# scope and evaluation

```
(let ([var1 expr1]
      ...
      [varn exprn])
  body)
```

Create local variables and bind them to expression results.

The **scope** of these variables is the body of the let statement.

**Evaluation:** `expr1, ..., exprn` are evaluated in some **undefined order**, saved, and then assigned to `var1, ..., varn`. In our interpreter, they have the appearance of being evaluated **in parallel**.

# scope and evaluation

Consider:

```
(define (sq-cube x)
  (let ([sqr (* x x)]
        [cube (* x (* x x))])
    (list sqr cube)))
```

Want to reuse `sqr`.

# scope and evaluation

Want to reuse sqr.

```
(define (sq-cube x)
  (let ([sqr (* x x)]
        [cube (* x sqr)])
    (list sqr cube)))
```

But this does not work: sqr is undefined at the time of evaluating (* x sqr)

# scope and evaluation

```
(let* ([var1 expr1]
        ...
       [varn exprn])
   body)
```

The **scope** of each variable is the part of the `let*`-expression to the right of the binding.

**Evaluation:** `expr1, ..., exprn` are evaluated **sequentially**, from left to right.

# scope and evaluation

Use let*:

```
(define (sq-cube x)
  (let* ([sqr (* x x)]
         [cube (* x sqr)])
    (list sqr cube)))
```

# scope and evaluation

```
(letrec ([var1 expr1]
          ...
         [varn exprn])
   body
)
```

**Scope:** Each binding of a variable has the entire letrec expression as its region.

**Evaluation:** `expr1, ..., exprn` are evaluated in an **undefined order**, saved, and then assigned to `var1, ..., varn`, with the appearance of being evaluated in parallel.

# scope and evaluation

```
(letrec ([my-even?
          (lambda (x)
            (if (= x 0)
                #t
                (my-odd? (- x 1))))]
         [my-odd?
          (lambda (x)
            (if (= x 0)
                #f
                (my-even? (- x 1))))])
  (if (and (my-even? 4) (not (my-odd? 4))
           (my-odd? 5)  (not (my-even? 5)))
      42
      0))
```

# scope and evaluation

```
(let ([x 2]) (* x x))
```
⇒ 4

```
(let ([x 4]) (let ([y (+ x 2)]) (* x y)))
```
⇒ 24

```
(let ([x 4] [y (+ x 2)]) (* x y))
```
⇒ is an error: unbound variable x

```
(let* ([x 4] [y (+ x 2)]) (* x y))
```
⇒ 24

# scope and evaluation

Question: Why would you ever prefer to use `let` instead of, say, `let*`?

# semantics of let

```
(let ((v1 e1)...(vn en)) expr)
                ⇕
((lambda (v1...vn) expr) e1...en)
```

AND

```
(let* ((v1 e1) (v2 e2)) expr)
                ⇕
((lambda (v1) ((lambda (v2) expr) e2)) e1)
```

All binding of values to variables is by parameter passing ($\equiv$ lambda reduction):
$\Rightarrow$ **no assignment**

# closure

A closure is a record that contains:

- a function and
- an environment

```
(define (make-inc x)
  (lambda (y) (+ x y)))
```

```
(define inc-by-5 (make-inc 5))
(define inc-by-10 (make-inc 10))
```

```
> (inc-by-5 100)
```

```
> (inc-by-10 100)
```

# closure

A closure is a record that contains:

- a function and
- an environment

In the expression `(lambda (y) (+ x y))` we say that `x` is a free variable.

An environment captured when a closure is created will contain bindings for all free variables.

# closure

Consider:

```
(define x 100)
(define (plus-x y)
  (+ x y))
(plus-x 10)

(let ([x 200])
  (plus-x 10))

(set! x 200)
(plus-x 10)
```

What is the value of the first `(plus-x 10)`?
What is the value of the second `(plus-x 10)`?
What is the value of the third `(plus-x 10)`?
Note: `set!` is **not** a functional construct.

# closure

In Python:

```
def make_inc(x):
    return lambda y: x + y


inc_by_5 = make_inc(5)
inc_by_10 = make_inc(10)


>>> inc_by_5(100)
105
>>> inc_by_10(100)
110
>>>
```

# closure

In Python, unlike in Racket:

```
def plus_x(y):
    return x + y
```

```
>>> plus_x
<function plus_x at 0x7fc2ff72f670>
>>> x
NameError: name 'x' is not defined
```

Can define later:

```
x = 100
print(plus_x(10))
x = 200
print(plus_x(10))
```

Output:

# closure

Consider:

```
(define counter
  (let ([count 0])
    (lambda ()
      (set! count (+ count 1))
      count)))

(counter)
(counter)
(counter)
```

An alternative to OOP?
Even more interesting...

# closure

"Local" and "global" state variables?

```
(define make-counter
  (let ([global-count 0])
    (lambda ()
      (let ([local-count 0])
        (lambda ()
          (set! global-count (+ global-count 1))
          (set! local-count (+ local-count 1))
          (cons global-count local-count))))))
(define counter1 (make-counter))
(define counter2 (make-counter))
(counter1)
(counter1)
(counter2)
(counter2)
(counter1)
```

An alternative to OOP?

# closure

Exercise: In Python, define a counter similar to the one we defined above in Scheme. Do not define any classes. Your counter should behave as follows:

```
>>> counter1 = make_counter()
>>> counter2 = make_counter()
>>> counter1()
(1, 1)
>>> counter1()
(2, 2)
>>> counter2()
(1, 3)
>>> counter2()
(2, 4)
>>> counter1()
(3, 5)
>>>
```

# recursion

**linear recursion**: there is at most one recursive call made in any execution of function body.

**flat recursion**: recursion applied over 'top' items of a list.

**deep recursion**: (aka tree recursion) recursion applied over all items.

**structural recursion**:

```
(define my-func
  (lambda (lst)
    (cond ((empty? lst) ... )
          (else ... (first lst) ...
                    (my-func (rest lst)) ... ))))
```

**mutual recursion**: functions call each other, rather than themselves.

# tail-recursion

- The recursive call is in the last function application in function body.
- A language can implement tail-call optimization: no stack required!
  - Any Scheme implementation is required to be tail-recursive.
  - Python does not implement tail-call optimization.
  - A choice of laguage designers. Pros? Cons?

Let's look at some examples...