# CIS 343 - Structure of Programming Languages

Nathan Bowman

Based on slides by Ira Woodring

---

## Lists and Let (Follows the Sebesta Text Chapter 15)

We investigate common instructions for working with lists

Later, we add more syntactic sugar in the form of `let` statement

When evaluating statements we need to be careful we are conveying the correct semantics. For instance:

```
1 ]=> (list a b c d)

;Unbound variable: d
;To continue, call RESTART with an option number:
; (RESTART 3) => Specify a value to use instead of d.
; (RESTART 2) => Define d to a given value.
; (RESTART 1) => Return to read-eval-print level 1.

2 error>
```

What is wrong here?

We are calling a function with parameters. The parameters must be bound before the function can be called.

a, b, c, and d don't have any bindings.

What we really want to do is this:

```
(list (quote a) (quote b) (quote c) (quote d))

;;; OR THE SHORTCUT:
(list 'a 'b 'c 'd)
```

The QUOTE and ' shortcut are functions that cause the parameter not to be evaluated, but to instead be used as is.

We must be very careful when we are programming that we QUOTE parameters we don't wish to be evaluated.

Since LISP was created to process lists, it makes sense that there are quite a few built-in functions for the manipulation of lists.

Two of the most common are CAR and CDR.

CAR returns the first element of a list, CDR returns all but the first.

Don't try to assign meaning to these names; they are holdovers from instructions on the IBM 704 machines LISP was originally created for. They have to do with accessing parts of memory locations.

```
2=> (define l (list 'apple 'banana 'grape 'orange))

;Value: l

2=> (car l)

;Value: apple

2=> (cdr l)

;Value 19: (banana grape orange)
```

To get the second element in a list, we could do the following:

```
(car (cdr l))
```

We could even create a function (from book):

```
(define (second a_list) (car (cdr a_list)))
(second l)
```

There are shortcuts as well:

```
CAAR = (car (car E))
CADR = (car (cdr E))
```

Others exist as well.

The LET statement is used to create a local binding. Once the LET function has created a binding in some local scope the binding cannot be rebound.

From the book, an example to compute quadratic roots:

```
(define (quadratic_roots a b c)
  (let (
    (root_part_over_2a
                (/ (sqrt (- (* b b) (* 4 a c))) (* 2 a)))
    (minus_b_over_2a (/ (- 0 b) (* 2 a)))
  )
  (list (+ minus_b_over_2a root_part_over_2a)    ; here
        (- minus_b_over_2a root_part_over_2a))   ; here
  )
)
```

We created two bindings and provided an expression inside of which we could use the named bindings.

`root_part_over_2a` and `minus_b_over_2a` are available only in lines marked `here`

Once again, this is merely syntactic sugar around `lambda`

The purpose of `let` is to

- create anonymous function, and
- call anonymous function with given arguments

in one instruction

```
(let ((x 5) (y 10))
  (* x y))
```

```
((lambda (x y)
   (* x y))
   5 10)
```

Both of the above are equivalent

Conceptually, it may be more helpful to think of `let` as "set these variables with local scope"

There is nothing wrong with using let, but be aware of how simple Scheme really is under the hood -- it is an extremely small and orthogonal language