

# LISP Interpreter Lab 4

## CS411 – Dr. Shaffer

### Due: November 12

---

## 1 Introduction

In this lab we will introduce `apply`, sequencing, strings, top-level bindings and 0 argument lambda expressions to our language. We will also add some important functionality to our parser/scanner. You should use the *test first* principle throughout this lab. If you can't write the test case then you surely can't implement the functionality...

## 2 LISPNil

If you haven't done so already you need to modify your parser (and scanner) to handle `nil` as a special list. That is, the expression `'()` should parse to the sole instance of `LISPNil` and not to a list with a `nil` `car` and `cdr`. Some parsers also accept the unquoted empty list although such acceptance is by no means universal (umb-scheme complains, for example). For us it is a matter of convenience for `()` to also parse as `nil` so add that as well.

## 3 Apply and some built-in functions

### 3.1 How it works

Basically, for now we're going to get a minimal version of function application working. The final version will look somewhat different from what you're building in this lab so it is essential to write thorough tests so you can tear things apart in the next labs.

### 3.2 Apply

Write test cases for the functions `car`, `cdr`, `cons`, and `eq?`. While you haven't written these functions yet, you should know how they work. Test first! Use the spec as a guide for what things must be `eq?`.

Now, to implement these functions add code in your `LISPCons>>evalIn:` method to eval each element of the `cdr` of the list producing a new list and send `applyTo:` to the `car` of the original list with the `cdr` as the argument. For the moment, don't eval the `car` of the list since that causes problems for built-in functions. We'll fix this later.

Implement `LISPsymbol>>applyTo:` so that it performs the desired function based on the `value` of the symbol. Right now this is just going to be one ugly "switch" statement (well, just use `ifTrue:`) but we will remedy that in the next lab.

The thing to keep in mind at this point is that built in function should be implemented in `LISPSymbol>>applyTo:` and built in forms (forms that don't evaluate their arguments) should be implemented in `LISPCons>>evalIn:`.

## 4 Sequencing

Most programming languages support the concept of sequencing (evaluating statements in order). We don't need to extend our syntax to add sequencing to our LISP language...we just need to add a new function: `begin`. The `begin` function really does nothing (since evaluation of its arguments is automatic). Its value is the value of its last argument. So, for example:

```
(begin
  (print (quote hello))
  (print (quote world))
  (quote a))
```

should print the symbols `hello` and `world` in the transcript and return the `LISPSymbol a` as its value. Write test cases for the `begin` function in a new test class. Implement the `begin` function in the `LISPSymbol>>applyTo:` method.

This is throw away code since we can implement `begin` as a `lambda`. We'll take care of that in the next lab.

## 5 Strings

Let's add a string data type to our LISP. Strings will be identified by their enclosing double quotes (`"`). Write test cases for your parser and your interpreter (string eval to themselves). Add support for strings to your parser (make sure that any text besides a double quote is valid in a string). Add a `LISPString` class to your system to hold the contents of the string. Keep in mind that strings are not atoms. Have `LISPString` respond to `lispPrintOn:` simply by printing it's contents (without double quotes) and `printOn:` by printing itself prefixed with `"L_"` and the double quoted string.

## 6 Top level bindings

In our LISP there will be a top level "namespace" where, as a last resort, variable bindings are resolved. We are not ready to implement the full machinery of name resolution yet so we'll stick with our `Dictionary` as a place to store bindings. A variable becomes bound to a value when a call to the form `define` is evaluated. For example, `(define a 10)` should bind the symbol `a` to the `LISPInteger 10`. Note that the first argument to `define` is not evaluated but the second is. For example `(define a (quote a))` would bind `a` to the `LISPSymbol a`. As a simple example,

```
(begin
  (define a 10)
  a)
```

should produce the `LISPInteger` 10. We accomplish this by supporting `define` in our `LISPCons>>evalIn:` method since it does not eval its first argument. As another example:

```
(begin
  (define a 't)
  (define b (if a 10 20))
  b)
```

should produce the `LISPInteger` 10.

## 7 Constructing and applying Lambdas

For the moment we avoid worrying about closures and simply deal with the zero argument lambda, also referred to as a “thunk”. Create a new class, `LISPLambda`, with instance variables to hold the argument list, definition context and the body of the lambda. Add support to `LISPCons>>evalIn:` so that a lambda expression creates a `LISPLambda` instance. So, for example, the expression `(lambda '() 5)`, when evaluated should produce a `LISPLambda` with a body consisting of the `LISPInteger` 5, the `LISPNil` as its argument list, and the definition context will simply be the dictionary supplied to `evalIn:`.

Now, when a `LISPLambda` gets sent the `applyTo:` message, it should evaluate (`evalIn:`) its body in the context it stored when it was created. As a simple test case:

```
(begin
  (define a (lambda '() (define z 10)))
  (define b (lambda '() (define f 20)))
  (a))
```

should cause `z` to be bound to 10 but should leave `f` undefined (since `b` is never called). This test case is admittedly awkward looking but it does make sure that you don't evaluate the body of the lambda unless it gets called. You should be able to write many other test cases.