DOI:10.1145/1498765.1498781

The ecosystem of purpose-built languages is a key part of systems development.

BY MIKE SHAPIRO

Purpose-Built Languages

In my college computer science lab, two eternal debates flourished during breaks from long nights of coding and debugging: “emacs versus vi?” and “what is the best programming language?” Later, as I began my career in industry, I noticed that the debate over programming languages was also going on in the hallways of Silicon Valley campuses. It was the 1990s, and at Sun Microsystems many of us were watching Java claim significant mindshare among developers, particularly those previously developing in C or C++.

I have always found the notion of *best* language to be too subjective and too dependent on the nature of the programming task at hand. Over my career, however, I have spent significant time pondering two related questions that I think are more fundamental. First, is software engineering at large being done on *fewer* languages over time? That is, is the set of computer languages *converging*? Second, what makes a particular language “better” or useful or more rapidly adopted for a particular task?

In examining these questions I have found it particularly interesting to look not at the battle of the heavyweights, but rather at their less well-studied offshoots, the *purpose-built* languages. These languages

sprout like weeds alongside the road of mainstream language development, and they exhibit properties and a history that lead one to reconsider instinctive answers to the fundamental language questions. Considering purpose-built languages, programming language development is not converging at all, and utility seems to have little to do with traditional notions of structure or properties that are empirically “better” from a language-design perspective. Purpose-built languages even defy a strict definition worthy of a prescriptive compiler grammarian: they somehow seem “smaller” than a full-fledged programming language; they are not always Turing-complete; they can lack formal grammars (and parsers); they are sometimes stand-alone but often a part of a more complex environment or containing program; they are often but not always interpreted; they are typically designed for a single purpose but often (accidentally) jump from one type of use to another. And some are even nameless.

Most significantly, purpose-built languages have often formed an essential part of the development of larger software systems such as operating systems, whether as a part of developer tools or as glue between distinct pieces of a larger environment. So it is particularly interesting to unearth some of these lesser-known creations and look at their connections to our larger language insights. In my career, while working on several commercial operating systems and large software components, I have come to conclude that not only are new languages developing all the time, but they are also often integral to the growth and maintenance of larger-scale software systems.

The Unix environment, with its philosophy of little tools that can be easily connected, was an ideal greenhouse for the growth of purpose-built languages. A cursory scan of Unix manuals from the early 1980s shows more than 20 little languages of vari-

ILLUSTRATION BY JOHN HERSEY

Figure 1: Little languages in Unix, early 1980s.

Debuggers adb dbx stabs	Shells/Utilities sh csh awk bc, dc ed, ex, vi sed	Text Formatting eqn pic nroff, troff
Programming Tools lex make m4 ratfor yacc	Games rogue, trek	Libraries regex Configuration sendmail.cf

ous forms in active use, as shown in Figure 1.

These languages vary from complete programming languages (sh) to preprocessors (yacc) to command-line syntax (adb) to representations of state machines or data structures (regular expressions, debugger “stabs”). Twenty years later, when Sun Microsystems released the modern Unix system Solaris 10, almost all of the new significant operating-system features involved the introduction of new purpose-built languages: the DTrace debugging software introduced the D language for tracing queries; the Fault Management system included a language for describing fault propagations; the Zones and Service Management features included XML configuration grammars and new command-line interpreters.

The history of one of these little Unix languages, that of the adb debugger, is particularly illustrative of the accidental evolution and stickiness of something small but useful in a larger system.

Evolution Trumps Intelligent Design

The early development of Unix occurred on DEC PDP systems, which had a very simple debugger available known as ODT, or Octal Debugging Technique. (This terrific name conjures thoughts of a secret kung fu maneuver used to render the PDP’s 12-bit registers paralyzed.) The ODT program supported an incredibly

primitive syntax: an octal physical memory address was specified at the start of each command and suffixed with a single character (say, B for breakpoint) or a slash (/) to read and optionally to write the content of that memory location, as shown in Figure 2.

Thus, a little language was born. The ODT syntax clearly inspired the form of the first debugger for the new Unix system being developed on the PDP, which was simply called db. At the time of Unix v3 in 1971, the db command syntax borrowed the basic ODT model and began extending it with additional character suffixes to define addressing modes and formatting options, as shown in Figure 3.

By 1980, db had been replaced by adb, which was included with the AT&T SVR3 Unix distribution. The syntax had evolved to add new debugging commands over the intervening years and now supported not just simple addresses but arithmetic expressions (123+456 / was now legal). Also, a character after “/” now indicated a data format, and a character after “\$” or “:” now indicated an action. The adb syntax is shown in Figure 4.

The addition of “\$<” to read an external file of commands was particularly interesting, because it spawned the development of primitive adb programs or macros that executed a series of commands to display the contents of a C data structure at a particular memory address. That is, to display a kernel proc structure, you

Figure 2: ODT-8 debugger syntax, circa 1967.

400 /	display the content of memory location 400
400 / 6046	observe the value is 6046 at that location
400 / 6046 2345	rewrite the value to 2345
3000B	set a breakpoint at 3000
<i>ODT-8 Octal Debugging Technique Manual, DEC-D8-COCO-D, Dec. 1967</i>	

Figure 3: Unix V3 “db” syntax, circa 1971

address /	Print a word in octal
address \	Print a byte in octal
address `	Print a byte in ASCII
address ?	Disassemble an instruction
\$	Print the machine registers
<i>Unix Version 3 Manual Pages, DB(1), Nov. 3, 1971.</i>	

Figure 4: AT&T SVR3 “adb” syntax, circa 1980.

expression /o	Print a two-byte integer in octal at the specified address
expression /x	Print a two-byte integer in hexadecimal at the specified address
expression /d	Print a two-byte integer in decimal at the specified address
\$a	Print an Algol stack backtrace
\$c	Print a C stack backtrace
\$r	Print the machine registers
expression :b	Set a breakpoint
\$< filename	Read and execute the command found in the specified file
<i>Kernighan, B. Ratfor: A preprocessor for a Rational Fortran. Software—Practice and Experience. Oct. 1975.</i>	

would take its address and then type “\$<proc” to execute a predefined series of commands to display each memory of the C data structure for a process. The content of the proc macro in SunOS 4 from 1984 is shown in Figure 5. To make this output understandable, the “/” command could now be suffixed with quoted string labels, newlines (n) and tabs (16t) to be included among the decoded data.

The “.” variable evaluates to the input address used when applying the macro, and the “+” variable evaluates to that input address incremented by the byte count of all preceding format characters. The macros were then maintained with the kernel source code.

More than a decade later, in 1997, I was working at Sun on what would become Solaris 7. This release was our first 64-bit kernel, but the kernel-debugging tool of choice was still *adb* just as it was in 1984, and our source base now contained hundreds of useful macro files. Unfortunately, the implementation of *adb* was essentially impossible to port cleanly from 32-bit to 64-bit to debug the new kernel, so it seemed the time was ripe for the development of a new clean code base with many more modern debugger features.

As I considered how best to approach the problem, I was struck by the fact that despite its brittle, unstructured code base, the key feature of *adb* was that its syntax was imbued deeply in the minds and behaviors of all of our most experienced and effective engineers. (As someone aptly put it at the time: “It’s in the fingers.”) So I set out to build a new modular debugger (*mdb*) that would support an API for advanced kernel debugging and other modern features, yet would remain precisely backward-compatible with existing syntax and macros. Sophisticated new features were added after a new prefix (“:”) so they would not break the existing syntax (for example, “:findleaks” to check for kernel memory leaks). The entire syntax was at last properly encoded as a yacc parser. Macro files were phased out in favor of compiler-generated debug information, but the “\$<” syntax was left as an alias. Another decade later, *mdb* remains the standard tool for postmortem debugging of the OpenSolaris kernel and has been extended by hundreds of programmers.

The debugger tale illustrates that a little purpose-built language can evolve essentially at random, have no clear design, no consistent grammar or parser, and no name, and yet endure and grow in shipping operating systems for more than 40 years. In the same time period, many mainstream

Figure 5: Debugging the “proc” structure on SunOS 4, circa 1984.

```
address $<proc
./"link"16t"rlink"16t"nxt"16t"prev"nXXXX
+/"as"16t"segu"16t"stack"16t"uarea"nXXXX
+/"upri"
+/"upri"8t"pri"8t"cpu"8t"stat"8t"time"8t"nice"nbbbbbb
+/"slp"8t"cursig"16t"sig"bbX
+/"mask"16t"ignore"16t"catch"nXXX
+/"flag"16t"uid"8t"suid"8t"pgrp"nXddd
+/"pid"8t"ppid"8t"xstat"8t"ticks"nddx
+/"cred"16t"ru"16t"tsize"nXXX
+/"dsize"16t"ssize"16t"rsize"nXXX
+/"maxrss"16t"swrss"16t"wchan"nXXX
+/"16+"%cpu"16t"pptr"16t"tptr"nXXX
+/"real itimer"n4D
+/"idhash"16t"swlocks"ndd
+/"aio forw"16t"aio back"8t"aio count"8t"threadcnt"nXXXX
```

Figure 6: Fortran and Ratfor, circa 1975.

<pre> if (a .gt. b) goto 10 sw = 0 write(6, 1) a, b goto 20 10 sw = 1 write(6, 1) b, a 20 ... </pre>	<pre> if (a <= b) { sw = 0; write(6, 1) a, b } else { sw = 1; write(6, 1) b, a } </pre>
--	--

Figure 7: C and Algol mutant from early “adb”.

```
#define IF if(
#define THEN ){
#define ELSE } else {
#define FI }
#define ANDF &&
...
localsym(cframe)
L _INT cframe;
{
    INT symflg;
    WHILE nextsym() ANDF localok
        ANDF symbol.symc[0]!='~'
        ANDF (symflg=symbol.symf)!=037
    DO IF symflg>=2 ANDF symflg<=4
        THEN localval=symbol.symv;
            return(TRUE);
        ELIF symflg==1
        THEN localval=leng(shorten(cframe)+symbol.symv);
            return(TRUE);
        ELIF symflg==20 ANDF lastframe
        THEN localval=leng(lastframe+2*symbol.symv-10);
            return(TRUE);
        FI
    OD
    return(FALSE);
}
```

languages came and went into the great beyond (Algol, Ada, Pascal, Cobol, and so on). Fundamentally, this debugger has survived for one reason: it concisely encoded the exact task its

users performed and thereby connected to those users. Take an address, dump out its content, find the next address, follow it to the next location of interest, dump out its content, and

so on. For purpose-built languages, a deep connection to a task and the user community for that task is often worth more than clever design or elegant syntax.

Mutation and Hybridization

Mutation, some accidental and some intentional, often plays a critical role in the development of purpose-built systems languages. One common form of mutation involves adding a subset of the syntax of one language (for example, expressions or regular expressions) to another language. This type of mutation can be implemented using a preprocessor that converts one high-level form to another or intermingles preprocessed syntax with the target syntax of a destination language. Mutations may diverge far enough that a new hybrid language is formed. The parser tools yacc and bison are the most well-known examples of a complete hybrid language: a grammar is declared as a set of parsing rules intermingled with C code that is executed in response to the rules; the utilities then emit a finished C program that includes the rule code and the code to execute a parsing state-machine on the grammar.

Another example of this type of mutation in early Unix was the Ratfor (Rational Fortran) preprocessor developed by Kernighan. Ratfor permitted the author to write Fortran code with C expressions and logical blocks, and the result was translated into Fortran syntax with line numbers and goto statements, as shown in Figure 6.

An even stranger mutant language was a hybrid of C and Algol syntax developed using the C preprocessor and used in the code for, what else, adb. Apparently, Steve Bourne, the author of the Algol-like Unix sh syntax, was determined that some of Algol's genome would carry on in the species. Some sample code is shown in Figure 7.

Alas, a later version of the code was run through the preprocessor and then checked in so as to ease maintenance. Many future languages have included more clearly designed cross-breeding to ease the transition from one environment to another. Following the widespread adoption of C, its expression syntax found its way into an incredible number of new lan-

Take an address, dump out its content, find the next address, follow it to the next location of interest, dump out its content, and so on. For purpose-built languages, a deep connection to a task is often worth more than clever design or elegant syntax.

guages, little and big, including Awk, C++, Java, JavaScript, D, Ruby, and many others. Similarly, following the success of Perl, many other scripting languages adopted its useful extensions to regular expression syntax as a new canonical form. Core concepts such as expression syntax often form the bulk of a small language, and borrowing from a well-established model permits rapid language implementation and rapid adoption by users.

Symbiosis

In the development of a larger software system, little languages often live in symbiotic partnership with the mainstream development language or with the software system itself. The adb macro language described earlier would likely not have survived outside of the source-code base of its Unix parent. The macro language of your favorite spreadsheet is another example: it exists to provide a convenient way to manipulate the user-visible abstractions of the containing software application.

In the operating-system world, my favorite little-known example of symbiosis is the union of Forth and SPARC assembly language created at Sun as part of the work on the Open-Boot firmware. The idea was to create a small interpreter used as the boot environment on SPARC workstations. Forth was chosen for the boot and hardware bring-up environment for new hardware because the language kernel was tiny and could be brought up immediately on a new processor and platform. Then, using the Forth dictionaries, new commands could be defined on the fly in the interpreter for debugging. Since Forth permits its dictionaries to override the definition of words (tokens) in the interpreter, someone developed the creative idea of using the interpreter as a macro assembler for the hardware. A set of dictionaries was created that redefined each of the opcodes in SPARC ("ld," "move," "add," and so on) with Forth code that would compute the binary representation of the assembled instructions and store them into memory. Therefore, entire low-level functions could be written in what appeared to be assembly language, prefixed with Forth headers, and

typed into the tiny interpreter, which would then assemble the object code in memory as it parsed the tokens and executed the resulting routine.

In recent years, Web browsers have become fertile ground for mutation and symbiosis. Two central figures in modern Web development are interpreted JavaScript and XML. (XML itself is the syntax for a variety of other languages and an abundant source of hybrid languages and mutations.) In the common Ajax programming model, JavaScript objects can be serialized to XML form, and XML encodings can be used to pass remote procedure calls back to a server. In one such encoding, XML-RPC, a standard extension called multicall is provided for the browser client to issue multiple procedure calls from the client to the server in a single transfer. An example of a single call to a method `x.foo` and then a series of calls to the same method using multicall is shown in Figure 8.

While implementing Ajax user-interface code for a new line of storage products, the Sun Fishworks team wanted to develop a way to minimize unnecessary client-server interactions. The first concept developed was the notion of a multicall invocation whose parameter was the result of another call. In the following example, the method `x.foo` is called on the result of `x.bar` in a single XML-RPC interaction as shown in Figure 9.

The trick here is that the new structure member `methodParams` indicates that the next members are not static parameters but more methods to be called recursively, with the result pushed onto a stack. Once a stack had been born, it was only natural to start throwing in operators from a stack-based language, forming an entirely new interpreted language that itself is declared as data in JavaScript, sent to the server by the existing XML-RPC serialization, and executed by extensions to our XML-RPC interpreter engine. A few of the operators that we implemented at Sun are shown in Figure 10.

This example illustrates that the symbiotic relationship between JavaScript and XML essentially allows our language to exist without requiring its own lexer or parser, and

Figure 8: JavaScript, XML-RPC, and Multicall.

```
x.foo({ bar: 123, baz: 456 });

system.multicall(
  { methodName: 'x.foo',
    params: [ { bar: 123, baz: 456 } ] },
  { methodName: 'x.foo',
    params: [ { bar: 789, baz: 654 } ] },
  { methodName: 'x.foo',
    params: [ { bar: 222, baz: 333 } ] }
)
```

Figure 9: Multicall with parameterized results.

```
system.multicall(
  { methodName: 'x.foo', methodParams: [
    { methodName: 'x.bar', params: [ 1, 2, 3 ] }
  ] },
  ...
)
```

Figure 10: Multicall with stack operators.

```
system.multicall(
  { foreach: [ [ 2, 4, 6 ], [
    { methodName: 'x.foo', params: [] },
    { push: [] },
    { div: [ { pop: [] }, 2 ] }
  ] ] },
  ...
)
```

fundamentally serves the purpose of offloading performance-critical code from JavaScript to our server and minimizing round-trips. In the videogame industry, a similar symbiosis (without the hybrid syntax) has developed between Lua and C/C++. The Lua scripting language provides a popular form for writing non-performance-critical code in videogame engines, and the Lua interpreter design makes it easy to bridge to C code.

Once two or more languages are interacting in a large software system, it becomes only natural for an ecosystem of tools (likely incorporating little languages with hybrid syntax) to spring up around them to ease the maintenance, development, and debugging of the entire system. If a rich ecosystem grows around the languages of a complete software system, both little and big, purpose-built

and general-purpose, the longer the overall environment will thrive and its constituents survive. Therefore, as we build our towers of software abstraction ever higher, we should expect to see and know more languages, not fewer. C

Mike Shapiro (mws@sun.com) is a Distinguished Engineer at Sun Microsystems and is currently leading Sun's Fishworks advanced engineering team in San Francisco. He previously worked in Sun kernel engineering where he developed a variety of technologies for Solaris including `pgrep`, `pkill`, `mdb`, `dumpadm`, `libproc`, `CTF`, `fmd`, the `DTrace` D language and compiler, `smbios`, and a variety of features related to CPU, memory, I/O, and software fault handling and diagnosis.

A previous version of this article appeared in *ACM Queue*, January 2009.