Functional MiniJava, Calling variations

Lecture 18

When to do assignments

Why have a "functional" language

- Silly examples from Scheme
  - (define (addn n)(lambda(x)(+ x n))) ;returns funct.
  - (define addthree (addn 3))

- Less silly examples: encapsulate state, e.g. bank account balance in CS61a examples
- Ways to accomplish similar results via Java "privacy" of variables or methods
- Motivation for first-class functions-
  - Eliminate need for other mechanisms
  - Unifies concepts of functions and data

Do we need functions returning functions?

- Less silly examples: a translator that returns an answer as a function:
- (translate "x+sin(x)") which might return
  (lambda(x)(+ x (sin x));

- (defdiff f(x))(+ x (sin x))
- ; could return (f x) and derivative e.g. 1+cos(x).
- ; or could return a (lambda (…) …) which could be compiled and called later

```
(lambda (g0) ;; function generated automatically
   (let ((t4 0) (f3 0) (t2 1) (f1 g0))
   (setf f3 (sin g0))
   (setf t4 (cos g0))
   (setf t2 (+ t4 t2))
   (setf f1 (+ f3 f1))
   (values f1 t2)))
```

Contrast Translate to Addthree

- Translate takes expressions, perhaps strings, and return expressions. Since expressions are also programs potentially everything is happy
- **Functions** are, however, more than expressions in that they also have environments… as in the addthree, where n is bound to 3 in the environment.

Changes to MJ to make it "functional" section 15.1

How will this work? Here's an example

```
type  foo= (int,String) -> int[]
                    //assume class String exists
    type bar=(int->String, int) -> int->int
```

These say that a variable F of type foo may be assigned a value which is a function which takes args that are int, String and returns an array (how long??) of ints.

And that a variable B of of type bar may be assigned a function which takes two args, one of which is a function int->String, and the other an int. and returns another function of type int -> int.

Lexical changes to MJ to make it "functional"

Lexical changes are simple. Add one keyword and one operator

tokens for MJ `->` **type**

Grammar changes to MJ to make it "functional"

New grammar rules for type declaration

- ClassDecl → **type** id = ty;

  ty → ty **->** ty

  ty → (ty {,ty}) **->** ty

  ty → () **->** ty

  And for CallExp

New grammar rules for CallExp

exp → exp (exp {,exp}) ;actually unlikely to be LALR(1)

What's this change to CallExp?

Must allow the use of an expression like F(x) in the place we previously expected only a name of a function/method. That is, the expression F(x) may evaluate to a function g, in which case one must call g.

We now allow a call like Add(3)(4)

Where Add(3) perhaps returns a function that adds three to its argument.

Type Check changes to MJ to make it "functional"

New typechecking for AST FunTy

Compare the signatures for 2 functions carefully.

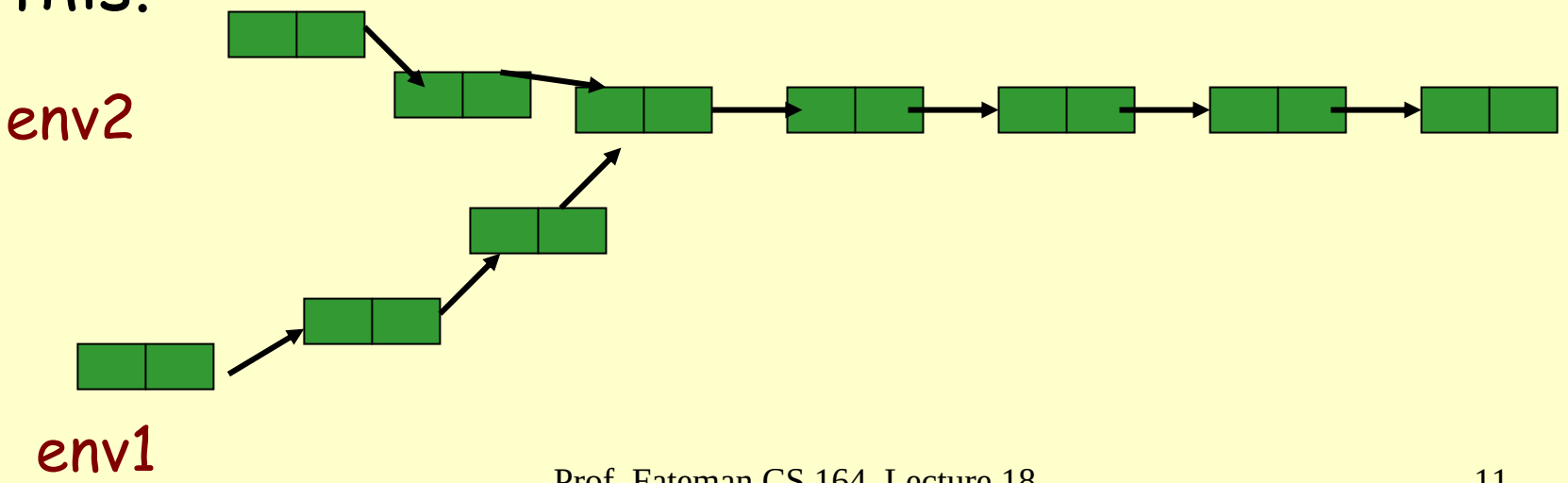
Deciding about structure vs name equivalence again.

(personally, I think this would be a delicate business: to match type bar=(int->string,int) -> int->int)

Interpreter Changes to MJ to make it "functional"

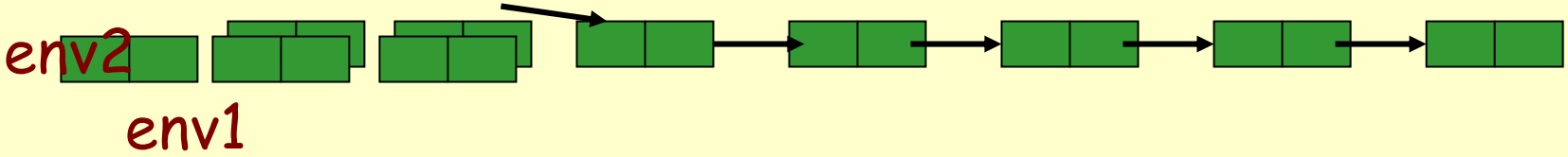
In a traditional Scheme environment, no change!

The environments persist until the Lisp GC removes them. Lists of bindings can look like this.



Interpreter Changes to MJ to make it "functional"

A stack interpreter cannot build such a tree. Its lexical environments persist only while within lexical scope. We need to preserve until function, returned upward, is called. Here env2 clobbers env1.



Example… of upward functional arg (return)

New function here..

What is "pure functional" ?

Equational reasoning about programs requires that (say) f(3) always is the same.

Prohibits "side effects" forbidding f from doing assignment, output, or input.

How to do this?

Assignment is easy: don't allow it. Just allow initial values to be set. Functions can bind new parameters on call. Just not reset anything.

What is "pure functional" output like?

How to get around the output restriction. It is easily to write it down in Lisp: Conceptually change this:

```
  ;main_prog ….
 (print  xyz);
   do_more_stuff;  maybe print other stuff
….
   return_from_main;
     ;;by converting print to  (cons xyz  (rest of program…))
     ;; any prints in "do_more_stuff" will also be changed to conses
…
  (print (cons xyz (cons  (do_more_stuff) ….  (return_from_main
     )…)
```

…so all the output is stacked up until the program finishes

What is "pure functional" input like?

Oh, MJ has no input.. But if it DID have input.. Change :

```
;main_prog ….
(let  ((x (read  *std-io*)) ;; remember, no assignment, only bindings
    (f x) ;  use x..
```

to

```
(let ((x 12345))  ;  whatever it would have read!
 (f x)
```

or just

```
(f 12345)
```

Any function like read must, under the restrictions of pure functional programming, always produce the same thing. That's what happens here.

What if we wanted to add this to MJ?

See pgm 15.4 in text for an new type, answer

This is the consing up of all the prints. A new method exit() prints the answer.

And also a hacked up ReadByte which instead of reading from some mysterious source, has an argument which is what it would have read.

Remind me again why we would do this?

We can reason about functions (otherwise equals will not be apparent. F(3) is equal to F(3) in FP.)

Some programs look nicer, e.g. tree insertion, program 15.3. But we probably knew that from CS61a. That non-destructive tree insertion was cool, destructive version was error-prone.

Remind me again why we would do this?

```
(defun ti(a tree) ;; tree insertion. Copy route to insert point
  (cond ((null tree) (list a nil nil)) ;; make a tree with node=a, left, right
      ((< a (node tree))  ;;
       (list (node tree) (ti a (left tree)) (right tree)))
      (t (list (node tree)(left tree)(ti a (right tree))))))

(defun st(a tree)  ;; search in a tree
  (cond ((null tree) 'not-found)
      ((= a (node tree)) tree)
      ((< a (node tree)) (st a (left tree)))
      (t (st a (right tree)))))

(defun node(x)(car x)) ;; data abstraction. Remember that?
(defun left(x)(cadr x)) ;; could also abstract out the comparison
(defun right(x)(caddr x))
```

(ti 5(ti 3 (ti 4 (ti 1 nil)))) →
(1 nil (4 (3 nil nil) (5 nil nil)))

Other issues of Chapter 15

- Call by name
  - Important for exams (GRE etc)
- Lazy evaluation
  - Variation, memoization based

Call by name (Important on CS GRE)

Invented for Algol 60. Often it has the same as call by reference, but not always.

Define a function Q(a,x) {x=3; a=4}

Declare int z[]; z= new array[10] of 0; ….

Call Q(z[i],i) // ASSUME CALL BY NAME

➔ results in setting z[3] to 4. versus…

Call Q(m,n)

➔ Simply sets global n to 3, m to 4. Same as call by ref.

To implement call by name: each value is a thunk that computes its value or location when used;

One implementation [equivalent, easier for humans to reason about]: copy over the body..

define function Q(a,x) {x=3; a=4}

When you see the call to Q, copy over the body with the parameters substituted in. Thus Q(z[i],i] becomes this:

{i=3;
z[i]=4}

An implementation must copy over the function body, renaming as appropriate to avoid "other" conflicts.

It must not matter if we had defined Q(a,z) {z=3,a=4}

(The parameter z must be renamed.. E.g.

Q(a,Q_z) {Q_z=3,a=4} before copying the body over.

Lazy vs Strict

- Lazy evaluation helps avoid equivalence "if programs halt", sometimes. Values are computed only when the results actually matter
- Opposite of lazy is "strict".. All expressions are evaluated as control flow reaches them, whether or not their results are needed.
- Two variants, Lazy eval vs call-by-need…

Lazy evaluation

= call by need++. Available in a pure-functional situation. Each "thunk" of call-by-name is now 2 cells: the thunk function + the memoized value if it has been visited before.

You've seen this in CS61a. …Set up fibonacci function to remember fib(0), fib(1), then [first time] compute fib(2)=fib(0)+fib(1) but REMEMBER fib(2). Compute

fib(3)= fib(1)+fib(2) remembered etc.

This is potentially an enormous savings in heavily-recursive purely-functional programs. For call-by-name or lazy evaluation, careful compilation can make this efficient.

Argument against this: what a doofus programmer wrote a program that was so inefficient?

Argument in favor: what could be simpler to write, and why force the programmer to "optimize" what the computer can do?

Final putdown: the remembered fib program is linear, the computation can be done in log time.

A thought: should cons evaluate its args? (Should it be by-need? Or lazy?)

If you make the following arrangements:

instead of (cons a b), do (list 'CONS 'a 'b)

Instead of (car x) do

(if (eq (car x) 'CONS) (eval (cadr x)))

Instead of (cdr x) do

(if (eq (car x) 'CONS) (eval (caddr x)))

;; really we can't do (list 'CONS 'a 'b) but need

(list 'CONS (closure of a, environment) (closure of b, environment)) … so we can do (eval a) or (eval b).

Look at it this way. No one knows what's inside a cons cell without looking at its car or cdr. So don't compute it until someone looks. Pprint looks..

Call by value, copy-in/out, etc

Value : imagine all arguments which are simple, are copied over into a space owned by the called program. For example, stack space.

Function foo(a,b,c) { a=b+c;… }

changes nothing outside foo

Function foo(x, b,c) { a=b+c;… }

perhaps changes a outside foo.

Copy-in/out sometimes referred to as call by value/return can be used (Fortran)– if there are no other "aliased" access routes to variables, they are equivalent. Local access fast, copying back is a cost.

Call by reference in FORTRAN

Function foo(a,b,c)= (a=b+c)

//this is not exactly FORTRAN syntax

call foo(3,4,5)

Print(3)

➔ may result in printing 9.

This is not "correct usage" but has historically happened in common implementations…

MJ function calls

As a practical semantic view, Java calls look rather like lisp. Call by value, but all values are refs to objects. The explicit passed arguments to a method are data on a stack, copied there in the call process.

The call Foo(x,y,z) cannot change these names x, y, or z.

Just as Foo(1,2,3) cannot change the values of constants.

What does Foo(int a) { a=35; ..} do? Only local value for a changes.

However, if x is an array, Foo(x) can change x[0], x[1]... because a pointer to the array x is given to Foo.