

Discover Lisp in your Web Browser with Javascript

Alan Dipert

July 3rd, 2011

Implementing Lisp's Ideas

Many ideas introduced originally in Lisp, such as conditional execution (the `if-then-else` construct), garbage collection, and first-class functions¹, can be found in modern non-Lisp programming languages.

Other ideas pioneered by Lisp, like the idea that programs can be composed solely of *expressions*, and *homoiconicity*, or the idea that a program can be written in its own data structures, are rarer.

Learning a Lisp dialect, whether it's Clojure, Scheme, or Common Lisp, is a great way to see for yourself how powerful all of these ideas can be when combined.

An even better way to understand the ideas is to implement your own Lisp²³.

After we explore some fundamental concepts, we'll look at how a language like Javascript works.

Then, we'll build our own small Lisp to Javascript compiler on top of Javascript in about 32 lines of Javascript code.

Expression vs. Statement

Programs in languages like Javascript are composed of two things:

- statements
- expressions

¹What Made Lisp Different by Paul Graham

²The Roots of Lisp by Paul Graham

³Lisp in Small Pieces

Lisp programs are composed only of expressions.

Both statements and expressions are, in some sense, instructions for the computer to do some piece of work. The difference between them is that a statement does not return a value, but an expression does.

Consider Javascript's `if-then-else` statement:

```
if (x > 10) {  
  alert("x was bigger than 10!");  
} else {  
  alert("x was not bigger than 10!")  
}
```

We call `alert` inside both the `then` and `else` bodies of the expression because `if` does not return a value. If it did, we could pass it directly to a single `alert` call, as this wishful pseudo-Javascript demonstrates:

```
alert(if (x > 10) {  
  return "x was bigger than 10!"  
} else {  
  return "x was not bigger than 10!"  
});
```

If we eliminate statements from our wishful pseudo-Javascript entirely, there's no longer a need for the `return` operator. It's assumed that values "return" themselves:

```
alert(if (x > 10) {  
  "x was bigger than 10!"  
} else {  
  "x was not bigger than 10!"  
});
```

Javascript actually provides an `if-then-else` construct that returns a value, the *ternary operator*⁴, but it is seldom used and often discouraged⁵.

With expressions, it's possible to write code in the *imperative style* the first example demonstrated. The opposite, writing expressions in terms of statements, is not.

If we agree that "power" means "possibility", then languages like Lisp, which are based on expressions instead of statements, might be considered powerful.

⁴Wikipedia: Ternary operation (Javascript)

⁵jQuery Core Style Guide: Blocks

Homoiconicity

Homoiconicity is a property of a programming language's syntax, and means that programs in a language are expressed using that language's own data structures. The property of homoiconicity is key to the way Lisp empowers the programmer to fabricate and manipulate syntax.

Homoiconicity might be understood by exploring the properties of source code and interpretation of that source code in a non-homoiconic language first.

Javascript, while influenced by Lisp in other ways⁶, is not homoiconic, and its popularity, availability, and conventional syntax make it a candidate for exploration.

Implementing Javascript

How a Javascript Interpreter Works

Consider this fragment of Javascript code, which passes the sum of 1 and the product of 2 and 3 to a web browser's `alert` function:

```
alert(1 + 2 * 3);
```

When a browser passes this code, or data, as a string to its Javascript interpreter, something like the following happens:

- The interpreter *lexically analyzes*⁷ the string, turning it into a stream of *tokens*. Tokens are objects in source code upon which other constructs are defined, such as function names, operators, and literals like numbers or strings. The tokens in the above fragment might be:

```
- alert
- (
- 1
- +
- 2
- *
- 3
- )
- ;
```

⁶Popularity by Brendan Eich

⁷Wikipedia: Lexical analysis

- The interpreter *parses* the token stream, attempting, recursively, to match the pattern of tokens to known language constructs. For instance, when the interpreter sees the function name `alert` followed by `(`, it knows to expect zero or more tokens that must be succeeded by a closing `)`.
- The interpreter *evaluates* the constructs it found from the inside out:
 - `2 * 3` happens first, because, `*` has higher precedence than `+` and because the interpreter knows it needs a value or identifier to pass to `alert`.
 - The product, 6, is returned.
 - `1 + 6` is evaluated, returning 7.
 - Finally, `alert(7)` is evaluated, and a message is displayed.

If you were to write your own Javascript interpreter or compiler, you would need to implement a **lexical analyzer**, a **parser**, and an **evaluator** in order to complete the above steps.

Even if you were to implement your interpreter in Javascript, you'd have a long way to go, because none of these functions are provided by Javascript interpreters to Javascript programmers.

Javascript provides the `eval` function, but it accepts only strings, and you need to create source code strings yourself. Creating source code strings to pass to `eval` is effectively compilation, and done correctly could be as difficult as writing a compiler.

The task is not as daunting if we take the Lisp approach and build our programs in a data format richer than strings. Let's meet the problem half way, and invent a Javascript syntax based on Javascript data.

A Homoiconic Javascript Subset: JSONScript

S-expressions

Consider this alternative representation of the code fragment, in a language we just invented, JSONScript:

```
[ 'alert', ['+', 1, ['*', 2, 3]] ]
```

Unlike the earlier example, which was a string that required multiple phases of analysis to understand and run, the above code is written using data structures native to Javascript: string, number, and array.

JSON, which stands for “Javascript Object Notation”, is a notation for expressing data in terms of Javascript data structures. JSON is expressive enough for us to use it to represent code for a simple programming language.

We could easily write a function called `compile_jsonjs` that converted this representation into a string that could be passed to `eval`, because:

- We don't need to write a lexical analyzer or parser: our code is already rich enough to express how it should be evaluated.
- `compile_jsonjs` doesn't need to know about operator precedence because we're using *s-expressions*.

S-expressions, short for "symbolic expressions", are a convention for writing code that allows us to represent a function call with an array. This convention is also known as *prefix notation*. The first element of the array is the function, and subsequent elements are arguments to that function that are evaluated first.

S-expressions leave no ambiguity about the order in which functions must be applied. In the original example, the expression `1 + 2 * 3` evaluated to 7 because it's in the Javascript specification that `*` must evaluate before `+`.

In JSONScript, we express our intent to multiply first by making `['*', 2, 3]` an argument to `+`.

S-expressions don't have to be arrays. `42` and `"hamster dance"` are also s-expressions, because they also evaluate, but to themselves.

Lisp uses lists instead of arrays as s-expressions, but the idea is the same.

Some Implementation Details

There are at least two implementation details we face in implementing `compile_jsonjs`:

- In JSONScript, as in Lisp, there is no distinction between function and operator. Functions, whether they are `alert` or `*`, must only appear as a string and as the first element of an array.
 - `compile_jsonjs` needs to recognize functions that Javascript considers to be operators, and return the correct syntax for them.
- In Javascript, functions are identified by symbols, not strings. For instance, this syntax is not correct: `'alert'(7)`.
 - `compile_jsonjs` must return function names as symbols.
 - Conflating symbol and string is necessary but has unfortunate implications. For instance, it will be impossible to pass a function name as an argument to a function in JSONScript.
 - Adding the ability to pass function names as arguments equires us to go beyond JSON as our source data format, because JSON permits symbol literals only as an object field names.

Building the Compiler

We're now equipped to reason about how `compile_jsonjs` will work.

- First, we check if the input is an array.
- If it is an array, we:
 - Compile the arguments
 - Check if the first element is an operator like `*` or `+`
 - * If it is an operator, interpose the compiled arguments with the operator, and concatenate everything into a single string. `compile_jsonjs(['+', 1, 2])` should return `"1+2"`.
 - * If it's not an operator, like `alert`, we return a function call. `compile_jsonjs(['alert', 'hello'])` should return `"alert('hello')"`
- If it isn't an array, we return it as a string. `compile_jsonjs(42)` should return `"42"`.

The Implementation

Next, we can translate our plan into code. We know that `compile_jsonjs` will need helper functions for:

- Testing if an object is an array: This is how we'll determine at the beginning of `compile_jsonjs` whether we're compiling a function call or a *literal*. Literals are data that evaluate to themselves, like numbers or strings.
- Testing if an array contains a particular object: We'll need to make a list of functions that are Javascript operators, because we must emit different Javascript syntax for them.

The first helper function we need, `isArray()`, isn't available on some browsers, and is actually difficult to write in a widely compatible way.

So, we can borrow it from the `underscore.js`⁸ library:

```
var isArray = function (obj) {  
  return Object.prototype.toString.call(obj) === '[object Array]';  
}
```

Next, we need a function for testing if an object is in an array. Such a function is also not available in some browsers, but we can implement it like so:

⁸`_isArray` in `underscore.js`

```

var inArray = function(array, item) {
  for(var i = 0; i < array.length; i++)
    if (array[i] === item) return true;
  return false;
}

```

With these functions in place, we can finally write our compiler:

```

var compile_jsonjs = function(expression) {

  var operators = ['*', '+'];

  if (isArray(expression)) {

    var func_name = expression[0];
    var func_args = expression.slice(1);

    var compiled_args = [];
    for (var i = 0; i < func_args.length; i++)
      compiled_args.push(compile_jsonjs(func_args[i]));

    if(inArray(operators, func_name)) {
      return compiled_args.join(func_name);
    } else {
      return func_name + "(" + compiled_args.join(',') + ")";
    }

  } else {
    return expression.toString()
  }
}

```

The full source listing, which you can copy and paste into a web browser's Javascript console, follows:

```

var isArray = function (obj) {
  return Object.prototype.toString.call(obj) === '[object Array]';
}

var inArray = function(array, item) {
  for(var i = 0; i < array.length; i++)
    if (array[i] === item) return true;
  return false;
}

var compile_jsonjs = function(expression) {

  var operators = ['*', '+'];

  if (isArray(expression)) {

    var func_name = expression[0];
    var func_args = expression.slice(1);

    var compiled_args = [];
    for (var i = 0; i < func_args.length; i++)
      compiled_args.push(compile_jsonjs(func_args[i]));

    if(inArray(operators, func_name)) {
      return compiled_args.join(func_name);
    } else {
      return func_name + "(" + compiled_args.join(',') + ")";
    }
  } else {
    return expression.toString()
  }
}

```

Trying it Out

We can test our compiler by seeing what code it generates with our JSONScript fragment:

```
compile_jsonjs(['alert', ['+', 1, ['*', 2, 3]]]);
```

And we can evaluate our compiled code by passing that result to `eval`:

```
eval(compile_jsonjs(['alert', ['+', 1, ['*', 2, 3]]]));
```

Going Further

JSONScript probably isn't a language you would want to use for your next programming project, but hopefully you now have a better understanding of what Lisp is and why it is different.

Further extending JSONScript will help you understand Lisp even more.

You can begin by adding support for other Javascript operators.

Next, you might implement `lambda` and `quote`. These will allow you to define functions in JSONScript, and to pass arrays to functions without evaluating them.

Once you have `lambda` and `quote`, you can implement other *special forms*, using Paul Graham's *The Roots of Lisp* as a guide⁹.

If you're feeling particularly ambitious, you might consider adding support for one of Lisp's most famous and unique features - *macros*, or functions that:

- are run before evaluation
- do not evaluate their arguments
- return JSONScript code

⁹The Roots of Lisp by Paul Graham