The Substitution Model

Principles of Programming Languages

CAS CS 320

Lecture 16

Practice Problem

```
<prog> ::= { <stmt> ; }
<stmt> ::= rot90 | refX | refY
```

$$\frac{}{(s,\ \texttt{rot90;}\ P) \longrightarrow (s\ \texttt{rotated 90 deg. clockwise,}\ P)}$$

$$\frac{}{(s,\ \texttt{refX;}\ P) \longrightarrow (s\ \texttt{reflected across x-axis,}\ P)}$$

$$\frac{}{(s,\ \texttt{refY;}\ P) \longrightarrow (s\ \texttt{reflected across x-axis,}\ P)}$$

What does ($\triangle$, `rot90; refY; rot90; refX;`) *evaluate to? Give a sequence of single step reductions (you do not need to give the full multi-step derivation)*

Answer

```
<prog> ::= { <stmt> ; }
<stmt> ::= rot90 | refX | refY
```

```
(△, rot90; refY; rot90; refX;)
```

Outline

Look formally at the lambda calculus and its semantics

Discuss substitution and the pitfalls to avoid

Demo an implementation of the lambda calculus.

Learning Objectives

- Write a derivation in the CBV and CBV small-step semantics for the lambda calculus
- Write a derivation in the CBV and CBN big-step semantics for the lambda calculus
- Describe the difference between CBV and CBV
- Write an expression which terminates when using CBN but not CBV
- Write and expression which takes much longer to compute when using CBN as opposed to CBV
- Describe what it means for a variable to be captured
- Perform capture-avoiding substitution on paper, and in code

The Lambda Calculus

High-Level View

```
(fun x -> x x)(fun x -> x x)
```

lambda term called Ω

The **lambda calculus** is the *simplest functional programming language.*

<u>It only has:</u>

» variables
» anonymous functions
» function application

It's **untyped**, so *anything* can be applied to *anything*

Demo (OCaml and Python)

History

The lambda calculus was introduced by **Alonzo Church** in the 1930s

It was kind of an accident (not exactly, there's a lot to this story)

Church was trying to give a *foundation of mathematics*, did not succeed, and extracted from that work the lambda calculus

The lambda calculus **as powerful** as every model of computation there is (Turing Machines, Register Machines, etc.)

Syntax

```
<expr> ::= fun <var> -> <expr>
         | <var>
         | <expr> <expr>
         | ( <expr> )
<var>  ::= a | b | ... | y | z
```

Again, we only have

» variables
» anonymous functions
» function application

We assume application is **left-associative**

This presentation is ambiguous. Why?

Syntax (Slightly Better)

```
<expr>  ::= fun <var> -> <expr>
          | <expr2> <expr2>*
<expr2> ::= <var>
          | ( <expr> )
<var>   ::= a | b | ... | y | z
```

In this grammar we can only use variables or functions in parentheses in applications

Syntax (Mathematical)

```
<expr> ::= λ<var>.<expr>
         | <var>
         | <expr><expr>
```

In mathematical settings, we use more compact syntax

Parentheses and variables are often implicit in the presentation

Recall: Semantics and Values

```
<val> ::= λ<var>.<expr>
```

When we specify a semantics, we have to specify our **values**, i.e., when are we *done* computing

In arithmetic, values are: **numbers**

In the lambda calculus, values are: **functions**

Small-Step Semantics

$$\text{(1)}\ \frac{e_1 \longrightarrow e_1'}{e_1 e_2 \longrightarrow e_1' e_2} \qquad \text{(2)}\ \frac{e_2 \longrightarrow e_2'}{(\lambda x \,.\, e_1) e_2 \longrightarrow (\lambda x \,.\, e_1) e_2'}$$

$$\text{(3)}\ \frac{}{(\lambda x \,.\, e)(\lambda y \,.\, e') \longrightarrow [(\lambda y \,.\, e')/x]e}$$

1. We can reduce the LHS of an application
2. We can reduce the RHS of an application *if the LHS is already a function*
3. We can apply a function to another function by **substitution.** This is also called ***β*-reduction**

Small-Step Semantics (Another Form)

$$(1)\,\frac{e_1 \longrightarrow e_1'}{e_1 e_2 \longrightarrow e_1' e_2} \qquad (2)\,\frac{}{(\lambda x\,.\,e)e' \longrightarrow [e'/x]e}$$

1. We can reduce the LHS of an application

2. We can apply a function to *any* expression via substitution

Recall: Stuck Terms

$$(\lambda x\,.\,yx)(\lambda x\,.\,x)$$

Recall that a **stuck term** is an expression which cannot be further reduced, but is not a value.

Based on our operational semantics, it's possible for the above expression to reduce to a value

This is because there is a variable which which is not "bound"

$$\frac{e_1 \longrightarrow e_1'}{e_1e_2 \longrightarrow e_1'e_2} \qquad \frac{}{(\lambda x\,.\,e)e' \longrightarrow [e'/x]e}$$

```
<val> ::= λ<var>.<expr>
```

Non-Termination

$$(\lambda x\,.\,xx)(\lambda x\,.\,xx)$$

`lambda term called Ω`

Unlike with arithmetic, it's now possible to define expressions which **do not terminate**

These expression do not have values, but also don't get stuck

$$\frac{e_1 \longrightarrow e_1'}{e_1 e_2 \longrightarrow e_1' e_2} \qquad \frac{}{(\lambda x\,.\,e)e' \longrightarrow [e'/x]e}$$

Big-Step Semantics

$$\texttt{(1)}\frac{}{\lambda x\,.\,e \Downarrow \lambda x\,.\,e}$$

$$\texttt{(2)}\frac{e_1 \Downarrow \lambda x\,.\,e \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}$$

1. A function evaluates to itself

2. If e_1 evaluates to the function $\lambda x\,.\,e$ and e_2 evaluates to the value v_2 and e with v_2 substituted for x evaluates to v, then the application $e_1 e_2$ evaluates to v

Big-Step Semantics (Another Form)

$$\frac{}{\lambda x \,.\, e \Downarrow \lambda x \,.\, e}$$

$$\frac{e_1 \Downarrow \lambda x \,.\, e \qquad [e_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}$$

These are the same rules as before except we're not required to evaluate e_2 first

Practice Problem

$$(\lambda x\,.\,\lambda y\,.\,y)((\lambda z\,.\,z)(\lambda q\,.\,q)) \Downarrow \lambda y\,.\,y$$

Give a derivation of the above judgment in both versions of the big-step semantics.

$$\frac{}{\lambda x\,.\,e \Downarrow \lambda x\,.\,e} \qquad\qquad \frac{}{\lambda x\,.\,e \Downarrow \lambda x\,.\,e}$$

$$\frac{e_1 \Downarrow \lambda x\,.\,e \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v} \qquad\qquad \frac{e_1 \Downarrow \lambda x\,.\,e \qquad [e_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}$$

Answer

$$(\lambda x\,.\,\lambda y\,.\,y)((\lambda z\,.\,z)(\lambda q\,.\,q)) \Downarrow \lambda y\,.\,y$$

Call-by-Value vs. Call-by-Name

$$\frac{e_1 \Downarrow \lambda x\,.\,e_1' \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e_1' \Downarrow v}{e_1 e_2 \Downarrow v}\ \texttt{(CBV)}$$

$$\frac{e_1 \Downarrow \lambda x\,.\,e_1' \qquad [e_2/x]e_1' \Downarrow v}{e_1 e_2 \Downarrow v}\ \texttt{(CBN)}$$

The two versions of semantics we've given correspond to **call-by-value** (CBV) and **call-by-name** (CBN). These are **evaluation strategies** for functional languages

CBV: evaluate the argument of a function *before* substituting it in the function

CBN: substitute the expression *directly* into the function

Benefits of CBV

$$\frac{}{\lambda x\,.\,e \Downarrow \lambda x\,.\,e} \qquad \frac{e_1 \Downarrow \lambda x\,.\,e_1' \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e_1' \Downarrow v}{e_1 e_2 \Downarrow v}$$

$$(\lambda x\,.\,x + x + x + x)e$$

If we compute the value of an argument before substituting it into the expression, we only have to compute the expression *once.*

This is good if the variable appears in *lots* of places in our function.

This is also called **eager**, or **applicative**, or **strict** evaluation.

This is what OCaml does.

Benefits of CBN

$$(\lambda x \,.\, \lambda y \,.\, x)e_1e_2$$

If a variables doesn't appear in our function, then the argument is *not evaluated at all*

This is good if it's possible that an argument is only seldomly used, it will only be computed when it is used. For example, something which is computed in a branch of an if-expression that is almost never reached.

Aside. It's possible to simulate CBN in CBV. Think about it for a bit, ask me after if you're interested

$$\frac{}{\lambda x \,.\, e \Downarrow \lambda x \,.\, e} \qquad \frac{e_1 \Downarrow \lambda x \,.\, e_1' \qquad [e_2/x]e_1' \Downarrow v}{e_1e_2 \Downarrow v}$$

Side Effects

```
let f x = x + x in
let y =
  let _ = print_int 2 in
  2
in f y
```

What does this program print? **It depends on if we're using CBN or CBV evaluation**

Definition. *(informal)* A **side effect** refers to something that happens during the evaluation of a program that is not a part of the formal semantics, e.g., printing to the console

We won't implement languages with side effects (not strictly true), but it's important to note that different evaluation strategies will yield different side-effectful behavior

Evaluation Strategies

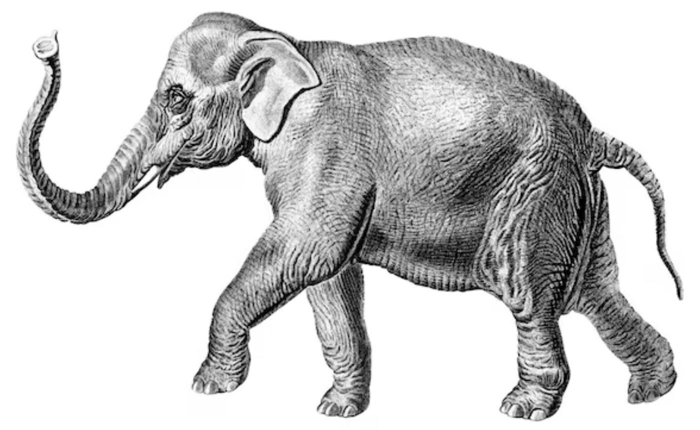
There are *a lot* more evaluation strategies, all of which optimize *something*

Haskell uses **lazy** evaluation also called **call-by-need**

In languages with pointers (gross) we also often have the option to use **call-by-reference** evaluation or **call-by-sharing**

We will exclusively implement **call-by-value** *(because, again, this is what OCaml does)*

Substitution



$$\frac{e_1 \Downarrow \lambda x \,.\, e \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}$$

It's time to get more formal about **substitution**

We've been able to get by on our intuitions for a while, but our intuitions won't help us *implement* substitution (which is <u>difficult</u>)

We need to understand why...

Capture-Avoiding Substitution

Notation

$$[y/x](\lambda x \, . \, y)$$

We write $[v/x]e$ to mean e with v substituted in for x

Informally. *Replace every instance of x with v*

Already things start to break down with this informal definition, e.g., consider the above substitution

Our definition of substitution shouldn't change the <u>underlying behavior of the function</u>

Aside. There are about 60 different notations for substitution, we'll stick with this one...

α-Equivalence

OCaml

```
let x = 2 in x + 1
```

$$=_\alpha$$

```
let z = 2 in z + 1
```

$$\lambda x\,.\,\lambda y\,.\,x =_\alpha \lambda v\,.\,\lambda w\,.\,v$$

λ-calculus

The **principle of name irrelevance** says that any two programs that are the same up to "renaming of variables" should behave exactly the same way

We say that a variable x is **bound** is an expression if it appears in the expression as $(\ldots \lambda x\,.\,e \ldots)$

Definition. *(informal)* Two expressions are **α-equivalent** is they are identical up to the naming of bound variables.

Substitution should preserve α-equivalence

Naive Definition

$$[v/y]x = \begin{cases} v & x = y \\ x & \texttt{else} \end{cases} \tag{1}$$

$$[v/y](\lambda x \,.\, e) = \lambda x \,.\, [v/y]e \tag{2}$$

$$[v/y](e_1 e_2) = ([v/y]e_1)([v/y]e_2) \tag{3}$$

1. Replace every *y* with *v*, leave other variables

2. Replace *y* with *v* in the body of a function

3. Replace *y* with *v* in both subexpressions of an application

Seems pretty reasonable...

This is an example of an **inductive definition**. These kinds of definitions have very natural implementations in functional programming languages.

Problem Case I

$$[v/y]x = \begin{cases} v & x = y \\ x & \texttt{else} \end{cases}$$

$$[v/y](\lambda x \, . \, e) = \lambda x \, . \, [v/y]e$$

$$[v/y](e_1 e_2) = ([v/y]e_1)([v/y]e_2)$$

$$[y/x](\lambda x \, . \, x)$$

What's the problem with this substitution?

It stems from the fact that we shouldn't be allowed to substitute x if it's the argument of a function

Otherwise we might <u>change the behavior</u> of a function

Less Naive Definition

$$[v/y]x = \begin{cases} v & x = y \\ x & \texttt{else} \end{cases}$$

$$[v/y](\lambda x . e) = \begin{cases} \lambda x . e & x = y \\ \lambda x . [v/y]e & \texttt{else} \end{cases}$$

$$[v/y](e_1 e_2) = ([v/y]e_1)([v/y]e_2)$$

We can handle the problem case directly to our definition

We just do check the bound variable before we substitute in the body of a function

Is there still a problem?

$$[v/y]x = \begin{cases} v & x = y \\ x & \texttt{else} \end{cases}$$

$$[v/y](\lambda x\,.\,e) = \begin{cases} \lambda x\,.\,e & x = y \\ \lambda x\,.\,[v/y]e & \texttt{else} \end{cases}$$

$$[v/y](e_1 e_2) = ([v/y]e_1)([v/y]e_2)$$

Problem Case II

$$[y/x](\lambda y\,.\,x)$$

What's the problem with this substitution?

We're not replacing a bound variable, but we *are* substituting an expression that has variables which *become* bound

The variable y is said to be **captured** in this (incorrect) substitution

Free and Bound Variables

$$FV(x) = \{x\} \tag{1}$$

$$FV(\lambda x \,.\, e) = FV(e)\backslash\{x\} \tag{2}$$

$$FV(e_1 e_2) = FV(e_1) \cup FV(e_2) \tag{3}$$

Definition. *(informal)* A variable x is **free** in e if it does not appear **bound** by a λ (but what does "bound" mean...)

1. x is free in x

2. x is free in $\lambda y \,.\, e$ if it is free in e and $x \neq y$

3. x is free in $e_1 e_2$ if x is free in e_1 or e_2

Definition. A variable x is **free** in e if $x \in FV(e)$ as above

Note that this is another **inductive definition**

Even Less Naive Definition

$$[v/y]x = \begin{cases} v & x = y \\ x & \texttt{else} \end{cases}$$

$$[v/y](\lambda x\,.\,e) = \begin{cases} \lambda x\,.\,e & x = y \\ \lambda z\,.\,[w/z][z/x]e & x \in FV(v) \\ \lambda x\,.\,[v/y]e & \texttt{else} \end{cases}$$

$$[v/y](e_1 e_2) = ([v/y]e_1)([v/y]e_2)$$

Since we're interested in α-equivalence, we can first *replace* the bound variable and *substitute* it in the body of the function

Is there still a problem?

(this is pretty much there, this definition isn't so much *wrong* as it is **underspecified**)

Problem Case III

$$[x/y](\lambda x\,.\,xyz)$$

What's the problem with this substitution?

Nothing exactly, but **we have to be careful about which variable to replace the bound variable *x* with**

If we choose z, then we capture a *different* variable!

$$FV(x) = \{x\}$$
$$FV(\lambda x\,.\,e) = FV(e)\backslash\{x\}$$
$$FV(e_1e_2) = FV(e_1) \cup FV(e_2)$$

$$[v/y]x = \begin{cases} v & x = y \\ x & \texttt{else} \end{cases}$$

$$[v/y](\lambda x\,.\,e) = \begin{cases} \lambda x\,.\,e & x = y \\ \lambda z\,.\,[w/z][z/x]e & x \in FV(v) \\ \lambda x\,.\,[v/y]e & \texttt{else} \end{cases}$$

$$[v/y](e_1e_2) = ([v/y]e_1)([v/y]e_2)$$

"Correct" Definition

$$[v/y]x = \begin{cases} v & x = y \\ x & \texttt{else} \end{cases}$$

$$[v/y](\lambda x\,.\,e) = \begin{cases} \lambda x\,.\,e & x = y \\ \lambda z\,.\,[w/z][z/x]e & x \in FV(v), z \notin FV(e) \\ \lambda x\,.\,[v/y]e & \texttt{else} \end{cases}$$

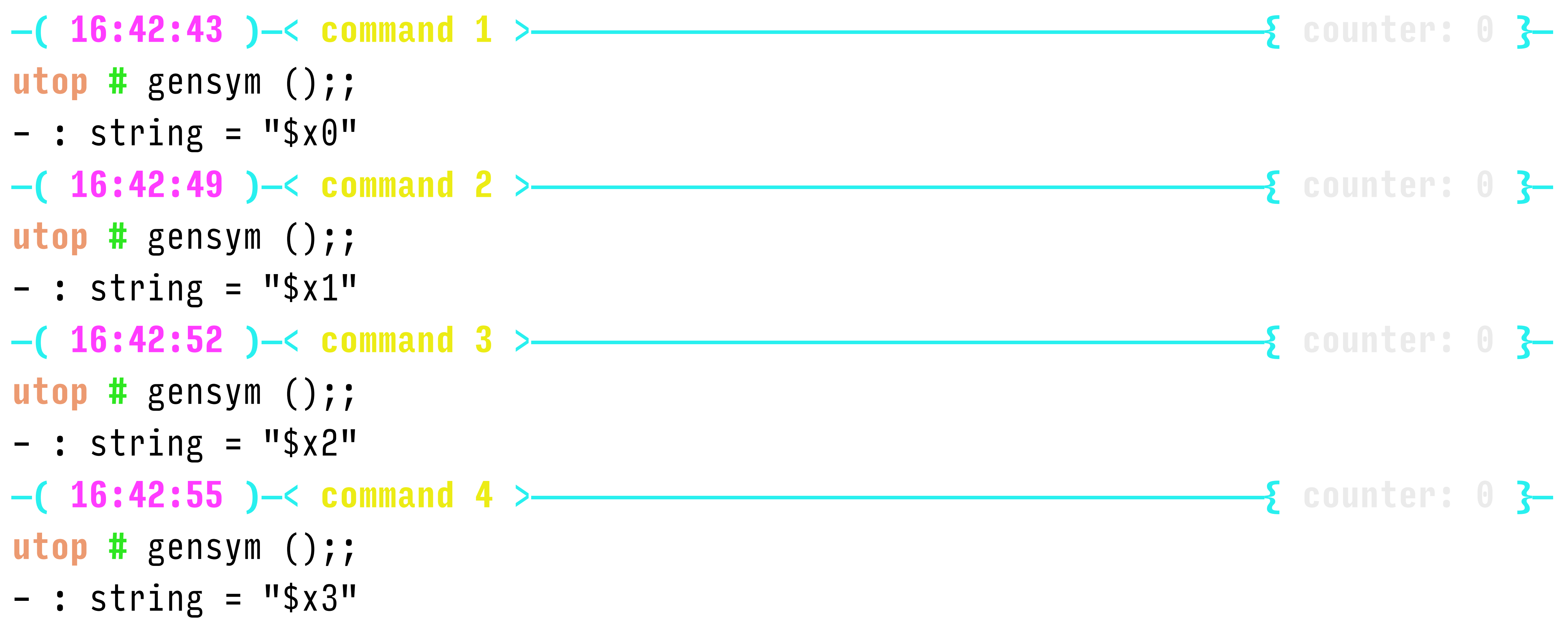
$$[v/y](e_1 e_2) = ([v/y]e_1)([v/y]e_2)$$

Finally a definition, that works. Sort of...

The only problem with this definition is that it now poses an implementation issue. **How do we come up with z?**

In mathematics, we can say it's **always possible** to come up with a variable z, but when we're implementing a programming language, we need an *actual* procedure

gensym



```
─( 16:42:43 )─< command 1 >──────────────────────────────────{ counter: 0 }─
utop # gensym ();;
- : string = "$x0"
─( 16:42:49 )─< command 2 >──────────────────────────────────{ counter: 0 }─
utop # gensym ();;
- : string = "$x1"
─( 16:42:52 )─< command 3 >──────────────────────────────────{ counter: 0 }─
utop # gensym ();;
- : string = "$x2"
─( 16:42:55 )─< command 4 >──────────────────────────────────{ counter: 0 }─
utop # gensym ();;
- : string = "$x3"
```

What we'll do: generate a new string on the fly which cannot coincide with user defined variables

But wait a second: isn't this something like a counter, which requires *state*? Didn't you say there's *no state* pure functional programming languages? Didn't you say this *isn't possible* in OCaml?

Yes. We lied. Sorry

This is the *only* time we will use something stateful in OCaml

Practice Problem

$$[(\lambda x \,.\, xy)/z](\lambda x \,.\, \lambda y \,.\, yz)$$

Perform the following substitution, avoiding variable capture.

$$[v/y]x = \begin{cases} v & x = y \\ x & \texttt{else} \end{cases}$$

$$[v/y](\lambda x \,.\, e) = \begin{cases} \lambda x \,.\, e & x = y \\ \lambda z \,.\, [w/z][z/x]e & x \in FV(v) \\ \lambda x \,.\, [v/y]e & \texttt{else} \end{cases}$$

$$[v/y](e_1 e_2) = ([v/y]e_1)([v/y]e_2)$$

Answer

$$[(\lambda x\,.\,xy)/z](\lambda x\,.\,\lambda y\,.\,yz)$$

Demo (The Lambda Calculus)