# Syntactic Forms

`if`, `begin`, `or`, `and` are useful **syntactic forms**.

They have *lazy evaluation*, i.e., their subexpressions are not evaluated until required.

Let's look at lazy evaluation and how to exploit it.

```
(if (= n 0)
     (display "oops")
     (/ 1 n))
```

`if` is evaluated left to right. The "else part" is only evaluated as necessary, so `(/ 1 n)` is only evaluated if the conditional expression is false.

Imagine if `if` were implemented as a procedure. We'd be in trouble!

```
(begin
    (display "this is line 1 of the message")
    (display "this is line 2 of the message")
    #f
)
```

`begin` evaluates it subexpressions from left to right and returns the value of the last subexpression.



# Syntactic Forms (cont.)

```
(or) => #f
(or (= 0 1) (= 0 2) (= 0 0))  => #t
(or #f) => #f
(or #f #t) => #t
(or #f 'a #f) => a   (treated as #t in a conditional)
```

`or` evaluates its subexpressions from left to right until either (a) one expression is true, or (b) no more expressions are left. In case (a), the value is true, in (b) the value is false.

**Important subtlety:** Every Scheme object is considered to be either true or false by conditional experssions and by the procedure `not`. Only `#f` (i.e., ()) is considered false; **all other objects are considered true**.

```
(and) => #t
(and (= 0 0) (= 0 1) (= 0 2)) => #f
(and #f) => #f
(and #t #t) => #t
(and #t #f) => #f
(and 'a 'b 'c) => c  (treated as #t in a conditional)
```

`and` evaluates its subexpressions from left to right until (a) one expression is false, or (b) no more expressions are left. In case (a), the value is false, in (b) the value is true.

## Clever Exploitation of Syntactic Forms and Lazy Evaluation

```
(define (validate-bindings expr bindings)
    (cond ((...) ...)
          ((...) ...)
          ((symbol? expr)
             (debug-display "Symbol:" expr)
             (or (get-binding expr bindings)
                 (builtin? expr)
                 (begin
                    (display-error 'unbound expr)
                     #f
                 )
             )
         ((...) ...)

   etc.
)
```



## When Lazy Evaluation isn't our friend

Problem: Sometimes lazy evaluation works *against* you.

Challenge with `validate-bindings` is that it's a predicate function, so it must return `#t/()` depending upon whether the expression has valid bindings, but you must go through the *entire* list, even after you generate your first (). How do you do this?

Hint: There is a construct in Scheme that forces evaluation of a series of expressions before performing some operation on it. (There are many, actually?) Let's think of one we've seen in class and use it.

## Summary: Functional Pgming

- Pure functional languages:
  - Referential transparency
  - No assignment
  - No iteration, only recursion
  - Implicit storage management (garbage collection)
  - Functions are values
- $\lambda$-calculus
- LISP, Common LISP, Scheme
- Built-In Procedures
- Lists (cons cells, proper/improper)
- Read-eval-print loop
- Inhibiting + Activating evaluation (quote, eval)
- Procedure definition and lambda expressions



- Conditionals (if, cond)
- Equality Checking (eq?, =, equal?, eqv?)
- Recursion (practice, practice)
- Efficiency Concerns
  - helper procedures
  - let, let*, ...
  - accumulators
- Higher-order functions (map, apply, reduce)
- Passing Procedures, Returning Procedures
- Anonymous Procedures
- Syntactic Forms and Lazy Evaluation