The Legacy of Lisp
“Observations/Rants”

Dedicated to Prof. E. Goto

Henry Baker, Ph.D.
Partner
Baker Capital Corp.
hbaker@bakercapital.com

Computing Bio

- 1401 (4K), 1620 (20K) experience
- 7040 (16KW) IBSYS experience
- 360/50 (256K) DOS experience
- 360/30 (64K?) PL/I experience
- 7090 (32KW) Timesharing
- PDP-10 (256KW) ITS Lisp experience
- PDP-8 (4KW) experience
- Lisp Machine (2-8KW) experience
- Spent ½ of career fighting memory issues

Computing Bio II

- Assemblers w/macros
- Fortran I (IF, subscripts)
- PL/I
- TECO
- Lisp
- APL
- Pascal/Ada
- C/C++
- Spent ¼ of career in “batch” processing

Computing Bio III

- Radiation treatment planning SW (7040/360)
- Bus DP – DB & RT Production Planning SW (now called "MRP"); Disk-based hash tables
- MIT – discrete simulation SW in Fortran
- MIT – Forrester simulations
- MIT – Asynchronous HW (Petri Nets & Marked Graphs)
- MIT – natural language in Lisp
- MIT – parallel processing ("futures")
- MIT – shallow binding
- MIT – RTGC
- Symbolics – Sales & Graphics
- Nimble – non-moving GC & "Cheney on the MTA"
- Spent significant time with applications & numeric applications

45 Years of Moore’s Law

- 1960: ~128Kbytes & ~100Kops/sec
- 2005: ~10Gbytes & ~4Gops/sec
- ~15 doublings in mem & proc in 45yrs (doesn’t count $$)
- Most arguments against Lisp have become obsolete

- So how come Lisp isn’t ubiquitous?

No Moore's Law for Software

- After 50 years of trying, US R&D establishment has thrown in the towel on SW – no silver bullets
- SW labor-intensive, so move SW offshore to Asia – lots of bright, cheap developers
- Why provide expensive tools for grunt labor?
- Why isn't this conference held in India or China?

Incremental SW Development

- “Debugging blank sheet of paper”
- Very low hurdle to execution => coding w/o thinking
- Old days of batch processing w/ one compile/day led to deep thinking
- Computer is tool to aid thought, but doesn’t replace thought
- Computer language is inherently a pun – needs to be interpreted by both men & machines

SW Development Process

- Personal style of programming
- Prototype idea – forget performance
- Define some test cases
- Refine structure & interfaces
- Rearrange code (substantial/global revisions)
- Refine for maintainability, performance
- Insert type checking
- Serious performance tuning
- Prove program correct
- Gets larger, more annotated/documented

SW Development II

- Want to move smoothly through preceding sequence
- Don’t want to retype or reprogram
- Need substantial global changes: (name changes, arglist changes, etc.)
- Incremental/local changes not enough
- Must integrate comments, annotations, test cases
- Must integrate type checking & proving

SW Development III

- SW development tools needed (how theory should guide practise)
- Input directly into symbols & conses – should never be necessary to find unbalanced parens
- Alpha renaming essential
- Beta expansion essential
- Beta abstraction essential
- Eta conversion essential
- Argument rearrangement essential
- Nested -> continuation-passing mode
- Datatype substitution
- “Homomorphic Image” (slice?) views

SW Development IV

- Holy Grail of Maintenance: Database evolution w/o tears
- Need to find & replace all references to data structures in programs AND
- Need to automatically generate programs to update the existing databases
- Lisp should be able to do this, but hasn’t

Programming in the Large v. Programming in the Small

- Fractal/scalable system would utilize same tools & mechanisms for small & large programs
- λ-calculus infinitely composable, BUT
- # free variables builds up non-scalably
- Largest # of free variables are function names (10's of thousands of names)
- Quite difficult to develop/edit/debug heavily lexically-nested programs
- C gave this up for multiple reasons
- Lisp has never addressed this issue

Lisp Features

- Trivial syntax
- Recursion only (originally no iteration)
- Recursive/fractal data structures
- Reflection (EVAL/APPLY)
- Macros
- Garbage Collection
- Hashed Atoms & property lists
- Read-Eval-Print loop
- Tagged Architecture

Chars Considered Harmful: “C Envy”

- Corollary 1: if you write a parser for some application, you probably have too much spare time on your hands
- Corollary 2: “Finite State Machines considered harmful” – beware any enterprise that requires new syntax, or the creation of a finite state machine
- Lisp was invented as a *symbol*-processing language, *not* a byte-processing language
- Proper rep for Lisp source code is S-expressions (or some other symbolic representation), NOT character files
- Adopting C approach of character source files was major step BACKWARD
- BBNLisp was better approach
- Proper way to edit Lisp is with structure, NOT character, editor (“Emacs considered harmful”)
- Comments & other annotations should be essential part of source code

Lisp Variables

- Original Lisp used dynamic/fluid binding rather than static/lexical binding
- Occurrence problem in dynamic binding is undecidable – you can’t find all name occurrences to rename
- Major screwup, which was slavishly copied by APL & many other interpreted languages
- Deep & unbounded variable searches led to “shallow binding” mechanism

Legacy of Shallow Binding

- Used in “undo”
- Two-phase transaction protocols (speculate/commit/rollback)
- Unwind-protect/try generalization of shallow binding
- Crash recovery protocols in databases
- Speculative execution in processors
- Reminiscent of “label-swapping” in networking protocols

Lisp Roots – Lambda Calculus

- λ-calculus uses application & abstraction
- λ-calculus has 3 rules:
- α-renaming (A rose by any other name…)
- β-reduction (fn application/argument binding)
- η-reduction (tail recursion)

What’s in a Name?

- Semantics of naming exposed when things are renamed
- Must know all & only occurrences of the name
- Must know what new names won’t conflict with existing names
- λ-calculus cares only about distinguishability of names, not spelling, per se
- (GC deals with names at different level; GC finds all & only occurrences; copying GC renames all & only occurrences)

Kinds of Names in Lisp

- Atom names (PNames)
- Keywords
- Macro names
- File names
- Record component names
- (Addresses for GC)

Why Renaming is Important

- Important to understanding someone else’s code
- Important to find all occurrences during development & debugging
- Important during program maintenance to upgrade programming documentation
- Important when importing program fragments for reuse

Argument Handling

- Long arg lists considered harmful
- Keyword/&rest is better, but still relatively unstructured – difficult to know who & when info is being used
- Need better idea of argument “bundles”
- Generally how to pass info though many levels of calls
- Sometimes, dynamic/fluid variables are more efficient!

Memory Management

- Dynamic – no fixed sizes for tables/arrays
- Don’t run out of space until *all* space is exhausted
- Break up memory into discrete chunks
- Dynamically allocate chunks
- Dynamically reclaim chunks not in use
- Emulate long arrays with multiple levels of short arrays – no significant slowdown (already done in HW, e.g.)
- Lisp didn’t invent dynamic memory allocation (IPL-V), but did invent tracing GC

GC is Cache-Friendly

- Write-allocate cache: allocate when written (don’t read from memory)
- Works well with sequential allocating copying collector
- Most cells live & die in cache & are never written to memory!

Real-time Time Management

- Analogous to Memory Management
- Time broken into discrete chunks
- Unbounded stretches of uninterrupted execution don’t happen
- Scheduling thru allocation/deallocation of these chunks
- Repetitive/cyclic tasks (filling/emptying buffers)

Efficiency Matters

- Efficiency hacking is major % of all programming effort
- If large builtin library is inefficient, then why bother with it?
- Need *smooth* transition from generic/slow library routines to efficient specialized routines (e.g., graphics 1/sqrt(x))
- Lisp never able to shake its bad reputation for inefficiency
- Too easy to write slow programs (ditto for PL/I)
- Size of program not correlated with efficiency
- Too many bad books
- Too many bad implementations – e.g., slow readers/interning/printers

Efficiency Matters II

- Not easy to write efficient code
- No static typing, horrible declaration language, non-existent tools
- Few profilers
- Difficult/impossible to replace buggy/slow builtin/library routines
- Need reflective system to replace stuff “under the hood”
- “Inline” declaration that is guaranteed

Type Checking in Lisp

- Why not? “Real men don’t type check”
- Lack has led to “hacker” view of Lisp programmers – always prototyping, never delivering production code
- Type checking doesn’t solve every problem, but is helpful in large systems
- Just the exercise of *trying* to “type” Lisp highlights some bad design features
- One of the many balls dropped by Lisp

Necessary Changes for Lisp

- A static language is a dead language (e.g., Latin)
- Common Lisp halted most innovation in Lisp

- Rationalize the type system – too many functions have bad typing that force inefficient implementations
- More efficient bit-hacking
- Immutable list cells & strings (Just Do It!)
- Linear variables (more controversial)
- Micro-kernel with reflective portions (decompose monolithic Lisp systems into simple pieces)
- Real-time scheduler
- Persistent DB for code, comments, test cases, etc.
- Better integration with threads & NUMA parallel processors
- Much better system construction tools ("ifdef considered harmful")

Bit Hacking -- Compression

- Compression is ubiquitous
- Gzip, jpeg, mpeg, etc.
- Disk, network, memory management
- Factors > 2 matter!
- Huge improvements in computer architecture from multiple levels of compression/encoding/decoding

Productivity Example – JPEG Decode

- Interpret bit strings
- Integer DCT
- Color space conversion

- No particular advantage for Lisp; very large potential disadvantage for Lisp

Immutable Cons Cells & Strings

- Long overdue – don’t need heavyweight CONS cells & strings
- Define your own structs if you want to
- EQ -> EQUAL
- Hash CONS if you like
- Substantial compiler efficiencies
  (e.g., treatment of &rest args, pnames)
- Substantial runtime efficiencies (e.g., cache coherence)
- Copying collectors don’t need forwarding
- Thread-safe
- Non-shared memory parallel processors
- Conversion from “linear” to “immutable” during CONS (“publishing”)

“Resources” are Linear

- “Hidden” arguments & returned values to/from subroutines:
  stack space, freelist, processor time
- Real-time systems must tightly manage resources (I/O devices, space, time)
- Need to make hidden arguments visible
- Analogy: Scheme provided access to return address & previous stack through “continuation”
- “Linear” Lisp would provide explicit access to freelist, scheduling queues, etc.

Linear Variables/Data Structs

- Linear variables are referenced exactly once within scope (once per if arm)
- Non-shared, so thread-safe
- Cache-friendly (access => dead)
- Reflection: Freelist is linear
- Shared variable = linear variable + semaphore

Lisp Systems Too Monolithic

- Traditional Lisp systems were monolithic – large amounts of non-Lisp code
- At the mercy of the implementor re quality & efficiency – can’t easily replace/upgrade inefficient parts
- Have to “re-invent the wheel” to get decent performance
- Efficiency matters!

Need Reflective Lisp Systems

- Need to be able to replace/upgrade significant portions of a system – Lisp reader, GC
- Much more productive to preserve application code & make “builtins” more efficient
- Efficiency matters!

Real-Time Lisp

- Lisp one of the 1st to automatically manage storage
- Break storage into small “packets”; Large objects are composed from such packets
- Packets dynamically allocated & freed

Real-Time Lisp II

- Why not automatically manage time?
- Break time up into small “packets”
- Allocate (& deallocate) packets with scheduler
- Never have to do “pre-emption” – no interrupts, no masking, etc.
- No god-given right to continuous execution

Real-Time Lisp III

- Early byte-addressed computers used variable-length data (1401/1410/1620) delimited by “word marks”
- Instructions ran arbitrarily long, depending upon size of the data
- Impossible to interrupt; large amount of state to save/restore
- Modern computers try to limit duration & state of single instruction by using fixed-size data packets (“words”)

Real-Time Lisp IV

- Modern computers have fixed-size cache lines
- Modern compilers & processors worry about jump-free instruction sequences
- Typical jump-free sequences are about the order of magnitude of a cache line
- No real overhead cost from limiting size of non-interruptible sequence

Real-Time Lisp V

- Need to allocate time in the future
- One-time allocations; cyclic allocations
- Allocate buffers & times for I/O transfers
- Often need to deallocate future slots (e.g., variable-length I/O transfer is complete)
- Need to allocate time for background tasks
  - e.g., GC

Seamlessly Integrated Persistent Database

- Original BBN Lisp model almost OK
- Dumped/restored Lisp symbols & properties
- Didn’t meet “ACID” test
- Lisp Machine was its own DB (kept running for years), but wasn’t sharable & didn’t meet ACID test; also, it stored code in char files
- Allegro OODB excellent, but not seamlessly integrated
- Lisp could have won big on this one feature alone

What Lisp Did Right

- Personal, interactive environment
- Fast prototyping
- Symbols & lists v. characters/bits/numbers
- Simple syntax
- Small core of intrinsics/”special forms”
- Spectacular success: Boyer-Moore theorem prover

What Lisp Did Wrong

- Garbage collection doesn’t solve leaks (accumulations of stuff unexpected by higher levels)
- Didn’t become the Operating System
- Didn’t handle real-time events, interrupts
- Didn’t incorporate persistent storage
- Didn’t provide some level of typing
- Didn’t provide good enough tools/editors
- Didn’t address large-scale programming

Missed Opportunities

- No persistent Lisp database for source code & applications
- Completely missed the PC revolution
- Dropped the ball on CAD – e.g., AutoCAD
- Dropped the ball on Macsyma – Mathematica & Matlab
- Dropped the ball on text editors – Emacs v. MS Word
- Dropping the ball on LispStat
- Dropping the ball on video games
- Dropping the ball on XML

Lisp Features Co-opted

- Interactive/immediate execution – APL, Smalltalk, Javascript, etc.
- Recursion – Pascal, C/C++, Java, even Fortran!
- Recursive data structures – Pascal/Ada, C/C++, Smalltalk, Java, etc.
- Garbage collection – Smalltalk, Java, etc.
- Lisp is too happy to play Greek slave to the Roman master

Major Problems for Lisp today

- Beowulf-style Linux clusters
- Lisp’s preference for global address space makes this infeasible
- Cache as cache can – Lisp doesn’t map well to modern memory hierarchies
- Only standard method of persistence is byte-based file systems

The “XML Question”

- By rights, Lisp should *own* XML
- Lisp should immediately embrace XML
- Lisp needs to quickly develop standard XML readers & printers
- Lisp needs to utilize XML as alternate syntax

Applications Matter

- People don't buy languages, they buy applications
- Matlab – language for accessing linear algebra library
- Emacs – language for accessing text-processing library
- LispStat – language for accessing statistics library
- AutoCAD – language for accessing 2D CAD drawing library
- => Differentiation is in the libraries

What Symbolics did right

- Raised enough $$ to start a real company
- Hired good production HW people
- Built good sales & service organization
- Implemented standards (Common Lisp, Fortran/Pascal, Ethernet)
- Developed excellent documentation
- Had excellent training courses

What Symbolics did wrong

- Technical issues (SW done before lex vars; “stack groups” horrible thread mechanism; stack architecture incapable of optimization; paging done in ucode)
- Missed the whole PC revolution
- Missed the Unix wave (“I will not work for a company that incorporates Unix into a product”)
- No “application delivery” box
- Lisp chip was too little, too late
- Could not respond with SW on commodity HW
- Could not respond with simpler software for non-wizards
- “Too Many Notes” – not enough focus

HW v. SW Design

- Mystery: why does Cadence get $$$ per seat for HW design, while SW tools are given away?

Why Aren’t SW Tools Expensive?

- HW tools cost $100K/seat/year
- How come people won’t pay $100K/seat/year for SW developers?
- SW development takes a long time & is very expensive
- SW bugs are extremely expensive to fix in the field
- SW lasts *longer* than HW, so it should be *more* important to do a good job in SW

Why Aren’t SW Tools Expensive II

- SW development has moved to Asia
- Lots of bright, cheap programmers
- SW productivity isn’t very good

Microsoft as a SW Black Hole

- Windows incorporates all else
- Embrace & Extend
- Bad money drives good out of circulation
- No incentive for non-MS innovation – all rewards accrue to MS
- Pace of SW innovation at the mercy of MS
- Zero SW progress in last 10 years
- Therefore, MS hiring of all those PhD’s is actually the *cause* of the lack of innovation

Wakeup Call for Lisp

- Lisp Conference 2006: Masada or China
- Masada: Die/suicide for religious purity

  Or

- China: Embrace dramatic change

Contact Info

- hbaker1@pipeline.com
- http:/home.pipeline.com/~hbaker1