(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 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, I 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" ⇒ 3.

Scheme syntax is different from most other programming languages. Consider:

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)))`

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`.
- 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.

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 of* ***I****rritating* ***S****illy* ***P****arentheses*"; I think it stand for "***L****isp* ***I****s* ***S****yntactically* ***P****ure*".)

In this page 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

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))
```

Here is a table of all the allowable expressions:

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`

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`.

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 ➡ `parse` ➡ abstract-syntax-tree ➡ `eval` ➡ result

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:

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

Here we apply tokenize to our sample program:

```
>>> 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 if it is neither of those, it must be a symbol. Here is the parser:

```
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)
```

`parse` works like this:

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

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

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

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`*`)`.

```
import math
import operator as op

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,
        '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`

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

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`

Here is the code for `eval`, which closely follows the table:

```
def eval(x: Exp, env=global_env) -> Exp:
    "Evaluate an expression in an environment."
    if isinstance(x, Symbol):        # variable reference
        return env[x]
    elif not 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)
```

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

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

```
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> (list (+ 1 1) (+ 2 2) (* 2 3) (expt 2 3))
lis.py>
```

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)`
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)))`

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 (+ 5 5))
314.159265359
```

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`. 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:

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

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:

```
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)

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))

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 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*).

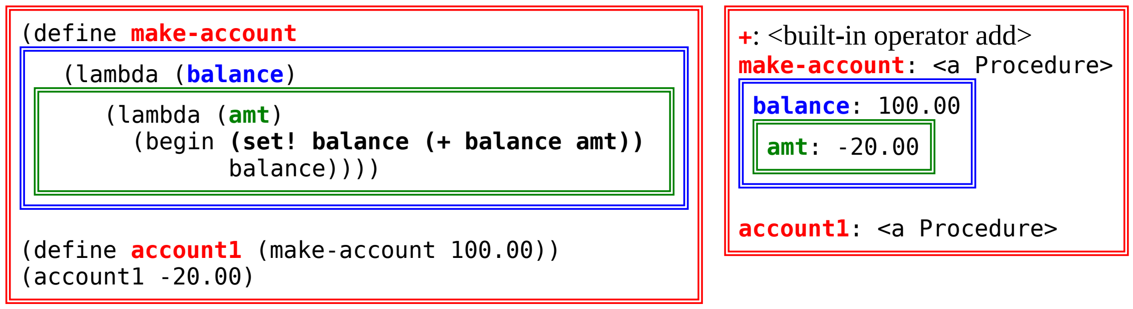
An environment is a subclass of `dict`, so it has all the methods that `dict` has. In addition there are two methods: the constructor `__init__` builds a new environment by taking a list of parameter names and a corresponding list of argument values, and creating a new environment that has those {variable: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

```
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
        return x
    op, *args = x
    if op == 'quote':            # quotation
```

```
        return args[0]
    elif op == 'if':             # conditional
        (test, conseq, alt) = args
        exp = (conseq if eval(test, env) else alt)
        return eval(exp, env)
    elif op == 'define':         # definition
        (symbol, exp) = args
        env[symbol] = eval(exp, env)
    elif op == 'set!':           # assignment
        (symbol, exp) = args
        env.find(symbol)[symbol] = eval(exp, env)
    elif op == 'lambda':         # procedure
        (parms, body) = args
        return Procedure(parms, body, env)
    else:                        # procedure call
        proc = eval(op, env)
        vals = [eval(arg, env) for arg in args]
        return proc(*vals)
```

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)
```

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

account1: <a Procedure>
```

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: 117 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 would disagree, because he would count the Python compiler as part of the code, putting me *well* over a page.)

```
bash$ grep "^\s*[^#\s]" lis.py | wc
     117     497    4276
```

- ***Fast:*** Lispy computes `(fact 100)` exactly in 0.003 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.
  - ***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 reading and 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 (by Friedman and Fellesein, Dybvig, Queinnec, Harvey and Wright or Sussman 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.

Peter Norvig

---

thedeemon • 9 years ago

Here's a Ruby analog in 60 lines with added support of continuations (call/cc, famous Scheme feature):
http://stuff.thedeemon.com/...
Requires Ruby 1.8.x.

22 • Reply • Share ›

Peter Norvig Mod thedeemon • 9 years ago

Very nice!

9 • Reply • Share ›

tarpsocks thedeemon • 8 years ago

60 lines but not an ounce of the clarity Mr. Norvig supplied.

4 • Reply • Share ›

Peter Norvig tarpsocks • 8 years ago

tarpsocks, I think it is only lacking clarity if you don't know the Ruby idioms. Once you do, it is clear.

One thing I would object to: call/cc should be a regular function, not a special form.

8 • Reply • Share ›

Show more replies

Lost Protocol • 8 years ago

Oh My Gawd. I searched for 'how to write simple language interpreters in python' and reached this page(thank you google). This article had two profound impacts on me. 1) I understood how to write lisp interpreters in python(obvious) and 2) I understood LISP for the first time. For atleast two years I have gone on and off about Lisp, never really 'getting' it. Always thought it was a lousy language no matter how much people loved it. That thought changed after this article. It has embarked me onto writing my own lisp dialect(not again) and also learning Lisp with a new attitude. Now I think of only one thing, I wish Python had equivalent of (defmacro) of lisp. :-) Thanks Mr. Norvig - I will buy your book on Artificial Intelligence to (somehow) support this webpage :-D.

18 • Reply • Share ›

Dave • 8 years ago

Chapter 11 of this book http://www.computingbook.org/ [http://www.computingbook. is all about a Python implementation of a mini-Scheme interpreter. (This problem set, http://www.cs.virginia.edu/..., is about extending that interpreter).

7 • Reply • Share ›

Sainamdar • 8 years ago

Here is a version of this in Javascript. I created this so that I could run through the examples/exercises in SICP while commuting. https://bitbucket.org/saina...

5 • Reply • Share ›

Sainamdar ➦ Sainamdar • 8 years ago

Here is another variation in Javascript that separates syntactic analysis from execution. https://bitbucket.org/saina...

2 • Reply • Share ›

affiszervmention ➦ Sainamdar • 6 years ago

strings don't seem to work on your interpreter

(= "str" "str") gives TypeError: outer.find is not a function

• Reply • Share ›

Shantanu Inamdar ➦ affiszervmention • 6 years ago

Thanks for the feedback affiszervmention. This is "by design". As Peter Norvig wrote in the completeness evaluation of Lispy, it is missing many data types like "Missing strings, characters, booleans, ports, vectors, exact/inexact numbers".

• Reply • Share ›

Tim Finin • 8 years ago

I was going over this to present it in a class and noticed that there is no variable bound to the empty list. We can use (quote ()) for the the empty list, as in (define x (cons 1 (cons 2 (quote ())))), but the way lists work in Python, (eq? (quote ()) (quote ())) is False. So I think it would be good to pre-define a global variable null initially bound to []: global_env['null'] = []

7 ^ | v • Reply • Share ›

PatHayes • 9 years ago

Very sweet. But you know, what would really be interesting would be to see how many lines of code you need to do this in something that doesn't have built-in recursion and in which you have to describe garbage collection. Something like an assembly language...

4 ^ | v • Reply • Share ›

Theo D'Hondt ➦ PatHayes • 9 years ago

Have a look at http://soft.vub.ac.be/francqui - it's "Slip" instead of "Lispy" and C instead of Python, hence trampolines and garbage collection ... Should make you happy.

4 ^ | v • Reply • Share ›

an691 ➦ Theo D'Hondt • 8 years ago

Hello Theo, are the slides and possibly videos of these lectures available ?

^ | v • Reply • Share ›

Peter Norvig Mod ➦ PatHayes • 9 years ago

Exactly, Pat! That is a different challenge, and a very good one. I got a nice note from Alan Kay reminding me that the "one page" did that, rather than relying on a host language to do it. (On the other hand, the "one page" did not include the definitions of individual procedures, including syntactic procedures.)

2 ^ | v • Reply • Share ›

Show more replies

Frakturfreund • 8 years ago

This code looks very nice, but i think that implementing a Lisp Interpreter in Python is some kind of cheating. Python is a high-level language, so you get very much for free. For an antipode, i suggest to have a look into Zozotez, a Lisp Interpeter in Brainfuck (which is a ridiculously low level toy language): https://code.google.com/p/z...

3 ^ | v • Reply • Share ›

Peter Norvig Mod ➦ Frakturfreund • 8 years ago

You are right -- we are relying on many features of Python: call stack, data types, garbage collection, etc. The next step would be to show a compiler to some sort of assembly language. I think either the Java JVM or the Python byte code would be good targets. We'd also need a runtime system with GC. I show the compiler in my PAIP book.

3 ^ | ⌄ • Reply • Share ›

Frakturfreund ➦ Peter Norvig • 8 years ago

Thanks for the hint! I’ll put the book on my christmas list :).

^ | ⌄ • Reply • Share ›

espin • 8 years ago

Very nice and inspiring!
FYI there is a nice small lisp implementation in C from Piumatra (a researcher with Alan Kay at VPRI)
at http://piumarta.com/softwar...

3 ^ | ⌄ • Reply • Share ›

missing paren • 5 years ago

there's a missing closing paren in below example

>> program = "(define area (lambda (r) (* 3.141592653 (* r r)))"

it should be

>> program = "(define area (lambda (r) (* 3.141592653 (* r r))))"

2 ^ | ⌄ • Reply • Share ›

fishyrecondite • 6 years ago

Fabulous...!!

2 ^ | ⌄ • Reply • Share ›

Reborn2266 • 7 years ago

Hi Peter
I am studying interpreters and compilers recently. Reading this article helps me a lot. I think this might also help other CS students in Taiwan so that I translate it into Traditional Chinese. If you mind, please let me know and I will remove the post.
http://reborn2266.blogspot....

Thanks.

2 ^ | ⌄ • Reply • Share ›

Peter Norvig Mod ➦ Reborn2266 • 7 years ago

Thank you for doing the translation. I appreciate it.

4 ^ | ⌄ • Reply • Share ›

Reborn2266 ➦ Peter Norvig • 7 years ago

Thank you very much. I will do my best to make sure the translation accurate. :)

2 ^ | ⌄ • Reply • Share ›

Jukka Välimaa • 9 years ago

Beautiful. Shouldn't cons be lambda x,y:[x]+[y]?

2 ^ | ⌄ • Reply • Share ›

Peter Norvig Mod ➦ Jukka Välimaa • 9 years ago

Shouldn't cons be lambda x,y:[x]+[y]? Mostly no. I'm trying to make it work for the case where y is a list (possibly empty) and x is an item. For example, (cons 1 '(2 3)) should be (1 2 3), and if x=1 and y=[2,3], then [1]+[2,3] is [1,2,3], so that's [x]+y. (I make it list(y) rather than y just in case y is a tuple, not a list.)

I still can't do the equivalent of (cons 1 2), which should yield (1 . 2); that can't be represented in my approach to lists.

2 • Reply • Share ›

Ralph Corderoy • 9 years ago

Thanks for another nice article. Would the "tokens.pop(0) # pop off ')'" be more clear as the faster, smaller byte-code, "del tokens[0]" since tokens[0], returned by pop(), isn't wanted.

2 • Reply • Share ›

Peter Norvig Mod • 9 years ago

Thanks for all the comments. Eric Cooper: I like your idea of eliminating the Procedure class. I had it because it is necessary to have access to the exp to do tail recursion elimination, but I don't need that in this version. bsagert: you are right that a proper repl needs to catch Exceptions. It also should be able to read multi-line input, which can't easily be done with the interface I have provided. Sainamdar: I did this because I still get a lot of interest in JScheme, and several people asked if I could do it in Python. Now the answer is: Yes I can! Massaro: you are right that a mutable pair can be implemented as a procedure with two variables. Or a vector of two elements. But the point is that if I did that, they would be different from Python's lists, and I couldn't use Python's map, etc.

2 • Reply • Share ›

Eric Cooper • 9 years ago

Here is a more meta-circular implementation of procedures:

def procedure(parms, exp, env):
"A user-defined Scheme procedure."
return lambda *args: eval(exp, Env(parms, args, env))

2 • Reply • Share ›

Peter Norvig Mod ➦ Eric Cooper • 9 years ago

Thanks, Eric. I adopted your solution here. I didn't use it originally because I was thinking ahead to Lispy2, where I couldn't use it. But for this version, your suggestion is right on.

Reply • Share ›

James • a year ago

Great article Mr Norvig!

Just wanted to point out that this line: `elif not isinstance(x, Number): # constant number`
should not have the `not`, because it's causing the function to

return it's argument which ends up just printing expressions instead of computing their value.

That said, I learnt a lot from playing with the code. Thank you!

1 ^ | ⌄ • Reply • Share ›

Kisitu Augustine • 5 years ago

may be am missing something here, what does (_, vars, exp) = x mean??

1 ^ | ⌄ • Reply • Share ›

steni ➜ Kisitu Augustine • 5 years ago

It means that whatever is in x, is "exploded into" three variables, of which the first is discarded (because it is not needed).

Let's say x is an array: x = [1, 2, 3].

Then (_, vars, exp) = x will discard the 1, and now vars = 2, and exp = 3.

^ | ⌄ • Reply • Share ›

Kisitu Augustine ➜ steni • 5 years ago

thanks steni

59 ^ | ⌄ • Reply • Share ›

cym13 • 5 years ago

Great interpretation of that magical part of SICP, happy to see that this book still inspire people !

1 ^ | ⌄ • Reply • Share ›

This comment is awaiting moderation. Show comment.

Konstantinos ➜ Lolol • 6 years ago

What /prog/, a board about programming, has to do with LISP, a programming language?
And /prog/ as a matter of fact liked LISP though not many are using it.
# Yes, fucking rule 14; well, I never liked following the rules anyway.

^ | ⌄ • Reply • Share ›

Lulzy • 7 years ago

in all honesty this is crap

1 ^ | ⌄ • Reply • Share ›

Alex Leibowitz • 3 months ago

I was trying to write a very sloppy recursive parse function:

def read_from_tokens(tokens: list, E=[]) -> Exp:
"convert a string into a tree-like expression"
# base case: the list is empty.
if len(tokens) == 0:
return E
# for the beginning of an expression, add sub-expression to the

```
# for the beginning of an expression, add sub-expression to the
array
now = tokens.pop(0)
if now == '(':
# read a new expression and put it into E
while len(tokens) and tokens[0] != ')':
E.append(read_from_tokens(tokens,[]))
return E
# for atom, process it and add it to E, then continue recursive
iteration
elif now == ')':
return E
else:
A = atom(now)
E.append(A)
return read_from_tokens(tokens, E)
```

but I couldn't figure out how to get it to properly handle a ')' -- when used incorrectly, the whole string should be thrown away as trash -- but a correct use should also be recognized. What I found was that either ')' was never allowed or else the nesting would work but the process would terminate the first time it encountered a ')' without a preceding '('.

Reply • Share ›

Alex Leibowitz • 3 months ago

I feel like the program should raise a syntax error when you give it the input "(" -- but instead it just crashes because it tries to read from an empty list.

Reply • Share ›

goodworldalchemy • 7 months ago

I think there's an error in the calculator parser. On the second line of your eval function, you should have `elif isinstance(x, Number):`. Right now you have `elif not isinstance(x, Number):`

Reply • Share ›