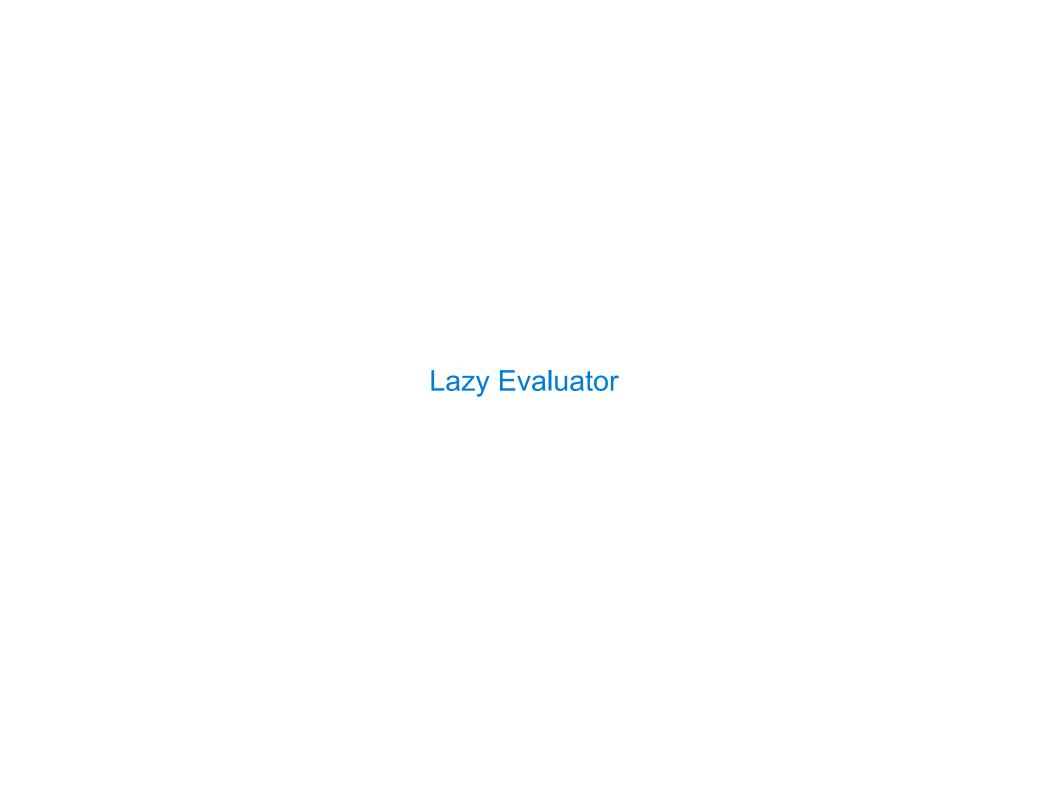


Implementing Streams with Delay and Force

```
A promise is an expression, along with an environment in which to evaluate it
Delaying an expression creates a promise to evaluate it later in the current environment
Forcing a promise returns its value in the environment in which it was defined
scm> (define promise (let ((x 2)) (delay (list (+ x 1)) ) ))
     (define promise (let ((x 2)) (lambda () (list (+ x 1)) ))
scm> (define x 5)
                               (define-macro (delay expr) `(lambda () ,expr))
scm> (force promise)
                               (3)
A stream is a list, but the rest of the list is computed only when forced:
scm> (define ones (cons-stream 1 ones))
(1 . #[promise (not forced)])
                               (define-macro (cons-stream a b) `(cons ,a (delay ,b)))
(1 . (lambda () ones))
                                        (cdr-stream s) (force (cdr s)))
```



Lazy Evaluation

When a procedure is applied:

• Built-in: The arguments are evaluated and the primitive procedure is applied to them

User-Defined: All arguments are delayed

When an if expression is evaluated:

• Predicate: Must be fully evaluated to determine which sub-expression to evaluate next

• Consequent/Alternative: Is evaluated, but call expressions within it are eval'd lazily

(Demo)