Scheme in Python

Scheme in Python

- We'll follow the approach taken in the scheme in scheme interpreter for scheme in Python
  - http://cs.umbc.edu/courses/331/fall10/code/scheme/mcscheme/
- While is similar to that used by Peter Norvig
  - http://cs.umbc.edu/courses/331/fall10/code/python/scheme/
- It's a subset of scheme with some additional limitations
- We'll also look a some extensions

Key parts

- **S-expression** representation
- **parsing** input into s-expressions
- **Print** s-expression, serializing as text that can be read by (read)
- **Environment** representation plus defin-ing, setting and looking up variables
- **Function** representation
- **Evaluation** of an s-expression
- **REPL** (read, eval, print) loop

S-expression

- Scheme numbers are Python numbers
- Scheme symbols are Python strings
- Scheme strings are not supported, so simplify the parsing
- Scheme lists are python lists
  - No dotted pairs
  - No structure sharing or circular list
- #t and #f are True and False
- Other Scheme data types aren't supported

Parse: string=>s-expression

```
def read(s):
    "Read a Scheme expression from a string."
    return read_from(tokenize(s))

parse = read

def tokenize(s):
    "Convert a string into a list of tokens."
    return s.replace('(',' ( ').replace(')',' ) ').\
           replace('\n', ' ').strip().split()
```

Parse: string=>s-expression

```
def read_from(tokens):
    "Read an expression from a sequence of tokens."
    if len(tokens) == 0:
        raise SchemeError('EOF while reading')
    token = tokens.pop(0)
    if '(' == token:
        L = []
        while tokens[0] != ')':
            L.append(read_from(tokens))
        tokens.pop(0)                    # pop off ')'
        return L
    elif ')' == token:
        raise SchemeError('unexpected )')
    else:
        return atom(token)
```

Making atoms, loading files

```
def atom(token):
    "Numbers become numbers; every other token is a symbol."
    try: return int(token)
    except ValueError:
        try: return float(token)
        except ValueError:
            return Symbol(token)

def load(filename):
    "Reads expressions from file and evaluates them, returns void."
    tokens = tokenize(open(filename).read())
    while tokens:
        eval(read_from(tokens))
```

print reverses read

```
def to_string(exp):
    "Convert a Python object back into a Lisp-readable string."
    if isa(exp, list):
        return '('+' '.join(map(to_string, exp))+')'
    else:
        return str(exp)
```

Environments

- An environment will just be a Python dictionary of symbol:value pairs
- Plus an extra key called **outer** whose value is the enclosing (parent) environment

Env class

```
class Env(dict):
   "An environment: a dict of {'var':val} pairs, with an outer Env."
   def __init__(self, parms=(), args=(), outer=None):
      self.update(zip(parms,args))
      self.outer = outer
   def find(self, var):
      ”Returns the innermost Env where var appears."
      if var in self:  return self
      elif self.outer:  return self.outer.find(var)
      else:  raise SchemeError("unbound variable " + var)
   def set(self, var, val): self.find(var)[var] = val
   def define(self, var, val): self[var] = val
   def lookup(self, var): return self.find(var)[var]
```

Eval

```
def eval(x, env=global_env):
  "Evaluate an expression in an environment."
  if isa(x, Symbol): return env.lookup(x)
  elif not isa(x, list): return x
  elif x[0] == 'quote': return x[1]
  elif x[0] == 'if': return eval((x[2] if eval(x[1], env) else x[3]), env)
  elif x[0] == 'set!': env.set(x[1], eval(x[2], env))
  elif x[0] == 'define': env.define(x[1], eval(x[2], env))
  elif x[0] == 'lambda': return lambda *args: eval(x[2], Env(x[1],
   args, env))
  elif x[0] == 'begin': return [eval(exp, env) for exp in x[1:]] [-1]
  else:
    exps = [eval(exp, env) for exp in x]
    proc = exps.pop(0)
    return proc(*exps)
```

Representing functions

- This interpreter represents scheme functions as Python functions. Sort of.
- Consider evaluating
  - (define sum (lambda (x y) (+ x y)))
- This binds sum to the evaluation of
  - [‘lambda’, [‘x’,’y’], [‘+’, ‘x’, ‘y’]]
- Which evaluates the Python expression
  - lambda *args: eval(x[3], Env(x[2],args,env)
  - = <function <lambda> at 0x10048aed8>
- Which remembers values of x and env

Calling a function

- Calling a built-in function
  - (+ 1 2) => [‘+’, 1, 2]
  - => [<built-in function add>, 1, 2]
  - Evaluates <built-in function add>(1, 2)
- Calling a user defined function
  - (sum 1 2) => [‘sum’, 1, 2]
  - => [<function <lambda> at…>, 1, 2]
  - => <function <lambda> at…>(1, 2)
  - => eval([‘+’,’x’,’y’], Env([‘x’,’y’], [1,2], env))

repl()

```
def repl(prompt='pscm> '):
  "A prompt-read-eval-print loop"
  print "pscheme, type control-D to exit"
  while True:
    try:
      val = eval(parse(raw_input(prompt)))
      if val is not None: print to_string(val)
    except EOFError:
      print "Leaving pscm"
      break
    except SchemeError as e:
      print "SCM ERROR: ", e.args[0]
    except:
     print "ERROR: ", sys.exc_info()[0]

# if called as a script, execute repl()
if __name__ == "__main__":  repl()
```

Extensions

- Pscm.py has lots of shortcomings that can be addressed
- More data types (e.g., strings)
- A better scanner and parser
- Macros
- Functions with variable number of args
- Tail recursion optimization

Strings should be simple

- But adding strings breaks our simple approach to tokenization
- We added spaces around parentheses and then split on white space
- But strings can have spaces in them
- ☹
- The solution is to use regular expres-sions to match any of the tokens
- While we are at it, we can add more token types, ; comments, etc.

quasiquote

- Lisp and Scheme use a single quote char to make the following s-expression a literal
- The back-quote char (`) is like ' except that any elements in the following expression preceded by a , or ,@ are evaluated
- ,@ means the following expression should be a list that is “spliced” into its place

```
> 'foo
foo
> (define x 100)
> `(foo ,x bar)
(foo 100 bar)
```

```
> (define y '(1 2 3))
> `(foo ,@y bar)
(foo 1 2 3 bar)
```

; comments

- Lisp and Scheme treat all text between a semi-colon and the following newline as a comment
- The text is discarded as it is being read

Big hairy regular expression

```
r''' \s* (,@|[('`,)]|"(?:[\\].|[^\\"])*"|;.*|[^\s('"`,;)]*) (.*) '''
```

- Whitespace
- The next token alternatives:
  - ,@ — quasiquote ,@ token
  - [('`,)] — single character tokens
  - "(?:[\\].|[^\\"])*" — string
  - ;.* — comment
  - [^\s('"`,;)]*) — atom
- The characters after the next token

Reading from ports

```
class InPort(object):
  "An input port. Retains a line of chars."
  tokenizer = r'''\s*(,@|[('`,)]|"(?:[\\].|[^\\"])*"|;.*|[^\s('"`,;)]*)(.*)'''
  def __init__(self, file):
    self.file = file; self.line = ''
  def next_token(self):
    "Return the next token, reading new text into line buffer if needed."
    while True:
      if self.line == '': self.line = self.file.readline()
      if self.line == '': return eof_object
      token, self.line = re.match(InPort.tokenizer, self.line).groups()
      if token != '' and not token.startswith(';'):
        return token
```

Tail recursion optimization

- We can add some tail recursion optimization by altering our eval() function
- Consider evaluating
  - (if (> v 0) (begin 1 (begin 2 (twice (- v 1)))) 0)
- In an environment where
  - v=1
  - twice = (lambda (x) (* 2 x))

Tail recursion optimization

- Here's a trace of the recursive calls to eval

```
⇒ eval(x=(if (> v 0) (begin 1 (begin 2 (twice (- v 1)))) 0),  env={'v':1})
  ⇒ eval(x=(begin 1 (begin 2 (twice (- v 1)))),            env={'v':1})
    ⇒ eval(x=(begin 2 (twice (- v 1)))),                    env={'v':1})
      ⇒ eval(x=(twice (- v 1)))),                           env={'v':1})
        ⇒ eval(x=(* 2 y),                                   env={'y':0})
        ⇐ 0
      ⇐ 0
    ⇐ 0
  ⇐ 0
⇐ 0
```

- Eliminate recursive eval calls by setting x and env to required new values & iterate

Tail recursion optimization

- Wrap the eval code in a loop, use return to exit, otherwise set x and env to new values
- Here's a trace of the recursive calls to eval

```
⇒ eval(x=(if (> v 0) (begin 1 (begin 2 (twice (- v 1))))),  env={'v':1})
  x = (begin 1 (begin 2 (twice (- v 1))))
  x = (begin 2 (twice (- v 1))))
  x = (twice (- v 1))))
  x = (* 2 y); env = {'y':0}
⇐ 0
```

- No recursion: faster and does not grow stack

User defined functions

- We'll have to unpack our representation for user defined functions
- Define a simple class for a proceedure

```
class Procedure(object):
   "A user-defined Scheme procedure."
   def __init__(self, parms, exp, env):
      self.parms, self.exp, self.env = parms, exp, env
   def __call__(self, *args):
      return eval(self.exp, Env(self.parms, args, self.env))
```

- Evaling a lambda will just instantiate this

Eval

```
def eval(x, env=global_env):
 while True:
   if isa(x, Symbol): return env.lookup(x)
   elif not isa(x, list): return x
   elif x[0] == 'quote': return x[1]
   elif x[0] == 'if': x = x[2] if eval(x[1], env) else x[3]
   elif x[0] == 'set!':
      env.set(x[1], x[2])
      return None
   elif x[0] == 'define':
      env.define(x[1], x[2])
      return None
   elif x[0] == 'lambda': return Procedure(x[1], x[2], env)
   elif x[0] == 'begin':
      for exp in x[1:-1]: eval(exp, env)
      x = exp[-1]
else:
     exps = [eval(exp, env) for exp in x]
     proc = exps.pop(0)
     if isa(proc, Procedure):
       x, env = proc.exp, Env(proc.params, exps, proc.env)
     else: return proc(*exps)
```