Evaluation
(Scheme and ML share model – focus on Scheme)

• Numbers are interpreted in base 10.
• Names are looked up in an environment, starting with the current environment, following static links, until the name is found. If it is not found, an error occurs.
• (define <var> <expr>) is evaluated by first evaluating <expr>, and binding, in the current environment, the name <var> to that value.
• (lambda …) is evaluated by creating an appropriate closure (sometimes called a procedure). The closure’s environment link refers to the environment which was active when the closure was created.
• All members of an application are evaluated, after which the first is applied to the rest.
• A procedure application is evaluated by creating an environment to bind the parameters of the procedure to its arguments, and then evaluating the body relative to this new environment. The static link of the environment refers to the same environment as the closure, whereas the dynamic link refers to the environment that was active when the procedure was applied.
Functional languages on CSE UNIX systems (e.g. pollux)

- **Installed in** /projects/alphonce/bin
  - Scheme
    - text-based interpreter: mzscheme
    - graphical ide: drscheme
      - must add /util/gnu/lib to LD_LIBRARY_PATH
  - Erlang: erl
- **Installed in** /util/bin
  - Lisp (Allegro Common Lisp):
    - standalone: acl
    - from within Emacs: run-acl
  - ML
    - Standard ML of NJ: sml
Examples

SCHEME

> 12
12
> (define x 12)
> x
12
> (define addOne
    (lambda (x) (+ x 1)))
> (addOne x)
13
> (define add
    (lambda (x y) (+ x y)))
> (add 3 x)
15
> (define adder
    (lambda (x)
        (lambda (y) (+ x y))))
> (define add3 (adder 3))
> (add3 x)
15
> (define add7 (adder 7))
> (add7 x)
19

ML

- 12
  val it=12:int
- val x=12;
  val x=12:int
- x
  val it=12:int
  - fun addOne x = x+1;
  val addOne=fn:int->int
  - addOne x
  val it=13:int
  - fun add (x,y) = x+y;
  val add=fn:int*int->int;
  - add(3,x);
  val it=15:int
  - fun adder x y = x+y;
  val adder=fn:int->int->int
  - val add3 = adder 3;
  val add3=fn:int->int
  - add3 x;
  val it=15:int;
  - val add7 = adder 7;
  val add7=fn:int->int
  - add7 x;
  val it=19:int;
Although Scheme is syntactically a simple language, it supports sophisticated modeling: higher-order functions are a powerful tool in describing problems.

Many ideas in design patterns have their roots in functional programming (e.g. strategy allows us to treat methods as first-class, a decorator acts like a function which maps a function to a function – think of the stream decorators in Java).

- We describe something as first-class if it can be created at runtime, created anonymously, named, stored in data structures, passed as an argument to a function, returned as a value from a function.
- With mutation (the ability to change a name-value binding in an environment) sophisticated systems can be built quite compactly.
Interactive session

- We worked on an interactive session:

\[
\begin{align*}
&\text{(define addOne (lambda (x) (+ x 1)))} \\
&\text{(define add (lambda (x y) (+ x y)))} \\
&\text{(define adder (lambda (x) (lambda (y) (+ x y)))))}
\end{align*}
\]