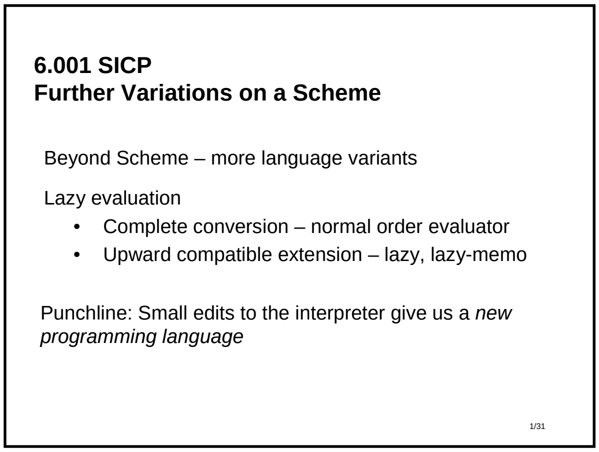


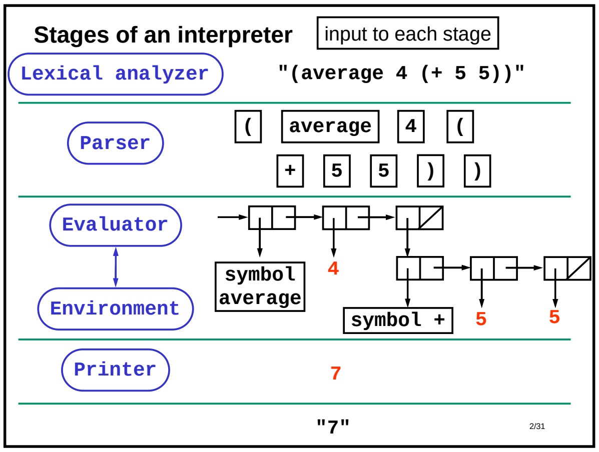
6.001 SICP
Further Variations on a Scheme

Beyond Scheme – more language variants

Lazy evaluation
- Complete conversion – normal order evaluator
- Upward compatible extension – lazy, lazy-memo

Punchline: Small edits to the interpreter give us a *new programming language*

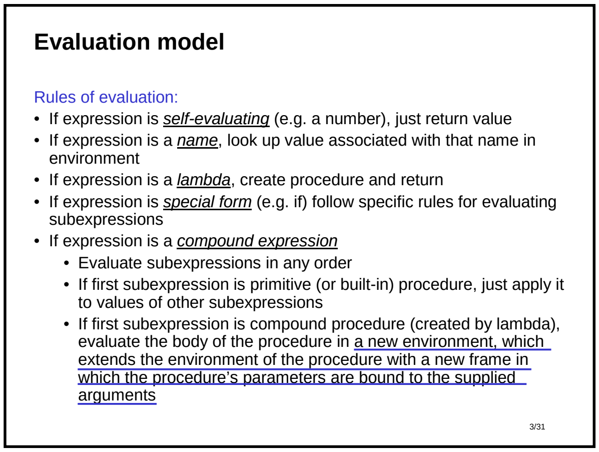
1/31



Stages of an interpreter

input to each stage

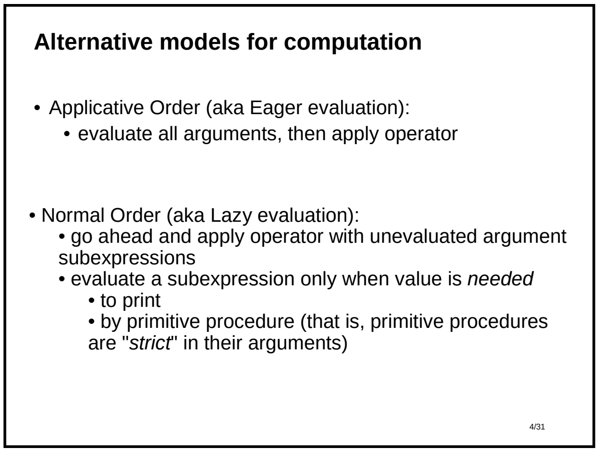
2/31



Evaluation model

Rules of evaluation:
- If expression is *self-evaluating* (e.g. a number), just return value
- If expression is a *name*, look up value associated with that name in environment
- If expression is a *lambda*, create procedure and return
- If expression is *special form* (e.g. if) follow specific rules for evaluating subexpressions
- If expression is a *compound expression*
  - Evaluate subexpressions in any order
  - If first subexpression is primitive (or built-in) procedure, just apply it to values of other subexpressions
  - If first subexpression is compound procedure (created by lambda), evaluate the body of the procedure in a new environment, which extends the environment of the procedure with a new frame in which the procedure's parameters are bound to the supplied arguments

3/31

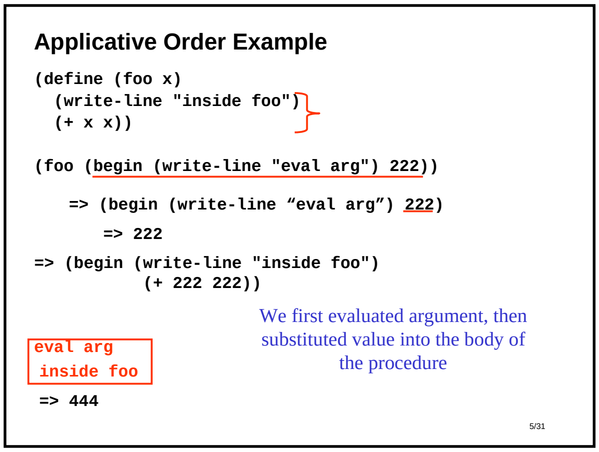


Alternative models for computation

- Applicative Order (aka Eager evaluation):
  - evaluate all arguments, then apply operator

- Normal Order (aka Lazy evaluation):
  - go ahead and apply operator with unevaluated argument subexpressions
  - evaluate a subexpression only when value is *needed*
    - to print
    - by primitive procedure (that is, primitive procedures are "*strict*" in their arguments)

4/31



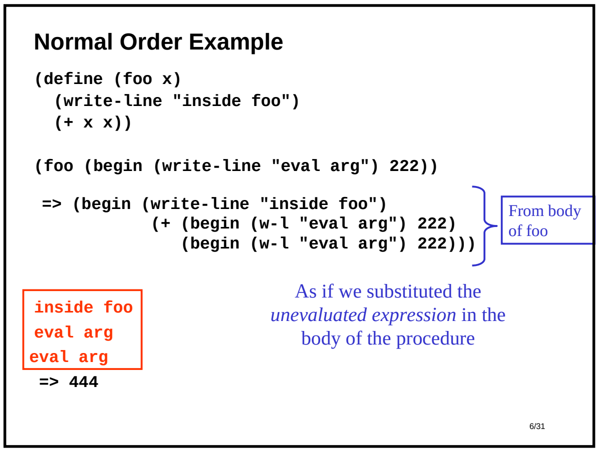
Applicative Order Example

```
(define (foo x)
  (write-line "inside foo")
  (+ x x))

(foo (begin (write-line "eval arg") 222))

   => (begin (write-line “eval arg”) 222)

      => 222

=> (begin (write-line "inside foo")
          (+ 222 222))
```

We first evaluated argument, then substituted value into the body of the procedure

```
eval arg
 inside foo
 => 444
```

5/31



Normal Order Example

```
(define (foo x)
  (write-line "inside foo")
  (+ x x))

(foo (begin (write-line "eval arg") 222))

 => (begin (write-line "inside foo")
           (+ (begin (w-l "eval arg") 222)
              (begin (w-l "eval arg") 222)))
```

From body of foo

As if we substituted the *unevaluated expression* in the body of the procedure

```
 inside foo
 eval arg
eval arg
 => 444
```

6/31

Applicative Order vs. Normal Order

```
(define (foo x)
  (write-line "inside foo")
  (+ x x))

(foo (begin (write-line "eval arg") 222))
```

Applicative order

```
eval arg
 inside foo
```

Think of as substituting values for variables in expressions

Normal order

```
 inside foo
 eval arg
eval arg
```

Think of as expanding expressions until only involve primitive operations and data structures

Normal order (lazy evaluation) versus applicative order

- How can we change our evaluator to use normal order?
  - Create “delayed objects” – expressions whose evaluation has been deferred
  - Change the evaluator to force evaluation only when needed
- Why is normal order useful?
  - What kinds of computations does it make easier?

Mapply – the original version

```
(define (mapply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure
            procedure
            arguments))
      ((compound-procedure? procedure)
       (eval-sequence
        (procedure-body procedure)
        (extend-environment
            (procedure-parameters procedure)
            arguments
            (procedure-environment procedure))))
      (else (error "Unknown procedure" procedure))))
```

Actual values

Actual values

How can we implement lazy evaluation?

```
(define (l-apply procedure arguments env)  ; changed
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure
           procedure
          (list-of-arg-values arguments env)))
        ((compound-procedure? procedure)
         (l-eval-sequence
           (procedure-body procedure)
          (extend-environment
            (procedure-parameters procedure)
            (list-of-delayed-args arguments env)
           (procedure-environment procedure))))
        (else (error "Unknown proc" procedure))))
```

Delayed expressions

Need to convert to actual values

Delayed Expressions

Need to create delayed version of arguments that will lead to values

Lazy Evaluation – `l-eval`

- Most of the work is in **`l-apply`**; need to call it with:
  - actual value for the operator
  - just expressions for the operands
  - the environment...

```
(define (l-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ...
        ((application? exp
         (l-apply (actual-value (operator exp) env)
                  (operands exp)
                  env))
        (else (error "Unknown expression" exp))))
```

Remember – this is just tree structure!!

Meval versus L-Eval

```
(define (meval exp env)
  (cond ((self-evaluating? exp) exp)
        …
       ((cond? exp) (meval (cond->if exp) env))
       ((application? exp)
        (mapply (meval (operator exp) env)
                (list-of-values (operands exp) env)))
       (else (error "Unknown expression type -- EVAL" exp))))
 (define (l-eval exp env)
   (cond ((self-evaluating? exp) exp)
         ...
         ((cond? exp)
         ((application? exp
          (l-apply (actual-value (operator exp) env)
                          (operands exp)
                         env))
         (else (error "Unknown expression" exp))))
```

Actual vs. Delayed Values

```
(define (actual-value exp env)
  (force-it (l-eval exp env)))

(define (list-of-arg-values exps env)
  (if (no-operands? exps) '()
    (cons (actual-value (first-operand exps) env)
          (list-of-arg-values (rest-operands exps)
                              env))))
```

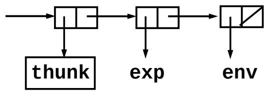
Used when applying a primitive procedure

```
(define (list-of-delayed-args exps env)
  (if (no-operands? exps)
      '()
      (cons (delay-it (first-operand exps) env)
            (list-of-delayed-args (rest-operands exps)
                                  env))))
```

Used when applying a compound procedure

Representing Thunks

- *Abstractly* – a **`thunk`** is a "promise" to return a value when later needed ("forced")
- *Concretely* – our representation:



Thunks – delay-it and force-it

```
(define (delay-it exp env) (list 'thunk exp env))
(define (thunk? obj) (tagged-list? obj 'thunk))
(define (thunk-exp thunk) (cadr thunk))
(define (thunk-env thunk) (caddr thunk))

(define (force-it obj)
  (cond ((thunk? obj)
         (actual-value  (thunk-exp obj)
                        (thunk-env obj)))
        (else obj)))

(define (actual-value exp env)
  (force-it (l-eval exp env)))
```

Memo-izing evaluation

- In lazy evaluation, if we reuse an argument, have to reevaluate each time
- In usual (applicative) evaluation, argument is evaluated once, and just referenced
- Can we keep track of values once we've obtained them, and avoid cost of reevaluation?

Sidebar on memoization

- Idea of memoization is for a procedure to remember if it has been called with a particular argument(s) and if so to simply return the saved value
- Can have problems if mutation is allowed – works best for functional programming

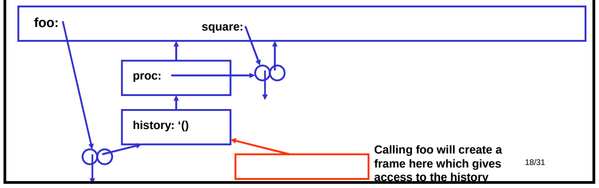
```
(define (memoize proc)
  (let ((history '()))
    (lambda (arg)
      (let ((already-there (in-history? arg history)))
        (if already-there
            (value already-there)
            (let ((return (proc arg)))
              (set! history
                    (insert-history return history))
              return))))))
```

Sidebar on memoization

```
(define (memoize proc)
  (let ((history '()))
    (lambda (arg)
      (let ((already-there (in-history? arg history)))
        (if already-there
            (value already-there)
            (let ((return (proc arg)))
              (set! history
                    (insert-history return history))
              return))))))
```

```
(define (square x) (* x x))

(define foo (memoize square))
```

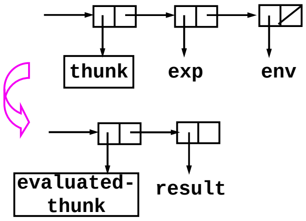
Store pairings of argument values and associated procedure values in history, e.g. an A-list



Calling foo will create a frame here which gives access to the history

Memo-izing Thunks

- *Idea*: once thunk **exp** has been evaluated, remember it
- If value is needed again, just return it rather than recompute
- *Concretely* – mutate a **thunk** into an **evaluated-thunk**



Why mutuate? – because other names or data structures may point to this thunk!

Thunks – Memoizing Implementation

```
(define (evaluated-thunk? obj)
  (tagged-list? obj 'evaluated-thunk))
(define (thunk-value evaluated-thunk)
  (cadr evaluated-thunk))

(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (actual-value (thunk-exp obj)
                                     (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)
           (set-cdr! (cdr obj) '())
           result))
        ((evaluated-thunk? obj)  (thunk-value obj))
        (else obj)))
```

Lazy Evaluation – other changes needed

- Example – need actual predicate value in conditional if...

```
(define (l-eval-if exp env)
  (if (true? (actual-value (if-predicate exp) env))
      (l-eval (if-consequent exp) env)
      (l-eval (if-alternative exp) env)))
```

- Example – don't need actual value in assignment...

```
(define (l-eval-assignment exp env)
  (set-variable-value!
     (assignment-variable exp)
     (l-eval (assignment-value exp) env)
     env)
  'ok)
```

Summary of lazy evaluation

- This completes changes to evaluator
  - Apply takes a set of expressions for arguments and an environment
    - Forces evaluation of arguments for primitive procedure application
    - Else defers evaluation and unwinds computation further
    - Need to pass in environment since don't know when it will be needed
  - Need to force evaluation on branching operations (e.g. if)
  - Otherwise small number of changes make big change in behavior of language

Laziness and Language Design

- We have a dilemma with lazy evaluation
  - Advantage: only do work when value actually needed
  - Disadvantages
    - not sure when expression will be evaluated; can be very big issue in a language with side effects
    - may evaluate same expression more than once

- Memoization doesn't fully resolve our dilemma
  - Advantage: Evaluate expression at most once
  - Disadvantage: What if we *want* evaluation on each use?

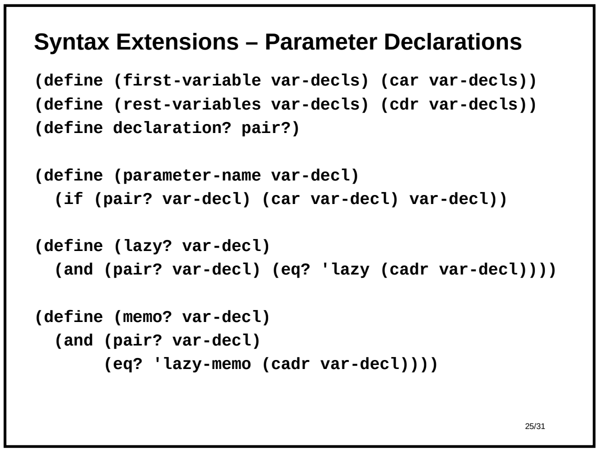
- Alternative approach: **give programmer control!**

Variable Declarations: `lazy` and `lazy-memo`

- Handle lazy and lazy-memo extensions in an upward-compatible fashion.;

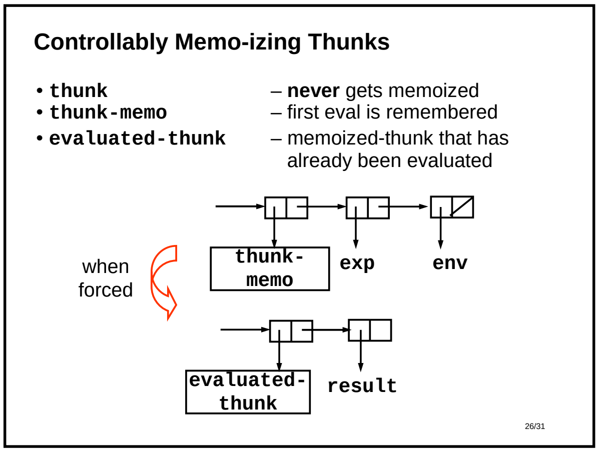
```
(lambda (a (b lazy) c (d lazy-memo)) ...)
```

- "a", "c" are usual variables (evaluated before procedure application)
- "b" is lazy; it gets (re)-evaluated each time its value is actually needed
- "d" is lazy-memo; it gets evaluated the first time its value is needed, and then that value is returned again any other time it is needed again.



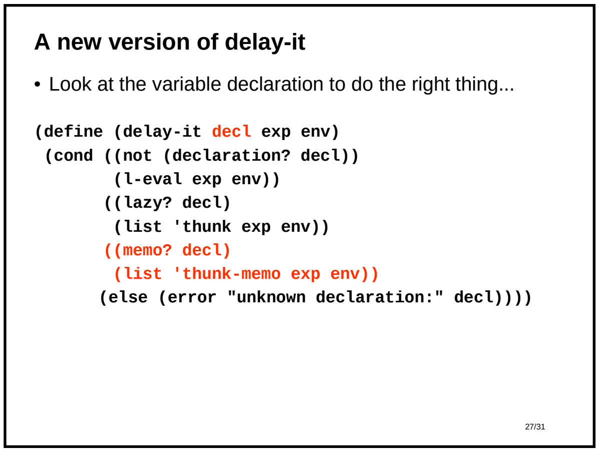
Syntax Extensions – Parameter Declarations

```
(define (first-variable var-decls) (car var-decls))
(define (rest-variables var-decls) (cdr var-decls))
(define declaration? pair?)

(define (parameter-name var-decl)
  (if (pair? var-decl) (car var-decl) var-decl))

(define (lazy? var-decl)
  (and (pair? var-decl) (eq? 'lazy (cadr var-decl))))

(define (memo? var-decl)
  (and (pair? var-decl)
       (eq? 'lazy-memo (cadr var-decl))))
```



Controllably Memo-izing Thunks

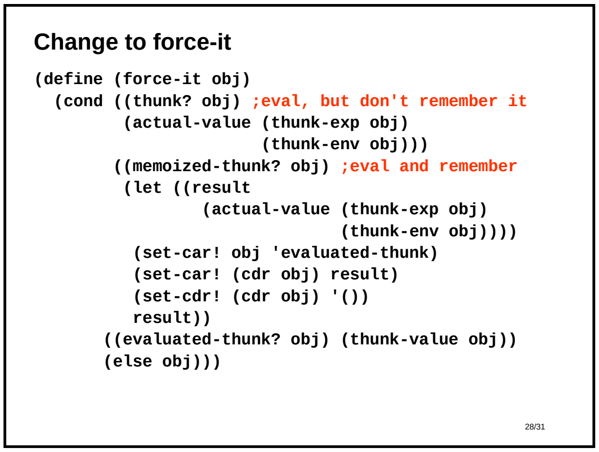
- **thunk** – **never** gets memoized
- **thunk-memo** – first eval is remembered
- **evaluated-thunk** – memoized-thunk that has already been evaluated



A new version of delay-it

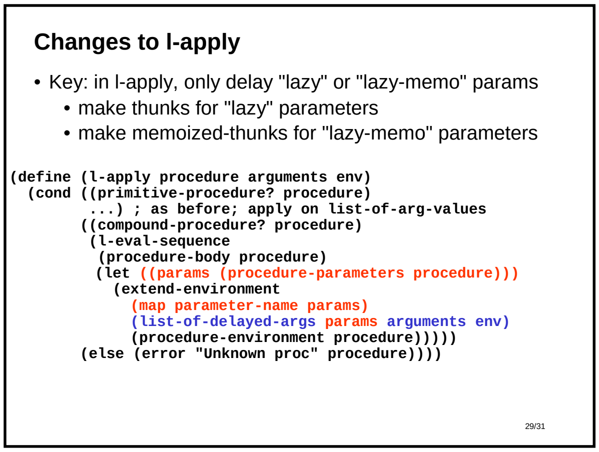
- Look at the variable declaration to do the right thing...

```
(define (delay-it decl exp env)
 (cond ((not (declaration? decl))
        (l-eval exp env))
       ((lazy? decl)
        (list 'thunk exp env))
       ((memo? decl)
        (list 'thunk-memo exp env))
       (else (error "unknown declaration:" decl))))
```



Change to force-it

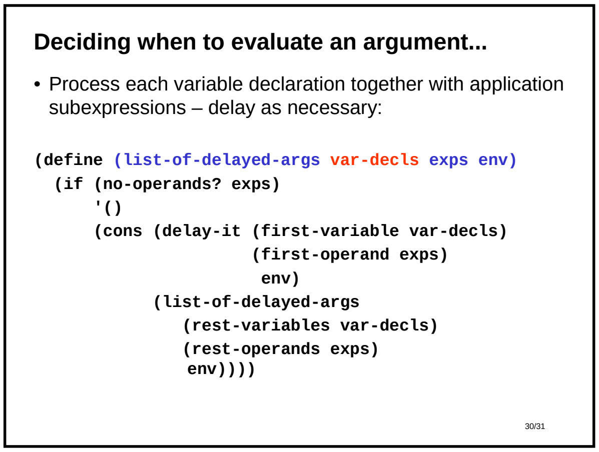
```
(define (force-it obj)
  (cond ((thunk? obj) ;eval, but don't remember it
         (actual-value (thunk-exp obj)
                       (thunk-env obj)))
        ((memoized-thunk? obj) ;eval and remember
         (let ((result
                (actual-value (thunk-exp obj)
                              (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)
           (set-cdr! (cdr obj) '())
           result))
       ((evaluated-thunk? obj) (thunk-value obj))
       (else obj)))
```



Changes to l-apply

- Key: in l-apply, only delay "lazy" or "lazy-memo" params
  - make thunks for "lazy" parameters
  - make memoized-thunks for "lazy-memo" parameters

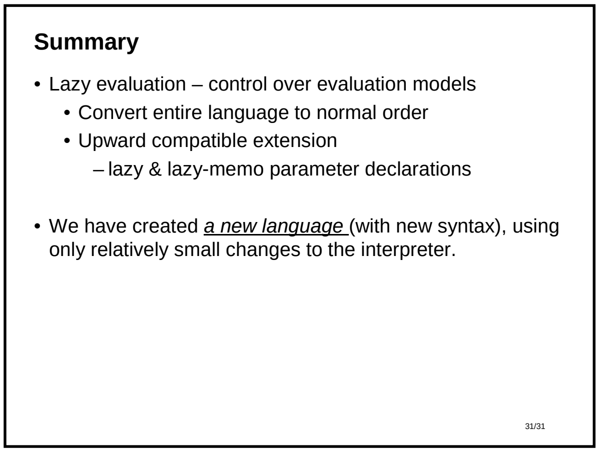
```
(define (l-apply procedure arguments env)
  (cond ((primitive-procedure? procedure)
         ...) ; as before; apply on list-of-arg-values
        ((compound-procedure? procedure)
         (l-eval-sequence
          (procedure-body procedure)
          (let ((params (procedure-parameters procedure)))
            (extend-environment
              (map parameter-name params)
              (list-of-delayed-args params arguments env)
              (procedure-environment procedure)))))
        (else (error "Unknown proc" procedure))))
```



Deciding when to evaluate an argument...

- Process each variable declaration together with application subexpressions – delay as necessary:

```
(define (list-of-delayed-args var-decls exps env)
  (if (no-operands? exps)
      '()
      (cons (delay-it (first-variable var-decls)
                      (first-operand exps)
                       env)
            (list-of-delayed-args
               (rest-variables var-decls)
               (rest-operands exps)
                env))))
```



Summary

- Lazy evaluation – control over evaluation models
  - Convert entire language to normal order
  - Upward compatible extension
    - – lazy & lazy-memo parameter declarations

- We have created ***a new language*** (with new syntax), using only relatively small changes to the interpreter.