Capture-Avoiding Substitution

Reminder: The Problem

 Our current version of substitution can turns free identifiers into bound identifiers

```
(subst {with {y 10} z}

'z

{fun {x} {+ x y}})

⇒ {with {y 10} {fun {x} {+ x y}}}
```

- The y in the function body was free, is now bound
- We can this process capture
- This shouldn't happen

Capture-Avoiding Substitution

- Solution: a new version of substitution that does not capture
- Strategy: look before we leap
 - As we substitute, rename binding and bound identifiers to use names that we know can't cause collisions

Capture-Avoiding Substitution: An Example

```
(subst {with {y 10} {+ y z}}

'z {fun {x} {+ x y}})
```

- We found a binding: the with binds y
- Let's rename y to something new

- That's equivalent; we renamed consistently
- And w is not free in either the expression we're substituting, or the expression we're substituting in
- So no risk of conflict!

Capture-Avoiding Substitution: An Example

```
⇒
(subst {with {w 10} {+ w z}}

'z {fun {x} {+ x y}})

⇒

{with {w 10} {+ w {fun {x} {+ x y}}}}
```

- And now we're done
- No capture; y was free, and it still is

Capture-Avoiding Substitution: The Rules

```
(subst | x | x | e |)
(subst | x | y | e |)
(subst | {e1 e2}
  { (subst | e1 | x | e |)
     (subst e2
(subst | {fun {x} e1} | x e |)
   {fun {x} e1}
```

Capture-Avoiding Substitution: The Rules

```
(subst {fun {x} e1} y e)
\Rightarrow \{fun \{w\} \}
(subst (subst e1 x w)
y e)\}
• where w is free in both {fun {x} e1} and e
```

Why do we care?

- For implementing an interpreter? No big deal
 - Only a problem when programs have free variables
 - And deferred substitution is usually better anyway
- But substitution has many other uses!
 - Compiler optimization
 - Polymorphic type systems (generics)
 - Proofs about languages
 - In such cases, it's important to get substitution right
- Comes up in subsequent classes
 - Jesse's statics of PLs (type systems)
 - Christos's dynamics of PLs (semantics)