CS46K Programming Languages

Structure and Interpretation of Computer Programs

Chapter 10C Scheme Interpreter Design

LECTURE 16: SCHEME INTERPRETER DESIGN

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- A subset of Scheme Language for the Interpreter Design
- REPL Loop
- Integration of Scheme Interpreter
- Language 1: Calculator Language
- Language 2: lis (simple Lispy)
- Language 3: lispy (advanced Lispy)

Overview

SECTION 1

(How to Write a (Lisp) Interpreter (in Python))

- This lecture has two purposes: to describe how to implement computer language interpreters in general, and in particular to build an interpreter for most of the *Scheme* dialect of Lisp using Python 3 as the implementation language. I call my language and interpreter *Lispy* (**lis.py**). Years ago, the author showed how to write a semi-practical Scheme interpreter Java and in in Common Lisp).
- This time around the goal is to demonstrate, as concisely and simply as possible, what Alan Kay called "*Maxwell's Equations of Software*."Why does this matter? As Steve Yegge said, *"If you don't know how compilers work, then you don't know how computers work."* Yegge describes 8 problems that can be solved with compilers (or equally well with interpreters, or with Yegge's typical heavy dosage of cynicism).

Syntax and Semantics of Scheme Programs

- The syntax of a language is the arrangement of characters to form correct statements or expressions; the semantics is the meaning of those statements or expressions. For example, in the language of mathematical expressions (and in many programming languages), the syntax for adding one plus two is "1 + 2" and the semantics is the application of the addition operation to the two numbers, yielding the value 3.
- We say we are evaluating an expression when we determine its value; we would say that "1 + 2" evaluates to 3, and write that as "1 + 2" $\Rightarrow$ 3.

Syntax and Semantics of Scheme Programs

Java	Scheme

if (x.val() > 0) {
  return fn(A[i] + 3 * i,
      new String[] {"one", "two"});
}
	(**if** (> (val x) 0)
	  (fn (+ (aref A i) (* 3 i))
	    (**quote** (one two)))

Syntax and Semantics of Scheme Programs

- Java has a wide variety of syntactic conventions (keywords, infix operators, three kinds of brackets, operator precedence, dot notation, quotes, commas, semicolons), but Scheme syntax is much simpler:
- Scheme programs consist solely of expressions. There is no statement/expression distinction.
- Numbers (e.g. 1) and symbols (e.g. A) are called atomic expressions; they cannot be broken into pieces. These are similar to their Java counterparts, except that in Scheme, operators such as + and > are symbols too, and are treated the same way as A and fn.

Syntax and Semantics of Scheme Programs

- Everything else is a list expression: a "(", followed by zero or more expressions, followed by a ")". The first element of the list determines what it means:
- A list starting with a keyword, e.g. (if ...), is a special form; the meaning depends on the keyword.
- A list starting with a non-keyword, e.g. (fn ...), is a function call.

Syntax and Semantics of Scheme Programs

- The beauty of Scheme is that the full language only needs 5 keywords and 8 syntactic forms. In comparison, Python has 33 keywords and 110 syntactic forms, and Java has 50 keywords and 133 syntactic forms. All those parentheses may seem intimidating, but Scheme syntax has the virtues of simplicity and consistency. (Some have joked that "Lisp" stands for "***L**ots **o**f **I**rritating **S**illy **P**arentheses*"; I think it stand for "***L**isp **I**s **S**yntactically **P**ure*".)

Syntax and Semantics of Scheme Programs

- In this Lecture we will cover all the important points of the Scheme language and its interpretation (omitting some minor details), but we will take two steps to get there, defining a simplified language first, before defining the near-full Scheme language.

Language 1: Lispy Calculator

SECTION 2

Language 1: Lispy Calculator

- *Lispy Calculator* is a subset of Scheme using only five syntactic forms (two atomic, two special forms, and the procedure call). Lispy Calculator lets you do any computation you could do on a typical calculator—as long as you are comfortable with prefix notation. And you can do two things that are not offered in typical calculator languages: "if" expressions, and the definition of new variables. Here's an example program, that computes the area of a circle of radius 10, using the formula π r^2:

```
(define r 10)
(* pi (* r r))
```

Language 1: Lispy Calculator

Expression	Syntax	Semantics and Example
variable reference	*symbol*	A symbol is interpreted as a variable name; its value is the variable's value. Example: r ⇒ 10 (assuming r was previously defined to be 10)
constant literal	*number*	A number evaluates to itself. Examples: 12 ⇒ 12 *or* -3.45e+6 ⇒ -3.45e+6
conditional	(if *test conseq alt*)	Evaluate *test*; if true, evaluate and return *conseq*; otherwise *alt*. Example: (if (> 10 20) (+ 1 1) (+ 3 3)) ⇒ 6
definition	(define *symbol exp*)	Define a new variable and give it the value of evaluating the expression *exp*. Examples: (define r 10)
procedure call	(*proc arg...*)	If *proc* is anything other than one of the symbols if, define, or quote then it is treated as a procedure. Evaluate *proc* and all the *args*, and then the procedure is applied to the list of *arg* values. Example: (sqrt (* 2 8)) ⇒ 4.0

Language 1: Lispy Calculator

- In the Syntax column of this table, symbol must be a symbol, number must be an integer or floating point number, and the other italicized words can be any expression. The notation arg... means zero or more repetitions of arg.

What A Language Interpreter Does

A language interpreter has two parts:

1. **Parsing:** The parsing component takes an input program in the form of a sequence of characters, verifies it according to the *syntactic rules* of the language, and translates the program into an internal representation. In a simple interpreter the internal representation is a tree structure (often called an *abstract syntax tree*) that closely mirrors the nested structure of statements or expressions in the program. In a language translator called a *compiler* there is often a series of internal representations, starting with an abstract syntax tree, and progressing to a sequence of instructions that can be directly executed by the computer. The Lispy parser is implemented with the function parse.

What A Language Interpreter Does

2.Execution: The internal representation is then processed according to the *semantic rules* of the language, thereby carrying out the computation. Lispy's execution function is called eval (note this shadows Python's built-in function of the same name).

What A Language Interpreter Does

- Here is a picture of the interpretation process:

program → **parse** → abstract-syntax-tree → **eval** → result

What A Language Interpreter Does

- And here is a short example of what we want parse and eval to be able to do (begin evaluates each expression in order and returns the final one):

```
>> program = "(begin (define r 10) (* pi (* r r)))"

>>> parse(program)
['begin', ['define', 'r', 10], ['*', 'pi', ['*',
'r', 'r']]]

>>> eval(parse(program))
314.1592653589793
```

Type Definitions

- Let's be explicit about our representations for Scheme objects:

```
Symbol = str              # A Scheme Symbol is implemented as a Python str
Number = (int, float)     # A Scheme Number is implemented as a Python int or float
Atom   = (Symbol, Number) # A Scheme Atom is a Symbol or Number
List   = list             # A Scheme List is implemented as a Python list
Exp    = (Atom, List)     # A Scheme expression is an Atom or List
Env    = dict             # A Scheme environment (defined below)
                          # is a mapping of {variable: value}
```

Parsing: `parse`, `tokenize` and `read_from_tokens`

- Parsing is traditionally separated into two parts: lexical analysis, in which the input character string is broken up into a sequence of tokens, and syntactic analysis, in which the tokens are assembled into an abstract syntax tree. The Lispy tokens are parentheses, symbols, and numbers. There are many tools for lexical analysis (such as Mike Lesk and Eric Schmidt's lex), but for now we'll use a very simple tool: Python's str.split. The function tokenize takes as input a string of characters; it adds spaces around each paren, and then calls str.split to get a list of tokens:

Parsing: **`parse`**, **`tokenize`** and **`read_from_tokens`**

tokenize.py

```
def tokenize(chars: str) -> list:
    "Convert a string of characters into a list of tokens."
    return chars.replace('(', ' ( ').replace(')', ' ) ').split()

program = "(begin (define r 10) (* pi (* r r)))"
results = tokenize(program)
print(results)
```

```
['(', 'begin', '(', 'define', 'r', '10', ')', '(', '*', 'pi', '(', '*',
'r', 'r', ')', ')', ')']
```

Parsing: **`parse`**, **`tokenize`** and **`read_from_tokens`**

- Our function parse will take a string representation of a program as input, call tokenize to get a list of tokens, and then call read_from_tokens to assemble an abstract syntax tree. read_from_tokens looks at the first token; if it is a ')' that's a syntax error. If it is a '(', then we start building up a list of sub-expressions until we hit a matching ')'. Any non-parenthesis token must be a symbol or number. We'll let Python make the distinction between them: for each non-paren token, first try to interpret it as an int, then as a float, and if it is neither of those, it must be a symbol.

parse.py

```python
def parse(program: str) -> Exp:
    "Read a Scheme expression from a string."
    return read_from_tokens(tokenize(program))
def read_from_tokens(tokens: list) -> Exp:
    "Read an expression from a sequence of tokens."
    if len(tokens) == 0:
        raise SyntaxError('unexpected EOF')
    token = tokens.pop(0)
    if token == '(':
        L = []
        while tokens[0] != ')':
            L.append(read_from_tokens(tokens))
        tokens.pop(0) # pop off ')'
        return L
    elif token == ')':
        raise SyntaxError('unexpected )')
    else:
        return atom(token)
def atom(token: str) -> Atom:
    "Numbers become numbers; every other token is a symbol."
    try: return int(token)
    except ValueError:
        try: return float(token)
        except ValueError:
            return Symbol(token)
```

Parsing: **`parse`**, **`tokenize`** and **`read_from_tokens`**

parse.py

```python
program = "(begin (define r 10) (* pi (* r r)))"
results = parse(program)
print(results)
```

```
['begin', ['define', 'r', 10], ['*', 'pi', ['*', 'r', 'r']]]
```

Environments

- An environment is a mapping from variable names to their values. By default, eval will use a global environment that includes the names for a bunch of standard functions (like sqrt and max, and also operators like *). This environment can be augmented with user-defined variables, using the expression (define symbol value).

environment.py

```python
def standard_env() -> Env:
    "An environment with some Scheme standard procedures."
    env = Env()
    env.update(vars(math)) # sin, cos, sqrt, pi, ...
    env.update({
        '+':op.add, '-':op.sub, '*':op.mul, '/':op.truediv,
        '>':op.gt, '<':op.lt, '>=':op.ge, '<=':op.le, '=':op.eq,
        'abs':     abs,
        'append':  op.add,
        'apply':   lambda proc, args: proc(*args),
        'begin':   lambda *x: x[-1],
        'car':     lambda x: x[0],
        'cdr':     lambda x: x[1:],
        'cons':    lambda x,y: [x] + y,
        'eq?':     op.is_,
        'expt':    pow,
        'equal?':  op.eq,
```

environment.py

```
        'length':  len,
        'list':    lambda *x: List(x),
        'list?':   lambda x: isinstance(x, List),
        'map':     map,
        'max':     max,
        'min':     min,
        'not':     op.not_,
        'null?':   lambda x: x == [],
        'number?': lambda x: isinstance(x, Number),
        'print':   print,
        'procedure?': callable,
        'round':   round,
        'symbol?': lambda x: isinstance(x, Symbol),
    })
    return env

global_env = standard_env()
```

Evaluation: **`eval`**

Expression	Syntax	Semantics and Example
variable reference	*symbol*	A symbol is interpreted as a variable name; its value is the variable's value. Example: r ⇒ 10 (assuming r was previously defined to be 10)
constant literal	*number*	A number evaluates to itself. Examples: 12 ⇒ 12 *or* -3.45e+6 ⇒ -3.45e+6
conditional	(if *test conseq alt*)	Evaluate *test*; if true, evaluate and return *conseq*; otherwise *alt*. Example: (if (> 10 20) (+ 1 1) (+ 3 3)) ⇒ 6
definition	(define *symbol exp*)	Define a new variable and give it the value of evaluating the expression *exp*. Examples: (define r 10)
procedure call	(*proc arg...*)	If *proc* is anything other than one of the symbols if, define, or quote then it is treated as a procedure. Evaluate *proc* and all the *args*, and then the procedure is applied to the list of *arg* values. Example: (sqrt (* 2 8)) ⇒ 4.0

environment.py

```python
def eval(x: Exp, env=global_env) -> Exp:
    "Evaluate an expression in an environment."
    if isinstance(x, Symbol):          # variable reference
        return env[x]
    elif isinstance(x, Number):        # constant number
        return x
    elif x[0] == 'if':                 # conditional
        (_, test, conseq, alt) = x
        exp = (conseq if eval(test, env) else alt)
        return eval(exp, env)
    elif x[0] == 'define':             # definition
        (_, symbol, exp) = x
        env[symbol] = eval(exp, env)
    else:                              # procedure call
        proc = eval(x[0], env)
        args = [eval(arg, env) for arg in x[1:]]
        return proc(*args)
```

Demo Program:

environment.py

```
program = "(begin (define r 10) (* pi (* r r)))"
results = eval(parse(program))
print(results)
```

```
314.1592653589793
```

Interaction: A REPL

- It is tedious to have to enter eval(parse("...")) all the time. One of Lisp's great legacies is the notion of an interactive read-eval-print loop: a way for a programmer to enter an expression, and see it immediately read, evaluated, and printed, without having to go through a lengthy build/compile/run cycle. So let's define the function repl (which stands for read-eval-print-loop), and the function schemestr which returns a string representing a Scheme object.

repl() Function

lispcalculator.py

```
def repl(prompt='lis.py> '):
    "A prompt-read-eval-print loop."
    while True:
        try:
            val = eval(parse(input(prompt)))
            if val is not None:
                print(schemestr(val))
        except: break    # ^Z to exit program

def schemestr(exp):
    "Convert a Python object back into a Scheme-readable string."
    if isinstance(exp, List):
        return '(' + ' '.join(map(schemestr, exp)) + ')'
    else:
        return str(exp)
```

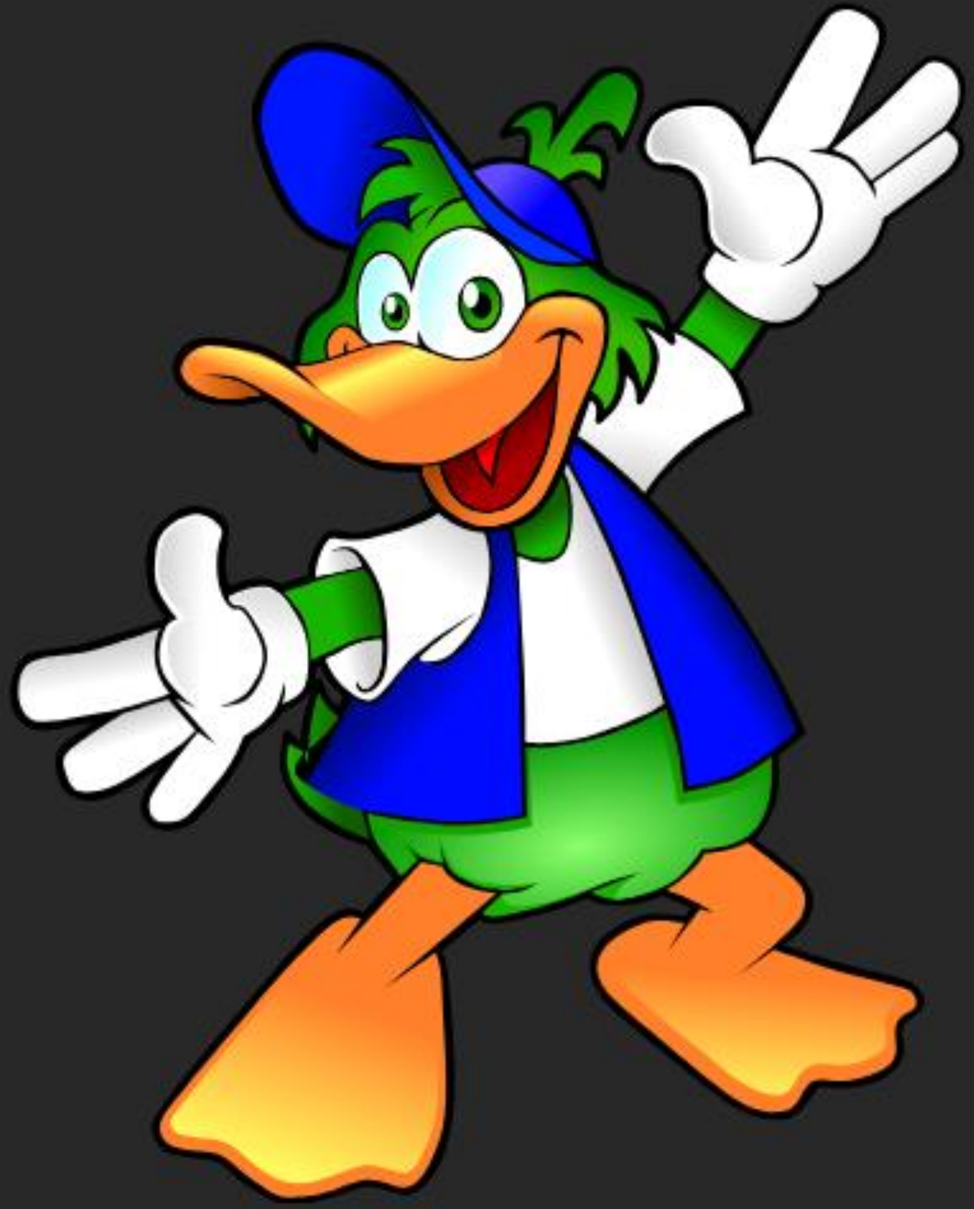
Demo Program

```
C:\Eric_Chou\Lewis University\CS 46K SIPC\PyDev\Lecture 16\Lispy1>python lispcalculator.py
lis.py> (define a 4)
lis.py> (+ a 3)
7
lis.py> (- (+ a 2) (* a 4))
-10
lis.py> ^Z

C:\Eric_Chou\Lewis University\CS 46K SIPC\PyDev\Lecture 16\Lispy1>
```

SCHEME PROGRAMMING

Part One



Language 2: Lispy

SECTION 3

Language 2: Full Lispy

- We will now extend our language with three new special forms, giving us a much more nearly-complete Scheme subset:

Expression	Syntax	Semantics and Example
quotation	*(quote exp)*	Return the exp literally; do not evaluate it. Example: (quote (+ 1 2)) ⇒ (+ 1 2) Example: r ⇒ 10 (assuming r was previously defined to be 10)
assignment	*(set! symbol exp)*	Evaluate exp and assign that value to symbol, which must have been previously defined (with a define or as a parameter to an enclosing procedure). Example: (set! r2 (* r r))
procedure	(lambda (symbol...) exp)	Create a procedure with parameter(s) named symbol... and exp as the body. Example: (lambda (r) (* pi (* r r)))

Demo Program:
lis.py + run_lis.py

- The lambda special form (an obscure nomenclature choice that refers to Alonzo Church's lambda calculus) creates a procedure. We want procedures to work like this:
- Run the run_lis.py which will call the lis.py

run_lis.py

```
from lis import *

if __name__ == "__main__":
    repl()
```

Run These Scheme Code

```
(define circle-area (lambda (r) (* pi (* r r))))

(circle-area (+ 5 5))
```

```
C:\Eric_Chou\Lewis University\CS 46K SIPC\PyDev\Lecture 16\Lispy2>python run_lis.py
lis.py> (define circle-area (lambda (r) (* pi (* r r))))
lis.py> (circle-area (+ 5 5))
314.1592653589793
lis.py> ^Z

C:\Eric_Chou\Lewis University\CS 46K SIPC\PyDev\Lecture 16\Lispy2>
```

Language 2: Full Lispy

- There are two steps here.
- In the first step, the lambda expression is evaluated to create a procedure, one which refers to the global variables pi and *, takes a single parameter, which it calls r. This procedure is used as the value of the new variable circle-area.

Language 2: Full Lispy

- In the second step, the procedure we just defined is the value of circle-area, so it is called, with the value 10 as the argument. We want r to take on the value 10, but it wouldn't do to just set r to 10 in the global environment. What if we were using r for some other purpose? We wouldn't want a call to circle-area to alter that value. Instead, we want to arrange for there to be a local variable named r that we can set to 10 without worrying about interfering with any global variable that happens to have the same name.

- The process for calling a procedure introduces these new local variable(s), binding each symbol in the parameter list of. the function to the corresponding value in the argument list of the function call.

Redefining Env as a Class

- To handle local variables, we will redefine Env to be a subclass of dict. When we evaluate (circle-area (+ 5 5)), we will fetch the procedure body, (* pi (* r r)), and evaluate it in an environment that has r as the sole local variable (with value 10), but also has the global environment as the "outer" environment; it is there that we will find the values of * and pi. In other words, we want an environment that looks like this, with the local (blue) environment nested inside the outer (red) global environment:

Redefining Env as a Class

```
pi: 3.141592653589793
*: <built-in function mul>
...
    r: 10
```

Redefining Env as a Class

- When we look up a variable in such a nested environment, we look first at the innermost level, but if we don't find the variable name there, we move to the next outer level. Procedures and environments are intertwined, so let's define them together:

lis.py

```python
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):
        "Find the innermost Env where var appears."
        return self if (var in self) else self.outer.find(var)

global_env = standard_env()

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

Redefining Env as a Class

- We see that every procedure has three components: a list of parameter names, a body expression, and an environment that tells us what other variables are accessible from the body. For a procedure defined at the top level this will be the global environment, but it is also possible for a procedure to refer to the local variables of the environment in which it was defined (and not the environment in which it is called).

Redefining Env as a Class

- An environment is a subclass of dict, so it has all the methods that dict has. In addition there are two methods: the constructor __init__ builds a new environment by taking a list of parameter names and a corresponding list of argument values, and creating a new environment that has those {variable: value} pairs as the inner part, and also refers to the given outer environment. The method find is used to find the right environment for a variable: either the inner one or an outer one.

Redefining Env as a Class

- To see how these all go together, here is the new definition of eval. Note that the clause for variable reference has changed: we now have to call env.find(x) to find at what level the variable x exists; then we can fetch the value of x from that level. (The clause for define has not changed, because a define always adds a new variable to the innermost environment.) There are two new clauses: for set!, we find the environment level where the variable exists and set it to a new value. With lambda, we create a new procedure object with the given parameter list, body, and environment.

lis.py

```python
def eval(x, env=global_env):
    "Evaluate an expression in an environment."
    if isinstance(x, Symbol):       # variable reference
        return env.find(x)[x]
    elif not isinstance(x, List):   # constant literal
        return x
    elif x[0] == 'quote':           # (quote exp)
        (_, exp) = x
        return exp
    elif x[0] == 'if':              # (if test conseq alt)
        (_, test, conseq, alt) = x
        exp = (conseq if eval(test, env) else alt)
        return eval(exp, env)
    elif x[0] == 'define':          # (define var exp)
        (_, var, exp) = x
        env[var] = eval(exp, env)
    elif x[0] == 'set!':            # (set! var exp)
        (_, var, exp) = x
        env.find(var)[var] = eval(exp, env)
    elif x[0] == 'lambda':          # (lambda (var...) body)
        (_, parms, body) = x
        return Procedure(parms, body, env)
    else:                           # (proc arg...)
        proc = eval(x[0], env)
        args = [eval(exp, env) for exp in x[1:]]
        return proc(*args)
```

Redefining Env as a Class

- To appreciate how procedures and environments work together, consider this program and the environment that gets formed when we evaluate (account1 -20.00):

```scheme
(define make-account
  (lambda (balance)
    (lambda (amt)
      (begin (set! balance (+ balance amt))
             balance))))

(define account1 (make-account 100.00))
(account1 -20.00)
```

```
+: <built-in operator add>
make-account: <a Procedure>
  balance: 100.00
    amt: -20.00

account1: <a Procedure>
```

Redefining Env as a Class

- Each rectangular box represents an environment, and the color of the box matches the color of the variables that are newly defined in the environment. In the last two lines of the program we define account1 and call (account1 -20.00); this represents the creation of a bank account with a 100 dollar opening balance, followed by a 20 dollar withdrawal. In the process of evaluating (account1 -20.00), we will eval the expression highlighted in yellow. There are three variables in that expression. amt can be found immediately in the innermost (green) environment. But balance is not defined there: we have to look at the green environment's outer env, the blue one. And finally, the variable + is not found in either of those; we need to do one more outer step, to the global (red) environment. This process of looking first in inner environments and then in outer ones is called lexical scoping. Env.find(var) finds the right environment according to lexical scoping rules.

Demo Program:

lis.py + run_lis.py (lambda2.scm)

```
>>> python run_lis.py
```

lambda2.scm

```
(define circle-area (lambda (r) (* pi (* r r))))
(circle-area 3)
(define fact (lambda (n) (if (<= n 1) 1 (* n (fact (- n 1))))))
(fact 10)
(fact 100)
(circle-area (fact 10))
(define first car)
(define rest cdr)
(define count (lambda (item L) (if L (+ (equal? item (first L)) (count item (rest L)))
0)))
(count 0 (list 0 1 2 3 0 0))
(count (quote the) (quote (the more the merrier the bigger the better)))
(define twice (lambda (x) (* 2 x)))
(twice 5)
(define repeat (lambda (f) (lambda (x) (f (f x)))))
((repeat twice) 10)
((repeat (repeat twice)) 10)
((repeat (repeat (repeat twice))) 10)
((repeat (repeat (repeat (repeat twice)))) 10)
(pow 2 16)
(define fib (lambda (n) (if (< n 2) 1 (+ (fib (- n 1)) (fib (- n 2))))))
(define range (lambda (a b) (if (= a b) (quote ()) (cons a (range (+ a 1) b)))))
(range 0 10)
(map fib (range 0 10))
(map fib (range 0 20))
```

```
C:\Eric_Chou\Lewis University\CS 46K SIPC\PyDev\Lecture 16\Lispy2>python run_lis.py
lis.py> (define circle-area (lambda (r) (* pi (* r r))))
lis.py> (circle-area 3)
28.274333882308138
lis.py> (define fact (lambda (n) (if (<= n 1) 1 (* n (fact (- n 1))))))
lis.py> (fact 10)
3628800
lis.py> (fact 100)
9332621544394415268169923885626670049071596826438162146859296389521759999322991560894146397615651828625369
7920827223758251185210916864000000000000000000000000
lis.py> (circle-area (fact 10))
41369087205782.695
lis.py> (define first car)
lis.py> (define rest cdr)
lis.py> (define count (lambda (item L) (if L (+ (equal? item (first L)) (count item (rest L))) 0)))
lis.py> (count 0 (list 0 1 2 3 0 0))
3
lis.py> (count (quote the) (quote (the more the merrier the bigger the better)))
4
lis.py> (define twice (lambda (x) (* 2 x)))
lis.py> (twice 5)
10
lis.py> (define repeat (lambda (f) (lambda (x) (f (f x)))))
lis.py> ((repeat twice) 10)
40
lis.py> ((repeat (repeat twice)) 10)
160
lis.py> ((repeat (repeat (repeat twice))) 10)
2560
lis.py> ((repeat (repeat (repeat (repeat twice)))) 10)
655360
lis.py> (pow 2 16)
65536.0
lis.py> (define fib (lambda (n) (if (< n 2) 1 (+ (fib (- n 1)) (fib (- n 2))))))
lis.py> (define range (lambda (a b) (if (= a b) (quote ()) (cons a (range (+ a 1) b)))))
lis.py> (range 0 10)
(0 1 2 3 4 5 6 7 8 9)
lis.py> (map fib (range 0 10))
<map object at 0x0000025F36EA5E10>
lis.py> (map fib (range 0 20))
<map object at 0x0000025F36EAF1D0>
lis.py>
```

Language 2: Full Lispy

- We now have a language with procedures, variables, conditionals (if), and sequential execution (the begin procedure). If you are familiar with other languages, you might think that a while or for loop would be needed, but Scheme manages to do without these just fine. The Scheme report says "Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language." In Scheme you iterate by defining recursive functions.

Language 3: Advanced Lispy

Part 1: New data types

SECTION 4

Language 3: Advanced Lispy

- In a previous essay the author showed how to write a simple Lisp interpreter in 90 lines of Python: **lis.py**.
- In this essay the author make the implementation, **lispy.py**, three times more complicated, but more complete. Each section handles an addition.

New data types: string, boolean, complex, port

Adding a new data type to Lispy has three parts: the internal representation of the data, the procedures that operate on it, and the syntax for reading and writing it. Here we add four types (using Python's native representation for all but input ports):

- **strings**: string literals are enclosed in double-quotes. Within a string, a \n means a newline and a \" means a double-quote.
- **booleans**: The syntax is #t and #f for True and False, and the predicate is boolean?.

New data types: string, boolean, complex, port

- **complex numbers**: we use the functions in the cmath module rather than the math module to support complex numbers. The syntax allows constants like 3+4i.
- **ports**: No syntax to add, but procedures port?, load, open-input-file, close-input-port, open-output-file, close-output-port, read, read-char, write and display. Output ports are represented as Python file objects, and input ports are represented by a class, InputPort which wraps a file object and also keeps track of the last line of text read. This is convenient because Scheme input ports need to be able to read expressions as well as raw characters and our tokenizer works on a whole line, not individual characters.

New data types: string, boolean, complex, port

Now, an old data type that becomes new:

- **symbol**: In the previous version of Lispy, symbols were implemented as strings. Now that we have strings, symbols will be implemented as a separate class (which derives from str). That means we no longer can write if x[0] == 'if', because 'if' is now a string, not a symbol. Instead we write if x[0] is _if and define _if as Sym('if'), where Sym manages a symbol table of unique symbols.

lispy.py

```python
class Symbol(str): pass

def Sym(s, symbol_table={}):
    "Find or create unique Symbol entry for str s in symbol table."
    if s not in symbol_table: symbol_table[s] = Symbol(s)
    return symbol_table[s]

_quote, _if, _set, _define, _lambda, _begin, _definemacro, = map(Sym,
"quote   if   set!  define   lambda   begin   define-macro".split())

_quasiquote, _unquote, _unquotesplicing = map(Sym,
"quasiquote   unquote   unquote-splicing".split())
```

Language 3: Advanced Lispy

Part 2: New syntax

SECTION 5

New syntax: strings, comments, quotes, # literals

- The addition of strings complicates tokenization. No longer can spaces delimit tokens, because spaces can appear inside strings. Instead we use a complex regular expression to break the input into tokens. In Scheme a comment consists of a semicolon to the end of line; we gather this up as a token and then ignore the token. We also add support for six new tokens: #t #f ' ` , ,@

New syntax: strings, comments, quotes, # literals

- The tokens #t and #f are the True and False literals, respectively. The single quote mark serves to quote the following expression. The syntax 'exp is completely equivalent to (quote exp). The backquote character ` is called quasiquote in Scheme; it is similar to ' except that within a quasiquoted expression, the notation ,exp means to insert the value of exp (rather than the literal exp), and ,@exp means that exp should evaluate to a list, and all the items of the list are inserted.

New syntax: strings, comments, quotes, # literals

- In the previous version of Lispy, all input was read from strings. In this version we have introduced ports (also known as file objects or streams) and will read from them. This makes the read-eval-print-loop (repl) much more convenient: instead of insisting that an input expression must fit on one line, we can now read tokens until we get a complete expression, even if it spans several lines. Also, errors are caught and printed, much as the Python interactive loop does. Here is the InPort (input port) class:

```
class InPort:
    "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
```

New syntax: strings, comments, quotes, # literals

- The basic design for the read function follows a suggestion (with working code) from Darius Bacon (who contributed several other improvements as well).

lispy.py

```python
def readchar(inport):
    "Read the next character from an input port."
    if inport.line != '':
        ch, inport.line = inport.line[0], inport.line[1:]
        return ch
    else:
        return inport.file.read(1) or eof_object

def read(inport):
    "Read a Scheme expression from an input port."
    def read_ahead(token):
        if '(' == token:
            L = []
            while True:
                token = inport.next_token()
                if token == ')': return L
                else: L.append(read_ahead(token))
        elif ')' == token: raise SyntaxError('unexpected )')
        elif token in quotes: return [quotes[token], read(inport)]
        elif token is eof_object: raise SyntaxError('unexpected EOF in list')
        else: return atom(token)
    # body of read:
    token1 = inport.next_token()
    return eof_object if token1 is eof_object else read_ahead(token1)

quotes = {"'":_quote, "`":_quasiquote, ",":_unquote, ",@":_unquotesplicing}
```

lispy.py

```python
def atom(token):
    'Numbers become numbers; #t and #f are booleans; "..." string; otherwise Symbol.'
    if token == '#t': return True
    elif token == '#f': return False
    elif token[0] == '"': return token[1:-1]
    try: return int(token)
    except ValueError:
        try: return float(token)
        except ValueError:
            try: return complex(token.replace('i', 'j', 1))
            except ValueError:
                return Sym(token)

def to_string(x):
    "Convert a Python object back into a Lisp-readable string."
    if x is True: return "#t"
    elif x is False: return "#f"
    elif isa(x, Symbol): return x
    elif isa(x, str): return repr(x)
    elif isa(x, list): return '('+' '.join(map(to_string, x))+')'
    elif isa(x, complex): return str(x).replace('j', 'i')
    else: return str(x)
```

lispy.py

```python
def load(filename):
    "Eval every expression from a file."
    repl(None, InPort(open(filename)), None)

def repl(prompt='lispy> ', out=sys.stdout):
    "A prompt-read-eval-print loop."
    print("Lispy version 2.0\n")
    while True:
        try:
            val = eval(parse(input(prompt)))
            if val is not None and out:
                print(to_string(val), file=out)
        except KeyboardInterrupt: # ^c
            break
        except EOFError: # ^Z
            break
        except Exception as e:
            print('%s: %s' % (type(e).__name__, e))
```

Language 3: Advanced Lispy

Part 3: Macros

SECTION 6

Macros: user-defined and builtin derived syntax

- We also add a facility for defining macros. This is available to the user, through the define-macro special form (which is slightly different than standard Scheme), and is also used internally to define so-called derived expressions, such as the and form. Macros definitions are only allowed at the top level of a file or interactive session, or within a begin form that is at the top level.

lispy.py

```
def let(*args):
    args = list(args)
    x = cons(_let, args)
    require(x, len(args)>1)
    bindings, body = args[0], args[1:]
    require(x, all(isa(b, list) and len(b)==2 and isa(b[0], Symbol)
                   for b in bindings), "illegal binding list")
    vars, vals = zip(*bindings)
    return [[_lambda, list(vars)]+list(map(expand, body))] + list(map(expand, vals))

macro_table = {_let:let} ## More macros can go here

eval(parse("""(begin

(define-macro and (lambda args
   (if (null? args) #t
       (if (= (length args) 1) (car args)
           `(if ,(car args) (and ,@(cdr args)) #f)))))

;; More macros can also go here

)"""))
```

Language 3: Advanced Lispy

Part 4: Better eval with tail recursion optimization

SECTION 7

Better eval with tail recursion optimization

- Scheme has no while or for loops, relying on recursion for iteration. That makes the language simple, but there is a potential problem: if every recursive call grows the runtime stack, then the depth of recursion, and hence the ability to loop, will be limited. In some implementations the limit will be as small as a few hundred iterations. This limitation can be lifted by altering eval so that it does not grow the stack on all recursive calls--only when necessary.

Better eval with tail recursion optimization

- Consider the evaluation of (if (> v 0) (begin 1 (begin 2 (twice (- v 1))))) when v is 1 and twice is the procedure (lambda (y) (* 2 y)). With the version of eval in lis.py, we would get the following trace of execution, where each arrow indicates a recursive call to eval:

```
⇒ eval(x=(if (> v 0) (begin 1 (begin 2 (twice (- v 1))))),     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
```

Better eval with tail recursion optimization

- But note that the recursive calls are not necessary. Instead of making a recursive call that returns a value that is then immediately returned again by the caller, we can instead alter the value of x (and sometimes env) in the original invocation of eval(x, env). We are free to do that whenever the old value of x is no longer needed. The call sequence now looks like this:

```
⇒ 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
```

Better eval with tail recursion optimization

- Here is an implementation of eval that works this way. We wrap the body in a while True loop, and then for most clauses, the implementation is unchanged. However, for three clauses we update the variable x (the expression being evaluated): for if, for begin, and for procedure calls to a user-defined procedure (in that case, we not ony update x to be the body of the procedure, we also update env to be a new environment that has the bindings of the procedure parameters).

lispy.py

```python
def eval(x, env=global_env):
    "Evaluate an expression in an environment."
    while True:
        if isa(x, Symbol):          # variable reference
            return env.find(x)[x]
        elif not isa(x, list):      # constant literal
            return x
        elif x[0] is _quote:        # (quote exp)
            (_, exp) = x
            return exp
        elif x[0] is _if:           # (if test conseq alt)
            (_, test, conseq, alt) = x
            x = (conseq if eval(test, env) else alt)
        elif x[0] is _set:          # (set! var exp)
            (_, var, exp) = x
            env.find(var)[var] = eval(exp, env)
            return None
        elif x[0] is _define:       # (define var exp)
            (_, var, exp) = x
            env[var] = eval(exp, env)
            return None
        elif x[0] is _lambda:       # (lambda (var*) exp)
            (_, vars, exp) = x
            return Procedure(vars, exp, env)
        elif x[0] is _begin:        # (begin exp+)
            for exp in x[1:-1]:
                eval(exp, env)
            x = x[-1]
        else:                       # (proc exp*)
            exps = [eval(exp, env) for exp in x]
            proc = exps.pop(0)
            if isa(proc, Procedure):
                x = proc.exp
                env = Env(proc.parms, exps, proc.env)
            else:
                return proc(*exps)
```

```python
class Procedure:
    "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))
```

Better eval with tail recursion optimization

- This implementation makes it possible to write procedures that recurse arbitrarily deeply without running out of storage. However, it may require some restructring of procedures to make this work. Consider these two implementations of a function to sum the integers from 0 to n:

sumto.scm

```scheme
(define (sum-to n)
  (if (= n 0)
      0
      (+ n (sum-to (- n 1)))))

(define (sum2 n acc)
  (if (= n 0)
      acc
      (sum2 (- n 1) (+ n acc))))
```

```
C:\Eric_Chou\Lewis University\CS 46K SIPC\PyDev\Lecture 16\Lispy3>python lispy.py
Lispy version 2.0
(define (sum-to n)
  (if (= n 0)
      0
      (+ n (sum-to (- n 1)))))

(define (sum2 n acc)
  (if (= n 0)
      acc
      (sum2 (- n 1) (+ n acc))))
(sum-to 10)
55
(sum2 10 0)
55
```

Execution Result: sumto.scm

Better eval with tail recursion optimization

- The first is more straightforward, but it yields a "RuntimeError: maximum recursion depth exceeded" on (sum-to 1000). The second version has the recursive call to sum2 in the last position of the body, and thus you can safely sum the first million integers with (sum2 1000000 0) and get 500000500000. Note that the second argument, acc, accumulates the results computed so far. If you can learn to use this style of accumulators, you can recurse arbitrarily deeply.

Language 3: Advanced Lispy

Part 5: Call-with-current-continuation (call/cc)

SECTION 9

Call-with-current-continuation (call/cc)

- We have seen that Scheme handles iteration using recursion, with no need for special syntax for for or while loops. But what about non-local control flow, as is done with try/except in Python or setjmp/longjmp in C? Scheme offers a primitive procedure, called call/cc for "call with current continuation". Let's start with some examples:

```
C:\Eric_Chou\Lewis University\CS 46K SIPC\PyDev\Lecture 16\Lispy3>python lispy.py
Lispy version 2.0
(call/cc (lambda (throw)
         (+ 5 (* 10 (call/cc (lambda (escape) (* 100 (escape 3))))))))
35
(call/cc (lambda (throw)
         (+ 5 (* 10 (call/cc (lambda (escape) (* 100 (throw 3))))))))
3
```

Execution Result: lambda3.scm

Call-with-current-continuation (call/cc)

- **In the first example**, evaluating (escape 3) causes Scheme to abort the current calculation and return 3 as the value of the enclosing call to call/cc. The result is the same as (+ 5 (* 10 3)) or 35.

- **In the second example**, (throw 3) aborts up two levels, throwing the value of 3 back to the top level. In general, call/cc takes a single argument, *proc*, which must be a procedure of one argument. *proc* is called, passing it a manufactured procedure which we will call *throw*. If *throw* is called with a single argument, then that argument is the value of the whole call to call/cc. If *throw* is not called, the value computed by *proc* is returned. Here is the implementation:

Call-with-current-continuation (call/cc)

```
def callcc(proc):
    "Call proc with current continuation; escape only"
    ball = RuntimeWarning("Sorry, can't continue this continuation any longer.")
    def throw(retval): ball.retval = retval; raise ball
    try:
        return proc(throw)
    except RuntimeWarning as w:
        if w is ball: return ball.retval
        else: raise w
```

Call-with-current-continuation (call/cc)

- This implementation allows for non-local escape from procedures. It does not, however, implement the full power of a real Scheme call/cc, with which we can not only call the continuation to return a value, we can also store the continuation away and call it multiple times, each time returning to the same place.

Language 3: Advanced Lispy

Part 6: Procedures with arbitrary number of arguments

SECTION 10

Procedures with arbitrary number of arguments

- The standard Scheme procedure list can be called with any number of arguments: (list 1 2), (list 1 2 3), etc. In Scheme a user can define a procedure like this using the syntax (lambda *args* *body*) where *args* is a single symbol representing the parameter that is bound to the list of arguments supplied in a procedure call, and *body* is the body of the procedure. The implementation takes just one small change in Env.__init__ to check if parms is a Symbol rather than a list:

```
class Env(dict):
    "An environment: a dict of {'var':val} pairs, with an outer Env."
    def __init__(self, parms=(), args=(), outer=None):
        # Bind parm list to corresponding args, or single parm to list of args
        self.outer = outer
        if isa(parms, Symbol):
            self.update({parms:list(args)})
        else:
            if len(args) != len(parms):
                raise TypeError('expected %s, given %s, '
                                % (to_string(parms), to_string(args)))
            self.update(zip(parms,args))
    def find(self, var):
        "Find the innermost Env where var appears."
        if var in self: return self
        elif self.outer is None: raise LookupError(var)
        else: return self.outer.find(var)
```

Procedures with arbitrary number of arguments

- If parms is a Symbol, we bind it to the list or arguments. Otherwise we bind each parm to the corresponding arg. Real Scheme also has the syntax (lambda (arg1 arg2 . rest) ...). We can't do that because we're using Python lists, and don't have dotted pairs.

Language 3: Advanced Lispy

Part 7: Earlier error detection and extended syntax

SECTION 11

Earlier error detection and extended syntax

- Consider the following erroneous code:

Lambda4.scm

```
(define f (lambda (x) (set! 3 x)))
(define g (lambda (3) (if (x = 0))))
(define h (lambda (x) (if (x = 0) 1 2 3)))
```

- In the first version of Lispy, evaluating these definitions would not yield any complaints. But as soon as any of the functions were called, a runtime error would occur. In general, errors should be reported as early as possible, so the new version of Lispy would give appropriate error messages as these functions are defined, not waiting for them to be called.

Earlier error detection and extended syntax

- We do this by improving the procedure parse. In the first version of Lispy, parse was implemented as read; in other words, any expression at all was accepted as a program. The new version checks each expression for validity when it is defined. It checks that each special form has the right number of arguments and that set! and define operate on symbols. It also expands the macros and quasiquote forms defined in section (2) above. It accepts a slightly more generous version of Scheme, as described in the table below. Each of the expressions on the left would be illegal in the first version of Lispy, but are accepted as equivalent to the corresponding expressions on the right in the new version:

Earlier error detection and extended syntax

Extended Expression	Expansion
(begin)	None
(if *test conseq*)	(if *test conseq* None)
(define (*f arg...*) *body...*)	(define *f* (lambda (*arg...*) *body...*)
(lambda (*arg...*) *e1 e2...*)	(lambda (*arg...*) (begin *e1 e2...*))
`*exp* (quasiquote *exp*)	expand , and ,@ within *exp*
(*macro-name arg...*)	expansion of (*macro-name arg...*)

Definition of Parse

```
def parse(inport):
    "Parse a program: read and expand/error-check it."
    # Backwards compatibility: given a str, convert it to an InPort
    if isinstance(inport, str): inport = InPort(io.StringIO(inport))
    return expand(read(inport), toplevel=True)
```

Definition of Expand

- And here is the definition of expand. It may seem odd that expand is twice as long as eval. But expand actually has a harder job: it has to do almost everything eval does in terms of making sure that legal code has all the right pieces, but in addition it must deal with illegal code, producing a sensible error message, and extended code, converting it into the right basic form.

lispy.py

```python
def expand(x, toplevel=False):
    "Walk tree of x, making optimizations/fixes, and signaling SyntaxError."
    require(x, x!=[])                    # () => Error
    if not isa(x, list):                 # constant => unchanged
        return x
    elif x[0] is _quote:                 # (quote exp)
        require(x, len(x)==2)
        return x
    elif x[0] is _if:
        if len(x)==3: x = x + [None]     # (if t c) => (if t c None)
        require(x, len(x)==4)
        return list(map(expand, x))
    elif x[0] is _set:
        require(x, len(x)==3)
        var = x[1]                       # (set! non-var exp) => Error
        require(x, isa(var, Symbol), "can set! only a symbol")
        return [_set, var, expand(x[2])]
```

```
elif x[0] is _define or x[0] is _definemacro:
    require(x, len(x)>=3)
    _def, v, body = x[0], x[1], x[2:]
    if isa(v, list) and v:          # (define (f args) body)
        f, args = v[0], v[1:]       #  => (define f (lambda (args) body))
        return expand([_def, f, [_lambda, args]+body])
    else:
        require(x, len(x)==3)       # (define non-var/list exp) => Error
        require(x, isa(v, Symbol), "can define only a symbol")
        exp = expand(x[2])
        if _def is _definemacro:
            require(x, toplevel, "define-macro only allowed at top level")
            proc = eval(exp)
            require(x, callable(proc), "macro must be a procedure")
            macro_table[v] = proc   # (define-macro v proc)
            return None             #  => None; add v:proc to macro_table
        return [_define, v, exp]
```

```
elif x[0] is _begin:
    if len(x)==1: return None          # (begin) => None
    else: return [expand(xi, toplevel) for xi in x]
elif x[0] is _lambda:                  # (lambda (x) e1 e2)
    require(x, len(x)>=3)              #  => (lambda (x) (begin e1 e2))
    vars, body = x[1], x[2:]
    require(x, (isa(vars, list) and all(isa(v, Symbol) for v in vars))
            or isa(vars, Symbol), "illegal lambda argument list")
    exp = body[0] if len(body) == 1 else [_begin] + body
    return [_lambda, vars, expand(exp)]
elif x[0] is _quasiquote:              # `x => expand_quasiquote(x)
    require(x, len(x)==2)
    return expand_quasiquote(x[1])
elif isa(x[0], Symbol) and x[0] in macro_table:
    return expand(macro_table[x[0]](*x[1:]), toplevel) # (m arg...)
else:                                  #        => macroexpand if m isa macro
    return list(map(expand, x))            # (f arg...) => expand each
```

```python
def require(x, predicate, msg="wrong length"):
    "Signal a syntax error if predicate is false."
    if not predicate: raise SyntaxError(to_string(x)+': '+msg)

_append, _cons, _let = map(Sym, "append cons let".split())

def expand_quasiquote(x):
    """Expand `x => 'x; `,x => x; `(,@x y) => (append x y) """
    if not is_pair(x):
        return [_quote, x]
    require(x, x[0] is not _unquotesplicing, "can't splice here")
    if x[0] is _unquote:
        require(x, len(x)==2)
        return x[1]
    elif is_pair(x[0]) and x[0][0] is _unquotesplicing:
        require(x[0], len(x[0])==2)
        return [_append, x[0][1], expand_quasiquote(x[1:])]
    else:
        return [_cons, expand_quasiquote(x[0]), expand_quasiquote(x[1:])]
```

Language 3: Advanced Lispy

Part 8: More primitive procedures

SECTION 12

More primitive procedures

- Here we augment add_globals with some more primitive Scheme procedures, bringing the total to 75. There are still around 80 missing ones; they could also be added here if desired.

```
def is_pair(x): return x != [] and isa(x, list)
def cons(x, y): return [x]+y
```

```python
def add_globals(self):
    "Add some Scheme standard procedures."
    import math, cmath, operator as op
    self.update(vars(math))
    self.update(vars(cmath))
    self.update({
     '+':op.add, '-':op.sub, '*':op.mul, '/':op.truediv, 'not':op.not_,
     '>':op.gt, '<':op.lt, '>=':op.ge, '<=':op.le, '=':op.eq,
     'equal?':op.eq, 'eq?':op.is_, 'length':len, 'cons':cons,
     'car':lambda x:x[0], 'cdr':lambda x:x[1:], 'append':op.add,
     'list':lambda *x:list(x), 'list?': lambda x:isa(x,list),  'map': map,
     'null?':lambda x:x==[], 'symbol?':lambda x: isa(x, Symbol),
     'boolean?':lambda x: isa(x, bool), 'pair?':is_pair,
     'port?': lambda x:isa(x,file), 'apply':lambda proc,l: proc(*l),
     'eval':lambda x: eval(expand(x)), 'load':lambda fn: load(fn), 'call/cc':callcc,
     'open-input-file':open,'close-input-port':lambda p: p.file.close(),
     'open-output-file':lambda f:open(f,'w'), 'close-output-port':lambda p: p.close(),
     'eof-object?':lambda x:x is eof_object, 'read-char':readchar,
     'read':read, 'write':lambda x,port=sys.stdout:port.write(to_string(x)),
     'display':lambda x,port=sys.stdout:port.write(x if isa(x,str) else to_string(x))})
    return self

isa = isinstance
global_env = add_globals(Env())
```

Language 3: Advanced Lispy

Part 9: Testing

SECTION 13

Testing

- Complicated programs should always be accompanied by a thorough test suite. We provide the program lispytest.py, which tests both versions of Lispy:
- Note: the input files for testing are divided into 9 different scheme files (.scm)
- You may copy and try one by one.

```
C:\Eric_Chou\Lewis University\CS 46K SIPC\PyDev\Lecture 16\Lispy3>python lispy.py
Lispy version 2.0
(quote (testing 1 (2.0) -3.14e159))
(testing 1 (2.0) -3.14e+159)
(+ 2 2)
4
(+ (* 2 100) (* 1 10))
210
(if (> 6 5) (+ 1 1) (+ 2 2))
2
(if (< 6 5) (+ 1 1) (+ 2 2))
4
(define x 3)
x
3
(+ x x)
6
(begin (define x 1) (set! x (+ x 1)) (+ x 1))
3
((lambda (x) (+ x x)) 5)
10
```

test1.scm

```
(define twice (lambda (x) (* 2 x)))
(twice 5)
10
(define compose (lambda (f g) (lambda (x) (f (g x)))))
((compose list twice) 5)
(10)
(define repeat (lambda (f) (compose f f)))
((repeat twice) 5)
20
((repeat (repeat twice)) 5)
80
(define fact (lambda (n) (if (<= n 1) 1 (* n (fact (- n 1))))))
(fact 3)
6
(fact 50)
30414093201713378043612608166064768844377641568960512000000000000
(define abs (lambda (n) ((if (> n 0) + -) 0 n)))
(list (abs -3) (abs 0) (abs 3))
(3 0 3)
```

test1.scm

```scheme
(define combine (lambda (f)
    (lambda (x y)
      (if (null? x) (quote ())
          (f (list (car x) (car y))
             ((combine f) (cdr x) (cdr y)))))))
(define zip (combine cons))
(zip (list 1 2 3 4) (list 5 6 7 8))
((1 5) (2 6) (3 7) (4 8))
(define riff-shuffle (lambda (deck) (begin
    (define take (lambda (n seq) (if (<= n 0) (quote ()) (cons (car seq) (take (- n 1) (cdr seq))))))
    (define drop (lambda (n seq) (if (<= n 0) seq (drop (- n 1) (cdr seq)))))
    (define mid (lambda (seq) (/ (length seq) 2)))
    ((combine append) (take (mid deck) deck) (drop (mid deck) deck)))))
(riff-shuffle (list 1 2 3 4 5 6 7 8))
(1 5 2 6 3 7 4 8)
((repeat riff-shuffle) (list 1 2 3 4 5 6 7 8))
(1 3 5 7 2 4 6 8)
(riff-shuffle (riff-shuffle (riff-shuffle (list 1 2 3 4 5 6 7 8))))
(1 2 3 4 5 6 7 8)
```

test2.scm

```
(quote (testing 1 (2.0) -3.14e159))
(testing 1 (2.0) -3.14e+159)
(+ 2 2)
4
(+ (* 2 100) (* 1 10))
210
(if (> 6 5) (+ 1 1) (+ 2 2))
2
(if (< 6 5) (+ 1 1) (+ 2 2))
4
(define x 3)
x
3
(+ x x)
6
(begin (define x 1) (set! x (+ x 1)) (+ x 1))
3
((lambda (x) (+ x x)) 5)
10
(define twice (lambda (x) (* 2 x)))
(twice 5)
10
```

test3.scm

```
(define compose (lambda (f g) (lambda (x) (f (g x)))))
((compose list twice) 5)
(10)
(define repeat (lambda (f) (compose f f)))
((repeat twice) 5)
20
((repeat (repeat twice)) 5)
80
(define fact (lambda (n) (if (<= n 1) 1 (* n (fact (- n 1))))))
(fact 3)
6
(fact 50)
30414093201713378043612608166064768844377641568960512000000000000
(define abs (lambda (n) ((if (> n 0) + -) 0 n)))
(list (abs -3) (abs 0) (abs 3))
(3 0 3)
```

test3.scm

```
(define combine (lambda (f)
    (lambda (x y)
      (if (null? x) (quote ())
          (f (list (car x) (car y))
             ((combine f) (cdr x) (cdr y)))))))
(define zip (combine cons))
(zip (list 1 2 3 4) (list 5 6 7 8))
((1 5) (2 6) (3 7) (4 8))
(define riff-shuffle (lambda (deck) (begin
    (define take (lambda (n seq) (if (<= n 0) (quote ()) (cons (car seq) (take (- n 1) (cdr seq)))))
)
    (define drop (lambda (n seq) (if (<= n 0) seq (drop (- n 1) (cdr seq)))))
    (define mid (lambda (seq) (/ (length seq) 2)))
    ((combine append) (take (mid deck) deck) (drop (mid deck) deck)))))
(riff-shuffle (list 1 2 3 4 5 6 7 8))
(1 5 2 6 3 7 4 8)
((repeat riff-shuffle) (list 1 2 3 4 5 6 7 8))
(1 3 5 7 2 4 6 8)
(riff-shuffle (riff-shuffle (riff-shuffle (list 1 2 3 4 5 6 7 8))))
(1 2 3 4 5 6 7 8)
```

test4.scm

```
;; Errors
()
SyntaxError: (): wrong length
(set! x)
SyntaxError: (set! x): wrong length
(define 3 4)
SyntaxError: (define 3 4): can define only a symbol
(quote 1 2)
SyntaxError: (quote 1 2): wrong length
(if 1 2 3 4)
SyntaxError: (if 1 2 3 4): wrong length
(lambda 3 3)
SyntaxError: (lambda 3 3): illegal lambda argument list
(lambda (x))
SyntaxError: (lambda (x)): wrong length
(if (= 1 2) (define-macro a 'a)
    (define-macro a 'b))
SyntaxError: (define-macro a (quote a)): define-macro only allowed at top level
;; another error
(define (twice x) (* 2 x))
(twice 2)
4
(twice 2 2)
TypeError: expected (x), given (2 2),
```

test4z_errors.scm

```
(define lyst (lambda items items))
(lyst 1 2 3 (+ 2 2))
(1 2 3 4)
(if 1 2)
2
(if (= 3 4) 2)
(define ((account bal) amt) (set! bal (+ bal amt)) bal)
(define a1 (account 100))
(a1 0)
100
(a1 10)
110
(a1 10)
120
```

test5.scm

```
(define (newton guess function derivative epsilon)
    (define guess2 (- guess (/ (function guess) (derivative guess))))
    (if (< (abs (- guess guess2)) epsilon) guess2
        (newton guess2 function derivative epsilon)))
(define (square-root a)
    (newton 1 (lambda (x) (- (* x x) a)) (lambda (x) (* 2 x)) 1e-8))
(> (square-root 200.) 14.14213)
#t
(< (square-root 200.) 14.14215)
#t
(= (square-root 200.) (sqrt 200.))
#t
```

test5.scm

test6.scm

```
(define (sum-squares-range start end)
         (define (sumsq-acc start end acc)
            (if (> start end) acc (sumsq-acc (+ start 1) end (+ (* start start) acc))))
         (sumsq-acc start end 0))
(sum-squares-range 1 3000)
9004500500
;;
(* 1i 1i)
(-1+0i)
(sqrt -1)
1i
(let ((a 1) (b 2)) (+ a b))
3
(let ((a 1) (b 2 3)) (+ a b)) ;; error
SyntaxError: (let ((a 1) (b 2 3)) (+ a b)): illegal binding list
(and 1 2 3)
3
(and (> 2 1) 2 3)
3
(and)
#t
(and (> 2 1) (> 2 3))
#f
(define-macro unless (lambda args `(if (not ,(car args)) (begin ,@(cdr args))))) ; test ` => None
(unless (= 2 (+ 1 1)) (display 2) 3 4)
```

```
;;
(call/cc (lambda (throw) (+ 5 (* 10 (throw 1))))) ;; throw => 1
1
(call/cc (lambda (throw) (+ 5 (* 10 1)))) ;; do not throw => 15
15
(call/cc (lambda (throw)
            (+ 5 (* 10 (call/cc (lambda (escape) (* 100 (escape 3)))))))) ; 1 level => 35
35
(call/cc (lambda (throw)
            (+ 5 (* 10 (call/cc (lambda (escape) (* 100 (throw 3)))))))) ; 2 levels => 3
3
(call/cc (lambda (throw)
            (+ 5 (* 10 (call/cc (lambda (escape) (* 100 1))))))) ; 0 levels => 1005
1005
```

test6z_callcc.scm

```
(unless (= 4 (+ 1 1)) (display 2) (display "\n") 3 4)
2\n4
(quote x)
x
(quote (1 2 three))
(1 2 three)
'x
x
'(one 2 3)
(one 2 3)
(define L (list 1 2 3))
`(testing ,@L testing)
(testing 1 2 3 testing)
`(testing ,L testing) =
(testing (1 2 3) testing)
<built-in function eq>
`,@L ;; error
SyntaxError: (unquote-splicing L): can't splice here
'(1 ;test comments '
     ;skip this line
     2 ; more ; comments ; ) )
     3) ; final comment
(1 2 3)
```

test7.scm

Conclusion

SECTION 14

Programming Language Course

- Programming language course is targeted at the study of programming paradigm, design of languages, and implementation of the interpretation of computer programs.
- In this course, we use Python language as our pilot language to study the programming language features, the programming paradigms, the design of the lexical analyzer, the syntactical analyzer, the generation of abstract syntax tree, the evaluation of syntax tree, the REPL evaluation loop, the virtual machine, and the integration of all of these techniques.

Programming Language Course

- We have the following program examples:
    1. Ad hoc lexical analyzer
    2. Use of **Lex** and **Yacc** for Lexical Analyzer and Parser design automation
    3. Recursive Descent parser design
    4. Interpreter design on a virtual machine
    5. REPL-loop based interpreter design

Programming Language Course

- Programming language pragmatics studies the Structure and Interpretation of Computer Programs (SICP). It is the most valuable software. The knowledge can be applied but not limited to:
  1. Compiler Design
  2. Interpreter Design
  3. Web-Engine Design
  4. Natural Language Processing
  5. XML to graphics conversion

Programming Language Course

- In this course, we try to provide the fundamental knowledge for the design of interpreters. In the real-world applications, there should be a lot of different engineering issues which we may not be able to cover.

End of Chapter 10C