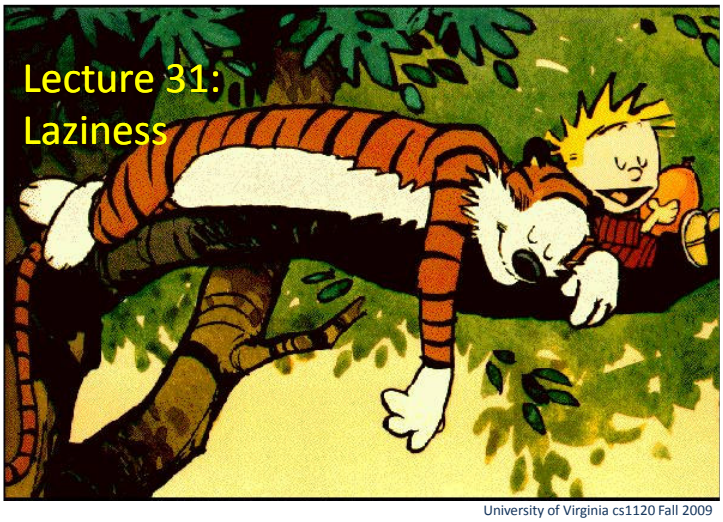


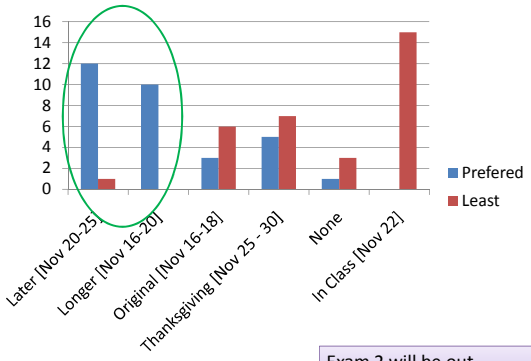
University of Virginia cs1120 Fall 2009
David Evans

Today's Class

- We can change the meaning of the language by changing the evaluation rules in our interpreter.
- Changing the **language** enables new ways of **programming**!
- Lazy evaluation changes the application rule to delay evaluating operand expressions
  - This eliminates the need for most special forms (all except **lambda**) and enables programming with "infinite lists"

Project details are now on the website. Team requests and project ideas are due next Thursday.

Exam 2 Poll



Nov 25 is actually a holiday! So...

Exam 2 will be out
Wed Nov 18, due Mon Nov 23

Eager Evaluation

To apply a constructed procedure:

1. **Construct a new environment**, whose parent is the environment of the applied procedure.
2. **For each procedure parameter, create a place** in the frame of the new environment with the name of the parameter. **Evaluate each operand expression in the environment of the application and initialize the value in each place to the value of the corresponding operand expression.**
3. **Evaluate** the body of the procedure **in the newly created environment.** The resulting value is the value of the application.

Our standard evaluation rule for applications is **eager:** we always evaluate all the operand expressions whether we need them or not.

Let's procrastinate...

Lazy Evaluation

- Don't evaluate expressions until their value is really needed
  - We might save work this way, since sometimes we don't need the value of an expression
  - We might change the meaning of some expressions, since the order of evaluation matters

Note: not a wise policy for problem sets (all answer values will always be needed!)

Lazy Examples

```
Charme> ((lambda (x) 3) (* 2 2))
3
LazyCharme> ((lambda (x) 3) (* 2 2))
3
Charme>((lambda (x) 3) (car 3))
error: car expects a pair, applied to 3
LazyCharme> ((lambda (x) 3) (car 3))
3
Charme> ((lambda (x) 3) (loop-forever))
no value – loops forever
LazyCharme> ((lambda (x) 3) (loop-forever))
3
```

(Assumes extensions from ps7)

Laziness can be useful!

How do we make our evaluation rules *lazier*?

To apply a constructed procedure:

1. **Construct a new environment**, whose parent is the environment of the applied procedure.
2. **For each procedure parameter, create a place** in the frame of the new environment with the name of the parameter.

   Put something (a "thunk") in that place that can be used later to get the value of that operand when it is needed.

   of the corresponding operand expression.
3. **Evaluate** the body of the procedure **in the newly created environment.** The resulting value is the value of the application.

Evaluation of Arguments

- Applicative Order ("eager evaluation")
  - Evaluate all subexpressions before applying
  - Scheme, original **Charme**, Java
- Normal Order ("lazy evaluation")
  - Evaluate arguments when the value is needed
  - Algol60 (sort of), Haskell, Miranda, **LazyCharme**

"Normal" Scheme order is **not** "Normal Order"!

What do we need to delay evaluation?

Put something (a "thunk") in that place that can be used later to get the value of that operand when it is needed.

- Need to record everything we will need to evaluate the expression later: the expression to evaluate **and** the environment
- After evaluating the expression, record the result for reuse: only evaluate operand expression once, even if it is used many times

I Thunk I Can

```
class Thunk:
    def __init__(self, expr, env):
        self._expr = expr
        self._env = env
        self._evaluated = False
    def value(self):
        if not self._evaluated:
            self._value = forceEval(self._expr, self._env)
            self._evaluated = True
        return self._value
```

Lazy Application

```
def evalApplication(expr, env):
    subexprvals = map (lambda sexpr: meval(sexpr, env), expr)
    return mapply(subexprvals[0], subexprvals[1:])
```

```
def evalApplication(expr, env):
    # make Thunk object for each operand expression
    ops = map (lambda sexpr: Thunk(sexpr, env), expr[1:])
    return mapply(forceEval(expr[0], env), ops)
```

Forcing Evaluation

```
class Thunk:
  def __init__(self, expr, env):
    self._expr = expr
    self._env = env
    self._evaluated = False
  def value(self):
    if not self._evaluated:
      self._value = forceEval(self._expr, self._env)
      self._evaluated = True
    return self._value
```

```
def forceEval(expr, env):
  value = meval(expr, env)
  if isinstance(value, Thunk):
    return value.value()
  else:
    return value
```

What else needs to change?

Hint: where do we need *real* values, instead of Thunks?

Primitive Procedures

Option 1: redefine all primitives to work on thunks

Option 2: assume primitives need values of all their operands and evaluate them eagerly

Primitive Procedures

```
def deThunk(expr):
  if isThunk(expr):
    return expr.value()
  else:
    return expr

def mapply(proc, operands):
  if (isPrimitiveProcedure(proc)):
    operands = map (lambda op: deThunk(op), operands)
    return proc(operands)
  elif ...
```

We need the deThunk procedure because Python's lambda construct can only have an expression as its body (not an if statement)

If Expressions

We need to know the actual value of the predicate expression, to know how to evaluate the if expression.

```
def evalIf(expr,env):
  if meval(expr[1], env) != False:
    return meval(expr[2],env)
  else:
    return meval(expr[3],env)
```

```
def evalIf(expr,env):
  if forceEval(expr[1], env) != False:
    return meval(expr[2],env)
  else:
    return meval(expr[3],env)
```

Do we really need if special form?

- Eager evaluation: yes
  - If we tried to define if as a procedure, all its operand expressions are always evaluated
- Lazy evaluation: **no!**
  - We can define if just like a regular procedure

```
(define true (lambda (a b) a))
(define false (lambda (a b) b))
(define ifp (lambda p c a) (p c a))
```

Lazy Data Structures

```
(define cons
  (lambda (a b)
    (lambda (p)
      (if p a b))))

(define car
  (lambda (p) (p #t)))

(define cdr
   (lambda (p) (p #f)))
```

Note: for PS7, you define these as *primitives*, which do not evaluate lazily.

Using Lazy Pairs

```
(define cons
  (lambda (a b)
    (lambda (p)
      (if p a b))))
```

```
(define car
  (lambda (p) (p #t)))
(define cdr
   (lambda (p) (p #f)))
```

```
LazyCharme> (define pair (cons 3 bogus))
LazyCharme> pair
<Procedure ['p'] / ['if', 'p', 'a', 'b']>
LazyCharme> (car pair)
3
LazyCharme> (cdr pair)
Error: Undefined name: bogus
```

Infinite Lists

```
(define ints-from
   (lambda (n)
     (cons n (ints-from (+ n 1)))))
```

```
LazyCharme> (define allnaturals (ints-from 0))
LazyCharme> (car allnaturals)
0
LazyCharme> (car (cdr allnaturals))
1
LazyCharme> (car (cdr (cdr (cdr (cdr allnaturals)))))
4
```

Charge

- Don't let LazyCharme happen to you!
  - **PS7** is due Monday
  - **Project:** don't delay forming teams and getting started on your project
- Monday: guest lecture by Kinga Dobolyi on *Web Applications*

Ordinary men and women, having the opportunity of a happy life, will become more kindly and less persecuting and less inclined to view others with suspicion. The taste for war will die out, partly for this reason, and partly because it will involve long and severe work for all. Good nature is, of all moral qualities, the one that the world needs most, and good nature is the result of ease and security, not of a life of arduous struggle. **Modern methods of production have given us the possibility of ease and security for all; we have chosen, instead, to have overwork for some and starvation for others. Hitherto we have continued to be as energetic as we were before there were machines; in this we have been foolish, but there is no reason to go on being foolish forever.**

Bertrand Russell, *In Praise of Idleness,* 1932