Little Lisp interpreter

Mary Rose Cook JUL 22, 2013

Little Lisp is an interpreter that supports function invocation, lambdas, lets, ifs, numbers, strings, a few library functions, and lists. I wrote it for a lightning talk at Hacker School to show how easy it is to write an interpreter. The code is 116 lines of JavaScript. I will explain how it works.

First, let's learn some Lisp.

Basic Lisp

This is an atom, the simplest Lisp form:

```
1
```

This is another atom, a string:

```
"a"
```

This is an empty list:

```
()
```

This is a list containing an atom:

```
(1)
```

This is a list containing two atoms:

```
(1 2)
```

This is a list containing an atom and another list:

```
(1 (2))
```

This is a function invocation. A function invocation comprises a list where the first element is the function and the rest of the elements are the arguments. `first` takes one argument, `(1 2)`, and returns `1`.

```
(first (1 2))

 => 1
```

This is a lambda, which is a function definition. The function takes a parameter, `x`, and just returns it.

```
(lambda (x)
 x)
```

This is a lambda invocation. A lambda invocation comprises a list where the first element is a lambda and the rest of the elements are the arguments. The lambda takes one argument, `"Lisp"`, and returns it.

```
((lambda (x)
  x)
 "Lisp")

 => "Lisp"
```

How Little Lisp works

Writing a Lisp interpreter is really easy.

The code for Little Lisp has two parts: the parser and the interpreter.

The parser

Parsing has two phases: tokenizing and parenthesizing.

`tokenize()` takes a string of Lisp code, puts spaces around every parenthesis and splits on whitespace. For example, it takes something like `((lambda (x) x) "Lisp")`, transforms it into `( ( lambda ( x )` `x ) "Lisp" )` and transforms that into `['(', '(', 'lambda', '(', 'x', ')', 'x', ')', '"Lisp"', ')']`.

```
var tokenize = function(input) {
  return replace(/\(/g, ' ( ')
          .replace(/\)/g, ' ) ')
          .trim()
          .split(/\s+/);
};
```

`parenthesize()` takes the tokens produced by `tokenize()` and produces a nested array that mimics the structure of the Lisp code. Each atom in the nested array is labelled as an identifier or a literal. For example, `['(', '(', 'lambda', '(', 'x', ')', 'x', ')', '"Lisp"', ')']` is transformed into:

```
[[{ type: 'identifier', value: 'lambda' }, [{ type: 'identifier', value: 'x' }],
  { type: 'identifier', value: 'x' }],
 { type: 'literal', value: 'Lisp' }]
```

`parenthesize()` goes through the tokens, one by one. If the current token is an opening parenthesis, it starts building a new array. If the current token is an atom, it labels it with its type and appends it to the current array. If the current token is a closing parenthesis, it stops building the current array and continues building the enclosing array.

```
var parenthesize = function(input, list) {
  if (list === undefined) {
    return parenthesize(input, []);
  } else {
    var token = input.shift();
    if (token === undefined) {
      return list.pop();
    } else if (token === "(") {
      list.push(parenthesize(input, []));
      return parenthesize(input, list);
    } else if (token === ")") {
      return list;
    } else {
      return parenthesize(input, list.concat(categorize(token)));
    }
  }
};
```

When `parenthesize()` is first called, the `input` parameter contains the array of tokens returned by `tokenize()`. For example:

```
['(', '(', 'lambda', '(', 'x', ')', 'x', ')', '"Lisp"', ')']
```

When `parenthesize()` is first called, the `list` parameter is undefined. Lines 2-3 run and `parenthesize()` recurses with `list` set to an empty array.

In the recursion, line 5 runs and removes the first opening parenthesis from `input`. Line 9 starts a new, empty list by recursing with a new, empty array.

In the recursion, line 5 runs and removes another opening parenthesis from `input`. Line 9 starts another new, empty list by recursing with another new, empty array.

In the recursion, `input` is `['lambda', '(', 'x', ')', 'x', ')', '"Lisp"', ')']`. Line 14 runs with `token` set to `lambda`. It calls `categorize()` and passes `lambda` as the `input` argument. Line 7 of `categorize()` runs and returns an object with `type` set to `identifier` and `value` set to `lambda`.

```
1   var categorize = function(input) {
2     if (!isNaN(parseFloat(input))) {
3       return { type:'literal', value: parseFloat(input) };
4     } else if (input[0] === '"' && input.slice(-1) === '"') {
5       return { type:'literal', value: input.slice(1, -1) };
6     } else {
7       return { type:'identifier', value: input };
8     }
9   };
```

Line 14 of `parenthesize()` appends to `list` the object returned by `categorize()` and recurses with the rest of the input and `list`.

```
1   var parenthesize = function(input, list) {
2     if (list === undefined) {
3       return parenthesize(input, []);
4     } else {
5       var token = input.shift();
6       if (token === undefined) {
7         return list.pop();
8       } else if (token === "(") {
9         list.push(parenthesize(input, []));
10        return parenthesize(input, list);
```

```
11      } else if (token === ")") {
12        return list;
13      } else {
14        return parenthesize(input, list.concat(categorize(token)));
15      }
16    }
17  };
```

In the recursion, the next token is a parenthesis. Line 9 of `parenthesize()` starts a new, empty list by recursing with an empty array. In the recursion, `input` is `['x', ')', 'x', ')', '"Lisp"', ')']`. Line 14 runs with `token` set to `x`. It makes a new object with a value of `x` and a type of `identifier`. It appends this object to `list` and recurses.

In the recursion, the next token is a closing parenthesis. Line 12 runs and returns the completed `list`: `[{ type: 'identifier', value: 'x' }]`.

`parenthesize()` continues recursing until it has processed all of the input tokens. It returns the nested array of typed atoms.

`parse()` is the successive application of `tokenize()` and `parenthesize()`:

```
var parse = function(input) {
  return parenthesize(tokenize(input));
};
```

Given a starting input of `((lambda (x) x) "Lisp")`, the final output of the parser is:

```
[[{ type: 'identifier', value: 'lambda' }, [{ type: 'identifier', value: 'x' }],
  { type: 'identifier', value: 'x' }],
 { type: 'literal', value: 'Lisp' }]
```

The interpreter

After parsing is complete, interpreting begins.

`interpret()` receives the output of `parse()` and executes it. Given the output from the parsing example above, `interpret()` would construct a lambda and invoke it with the argument `"Lisp"`. The lambda invocation would return `"Lisp"`, which would be the output of the whole program.

As well as the input to execute, `interpret()` receives an execution context. This is the place where variables and their values are stored. When a piece of Lisp code is executed by `interpret()`, the execution context contains the variables that are accessible to that code.

These variables are stored in a hierarchy. Variables in the current scope are at the bottom of the hierarchy. Variables in the enclosing scope are in the level above. Variables in the scope enclosing the enclosing scope are in the level above that. And so on. For example, in the following code:

```
((lambda (a)
  ((lambda (b)
    (b a))
   "b"))
 "a")
```

On line 3, the execution context has two active scopes. The inner lambda forms the current scope. The outer lambda forms an enclosing scope. The current scope has `b` bound to `"b"`. The enclosing scope has `a` bound to `"a"`. When line 3 runs, the interpreter tries to look up `b` in the context. It checks the current scope, finds `b` and returns its value. Still on line 3, the interpreter tries to look up `a`. It checks the current scope and does not find `a`, so it tries the enclosing scope. There, it finds `a` and returns its value.

In Little Lisp, the execution context is modeled with an object made by calling the `Context` constructor. This takes `scope`, an object that contains variables and their values in the current scope. And it takes `parent`. If `parent` is `undefined`, the scope is the top, or global scope.

```
var Context = function(scope, parent) {
  this.scope = scope;
  this.parent = parent;

  this.get = function(identifier) {
    if (identifier in this.scope) {
      return this.scope[identifier];
    } else if (this.parent !== undefined) {
      return this.parent.get(identifier);
    }
  };
};
```

We have seen how `((lambda (x) x) "Lisp")` gets parsed. Let us see how the parsed code gets executed.

```
var interpret = function(input, context) {
  if (context === undefined) {
    return interpret(input, new Context(library));
  } else if (input instanceof Array) {
    return interpretList(input, context);
  } else if (input.type === "identifier") {
```

```
7       return context.get(input.value);
8     } else {
9       return input.value;
10    }
11  };
```

The first time `interpret()` is called, `context` is `undefined`. Lines 2-3 are run to make an execution context.

When the initial context is instantiated, the constructor function takes the `library` object. This contains the functions built in to the language: `first`, `rest` and `print`. These functions are written in JavaScript.

`interpret()` recurses with the original `input` and the new `context`.

`input` contains the full example output from the parsing section:

```
[[{ type: 'identifier', value: 'lambda' }, [{ type: 'identifier', value: 'x' }],
  { type: 'identifier', value: 'x' }],
 { type: 'literal', value: 'Lisp' }]
```

Because `input` is an array and `context` is defined, lines 4-5 are run and `interpretList()` is called.

```
1   var interpretList = function(input, context) {
2     if (input.length > 0 && input[0].value in special) {
3       return special[input[0].value](input, context);
4     } else {
5       var list = input.map(function(x) { return interpret(x, context); });
6       if (list[0] instanceof Function) {
7         return list[0].apply(undefined, list.slice(1));
8       } else {
9         return list;
10      }
11    }
12  };
```

In `interpretList()`, line 5 maps over the input array and calls `interpret()` on each element. When `interpret()` is called on the lambda definition, `interpretList()` gets called again. This time, the `input` argument to `interpretList()` is:

```
[{ type: 'identifier', value: 'lambda' }, [{ type: 'identifier', value: 'x' }],
```

```
  { type: 'identifier', value: 'x' }]
```

Line 3 of `interpretList()` gets called, because `lambda`, the first element in the array, is a special form. `special.lambda()` is called to create the lambda function.

```
1   var special = {
2     lambda: function(input, context) {
3       return function() {
4         var lambdaArguments = arguments;
5         var lambdaScope = input[1].reduce(function(acc, x, i) {
6           acc[x.value] = lambdaArguments[i];
7           return acc;
8         }, {});
9
10        return interpret(input[2], new Context(lambdaScope, context));
11      };
12    }
13  };
```

`special.lambda()` takes the part of the input that defines the lambda. It returns a function that, when invoked, invokes the lambda on some arguments.

Line 3 begins the definition of the lambda invocation function. Line 4 stores the arguments passed to the lambda invocation. Line 5 starts creating a new scope for the lambda's invocation. It reduces over the part of the input that defines the parameters of the lambda: `[{ type: 'identifier', value: 'x' }]`. It adds a key/value pair to the lambda scope for each lambda parameter in `input` and argument passed to the lambda. Line 10 invokes the lambda by calling `interpret()` on the lambda body: `{ type: 'identifier', value: 'x' }`. It passes in the lambda context that contains the lambda's scope and the parent context.

The lambda is now represented by the function returned by `special.lambda()`.

`interpretList()` continues mapping over the `input` array by calling `interpret()` on the second element of the list: the `"Lisp"` string.

```
var interpret = function(input, context) {
  if (context === undefined) {
    return interpret(input, new Context(library));
  } else if (input instanceof Array) {
    return interpretList(input, context);
  } else if (input.type === "identifier") {
    return context.get(input.value);
  } else {
```

```
9       return input.value;
10    }
11  };
```

This runs line 9 of `interpret()` which just returns the `value` attribute of the literal object: `'Lisp'`. The map operation on line 5 of `interpretList()` is complete. `list` is:

```
[function(args) { /* code to invoke lambda */ },
 'Lisp']
```

Line 6 of `interpretList()` runs and finds that the first element of `list` is a JavaScript function. This means that the list is an invocation. Line 7 runs and invokes the lambda, passing the rest of `list` as arguments.

```
1   var interpretList = function(input, context) {
2     if (input.length > 0 && input[0].value in special) {
3       return special[input[0].value](input, context);
4     } else {
5       var list = input.map(function(x) { return interpret(x, context); });
6       if (list[0] instanceof Function) {
7         return list[0].apply(undefined, list.slice(1));
8       } else {
9         return list;
10      }
11    }
12  };
```

In the lambda invocation function, line 8 calls `interpret()` on the lambda body, `{ type: 'identifier', value: 'x' }`.

```
1   function() {
2     var lambdaArguments = arguments;
3     var lambdaScope = input[1].reduce(function(acc, x, i) {
4       acc[x.value] = lambdaArguments[i];
5       return acc;
6     }, {});
7
8     return interpret(input[2], new Context(lambdaScope, context));
9   };
```

Line 6 of `interpret()` finds that `input` is an identifier atom. Line 7 looks up the identifier, `x`, in `context` and returns `'Lisp'`.

```
1  var interpret = function(input, context) {
2    if (context === undefined) {
3      return interpret(input, new Context(library));
4    } else if (input instanceof Array) {
5      return interpretList(input, context);
6    } else if (input.type === "identifier") {
7      return context.get(input.value);
8    } else {
9      return input.value;
10   }
11 };
```

`'Lisp'` is returned by the lambda invocation function, which is returned by `interpretList()`, which is returned by `interpret()`, and that's it.

Go to the GitHub repository to see all the code. And look at lis.py, the dazzlingly simple Scheme interpreter that Peter Norvig wrote in Python.