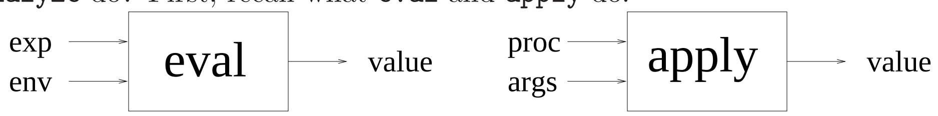
MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1999

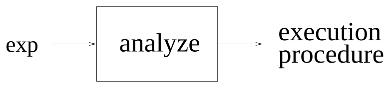
Recitation – Wednesday, April 14

1. Analyzing an Expression before Evaluation

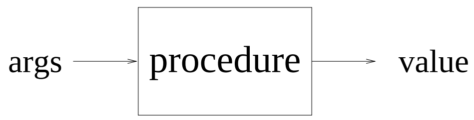
We briefly saw this idea of using `analyze` to cut down on the amount of work we do at run time. What does `analyze` do? First, recall what `eval` and `apply` do.



Consider the form of `analyze`:



Ok, so what's an `execution procedure`? Compare it to what we know about regular procedures:



- Procedures pack up computation.
- Can call multiple times w/ different arguments.
- Don't have to be re-write the procedure each time.

- Execution Procedures pack up computation.
- Can call multiple times w/ different environments.
- Don't have to re-analyze the expression each time.

Using this method, we `analyze` an expression only once but can `evaluate` it many times, with respect to different environments.

2. Code for Analyze

```
(define (eval exp env) ((analyze exp) env))

(define (analyze exp)
  (cond ((self-evaluating? exp) (analyze-self-evaluating exp))
        ((variable? exp) (analyze-variable exp))
        ((definition? exp) (analyze-definition exp))
        ((lambda? exp) (analyze-lambda exp))
        ((cond? exp) (analyze (cond->if exp)))
        ((application? exp) (analyze-application exp))))

(define (analyze-self-evaluating exp) (lambda (env) exp))

(define (analyze-variable exp)
  (lambda (env)
     (lookup-variable-value exp env)))

(define (analyze-lambda exp)
  (let ((bproc (analyze (lambda-body exp))))
    (lambda (env)
      (make-procedure (lambda-parameters exp) bproc env))))
```

3. An Example

Consider evaluating the following expressions with and without using `analyze`.

```
(define foo (lambda () (+ 1 2)))
```

```
(foo)
```

```
(foo)
```

4. Adding `and` to the Evaluator

Recall that `and` is a special form that takes an arbirary number of arguments. `And` evalutes each argument in turn until one of its arguments is false, in which case it returns false. For example,

```
(and)                   ==>  #t
(and #t)                ==>  #t
(and #t #f (/ 1 0))     ==>  #f
```

Write the function `and?` to see if an expression is an `and`.

```
(define (and? exp)

)
```

Consider adding `and` to the Evaluator. Write a version of `eval-and` that does not do any desugaring.

```
(define (eval-and exp env)

)
```

Write a version of `eval-and` that desugars into an if-statement.

```
(define (eval-and exp env)

)
```