A Retrospective on *Paradigms of AI Programming*

If you can't hear me, it's because I'm in parentheses.
- Steven Wright

October 1997: It has been six years since I finished *Paradigms of AI Programming* (or *PAIP*), and now seems like a good time to look back at how Lisp and AI programming have changed.

Update: April 2002: It has now been over ten years, so I've updated this page.

Is Lisp Still Unique? Or at Least Different?

In 1991 Lisp offered a combination of features that could not be found in any other language. This combination made Lisp almost a necessity for certain kinds of work (in AI and other applications) and made Lisp the language of choice for a large community of users. Since then Lisp has maintained and added to features, but other languages have caught up. Let's look at whether Lisp is still unique, or at least different from other languages in 2002.

On [p. 25] of *PAIP*
I list eight important factors that make Lisp a good language for AI applications, and remarked that mostly these allow a programmer to delay making decisions. Let's look at how Lisp compares to, say, Java and Python in terms of these eight factors:

- **Built-in Support for Lists.**
  Java has the Vector type, which allows for sequences of dynamically varying length. Lists are better suited for functional programming (but have the disadvantage of linear access time), while Vectors are better suited for Object-Oriented programming (but have the disadvantage of linear time for non-destructive push). Support for Vectors in Java is not nearly as complete as support for lists and sequences in Lisp, mostly because Lisp endorses first-class functions as arguments to functions. Java may soon have a Collection class which cleans up some of the faults with enumerators, vectors and hashtables. Python too has a vector class (which it calls `list`, and Python's support is similar to Lisp's.
- **Automatic Storage Management.**
  Java and Python support this. Lisp implementations tend to be more mature and perform better.
- **Dynamic Typing.**
  Java attaches run-time type information to instances of the class Object, but not to primitive data elements. However, Java requires a type declaration for every variable. This has some advantages for production code, but has disadvantages for rapid prototyping and evolution of programs. Java does not have a generic/template system that would allow types life `Vector<String>`, and suffers greatly because of it. Python's object model is the same as Lisp's, but Python does not allow optional type declarations as Lisp does.
- **First-Class Functions.**
  Java has anonymous classes, which serve some of the purposes of closures, although in a less versatile way with a more clumsy syntax. In Lisp, we can say `(lambda (x) (f (g x)))` where in Java we would have to say

  ```
  new UnaryFunction() {
    public Object execute(Object x) {
      return (Cast x).g().f();
    }
  }
  ```

  where `Cast` is the real type of `x`. This would only work with classes that observe the `UnaryFunction` interface (which comes from JGL and is not built-in to Java). Python, as of version 2.1, supports first-class functions almost as well as Lisp: you would write `lambda x: f(g(x))`. The only drawback is that closed-over variables are read-only.
- **Uniform Syntax.**
  Java syntax is of medium complexity; certainly more complex than Lisp. Java does not encourage (or even

permit) macros. The JDK compilers are also rather hostile to computer-generated code; things which should be warnings (like unreachable code) are errors. This is somewhat helpful for human-written code, but is just downright annoying for computer-generated code. Some features of the language (like the rules for when you can have a variable of the same name declared) also make it hard to generate code. Python syntax is somewhat simpler than Java, and the presence of `eval` makes it feasible to generate code at runtime, but the lack of a macro system makes it more tedious than in Lisp.

- **Interactive Environment.**
  Some Java environments allow Lisp-like features such as an intgeractive command loop and stop-and-fix debugging. Lisp environments are still ahead, but that probably won't last long. BlueJ in particular has most of the major features of a good Lisp environment: you can recompile a method into a running program, and you can type in an expression and have it evaluated immediately. It is intended for teaching purposes, and I can't tell if it is suitable for production use. Python has the same interactive approach as Lisp, but the environment is less mature than Lisp's.
- **Extensibility.**
  This may prove to be Java's weak point. Java works well as long as you are willing to make everything a class. If something new like, say, aspect-oriented programming takes off, Lisp would be able to incorporate it with macros, but Java would not.
- **History.** Lisp has over 45 years of history; Java has 7, Python 12.

One more important feature that didn't make this list is **efficiency**. Lisp is about 1.5 to 4 times faster than Java, and about 10 to 50 times faster than Python. Lisp is probably within 20% to 60% of C/C++ in efficiency on most tasks, which is close enough that the differences depend more on the programmers involved than the language, and close enough that for most applications that speed is not an issue for Lisp. (You will need to code certain things, particularly involving memory management, using the techniques in the book such as memory resources. This is more work for the programmer, but is o different from what C++ offers with STL allocators.) Python is a different story: there is a large class of problems for which Python is too slow. Its hard to get benchmark data that is relevent to *your* set of applications, but here's some data:

Test	Lisp	Java	Python	Perl	C++	
exception handling	0.01	0.90	1.54	1.73	1.00	
hash access	1.06	3.23	4.01	1.85	1.00	
sum numbers from file	7.54	2.63	8.34	2.49	1.00	100+ x C++
reverse lines	1.61	1.22	1.38	1.25	1.00	50-100 x C++
matrix multiplication	3.30	8.90	278.00	226.00	1.00	10-50 x C++
heapsort	1.67	7.00	84.42	75.67	1.00	5-10 x C++
array access	1.75	6.83	141.08	127.25	1.00	1-5 x C++
list processing	0.93	20.47	20.33	11.27	1.00	0-1 x C++
object instantiation	1.32	2.39	49.11	89.21	1.00	
word count	0.73	4.61	2.57	1.64	1.00	
25% to 75%	0.93 to 1.67	2.63 to 7.00	2.57 to 84.42	1.73 to 89.21	1.00 to 1.00	

Relative speeds of 5 languages on 10 benchmarks from The Great Computer Language Shootout. Speeds are normalized so the g++ compiler for C++ is 1.00, so 2.00 means twice as slow; 0.01 means 100 times faster. Background colors are coded according to legend on right. The last line estimates the 25% to 75% quartiles by throwing out the bottom two and top two scores for each language.

Overall, Lisp does very well on the nine features. Anyone making an objective choice based on these features would continue to find Lisp the best choice for a wide range of applications. But there are now viable contenders that did not exist in 1991.

Is Lisp Unpopular Now?

One criticism of Lisp is that it is not as popular as other languages, and thus has a smaller community of users and less development activity around it.

The majority of the industry probably feels that Lisp has become less relevant, and that Java is taking over as the language of choice. My perception in 1997 was that Lisp had held on to most of its faithful users and added some new ones, while indeed much of the rest of the world had rushed first to C++, and then to Java (which is catching up fast, but C++ still holds a big lead). Thus, Lisp was in a stable absolute position, but a weaker relative position because of the increased support for the "majority" languages, C++ and Java, and less acceptance of a wide range of languages. The major Lisp vendors, Harlequin (for whom I worked for two years), and Franz, were at that time reporting steady increasing sales. I wasn't so sure about Digitool; they make a fine product, but it is Mac only, and the Mac is rapidly losing market share. Maybe they will come back stronger on the heels of OS X. Lisp was still being promoted, in a low-key way, by publishers and distributers like Amazon. Lisp continues to enjoy pockets of commercial success. Paul Graham recently sold a Lisp program (and the company around it) to Yahoo for $40 million. (It was an authoring tool for online stores.) Orbitz, one of the leading travel e-commerce sites, does schedule optimization using a Lisp system supplied by Carl de Marcken's company, ITA software. The CL-HTTP web server is in wide use. Some of the best work in bioinformatics is done in Lisp. For more examples, see:

- The Franz web site
- The Cliki index lists Common Lisp resources, but they are not all portable across CL implementations
- The Linux Weekly News
- The Open Directory entry for Lisp
- The comp.lang.lisp newsgroup

In 2002, the grand hopes for Java have not panned out. Java enjoys great popularity in the Enterprise space, but has not taken over as a general purpose rapid development language, nor as an efficient systems language. Performance of Java remains dismal when compared to C++ or to Lisp. I'm not sure why that is; Sun certainly has had enough time and resources to implement a better system. Maybe they should have hired more ex-Lisp-compiler-writers.

Fred Brooks is reported to have said ``More users find more bugs.'' I don't think that's a problem for Lisp. Lisp has had enough user-years that it is more stable than the other languages we will discuss here.

But the situation for Lisp in terms of popularity still reveals a weakness: the language standard has stagnated, without addressing some key issues like threading, sockets, and others. Furthermore, there is no well-known standard repository of libraries for new protocols like HTTP, HTML, XML, SOAP, etc. Individual implementations add libraries for these, and individual programmers create open-source implementations, but you don't get them right out of the box like you do with Java or Python, nor can you find them at a single location, like Perl's CPAN. This means that it takes more work to hunt these libraries down, and some programmers dismiss Lisp for a project because they don't immediately find the libraries they need.

C++ finally has the Standard Template Library, which remains much harder to use than Lisp's sequence functions, but is capable of great performance in the hands of wizard STL programmers (and of great headaches in the hands of average programmers). Lisp experts are still as productive as ever, but newer programmers are less likely to pick up Lisp. Consider the following measures of language popularity. In 1997 I looked at 10 languages; in 2002 I narrowed the field to 5:

1997

Language	Books	Usenet Articles	Recent Articles	URLs	Jobs	Avg. Rank
C++	479	166,686	13,526	3,329	1,006	1.2
Java	161	160,276	30,663	2,450	280	1.8
All Lisps	140	31,501	3,833	1,565	12	4.1
Perl	67	12,376	9,919	1,031	174	4.6

2002

Language	Books	Usenet Articles	URLs	Jobs	Avg. Rank
Java	1695	68,000	1,790,000	1109	1.75
C++	1208	25,800	1,170,800	517	2.00
Perl	424	57,800	978,000	364	2.50
Python	132	6,810	354,000	33	4.00

Smalltalk	35	36,176	2,187	537	31	5.2
Lisp	127	15,367	1,775	734	11	6.1
Prolog	150	7,101	518	846	2	6.6
Eiffel	10	16,367	2,080	214	0	7.5
Scheme	10	11,458	1,846	408	0	8.3
Dylan	3	4,676	212	21	1	9.6

For 10 computer languages, this chart summarizes the number of books offered at Amazon, the number of archived and recent news articles in `comp.lang.*` at Dejanews, the number of hits on the query `"+Language +computer"` at Infoseek, and the number of jobs offered at Westech. (The `"+computer"` is used because many of the language names are ambiguous (Eiffel Tower, Java drink). "All Lisps" means Lisp+Scheme+Dylan.)

Lisp is clearly behind C++ and Java, but at least in the ballpark for all measures except jobs. Lisp is slightly ahead of Perl, beating it in 3 of 5 measures.

Lisp	159	6,330	305,000	4	4.75

For 5 computer languages, this chart summarizes the number of books offered at Amazon, the number of news articles in `comp.lang.*` at Google Groups, the number of hits on the query `"Language computer"` at Google, and the number of jobs offered at Monster.com. (Congratulations to Amazon for being the only one to remain the definitive category-leader for five years.)

Java has now moved ahead of C++ in popularity, Perl has moved up to where it is nearly competitive with the big two, and Lisp has moved down, now beaing an order of magnitude behind the others in most categories. Python edges out Lisp in 3 of 4 categories.

These measurements are pretty unscientific and may not mean much. After I did them, I became aware of another study done by Tiobe Software, that has Java number 1, C and C++ at 2, 3, and Lisp at 17, with a market share 1/30th of Java. Popular does not mean better, and these numbers may not correlate perfectly with popularity, but they do say something.

Is Lisp at Least Popular for AI?

In 1991 and 1997, the answer was clearly yes. But in 2002, the answer is less clear. Consider the number of results for the following Google searches:

Search	Number of Results	Search	Number of Results
Java AI	401,000	java "artificial intelligence"	136,000
C++ AI	232,000	C++ "artificial intelligence"	104,000
Lisp AI	136,000	lisp "artificial intelligence"	81,000
Perl AI	193,000	Perl "artificial intelligence"	54,000
Python AI	84,000	Python "artificial intelligence"	40,000

It has been obvious to most observers that the machine learning community in particular has moved away from Lisp towards C++ or to specialized mathematical packages like Matlab. But I'm actually surprised that the trend has gone this far, and I'm very surprised to see Java on top, because I just haven't seen that much top-notch Java AI code. I'm also amazed that "Perl AI" surpasses "Lisp AI", but I suspect that there's some other meaning of "AI" in the Perl world, because "Perl Artificial Intelligence" is well behind "Lisp Artificial Intelligence".

New Lisp Books

The best, in my opinion, are Paul Graham's *On Lisp* and *ANSI Common Lisp*. Probably the best book ever on how to write Lisp compilers and interpreters is Christian Queinnec's *Lisp in Small Pieces*. In the Scheme world, Abelson and Sussman have a new edition of *Structure and Interpretation of Computer Programs*, and Daniel Friedman has a new

version of *The Little Lisper* called *The Seasoned Schemer*. Stephen Slade has a new book which I have not had a chance to read yet.

What Lessons are in *PAIP*?

Here is my list of the 52 most important lessons in *PAIP:*

1. Use anonymous functions. [p. 20]
2. Create new functions (closures) at run time. [p. 22]
3. Use the most natural notation available to solve a problem. [p. 42]
4. Use the same data for several programs. [p. 43]
5. Be specific. Use abstractions. Be concise. Use the provided tools. Don't be obscure. Be consistent. [p. 49]
6. Use macros (if really necessary). [p. 66]
7. There are 20 or 30 major data types; familiarize yourself with them. [p. 81]
8. Whenever you develop a complex data structure, develop a corresponding consistency checker. [p. 90]
9. To solve a problem, describe it, specify it in algorithmic terms, implement it, test it, debug and analyze it. Expect this to be an iterative process. [p. 110]
10. AI programming is largely exploratory programming; the aim is often to discover more about the problem area. [p. 119]
11. A general problem solver should be able to solve different problems. [p. 132]
12. We must resist the temptation to belive that all thinking follows the computational model. [p. 147]
13. The main object of this book is to cause the reader to say to him or herself "I could have written that". [p. 152]
14. If we left out the prompt, we could write a complete Lisp interpreter using just four symbols. Consider what we would have to do to write a Lisp (or Pascal, or Java) interpreter in Pascal (or Java). [p. 176]
15. Design patterns can be used informally, or can be abstracted into a formal function, macro, or data type (often involving higher-order functions). [p. 177]
16. Use data-driven programming, where pattern/action pairs are stored in a table. [p. 182]
17. Sometimes "more is less": its easier to produce more output than just the right output. [p. 231]
18. Lisp is not inherently less efficient than other high-level languages - *Richard Fateman*. [p. 265]
19. First develop a working program. Second, instrument it. Third, replace the slow parts. [p. 265]
20. The expert Lisp programmer eventually develops a good "efficiency model". [p. 268]
21. There are four general techniques for speeding up an algorithm: caching, compiling, delaying computation, and indexing. [p. 269]
22. We can write a compiler as a set of macros. [p. 277]
23. Compilation and memoization can yield 100-fold speed-ups. [p. 307]
24. Low-level efficiency concerns can yield 40-fold speed-ups. [p. 315]
25. For efficiency, use declarations, avoid generic functions, avoid complex argument lists, avoid unnecessary consing, use the right data structure. [p. 316]
26. A language that doesn't affect the way you think about programming is not worth knowing - *Alan Perlis*. [p. 348]
27. Prolog relies on three important ideas: a uniform data base, logic variables, and automatic backtracking. [p. 349]
28. Prolog is similar to Lisp on the main points. [p. 381]
29. Object orientation = Objects + Classes + Inheritance - *Peter Wegner* [p. 435]
30. Instead of prohibiting global state (as functional programming does), object-oriented programming breaks up the unruly mass of global state and encapsulates it into small, manageable pieces, or objects. [p. 435]
31. Depending on your definition, CLOS is or is not object-oriented. It doesn't support encapsulation. [p. 454]
32. Prolog may not provide exactly the logic you want [p. 465], nor the efficiency you want [p. 472]. Other representation schemes are possible.
33. Rule-based translation is a powerful idea, however sometimes you need more efficiency, and need to give up the simplicity of a rule-based system [p. 509].
34. Translating inputs to a canonical form is often a good strategy [p. 510].
35. An "Expert System" goes beyond a simple logic programming system: it provides reasoning with uncertainty, explanations, and flexible flow of control [p. 531].
36. Certainty factors provide a simple way of dealing with uncertainty, but there is general agreement that probabilities provide a more solid foundation [p. 534].
37. The strategy you use to search for a sequence of good moves can be important [p. 615].
38. You can compare two different strategies for a task by running repeated trials of the two [p. 626].

39. It pays to precycle [p. 633].
40. Memoization can turn an inefficient program into an efficient one [p. 662].
41. It is often easier to deal with preferences among competing interpretations of inputs, rather than trying to strictly rule one interpretation in or out [p 670].
42. Logic programs have a simple way to express grammars [p. 685].
43. Handling quantifiers in natural languiage can be tricky [p. 696].
44. Handling long-distance dependencies in natural language can be tricky [p. 702].
45. Understanding how a Scheme interpreter works can give you a better appreciation of how Lisp works, and thus make you a better programmer [p. 753].
46. The truly amazing, wonderful thing about `call/cc` is the ability to return to a continuation point more than once. [p. 771].
47. The first Lisp interpreter was a result of a programmer ignoring his boss's advice. [p. 777].
48. Abelson and Sussman (1985) is probably the best introduction to computer science ever written [p. 777].
49. The simplest compiler need not be much more complex than an interpreter [p. 784].
50. An extraordinary feature of ANSI Common Lisp is the facility for handling errors [p. 837].
51. If you can understand how to write and when to use `once-only`, then you truly understand macros [p. 853].
52. A word to the wise: don't get carried away with macros [p. 855].

What did PAIP Accomplish?

- As an advanced Lisp text, *PAIP*
  stands up very well. There are still very few other places to get a thorough treatment of efficiency issues, Lisp design issues, and uses of macros and compilers. (For macros, Paul Graham's books have done an especially excellent job.)

- As an AI programming text, *PAIP*
  does well. The only real competing text to emerge recently is Forbus and de Kleer, and they have a more limited (and thus more focused and integrated) approach, concentrating on inference systems. (The Charniak, Riesbeck, and McDermott book is also still worth looking at.) One change over the last six years is that AI programming has begun to look more like "regular" programming, because (a) AI programs, like "regular" programs, are increasingly concerned with large data bases, and (b) "regular" programmers have begun to address things such as searching the internet and recognizing handwriting and speech. An AI programming text today would have to cover data base interfaces, http and other network protocols, threading, graphical interfaces, and other issues.

- As an AI text, *PAIP*
  does not fare as well. It never attempted to be a comprehensive AI text, stressing the "Paradigms" or "Classics" of the field rather than the most current programs and theories. Happily, the classics are beginning to look obsolete now (the field would be in sorry shape if that didn't happen eventually). For a more modern approach to AI, forget *PAIP* and look at Artificial Intelligence: A Modern Approach.

Peter Norvig