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))
```
$\Rightarrow$

```
(let ([x 4]) (let ([y (+ x 2)]) (* x y)))
```
$\Rightarrow$

```
(let ([x 4] [y (+ x 2)]) (* x y))
```
$\Rightarrow$

```
(let* ([x 4] [y (+ x 2)]) (* x y))
```
$\Rightarrow$

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)

>>> inc_by_10(100)

>>>
```

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...

Evaluation and Continuations

Racket evaluation can be viewed as the simplification of expressions to obtain values.

`(+ 1 1)` $\rightarrow$ 2

An expression that is not a value can always be partitioned into two parts:

- a **redex** ("reducible expression"), and
- the **continuation**, which is the evaluation context surrounding the redex.

In `(- 4 (+ 1 1))` , we have:

- `(+ 1 1)` is a redex, and
- `(- 4 [])` is a corresponding continuation.

`(- 4 (+ 1 1))` $\rightarrow$ `(- 4 2)` $\rightarrow$ 2

Evaluation and Continuations

```
(+ (* 3 5) (- 10 3))
```

- We can think of a **continuation** as "what needs to be done in order to complete the evaluation".
- The continuation of `(- 10 3)` is the context:

  or, in Racket:

- The continuation of `(* 3 5)` is the context:

  or, in Racket:

- Given a well-formed Racket program, every well-formed sub-expression in this program corresponds to a point in the evaluation of the program, and every such point corresponds to a continuation.

Continuation-Passing Style (CPS)

Compute $n!$ (n factorial)

- a "direct-style" definition:

```
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))
```

Continuation-Passing Style (CPS)

Compute $n!$ (n factorial)

- a tail-recursive definition:

```
(define (fact-tail n)                    ; return n!
    (local [(define (ft                  ; return
```

Continuation-Passing Style (CPS)

Compute $n!$ (n factorial)

- a CPS definition:

```
(define (fact-cps n)                    ; return n!
    (local [(define (fcps       ; return
```

Evaluation and Continuations

- In Racket, continuations are first class values, just like regular functions.
- The current continuation can be reified as a function by using the built-in function `call-with-current-continuation` (or `call/cc` for short).

`(call/cc f)` works like this:

- Take a unary function `f`.
- Apply `f` to the current continuation.

A powerful control mechanism that can be used to implement exceptions, non-determinism, (lazy) generators, and more. But this is more advanced than what we cover in this course.

pure functional languages

Programs are viewed as collections of **functions**.

Execution of programs is viewed as **evaluation**.

In contrast, what are programs in an imperative language? what is execution of a program in an imperative language?

pure functional languages

- **Referential transparency:**

  The value of a function application is independent of the context in which it occurs.

  A language is referentially transparent if we may replace one expression with another of equal value anywhere in a program without changing the meaning of the program.

  - Value of `f(a,b,c)` depends only on the values of `f`, `a`, `b` and `c`.
  - It does not depend on the global state of computation.

pure functional languages

- **Referential transparency:**

  Main advantage: facilitates reasoning about programs and applying program transformations.

  Aside: Referential transparency is no longer considered a distinguishing feature of functional programming languages. It arguably holds in well-designed imperative languages too.

pure functional languages

- **No assignment statement:**

  A variable may not have a value, or its value might be a function that has not been applied to its arguments yet, but variables in pure FP are said to be **logical** in that, having acquired a value in the course of an evaluation, they retain that value until the end of evaluation.

  This is similar to the manner in which variables are used in mathematics. Remember when you had to unlearn your mathematics training to make sense of `x := x+1`? In FP, this is anathema!

pure functional languages

- **No side effects:**

  Function calls have no side effects.
  - No need to consider global state.
  - Programs are easier to reason about.
  - Programs are easier to prove correct.
  - Historically, imperative programming had to refer to Turing machines to talk about state, however there are very good techniques now.

pure functional languages

- **Functions are first-class values:**
  - Can be returned as the value of an expression.
  - Can be passed as an argument.
  - Can be put in a data structure as a value.
  - Unnamed functions exist as values.

pure functional languages

- **a higher-level language:**
  - All storage management is implicit:

    we don't have to think about how program state maps to a computer's memory (or at least, not until we want to write super-efficient code). That means no assignment, no new/free calls to manage our own memory, and no pointers.
  - The state of a computation is much easier to think about.

FP and Formal Proofs

Consider:

if `op` is commutative, then
`(foldr op id lst) = (foldl op id lst)`
for all `op, id, lst`.

is it true?
how would one prove it?

FP and Formal Proofs

Given

```
(define (len lst)
  (if (empty? lst))
      0
      (+ 1 (len (rest lst)))))
```

and

```
(define (append lst1 lst2)
  (if (empty? lst1))
      lst2
      (cons (first lst1)
            (append (rest lst1) lst2))))
```

Prove: for all lists `l1, l2`

`(len (append l1 l2)) = (+ (len l1) (len l2))`

FP and Formal Proofs

Need to establish some axioms / facts first:

If `lst` is a list then

1. `lst = ()`, or
2. exist element `X` and list `Y`, s.t.
   `lst = (cons X Y)`

FP and Formal Proofs

Need to establish some axioms / facts first:

```
0. (define len
1.  (lambda (lst)
2.    (if (empty? lst))
3.        0
4.        (+ 1 (len (rest lst)))))
```

```
[1] (len '()) = 0                          [lines 2,3]
[2] (len (cons X Y)) = (+ 1 (len Y))       [lines 2,4]
```

FP and Formal Proofs

Need to establish some axioms / facts first:

```
0. (define append
1.   (lambda (lst1 lst2)
2.     (if (empty? lst1))
3.         lst2
4.         (cons (first lst1)
5.               (append (rest lst1) lst2))))
```

```
[3]  (append '() l2) = l2     [lines 2,3]
[4]  (append (cons X Y) l2)
   = (cons X (append Y l2))   [lines 2-5]
```

FP and Formal Proofs

Now, given that If `lst` is a list then

1. `lst = ()`, or
2. exist element `X` and list `Y`, s.t.
   `lst = (cons X Y)`

and the following four facts:

```
[1] (len '()) = 0
[2] (len (cons X Y)) = (+ 1 (len Y))
[3]  (append '() l2) = l2
[4]  (append (cons X Y) l2)
   = (cons X (append Y l2))
```

and assuming + performs correct addition,

Prove: for all lists `l1, l2`

```
(len (append l1 l2)) = (+ (len l1) (len l2))
```

Recursion $\Leftrightarrow$ Induction

Prove:

for all lists `l1, l2`

`(len (append l1 l2)) = (+ (len l1) (len l2))`

Proof by structural induction on the list `l1`.

```
Case:    [l1 is ()]

  (len (append l1 l2))
= (len (append () l2))    [case]
= (len l2)                [3]
= (+ 0 (len l2))          [arith]
= (+ (len ()) (len l2))   [1]
= (+ (len l1) (len l2))   [case]
```

Recursion $\Leftrightarrow$ Induction

prove:

for all lists `l1, l2`

```
(len (append l1 l2)) = (+ (len l1) (len l2))
```

```
 Case:     [l1 = (cons X Y)]

 IH: Assume
  (len (append Y l2)) = (+ (len Y) (len l2))

   (len (append l1 l2))
 = (len (append (cons X Y) l2))    [case]
 = (len (cons X (append Y l2)))    [4]
 = (+ 1 (len (append Y l2)))       [2]
 = (+ 1 (+ (len Y) (len l2)))      [IH]
 = (+ (+ 1 (len Y)) (len l2))      [arith]
 = (+ (len (cons X Y)) (len l2))   [2]
 = (+ (len l1) (len l2))           [case]
```