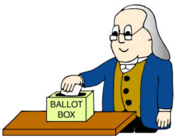
Menu

Checking Grades
Plans for Exam 2

- Completing the Charme Interpreter
- History of Object-Oriented Programming

Software Voting



Univac predicts big win for Eisenhower (1952)

"We do have people complain and say they don't get it, I completely understand what they're saying, but it's not something I can control."
– Sheri Iachetta, Charlottesville general registrar

Recap

To build an evaluator we need to:

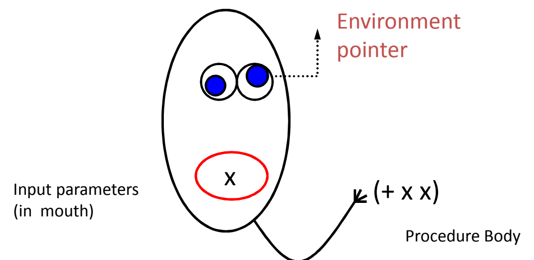
- Figure out how to represent data in programs
  - **Last class: program text, Environment**
  - **Today: Procedures, primitives**
- Implement the evaluation rules
  - For each evaluation rule, define a procedure that follows the behavior of that rule.
  - **Friday: application**
  - **Last class: definitions, names**
  - **Today: special forms, primitives**

```
def meval(expr, env):
    if isPrimitive(expr):
      return evalPrimitive(expr)
    elif isIf(expr):
      return evalIf(expr, env)
    elif isDefinition(expr):
      evalDefinition(expr, env)
    elif isName(expr):
      return evalName(expr, env)
    elif isLambda(expr):
      return evalLambda(expr, env)
    elif isApplication(expr):
      return evalApplication(expr, env)
    else:
      error ('Unknown expression ...)
```

How should we represent procedures?

(lambda (x) (+ x x))

Representing Procedures



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(se
        return self._
    def getEnviron
        return self._
    def __str__(se
        return '<Pro
```

```
def mapply(proc, operands):
    if (isPrimitiveProcedure(proc)): ...
    elif isinstance(proc, Procedure):
        params = proc.getParams()
        newenv = Environment(proc.getEnvironment())
        for i in range(0, len(params)):
            newenv.addVariable(params[i], operands[i])
        return meval(proc.getBody(), newenv)
```

Evaluating Lambda Expressions

parse("(lambda (a) (+ a 1))")[0]

['lambda', ['a'], ['+', 'a', '1']]

```
def isSpecialForm(expr, keyword):
    return isinstance(expr, list) and len(expr) > 0 and expr[0] == keyword

def isLambda(expr):
    return isSpecialForm(expr, 'lambda')

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)
```

Recap

```
def meval(expr, env):
    if isPrimitive(expr):
        return evalPrimitive(expr)
    elif isIf(expr):
        return evalIf(expr, env)
    elif isDefinition(expr):
        evalDefinition(expr, env)
    elif isName(expr):
        return evalName(expr, env)
    elif isLambda(expr):
        return evalLambda(expr, env)
    elif isApplication(expr):
        return evalApplication(expr, env)
    else:
        error ('Unknown expression ...)
```

Making Primitives

Evaluation Rule 1: Primitives

A *primitive* evaluates to its pre-defined value.

To implement a primitive, we need to give it its pre-defined value!

Primitives

```
def isPrimitive(expr):
    return (isNumber(expr) or isPrimitiveProcedure(expr))

def isNumber(expr):
    return isinstance(expr, str) and expr.isdigit()

def isPrimitiveProcedure(expr):
    return callable(expr)

def evalPrimitive(expr):
    if isNumber(expr):
        return int(expr)
    else:
        return expr
```

Making Primitive Procedures

```
def primitivePlus (operands):
    if (len(operands) == 0):
        return 0
    else:
        return operands[0] + primitivePlus (operands[1:])

def primitiveEquals (operands):
    checkOperands (operands, 2, '=')
    return operands[0] == operands[1]
```

```
def mapply(proc, operands):
    if (isPrimitiveProcedure(proc)):
        return proc(operands)
    elif isinstance(proc, Procedure):
        ...
```

To apply a primitive procedure, “just do it”.

So, what do we do with the primitives?

```
def initializeGlobalEnvironment():
    global globalEnvironment
    globalEnvironment = Environment(None)
    globalEnvironment.addVariable('true', True)
    globalEnvironment.addVariable('false', False)
    globalEnvironment.addVariable('+', primitivePlus)
    globalEnvironment.addVariable('-', primitiveMinus)
    globalEnvironment.addVariable('*', primitiveTimes)
    globalEnvironment.addVariable('=', primitiveEquals)
    globalEnvironment.addVariable('zero?', primitiveZero)
    globalEnvironment.addVariable('>', primitiveGreater)
    globalEnvironment.addVariable('<', primitiveLessThan)
```

Evaluation Rule 5: If

IfExpression
::= (if $Expression_{Predicate}$
$Expression_{Consequent}$
$Expression_{Alternate}$)

To evaluate an if expression:

(a) Evaluate $Expression_{Predicate}$.

(b) If it evaluates to a false value, the value of the if expression is the value of $Expression_{Alternate}$; otherwise, the value of the if expression is the value of $Expression_{Consequent}$.

Making Special Forms

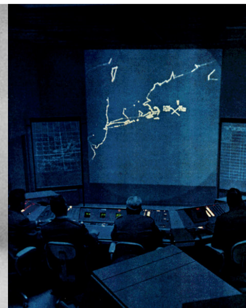
```
def isIf(expr):
    return isSpecialForm(expr, 'if')

def evalIf(expr,env):
    assert isIf(expr)
    if len(expr) != 4:
        evalError ('Bad if expression: %s' % str(expr))
    if meval(expr[1], env) != False:
        return meval(expr[2],env)
    else:
        return meval(expr[3],env)
```

All Done!

```
def meval(expr, env):
  if isPrimitive(expr):
    return evalPrimitive(expr)
  elif isIf(expr):
     return evalIf(expr, env)
  elif isDefinition(expr):
    evalDefinition(expr, env)
  elif isName(expr):
    return evalName(expr, env)
  elif isLambda(expr):
    return evalLambda(expr, env)
  elif isApplication(expr):
    return evalApplication(expr, env)
  else:
    error ('Unknown expression ...)
```

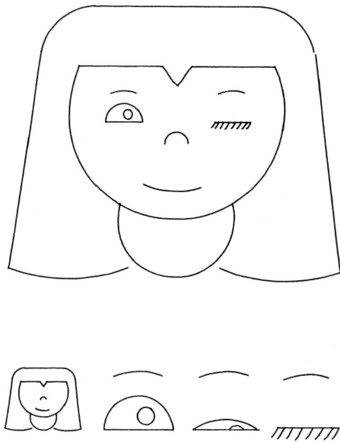
This is a universal programming language – we can define every procedure in Charme.

History of Object-Oriented Programming



Object-oriented programming is an exceptionally bad idea which could only have originated in California.
Edsger Dijkstra

Components in Sketchpad



Objects in Sketchpad

In the process of making the Sketchpad system operate, **a few very general functions were developed which make no reference at all to the specific types of entities on which they operate.** These general functions give the Sketchpad system the ability to operate on a wide range of problems. The motivation for making the functions as general as possible came from the desire to get as much result as possible from the programming effort involved. For example, the general function for expanding instances makes it possible for Sketchpad to handle *any* fixed geometry subpicture. The rewards that come from implementing general functions are so great that the author has become reluctant to write any programs for specific jobs. **Each of the general functions implemented in the Sketchpad system abstracts, in some sense, some common property of pictures independent of the specific subject matter of the pictures themselves.**

Ivan Sutherland,
Sketchpad: a Man-Machine Graphical Communication System, 1963

Simula

- Considered the first "object-oriented" programming language
- Language designed for *simula*tion by Kristen Nygaard and Ole-Johan Dahl (Norway, 1962)
- Had special syntax for defining classes that packages state and procedures together

Counter in Simula

```
class counter;
  integer count;
  begin
    procedure reset(); count := 0; end;
    procedure next();
        count := count + 1; end;
    integer procedure current();
        current := count; end;
  end
```

Does this have everything we need for "object-oriented programming"?

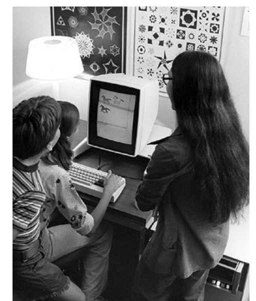
Object-Oriented Programming

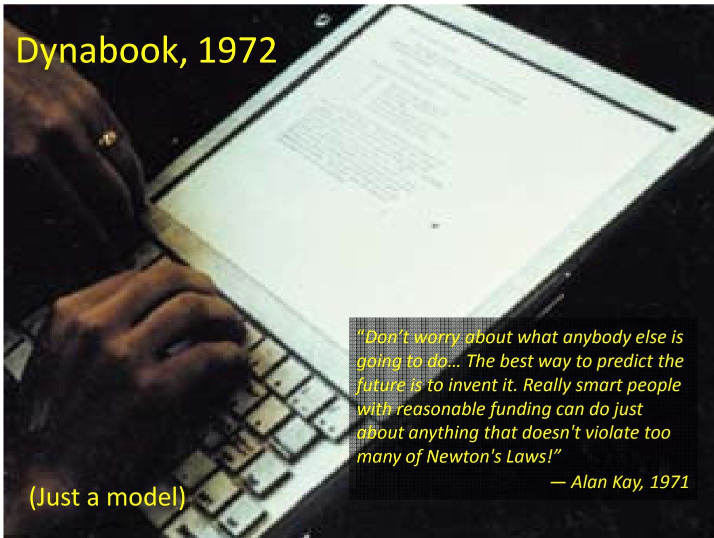
- Object-Oriented Programming is a *state of mind* where you program by *thinking about objects*
- It is difficult to reach that state of mind if your language doesn't have mechanisms for packaging state and procedures (Python has **class**, Scheme has lambda expressions)
- Other things can help: dynamic dispatch, inheritance, automatic memory management, mixins, good donuts, etc.

XEROX Palo Alto Research Center (PARC)

1970s:

- Bitmapped display
- Graphical User Interface
  - Steve Jobs paid $1M to visit and PARC, and returned to make Apple Lisa/Mac
- Ethernet
- First personal computer (Alto)
- PostScript Printers
- **Object-Oriented Programming**

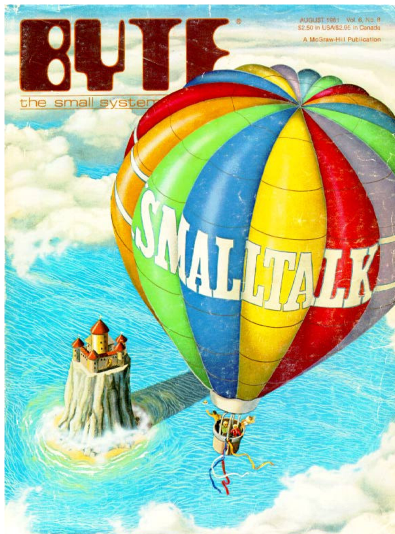




Dynabook 1972

- Tablet computer intended as tool for learning
- Alan Kay wanted children to program it also
- Hallway argument, Kay claims you could define "the most powerful language in the world in a page of code"

Proof: **Smalltalk**
Scheme is as powerful, but takes two pages
Before the end of the course, we will see an equally powerful language that fits in ¼ page



BYTE Magazine, August 1981

Smalltalk

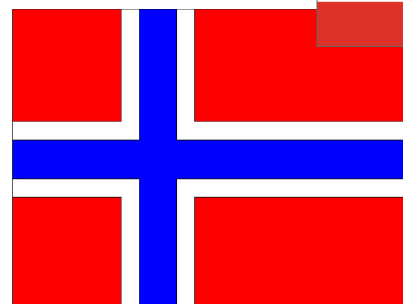
- Everything is an *object*
- Objects communicate by sending and receiving *messages*
- Objects have their own state (which may contain other objects)
- How do you do 3 + 4?

send the object **3** the message **"+ 4"**

Counter in Smalltalk

```
class name counter
  instance variable names count
  new count <- 0
  next count <- count + 1
  current ^ count
```

So, who really was the first object-oriented programmer?



Object-oriented programming is an exceptionally bad idea which could only have originated in California.
Edsger Dijkstra

By the word operation, we mean any process which alters the mutual relation of two or more things, be this relation of what kind it may. This is the most general definition, and would include all subjects in the universe. Again, it might act upon other things besides number, were objects found whose mutual fundamental relations could be expressed by those of the abstract science of operations... Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent.

Ada, Countess of Lovelace, around 1843

Charge

- Friday: changing the language rules
- PS7 due Monday

Remember to vote (early and often!) in the Exam 2 poll before 5pm Thursday.