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

This page has two purposes: to describe how to implement computer language interpreters in general, and in particular to show how to implement a subset of the Scheme dialect of Lisp using Python. I call my language and interpreter Lispy(lis.py). Years ago, I showed how to write a Scheme interpreter in Java as well as one in Common Lisp. This time around the goal is to demonstrate, as concisely and accessibly 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 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 operator to the two numbers, yielding the value 3. We say we are evaluatingan expression when we determine its value; we would say that "1 + 2" evaluates to 3, and write that as "1 + 2" ⇒ 3.

Scheme syntax is different from most other languages you may be familiar with. Consider:

Java

```
if (x.val() > 0) {
  fn(A[i] + 1,
     new String[] {"one", "two"});
}
```

Scheme

```
(if (> (val x) 0)
    (fn (+ (aref A i) 1)
        (quote (one two)))
```

Java has a wide variety of syntactic conventions (keywords, infix operators, brackets, operator precedence, dot notation, quotes, commas, semicolons, etc.), 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, `+` and `<` and the like are symbols, exactly like `A`.
- Everything else is a list expression. A list is a "(", followed by zero or more expressions, followed by a ")". The first element of the list determines what it means.
- A list expression starting with a keyword, e.g. `(if ...)`, is known as a special form; we will see how each special form is interpreted.
- A list starting with a non-keyword, e.g. `(fn ...)`, is a function call.

The beauty of Scheme is that the full language only needs six basic special forms. (In comparison, Python has 110 syntactic forms and Java has 133.) Using parentheses for everything may seem unfamiliar, but it has the virtues of simplicity and consistency. (Some have joked that "Lisp" stands for "Lots of Irritating Silly Parentheses"; I think it stand for "Lisp Is Syntactically Pure".)

In this page we will cover all the important points of Scheme (omitting some minor details), but we will take two steps to get there.

Language 1: Lispy Calculator

Step one is to define a language I call Lispy Calculator that is a subset of Scheme using only three of the six special forms. 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 some things that are not offered in typical calculator languages: "if" expressions, and the definition of new variables, for example. Here is a table of all the allowable expressions in the Lispy Calculator language:

Expression	Syntax	Semantics and Example
variable reference	var	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
quotation	(quote exp)	Return the exp literally; do not evaluate it. Example: (quote (+ 1 2)) ⇒ (+ 1 2)
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 varexp)	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 quotethen 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

In the Syntax column of this table, var must be a symbol,number must be an integer or floating point number, and the other italicized words can be any expression. The notationarg... 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`.

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).

Here is a picture of the interpretation process:

program (str) ➡ $\boxed{\texttt{parse}}$ ➡ abstract syntax tree (list) ➡ $\boxed{\texttt{eval}}$ ➡ result (object)

And here is a short example of what we want `parse` and `eval` to be able to do:

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

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

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

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 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:

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

```
>>> program = "(begin (define r 10) (* pi (* r r)))"
>>> tokenize(program)
['(', 'begin', '(', 'define', 'r', '10', ')', '(', '*', 'pi', '(', '*', 'r', 'r', ')', ')', ')']
```

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 finally as a symbol. Here is the parser:

```
def parse(program):
    "Read a Scheme expression from a string."
    return read_from_tokens(tokenize(program))

def read_from_tokens(tokens):
    "Read an expression from a sequence of tokens."
    if len(tokens) == 0:
        raise SyntaxError('unexpected EOF while reading')
    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):
    "Numbers become numbers; every other token is a symbol."
    try: return int(token)
    except ValueError:
        try: return float(token)
        except ValueError:
            return Symbol(token)
```

`parse` works like this:

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

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

We have made some choices about the representation of Scheme objects. Here we make the choices explicit:

```
Symbol = str          # A Scheme Symbol is implemented as a Python str
List   = list         # A Scheme List is implemented as a Python list
Number = (int, float) # A Scheme Number is implemented as a Python int or float
```

We're almost ready to define `eval`. But we need one more concept first.

Environments

The function `eval` takes two arguments: an expression, `x`, that we want to evaluate, and an environment, `env`, in which to evaluate it. An environment is a mapping from variable names to their values. By default, `eval` will use a global environent that includes the names for a bunch of standard things (like the functions `max` and `min`). This environment can be augmented with user-defined variables, using the expression `(define variable value)`. For now, we can implement an environment as a Python dict of {variable: value} pairs.

```
Env = dict          # An environment is a mapping of {variable: value}
```

```
def standard_env():
    "An environment with some Scheme standard procedures."
    import math, operator as op
    env = Env()
    env.update(vars(math)) # sin, cos, sqrt, pi, ...
    env.update({
        '+':op.add, '-':op.sub, '*':op.mul, '/':op.div,
        '>':op.gt, '<':op.lt, '>=':op.ge, '<=':op.le, '=':op.eq,
        'abs':     abs,
        'append':  op.add,
        'apply':   apply,
        'begin':   lambda *x: x[-1],
        'car':     lambda x: x[0],
        'cdr':     lambda x: x[1:],
        'cons':    lambda x,y: [x] + y,
        'eq?':     op.is_,
        'equal?':  op.eq,
        '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),
        'procedure?': callable,
        'round':   round,
        'symbol?': lambda x: isinstance(x, Symbol),
    })
    return env

global_env = standard_env()
```

Evaluation: eval

We are now ready for the implementation of eval. As a refresher, we repeat the table of Scheme forms:

Expression	Syntax	Semantics and Example
variable reference	var	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
quotation	(quote exp)	Return the exp literally; do not evaluate it. Example: (quote (+ 1 2)) ⇒ (+ 1 2)
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 varexp)	Define a new variable and give it the value of evaluating the expression exp. Examples: (define r 10)
		If proc is anything other than one of the

procedure call	(proc arg...)	symbols if, define, or quotethen 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

Notice how closely the code for eval follows the table:

```
def eval(x, env=global_env):
    "Evaluate an expression in an environment."
    if isinstance(x, Symbol):      # variable reference
        return env[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)
    else:                          # (proc arg...)
        proc = eval(x[0], env)
        args = [eval(arg, env) for arg in x[1:]]
        return proc(*args)
```

We're done! You can see it all in action:

```
>>> eval(parse("(define r 10)"))
>>> eval(parse("(* pi (* r r))"))
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 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.

```
def repl(prompt='lis.py> '):
    "A prompt-read-eval-print loop."
    while True:
        val = eval(parse(raw_input(prompt)))
        if val is not None:
            print(schemestr(val))

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)
```

Here is repl in action:

```
>>> repl()
lis.py> (define r 10)
lis.py> (* pi (* r r))
314.159265359
lis.py> (if (> (* 11 11) 120) (* 7 6) oops)
42
lis.py>
```

Language 2: Full Lispy

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

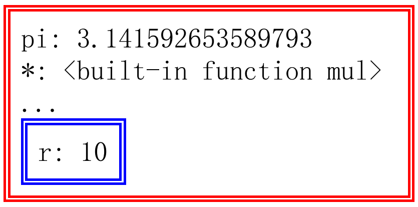
Expression	Syntax	Semantics and Example
assignment	`(set! var exp)`	Evaluate exp and assign that value to var, which must have been previously defined (with a `define` or as a parameter to an enclosing procedure). Example: `(set! r2 (* r r))`
procedure	`(lambda (var...) exp)`	Create a procedure with parameter(s) named var... and exp as the body. Example: `(lambda (r) (* pi (* r r)))`

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:

```
lis.py> (define circle-area (lambda (r) (* pi (* r r)))
lis.py> (circle-area 10)
314.159265359
```

The procedure call `(circle-area 10)` causes us to evaluate the body of the procedure, `(* pi (* r r))`, in an environment in which `pi`and `*` have the same global values they always did, but now `r`has the value 10. However, it wouldn't do to just set `r` to be 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 localvariable named `r` that we can set to 10 without worrying about interfering with any other variable that happens to have the same name. We will create a new kind of environment, one which allows for both local and global variables.

The idea is that when we evaluate `(circle-area 10)`, we will fetch the procedure body, `(* pi (* r r))`, and evaluate it in an environment that has `r` as the sole local variable, but also has access to the global environment. In other words, we want an environment that looks like this, with the local environment nested inside the "outer" global environment:



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.

It is clear that procedures and environments are intertwined, so let's define them together:

```python
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))

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()
```

We see that every procedure has three components: a list of parameter names, a body expression, and an environment that tells us what non-local variables are accessible from the body.

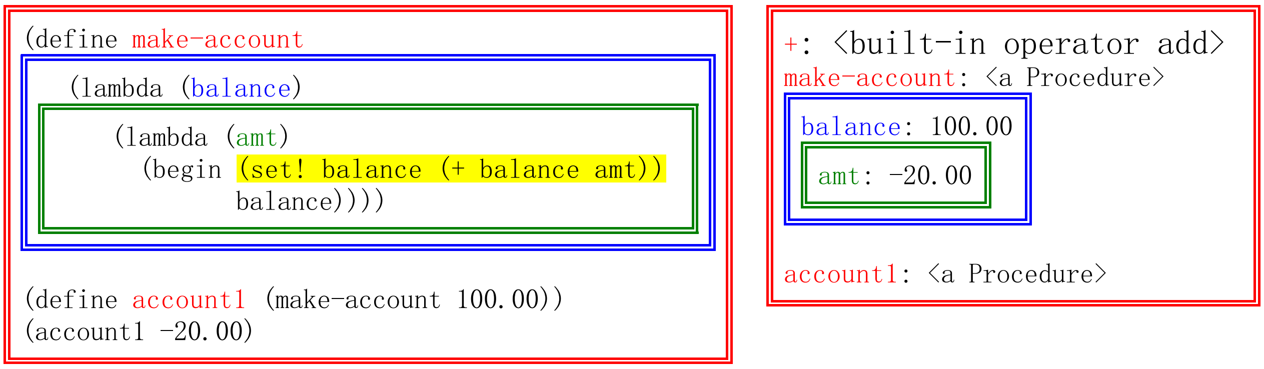
An environment is a subclass of `dict`, so it has all the methods that `dict` has. In addition there are two methods: the constructor `__init__` builds a new environment by taking a list of parameter names and a corresponding list of argument values, and creating a new environment that has those {variable: 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.

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.

```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(arg, env) for arg in x[1:]]
        return proc(*args)
```

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



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

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

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.

Let's see what we can do now:

```
>>> repl()
lis.py> (define circle-area (lambda (r) (* pi (* r r))))
lis.py> (circle-area 3)
28.274333877
lis.py> (define fact (lambda (n) (if (<= n 1) 1 (* n (fact (- n 1))))))
lis.py> (fact 10)
3628800
lis.py> (fact 100)
9332621544394415268169923885626670049071596826438162146859296389521759999322991
5608941463976156518286253697920827223758251185210916864000000000000000000000000
lis.py> (circle-area (fact 10))
4.1369087198e+13
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))
(1 1 2 3 5 8 13 21 34 55)
lis.py> (map fib (range 0 20))
(1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765)
```

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.

How Small/Fast/Complete/Good is Lispy?

In which we judge Lispy on several criteria:

- Small: Lispy is very small: 116 non-comment non-blank lines; 4K of source code. (An earlier version was just 90 lines, but had fewer standard procedures and was perhaps a bit too terse.) The smallest version of my Scheme in Java, Jscheme, was 1664 lines and 57K of source. Jscheme was originally called SILK (Scheme in Fifty Kilobytes), but I only kept under that limit by counting bytecode rather than source code. Lispy does much better; I think it meets Alan Kay's 1972 claim that you could define the "most powerful language in the world" in "a page of code."(However, I think Alan might disagree, because he would count the Python compiler as part of the code, putting mewell over a page.)

  ```
  bash$ grep "^\s*[^#\s]" lis.py | wc
       116     496    4274
  ```

- Fast: Lispy computes `(fact 100)` in 0.004 seconds. That's fast enough for me (although far slower than most other ways of computing it).

- Complete: Lispy is not very complete compared to the Scheme standard. Some major shortcomings:
  - Syntax: Missing comments, quote and quasiquote notation, # literals, the derived expression types (such as `cond`, derived from `if`, or `let`, derived from`lambda`), and dotted list notation.

    - Semantics: Missing call/cc and tail recursion.
    - Data Types: Missing strings, characters, booleans, ports, vectors, exact/inexact numbers. Python lists are actually closer to Scheme vectors than to the Scheme pairs and lists that we implement with them.
    - Procedures: Missing over 100 primitive procedures: all the ones for the missing data types, plus some others like `set-car!` and `set-cdr!`, because we can't implement `set-cdr!` completely using Python lists.
    - Error recovery: Lispy does not attempt to detect, reasonably report, or recover from errors. Lispy expects the programmer to be perfect.
- Good: That's up to the readers to decide. I found it was good for my purpose of explaining Lisp interpreters.

True Story

To back up the idea that it can be very helpful to know how interpreters work, here's a story. Way back in 1984 I was writing a Ph.D. thesis. This was before LaTeX, before Microsoft Word for Windows—we used troff. Unfortunately, troff had no facility for forward references to symbolic labels: I wanted to be able to write "As we will see on page @theorem-x" and then write something like "@(set theorem-x \n%)" in the appropriate place (the troff register \n% holds the page number). My fellow grad student Tony DeRose felt the same need, and together we sketched out a simple Lisp program that would handle this as a preprocessor. However, it turned out that the Lisp we had at the time was good at reading Lisp expressions, but so slow at reading character-at-a-time non-Lisp expressions that our program was annoying to use.

From there Tony and I split paths. He reasoned that the hard part was the interpreter for expressions; he needed Lisp for that, but he knew how to write a tiny C routine for echoing the non-Lisp characters and link it in to the Lisp program. I didn't know how to do that linking, but I reasoned that writing an interpreter for this trivial language (all it had was set variable, fetch variable, and string concatenate) was easy, so I wrote an interpreter in C. So, ironically, Tony wrote a Lisp program (with one small routine in C) because he was a C programmer, and I wrote a C program because I was a Lisp programmer.

In the end, we both got our theses done (Tony, Peter).

The Whole Thing

The whole program is here: lis.py).

Further Reading

To learn more about Scheme consult some of the fine books (byFriedman and Fellesein, Dybvig, Queinnec, Harvey and Wright orSussman and Abelson), videos (by Abelson and Sussman), tutorials (by Dorai, PLT, or Neller), or the reference manual.

I also have another page describing a more advanced version of Lispy.