Language definition by interpreter, translator, continued

Lecture 15

So far, two ways of looking at MJ for executing code

- MJ interpreter (see simple-interp.lisp)
- MJ translator into Common Lisp (see simple-translate.lisp)
  - Common Lisp can be translated into machine language

MJ interpreter

- Environment includes all Class data
- Method calls require setting up a new stack frame
  - All formal parameters matched to actual parameters
  - Stack environment then passed to interpreter recursively
  - Statements in method body evaluated
  - Return from method call pops frame from stack
  - To test --Try running (say) Factorial, tracing mj-call or mj-dispatch or mj-new-frame or insert output statements `(format t "~%xxx=~s"  xxx)`

MJ translator

- Running AST through the translator produces a new (file) of Lisp code.
- Environment still has Class methods.
- “New” programs produce instances.
- Method calls changed into Lisp function calls.
- Lisp takes care of all the rest, in particular
  - All formal parameters matched to actual parameters
  - Stack environment then passed to interpreter recursively
  - Statements in method body executed by Lisp
  - Return from method returns Lisp stack to state previous to method call
  - Just trace names, e.g. mjl-ClassName-MethodName

MJ translator

- Consider Factorial.java.
- The translator produces about 24 lines of lisp.

Factorial in MJ

```
class Factorial{
    public static void main(String[] a){
     System.out.println(new Fac().ComputeFac(10));
    }}

class Fac {
    public int ComputeFac(int num){
     int num_aux ;
     if (num < 1)
        num_aux = 1 ;
     else
        num_aux = num * (this.ComputeFac(num-1)) ;
     return num_aux ;
    }
}
```

Compiled Factorial

```
;; Fac::ComputeFac
(defun mjl-Fac-ComputeFac (this L1)  ; L1 is num
  (let ((L2 0))                      ; L2 is num_aux
    (if (< L1 1)
        (setf L2 1)
      (setf L2
            (* L1
               (let* ((obj this) (vtable (elt obj 0))) ;recursive call. Explain obj
                 (funcall (gethash 'ComputeFac vtable) obj (- L1 1))))))
    L2))
;; Vtable for Fac
(setf mjl-Fac (make-hash-table))
(setf (gethash 'ComputeFac mjl-Fac) #'mjl-Fac-ComputeFac)
;; Constructor for Fac
(defun mjl-Fac+ ()
  (let ((obj (make-array '(1)))) (setf (elt obj 0) mjl-Fac) obj))
```

Compiled Factorial

```
;; Main routine
(format *mj-stdout* "~A~%"    ;print the result
      (let* ((obj (mjl-Fac+)) (vtable (elt obj 0))) ;create a Fac object
        (funcall (gethash 'ComputeFac vtable) obj 10)))
```

Looking up vars at runtime

- Lisp translation makes the separation clear. There are local variables handled by Lisp, and other variables or methods are looked up in the object.
- If we are doing our own interpreting, we have two places to look
  - Local parameters to a method
  - Vars relative to this. (the object whose method is being called)
    - Variables in this class
    - Variables in the parent class(es)

Adding bindings to an environment.. Scheme

```
(defun extend-env1 (p1 a1 e)
  ;; add one binding to the environment
  (cons (cons p1 a1) e))

(defun extend-env (p a e)
  ;; add a list of bindings to the environment. We have two
  ;; variable/value lists.  (extend-env '(a b c) '(1 2 3) '((d . 4)))
  ;; -> ((a . 1) (b . 2) (c . 3) (d . 4))
  (if (null p) e
    (extend-env1 (car p)(car a)(extend-env (cdr p)(cdr a) e))))
```

Adding bindings to a MJ env

```
(defun mj-new-frame(method  this actuals e)
 ;; pick out the formal args from method.
;; we will map them to the actuals.
;; make a new env, same as the old layout but with a new frame.
;; the frame is an array, and will contain formal <-> actual mapping.
;; Make a new environment.. Copy all the old bindings into it and put in
;; a few more. It looks like…
  (cons (layout method)  new-frame))
```

We need less info for typechecking

- The layout (classes, methods)
- The variables: we do not really need space for values (though it won't hurt much); we need the types and the line/column locations of declarations for error messages.
- We don't need to actually locate (with array offsets etc) the location of values, but if we compute them, we can use this for compiling, later.