PyLisp

This is the reference for PyLisp, a small but functional Lisp interpreter written in pure Python. A number of its features are motivated by Scheme, but is in no way a complete version of that.

PyLisp was written because I needed something slightly more complex than a simple configuration file but something less complex than all of Python. There are a lot of sophisticated bits of Lisp trickery in PyLisp -- macros and closures for example -- but I wouldn't recommend using it for any genuinely complex development. It might be useful for people who like to experiment with language features.

I assume at least some familiarity with a Lisp-like language in this description. There are many Scheme tutorials on the web which should get you started.

The Core Langauge

PyLisp is a lisp-1. That means function names and variable names share the same namespace. This is like Scheme, but unlike Common Lisp (or Perl, for that matter) where you can have functions and variables with the same name.

PyLisp has the following types: numbers (real, integer or Python long integers), strings, pairs, lambdas, macros, logic objects and of course lists.

`(+ arg+)`
: As in many lisps, you can feed as many arguments to the math operators as makes sense: **`(+ 2 3 5.3 2 4)`**

`(- arg+)`
: Subtraction.

`(* arg+)`
: Multiplication.

`(/ arg+)`
: Division.

`(% arg+)`
: Modulo.

PyLisp's logic operations are a little different. If you don't pay attention, they appear to work in the normal way:

`(== arg0 arg1)` or **`(eql arg0 arg1)`**
: Equality test.

`(> arg0 arg1)`
`(>= arg0 arg1)`
`(< arg0 arg1)`
`(<= arg0 arg1)`
`(!= arg0 arg1)`

However, all of these will test string and symbol-name relations by alphabetical order, too:

```
lisp> (> 5 2)
        *true*
lisp> (> "wow" "fred")
        *true*
lisp> (< "wow" "fred")
        *false*
lisp> (== 'bob 'bob)
        *true*
lisp> (== 'bob 'fred)
        *false*
```

You can join several boolean tests with:

(and *arg+*)
: logical and

(or *arg+*)
: logical or

(not *arg*)
: logical negation

The logic gets tricky though when you use the **logic** function to create new logic object. In PyLisp, logic is fuzzy:

```
lisp> (logic 1.0)
        *true*
lisp> (logic 0.0)
        *false*
lisp> (and (logic 0.326) (logic 0.9532))
        (logic 0.326)
lisp> (not (and (logic 0.326) (logic 0.9532)))
        (logic 0.674)
```

There is a built-in variable ***true-enough*** with a default value of ***true***. It is used in conditionals, and allows you to lower the threshold at which a condition becomes true.

You can do math on logic objects, so it's easy to turn them into a percent, for example:

```
lisp> (* 100 (logic 0.3))
        30.0
```

PyLisp's lists are implemented using Python's lists. However, the standard sort of list functions are available:

(first *l*) or **(car *l*)**
(rest *l*) or **(cdr *l*)**
(replaca *l*)
: like **set-car!** in Scheme

(replacd *l*)
: like **set-cdr!** in Scheme

(second *l*)
(third *l*)

The standard list building functions are also available:

(cons *item list*)
(list *item)**
(append *list+*)

Technically, PyLisp supports pairs:

```
lisp> (cons 'a 'b)
        (a . b)
lisp> (def pair (cons 'a 'b))
        (a . b)
lisp> (replaca pair 'x)
        x
lisp> pair
        (x . b)
lisp> (replacd pair 'y)
        y
lisp> pair
        (x . y)
```

However, you can't do the sort of fiddling that results in turing a pair into a genuine list since they are not quite the same thing. PyLisp has pairs only to support a slow but traditional Lisp construction, the association list:

```
lisp> (def alist (list (cons 'a 'b) (cons 'wow 'fun)))
        ((a . b) (wow . fun))
lisp> (assoc 'a alist)
        b
lisp> (assoc 'wow alist)
        fun
lisp> (assoc 'spam alist)
        *false*
```

Symbols may have property lists. The properties themselves must also be symbols:

(put *symbol key value*)
: This assigns a symbol's property value

(get *symbol key*)
: This gets a symbol's property value

PyLisp has the standard Lisp conditionals and structuring forms:

(begin *form+*)
: The value of the last form evaluated is the value of the expression.

(if *test then-form { else-form }*)
: The **else-form** is optional.

(cond (*test consequent*)*)
: You can have as many of the test/consequent pairs as necessary. From the example code:

```
(def member
  (lambda (elt lst)
    (cond
     ((null? lst) *false*)
     ((== elt (first lst)) elt)
     (*true* (member elt (rest lst))))))
```

: Notice the use of ***true*** to handle the ultimate else case.

Predicates:

(null? *arg*)
: checks if the argument is a null (empty) list, true otherwise

(list? *arg*)
: checks if the argument is a list

(pair? *arg*)
: checks if the argument is a pair

(symbol? *arg*)
: checks if the argument is a symbol

(number? *arg*)
: checks if the argument is a number

(string? *arg*)
: checks if the argument is a string

(logic? *arg*)
: checks if the argument is a logic object

There are a few functions for messing around with the PyLisp environment, by which I mean the lexical and scoping environment. If some PyLisp code bombs deep within some code, the error will percolate to the top, but the environment the code was in remains. The interactive loop prompt will have a number in it telling you how deep in the stack you are. Also, it is possible to define things functions in nested environments, which will cause the enclosing environment to linger.

env
: a variable, contains the top-level environment

(the-environment)

returns the current, possibly nested, environment

(top-level)

unwinds the environment stack and brings you to the top level

(get-environment *obj*)

this grabs the environment of the defined object, which has to be a lambda or macro

(env-get *environment symbol*)

pulls the value of the symbol out of the given environment

(env-set *environment symbol value*)

binds the value to the symbol in the given environment

As an example:

```
lisp> (let ((x 5)) (def bar (lambda () x)))
        (lambda () x)
; The enclosing environment created by the let is bundled up with
; the lambda assigned to bar.  That environment can be manipulated:
lisp> (bar)
        5
lisp> (env-set (get-environment bar) 'x 33)
        33
lisp> (bar)
        33
```

As you have seen in the code above, PyLisp supports the **let** form:

(let (*init+*) *body*)

The initializer form may either be a simple symbol, or it may be a list with the symbol name first and code to initialize the value second. PyLisp does not support the named let. Any intializer may refer to the symbol bound in a previous initializer, so PyLisp's **let** is more like Scheme's **letrec**.

(with-py-hash-env *hash body*)

This takes a Python has table, and inserts every hash key into the current PyLisp namespace with the value of the hash value. This is very useful when used with the **intern** method: it gives you a way to pass a bunch of variables into the PyLisp environment easily and quickly:

```
from lisp import *
code = "(with-py-hash-env some-hash (+ a b))"

interp = Lisper()
interp.intern('some-hash', {'a':1, 'b':2, 'c': 3, 'd':4})
print interp.evalstring(code)   # will give 3
```

The form **def** is used to bind a value to a symbol. Thus, it is used not only to assign variable values in the traditional sense, but it is used to assign functions to their names.

(def *name value*)

PyLisp supports both functions and macros. The defining syntax is similar:

(lambda (*arg) *body*)**

The argument list may also be empty. Preserves the environment it is defined in if it isn't the top level.

(macro (*arg) *body*)**

The argument list may also be empty for macros. A macro also preserves the envionment it was defined in.

The argument list for both **lamdba** and **macro** may have a special **&rest** form. In that case, the name following the **&rest** is given the value of all remaining arguments as a list:

```
lisp> (def foo (lambda (a &rest b) (print a b)))
        (lambda (a &rest b) (print a b))
```

```
lisp> (foo 1 2 3 4 5)
1 (2 3 4 5)
        *true*
```

PyLisp's macros work as you'd expect most of the time:

```
(def incf
  (macro (x)
    `(setq ,x (+ ,x 1))))
(def bob 12)
(incf bob)
```

And if you need to take a look at what a macro is doing, you can expand it with **`macro-expand`**:

```
lisp> (macro-expand (incf bob))
        (begin (setq bob (+ bob 1)))
```

The interactive PyLisp interpreter is a bit brittle. For example, you can't spread code over multiple lines. But you can put all your PyLisp code into a file and load it with the function **`(load "filename")`**.

The Python Interface

Calling PyLisp from within Python is as simple as instantiating the **`Lisper`** class:

```
from lisp import Lisper

interp = Lisper()
```

Once you have the interpreter instance you can do several things:

- **`interp.repl()`** - this starts up an interactive read-eval-print loop
- **`interp.read(some_file)`** - reads in a file of PyLisp code and evaluates all of it. This is very useful if you want to make changes to the PyLisp environment before handing control off to the **`repl()`** method.
- **`interp.intern(name, value)`** - this lets you shove a new name and value into the current PyLisp environment.

Extending PyLisp

Wanting.

Explain how to add functions written in Python. Adding syntax, too.

This page last altered on: Fri Aug 10 14:23:31 2001
Send comments or problems to
William Annis.