CSEP505: Programming Languages
Lecture 4: Untyped Lambda-Calculus, Formal Operational Semantics, ...

Dan Grossman
Autumn 2016
Where are we

- To talk about functions more precisely, we need to define them as carefully as we did IMP’s constructs.

- First try adding functions & local variables to IMP “on the cheap”
  – It didn’t work [see last week]

- Now back up and define a language with *nothing* but functions
  – [started last week]
  – And then *encode* everything else
Review

- **Cannot** properly model local scope via a global heap of integers
  - Functions are not syntactic sugar for assignments to globals
- So let’s build a model of this key concept
  - Or just borrow one from 1930s logic
- And for now, drop mutation, conditionals, and loops
  - We won’t need them!
- The Lambda calculus in BNF

  **Expressions:** \( e ::= x | \lambda x. e | e e \)

  **Values:** \( v ::= \lambda x. e \)
Expressions: \[ e ::= x \mid \lambda x. \ e \mid e \ e \]

Values: \[ v ::= \lambda x. \ e \]

A program is an \( e \). To call a function:

*substitute the argument for the bound variable*

That’s the key operation we were missing

Example substitutions:

\[(\lambda x. \ x) (\lambda y. \ y) \to \lambda y. \ y\]

\[(\lambda x. \ \lambda y. \ y \ x) (\lambda z. \ z) \to \lambda y. \ y (\lambda z. \ z)\]

\[(\lambda x. \ x \ x) (\lambda x. \ x \ x) \to (\lambda x. \ x \ x) (\lambda x. \ x \ x)\]
Why substitution [More review]

• After substitution, the bound variable is *gone*
  – So clearly its name didn’t matter
  – That was our problem before

• Given substitution we can define a little programming language
  – (correct & precise definition is subtle; we’ll come back to it)
  – This microscopic PL turns out to be Turing-complete
Full large-step interpreter

type exp = Var of string
    | Lam of string*exp
    | Apply of exp * exp

exception BadExp

let subst e1_with e2_for x = ...(*to be discussed*)

let rec interp_large e =
    match e with
    var _ -> raise BadExp(* unbound variable *)
    | lam _ -> e (* functions are values *)
    | apply(e1,e2) ->
        let v1 = interp_large e1 in
        let v2 = interp_large e2 in
        match v1 with
        lam(x,e3) -> interp_large (subst e3 v2 x)
        | _ -> failwith "impossible" (* why? * )
Interpreter summarized

- Evaluation produces a value $\text{Lam}(x, e3)$ if it terminates

- Evaluate application (call) by
  1. Evaluate left
  2. Evaluate right
  3. Substitute result of (2) in body of result of (1)
  4. Evaluate result of (3)

A different semantics has a different \textit{evaluation strategy}:
  1. Evaluate left
  2. Substitute right in body of result of (1)
  3. Evaluate result of (2)
Another interpreter

type exp = Var of string
  | Lam of string*exp
  | Apply of exp * exp
exception BadExp
let subst e1_with e2_for x = ...(*to be discussed*)
let rec interp_large2 e =
  match e with
  Var _ -> raise BadExp(*unbound variable*)
| Lam _ -> e (*functions are values*)
| Apply(e1,e2) ->
  let v1 = interp_large2 e1 in
  (* we used to evaluate e2 to v2 here *)
  match v1 with
  Lam(x,e3) -> interp_large2 (subst e3 e2 x)
  | _ -> failwith "impossible" (* why? *)
What have we done

- Syntax and two large-step semantics for the *untyped lambda calculus*
  - First was “call by value”
  - Second was “call by name”

- Real implementations don’t use substitution
  - They do something *equivalent*

- Amazing (?) fact:
  - If call-by-value terminates, then call-by-name terminates
  - (They might both not terminate)
What will we do

• Go back to math metalanguage
  – Notes on concrete syntax (relates to OCaml)
  – Define semantics with inference rules
• Lambda encodings (show our language is mighty)
• Define substitution precisely
• Environments

Next time??
• Small-step
• Play with continuations (“very fancy” language feature)
Syntax notes

• When in doubt, put in parentheses

• Math (and OCaml) resolve ambiguities as follows:

1. \( \lambda x. e_1 e_2 \) is \( (\lambda x. e_1) e_2 \)

   – *not* \( (\lambda x. e_1) e_2 \)

General rule: Function body “starts at the dot” and “ends at the first unmatched right paren”

Example:

\( (\lambda x. y (\lambda z. z) w) q \)
Syntax notes

2. \( e_1 \ e_2 \ e_3 \) is \((e_1 \ e_2) \ e_3\)
   
   – \( \text{not } e_1 \ (e_2 \ e_3)\)

General rule: Application “associates to the left”

So \( e_1 \ e_2 \ e_3 \ e_4 \) is \(((e_1 \ e_2) \ e_3) \ e_4\)
It’s just syntax

- As in IMP, we really care about abstract syntax
  - Here, internal tree nodes labeled “λ” or “apply” (i.e., “call”)

- Previous 2 rules just reduce parens when writing trees as strings

- Rules may seem strange, but they’re the most convenient
  - Based on 70 years experience
  - Especially with currying
What will we do

• Go back to math metalanguage
  – Notes on concrete syntax (relates to OCaml)
  – Define semantics with inference rules
• Lambda encodings (show our language is mighty)
• Define substitution precisely
• Environments

Next time??
• Small-step
• Play with continuations (“very fancy” language feature)
Inference rules

- A metalanguage for operational semantics
  - Plus: more concise (& readable?) than OCaml
  - Plus: useful for reading research papers
  - Plus: natural support for *nondeterminism*
    - *Definition* allowing observably different implementations
  - Minus: less tool support than OCaml (no compiler)
  - Minus: one more thing to learn
  - Minus: painful in Powerpoint
Informal idea

Want to know:

what values (0, 1, many?) an expression can evaluate to

So define a relation over pairs \((e, v)\):

– Where \(e\) is an expression and \(v\) is a value
– Just a subset of all pairs of expressions and values

If the language is deterministic, this relation turns out to be a function from expressions to values

Metalanguage supports defining relations

– Then prove a relation is a function (if it is)
Making up metasyntax

Rather than write \((e,v)\), we’ll write \(e \downarrow v\).

– It’s just metasyntax (!)
  • Could use \(\text{interp}(e,v)\) or « \(v \bowtie e \) » if you prefer
– Our metasyntax follows PL convention
  • Colors are not conventional (slides: \textcolor{green}{green} = metasyntax)
– And distinguish it from other relations

First step: define the form (arity and metasyntax) of your relation(s):

This is called a \textit{judgment}
What we need to define

So we can write $e \Downarrow v$ for any $e$ and $v$

- But we want such a thing to be “true” to mean $e$ can evaluate to $v$ and “false” to mean it cannot

Examples (before the definition):

- $(\lambda x. \lambda y. y \ x)\ ((\lambda z. z)\ (\lambda z. z)) \Downarrow \lambda y. y\ (\lambda z. z)$ in the relation
- $(\lambda x. \lambda y. y \ x)\ ((\lambda z. z)\ (\lambda z. z)) \Downarrow \lambda z. z$ not in the relation
- $\lambda y. y \Downarrow \lambda y. y$ in the relation
- $(\lambda y. y)\ (\lambda x. \lambda y. y \ x) \Downarrow \lambda y. y$ not in the relation
- $(\lambda x. x\ x)\ (\lambda x. x\ x) \Downarrow \lambda y. y$ not in the relation
- $(\lambda x. x\ x)\ (\lambda x. x\ x) \Downarrow (\lambda x. x\ x)\ (\lambda x. x\ x)$ metasyntactically bogus
Inference rules

\[ \lambda x. e \downarrow \lambda x. e \]

\[ e_1 \downarrow \lambda x. e_3 \quad e_2 \downarrow v_2 \quad e_3[v_2/x] = e_4 \quad e_4 \downarrow v \]

\[ e_1 e_2 \downarrow v \]

- Using definition of a set of 4-tuples for substitution
  - (exp * value * variable * exp)
  - Will define substitution later
### Inference rules

- **Rule top: hypotheses** (0 or more)
- **Rule bottom: conclusion**
- **Metasemantics:** If all hypotheses hold, then conclusion holds

Example:

\[
\lambda x. e \downarrow \lambda x. e
\]

\[
e_1 \downarrow \lambda x. e_3 \quad e_2 \downarrow v_2 \quad e_3[v_2/x] = e_4 \quad e_4 \downarrow v
\]

\[
\Rightarrow e_1 e_2 \downarrow v
\]
Rule schemas

\[ e_1 \downarrow \lambda x. \ e_3 \quad e_2 \downarrow v_2 \quad e_3\{v_2/x\} = e_4 \quad e_4 \downarrow v \]

\[ \text{[app]} \]

\[ e_1 \ e_2 \downarrow v \]

- Each rule is a schema you “instantiate consistently”
- So [app] “works” “for all” x, e_1, e_2, e_3, e_4, v_2, and v
- But “each” e_1 has to be the “same” expression
  - Replace \textit{metavariabes} with appropriate terms
  - Deep connection to logic programming (e.g., Prolog)
Instantiating rules

\[ \lambda x. e \Downarrow \lambda x. e \]

- Two example legitimate instantiations:
  - \( \lambda z. z \Downarrow \lambda z. z \)
    - \( x \) instantiated with \( z \), \( e \) instantiated with \( z \)
  - \( \lambda z. \lambda y. y z \Downarrow \lambda z. \lambda y. y z \)
    - \( x \) instantiated with \( z \), \( e \) instantiated with \( \lambda y. y z \)

- Two example illegitimate instantiations:
  - \( \lambda z. z \Downarrow \lambda y. z \)
  - \( \lambda z. \lambda y. y z \Downarrow \lambda z. \lambda z. Z \)

*Must get your rules “just right” so you don’t allow too much or too little*
Derivations

• Tuple is “in the relation” if there exists a derivation of it
  – An upside-down (or not?!) tree where each node is an instantiation and leaves are axioms (no hypotheses)

• To show $e \Downarrow v$ for some $e$ and $v$, *give a derivation*
  – But we rarely “hand-evaluate” like this
  – We’re just defining a semantics remember

• Let’s work through an example derivation for
  $$(\lambda x. \lambda y. y x) ((\lambda z. z) (\lambda z. z)) \Downarrow \lambda y. y (\lambda z. z)$$
Which relation?

So exactly which relation did we define
   - The pairs at the bottom of finite-height derivations

Note: A derivation tree is like the tree of calls in a large-step interpreter
   - [when relation is a function]
   - Rule being instantiated is branch of the match-expression
   - Instantiation is arguments/results of the recursive call
A couple extremes

- This rules are a **bad idea** because either one adds all pairs to the relation

\[
\begin{array}{c}
  e \downarrow v \\
  e \downarrow v
\end{array}
\]

- This rule is **pointless** because it adds no pairs to the relation

\[
\begin{array}{c}
  e \downarrow v \\
  e \downarrow v
\end{array}
\]
Summary so far

• Define judgment via a collection of inference rules
  – Tuple in the relation ("judgment holds") if a derivation (tree of instantiations ending in axioms) exists

As an interpreter, could be “nondeterministic”:
• Multiple derivations, maybe multiple $\nu$ such that $e \Downarrow \nu$
  – Our example language is deterministic
  – In fact, “syntax directed” (≤1 rule per syntax form)

• Still need rules for $e[\nu/x] = e'$

• Let’s do more judgments (i.e., languages) to get the hang of it…
Call-by-name large-step

\[
\begin{align*}
&e \downarrow_N v \\
&e\{v/x\} = e'
\end{align*}
\]

\[
\begin{align*}
\lambda x. e \downarrow_N \lambda x. e
\end{align*}
\]

\[
\begin{align*}
e1 \downarrow_N \lambda x. e3 & \quad e3\{e2/x\} = e4 & \quad e4 \downarrow_N v \\
\end{align*}
\]

\[
\begin{align*}
e1 \ e2 \downarrow_N v
\end{align*}
\]

- Easier to see the difference than in OCaml
- Formal statement of amazing fact:
  
  For all \( e \), if there exists a \( v \) such that \( e \downarrow v \) then there exists a \( v2 \) such that \( e \downarrow_N v2 \)

  \( (Proof\ is\ non-trivial\ &\ must\ reason\ about\ substitution) \)
IMP

• Two judgments $H; e \Downarrow i$ and $H; s \Downarrow H2$
• Assume $\text{get}(H,x,i)$ and $\text{set}(H,x,i,H2)$ are defined
• Let’s try writing out inference rules for the judgments…
What will we do

- Go back to math metalanguage
  - Notes on concrete syntax (relates to OCaml)
  - Define semantics with inference rules
- Lambda encodings (show our language is mighty)
- Define substitution precisely
- Environments

Next time??
- Small-step
- Play with continuations ("very fancy" language feature)
Encoding motivation

• Fairly crazy: we left out integers, conditionals, data structures, …

• Turns out we’re Turing complete
  – We can encode whatever we need
  – (Just like assembly language can)

• Motivation for encodings
  – Fun and mind-expanding
  – Shows we are not oversimplifying the model
    (“numbers are syntactic sugar”)
  – Can show languages are too expressive
    Example: C++ template instantiation

• Encodings are also just “(re)definition via translation”
Encoding booleans

The “Boolean Abstract Data Type (ADT)”

• There are 2 booleans and 1 conditional expression
  – The conditional takes 3 (curried) arguments
    • If 1st argument is one bool, return 2nd argument
    • If 1st argument is other bool, return 3rd argument

• Any set of 3 expressions meeting this specification is a proper encoding of booleans

• Here is one (of many):
  – “true”  \(\lambda x. \lambda y. x\)
  – “false”  \(\lambda x. \lambda y. y\)
  – “if”  \(\lambda b. \lambda t. \lambda f. b \, t \, f\)
Example

Given our encoding:

- “true” \( \lambda x. \lambda y. x \)
- “false” \( \lambda x. \lambda y. y \)
- “if” \( \lambda b. \lambda t. \lambda f. b \ t \ f \)

We can derive “if” “true” \( v1 \ v2 \downarrow v1 \)

And every “law of booleans” works out
  - And every non-law does not

By the way, this is OOP
But…

- Evaluation order matters!
  - With ↓, our “if” is not YFL’s if

  “if” “true” (λx. x) (λx. x x) (λx. x x) doesn’t terminate
  but

  “if” “true” (λx. x) (λz. (λx. x x) (λx. x x) z) terminates

- Such “thunking” is unnecessary using ↓ₙ
Encoding pairs

- The “Pair ADT”
  - There is 1 constructor and 2 selectors
  - 1\textsuperscript{st} selector returns 1\textsuperscript{st} argument passed to the constructor
  - 2\textsuperscript{nd} selector returns 2\textsuperscript{nd} argument passed to the constructor
- This does the trick:
  - “make\_pair” \( \lambda x. \lambda y. \lambda z. z \times y \)
  - “first” \( \lambda p. p \ (\lambda x. \lambda y. x) \)
  - “second” \( \lambda p. p \ (\lambda x. \lambda y. y) \)

- Example:
  “snd” (“fst” (“make\_pair” (“make\_pair” v1 v2) v3)) \(\downarrow\) v2
Reusing Lambda

- Is it weird that the encodings of Booleans and pairs both used $\lambda x. \lambda y. x$ and $\lambda x. \lambda y. y$ for different purposes?

- Is it weird that the same bit-pattern in binary code can represent an int, a float, an instruction, or a pointer?

- Von Neumann: Bits can represent (all) code and data

- Church (?): Lambdas can represent (all) code and data

- Beware the “Turing tarpit”
Encoding lists

- Why start from scratch? Can build on bools and pairs:
  - “empty-list” is “make_pair” “false” “false”
  - “cons” is $\lambda h. \lambda t. \text{make_pair} \text{ true} \text{ make_pair} h t$
  - “is-empty” is …
  - “head” is …
  - “tail” is …

- Note:
  - Not too far from how lists are implemented
  - Taking “tail” (“tail” “empty”) will produce some lambda
    - Just like, without page-protection hardware, $\text{null}->\text{tail}->\text{tail}$ would produce some bit-pattern
Encoding natural numbers

• Known as “Church numerals”
  – Will skip in the interest of time

• The “natural number” ADT is basically:
  – “zero”
  – “successor” (the add-one function)
  – “plus”
  – “is-equal”

• Encoding is correct if “is-equal” agrees with elementary-school arithmetic

• [Don’t need “full” recursion, but with “full” recursion, can also just do lists of Booleans…]
Recursion

• Can we write *useful* loops? Yes!

To write a recursive function:
• Write a function that takes an \( f \) and call \( f \) in place of recursion:
  – Example (in enriched language):
    \[
    \lambda f. \lambda x. \text{if } x=0 \text{ then } 1 \text{ else } (x \times f(x-1))
    \]
• Then apply “fix” to it to get a recursive function
  “fix” \( \lambda f. \lambda x. \text{if } x=0 \text{ then } 1 \text{ else } (x \times f(x-1)) \)

• Details, especially in CBV are icky; but it’s possible and need be done only once. *For the curious:*
  “fix” is \( \lambda f. (\lambda x.f (\lambda y.x x y)) (\lambda x.f (\lambda y.x x y)) \)
More on “fix”

• “fix” is also known as the Y-combinator

• The informal idea:
  – “fix”(λf.e) becomes something like
    
    \[
    e\{(\text{fix}(\lambda f.e)) / f\}
    \]
  – That’s unrolling the recursion once
  – Further unrollings are delayed (happen as necessary)

• Teaser: Most type systems disallow “fix”
  – So later we’ll add it as a primitive
  – Example: OCaml can never type-check (x x)
What will we do

- Go back to math metalanguage
  - Notes on concrete syntax (relates to OCaml)
  - Define semantics with inference rules
- Lambda encodings (show our language is mighty)
- Define substitution precisely
- Environments

Next time??
- Small-step
- Play with continuations (“very fancy” language feature)
Our goal

Need to define

\[ e_1(e_2/x) = e_3 \]

- Used in \([\text{app}]\) rule
- Informally, “replace occurrences of \(x\) in \(e_1\) with \(e_2\)”
- Shockingly subtle to get right (in theory or programming)

- (Under call-by-value, only need \(e_2\) to be a value, but that doesn’t make it much easier, so define the more general thing.)
Try #1 [WRONG]

\[
e_1\{e_2/x\} = e_3
\]

\[
x\{e/x\} = e \\
y\{e/x\} = y \\
(\lambda y . e_1)\{e_2/x\} = \lambda y . e_3
\]

\[
e_{a\{e_2/x\}} = e_{a'} \\
e_{b\{e_2/x\}} = e_{b'}
\]

\[
(e_a \, e_b)\{e_2/x\} = e_{a'} \, e_{b'}
\]

- Recursively replace every x leaf with e2
- But the rule for substituting into (nested) functions is wrong: If the function’s argument binds the same variable (shadowing), we should not change the function’s body
- Example program: \((\lambda x . \lambda x . x) \, 42\)
Try #2 [WRONG]

\[ e_1[e_2/x] = e_3 \]

\[
\begin{align*}
\text{y \neq x} & \quad e_1[e_2/x] = e_3 \quad \text{y\neq x} \\
\text{x}_{e/x} = e & \quad y_{e/x} = y \\
\text{(\lambda y. e_1)e_2/x} = \lambda y. e_3 & \\
\text{e_a[e_2/x]} = \text{e_a}' & \quad \text{e_b[e_2/x]} = \text{e_b}' \\
(\text{e_a e_b})_{e_2/x} = \text{e_a}' \text{ e_b}' & \quad (\lambda x. e_1)e_2/x} = \lambda x. e_1
\end{align*}
\]

- Recursively replace every x leaf with e2, but respect shadowing
- Still wrong due to capture [actual technical term]:
  - Example: \((\lambda y. e_1){y/x}\)
  - Example \((\lambda y. e_1)((\lambda z. y)/x)\)
  - In general, if “y appears free in e2”
More on capture

• Good news: capture can’t happen under CBV or CBN
  – *If* program starts with no unbound (“free”) variables

• Bad news: Can still result from “inlining”

• Bad news: It’s still “the wrong definition” in general
  – My experience: The nastiest of bugs in language tools
Try #3 [Almost Correct]

• First define an expression’s “free variables” (braces here are set notation)
  – $\text{FV}(x) = \{x\}$
  – $\text{FV}(e_1 e_2) = \text{FV}(e_1) \cup \text{FV}(e_2)$
  – $\text{FV}(\lambda y.e) = \text{FV}(e) - \{y\}$

• Now require “no capture”:

  \[
  e_1{e_2/x} = e_3 \quad y \neq x \quad \text{y not in FV(e2)}
  \]

  \[
  (\lambda y. e_1){e_2/x} = \lambda y. e_3
  \]
Try #3 in Full

\[
e_1[e_2/x] = e_3
\]

- No mistakes with what is here...
- … but only a partial definition
  - What if \( y \) is in the free-variables of \( e_2 \)
Implicit renaming

• But this is a partial definition due to a “syntactic accident”, until…

• We allow “implicit, systematic renaming” of any term
  – In general, we never distinguish terms that differ only in variable names
  – A key language-design principle
  – Actual variable choices just as “ignored” as parens
  – Means rule above can “always apply” with a lambda

• Called “alpha-equivalence”: terms differing only in names of variables are *the same term*
Try #4 [correct]

- [Includes systematic renaming and drops an unneeded rule]

\[ e_1[e_2/x] = e_3 \]

\[
\begin{array}{c}
\text{y} \neq x \\
x[e/x] = e \\
y[e/x] = y
\end{array}
\]

\[
\begin{array}{c}
e_1[e_2/x] = e_3 \\
y \neq x \\
y \text{ not in FV(e_2)} \\
(\lambda y . e_1)e_2/x = \lambda y . e_3
\end{array}
\]

\[
\begin{array}{c}
e_1[e_2/x] = e_3 \\
y \neq x \\
e a[e_2/x] = e a' \\
e b[e_2/x] = e b'
\end{array}
\]

\[
\begin{array}{c}
(\lambda x . e_1)e_2/x = \lambda x . e 1
\end{array}
\]
More explicit approach

• While “everyone in the PL field”:
  – Understands the capture problem
  – Avoids it by saying “implicit systematic renaming” you may find that unsatisfying…
  … especially if you have to implement substitution while avoiding capture

• So this more explicit version also works (“fresh z for y”):

\[
\text{z not in FV}(e_1) \cup \text{FV}(e_2) \cup \{x\} \quad e_1\{z/y\} = e_3 \quad e_3\{e_2/x\} = e_4
\]

\[
(\lambda y. e_1)\{e_2/x\} = \lambda z. e_4
\]

– You have to “find an appropriate z”, but one always exists and __$$tmp$$ appended to a global counter “probably works”
Note on metasyntax

• Substitution often thought of as a metafunction, not a judgment
  – I’ve seen many nondeterministic languages
  – But never a nondeterministic definition of substitution

• So instead of writing:

\[
e_1 \Downarrow \lambda x. e_3 \quad e_2 \Downarrow v_2 \quad e_3[v_2/x] = e_4 \quad e_4 \Downarrow v
\]

\[
\text{[app]}
\]

\[
e_1 \Downarrow v
\]

• Just write:

\[
e_1 \Downarrow \lambda x. e_3 \quad e_2 \Downarrow v_2 \quad e_3[v_2/x] \Downarrow v
\]

\[
\text{[app]}
\]

\[
e_1 \Downarrow v
\]
What will we do

• Go back to math metalanguage
  – Notes on concrete syntax (relates to OCaml)
  – Define semantics with inference rules
• Lambda encodings (show our language is mighty)
• Define substitution precisely
• Environments

Next time??
• Small-step
• Play with continuations (“very fancy” language feature)
Where we’re going

- Done: large-step for untyped lambda-calculus
  - CBV and CBN
  - Note: infinite number of other “reduction strategies”
  - Amazing fact: all equivalent if you ignore termination!

- Now other semantics, all equivalent to CBV:
  - With environments (in OCaml to prep for Homework 3)
  - Basic small-step (easy)
  - Contextual semantics (similar to small-step)
    - Leads to precise definition of continuations
type exp = Var of string
   | Lam of string*exp
   | Apply of exp * exp

exception BadExp

let subst e1_with e2_for x = ...(*to be discussed*)

let rec interp_large e =
    match e with
    Var _ -> raise BadExp(*unbound variable*)
   | Lam _ -> e (*functions are values*)
   | Apply(e1,e2) ->
      let v1 = interp_large e1 in
      let v2 = interp_large e2 in
      match v1 with
      Lam(x,e3) -> interp_large (subst e3 v2 x)
      | _ -> failwith "impossible" (* why? *)
Environments

- Rather than substitute, let’s keep a map from variables to values
  - Called an environment
  - Like IMP’s heap, but immutable and 1 not enough
- So a program “state” is now exp and environment
- A function body is evaluated under the environment where it was defined!
  - Use closures to store the environment
  - See also Lecture 1
No more substitution

define exp = Var of string
  | Lam of string * exp
  | Apply of exp * exp
  | Closure of string * exp * env

and env = (string * exp) list

let rec interp env e =
  match e with
  Var s -> List.assoc s env (* do the lookup *)
  | Lam(s,e2) -> Closure(s,e2,env) (* store env! *)
  | Closure _ -> e (* closures are values *)
  | Apply(e1,e2) ->
    let v1 = interp env e1 in
    let v2 = interp env e2 in
    match v1 with
    Closure(s,e3,env2) -> interp((s,v2)::env2) e3
    _ -> failwith "impossible"
Worth repeating

- A closure is a pair of code and environment
  - Implementing higher-order functions is not magic or run-time code generation
- An okay way to think about OCaml
  - Like thinking about OOP in terms of vtables
- Need not store whole environment of course
  - See Homework 3
What will we do

- Go back to math metalanguage
  - Notes on concrete syntax (relates to OCaml)
  - Define semantics with inference rules
- Lambda encodings (show our language is mighty)
- Define substitution precisely
  - And revisit function equivalences
- Environments

Next time??
- Small-step
- Play with continuations (“very fancy” language feature)