Lecture 30: Laziness

CS150: Computer Science
University of Virginia
Computer Science

David Evans
http://www.cs.virginia.edu/evans

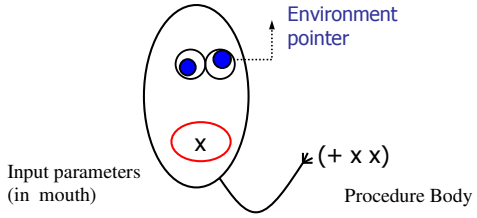
Menu

- Finishing Charme Interpreter
  - Application
- Lazy Evaluation


```
def meval(expr, env):
    if isPrimitive(expr):
        return evalPrimitive(expr)
    elif isConditional(expr):
        return evalConditional(expr, env)
    elif isLambda(expr):
        return evalLambda(expr, env)
    elif isDefinition(expr):
        evalDefinition(expr, env)
    elif isName(expr):
        return evalName(expr, env)
    elif isApplication(expr):
        return evalApplication(expr, env)
    else:
        evalError ("Unknown expression type: " + str(expr))
```


Implementing Procedures

What do we need to record?



Procedure Class

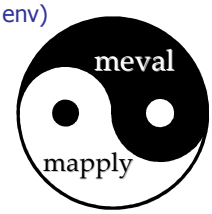
```
class Procedure:
    def __init__(self, params, body, env):
        self._params = params
        self._body = body
        self._env = env
    def getParams(self):
        return self._params
    def getBody(self):
        return self._body
    def getEnvironment(self):
        return self._env
```


Evaluating Lambda Expressions

```
def evalLambda(expr,env):
    assert isLambda(expr)
    if len(expr) != 3:
        evalError ("Bad lambda expression: %s" % str(expr))
    return Procedure(expr[1], expr[2], env)
```

Evaluating Applications

```
def meval(expr, env):
    ...
    elif isApplication(expr):
        return evalApplication(expr, env)
    else:
        evalError (...)
```



evalApplication

```
def evalApplication(expr, env):
    # To evaluate an application, evaluate all the subexpressions
    subexprvals = map (lambda sexpr: meval(sexpr, env), expr)
    # then, apply the value of the first subexpression to the rest
    return mapply(subexprvals[0], subexprvals[1:])
```

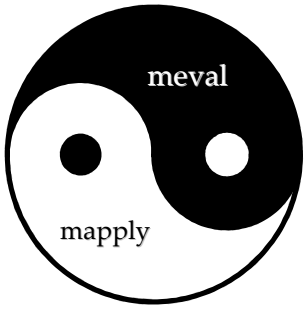

mapply

```
def mapply(proc, operands):
    if (isPrimitiveProcedure(proc)):
        return proc(operands)
    elif isinstance(proc, Procedure):
        params = proc.getParams()
        newenv =
        if len(params) != len(operands):
            evalError ("Parameter length mismatch ... ")
        for i in range(0, len(params)):
            
        return
    else:
        evalError("Application of non-procedure: %s" % (proc))
```


Implemented Interpreter!

What's missing?

Special forms:
  if, begin, set!
Primitive procedures:
  lots and lots
Built-in types:
  floating point numbers, strings, lists, etc.



"Surprise" Quiz

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
- 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!

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
(co-author of *Principia Mathematica*,
proved wrong by Gödel's proof)

How do we make our evaluation rules *lazier*?

Evaluation Rule 3: Application.

To **evaluate** an application,

a. **evaluate** all the subexpressions

b. **apply** the value of the first subexpression to the values of the other subexpressions.

a. evaluate the first subexpression, and **delay** evaluating the operand subexpressions until their values are needed.

Evaluation of Arguments

- Applicative Order ("eager evaluation")
  - Evaluate all subexpressions before apply
  - 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"!

Delaying Evaluation

- Need to record everything we will need to evaluate the expression later
- After evaluating the expression, record the result for reuse

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 primitives to work on thunks
- Option 2: assume primitives need values of all their arguments

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)

Conditionals

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

```
def evalConditional(expr, env):
    assert isConditional(expr)
    if len(expr) <= 2:
        evalError ("Bad conditional expression: %s" % str(expr))
    for clause in expr[1:]:
        if len(clause) != 2:
            evalError ("Bad conditional clause: %s" % str(clause))
        predicate = clause[0]
        result = meval(predicate, env)
        if not result == False:
            return meval(clause[1], env)
    evalError (...)
    return None
```

```
result = forceeval(predicate, env)
```


Charge

- Don't let Lazy Scheme happen to you!
  - PS7 is long and hard – don't wait to start it!
- Chapter 13: describes LazyCharme
- Wednesday:
  - Delayed lists in LazyCharme (if you want to seem really smart in class, read Chapter 13 before Wednesday!)
- Wednesday, Friday: Type Checking