Parenthetically Speaking with Kent M. Pitman

Accelerating Hindsight

Lisp as a Vehicle for Rapid Prototyping

From time to time, I meet people who really like Lisp but who find themselves at a loss for words when trying to articulate to others the various reasons why. For example, many people agree that good support for rapid prototyping is a key strength of Lisp. But what *is* rapid prototyping and how exactly is it that Lisp supports it better than other languages?

In this article, I will survey the various reasons that I think Lisp—particularly Common Lisp—is good for rapid prototyping. I hope to show that there is a sound reason for the warm feeling many show toward Lisp, even when the words of explanation don't flow readily off the tongue. I also hope this is information you can use to strengthen the case for new or continued use of Lisp at your workplace.

Although the focus in this paper is Common Lisp, many of the issues raised are general to the entire Lisp family of languages.

Why Rapid Prototyping?

The use of rapid prototyping enables early review and earlier discovery of problems. The sooner you get feedback, the more time you have to fix things and the higher the quality of the final product.

Hindsight is 20-20

"As soon as I finished the project, I knew what I should have done instead."

This familiar refrain is the genesis of the desire to build prototypes. If there's one thing that's certain, it's that hindsight is always better than foresight. This truth shines through as strongly in the domain of programming as in any other arena.

It's often easy to look back at a finished project and realize why it shouldn't have been built, or how it should have been built differently, or why it wasn't the right thing to build first. But by the time you finish that project, your budget may be so exhausted that you can't afford to just throw away what you've done and start over. You need to reach a point early enough and cheaply enough that you can take a look back, reflect and decide to make changes while you can still afford to.

Having a prototype lets you do just that—look back and see the consequences of your design, enhancing the things you like and discarding the things you don't. The reason for wanting to build a prototype rapidly is so there's still time to *do* something with all of your newfound wisdom.

Products vs. Prototypes

"I want products, not prototypes." This is one of the two most common reasons I've heard for eschewing the construction of prototypes. The apparent assumption is that the temptation will be too great and the prototype will end up being sold as a poor-quality substitute for a real product. It is as if the speaker is saying, "Don't make a poor quality product—make a good quality product instead." If only it *were* so simple.

In fact, it probably trades one problem for another. It's true—you can definitely run into trouble by building a prototype and then selling it. But in that scenario, the problem was not the decision to *build* the prototype; it was the decision to *sell* it. In fact, you're at pretty much the same risk if you go straight to building a "product" as you would be if you were building a "prototype" for sale. It doesn't matter what you call it—a first effort is still a first effort.

Naming issues aside, the question is not *whether* you will build a prototype. The question is only whether you'll be forced by time constraints to call your first effort a product and to sell it.

Putting a Value on Early Review

"I don't have time and resources for prototypes." This is the other common objection I've heard to the idea of building prototypes. I always feel compelled to respond, "Do you have the time and resources to cope with the consequences of *not* prototyping?"

The traditional "waterfall model" of software development, with its nearly unidirectional flow from requirements to development to testing to delivery, may mean you only notice critical problems just as the product is on its way out the door. This, in turn, can lead to hasty patches or none at all because of fixed time constraints. Having an early, working prototype can help you spot many embarrassing design errors or unexpected needs in parallel with development, and early enough that corrective action can be taken in time to make a difference in product quality.

If you are not selective, the buying public will be. Darwin's theory of evolution applies equally well to the marketplace. Bad design will be weeded out. The more variations you can afford to experiment with internally before your product sees the light of day, the better sense of confidence you'll have that the design you place before the market will stand up to public scrutiny.

Lisp's Support for Rapid Prototyping

"Aren't all languages equivalent in power? What makes Lisp better than any other language?"

The notion of Turing equivalence does not take into account issues like speed of development, or even execution speed for that matter. When comparing languages, it's not much help. There are real differences between languages in terms of what they make easy or hard, concise or not, and so on.

Some features of Lisp that strongly facilitate rapid prototyping are:

- It is comprehensive.
- It supports modular design.
- It supports interactive development, debugging, *and* update.

These features are unified by a single common thread: elimination of distraction. Hoops through which a language makes you jump that do not contribute directly toward your near-term goal are a distraction. Lisp minimizes this distraction both by directly supporting actions that are needed to achieve your goal, and by helping you to avoid the need to take actions that are not.

Lisp is Comprehensive

A programming language establishes a base from which one begins work on a project. Anything a language doesn't provide is left as an exercise to the programmer. But if the programmer wants to get straight to the business of prototyping, such exercises are a distraction.

A bare Common Lisp comes with an extraordinarily powerful set of pre-defined tools that allow the programmer to hit the ground running and move immediately to the task of defining application-related behavior. A detailed review of these would go on for pages. To conserve space in this article, I'll cite some of the basic datatypes and then rest my case. I think it's plain that Lisp *is* comprehensive.

- *Numbers.* Common Lisp provides not just machine integers and floats, but also complexes and arbitrary precision integers and rationals.
- *Characters.* Common Lisp's decision to make characters distinct from integers avoids confusion between characters and codes, and permits dynamic type dispatch on characters.
- *Arrays.* The Common Lisp array facility is powerful and flexible. Arrays can be specialized for a wide variety of element types, such as bits and characters. Arrays can also be created whose size can be easily adjusted dynamically through the use of fill pointers and displacement.
- *Conses.* A signature type for Lisp, conses (sometimes called pairs) can be composed into a variety of types for representing ordered sets and relationships, such as lists and trees.
- *Symbols, Packages, and Readtables.* These datatypes are used as the basis of Common Lisp's processing of programs, but are also available to programmers for use in applications.
- *Functions.* The ability to use named and anonymous functions as data objects facilitates the use of higher order functions, a powerful tool for procedural abstraction.
- *Hash Tables.* These widely accepted data structures for fast access to large tables of data are not present in most languages, but are standard in Common Lisp.
- *Pathnames.* Common Lisp pathnames are structured filenames that permit defaulting and merging to occur by slot-filling rather than by painstaking and error-prone string searching and concatenation.
- *Streams.* Opening and processing streams to files is easy and flexible. Streams to character data can be processed at the character, line, or expression-oriented level. Binary data can be manipulated by byte or by block.
- *Classes, Methods, and Generic Functions.* The Common Lisp Object System is known and respected throughout the Object-Oriented Programming Community for its power and dynamic flexibility. It provides a unified framework that both explains initial, system-provided functionality and enables program-defined extensions and customizations.
- *Conditions and Restarts.* The Common Lisp condition system sets the standard for condition handling in other languages. Through the use of simple, abstract, object-oriented data and operations, Common Lisp has shown that flexible error handling is a high-level operation that any programmer can turn to advantage, not just the realm of system programmers.

Lisp Supports Modular Design

Many modern languages make a claim to certain kinds of modularity but there is wide variance in what this means. For some languages, modularity merely means independent design and development. Lisp, too, supports these things—by providing for opaque interfaces (where requested) between programs and other programs, or between programs and data. But Lisp goes much farther than many languages:

Using complex literal constants in code

In production code, you might make a global variable to keep track of a quantity shared by several modules. For example:

```
(DEFVAR *CLUB-MEMBERS* '())
(DEFUN ADD-CLUB-MEMBER (WHO) (PUSHNEW WHO *CLUB-MEMBERS*))
(DEFUN IS-CLUB-MEMBER? (WHO) (MEMBER WHO *CLUB-MEMBERS*))
```

However, for prototyping it may not matter that the club can be extended and you may prefer not to spend much time on this part of the program—doing only the minimal amount of work needed to make the other parts of the program run. In that case, you might instead write the following:

```
(DEFUN IS-CLUB-MEMBER? (WHO) (MEMBER WHO '(JOE SALLY FRED)))
```

The ability to use complicated literal data in programs is useful in functional situations as well. Shown below are two equivalent ways to call some function F with a functional parameter. Only the second, by using an "anonymous" function object, keeps the programmer from the distracting task of wondering whether the functional parameter will be used more than once, what would be a good name for it, *etc.*

```
[a] (DEFUN PATHNAME-LESSP (X Y) (STRING-LESSP (NAMESTRING X) (NAMESTRING Y)))
    (F #'STRING-OR-NUMBER-LESSP)

[b] (F #'(LAMBDA (X Y) (STRING-LESSP (NAMESTRING X) (NAMESTRING Y))))
```

Condition System

The presence of a condition system means that modular customization can be done of new situations without disturbing the modularity of an existing program. Consider the following program, which tries to open a file and search it, but which signals an error if the file does not exist:

```
(DEFUN SEARCH-LINES-OF-FILE (STRING FILE)
  (WITH-OPEN-FILE (STREAM FILE :DIRECTION :INPUT)
    (LOOP FOR LINE = (READ-LINE STREAM)
          FOR I FROM 0
          WHEN (SEARCH STRING LINE)
            DO (FORMAT T "~&Line ~D: ~A" I LINE)
          FINALLY (FORMAT T "~&Done.~%"))))
```

Such a program may be extended in a modular fashion by another programmer, even without access to the original source. The second programmer, upon observing that a file error has occurred, might write:

```
(DEFUN SEARCH-FILE (STRING FILE)
  (HANDLER-CASE (SEARCH-LINES-OF-FILE STRING FILE)
    (FILE-ERROR (CONDITION)
      (FORMAT T "~&File problem: ~A~%" CONDITION))))
```

The second programmer's ability to use a modular solution is often accidental, not something the first programmer planned. But it is not accidental at the language design level. Common Lisp's condition system is specifically designed to encourage modularity which promotes such fortunate "accidents." This is very important in prototyping, since often tracking down the original programmer and convincing him to return an appropriate error value can be anything from time consuming to impossible.

Macros

Macros can be a real boon to rapid prototyping. Few things in code are as tedious as the needless repetition of a clumsy idiom throughout a large body of code. It is as painful to type in originally as it is distracting to later read. It is also error-prone. Consider the following code fragment, which might be part of a state machine that optionally keeps a history of its prior states for debugging purposes:

```
(IF TRACING (PUSH (CONS STATE (COPY-LIST REGISTERS)) STATE-HISTORY))
(GO STATE-17)
```

Writing such code over and over bloats the source text needlessly, and replicates dependence on the representation of the state and the registers making it hard to experiment with alternatives. A better approach would be to write a macro definition such as:

```
(DEFMACRO NEW-STATE (TAG)
  `(PROGN (IF TRACING (PUSH (CONS STATE (COPY-LIST REGISTERS)) STATE-HISTORY))
          (GO ,TAG)))
```

Given this, one could do a state transition to STATE-17 by merely writing:

```
(NEW-STATE STATE-17)
```

Also, because Lisp macros use structured objects and not textual substitution, macros like this are more reliable than macros in most other languages. This permits more complex uses of macros *without* diminished robustness that heavily cascaded textual macros might produce.

Optional Declarations

Efficient code in any language often requires declarations about type information that would not otherwise be possible for the compiler to infer. However, in most languages, this results in a need to *always* make type declarations, presumably on the assumption that efficient programs are always desired. Often when prototyping, efficiency is not the paramount concern. A definition like:

```
(DEFUN TWICE (X) (* 2 X))
```

might suffice for prototyping. This is a perfectly valid program that will use very generic arithmetic that expects arbitrary numeric types and that is prepared to actively signal an error if a non-numeric type is received. In some situations, even production code will want this level of generality. However, in most cases, additional type constraints exist and the programmer could instead write:

```
(DECLAIM (INLINE TWICE))
(DEFUN TWICE (X)
  (DECLARE (FIXNUM X))
  (THE FIXNUM (* 2 X)))
```

The `INLINE` declaration declares that the definition may be directly expanded into places where the definition is used (unless inhibited by a local `NOTINLINE` declaration at the point of use). This declaration, while not about type, is also optional.

Other, more general declarations allow users to specify varying degrees of care about common trade-offs in compilation, such as size versus space. Consider:

```
(DEFUN TWICE-ALL (LIST)
  (MAPCAR #'TWICE LIST))
```

The `MAPCAR` operator performs iterative application across a list. At a cost in speed, the program writer might prefer code compactness, as in:

```
(DEFUN TWICE-ALL (LIST)
  (DECLARE (OPTIMIZE (SPACE 3) (SPEED 2)))
  (MAPCAR #'TWICE LIST))
```

By reversing the numeric values associated with these optimization qualities, it is possible to say that speed is more important than space, as in:

```
  ... (DECLARE (OPTIMIZE (SPACE 2) (SPEED 3))) ...
```

Decades of experience with Lisp have shown that it is best to *first* develop a working program that you are happy with and *then* add declarations to optimize it. The flexibility to use this ordering is absolutely essential to rapid prototyping. To require the initial insertion of type declarations is to work *against* rapid prototyping by increasing the likelihood that you will spend lots of time optimizing programs that you are later going to throw away.

Also, you may not know in the beginning which parts require optimization, or what the specific nature of your data will be. Well-designed languages permit you to make such decisions as information becomes available, rather than forcing you to make premature decisions on an arbitrary and perhaps unnecessarily limiting basis.

Higher Order Functions

Another feature of Lisp which supports delayed decision-making is procedural abstraction—the ability to pass functions as arguments to other functions. Through this technique, a certain decision may be delayed until a later time in development. For example, a simple but general-purpose data browsing tool might look like:

```
(DEFUN BROWSE-LOOP (ITEMS DISPLAYER COMMAND-READER PROCESSOR)
  (LOOP (DOLIST (ITEM ITEMS) (FUNCALL DISPLAYER ITEM))
        (SETQ ITEMS (FUNCALL PROCESSOR (FUNCALL COMMAND-READER) ITEMS))))
```

This program acknowledges a general theory of browsing; that is, that a set of items is presented, then an opportunity is offered to operate on that set, and then the process begins anew (perhaps with the same items, perhaps with some new set of items). This can be done in advance of knowing the details of how items will be displayed or how commands will be read and processed.

The flexibility provided by this paradigm means that programs don't have to change as often because assumptions are not built into every part of the code. Or if they do have to change, they might do so in localized places; for example, only in BROWSE-LOOP and not its callers, or vice versa.

Lisp Supports Incremental Development, Debugging, *and* Update

In this final section, we'll talk about the highly dynamic aspects of Lisp which support incremental development.

Read-Eval-Print Loop

Nearly all Common Lisp implementations come with a command interpreter which allows one to interactively type Lisp expressions to be evaluated. The presence of an interactive command interpreter, while not unique to Lisp (Basic, APL, Teco, and Hypertalk are other examples of languages that offer it), is nevertheless a factor contributing to Lisp's power for rapid prototyping. In particular, C and C++ environments generally do not offer equivalent functionality.

Complementing Lisp's Read-Eval-Print loop is the ability to modify the reader (parser) and the printer (which displays data resulting from evaluation) to permit the textual inclusion of new datatypes as literal constants in code. The following example shows how easily the language and environment can be extended to integrate a new facility. This example assumes that the functions PARSE-HOST, MAKE-HOST, HOST-EQUAL, and HOST-NAME-COMPONENTS are already defined, but that the syntax `#"hostname"` is not. It modifies the reader and the printer to know about such a syntax and shows examples of the usage.

```
(SETQ H1 (MAKE-HOST :NAME-COMPONENTS '("HARLEQUIN" "COM")))
   ⇒    #S(HOST :NAME-COMPOMENTS ("HARLEQUIN" "COM"))
(DEFUN READ-HOST (STREAM SUBCHAR ARG)
  (UNREAD-CHAR SUBCHAR STREAM)
  (PARSE-HOST (READ STREAM T T T)))
(SET-DISPATCH-MACRO-CHARACTER #\# #\" #'READ-HOST)
(DEFMETHOD PRINT-OBJECT ((H HOST) STREAM)
  (FORMAT STREAM "#\"~{~A~^.~}\"" (HOST-NAME-COMPONENTS H)))
H1 ⇒    #"HARLEQUIN.COM"
(SETQ H2 (MAKE-HOST :NAME-COMPONENTS '("HARLEQUIN" "COM")))
   ⇒    #"HARLEQUIN.COM"
(HOST-EQUAL H2 #"HARLEQUIN.COM")
   ⇒    true
```

Debugging Language is Programming Language

Some programming languages are designed in such a way that you write a program and then the compiler processes it and then the program is no more. The idea of typing more expressions in the source language would not make sense because the meaning of the words in the source language are inferred statically from the surrounding program, and any further statements in that language would be meaningless because it would fly in the face of the "closed world" assumptions that were in effect at compilation time.

So in C, when programs crash, one doesn't use C to debug the crash. One uses a debugger which tries to crudely approximate certain information relating to the source.

By contrast, in Lisp, when programs get errors, the debugger runs in Lisp and the data the programmer sees is in the same presentational format as it would be in the source program. This means that the programmer has the full range of linguistic tools available for debugging that would be available for programming. And since programmers do a lot of programming, it means that the programmer is automatically *very* familiar with the debugging tools.

For example, the user sees an output from (FACTORIAL 50) and wonders if it is correct, so the user writes:

```
(/ (FACTORIAL 50) (FACTORIAL 49))      ⇒    50
```

The ability to interactively write a program which computes a result some alternate way, that consults alternate data, or that performs some other kind of validity check can be very reassuring, and goes far beyond the power provided by debuggers for most other languages.

Debugging Language as a Scripting Language

Lisp is also its own scripting language. Just as with debugging, this means someone already familiar with programming ordinary Lisp programs is automatically equipped with the full power of Lisp to write scripts for testing, archiving, and other repetitive tasks.

Programmatic Introspection Capabilities

The availability of numerous tools for manipulating and inspecting the set of definitions that have been done in your environment allows a strong handle for portable programs to get a foothold in an uncertain environment. For example, a program that needs to assure that a PARSE-DATE function exists might probe the present state of the environment and then adjust it to its needs, as in this example:

```
(MULTIPLE-VALUE-BIND (SYMBOL STATUS)
    (FIND-SYMBOL "PARSE-DATE" "MY-PKG")
  (COND ((NOT STATUS) (EXPORT (IMPORT 'TIME:PARSE-DATE "MY-PKG") "MY-PKG"))
        ((NOT (FBOUNDP SYMBOL))
         (SETF (SYMBOL-FUNCTION SYMBOL) #'TIME:PARSE-DATE))))
```

This ability to look around in the environment under program control, viewing programs as data to be examined and modified, can be very important not just to programs but to people. It can make it easy to test assumptions, check the completeness of changes, *etc.*

Loading New Code Into a Running Image

Lisp allows you to enter a running image, build up a lot of state by running programs, and then at that point load new definitions. This gives product maintainers the powerful ability to have customers load patches that can fix bugs in running programs without requiring those programs to be exited. The value to a customer in some situations can be immeasurable.

But the value to rapid prototyping is also not to be understated—if you have created a situation in which your running image has amassed a large amount of state and suddenly you want to change something, the last thing you want to do is stop to compile a new image and restart that. You want to do what Lisp lets you do: edit the single definition you want to change, press a key or two to get just that definition recompiled and reloaded, and then continue using the exact same image you've been running in—but with the bug fixed or the new feature added.

Mixing compiled and interpreted code

Because of Lisp's dynamic nature and heavy emphasis on the Read-Eval-Print loop as a development tool, there is a continuum between programs written at compilation time and those later typed interactively in the environment where a compiled program has been loaded. Compiled code can call interpreted code and interpreted code can call compiled code *completely transparently.*

For example, in this case we are calling the compiled function * (which performs multiplication) on arguments 3 and 4, supplied by the interpreter:

```
(* 3 4)  ⇒    12
```

In this case, an interpreted MAPCAR expression calls the compiled MAPCAR function which calls back to interpreted code (specified by the lambda expression, a notation for an anonymous function):

```
(MAPCAR #'(LAMBDA (X) (+ (* X X) (* X 2) 1)) '(1 2 3)) ; computes (x+1)^2
   ⇒    (4 9 16)
```

Dynamic Redefinition

Just about everything in Common Lisp can be dynamically redefined.

° Having loaded a compiled definition of a function, an interpreted definition can be typed into the interpreter and will take its place for debugging.

- Having loaded a compiled definition, a different compiled definition can be loaded to replace it.
- Having loaded a number of compiled methods for a generic function, additional methods can be typed to the interpreter and will be correctly combined—or previously combined methods can be redefined and the generic function will be adjusted accordingly.
- Having defined a package full of symbols, the package can be killed and a fresh definition can be loaded.
- Having created an object with a certain number of slots, a definition of that object's class can be loaded which adds or removes slots and all existing instances of that class will be automatically updated to reflect the new definition, without perturbing the data in slots whose definitions were not changed.

The list goes on and on.

The ability to dynamically redefine code in a running image is extraordinarily powerful, especially in the context of rapid prototyping. The core methodology of rapid prototyping is the iterative process of trying an approach, evaluating the results, and then trying a new approach. It involves lots of starting over. The fact that you don't have to restart your Lisp image just to start over on a few definitions saves lots of time and effort that would be spent relinking, reloading, restarting, and restoring your previous state had the program been written in another language.

Editor Integration with Environment

Most commercial quality Lisps provide a resident editor, written in Lisp. This makes for very high bandwidth of communication between the editor and the surrounding programming environment. A consequence of this is that there tend to be a large number of editor commands which can manipulate the environment directly (for example, evaluate or compile the current definition), ask "smarter" questions, offer more specific help information, or do better defaulting.

Conclusion

In this article, we began by looking at why rapid prototyping is important. Then we surveyed why Lisp in general, and Common Lisp specifically, is rich in data, tools, and interaction paradigms that make it an ideal vehicle for rapid prototyping.

We have shown that this claim is not a superficial attempt to associate Lisp with yet another marketing buzzword, but rather a fundamental aspect of the language that has in very specific ways earned the respect of its users.

We have seen that there is no single specific feature of Lisp which by itself enables rapid prototyping. Rather, there are myriad small design decisions in the language and the environment which contribute incrementally to its collective utility.

Acknowledgments

I'd like to thank Nigel Dolby for attending LUV'94 and asking me the questions that led to the writing of this paper. Thanks also to Christopher Fry and Rebecca Spainhower, who reviewed prototypes of this paper and offered much useful hindsight.

References

Kent Pitman and Kathy Chapman (editors), *draft proposed American National Standard for Information Systems—Programming Language—Common Lisp*, X3J13 document 94-101R, 1994. Available by anonymous FTP from "/pub/cl/dpANS3R" on PARCFTP.Xerox.COM.

Charles Rich and Richard C. Waters, "The Disciplined Use of Simplifying Assumptions," Working Paper 220, MIT Artificial Intelligence Laboratory, Cambridge, MA. December, 1981.