
Crossing borders: The beauty of Lisp

The El Dorado of programming languages

Skill Level: Intermediate

[Bruce Tate \(bruce.tate@j2life.com\)](mailto:bruce.tate@j2life.com)

CTO

WellGood LLC

06 Feb 2007

Updated 07 Feb 2007

Lisp has long been recognized as one of the great programming languages. The fanatical following it has inspired throughout its long history -- nearly 50 years -- tells you it's something special. At MIT, Lisp plays a foundational role in the curriculum for all programmers. Entrepreneurs like Paul Graham used Lisp's incredible productivity as the jet fuel for successful startups. But to the chagrin of its followers, Lisp never made it into the mainstream. As a Java™ programmer, if you spend some time with Lisp -- this lost city of gold -- you'll discover many techniques that will change the way you code, for the better.

I recently finished my first marathon and found running much more rewarding than I ever could have expected. I turned an act as simple as taking a step into something extraordinary for the human body, running 26.2 miles. Some languages, like Smalltalk and Lisp, give me a similar feeling. For Smalltalk, the step is the object; everything in Smalltalk deals with objects and message passing. With Lisp, the foundational step is even simpler. This language is composed entirely of lists. But don't let the simplicity fool you. This 48-year-old language comes with incredible power and flexibility that the Java language can't begin to match.

My first interaction with Lisp, as an undergraduate, didn't go smoothly. I fought the language tooth and nail as I tried to cram it into the procedural paradigms that I knew, rather than work within Lisp's functional structure. Though Lisp is not strictly a functional language -- some of its features disqualify it from the strictest sense of the

term -- many of Lisp's idioms and features have a strong functional flair to them. Since then, I've learned to embrace lists and functional programming.

About this series

In the [Crossing borders series](#), author Bruce Tate advances the notion that today's Java programmers are well served by learning other approaches and languages. The programming landscape has changed since Java technology was the obvious best choice for all development projects. Other frameworks are shaping the way Java frameworks are built, and the concepts you learn from other languages can inform your Java programming. The Python (or Ruby, or Smalltalk, or ... fill in the blank) code you write can change the way that you approach Java coding.

This series introduces you to programming concepts and techniques that are radically different from, but also directly applicable to, Java development. In some cases, you'll need to integrate the technology to take advantage of it. In others, you'll be able to apply the concepts directly. The individual tool isn't as important as the idea that other languages and frameworks can influence developers, frameworks, and even fundamental approaches in the Java community.

This installment of *Crossing borders* breaks out this lost treasure. I take you on a simple stroll through Lisp's basic constructs and then ramp up quickly. You'll see Lambda expressions, recursion, and macros. This quick tour should give you an appreciation of Lisp's productivity and flexibility.

Getting started

For this article, I'm using GNU's GCL, which has a free download for many operating systems. But you should be able to use just about any version of Common Lisp, with only a little tinkering. See [Resources](#) for details on available Lisp versions.

As with most other languages, the best way to learn Lisp is to play. Open up your interpreter and code along with me. Lisp is primarily a compiled language, so you can easily explore it by simply typing commands.

A language of lists

Fundamentally, Lisp is a language of lists. Everything in Lisp, from data to the code making up your application, is a list. Each list is made up of *atoms* and lists. Numbers are atoms. Typing a number simply returns that number as a result:

Listing 1. Simple atoms

```
>1
1
>a
Error: The variable A is unbound.
```

If you type a letter, the interpreter complains, as in Listing 1. Letters are variables, so you must assign a value to them before you use them. If you want to refer to a letter or word rather than a variable, enclose it in quotes. Preceding the variable with a single quote tells Lisp to delay the evaluation of the list or atom that follows, as in Listing 2:

Listing 2. Delaying evaluation and quoting

```
>"a"
"a"
>'a
A
```

Notice that Lisp capitalizes the A. Lisp assumes that you want to use A as a symbol because you have not enclosed it in quotes. I'll discuss assignments, but I first need to cover lists to do so. A Lisp list is simply a sequence of atoms, enclosed in parentheses and separated by spaces. Try to type a list, such as the one in Listing 3. It doesn't work, unless you precede the list with a '.

Listing 3. Typing a simple list

```
>(1 2 3)
Error: 1 is invalid as a function.
>'(1 2 3)
(1 2 3)
```

Unless you precede a list with a ', Lisp evaluates each list as a function. The first atom is the operator, and the rest of the atoms in the list are the arguments. Lisp has numerous primitive functions, including many math functions -- for example, +, *, and `sqrt` -- as you might expect. `(+ 1 2 3)` returns 6, and `(* 1 2 3 4)` returns 24.

Two types of functions manipulate lists: *constructors* and *selectors*. Constructors build lists, and selectors break lists down. The core selectors are `first` and `rest`. The `first` selector returns the first atom of a list, and the `rest` selector returns the whole list except for the first atom. Listing 4 shows these selectors:

Listing 4. Basic Lisp functions

```
> (first '(lions tigers bears))
LIONS

> (rest '(lions tigers bears))
(TIGERS BEARS)
```

Both of the selectors take entire lists and return strategic pieces of the list. Later you'll see how recursion takes advantage of these selectors.

If you want to build lists instead of taking them apart, you need constructors. As in the Java language, constructors build new elements: objects for the Java language and lists for Lisp. `cons`, `list`, and `append` are examples of constructors. The core constructor, `cons`, takes two arguments: an atom and a list. `cons` adds the atom to the list as the first element. If you call `cons` on `nil`, Lisp treats `nil` as an empty list and builds a list of one element. `append` concatenates two lists. `list` contains a list of all of the arguments. Listing 5 shows the constructors in action:

Listing 5. Using constructors

```
> (cons 'lions '(tigers bears))
(LIONS TIGERS BEARS)

> (list 'lions 'tigers 'bears)
(LIONS TIGERS BEARS)

> (append '(lions) '(tigers bears))
(LIONS TIGERS BEARS)
```

`cons` can build any list when you use it in conjunction with `first` and `rest`. The `list` and `append` operators are just convenience attributes, but you'll use them frequently. In fact, you can build any list, or return any list fragment, using `cons`, `first`, and `rest`. For example, to get the second or third element of a list, you'd get the `first` of the `rest`, or the `first` of the `rest` of the `rest`, as in Listing 6. Or, to build a list of two or three elements, you can use `cons` in conjunction with `first` and `rest` to simulate `list` and `append`.

Listing 6. Building second, third, list, and append

```
>(first (rest '(1 2 3)))
2

>(first (rest (rest '(1 2 3))))
3

>(cons '1 (cons '2 nil))
(1 2)

>(cons '1 (cons '2 (cons '3 nil)))
(1 2 3)
```

```
>(cons (first '(1)) '(2 3))  
(1 2 3)
```

These examples might not get you excited, but the principle of building a clean and beautiful language on such simple primitives is intoxicating to some programmers. These simple list-building instructions form the foundation of recursions, higher-order functions, and even higher-order abstractions like closures and continuations. It's time to move up to higher abstractions.

Building functions

Lisp function declarations are -- you guessed it -- lists. Listing 7, which builds a function returning a list's second element, shows the form of function declarations:

Listing 7. Building the second function

```
(defun my_second (lst)  
  (first (rest lst))  
)
```

`defun` is the function that defines custom functions. The first argument is the function name, the second the argument list, and the third is the code you wish to execute. You can see that all Lisp code is expressed as lists. This flexibility and power let you manipulate your application as well as you can manipulate any other data. You'll later see some examples that blur the distinction between code and data.

Lisp also handles conditional constructs such as the `if` statement. The form is `(if condition_statement then_statement else_statement)`. Listing 8 shows a simple `my_max` function that computes the maximum of two input variables:

Listing 8. Computing the max of two integers

```
(defun my_max (x y)  
  (if (> x y) x y)  
)  
  
MY_MAX  
(my_max 2 5)  
  
5  
(my_max 6 1)  
  
6
```

Let me review what you've seen so far:

- Lisp uses lists and atoms to represent both data and programs.
- Evaluating a list treats the first element as a list function and the other elements as function arguments.
- Lisp conditional statements use true/false expressions in conjunction with code.

Recursion

Lisp provides coding structures for iteration, but recursion is a far more popular way to navigate lists. The combination of `first` and `rest` works well with recursion. The `total` function in Listing 9 shows how it works:

Listing 9. Using recursion to compute the total of a list

```
(defun total2 (lst)
  (labels ((total-rec (tot lst)
            (if lst (total-rec (+ tot (first lst)) (rest lst)) tot)))
    (total-rec 0 lst)))
```

The `total` function in Listing 9 takes a list as a single argument. The first `if` statement breaks the recursion if the list is empty, returning zero. If not, the function adds the first element to the sum of the rest of the list. You can now see why `first` and `rest` are built the way they are. `first` can peel off the first element of the list, and `rest` makes it easy to apply *tail recursion* -- the type of recursion in Listing 9 -- to the rest.

Recursion in the Java language is limited for performance reasons. Lisp offers a performance optimization called *tail recursion optimization*. The Lisp compiler or interpreter can translate certain forms of recursion to iteration, allowing a simpler, cleaner way to work with recursive data structures, such as trees.

Higher-order functions

Lisp gets more interesting when you blur the distinction between data and code. In the last two articles in this series, you saw higher-order functions in JavaScript and closures in Ruby. Both of these features pass functions as arguments. In Lisp, higher-order functions are trivial because functions are no different from any other kind of list.

Perhaps the most common use of higher-order functions is the *lambda expression*, which is Lisp's version of a closure. A lambda function is a function definition that you use to pass higher-order functions into Lisp functions. For example, the lambda expression in Listing 10 computes the sum of two integers:

Listing 10. Lambda expressions

```
>(setf total '(lambda (a b) (+ a b)))  
(LAMBDA (A B) (+ A B))  
  
>total  
(LAMBDA (A B) (+ A B))  
  
>(apply total '(101 102))  
203
```

If you've ever used closures or higher-order functions, you might have a better understanding of the code in Listing 10. The first line defines a lambda expression and binds it to the symbol `total`. The second line simply shows the lambda function bound to `total`. Finally, the last expression applies the lambda expression to the list containing `(101 102)`.

Higher-order functions offer a higher level of abstraction than object-oriented concepts. You can use them to express ideas more concisely and clearly. The holy grail of programming is to provide more power and flexibility with fewer lines of code, without sacrificing readability or performance. Higher-order functions do all of these things.

Share this...



Digg
this
story



Post
to
del.icio.us

Slashdot
it!

Conclusion

Lisp may be old in terms of years, and even syntax. But if you dig a little bit, you'll find an incredibly powerful language with higher abstractions that are as valid and productive today as when they were created 50 years ago. Many more-modern languages borrow from Lisp, and most still do not provide as much power. If Lisp had a fraction of the marketing behind Java or .NET and similar mindshare across universities, we might well all be writing it right now.

Resources

Learn

- [Beyond Java](#) (O'Reilly, 2005): The author's book about the Java language's rise and plateau and the technologies that could challenge the Java platform in some niches.
- [GNU Common Lisp](#): One of the more popular Lisp implementations, and the Lisp interpreter used for this article.
- [Carl de Marcken: Inside Orbitz](#): This discussion of Lisp features in production shows what Lisp can do in the real world.
- [Structure and Interpretation of Computer Programs](#), 2d ed. (Harold Abelson et al., McGraw-Hill, 1996): A timeless classic that builds on Lisp philosophies quickly.
- [Association of Lisp Users](#): An international organization supporting the Lisp community.
- [The Common Lisp Directory](#): This excellent site provides a wealth of Lisp and related resources, including [CLiki](#), which links to resources for free software implemented in Common Lisp and additional [ALU Lisp resources](#).
- [The Java technology zone](#): Hundreds of articles about every aspect of Java programming.

Get products and technologies

- [Common Lisp Implementations](#): Commercial and free Common Lisp implementations.

Discuss

- Check out [developerWorks blogs](#) and get involved in the [developerWorks community](#).

About the author

Bruce Tate

Bruce Tate is a father, mountain biker, and kayaker in Austin, Texas. He's the author of three best-selling Java books, including the Jolt winner *Better, Faster, Lighter Java*. He recently released *From Java to Ruby* and *Rails: Up and Running*. He spent 13 years at IBM and later formed the RapidRed consultancy, where he specialized in lightweight development strategies and architectures based on Ruby, and in the Ruby on Rails framework. He is now the CTO of WellGood LLC, a company that is forming a

marketplace for nonprofit organizations and charitable giving.