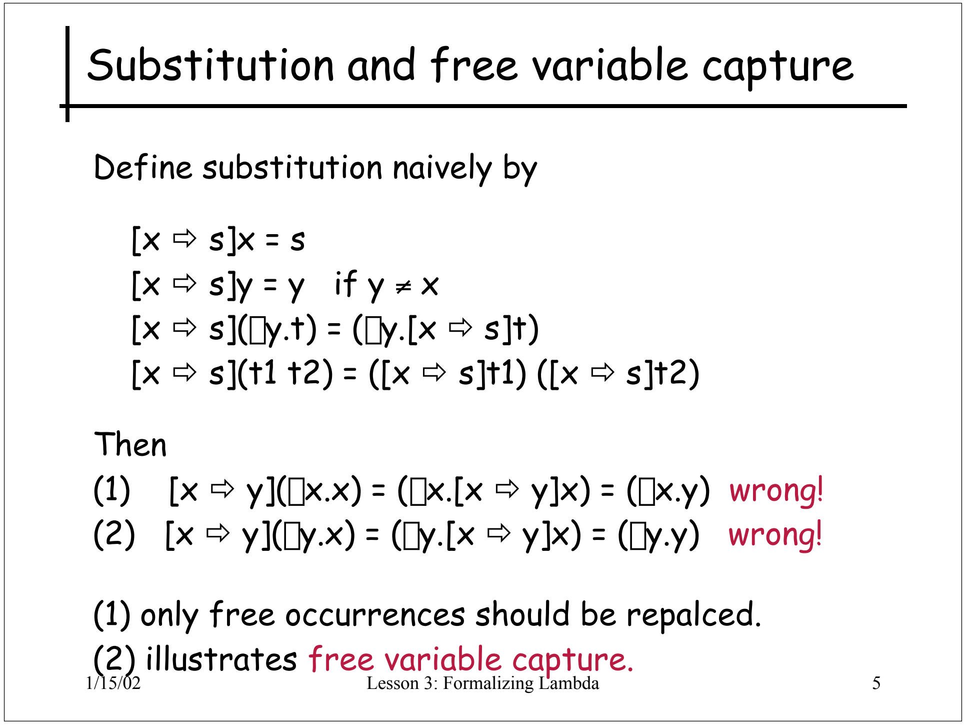
$$
\begin{array}{lll}
t ::= & x & (x \in \mathcal{V}) \\
 & \mid\ \lambda x.t & (x \in \mathcal{V}) \\
 & \mid\ t\ t &
\end{array}
$$

Free variables

The set of free variables of a term is defined by

$FV(x) = \{x\}$
$FV(\lambda x.t) = FV(t) \setminus \{x\}$
$FV(t1\ t2) = FV(t1) \cup FV(t2)$

E.g. $FV(\lambda x.\ y(\lambda y.\ xyu)) = \{y,u\}$



Substitution and free variable capture

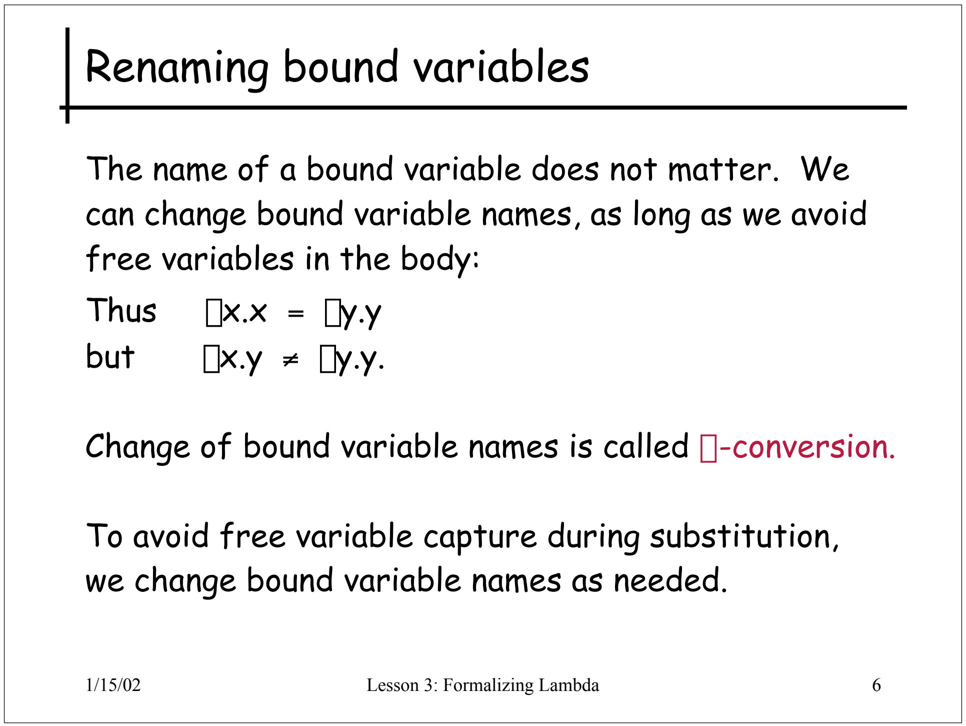
Define substitution naively by

[x ⇨ s]x = s
[x ⇨ s]y = y   if y ≠ x
[x ⇨ s](□y.t) = (□y.[x ⇨ s]t)
[x ⇨ s](t1 t2) = ([x ⇨ s]t1) ([x ⇨ s]t2)

Then
(1)   [x ⇨ y](□x.x) = (□x.[x ⇨ y]x) = (□x.y) wrong!
(2)   [x ⇨ y](□y.x) = (□y.[x ⇨ y]x) = (□y.y)   wrong!

(1) only free occurrences should be repalced.
(2) illustrates free variable capture.

1/15/02   Lesson 3: Formalizing Lambda   5



Renaming bound variables

The name of a bound variable does not matter. We can change bound variable names, as long as we avoid free variables in the body:

Thus   □x.x = □y.y
but   □x.y ≠ □y.y.

Change of bound variable names is called □-conversion.

To avoid free variable capture during substitution, we change bound variable names as needed.

1/15/02   Lesson 3: Formalizing Lambda   6

Substitution refined

Define substitution

$[x \Rightarrow s]x = s$

$[x \Rightarrow s]y = y$ *if* $y \neq x$

$[x \Rightarrow s](\lambda y.t) = (\lambda y.[x \Rightarrow s]t)$ *if* $y \neq x$ *and* $y \notin FV(s)$

$[x \Rightarrow s](t1\ t2) = ([x \Rightarrow s]t1)\ ([x \Rightarrow s]t2)$

When applying the rule for $[x \Rightarrow s](\lambda y.t)$, we change the bound variable y if necessary so that the side conditions are satisfied.

Substitution refined (2)

The rule

$[x \Rightarrow s](\lambda y.t) = (\lambda y.[x \Rightarrow s]t)$ *if* $y \neq x$ *and* $y \notin FV(s)$

could be replaced by

$[x \Rightarrow s](\lambda y.t) = (\lambda z.[x \Rightarrow s][y \Rightarrow z]t)$
where $z \notin FV(t)$ and $z \notin FV(s)$

Note that $(\lambda x.t)$ contains no free occurrences of x, so
$[x \Rightarrow s](\lambda x.t) = \lambda x.t$

Operational semantics (call by value)

Syntax:

t ::=		*Terms*
	x	$(x \in \mathcal{V})$
	| λx.t	$(x \in \mathcal{V})$
	| t t	

v ::= λx.t *Values*

We could also regard variables as values:

v ::= x | λx.t

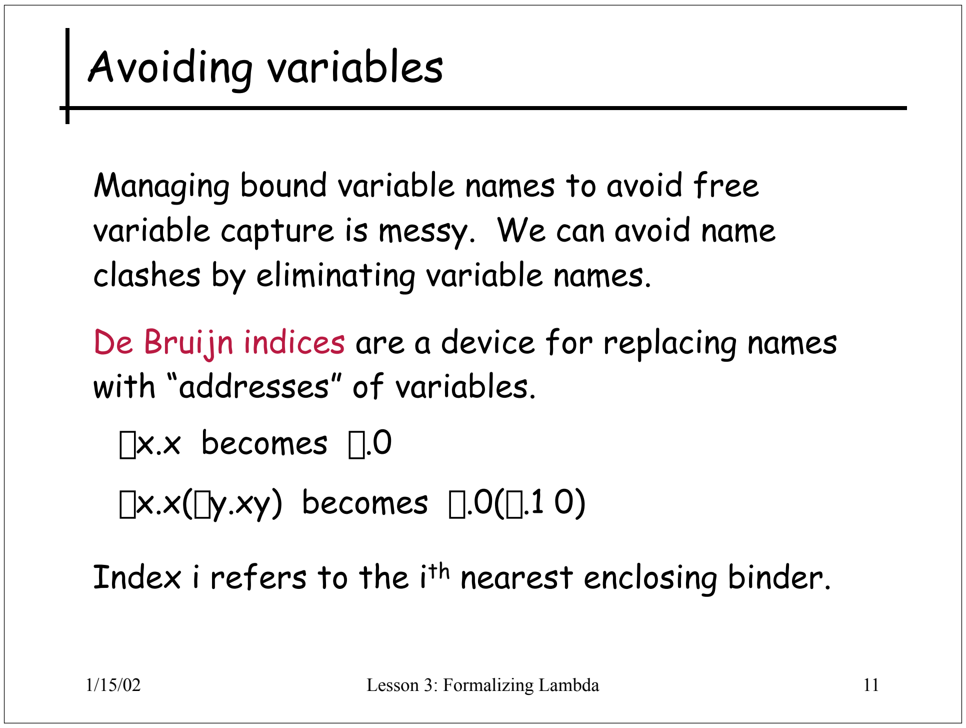
Operational semantics: rules

$(\lambda x.t1)\ v2 \rightarrow [x \mapsto v2]\ t1$

$$\frac{t1 \rightarrow t1'}{t1\ t2 \rightarrow t1'\ t2}$$

$$\frac{t2 \rightarrow t2'}{v1\ t2 \rightarrow v1\ t2'}$$

- evaluate function before argument
- evaluate argument before applying function



Avoiding variables

Managing bound variable names to avoid free variable capture is messy. We can avoid name clashes by eliminating variable names.

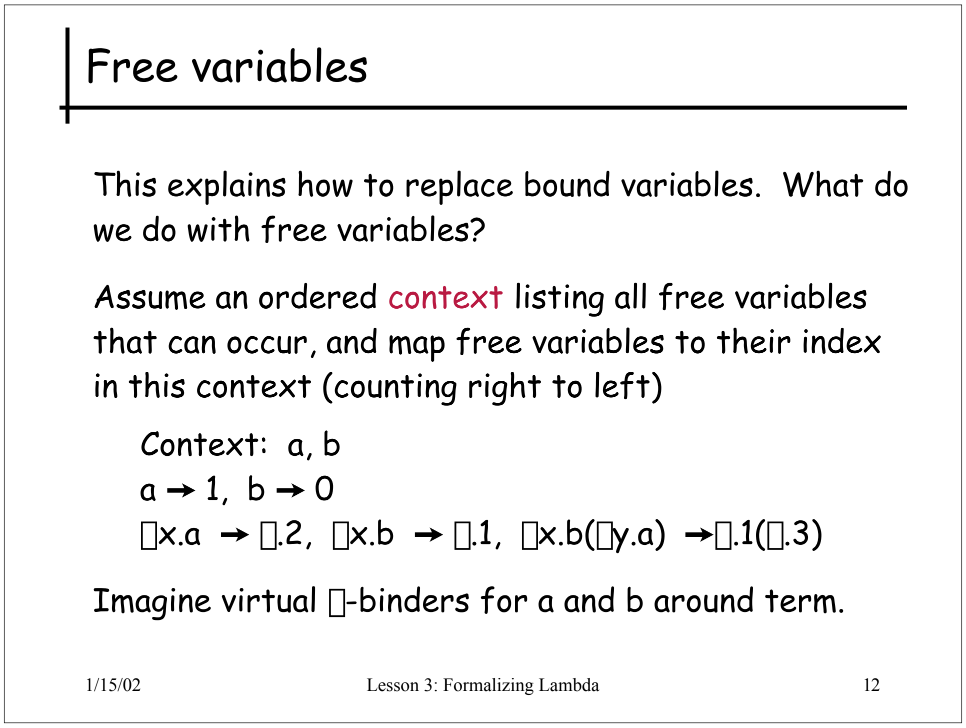
De Bruijn indices are a device for replacing names with "addresses" of variables.

$\lambda x.x$ becomes $\lambda.0$

$\lambda x.x(\lambda y.xy)$ becomes $\lambda.0(\lambda.1\ 0)$

Index i refers to the i^{th} nearest enclosing binder.

1/15/02 Lesson 3: Formalizing Lambda 11



Free variables

This explains how to replace bound variables. What do we do with free variables?

Assume an ordered context listing all free variables that can occur, and map free variables to their index in this context (counting right to left)

Context: a, b

$a \to 1,\ b \to 0$

$\lambda x.a \to \lambda.2,\ \lambda x.b \to \lambda.1,\ \lambda x.b(\lambda y.a) \to \lambda.1(\lambda.3)$

Imagine virtual λ-binders for a and b around term.

1/15/02 Lesson 3: Formalizing Lambda 12

Substitution

When substituting into a lambda term, the indices have to be adjusted:

[x ⇨ z] (λy.x)   in context x,y,z

[1 ⇨ 0] (λ.2) = (λ.[2 ⇨ 1] 2) = (λ.1)

```
shift(d,c) (k) = k if k < c
                 k+d if k >= c
shift(d,c) (λ.t) = (λ.shift(d,c+1)(t))
shift(d,c) (t1 t2) = (shift(d,c) (t1)) (shift(d,c) (t2))
```

Substitution

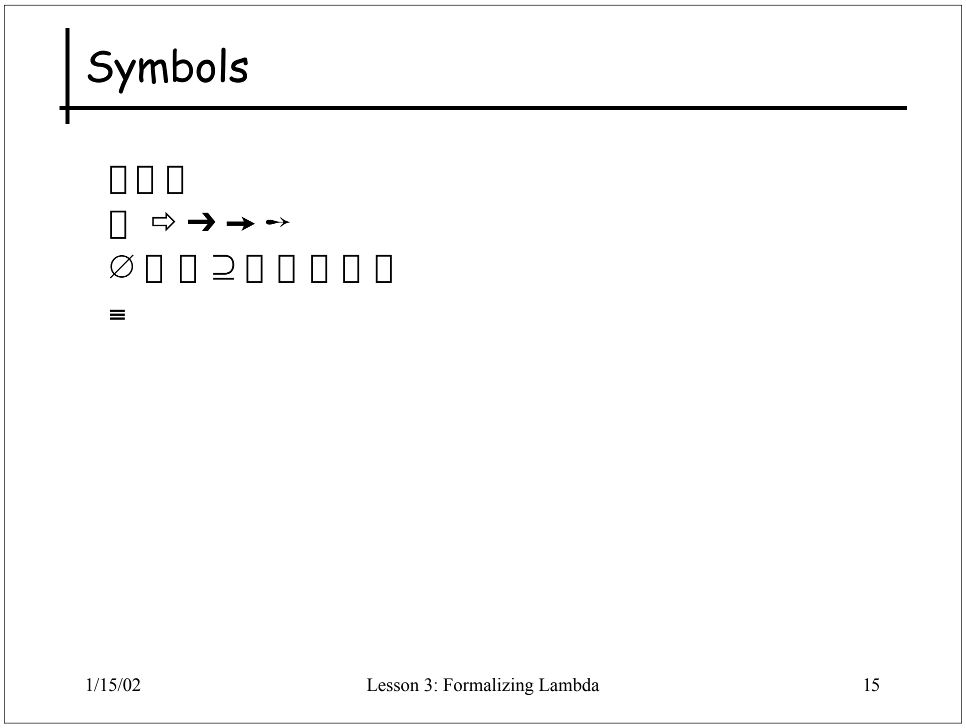
```
[j ⇨ s] k  =  s if k = j
              k otherwise
```

[j ⇨ s] (λ.t) = λ.[j+1 ⇨ shift(1,0)s] t

[j ⇨ s] (t1 t2) = ([j ⇨ s] t1) ([j ⇨ s] t2)

Beta-reduction

(λ.t) v → shift(-1,0)([0 ⇨ shift(1,0)(v)] t)



Symbols

□ □ □

□ ⇨ ➔ ➙ ➻

∅ □ □ ⊇ □ □ □ □ □

≡