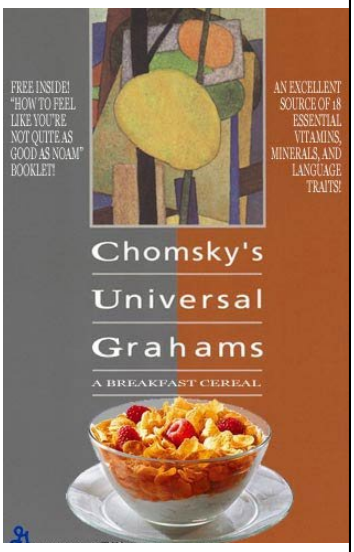
One-Slide Summary

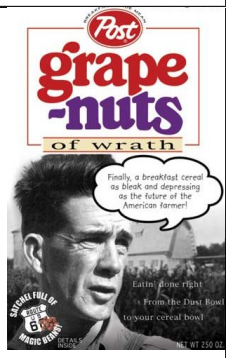
- Building an **interpreter** is a fundamental idea in computing. Eval and Apply are **mutually recursive**.
- The most complicated parts of **meval** are the handling of function abstraction (lambda) and function application.
- The most complicated part of **mapply** is handling a non-primitive procedure: create a new environment, add variables to that environment corresponding to the arguments, and then apply the procedure body in that new environment.
- In **lazy evaluation**, a value is not computed until it is needed. A **thunk** is a piece of code that performs a delayed computation.

Outline

- Eval
- Apply
- Lazy
- Thunk




```
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))
```



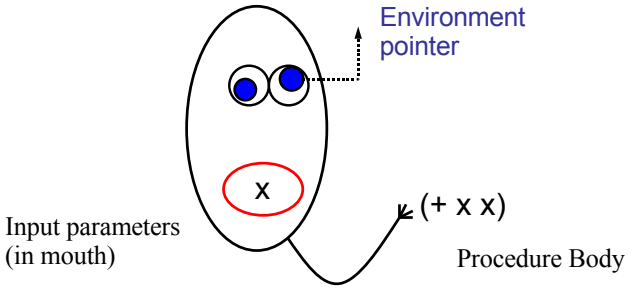
Conditionals

```
def evalConditional(expr, env):
    assert isConditional(expr)
    if len(expr) <= 2:
        evalError ("Bad ...")
    for clause in expr[1:]:
        if len(clause) != 2:
            evalError ("Bad ...")
        predicate = clause[0]
        result = meval(predicate, env)
        if not result == False:
            return meval(clause[1], env)
    evalError ("No ...")
    return None
```

Recall the conditional:
(cond ((< x 5) “small”)
(< x 10) “medium”)
(< x 99) “large”)))

Implementing Procedures

What do we need to record?



Procedure Class

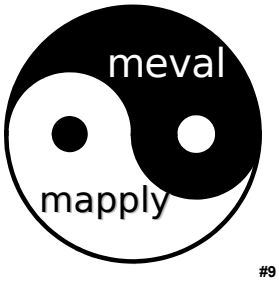
```
class Procedure:
    def __init__(self, params, body, env):
        self._params = params
        self._body = body
        self._env = env
    def getParams(self):
        return self._params
    def getBody(self):
        return self._body
    def getEnvironment(self):
        return self._env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

Evaluating Lambda Expressions

```
def evalLambda(expr,env):
    assert isLambda(expr)
    if len(expr) != 3:
        evalError ("Bad lambda ...")
    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 subexpr:
        meval(subexpr, env), expr)
    # then, apply the value of the
    # first subexpression to the rest
    return mapply(subexprvals[0], subexprvals[1:])
```


Liberal Arts Trivia: Geography

- This island nation in southeast Asia is located about 20 miles off the southern coast of India. It is home to 20 million people (mostly Sinhalese and Tamils), is a center of the Buddhist religion, and possesses rich tropical forests. during WWII it was used as a base for Allied forces against the Japanese Empire. It was known as Ceylon before 1972.

Liberal Arts Trivia: Italian Literature

- This Florentine poet of the Middle Ages is called *il Sommo Poeta*. His central work, the *Divinia Commedia*, is often considered the greatest literary work composed in the Italian language and is a masterpiece of world literature. He is often called the Father of the Italian Language: he wrote the *Commedia* in th early 14th century in a new language he called "Italian" based on the regional dialect of Tuscany with some Latin and other bits thrown in.
- Bonus: Who guides him in Hell and Purgatory?

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))
```


mapply

```
def mapply(proc, operands):
    if (isPrimitiveProcedure(proc)):
        return proc(operands)
    elif isinstance(proc, Procedure):
        params = proc.getParams()
        newenv = Environment(proc.getEnvironment())
        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))
```


mapply

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


mapply

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

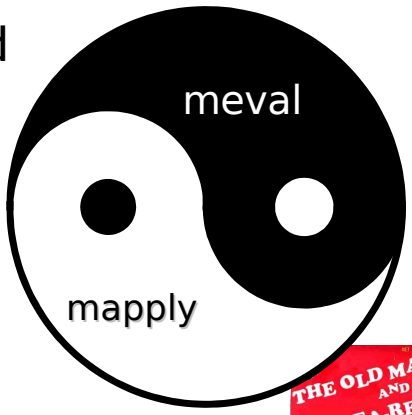

Implemented Interpreter!

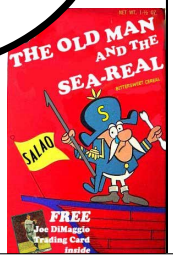
What's missing?

Special forms:
- if, begin, set!

Primitive procedures:
- lots and lots

Built-in types:
- floating point numbers, strings, lists, etc.





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*?

Original 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.

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.

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

Liberal Arts Trivia: Canadian Literature

- In this 1908 book, the title character is a talkative red-haired orphan. She moves to the village of Avonlea to live with farmers Matthew and Marilla Cuthbert. She becomes bosom friends with Diana Barry and has a complex relationship with Gilbert Blythe. Her vivid imagination and cheerful outlook often land her in trouble.
- Bonus: Name the setting's Canadian Province.

Liberal Arts Trivia: Neuroscience

- This medical visualization technique is most commonly used to visualize the internal structure and function fo the body. Notably, it uses no ionizing radiation, but instead uses powerful fields to align the hydrogen atoms in water in the body. Radiofrequency fields are used to alter the alignment of the hydrogen atoms, which can then be detected by a scanner. The process was first used on humans in 1977.

Evaluation of Arguments

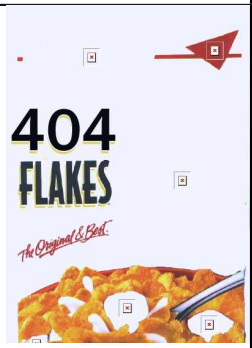
- Applicative Order ("eager evaluation")
  - Evaluate all subexpressions before apply
  - Scheme, original Charme, Java
- Normal Order ("lazy evaluation")
  - Evaluate arguments when the value is needed
  - Algol60 (sort of), Haskell, Miranda, LazyCharme

"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
- A thunk is a piece of code that performs a delayed computation

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)
```

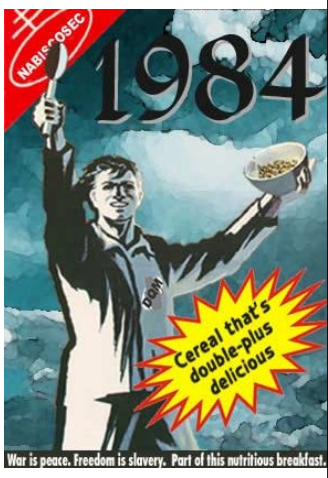

Forcing Evaluation

```
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
```

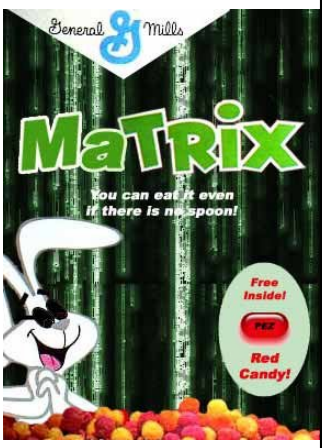

What else needs to change?

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

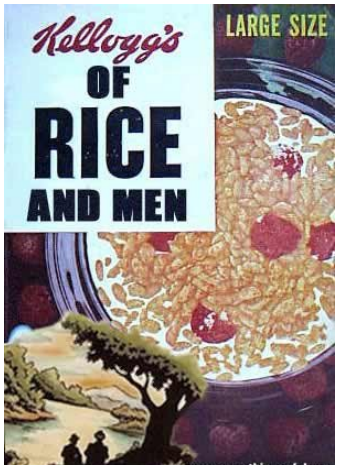
```
def deThunk(expr):
    # how am I different from forceeval?
    if isThunk(expr):
        return expr.value()
    else:
        return expr

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)

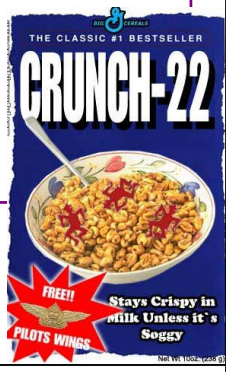
Lazy Conditionals

- 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
```

What do we need to change?




```
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)

Homework

- Read Chapter 13 for Wednesday
- PS 7 due Monday April 06
- Guest Lecture Monday April 06