CSEP505: Programming Languages
Lecture 3: Small-step operational semantics, semantics via translation, state-passing, introduction to lambda-calculus

Dan Grossman

Autumn 2016

Where are we

- Finished our first syntax definition and interpreter
  - Was “large-step”
- Now a “small-step” interpreter for same language
  - Equivalent results, complementary as a definition
- Then a third equivalent semantics via translation
  - Trickier, but worth seeing
- Then quick overview of Homework 2
- Then a couple useful digressions
- Then start on lambda-calculus [if we have time]

Syntax (review)

- Recall the abstract syntax for IMP
  - Abstract = trees, assume no parsing ambiguities
- Two metalanguages for "what trees are in the language"

```
type exp = Int of int | Var of string
         | Plus of exp * exp | Times of exp * exp
type stmt = Skip | Assign of string * exp
          | Seq of stmt * stmt
          | If of exp * stmt * stmt
          | While of exp * stmt
```

e ::= *c* | *x* | *e* + *e* | *e* * *e*
s ::= skip | *x* := *e* | *s* ; *s* | if *e* then *s* else *s* | while *e* *s*

(*x* in {x1,x2,…,y1,y2,…,z1,z2,…,…})
(*c* in {…,-2,-1,0,1,2,…})

Expression semantics (review)

- Definition by interpretation: Program means what an interpreter written in the metalanguage says it means

```
type exp = Int of int | Var of string
       | Plus of exp * exp | Times of exp * exp
type heap = (string * int) list

let rec lookup h str = … (*lookup a variable*)

let rec interp_e (h:heap) (e:exp) =
 match e with
  Int i        ->i
 |Var str      ->lookup h str
 |Plus(e1,e2) ->(interp_e h e1)+(interp_e h e2)
 |Times(e1,e2)->(interp_e h e1)*(interp_e h e2)
```

Statement semantics (review)

- In IMP, expressions produce numbers (given a heap)
- In IMP, statements change heaps, i.e., they produce a heap (given a heap)

```
let rec interp_s (h:heap) (s:stmt) =
  match s with
   Skip -> h
  |Seq(s1,s2) -> let h2 = interp_s h s1 in
                 interp_s h2 s2
  |If(e,s1,s2) -> if (interp_e h e) <> 0
                  then interp_s h s1
                  else interp_s h s2
  |Assign(str,e) -> update h str (interp_e h e)
  |While(e,s1) -> (* two slides ahead *)
```

Heap access (review)

- In IMP, a heap maps strings to values
- Yes, we could use mutation, but that is:
  - less powerful (old heaps do not exist)
  - less explanatory (interpreter passes current heap)

```
type heap = (string * int) list
let rec lookup h str =
  match h with
    [] -> 0 (* kind of a cheat *)
   |(s,i)::tl -> if s=str then i else lookup tl str
let update h str i = (str,i)::h
```

- As a *definition*, this is great despite terrible waste of space

Meanwhile, **`while`** (review)

- Loops are *always* the hard part!

```
let rec interp_s (h:heap) (s:stmt) =
  match s with
   …
  | While(e,s1) -> if (interp_e h e) <> 0
                   then let h2 = interp_s h s1 in
                        interp_s h2 s
                   else h
```

- **`s`** is **`While(e,s1)`**
- Semi-troubling circular definition
  - That is, **`interp_s`** might not terminate

Finishing the story

- Have **`interp_e`** and **`interp_s`**
- A “program” is just a statement
- An initial heap is (say) one that maps everything to 0

```
type heap = (string * int) list

let empty_heap = []

let interp_prog s =
  lookup (interp_s empty_heap s) “ans”
```

Fancy words: We have defined a large-step operational-semantics using OCaml as our metalanguage

Fancy words

- Operational semantics
  - Definition by interpretation
  - Often implies metalanguage is "inference rules"
    (a mathematical formalism we'll learn in a couple weeks)

- Large-step
  - Interpreter function "returns an answer" (or doesn't)
  - So definition says nothing about intermediate computation
  - Simpler than small-step when that's okay

Language properties

- A semantics is *necessary* to prove language properties
- Example: Expression evaluation is *total* and *deterministic*
  "For all heaps **`h`** and expressions **`e`**, there is exactly one integer **`i`** such that **`interp_e h e`** returns **`i`**"
  - Rarely true for "real" languages
  - But often care about subsets for which it is true
- Prove for all expressions by induction on the tree-height of an expression

Where are we

- Finished our first syntax definition and interpreter
  - Will quickly review
- Then a second “small-step” interpreter for same language
  - Equivalent results, complementary as a definition
- Then a third equivalent semantics via translation
  - Trickier, but worth seeing
- Then quick overview of Homework 2
- Then a couple useful digressions
- Then start on lambda-calculus [if we have time]

Small-step

- Now redo our interpreter with small-step
  - An expression/statement “becomes a slightly simpler thing”
  - A less efficient interpreter, but has advantages as a definition (discuss after interpreter)

	Large-step	**Small-step**
`interp_e`	`heap->exp->int`	`heap->exp->exp`
`interp_s`	`heap->stmt->heap`	`heap->stmt->(heap*stmt)`

Example

Switching to concrete syntax, where each → is one call to **`interp_e`** and heap maps everything to 0

```
(x+3)+(y*z) → (0+3)+(y*z)
            → 3+(y*z)
            → 3+(0*z)
            → 3+(0*0)
            → 3+0
            → 3
```

Small-step expressions

"We just take one little step"

```
exception AlreadyValue

let rec interp_e (h:heap) (e:exp) =
 match e with
   Int i    -> raise AlreadyValue
  |Var str -> Int (lookup h str)
  |Plus(Int i1,Int i2)-> Int (i1+i2)
  |Plus(Int i1, e2)    -> Plus(Int i1,interp_e h e2)
  |Plus(e1, e2)        -> Plus(interp_e h e1,e2)
  |Times(Int i1,Int i2) -> Int (i1*i2)
  |Times(Int i1, e2)-> Times(Int i1,interp_e h e2)
  |Times(e1, e2)     -> Times(interp_e h e1,e2)
```

We chose "left to right", but not important

Small-step statements

```
let rec interp_s (h:heap) (s:stmt) =
 match s with
  Skip                -> raise AlreadyValue
 |Assign(str,Int i)-> ((update h str i),Skip)
 |Assign(str,e)      -> (h,Assign(str,interp_e h e))
 |Seq(Skip,s2)       -> (h,s2)
 |Seq(s1,s2)         -> let (h2,s3) = interp_s h s1
                        in (h2,Seq(s3,s2))
 |If(Int i,s1,s2)   -> (h, if i <> 0
                            then s1
                            else s2)
 |If(e,s1,s2) -> (h, If(interp_e h e, s1, s2))
 |While(e,s1) -> (*???*)
```

Meanwhile, `while`

- Loops are *always* the hard part!

```
let rec interp_s (h:heap) (s:stmt) =
  match s with
   …
  | While(e,s1) -> (h, If(e,Seq(s1,s),Skip))
```

- "A loop takes one step to its unrolling"
- **`s`** is **`While(e,s1)`**
- **`interp_s`** always terminates
- **`interp_prog`** may not terminate…

Finishing the story

- Have **`interp_e`** and **`interp_s`**
- A "program" is just a statement
- An initial heap is (say) one that maps everything to 0

```
type heap = (string * int) list
let empty_heap = []
let interp_prog s =
  let rec loop (h,s) =
     match s with
       Skip -> lookup h "ans"
     | _    -> loop (interp_s h s)
  in loop (empty_heap,s)
```

Fancy words: We have defined a small-step operational-semantics using OCaml as our metalanguage

Small vs. large again

- Small is really inefficient
  - Descends and rebuilds AST at every tiny step
- But as a *definition*, it gives a trace of program states
  - A state is a pair **`heap*stmt`**
  - Can talk about them e.g., “no state has x>17…”
  - Infinite loops now produce infinite traces rather than OCaml just “hanging forever”
- Theorem: Total equivalence: **`interp_prog`** (large) returns **`i`** for **`s`** if and only if **`interp_prog`** (small) does
  - Proof is pretty tricky
- With the theorem, we can choose whatever semantics is most convenient for whatever else we want to prove

Where are we

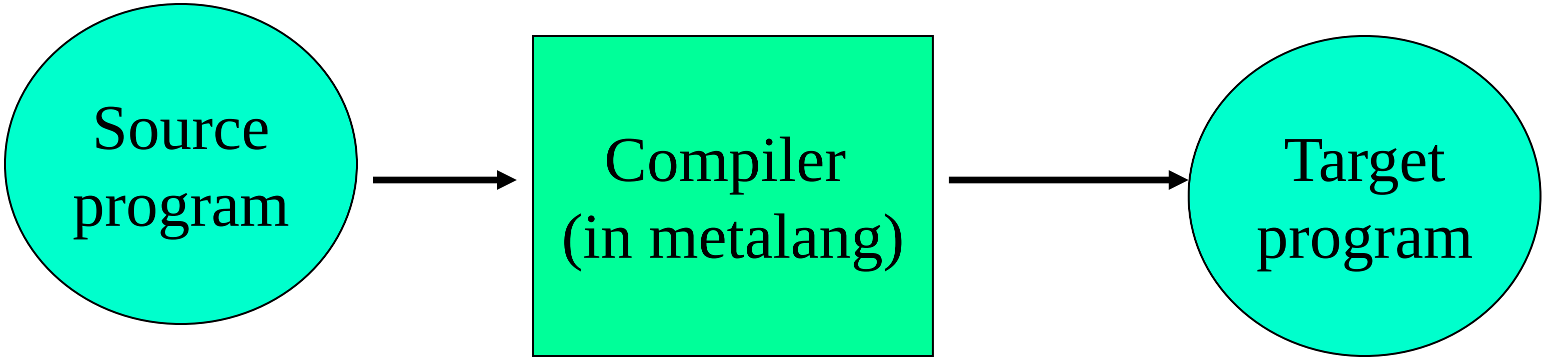
Definition by interpretation

- We have *abstract syntax* and two *interpreters* for our *source language* IMP
- Our metalanguage is OCaml

Now definition by translation

- Abstract syntax and source language still IMP
- Metalanguage still OCaml
- *Target language* now "OCaml with just functions strings, ints, and conditionals"
  - tricky stuff?

In pictures and equations



- If the target language has a semantics, then:

  compiler + targetSemantics = sourceSemantics

What we're "doing"

- Meta and target can be the same language
  - Unusual for a "real" compiler
  - Makes example harder to follow ☹
- Our target will be a subset of OCaml
  - After translation, you could "unload" the AST definition
    - (in theory)
  - An IMP while loop becomes a function
    - Not a piece of data that says "I'm a while loop"
    - Shows you can really think of loops, assignments, etc. as "functions over heaps"

Goals

- **`xlate_e:`**

  `exp -> ((string->int)->int)`

  - "given an exp, produce a function that given a function from strings to ints returns an int"
  - (**`string->int`** acts like a heap)
  - An expression "is" a function from heaps to ints
- **`xlate_s`**:

  `stmt->((string->int)->(string->int))`

  - A statement "is" a function from heaps to heaps
    - A "heap transformer"

Expression translation

```
xlate_e: exp -> ((string->int)->int)
```

```
let rec xlate_e (e:exp) =
  match e with
    Int i          ->  (fun h -> i)
   |Var str        ->  (fun h -> h str)
   |Plus(e1,e2)    ->  let f1 = xlate_e e1 in
                       let f2 = xlate_e e2 in
                       (fun h -> (f1 h) + (f2 h))
   |Times(e1,e2)   ->  let f1 = xlate_e e1 in
                       let f2 = xlate_e e2 in
                       (fun h -> (f1 h) * (f2 h))
```

What just happened

```
(* an example *)
let e = Plus(Int 3, Times(Var “x”, Int 4))
let f = xlate_e e (* compile *)
(* the value bound to f is a function whose body
   does not use any IMP abstract syntax! *)
let ans = f (fun s -> 0)(* run w/ empty heap *)
```

- Our target sublanguage:
  - Functions (including **+** and *****, not **interp_e**)
  - Strings and integers
  - Variables bound to things in our sublanguage
  - (later: if-then-else)
- Note: No lookup until “run-time” (of course)

Wrong

- This produces a program not in our sublanguage:

```
let rec xlate_e (e:exp) =
  match e with
    Int i        ->  (fun h -> i)
   |Var str      ->  (fun h -> h str)
   |Plus(e1,e2)  ->  (fun h -> (xlate_e e1 h) +
                                (xlate_e e2 h))
   |Times(e1,e2) -> (fun h -> (xlate_e e1 h) *
                               (xlate_e e2 h))
```

- OCaml evaluates function bodies when called (like YFL)
- Waits until run-time to translate **`Plus`** and **`Times`** children!

Statements, part 1

```
xlate_s:
        stmt->((string->int)->(string->int))
```

```
let rec xlate_s (s:stmt) =
 match s with
  Skip             -> (fun h -> h)
 |Assign(str,e) ->
   let f = xlate_e e in
   (fun h -> let i = f h in
             (fun s -> if s=str then i else h s))
 |Seq(s1,s2) ->
    let f2 = xlate_s s2 in (* order irrelevant! *)
    let f1 = xlate_s s1 in
    (fun h -> f2 (f1 h)) (* order relevant *)
 | …
```

Statements, part 2

```
xlate_s:
        stmt->((string->int)->(string->int))
```

```
let rec xlate_s (s:stmt) =
 match s with …
 |If(e,s1,s2) ->
   let f1 = xlate_s s1 in
   let f2 = xlate_s s2 in
   let f  = xlate_e e  in
   (fun h -> if (f h) <> 0 then f1 h else f2 h)
 |While(e,s1) ->
   let f1 = xlate_s s1 in
   let f  = xlate_e e  in
   (*???*)
```

- Why is translation of **`while`** tricky???

Statements, part 3

```
xlate_s:
        stmt->((string->int)->(string->int))
```

```
let rec xlate_s (s:stmt) =
 match s with
 …
 |While(e,s1) ->
   let f1 = xlate_s s1 in
   let f  = xlate_e e  in
   let rec loop h = (* ah, recursion! *)
     if f h <> 0
     then loop (f1 h)
     else h
   in loop
```

- Target language *must* have some recursion/loop!

Finishing the story

- Have **`xlate_e`** and **`xlate_s`**
- A “program” is just a statement
- An initial heap is (say) one that maps everything to 0

```
let interp_prog s =
  ((xlate_s s) (fun str -> 0)) “ans”
```

Fancy words: We have defined a “denotational semantics”

- But target was not math

Summary

- Three semantics for IMP
  – Theorem: they are all *equivalent*

- Avoided
  – Inference rules (for “real” operational semantics)
  – Recursive-function theory (for “real” denotational semantics)

- Inference rules useful for reading PL research papers
  – So we’ll start using them some soon

- If we assume OCaml already has a semantics, then using it as a metalanguage and target language makes sense for IMP

- Loops and recursion are deeply connected!

HW2 Primer

- Problem 1:
  - Extend IMP with saveheap, restoreheap
  - Requires 10-ish changes to our *large-step interpreter*
  - Minor OCaml novelty: mutually recursive types
- Problem 2:
  - Syntax plus *3 semantics* for a little Logo language
  - Intellectually transfer ideas from IMP
  - A lot of skeleton provided
- In total, less code than Homework 1
  - But more interesting code

HW2 Primer cont'd

```
e   ::= home | forward f | turn f | for i lst
lst ::= [] | e::lst
```

- Semantics of a move list is a "places-visited" list
  - type: **`(float*float) list`**
- Program state = move list, x,y coordinates, and current direction
- Given a list, "do the first thing then the rest"
- As usual, loops are the hardest case

This is all in the assignment

- With Logo description separated out

Where are we

- Finished our first syntax definition and interpreter
  - Will quickly review
- Then a second "small-step" interpreter for same language
  - Equivalent results, complementary as a definition
- Then a third equivalent semantics via translation
  - Trickier, but worth seeing
- Then quick overview of homework 2
- Then a couple useful digressions
  - Packet filters and other code-to-data examples
  - State-passing style; monadic style
- Then start on lambda-calculus [if we have time]

Digression: Packet filters

- If you're not a language semanticist, is this useful?

A very simple view of packet filters:

- Some bits come in off the wire
- Some applications want the "packet" and some do not
  - e.g., port number
- For safety, only the O/S can access the wire
- For extensibility, the applications accept/reject packets

Conventional solution goes to user-space for every packet and app that wants (any) packets.

Faster solution: Run app-written filters in kernel-space

What we need

- Now the O/S writer is defining the packet-filter language!

Properties we wish of (untrusted) filters:

1. Don’t corrupt kernel data structures
2. Terminate within a reasonable time bound
3. Run fast (the whole point)

Sould we allow arbitrary C code and an unchecked API?

Should we make up a language and “hope” it has these properties?

Language-based approaches

1. Interpret a language
   - + clean operational semantics, portable
   - - *may* be slow (or not since specialized), unusual interface
2. Translate (JIT) a language into C/assembly
   - + clean denotational semantics, existing optimizers,
   - - upfront (pre-1st-packet) cost, unusual interface
3. Require a conservative subset of C/assembly
   - + normal interface
   - - too conservative without help
   - related to type systems (we'll get there!)

More generally…

Packet filters move the code to data rather than data to code

- General reasons: performance, security, other?

- Other examples:
  - Query languages
  - Active networks
  - Client-side web scripts
  - …

State-passing

- Translation of IMP produces programs that take/return heaps
  - You could do that yourself to get an imperative "feel"
  - Stylized use makes this a useful, straightforward idiom

```
(* functional heap interface written by a guru
   to encourage stylized state-passing *)
let empty_heap = []
let lookup str heap =
  ((try List.assoc str heap with _ -> 0), heap)
let update str v heap = ((),(str,v)::heap)
(* … could have more operations … *)
```

- Each operation:
  - Takes a heap (last)
  - returns a pair: an "answer" and a (new) heap

State-passing example

```
let empty_heap = []
let lookup str heap =
   ((try List.assoc str heap with _ -> 0), heap)
let update str v heap = ((),(str,v)::heap)
```

```
(* increment "z", if original "z" is positive set
   "x" to "y" else set "x" to 37 *)
let example1 heap = (* take a heap *)
  let x1,heap = lookup "z" heap in
  let x2,heap = update "z" (x1+1) heap in
  let x3,heap = if x1>0
                then lookup "y"
                else (37,heap) in
  update "x" x3 heap (*return () and new heap*)
```

From state-passing to monads

- That was good and clearly showed sequence
  - But the explicit heap-passing was annoying
  - Can we abstract it to get an even more imperative feel?
- Two brilliant functions with “monadic interface” (obscure math)

```
(* written by a guru
   f1: function from heap to result & heap
   f2: function from arg & heap to result & heap *)
let bind f1 f2 =
 (fun heap ->
    let x,heap = f1 heap in
    f2 x heap)
(* just return e with unchanged heap *)
let ret e = (fun heap -> (e,heap))
```

Back to example

```
let bind f1 f2 =
 (fun heap -> let x,heap = f1 heap in f2 x heap)
let ret e = (fun heap -> (e,heap))
```

Naively rewriting our example with **`bind`** and **`ret`** seems awful

- But systematic from **`example1`**

```
let example2 heap =
 (bind (fun heap -> lookup "z" heap)
       (fun x1 ->
        (bind(fun heap -> update "z" (x1+1) heap)
             (fun x2 ->
              (bind(fun heap -> if x1 > 0
                                then lookup "y" heap
                                else ret 37 heap)
                   (fun x3 ->
                    (fun heap->update "x" x3 heap)))))
  heap
```

Clean-up

- But **`bind`**, **`ret`**, **`update`**, and **`lookup`** are written "just right" so we can remove every explicit mention of a heap
  - All since **`(fun h -> e1 … en h)`** is **`e1 … en`**
  - Like in imperative programming!

```
let example3 =
 bind (lookup "z")
      (fun x1 ->
         bind(update "z" (x1+1))
             (fun x2 ->
               bind(if x1 > 0
                    then lookup "y"
                    else ret 37)
                   (fun x3 ->
                      (update "x" x3))))
```

More clean-up

- Now let's just use "funny" indentation and line-breaks

```
let example4 =
 bind (lookup "z")             (fun x1 ->
 bind (update "z" (x1+1)) (fun x2 ->
 bind (if x1 > 0
       then lookup "y"
       else ret 37)            (fun x3 ->
      (update "x" x3))))
```

- This is imperative programming "in Hebrew"
  - Within a functional semantics

Adding sugar

- Haskell (not OCaml) then just has syntactic sugar for this "trick"
  - `x <- e1; e2` desugars to `bind e1 (fun x -> e2)`
  - `e1; e2` desugars to `bind e1 (fun _ -> e2)`

```
(*does not work in OCaml; showing Haskell
  sugar via pseudocode*)
let example5 =
  x1 <- (lookup "z") ;
  update "z" (x1+1)  ;
  x3 <-  if x1 > 0
         then lookup "y"
         else ret 37 ;
  update "x" x3
```

Adding sugar

- F# supports this idea with *workflows*
  - Better branding than *monads??* ☺ ☺
  - *Mostly* just syntactic sugar (but exceptions and other corners)

```
(* F#, do once to define state computation *)
type HeapBuilder () =
  member this.Bind(susp, func) = bind susp func
  member this.Return(x) = ret x
  member this.ReturnFrom(x) = x

let heap_monad = new HeapBuilder()
```

Adding sugar

- F# supports this idea with *workflows*
  - Better branding than *monads??* ☺ ☺
  - *Mostly* just syntactic sugar (but exceptions and other corners)

```
(* F#, example using heap_monad *)
let example5 =
  heap_monad {
   let! x1 = lookup "z"
   let! x2 = update "z" (x1+1)
   let! x3 = heap_monad {
               if x1 > 0 then lookup "y"
               else return 37
              }
   return! update "x" x3
}
```

What we did

We derived and used the *state monad*

Many imperative features (I/O, exceptions, backtracking, …) fit into a functional setting via monads (**`bind`** + **`ret`** + other operations)

– Essential to Haskell, the modern purely functional language
– "Just" redefine **`bind`** and **`ret`**

A key topic to return to if/when we spend a week on Haskell!

Relevant tutorial (using Haskell):

Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell

Simon Peyton Jones, MSR Cambridge

Where are we

- Finished our first syntax definition and interpreter
  - Will quickly review
- Then a second “small-step” interpreter for same language
  - Equivalent results, complementary as a definition
- Then a third equivalent semantics via translation
  - Trickier, but worth seeing
- Then quick overview of homework 2
- Then a couple useful digressions
- Then start on lambda-calculus [if we have time]
  - First motivate

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 won't work

- Then back up and define a language with *nothing* but functions
  - And we'll be able to encode everything else

Worth a try…

```
type exp  = … (* no change *)
type stmt = … | Call of string * exp
(*prog now has a list of named 1-arg functions*)
type funs = (string*(string*stmt)) list
type prog = funs * stmt

let rec interp_s (fs:funs) (h:heap) (s:stmt) =
 match s with
   …
   | Call(str,e) ->
     let (arg,body) = List.assoc str fs in
     (* str(e) becomes arg:=e; body *)
     interp_s fs h (Seq(Assign(arg,e),body))
```

- A definition yes, but one we want?

The "wrong" definition

- The previous slide makes function call assign to a global variable
  - So choice of argument name matters
  - And affects caller

- Example (with IMP-like concrete syntax):

```
[ (fun f x -> y:=x) ]
x := 2; f(3); ans := x
```

- We could try "making up a new variable" every time…

2nd wrong try

```
(* return some string not used in h or s *)
let fresh h s = …

let rec interp_s (fs:funs) (h:heap) (s:stmt) =
 match s with
   …
   | Call(str,e) ->
     let (arg,body) = List.assoc str fs in
     let y = fresh h s in
   (* str(e) becomes y:=arg; arg:=e; body; arg:=y
      where y is "fresh" *)
     interp_s fs h (Seq(Assign(y,Var arg),
                        Seq(Assign(arg,e),
                        Seq(body,
                            Assign(arg,Var y)))))
```

Did that work?

```
(* str(e) becomes y:=arg; arg:=e; body; arg:=y
    where y is "fresh" *)
```

- “fresh” is pretty sloppy (but okay, it’s malloc)
- Not an elegant model of a key PL feature
- Still wrong:
  - In functional or OOP: variables in **`body`** should be looked up based on where **`body`** came from
  - Even in C: If **`body`** calls a function that accesses a global variable named **`arg`**
  - Examples…

Examples

- Using higher-order functions

```
[ (fun f1 x -> g := fun z -> ans := x + z) ]
f1(2); x:=3; g(4);
```

  - "Should" set **ans** to 6, but instead we get 7 because of "when/where" we look up x

- Using globals and function pointers

```
[  (fun f1 x -> f2(y); ans := x) ;
   (fun f2 z -> x:=4) ]
 f1(3);
```

  - "Should" set **ans** to 3, but instead we get 4 because **x** is still fundamentally a global variable

Let's give up

- 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: $e ::= x \mid \lambda x.\ e \mid e\ e$

Values: $v ::= \lambda x.\ e$

That's all of it!

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:

$$(\lambda x.\ x)\ (\lambda y.\ y) \rightarrow \lambda y.\ y$$

$$(\lambda x.\ \lambda y.\ y\ x)\ (\lambda z.\ z) \rightarrow \lambda y.\ y\ (\lambda z.\ z)$$

$$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \rightarrow (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$$

Why substitution

- After substitution, the bound variable is *gone*
  - So clearly its name did not 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
- Evaluate application (call) by
  1. Evaluate left
  2. Evaluate right
  3. Substitute result of (2) in body of result of (1)
     - And evaluate result

A different semantics has a different *evaluation strategy*:

1. Evaluate left
2. Substitute right in body of result of (1)
   - And evaluate result

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)