Lisp interpreter in Python (Lispy)

Interpreter has two parts

a) Parsing: It 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. The Lispy parser is implemented with the function parse.
b) Execution: The internal representation is then processed according to the *semantic rules* of the language, thereby carrying out the computation. Here it is implemented with function eval

The tokens here are parentheses, symbols and numbers. str.split is used for lexical analysis. 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.

The function parse takes 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 ')' its a syntax error. If it is a '(', then we start building up a list of sub-expressions until we get a matching ')'. Any non-parenthesis token must be a symbol or number. For each non-paren token, first try to interpret it as an int, then as a float, and finally as a symbol.

Symbol = str makes a Scheme Symbol implemented as a Python str

List = list makes a Scheme list interpreted as a Python list.

Number = (int, float) makes a Scheme Number implemented as a Python int or float.

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 environment 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*). An environment is implemented as a Python dictionary of {variable: value} pairs.

The function eval takes care of the following expressions variable reference (Syntax: var), constant literal (Syntax: number), quotation (Syntax: quote exp),

conditional (Syntax: if test conseq alt), definition (Syntax: define var exp), procedurecall (Syntax: proc arg..).

The function repl is a read-eval-print loop which is an easy way to enter an expression and see it immediately read, evaluated, and printed, without having to go through a lengthy build/compile cycle. The function schemestr returns a string representing a Scheme object.

The code is then extended with two special forms which nearly completes Scheme-subset to support expressions assignment and procedure.

The procedure call (circle-area 20) 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 20. However, it wouldn't do to just set r to be 20 in the global environment. We will create a new kind of environment, one which allows for both local and global variables in case we are using r for some other purpose.

The idea is that when we evaluate (circle-area 20), 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.

Every procedure has three components: a list of parameter names, a body expression, and an environment that tells us which 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.

New definition of eval. The clause for variable reference has changed: we 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.

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.

This interpreter is small (116 non-comment non-blank lines), fast (fact 100) computes in 0.004 seconds. It is not complete compared to the scheme standard as there are some major shortcomings in Syntax, Semantics, Data types, Procedures, Error recovery.

Lisp interpreter in Java (JScheme)

Environment.java:

Environments allow you to look up the value of a variable, given its name. It keeps a list of variables and values, and a pointer to the parent environment. If a variable list ends in a symbol rather than null, it means that symbol is bound to the remainder of the values list.

It extends SchemeUtils class. It consists of the following:

i) A constructor to extend an environment with var/val pairs.
ii) A constructor to construct an empty environment: no bindings.
iii) A function to find the value of a symbol in this environment or a parent. It checks if its bound locally if not looks for the parent.
iv) A function to add a new variable , value pair to this environment.
v) A function to set the value of an existing variable.
vi) A function to check if there is an appropriate number of vals for these vars.

Closure.java:

A closure is a user-defined procedure. It is closed over the environment in which it was created. In order to apply the procedure, it binds the parameters to the passed in variables, and evaluates the body.

It consists of the following:

i) A constructor which makes a closure from a parameter list, body, and environment.
ii) A function to apply a closure to a list of arguments.

Continuation.java:

It extends procedure. It has a constructor which assigns a runtime exception cc.

A function which throws the run time exception cc and assigns value to args passed.

InputPort.java:

InputPort is to Scheme as InputStream is to Java. It has the following:

i) A constructor to construct an InputPort from an InputStream.

ii) A constructor to construct an InputPort from a Reader.
iii) A function which reads and returns a Scheme character or EOF.
iv) A function which push a character back to be re-used later.
v) A function which pops off the previously pushed character.
vi) A function which peeks at and return the next Scheme character as an int, -1 for EOF.

JavaMethod.java:

It extends Procedure class. It has the following:

i) A function to apply the method to a list of arguments.
ii) A function to convert a list of Objects into an array. The argClasses enables us to convert between Double and Integer, something Java won't do automatically.
iii) A function to convert a list of class names into an array of classes.

Macro.java:

It consists of the following:

i) A constructor which makes a macro from a parameter list, body, and environment.
ii) A function which replaces the old cons cell with the macro expansion, and return it.
iii) A function to Macro expand an expression.

Pair.java:

A Pair has two fields, first and rest (or car and cdr). The empty list is represented by null. The methods that we might expect here, like first, second, list, etc. are instead static methods in class SchemeUtils.

It has the following:

i) Functions to find first element of the pair, the other element of the pair, build a pair from the two components.
ii) Functions to return a String representation of the pair, build up a String representation of the Pair in a StringBuffer.

Primitive.java:

A primitive is a procedure that is defined as part of the Scheme report, and is implemented here in Java code. For every function like car, cdr, cadddr, write, vector => it is mapped to corresponding functions in Java which performs the same function.

It has got separate sections for Booleans, Equivalence predicates, Lists and pairs, Symbols, Numbers, Characters, Strings, Vectors, Control features, Input and Output.

Map proc over a list of lists of args, in the given interpreter. If result is non-null, it accumulates the results of each call there and returns that at the end otherwise just returns null.

Primitives.scm defines some standard scheme macros. It also defines some extensions for delay and time, quasiquote.

Procedure.java:

It extends SchemeUtils

It consists of a function which coerces a Scheme object to a procedure.

Scheme.java:

It represents a scheme interpreter.

It consists of the following:

i) A function which creates a Scheme interpreter and load an array of files into it.
ii) A function which creates a new Scheme interpreter, passing in the command line args as files to load, and then enter a read eval write loop.
iii) A function which prompts, reads, eval, and writes the result also sets up a catch for any RuntimeExceptions encountered.
iv) A function which evals all the expressions in a file, calls load(InputPort).
v) A function which evals all the expressions coming from an InputPort.
vi) A function which evaluates an object x in an environment. This is done with tail recursion.
vii) A function which evaluates in the global environment, evaluates each of a list of expressions.

viii) A function which reduces a cond expression to some code which, when evaluated, gives the value of the cond expression. It is done that way to maintain tail recursion.

SchemePrimitive.java:

It holds a string representation of some Scheme code. It is an interface between primitives.scm and java.

SchemeUtils.java

It is similar to eval function in python. It consists of functions which convert or coerce objects to the right type ie Scheme object to a boolean, Scheme object to a double etc.

It has got the following:

i) A function which converts x to a Java String giving its external representation Strings and characters are quoted.
ii) A function converts x to a Java String giving its external representation.
iii) A function which converts a char to a Character, coerces a Scheme object to a Scheme string, which is a char[], coerces a Scheme object to a Scheme symbol, which is a string, coerces a Scheme object to a Scheme vector, which is a Object[], coerces a Scheme object to a Scheme input port, which is an InputPort. If the argument is null, returns interpreter.input.
iv) A function which coerces a Scheme object to a Scheme input port, which is a PrintWriter. If the argument is null, returns System.out.
v) Some manipulation routines like Common Lisp first; car of a Pair, or null for anything else.
vi) A function which converts a list of Objects to a Scheme vector, which is a Object[].
vii) A function which converts a list of characters to a Scheme string, which is a char[].
viii) A function which converts a Scheme object to its printed representation, as a java String (not a Scheme string). If quoted is true, use "str" and #\c, otherwise use str and c. StringBuffer is passed to accumulate the results.
ix) A function which writes the object to a port. If quoted is true, use "str" and #\c, otherwise use str and c.

Comparisions:

i) Java code (Jscheme) is big (about 1664 lines and 57K of source) whereas Python code (Lispy) is short (116 non-comment, non-blank lines and 4K of source code). Jscheme was also called Scheme in Fifty Kilobytes (SILK).

ii) Lispy is very fast for instance (fact 100) is computed in 0.004s whereas Jscheme is slow.

iii) Jscheme consists of many classes, each of which perform specified tasks which makes it more readable, understandable for the user. Lispy on the other hand is less readable and lack of sufficient commenting in Lispy can also factor to this.

Improvement in Lispy:

Lispy is not very complete compared to the Scheme standard.

i) **Syntax**: Missing comments, quote and quasiquote notation, # literals, the derived expression types (such ascond, derived from if, or let, derived from lambda), and dotted list notation.

ii) **Semantics**: Missing call/cc and tail recursion.

iii) **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.

iv) **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.

v) **Error recovery**: Lispy does not attempt to detect, reasonably report, or recover from errors. It expects the programmer to be perfect.

Improvement in Jscheme:

1. The distinction between exact and inexact is not right. It represents all numbers as double precision floating point.
2. Better support for errors: catch some errors that kill the system, allow user to interrupt and get into a break loop or back to top level, provide for recovery from errors.
3. Better integration with Java methods.
4. Better way to use multi-threading.
5. Making everything 10 to 50 times faster.