Recall Lexical Scoping

Variables are bound when procedures are defined not when they are called. For example:

```
(define x 10)
(define (x-plus y) (+ x y))

(x-plus 2)                       => 12

(define (x-plus-double x) (x-plus (* 2 x)))

(x-plus-double 2)                => 14
```


Their Environments

The environment where `(* 2 x)` is evaluated is:

```
(((x 2))
 ((x 10)
  (+ #<primitive-procedure +>)
  (- #<primitive-procedure +>)
  ...
 ))
```

And we get 4.
The environment where `(+ x y)` is evaluated is:

```
(((y 4))
 ((x 10)
  (+ #<primitive-procedure +>)
  (- #<primitive-procedure +>)
  ...
 ))
```

And we get 14.

Dynamic Scoping

The *free variables* are bound dynamically at procedure call.
Local environment frame is added to calling procedure's environment.

```
(define x 10)
(define (x-plus y) (+ x y))

(x-plus 2)                       => 12

(define (x-plus-double x) (x-plus (* 2 x)))

(x-plus-double 2)                => 6
```

Environment frames in dynamic scoping correspond with call stack.

Their Environments

The environment where `(* 2 x)` is evaluated is:

```
(((x 2))
 ((x 10)
  (+ #<primitive-procedure +>)
  (- #<primitive-procedure +>)
  ...
 ))
```

And we get 4.
The environment where `(+ x y)` is evaluated is:

```
(((y 4))
 ((x 2))
 ((x 10)
  (+ #<primitive-procedure +>)
  (- #<primitive-procedure +>)
  ...
 ))
```

And we get 6.

Why Dynamic Scoping?

- It is a way to pass parameters.
- Unix environment variables use dynamic scoping.
- Exceptions handlers are dynamically scoped.

Scheme Programs

Want to execute your scheme code as a program? Write "shell" scripts in Scheme?

```
#!/usr/bin/guile -s
!#

;;; this shell script prints out its
;;; arguments in reverse order

(for-each            ;; like 'map', but ignores value.
   (lambda (x)
      (display x)
      (newline)
   )
   (reverse (command-line))
)
```


Where is Scheme?

- Education: used for introductory programming courses.
- Artificial Intelligence.
- Application Extension Language.
- Rapid Prototyping.

Scheme Summary

- Expression oriented.
- Uses lists extensively.
- Uses recursive extensively, has tail recursive optimization.
- Lexically scoped.
- Functional style (not pure).
- Interpreted, but compilers exist.
- Simple syntax but expressive.
- Dynamically typed (and type-safe).
- Garbage collected.